Promenade au-delà de NP

Jean-Marie Lagniez

Promenade au-delà de NP

Habilitation à Diriger des Recherches

(Spécialité Informatique)

Université d'Artois

présentée et soutenue publiquement le 11 décembre 2019

par

Jean-Marie Lagniez

Composition du jury

Presidents:	Christine Solnon	Professeur à l'INSA, LIRIS, Lyon, France
Rapporteurs:	Hélène Fargier	Directeur de recherche CNRS, IRIT, Toulouse, France
	Andreas Herzig	Directeur de recherche CNRS, IRIT, Toulouse, France
	Thomas Schiex	Directeur de recherche à l'INRA, Toulouse, France
Examinateurs:	Philippe Dague	Professeur à l'Université Paris-Sud, LRI, Paris, France
	Pierre Marquis	Professeur à l'Université d'Artois, CRIL, France

Centre de Recherche en Informatique de Lens – CNRS UMR 8188
Artois University, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secretariat : Tél.: +33 (0)3 21 79 17 23
http://www.cril.univ-artois.fr

Mise en page par thesul v0.14 (D. Roegel, LORIA) et memcril v0.20 (B. Mazure, CRIL)
La conception des boîtes est faite par xeboiboites (A. Flesch)

À mes frères,

Sommaire

III Sélection de publications

Première partie
Synthèse des travaux de recherche

Chapitre 1

Introduction

Essayons pour commencer de situer l'environnement dans lequel les travaux présentés dans ce manuscrit ont été réalisés. Il y a deux événements marquants qui ont façonné ma manière de concevoir la recherche. Le premier est mon stage post-doctoral qui s'est déroulé dans l'équipe d'Armin Biere au sein du laboratoire FMV (*Institute for Formal Models and Verification*), rattaché à l'université Johannes Kepler, à Linz en Autriche. Lors de ce postdoc, j'ai appris la rigueur quant au développement de logiciels, et j'ai aussi pu explorer un nouveau domaine de recherche, la résolution incrémentale du problème SAT. Le second fait marquant est ma prise de fonction en tant que maître de conférences à l'université d'Artois au sein du centre de recherche en informatique de Lens (CRIL). Le CRIL est un laboratoire dont la thématique fédératrice concerne l'intelligence artificielle et ses applications (voir www.cril.fr pour plus d'informations). Puisque j'ai aussi réalisé ma thèse de doctorat au CRIL, c'est tout à fait naturellement que mes travaux de recherche se sont orientés vers les thématiques étudiées dans ce laboratoire.

Avant d'aller plus loin, il faudrait essayer de répondre à la question suivante : qu'est-ce que l'intelligence artificielle ? La réponse à cette question a pas mal changé depuis le moment où j'ai commencé ma thèse (en 2008). À l'époque, l'intelligence artificielle n'avait pas l'aura qu'elle a aujourd'hui, pour être plus précis il était parfois « mal vu » dans la communauté scientifique de travailler sur ce thème de recherche. Que s'est il donc passé entre le moment où j'ai commencé ma thèse et aujourd'hui ? La réponse est assez simple : l'apprentissage automatique a fait des progrès significatifs permettant aujourd'hui de traiter efficacement des problèmes qui ont trait au traitement du langage ou à la reconnaissance d'images. Pour faire simple, maintenant, l'intelligence artificielle a une visibilité à travers des applications impressionnantes. Afin de comprendre précisément ce qui s'est passé, décortiquons la citation suivante issue d'une interview de Yann Le Cun (chercheur français en intelligence artificielle et vision artificielle (robotique)) pour France Culture :

> "Des techniques qui avaient été inventées il y a assez longtemps, fin des années 1980, début des années 1990, ont été remises au goût du jour grâce au progrès des ordinateurs et à l'augmentation des tailles des bases de données sur lesquelles on peut intervenir, ça s'appelle « l'apprentissage profond ». L'apprentissage profond, c'est l'idée de construire une machine à apprentissage. La raison pour laquelle on l'appelle « profond », c'est que le calcul qui se déroule dans ces machines est constitué de plusieurs couches de calcul, effectuées les unes après les autres. Chaque couche implique quelques millions, dizaines de millions de calculs élémentaires. Et on peut entraîner cette machine de bout en bout, tous les modules à la fois. Mais dans le processus d'apprentissage, les couches intermédiaires internes de cette machine apprennent des concepts intermédiaires qui ne sont pas directement programmés dans la machine. Donc c'est clair que si on veut reconnaître une voiture, c'est probablement une bonne idée de détecter les roues, pour reconnaître un avion, peut-être détecter les ailes. Ce système apprend par lui-même à détecter les parties importantes de l'objet."

Ce qu'il faut retenir de cette citation est que les méthodes d'apprentissage fonctionnent particulièrement bien aujourd'hui car les machines pour les faire fonctionner sont disponibles et aussi parce que nous avons actuellement une quantité très importante de données à notre disposition. Grâce à cela, nous sommes actuellement capables de produire des programmes jouant à des jeux de stratégie en temps réel ou conduisant des voitures. Faut-il conclure que l'intelligence artificielle est en passe de supplanter l'intelligence humaine ? La réponse à cette question n'est pas forcément facile. Ce que nous pouvons assurer, c'est que les méthodes d'apprentissage ont besoin d'énormément de données pour être « fiables ». Pouvons-nous alors restreindre l'intelligence humaine à un processus capable de compiler les données que nous avons récupérées (notre éducation, notre vécu, ...) ? Pour répondre à cette question considérons la citation suivante, qui nous vient d'une interview d'Étienne Klein pour la revue l'`ADN` (physicien et philosophe des sciences français) au sujet de la théorie de la relativité générale d'Albert Einstein et du fait qu'elle ne s'appuie sur aucune donnée :

> "En 1915, lorsqu'Einstein publiait la théorie de la relativité générale, une nouvelle théorie de la gravitation, on n'avait que très peu de données sur l'univers : on ignorait, par exemple, qu'existassent d'autres galaxies que la nôtre, on ne savait pas d'où vient que les étoiles brillent, ni que l'univers est en expansion, etc. Mais les équations d'Einstein, d'une part, se sont parfaitement accommodées de la quantité gigantesque de données recueillies depuis un siècle par les télescopes et les satellites, d'autre part ont permis de prédire l'existence de nouvelles sortes d'objets physiques, tels les trous noirs ou les ondes gravitationnelles. Imaginons maintenant que les choses se soient passées dans l'ordre inverse, c'est-à-dire que nous ayons commencé avec toutes les données dont nous disposons aujourd'hui, mais sans avoir à notre disposition la théorie de la relativité générale. Pourrions-nous, par une sorte d'induction théorique permettant de passer des données aux lois, découvrir les équations d'Einstein ? Je pense que non."

Ce qui est important de noter ici c'est que l'intelligence n'est pas uniquement liée à la quantité de données et des corrélations que nous sommes capables d'en extraire. En fait, comme le souligne Étienne Klein dans la citation suivante, dans certaines situations, ces corrélations peuvent être mal interprétées.

> "Il est en revanche envisageable que nous nous perdions dans l'identification de multiples corrélations, pas forcément bien interprétées, par exemple entre vie passée, vie professionnelle, activités associatives, goûts musicaux, positions politiques, relations amicales ... Or, une corrélation n'est pas la même chose qu'une relation de cause à effet : ce n'est pas parce qu'il y a des grenouilles après la pluie que l'on a le droit de dire qu'il a plu des grenouilles. Mais il arrive très souvent que nous confondions les deux choses, à la manière d'un Coluche conseillant de ne jamais aller à l'hôpital au motif que l'on y meurt plus souvent que chez soi ... Une certaine vigilance épistémologique s'impose donc."

Ce que nous pouvons tirer de ces citations est que le raisonnement ne peut pas se limiter à un ensemble de réflexes programmés et issus d'un conglomérat de données. De mon point de vue, le raisonnement humain est quelque chose de plus subtil qui allie à la fois expérience passée et mécanisme d'inférence, se basant sur une exploration plus ou moins exhaustive des possibilités qui s'offrent à nous quant à la réalisation d'une tâche. Cette manière d'envisager la production de systèmes intelligents et fiables a été très bien exprimée par Moshe Vardi (professeur

d'informatique à l'Université Rice, aux États-Unis) lors de la *Federated logic conference* de 2018, où il met en avant l'importance de combiner la logique et l'apprentissage.

> "There is a recent perception that computer science is undergoing a Kuhnian paradigm shift, with CS as a model-driven science being replaced as a data-driven science. I will argue that, in general new scientific theories refine old scientific theories, rather than replace them. Thus, data-driven CS and model-driven CS complement each other, just as fast thinking and slow thinking complement each other in human thinking, as explicated by Daniel Kahneman. I will then use automated vehicles as an example, where in recent years, car makers and tech companies have been racing to be the first to market. In this rush there has been little discussion of how to obtain scalable standardization of safety assurance, without which this technology will never be commercially deployable. Such assurance requires formal methods, and combining machine learning with logic is the challenge of the day."

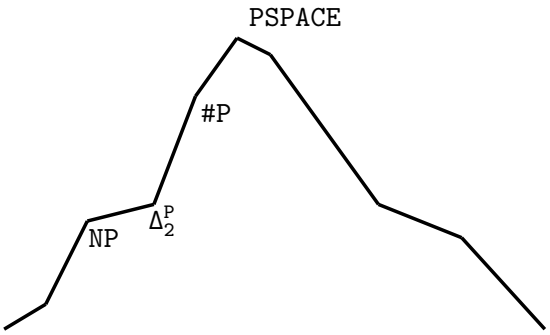
En fait, Moshe Vardi exprime un fait qui a déjà été validé expérimentalement dans le cadre du jeu de `go` avec le programme `AlphaGo` (Silver et al. 2016a). En effet, `AlphaGo` ne s'appuie pas uniquement sur l'apprentissage, mais construit un arbre de recherche qui est guidé par un réseau de neurones permettant d'évaluer le potentiel des coups. Les résultats impressionnants de `AlphaGo` montrent l'intérêt d'utiliser l'apprentissage, mais ils montrent aussi qu'il est nécessaire de s'appuyer sur un processus de raisonnement systématique pour réaliser des tâches complexes. Une autre particularité de l'approche `AlphaGo` est qu'elle est incomplète, c'est-à-dire que nous n'avons aucune garantie quant à l'issue de la partie. Pour des situations, comme le `go`, où le résultat n'est pas vital, cela ne pose pas de problèmes. Néanmoins, il y a des situations critiques où il n'est pas envisageable de se satisfaire d'une configuration incorrecte. Nous pouvons par exemple citer l'accident intervenu sur la fusée Ariane lors de son vol inaugural et sa première mise en orbite de satellites, le vol 501. Après seulement 36.7 secondes, la fusée se brise en plein vol et explose à une altitude de 4 000 mètres au-dessus du centre spatial de Kourou, en Guyane française. La cause du crash était due à une erreur de *cast*, plus précisément une conversion d'un entier de 64 bits à un entier non signé de 16 bits. Cette exception entraîne alors une propagation en dominos à l'ensemble du programme qui provoque le crash complet du logiciel de navigation et donc de la fusée. Dans ce cas, une réponse du type « vous avez 99.9% de chance que le programme ne plante pas » n'est pas suffisante, et il faut aller plus loin en utilisant une méthode de vérification formelle.

De manière générale, pour vérifier formellement une propriété, il est nécessaire de parcourir l'ensemble des possibilités du problème. L'ennui est qu'il est souvent le cas que le nombre de configurations à considérer est exponentiel par rapport à la taille de l'entrée. Dans le cadre de ce manuscrit, nous nous sommes principalement focalisés sur la résolution de problèmes de décision, c'est-à-dire des problèmes pour lesquels nous souhaitons uniquement déterminer si une solution existe ou pas. Plus précisément, notre champ d'investigation s'est limité à la résolution de problèmes qui peuvent être décidés par une machine de Turing déterministe avec un espace polynomial (c'est-à-dire les problèmes `PSPACE`) et où chaque variable a un domaine binaire (la logique propositionnelle).

D'une certaine manière, les problèmes `PSPACE` peuvent être vus comme des jeux à deux joueurs, où le premier joueur (existentiel) doit trouver un ensemble de coups lui permettant de gagner quelle que soit la réponse du second joueur (universel). Comme nous pouvons nous en douter, un problème sera plus ou moins difficile en fonction du nombre de coups que le premier joueur doit prévoir (le nombre d'alternances des quantificateurs). Dans le cadre qui nous intéresse,

la difficulté la plus faible est représentée par les problèmes de décision qui sont NP-complets, c'est-à-dire les problèmes pouvant être décidés en temps polynomial sur une machine de Turing non déterministe. Et la difficulté la plus élevée est atteinte par le cas où il n'y a pas de limite sur le nombre d'alternance de quantificateurs (les problèmes PSPACE).

Ce manuscrit a été pensé comme un voyage allant de la résolution pratique du problème qui consiste à décider la satisfaisabilité d'une formule exprimée en logique propositionnelle classique (problème NP-complet), à la résolution pratique du problème consistant à décider la satisfaisabilité d'une formule exprimée en logique propositionnelle modale (problème PSPACE-complet). L'aventure que nous relatons à travers ce manuscrit est schématisée par la figure suivante :



Comme toute bonne randonnée, nous commençons notre voyage par la base, c'est-à-dire par présenter le matériel et les concepts nécessaires au voyage. C'est donc naturellement que nous retrouvons au chapitre 2 l'ensemble des notations et des définitions que nous utiliserons tout au long de ce document. Plus précisément, ces préliminaires permettent d'introduire la classe de complexité NP et quelques classes de complexité au-delà de NP, ainsi que les formalismes de la logique propositionnelle classique et de la logique propositionnelle modale.

Ensuite, au chapitre 3, nous traiterons de la résolution pratique du problème NP de référence : SAT. Ce chapitre a pour objectif d'introduire l'algorithme CDCL ainsi que les heuristiques qui le rendent efficace en pratique, et de présenter nos contributions à la résolution du problème SAT en parallèle dans le cadre de l'utilisation d'une architecture à mémoire distribuée. Ce chapitre a une importance capitale puisqu'il est le socle commun à tous les travaux présentés dans ce document. Pour être plus précis, nous verrons dans les chapitres suivants qu'il est possible d'exploiter les solveurs SAT comme des oracles afin d'attaquer des problèmes beaucoup plus difficiles.

Au chapitre 4, nous continuons notre ascension en considérant la résolution de problèmes Δ_2^P, qui sont des problèmes pouvant être résolus en appelant un nombre polynomial de fois un oracle NP. Les travaux présentés dans ce chapitre sont focalisés sur le problème de la restauration de la cohérence d'une formule de la logique propositionnelle exprimée en forme normale conjonctive (CNF). Ainsi, ce chapitre commence par introduire le concept de sous-ensemble de correction minimal (MCS) et par présenter rapidement les méthodes utilisées en pratique afin d'extraire ou d'énumérer les MCSes d'une formule. Nous montrons ensuite comment il est possible d'exploiter intelligemment un solveur SAT lorsque l'objectif est de résoudre un problème de manière incrémentale. Nous terminons ce chapitre par la présentation d'un nouvel algorithme d'extraction de MCS et par la présentation d'un nouveau processus permettant d'accélérer l'énumération de MCSes en considérant des familles de MCSes. Cette étape nous permettra de voir en quoi l'exploitation du résultat retourné par le solveur SAT peut être intéressante pour la résolution de problèmes plus complexes.

Le chapitre 5 traite de nos travaux autour de la résolution pratique du problème #P. Ce type de problème est beaucoup plus complexe que SAT (d'où la pente sur la figure), puisqu'il revient à compter le nombre de chemins qui satisfont la formule considérée. Nous commençons donc ce chapitre en dressant un bref panorama des méthodes utilisées en pratique pour compter de manière exacte le nombre de modèles d'une formule de la logique propositionnelle. Ensuite, nous présentons différentes méthodes de pré-traitement permettant de réécrire une formule exprimée CNF afin d'améliorer les processus de comptage de modèles sur cette dernière. Nous continuerons notre ascension vers #P en présentant nos travaux qui ont eu trait à l'amélioration des processus permettant le comptage de modèles. Plus précisément, nous présentons une nouvelle heuristique et de nouvelles structures de données pouvant être utilisées dans les compteurs de modèles de l'état de l'art. Nous introduisons aussi un nouveau langage de représentation permettant le comptage de modèles, ainsi qu'un compilateur permettant de transformer une formule de la logique propositionnelle dans ce langage. Nous présentons aussi dans ce chapitre un nouvel algorithme récursif pour le problème du comptage de modèles projetés.

Avant de conclure notre expédition, nous terminons l'ascension par la présentation d'un nouveau schéma algorithmique permettant de résoudre des problèmes PSPACE. Nous verrons comment il est possible de résoudre en pratique un problème PSPACE, en résolvant successivement des approximations de ce problème *via* un oracle NP.

Chapitre 2

Préliminaires

Sommaire

Avant de démarrer toute aventure, il est nécessaire d'étudier le terrain sur lequel nous allons évoluer. Ici, nous souhaitons nous promener au delà de NP, c'est donc naturellement que nous commencerons nos préliminaires par une brève introduction à la classe de complexité NP et à quelques classes de complexité au delà de NP. Une fois le terrain cartographié, il est nécessaire de préparer les outils que nous allons utiliser tout au long de notre voyage. Les travaux présentés dans ce manuscrit tournent autour de la logique propositionnelle classique et de la logique propositionnelle modale. Nous présenterons donc formellement ces deux cadres.

L'objectif de ce chapitre n'est pas de réaliser un état de l'art complet sur les domaines sus-cités, mais juste de définir formellement les problèmes abordés dans le mémoire et surtout introduire les notations qui seront utilisées tout au long de notre promenade. Pour un état de l'art plus complet sur ces domaines, le lecteur intéressé pourra se référer aux manuscrits suivants : pour la complexité (Perifel 2014), pour la logique propositionnelle classique (Genesereth et Kao 2016) et pour la logique propositionnelle modale (Chellas 1980).

2.1 Brève introduction à la complexité

La complexité d'un programme est une mesure des ressources nécessaires à son exécution. Les ressources qui sont prises en compte sont, usuellement, le temps et l'espace. La théorie de la complexité étudie les problèmes qui sont calculables avec une certaine quantité de ressources. Il ne faut pas confondre la complexité d'un problème avec sa calculabilité. En effet, la théorie de la complexité se concentre uniquement sur les problèmes qui peuvent être résolus, la question étant de savoir s'ils peuvent être résolus efficacement ou pas en se basant sur une estimation (théorique) souvent dans le pire des cas des ressources nécessaires. Plus précisément, l'objectif de

cette théorie est d'étudier l'évolution des ressources nécessaires à la résolution d'un problème en fonction de la taille des données fournies en entrée. Autrement dit, étant donnée une procédure $\mathcal{P}$, la théorie de la complexité s'intéresse à la variation de la durée ou l'espace nécessaire au calcul de $\mathcal{P}(n)$ en fonction de la taille de l'argument n.

2.1.1 Notions de bases

L'évaluation de la complexité en temps des algorithmes repose sur une définition précise du temps qui ne dépend pas de la vitesse d'exécution de la machine ni de la qualité du code produit par le compilateur. Afin d'estimer plus finement la complexité d'un algorithme, il est nécessaire de s'abstraire de cette notion temporelle dépendante du matériel et de considérer le temps comme le nombre d'opérations élémentaires $T_{\mathcal{P}}(n)$ nécessaires à l'exécution d'un programme pour une certaine donnée de taille n. En pratique, lorsqu'aucune précision n'est apportée, c'est souvent le comportement de l'algorithme dans le pire des cas qui est étudié. En effet, le nombre d'opérations effectuées dans le meilleur des cas n'apporte aucune information sur le comportement réel de l'algorithme. Quant à la valeur obtenue en moyenne, bien que permettant de révéler le comportement « réel » de l'algorithme, elle est difficile à calculer et dépend fortement d'hypothèses effectuées sur la distribution des données. En général, la complexité n'est pas donnée de manière exacte. En pratique, seul un ordre de grandeur asymptotique est calculé. Par exemple, supposons que le nombre d'opérations nécessaire à la terminaison d'un programme $\mathcal{P}$ sur une donnée de taille n soit précisément de $n^2 + 2n + 7$. Dans ces conditions, le programme $\mathcal{P}$ est en $\mathcal{O}(n^2)$. Un programme $\mathcal{P}$ est de complexité $\mathcal{O}(f(n))$ dans le pire des cas s'il existe une certaine constante $c < \infty$ tel que $\lim\limits_{n\to\infty} \frac{T_{\mathcal{P}}(n)}{f(n)} = c$.

Définition 1 (Algorithme polynomial)

Soit $\mathcal{P}$ un algorithme exécuté sur une entrée de taille n. $\mathcal{P}$ est dit polynomial s'il existe un entier i tel que $\mathcal{P}$ est de complexité $\mathcal{O}(n^i)$, c'est-à-dire que le nombre d'opérations nécessaire à la terminaison de $\mathcal{P}(n)$ est majoré par un polynôme en la taille n de l'entrée.

Définition 2 (Algorithme exponentiel)

Soit $\mathcal{P}$ un algorithme exécuté sur une entrée de taille n, $\mathcal{P}$ est simplement exponentiel s'il existe un réel τ strictement supérieur à 1 et un polynôme $f(n)$ en n tels que $\mathcal{P}$ est de complexité $\mathcal{O}(\tau^{f(n)})$.

Du point de vue pratique, il y a une nette différence de temps de résolution entre les algorithmes polynomiaux et les algorithmes exponentiels. Plus particulièrement, les problèmes d'une complexité exponentielle (et plus) sont généralement impossibles à résoudre sur des données de taille raisonnable ($n > 100$). Compte tenu de cette particularité, deux questions s'imposent :

— existe-t-il des problèmes qui n'admettent pas d'algorithme polynomial ?

— si oui, pouvons nous déterminer qu'un problème n'admet pas d'algorithme polynomial ?

Ce sont à ces questions que tente de répondre la théorie de la complexité. Pour cela, il est possible d'utiliser un modèle de calcul théorique pour les ordinateurs et les algorithmes qui permet de travailler sur ces problèmes : la machine de Turing. La machine de Turing se compose d'une partie de contrôle et d'une bande infinie sur laquelle se trouvent écrits des symboles. La partie de contrôle est constituée d'un nombre fini d'états possibles et de transitions qui régissent les calculs de la machine. Les symboles de la bande sont lus et écrits par l'intermédiaire d'une tête de lecture. Lorsque les transitions sont réalisées de manière déterministe ces machines sont dites déterministes et elles peuvent être vues comme une abstraction des ordinateurs « classiques ». Si pour une machine de Turing plusieurs transitions sont activables pour un état donné alors elle est dite non déterministe. En dépit de la contradiction apparente des termes, une machine de Turing déterministe est un cas particulier de machine de Turing non déterministe.

Bien que d'apparence très frustes, les machines de Turing fournissent un modèle universel pour les ordinateurs et les algorithmes. Elles permettent de calculer toutes les fonctions récursives [1] et la « thèse de Church » (de ce fait présentée parfois comme la « thèse de Church-Turing ») peut se reformuler, en termes non techniques, de la manière suivante : tout ce qui est calculable l'est avec une machine de Turing (Papadimitriou 1994). Cette propriété permet l'utilisation de la machine de Turing comme référence dans les définitions théoriques des classes de complexité.

2.1.2 Classes de complexité

Nous allons maintenant nous intéresser à l'utilisation que nous pouvons faire des machines de Turing pour définir des classes de problèmes selon leurs difficultés intrinsèques, ce que nous appelons de nouveau complexité (des problèmes et non plus des algorithmes). Dans ce cas, la complexité spatiale et la complexité temporelle sont respectivement évaluées en fonction du nombre de cases du ruban et du nombre de transitions nécessaires à la résolution du problème.

Définition 3 (TIME)

Un problème appartient à la classe de complexité $\texttt{TIME}(f(n))$ si et seulement s'il existe une machine de Turing déterministe le résolvant en temps $\mathcal{O}(f(n))$.

Définition 4 (SPACE)

Un problème appartient à la classe de complexité $\texttt{SPACE}(f(n))$ si et seulement s'il existe une machine de Turing déterministe le résolvant en espace $\mathcal{O}(f(n))$.

Il existe différentes natures de problèmes. Focalisons nous dans un prermier temps sur les *problèmes de décision* (parfois aussi appelés *problèmes de reconnaissance*), c'est-à-dire posant une

1. Les fonctions récursives sont les fonctions calculables, autrement dit les fonctions dont les valeurs peuvent être calculées à partir de leurs paramètres par un processus mécanique.

question dont la réponse est « Oui » ou « Non ». En effet, de nombreux problèmes informatiques peuvent se réduire à des problèmes de décision.

Définition 5 (Problème de décision)

Un problème Π de décision est un problème n'ayant que deux solutions possibles : « Oui » ou « Non ». C'est-à-dire que l'ensemble D_Π des instances de Π peut être scindé en deux ensembles disjoints :

1. Y_Π : les instances telles qu'il existe un programme les résolvant et répondant « Oui » ;
2. N_Π : les instances pour lesquelles la réponse est « Non ».

Nous pouvons donc assimiler un problème de décision Π au langage formel Y_Π formé par ses instances positives.

Définition 6 (Problème de décision complémentaire)

Le problème complémentaire d'un problème de décision Π est le problème de décision Π^c tel que :

$$D_{\Pi^c} = D_\Pi \text{ et } Y_{\Pi^c} = N_\Pi.$$

La théorie de la complexité repose sur la définition de classes de complexité qui permettent d'ordonner les problèmes en fonction des algorithmes qui existent pour les résoudre.

Définition 7 (Classe `P`)

La classe de complexité `P` regroupe l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par une machine de Turing déterministe.

Par définition, la classe `P` (pour polynomial) contient les problèmes de décision polynomiaux, c'est-à-dire que nous pouvons résoudre à l'aide d'un algorithme déterministe en temps polynomial par rapport à la taille de l'instance. S'il est vrai que la classe `P` contient de nombreux problèmes de décision, certains autres ne sont pas connus pour être polynomiaux (ce qui ne signifie pas qu'ils ne le sont pas). Nous allons maintenant définir une classe plus vaste : la classe `NP`.

Définition 8 (Classe `NP`)

La classe de complexité `NP` regroupe l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial par une machine de Turing non déterministe.

Les problèmes de la classe NP (pour *Non déterministe Polynomial*) sont les problèmes de décision pour lesquels la réponse « Oui » peut être décidée par un algorithme non déterministe en un temps polynomial par rapport à la taille de l'instance. De façon équivalente, c'est la classe des problèmes qui admettent un algorithme polynomial qui, étant donnée une solution du problème NP (certificat) de taille polynomiale en la taille de l'instance correspondante, sont capables de répondre si « Oui » ou « Non » cette instance est positive. Il est possible de définir le problème complémentaire (dual) de tout problème NP.

Définition 9 (Classe CoNP)

La classe CoNP regroupe l'ensemble des problèmes de décision dont les problèmes complémentaires appartiennent à la classe NP.

Puisqu'une machine de Turing déterministe peut être considérée comme une machine de Turing non déterministe particulière, nous obtenons immédiatement que $\texttt{P} \subseteq \texttt{NP}$ et $\texttt{P} \subseteq \texttt{CoNP}$. La question de déterminer si ces inclusions sont strictes ou s'il s'agit d'une égalité constitue l'une des questions ouvertes fondamentales de la théorie de la complexité. Sa résolution aurait bien sûr des conséquences en théorie de la calculabilité, mais aussi des répercussions pratiques dans tous les domaines où interviennent l'informatique et l'ordinateur, c'est-à-dire finalement à peu près tous les domaines ...

Afin de définir une véritable classification des problèmes une notion de réduction fonctionnelle polynomiale est utilisée. Intuitivement, la notion de réduction permet d'établir qu'un problème est au plus aussi « dur » qu'un autre si le premier problème se réduit « facilement » au second (le second est au moins aussi difficile que le premier).

Définition 10 (Réduction fonctionnelle polynomiale)

Soient deux problèmes de décisions Π et Π', et un algorithme f qui prend en entrée des instances de Π et qui retourne des instances de Π'. f est appelée réduction fonctionnelle polynomiale si et seulement si :

- f est un algorithme polynomial ;
- PI est une instance positive de Π si et seulement si $f(\texttt{PI})$ est une instance positive de Π'.

Définition 11 (Problème C-difficile)

Soit C une classe de complexité, un problème est dit C-difficile si et seulement s'il est au moins aussi difficile que tous les problèmes dans C.

Il est important de noter que ce n'est pas parce qu'un problème est dans une classe C qu'il est difficile pour cette classe (par exemple tout problème Π de P est dans NP mais ce n'est pas pour autant que Π est difficile pour la classe NP).

Définition 12 (Complétude)

Un problème Π est complet pour sa classe de complexité C si et seulement si $\Pi \in \mathtt{C}$ et Π est C-difficile.

Il est possible de situer schématiquement sur un diagramme de Venn (voir figure 2.1) les classes de complexité P, NP et CoNP ainsi que leur complétude en se basant sur les deux conjectures $\mathtt{P} \neq \mathtt{NP}$ et $\mathtt{CoNP} \neq \mathtt{NP}$ (Papadimitriou 1994).

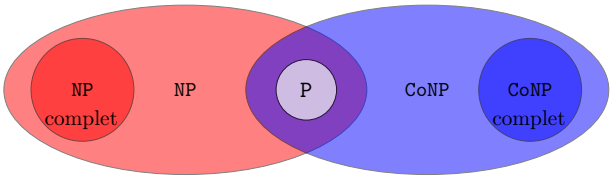


FIGURE 2.1 – Structuration possible des classes P, NP et CoNP.

Une investigation plus profonde nous conduit à nous interroger sur la possibilité de poursuivre la classification des problème de décision au delà de NP. Cette classification, appelée hiérarchie polynomiale, a pour base les classes P, CoNP et NP et peut être définie à l'aide d'une machine de Turing à oracle. Comme son nom l'indique, une machine de Turing à oracle a pour mission de deviner une solution permettant de justifier la réponse « oui ». Nous pouvons donc classer un problème en fonction d'un oracle A, c'est-à-dire d'un dispositif de calcul permettant de décider en temps constant l'appartenance d'un mot quelconque (stocké sur un ruban additionnel de la machine) au langage noté A également.

Définition 13 (classe $\mathtt{C}^{\mathtt{A}}$)

Soit C une classe de complexité quelconque déterministe ou non, $\mathtt{C}^{\mathtt{A}}$ est la classe des langages décidés par une machine de Turing et dans un temps borné comme dans C, mais utilisant un oracle A.

La notation $\mathtt{C}^{\mathtt{A}}$ peut être étendu au cas où A est un ensemble de langages, en particulier une classe de complexité. Ainsi $\mathtt{P}^{\mathtt{NP}}$ représente l'ensemble de langages décidés en temps polynomial par une machine de Turing déterministe munie d'un oracle capable de répondre en un pas de calcul aux questions pour tous les langages décidés en temps polynomial par une machine de Turing non déterministe. À l'aide de cette notion, il est possible de définir un nouvel ensemble de classes de manière récursive :

Définition 14 (Hiérarchie polynomiale)

La hiérarchie polynomiale PH (Stockmeyer 1976), est une classe de complexité définie récursivement à partir de ses sous-classes à l'aide des machines de Turing avec oracle de la manière suivante :

- $\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathtt{P}$;
- $\Delta_{k+1}^p = \mathtt{P}^{\Sigma_k^p}$;
- $\Sigma_{k+1}^p = \mathtt{NP}^{\Sigma_k^p}$;
- $\Pi_{k+1}^p = \mathtt{Co}\Sigma_{k+1}^p$;
- $\mathtt{PH} = \cup_{i \geq 0} \Sigma_i^p$.

Les problèmes étant classés de façon incrémentale, il a été nécessaire de définir un « premier » problème NP-complet afin de classer tous les autres. Ce premier problème à avoir été démontré NP-complet, par Cook (1971), est le problème de la satisfaisabilité propositionnelle (voir la section suivante pour la définition de ce problème). Un point commun entre toutes ces classes de complexité est qu'il existe un algorithme permettant de répondre à l'appartenance d'un problème à une classe en espace polynomial. Ces problèmes sont dans la classe de complexité PSPACE.

Définition 15 (classe PSPACE)

La classe PSPACE est l'ensemble des langages reconnus par une machine de Turing fonctionnant en espace polynomial.

Nous déduisons les inclusions suivantes : $\mathtt{P} \subseteq \mathtt{NP} \subseteq \mathtt{PH} \subseteq \mathtt{PSPACE}$. La question de savoir si ces inclusions sont strictes est ouverte, nous supposons dans la suite que c'est le cas.

Jusqu'à présent, nous nous sommes uniquement intéressés aux problèmes de décision, mais nous pouvons également vouloir considérer des problèmes dont le résultat attendu est un peu plus complexe. En particulier, il existe des situations où un problème est exprimé comme deux problèmes dont l'un est dans NP et l'autre dans CoNP. Dans ce cas, il est clair que le problème ainsi exprimé n'est ni dans NP ni dans CoNP, il est en fait dans la classe DP :

Définition 16 (classe DP)

Un problème Π est dans la classe DP si et seulement s'il y a deux problèmes $\Pi_1 \in \mathtt{NP}$ et $\Pi_1 \in \mathtt{CoNP}$ tels que $\Pi = \Pi_1 \cap \Pi_2$.

Cette classe a été introduite par Papadimitriou et Yannakakis (1984) et certains problèmes ont été montrés complets pour cette classe (Papadimitriou et Wolfe 1988). Il est important de noter que DP n'est pas $\mathtt{NP} \cap \mathtt{CoNP}$. En fait, DP n'est vraisemblablement pas contenue dans $\mathtt{NP} \cup \mathtt{CoNP}$.

Il peut être aussi utile de considérer des problèmes où le résultat attendu n'est pas uniquement « vrai » ou « faux », mais une valeur qui peut être exprimé comme une valeur numérique. Pour les problemes fonction associée à une relation binaire $R(I, w)$, il existe des classes équivalentes à P et NP : FP et FNP. La classe FP contient les problèmes pour lesquels il existe un algorithme A de complexité polynomiale tel que $\forall I, A(I)$ renvoit w telle que $R(I, w)$ est vrai (ou « faux » si une telle w n'existe pas). La classe FNP contient quant à elle les problème pour lesquels il est possible de vérifier $R(I, w)$ en temps polynomial.

En particulier, plutôt que de décider, comme dans NP, s'il existe une solution à un problème, nous pouvons vouloir compter le nombre de solutions. La classe de comptage, nommée #P, a été introduite par Valiant (1979) et a plusieurs intérêts. De nombreux problèmes pratiques peuvent se réduire à un problème de comptage : inférence probabiliste (Sang et al. 2005, Apsel et Brafman 2012, Choi et al. 2013) certaines formes de planification (Palacios et al. 2005, Domshlak et Hoffmann 2006), vérification (Heinz et Sachenbacher 2009, Klebanov et al. 2013), sécurité (Feiten et al. 2012), ... D'un point de vue théorique, le théorème de (Toda 1989) met en exergue toute la puissance qu'apporte le comptage de modèles : il indique que $\mathtt{PH} \subseteq \mathtt{P}^{\#\mathtt{P}}$, donc que tout problème de PH peut être résolu en temps polynomial (déterministe) pour autant qu'un oracle de comptage #P existe. De plus, il est important de noter que même pour des fragements très contraints de la logique propositionnelle le problème reste difficile et non approximable (Creignou et al. 2001).

Définition 17 (classe #P)

Une fonction f est dans #P si et seulement s'il existe une machine de Turing non déterministe N fonctionnant en temps polynomial telle que pour tout x, $f(x)$ est le nombre de chemins acceptants de $N(x)$.

Le problème qui consiste à compter le nombre de modèles d'une formule propositionnelle, appelé #SAT, est connu pour être un problème #P-complet. Dit autrement, tout problème qui se trouve dans la classe de complexité #P peut se réduire à #SAT (pour un type de réduction particulier, dit parcimonieux, c'est-à-dire préservant les solutions, que nous ne détaillerons pas).

Nous venons à deux reprises de mentionner le concept de formule propositionnelle, mais nous n'avons pas encore introduit cette notion. La section suivante permet de combler ce manque en présentant formellement la logique propositionnelle.

2.2 La logique propositionnelle classique

La logique est une discipline très ancienne qui nous vient de l'Antiquité grecque et qui sert à préciser ce qu'est un raisonnement correct, indépendamment du domaine d'application. Un raisonnement est un processus permettant, à partir d'un ensemble de règles et d'hypothèses de dériver des conclusions à partir de prémisses par l'application de règles d'inférence. Les conclusions ainsi obtenues ne disent rien sur la vérité des hypothèses, mais seulement qu'à partir de la vérité des hypothèses nous pouvons déduire la vérité des conclusions.

Il n'existe pas une logique unique, mais des logiques correspondant à des choix quant à la construction des formules, aux systèmes de preuve et à la notion de vérité. La logique propositionnelle est une logique qui s'intéresse à la notion de propositions. Une proposition est un énoncé qui peut être « vrai » ou « faux ». La logique propositionnelle permet de construire des raisonnements à partir de propositions et d'un ensemble de connecteurs logiques. Dans la suite, nous décrivons le langage de la logique propositionnelle classique et sa sémantique. Nous situons cette dernière d'un point de vue complexité calculatoire. Pour terminer, nous présentons certaines formules particulières : les formes normales.

2.2.1 Syntaxe

Pour spécifier la syntaxe, il est nécessaire de définir les symboles pouvant être utilisés pour formuler des énoncés. Ces symboles forment l'alphabet du langage :

Définition 18 (Alphabet de la logique propositionnelle)

L'alphabet de la logique propositionnelle est constitué :

- des symboles $\bot$ et $\top$ représentant respectivement les constantes propositionnelles « faux » et « vrai » ;
- d'un ensemble infini dénombrable de symboles (ou variables) propositionnels, noté PS ;
- de l'ensemble des connecteurs logiques usuels : le symbole $\wedge$ est utilisé pour la conjonction, $\vee$ pour la disjonction, $\neg$ pour la négation, $\Rightarrow$ pour l'implication, $\Leftrightarrow$ pour l'équivalence et $\oplus$ pour le ou exclusif ;
- des symboles de ponctuation "(" et ")".

L'ensemble des formules formées à partir de cet alphabet est noté $\mathcal{CPL}$ et est défini inductivement de la manière suivante :

Définition 19 (Formule propositionnelle)

Soit PS un ensemble fini de variables propositionnelles. L'ensemble des formules bien formées de $\mathcal{CPL}$ est le plus petit ensemble tel que :

- tout élément de $PS \cup \{\bot, \top\}$ est élément de $\mathcal{CPL}$;
- si $\alpha \in \mathcal{CPL}$ alors $(\neg\alpha)$ est élément de $\mathcal{CPL}$;
- si α et β sont des éléments de $\mathcal{CPL}$ alors $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$, $(\alpha \Leftrightarrow \beta)$ et $(\alpha \oplus \beta)$ sont éléments de $\mathcal{CPL}$.

Exemple 1

Soit $\{a, b, c, d\}$ un ensemble de symboles propositionnels, alors $(((a \Leftrightarrow b) \vee (\neg d)) \wedge c)$ appartient à $\mathcal{CPL}$, contrairement à $(((a \Leftrightarrow b)$ et $((\neg d) \wedge)$.

Il est commun de nommer *littéral* toute formule propositionnelle particulière qui ne fait intervenir qu'une variable propositionnelle et au plus une seule occurrence du connecteur $\neg$.

Définition 20 (Littéral)

Un *littéral* est un symbole propositionnel ou bien sa négation. Soit ℓ un symbole propositionnel, alors ℓ est appelé *littéral positif*, $\neg\ell$ est appelé *littéral négatif*, ℓ et $\neg\ell$ sont des *littéraux complémentaires* et le complémentaire d'un littéral ℓ est noté $\tilde{\ell}$.

Par souci de clarté, il est courant d'omettre certaines parenthèses à condition de ne pas rendre la proposition ambiguë. Il arrive aussi que certaines parenthèses soient remplacées par les symboles "[" et "]" afin de rendre plus visibles certaines formules ou sous-formules. Sur notre exemple la première formule peut s'écrire : $[((a \Leftrightarrow b) \vee (\neg d)) \wedge c]$. Nous choisissons dans la suite de ce manuscrit d'adopter les conventions suivantes :

- les variables sont représentées par des lettres latines minuscules ;
- les ensemble de variables sont représentés par des lettres latines majuscules ;
- les formules sont représentées par des lettres grecques majuscules ;
- les ensembles de formules sont représentés par des lettres grecques majuscules en gras ;
- soit Σ une formule propositionnelle, $\mathcal{V}_\Sigma$ dénote l'ensemble des variables propositionnelles de Σ ;
- la négation d'une formule Σ est notée aussi $\overline{\Sigma}$.

Dans la section suivante, nous nous intéressons à la sémantique des formules bien formées, c'est-à-dire aux conditions sous lesquelles un énoncé est « vrai » ou « faux ».

2.2.2 Sémantique

La sémantique de la logique propositionnelle associe une signification à ses formules et explique les conditions qui rendent les formules vraies ou fausses. La notion de vérité d'une formule est régie par deux postulats. Premièrement, le postulat de bivalence permet d'établir que la valeur de vérité d'un symbole propositionnel appartient à l'ensemble des valeurs de vérité $\mathbb{B} = \{vrai, faux\}$. Deuxièmement, le postulat de vérifonctionnalité permet de déterminer la valeur de vérité d'une formule complexe en fonction des valeurs de vérité de ses composants immédiats. Il s'ensuit que pour déterminer la valeur de vérité d'une formule il suffit de connaître les valeurs de vérité de ces composantes les plus simples, c'est-à-dire les variables et les opérateurs. La sémantique de $\top$ (resp. $\top$) est $vrai$ (resp. $faux$) et la sémantique de l'ensemble des opérateurs logiques est définie comme reporté dans le tableau suivant :

x	y	$\neg x$	$(x \vee y)$	$(x \wedge y)$	$(x \Rightarrow y)$	$(x \Leftrightarrow y)$	$(x \oplus y)$
$vrai$	$vrai$	$faux$	$vrai$	$vrai$	$vrai$	$vrai$	$faux$
$vrai$	$faux$	$faux$	$vrai$	$faux$	$faux$	$faux$	$vrai$
$faux$	$vrai$	$vrai$	$vrai$	$faux$	$vrai$	$faux$	$vrai$
$faux$	$faux$	$vrai$	$faux$	$faux$	$vrai$	$vrai$	$faux$

La sémantique permet de définir des règles d'interprétation pour les formules à partir de la valeur de vérité de chacune des propositions qui les composent. Ces règles sont les suivantes :

Définition 21 (Interprétation)

Une *interprétation* $\mathcal{I}$ est une application de PS dans $\mathbb{B}$ qui attribue une valeur de vérité à chaque symbole propositionnel. Soit $\mathcal{I}$ une interprétation, la sémantique des propositions selon cette interprétation $\mathcal{I}$ est définie inductivement telle que $\forall \alpha, \beta \in \mathcal{CPL}$:

- $\mathcal{I}(\top) = vrai$ et $\mathcal{I}(\bot) = faux$;
- $\mathcal{I}(\neg\alpha) = vrai$ si et seulement si $\mathcal{I}(\alpha) = faux$;
- $\mathcal{I}(\alpha \circ \beta) = \mathcal{I}(\alpha) \circ \mathcal{I}(\beta)$ tel que $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow, \oplus\}$ respecte la sémantique énoncée précédemment.

La sémantique d'une formule dans une interprétation $\mathcal{I}$ est donc définie par la valeur de vérité dans $\mathcal{I}$ des variables qui la composent. Toute interprétation $\mathcal{I}$ est communément représentée comme l'ensemble des littéraux interprétés à vrai et dans ce cas la sémantique des opérateurs ensemblistes est conservée.

Exemple 2

Soient $\Sigma = (a \vee b) \wedge (a \Rightarrow c)$ une formule propositionnelle et $\mathcal{I} = \{a, b, \neg c\}$ une interprétation. Nous avons $\mathcal{I}(\Sigma) = \mathcal{I}(\top \vee \top) \wedge \mathcal{I}(\top \Rightarrow \bot) = \top \wedge \bot = \bot$.

L'ensemble des interprétations, noté $\mathfrak{S}$, pouvant être construit sur les variables d'une formule propositionnelle Σ a un cardinal de $2^{\mathcal{V}_\Sigma}$. Une interprétation n'est pas forcément définie sur l'ensemble des variables propositionnelles de la formule. Dans ce cas, l'interprétation peut être partielle ou incomplète, ce qui se définit formellement de la manière suivante :

Définition 22 (Interprétation partielle, complète et incomplète)

Soit Σ une formule propositionnelle, une interprétation $\mathcal{I}$ construite sur les variables de Σ est dite :

- partielle si $|\mathcal{I}| \leq |\mathcal{V}_\Sigma|$;
- complète si $|\mathcal{I}| = |\mathcal{V}_\Sigma|$;
- incomplète si $|\mathcal{I}| < |\mathcal{V}_\Sigma|$.

Dans la suite de ce manuscrit, lorsqu'aucune information n'est apportée sur la nature de l'interprétation, elle est considérée comme complète. La définition suivante permet d'établir les notions de modèle, de satisfaisabilité, d'impliquant, d'impliqué et de conséquence logique d'une formule propositionnelle.

Définition 23 (Validité, modèle, satisfaisabilité, impliquant, impliqué et conséquence logique)

Soient Σ et Ψ deux formules, nous avons :

- un **modèle** $\mathcal{I}$ de Σ, noté $\mathcal{I} \models \Sigma$, est une interprétation $\mathcal{I}$ telle $\mathcal{I}(\Sigma) = \top$;
- une interprétation $\mathcal{I}$ **falsifie** Σ, notée $\mathcal{I} \not\models \Sigma$, si $\mathcal{I}(\Sigma) = \bot$. Une telle interprétation est aussi appelée **contre-modèle** ;
- Σ est **satisfiable** si elle admet au moins un modèle, sinon elle est **insatisfiable** ;
- Σ est **valide** si elle n'admet pas de contre-modèle ;
- si tout modèle de Ψ est modèle de Σ, noté $\Psi \models \Sigma$, alors Σ est une conséquence logique de Ψ ;
- un **impliquant** de Σ est une conjonction ρ de littéraux telle $\rho \models \Sigma$. Un **impliquant premier** de Σ est un impliquant qui est minimal pour l'inclusion ;
- un **impliqué** α de Σ est une disjonction de littéraux telle que $\Sigma \models \alpha$. Un **impliqué premier** de Σ est un impliqué qui est minimal pour l'inclusion ;
- Ψ et Σ sont logiquement équivalentes, noté $\Psi \equiv \Sigma$, si $\Psi \models \Sigma$ et $\Sigma \models \Psi$.

Le théorème suivant permet d'énoncer un résultat fondamental en démonstration automatique (preuve par l'absurde).

Théoreme 1 (Preuve par l'absurde)

Soient Σ et Ψ deux formules propositionnelles, $\Sigma \models \Psi$ si et seulement si $\Sigma \wedge \neg\Psi$ est une formule insatisfiable.

Ce théorème montre que prouver l'implication sémantique revient à prouver l'incohérence d'une formule. Il est très important en pratique puisque la grande majorité des algorithmes de démonstration automatique présentés dans ce manuscrit exploitent ce théorème.

Dans la section suivante, nous présentons différentes formes normales et nous énonçons le fait que toute formule propositionnelle peut être définie sous l'une de ces formes.

2.2.3 Formes normales

Dans cette section, nous nous intéressons aux différentes formes normales. Plus particulièrement, nous énonçons un théorème permettant de limiter l'établissement de la satisfaction d'une formule propositionnelle au cas d'une formule mise sous forme normale conjonctive.

Nous commençons par introduire les notions de clauses et de termes :

Définition 24 (Clause et terme/cube)

Soit PS un ensemble de variables, nous avons :

- une **clause** est soit $\bot$ soit une disjonction de littéraux, c'est-à-dire une formule de la forme $(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n)$ où chaque ℓ_i est un littéral associé à une variable de PS ;
- un **terme/cube** est soit $\top$ soit une conjonction de littéraux, c'est-à-dire une formule de la forme $(\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n)$ où chaque ℓ_i est un littéral associé à une variable de PS.

Lorsqu'il n'y a aucune ambiguïté, les clauses et les termes sont représentés comme des ensembles de littéraux. Dans ce cas, comme pour les interprétations, la signification des symboles ensemblistes définie sur les littéraux par rapport aux clauses et aux termes est conservée. Nous définissons à présent trois cas particuliers de formules propositionnelles.

Définition 25 (Formes normales)

Nous distinguons trois formes normales particulières pour les propositions :

- une proposition est sous forme normale négative (`NNF` pour « *Negative Normal Form* ») si elle est exclusivement constituée de conjonctions, de disjonctions et de littéraux ;
- une proposition est sous forme normale disjonctive (`DNF` pour « *Disjonctive Normal Form* ») si c'est une disjonction de termes ;
- une proposition est sous forme normale conjonctive (`CNF` pour « *Conjonctive Normal Form* ») si c'est une conjonction de clauses.

Exemple 3

Les formules $[(a \vee b) \vee ((\neg c) \wedge d)]$, $[(a \wedge b) \vee ((\neg c) \wedge d)]$ et $[(a \vee b) \wedge ((\neg c) \vee d)]$ sont respectivement sous forme normale négative, disjonctive et conjonctive.

De la même manière que les clauses et les termes, les formules `CNF` et `DNF` sont respectivement représentées comme des ensembles de clauses et de termes. Comme nous pouvons le remarquer les formes normales conjonctives et disjonctives sont des cas spécifiques de la forme normale négative. De plus, la propriété suivante spécifie que toute formule peut être mise sous forme normale.

Propriété 1

Toute formule de la logique propositionnelle peut être réécrite sous une forme normale.

Cependant, la transformation classique peut nécessiter une croissance exponentielle de la taille de la formule obtenue. Néanmoins, Tseitin (1968) propose une approche permettant de transformer en temps (et donc espace) linéaire toute formule en une formule sous forme normale conjonctive équivalente du point de vue de la satisfaisabilité et du nombre de modèles. Cette transformation permet de focaliser les recherches sur le problème de satisfaisabilité au cas des formules `CNF`.

Définition 26 (Problème `SAT`)

Le problème `SAT` est le problème de décision qui consiste à déterminer si une formule sous forme normale conjonctive possède ou pas un modèle.

Dans la suite de ce manuscrit, lorsqu'il est fait référence à une formule propositionnelle, sauf mention contraire, il est admis qu'il s'agit d'une formule sous forme normale conjonctive.

À présent, nous soulignons quelques particularités concernant l'aspect syntaxique et sémantique des clauses.

Définition 27 (Aspect syntaxique des clauses)

Une clause α est :

- **satisfaite** par une interprétation $\mathcal{I}$ si au moins un littéral de α est affecté à vrai ;
- **falsifiée** par une interprétation $\mathcal{I}$ si tous les littéraux de α sont affectés à faux ;
- **unisatisfaite** par une interprétation $\mathcal{I}$ si un seul littéral de α est affecté à vrai et que les autres sont affectés à faux ;
- **unitaire**, **binaire**, **ternaire** et **n-aire** si et seulement si elle contient respectivement exactement un, deux, trois et $n > 3$ littéraux différents ;
- **positive**, **négative** ou **mixte** si et seulement si elle est constituée respectivement uniquement de littéraux positifs, négatifs ou positifs et négatifs ;
- **vide**, noté $\bot$, si et seulement si elle ne contient aucun littéral. Elle est par définition insatisfaisable ;
- **tautologique** si et seulement si elle contient au moins deux littéraux complémentaires. Elle est par définition valide ;
- **Horn** (respectivement **reverse-Horn**) si et seulement si elle contient au plus un littéral positif (respectivement négatif).

Dans certaines conditions, le choix des clauses utilisées dans une formule peut influencer sa complexité. La définition suivante permet d'établir certaines restrictions de problème SAT.

Définition 28 (Différents types d'instances)

Soit Σ une formule CNF, Σ est une instance :

- k-SAT si toutes les clauses contiennent au plus $k \in \mathbb{N}$ littéraux. Hormis 0-SAT, 1-SAT et 2-SAT (Cook 1971), le problème k-SAT est NP-complet ;
- Horn-SAT (respectivement Reverse-Horn-SAT) si toutes les clauses de Σ sont Horn (respectivement Reverse-Horn). Le test de satisfaisabilité d'un ensemble de clauses de Horn ou Reverse-Horn est polynomial voire même linéaire (Minoux 1988, Dalal 1992, Rauzy 1995) ;
- Horn-renommable si et seulement s'il existe un renommage ρ des littéraux de Σ tel que $\rho(\Sigma)$ soit une formule Horn. Déterminer un tel renommage [a] est un problème polynomial (Lewis 1978).

a. Un renommage est une application de l'ensemble des littéraux dans lui-même qui change certains littéraux en leur complémentaire (une telle application s'étend de façon unique aux formules).

Plusieurs techniques peuvent être appliquées sur les clauses d'une formule Σ sans pour autant en changer sa satisfaisabilité. Ces techniques permettent soit de supprimer des clauses soit de supprimer des littéraux de clauses de Σ.

La première approche présentée est la méthode de résolution. Informellement, considérons deux clauses $(\alpha \vee x)$ et $(\beta \vee \neg x)$. En supposant qu'il existe une interprétation qui satisfait ces clauses, si cette interprétation satisfait x (respectivement $\neg x$) alors nous pouvons en déduire qu'elle satisfait β (respectivement α), dans les deux cas cette interprétation satisfait donc $\alpha \vee \beta$, ce qui s'écrit plus formellement :

Définition 29 (résolution)

Soient $\alpha_1 = (x \vee \beta_1)$ et $\alpha_2 = (\neg x \vee \beta_2)$ deux clauses ayant en commun une variable x présente dans les deux clauses sous la forme de littéraux complémentaires, la clause $\alpha = (\beta_1 \vee \beta_2)$ peut être obtenue par la suppression de toutes les occurrences du littéral x et $\neg x$ dans α_1 et α_2, et en ne conservant qu'une occurrence de chaque littéral. Cette opération, notée $\eta[x, \alpha_1, \alpha_2]$, est appelée résolution et la clause produite est appelée résolvante de α_1 et α_2 sur x.

Exemple 4

Soient $\alpha = (a \vee b \vee c)$ et $\beta = (\neg a \vee c \vee d)$ deux clauses, nous avons $\eta[a, \alpha, \beta] = (b \vee c \vee d)$.

L'opération de résolution peut être étendue à un ensemble de clauses (Eén et Biere 2005) de la manière suivante.

Définition 30 (Résolution appliquée à un ensemble de clauses)

Soient Σ_x un ensemble de clauses contenant le littéral x et $\Sigma_{\neg x}$ un ensemble de clauses contenant le littéral $\neg x$, la résolution de Σ_x et $\Sigma_{\neg x}$ est définie comme suit :

$$\eta[x, \Sigma_x, \Sigma_{\neg x}] = \{\eta[x, \alpha, \beta] \text{ telle que } \alpha \in \Sigma_x \text{ et } \alpha \in \Sigma_{\neg x}\}$$

Exemple 5

Soient $\Sigma_1 = \{(a \vee b), (a \vee c)\}$ et $\Sigma_2 = \{(\neg a \vee d)\}$ deux formules, nous avons que $\eta[a, \Sigma_1, \Sigma_2] = \{(b \vee d), (c \vee d)\}$.

La deuxième technique, appelée *subsumption*, permet la suppression de clauses de la formule.

Définition 31 (Sous-sommation ou subsumption)

Une clause $\alpha_1 = (a_1 \vee a_2 \vee \cdots \vee a_n)$ subsume une clause $\alpha_2 = (a_1 \vee a_2 \vee \cdots \vee a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$ si α_2 contient tous les littéraux de α_1.

Exemple 6

Soient $\alpha_1 = (a \vee b)$ et $\alpha_2 = (a \vee b \vee c)$ deux clauses, la clause α_1 subsume la clause α_2.

Il est possible d'effectuer la clôture par subsumption d'une formule. Cette clôture s'énonce de la manière suivante :

Définition 32 (Clôture par subsumption)

Une formule Σ est close par subsumption si et seulement si $\forall \alpha_1 \in \Sigma$, $\nexists \alpha_2 \in \Sigma$ telle que $\alpha_1 \neq \alpha_2$ et α_2 subsume α_1.

Exemple 7

La formule $\Sigma = \{(a \vee b), (a \vee c), (\neg a \vee d)\}$ est close par subsumption.

La troisième technique présentée est une combinaison des deux précédentes. Cette approche appelée *self-subsumption* est définie formellement de la manière suivante.

Définition 33 (Self-subsumption)

La clause α_1 self-subsume la clause α_2 sur la variable x si et seulement si la résolvante de α_1 et α_2 sur x subsume α_2.

Exemple 8

Soient $\alpha_1 = (a \vee b)$ et $\alpha_2 = (\neg a \vee b \vee c)$ deux clauses, les clauses α_1 et α_2 se résolvent en a et produisent la résolvante $(b \vee c)$ qui subsume α_2. α_2 est donc self-subsumée par α_1 en a.

Les deux dernières techniques présentées ci-avant permettent la suppression de clauses dites *redondantes*. Elles sont, dans ces deux cas, syntaxiques mais peuvent être étendues sémantiquement de la manière suivante :

Définition 34 (Clause redondante)

Soient Σ une formule `CNF` et α une clause de Σ, α est redondante dans Σ si et seulement si $\Sigma \setminus \{\alpha\} \models \alpha$.

Lorsque une base de connaissance ne possède aucune clause redondante il est dit que cette base est irredondante minimale. La dernière méthode présentée ci dessous, consiste à simplifier une formule par un littéral. Cette approche est différente des précédentes puisqu'elle peut changer la satisfaisabilité de la formule considérée.

Définition 35 (Conditionnement par un littéral)

Soit Σ une formule, nous notons $\Sigma_{|\ell}$ la formule obtenue en éliminant de Σ les clauses où le littéral ℓ apparaît et en supprimant l'occurrence du littéral $\tilde{\ell}$ des clauses le contenant. Formellement, nous avons :

$$\Sigma_{|\ell} = \{\alpha | \alpha \in \Sigma \text{ et } \{\ell, \tilde{\ell}\} \cap \alpha = \varnothing\} \cup \{\alpha \setminus \{\tilde{\ell}\} | \alpha \in \Sigma, \tilde{\ell} \in \alpha\}$$

Exemple 9

Soit $\Sigma = \{(a \vee b \vee \neg c), (\neg a \vee c \vee \neg d), (\neg a \vee \neg b \vee d)\}$, $\Sigma_{|\neg a} = (b \vee \neg c)$.

Il est possible d'étendre cette notion à un ensemble de littéraux de la manière suivante :

Définition 36 (Conditionnement par un ensemble de littéraux)

Soit Σ une formule propositionnelle et $\mathcal{A} = \{x_1, x_2, \ldots, x_n\}$ un ensemble cohérent de littéraux, $\Sigma_{|\mathcal{A}} = (\ldots((\Sigma_{|x_1})_{|x_2})\ldots)_{|x_n}$ est la formule obtenue par l'application successive de la simplification de Σ par l'ensemble des littéraux de $\mathcal{A}$.

Exemple 10

Soient Σ la `CNF` de l'exemple 9 et $\mathcal{A} = \{a, b\}$, nous obtenons $\Sigma_{|\mathcal{A}} = \{(c \vee \neg d), (d)\}$.

Ici se termine la présentation de la logique propositionnelle classique. Dans la section suivante nous présentons une extension de cette dernière qui incorpore la notion de modalité.

2.3 La logique modale propositionnelle

La logique propositionnelle décrit des raisonnements sur des formules visant à décider si elles sont vraies ou fausses. Cependant, lorsque nous discutons au sujet d'une proposition, notre jugement est généralement plus nuancé. Par exemple, l'expression « il peut pleuvoir cet après-midi » représente une éventualité et l'expression « je dois me lever à 6h30 demain » représente une nécessité. Pour rendre compte du fait qu'une proposition peut aussi être possible, nécessaire, ... nous devons raffiner un peu plus la description sémantique et considérer ce qu'on appelle la sémantique des mondes possibles. Un monde reste caractérisé par les propositions qui y sont vraies ou fausses, mais un modèle comporte plusieurs mondes indiscernables (ce qui est modélisé par une notion d'accessibilité).

Dans cette section, nous introduisons brièvement la syntaxe et la sémantique de la logique modale propositionnelle. Concernant la sémantique, nous nous focaliserons sur la présentation de la sémantique de Kripke (1959). Pour plus de détails au sujet de la logique modale, le lecteur intéressé peut se référer à Chellas (1980).

2.3.1 Syntaxe

La logique modale propositionnelle peut être vue comme une extension de la logique propositionnelle classique où nous considérons une modalité de nécessité $\Box$. Intuitivement, si α est une formule alors $\Box\alpha$ exprime le fait que α est nécessairement vraie. De manière duale, la formule $\neg\Box\neg\alpha$, qui est aussi notée $\Diamond\alpha$, représente le fait que la formule α n'est pas nécessairement fausse et donc qu'elle est possiblement vraie.

Définition 37 (Formule modale propositionnelle)

Soit PS un ensemble fini de variables propositionnelles. L'ensemble des formules bien formées de $\mathcal{L}$ est le plus petit ensemble tel que :

- tout élément de $PS \cup \{\bot, \top\}$ est élément de $\mathcal{L}$;
- si $\alpha \in \mathcal{L}$ alors $(\neg\alpha), (\Box\alpha)$ et $(\Diamond\alpha)$ sont des éléments de $\mathcal{L}$;
- si α et β sont des éléments de $\mathcal{L}$ alors $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$, $(\alpha \Leftrightarrow \beta)$ et $(\alpha \oplus \beta)$ sont éléments de $\mathcal{L}$.

Comme pour la logique propositionnelle classique, il est toujours possible de représenter n'importe quelle formule de $\mathcal{L}$ sous forme normale négative (`NNF`)[2] en utilisant les règles de réécriture suivantes :

2. Ici la notion de `NNF` est légèrement différente de celle présentée dans le cas classique, nous avons des formules modales construites seulement à partir des connecteurs $\wedge$, $\vee$, des modalités $\Box$, $\Diamond$, des constantes $\bot$, $\top$ et des littéraux.

— $(\phi \Rightarrow \psi) ::= (\neg\phi \vee \psi)$
— $\neg(\phi \vee \psi) ::= (\neg\phi \wedge \neg\psi)$
— $\neg(\phi \wedge \psi) ::= (\neg\phi \vee \neg\psi)$
— $\neg\neg\phi ::= \phi$
— $\neg\Box\phi ::= \Diamond\neg\phi$
— $\neg\Diamond\phi ::= \Box\neg\phi$
— $(\phi \Leftrightarrow \psi) ::= (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
— $(\phi \oplus \psi) ::= \neg(\phi \Leftrightarrow \psi)$

Dans le reste de ce document, à part si cela est précisé, nous supposons que les formules de $\mathcal{L}$ sont représentées en `NNF`. Dans la section suivante, nous nous intéressons à la manière d'interpréter sémantiquement les connecteurs modaux.

2.3.2 Sémantique

Intuitivement, nous aimerions qu'un modèle soit un ensemble de mondes, et qu'une interprétation donne pour chaque symbole propositionnel les mondes où il est interprété à vrai (dans les autres, il vaut faux). Nous pouvons donc faire la distinction entre vrai et nécessairement vrai : une proposition est nécessairement vraie dans un modèle si elle est vraie dans tous les mondes de ce modèle. Ce cas particulier de la définition générale est appelé *modèle universel*. Cependant cette définition n'est clairement pas satisfaisante, puisque si $\Box\alpha$ est vraie dans un monde, alors il en va de même pour $\Box\Box\alpha$, $\Box\Box\Box\alpha$, ... De plus, soit $\Box\alpha$ est nécessaire dans ce modèle, soit $\neg\Box\alpha$ est nécessaire dans ce modèle. Le modèle se comporte ainsi pour les formules sous la portée de la modalité $\Box$ comme un modèle classique pour les formules non modales.

Pour dépasser cette limitation, il est nécessaire de considérer une relation d'accessibilité entre les mondes : un monde n'a d'information que sur certaines parties du modèle, les mondes qui lui sont accessibles. L'idée est que d'un monde, il est uniquement possible de voir les mondes qui sont directement accessibles. Grâce à cela, il est possible de faire une différence entre nécessité dans un monde et validité dans le modèle. La notion de modèle que nous allons définir maintenant est due à Kripke (1959), elle est appelée *modèle standard*.

Définition 38 (Structure de Kripke)

Soit PS un ensemble fini de variables propositionnelles. Une structure de Kripke est un triplet $\mathcal{K} = \langle W, R, V \rangle$, avec :

— W est un ensemble non vide de mondes possibles ;
— $R \subseteq W \times W$ est une relation d'accessibilité entre les mondes de W ;
— $V : PS \to 2^W$ est une application de PS dans 2^W qui au symbole propositionnel p_i fait correspondre le sous-ensemble $V(p_i)$ de W (l'ensemble des mondes où p_i est évalué à vrai).

Contrairement à l'évaluation d'une interprétation dans le cadre de la logique propositionnelle, ici il est nécessaire de définir la validité dans un monde du modèle. Une structure de Kripke pointée est définie comme un couple $\langle \mathcal{K}, w \rangle$, où $\mathcal{K}$ est une structure de Kripke et w est un monde possible de W. Dans la suite, lorsque nous utilisons le terme « structure de Kripke » nous faisons référence à une « structure de Kripke pointée ».

Définition 39 (Validité dans un monde)

Soit $\mathcal{K} = \langle \mathcal{K}, w \rangle$ une structure de Kripke avec $w \in W$, la relation $\models$ entre $\mathcal{K}$ et les formules de $\mathcal{L}$ est définie récursivement de la manière suivante :

- $\langle \mathcal{K}, w \rangle \models p$ si et seulement si $w \in V(p)$;
- $\langle \mathcal{K}, w \rangle \models \neg\phi$ si et seulement si $\langle \mathcal{K}, w \rangle \not\models \phi$;
- $\langle \mathcal{K}, w \rangle \models \phi_1 \wedge \phi_2$ si et seulement si $\langle \mathcal{K}, w \rangle \models \phi_1$ et $\langle \mathcal{K}, w \rangle \models \phi_2$;
- $\langle \mathcal{K}, w \rangle \models \phi_1 \vee \phi_2$ si et seulement si $\langle \mathcal{K}, w \rangle \models \phi_1$ ou $\langle \mathcal{K}, w \rangle \models \phi_2$;
- $\langle \mathcal{K}, w \rangle \models \Box\phi$ si et seulement si $\forall w'$ tel que $(w, w') \in R$ alors $\langle \mathcal{K}, w' \rangle \models \phi$;
- $\langle \mathcal{K}, w \rangle \models \Diamond\phi$ si et seulement si $\exists w'$ tel que $(w, w') \in R$ et $\langle \mathcal{K}, w' \rangle \models \phi$.

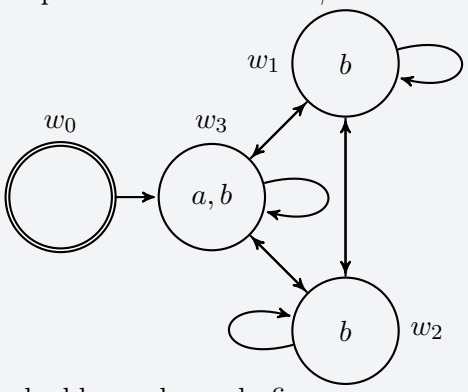
Définition 40 (Validité)

Une formule $\phi \in \mathcal{L}$ est valide (noté $\models \phi$) si et seulement si elle est satisfaite par toutes les structures de Kripke $\langle \mathcal{K}, w \rangle$. ϕ est satisfaisable si et seulement $\not\models \neg\phi$.

Exemple 11

Soit $\phi = \Diamond(a \wedge \Box b)$ une formule de $\mathcal{L}$, la structure de Kripke suivante satisfait ϕ :

- $W = \{w_0, w_1, w_2, w_3\}$,
- $R = \{\langle w_0, \{w_3\}\rangle, \langle w_1, \{w_1, w_2, w_3\}\rangle, \langle w_2, \{w_1, w_2, w_3\}\rangle, \langle w_3, \{w_1, w_2, w_3\}\rangle\}$,
- $V = \{\langle a, \{w_3\}\rangle, \langle b, \{w_1, w_2, w_3\}\rangle\}$.



Le monde pointée est ici w_0 et il est représenté par un double cercle sur la figure.

La notion d'accessibilité permet de représenter le fait qu'une formule peut être nécessaire dans un monde sans être valide dans tous. En revanche, une formule valide dans le modèle est

nécessaire dans tous ses mondes. Certains schémas de formules sont valides quelle que soit la structure de Kripke utilisée.

Définition 41 (Axiomes de la logique K)

$$\Diamond\phi \Rightarrow \neg\Box\neg\phi \qquad (\text{def}_\Diamond)$$
$$\Box\top \qquad (\text{N})$$
$$\Box(\phi \Rightarrow \psi) \Rightarrow (\Box\phi \Rightarrow \Box\psi) \qquad (\text{K})$$

Donc dans tous les modèles standard ($\text{def}_\Diamond$), (N) et (K) sont valides. C'est pour cette raison que nous nommons ces structures des K-modèles. L'ensemble des formules valides dans tous les modèles standard s'appelle la logique K. Ce n'est pas la seule logique possible, puisqu'il est possible de se contraindre à une partie des modèles standard en choisissant une classe de modèles.

Définition 42 (Validité dans une classe de modèles)

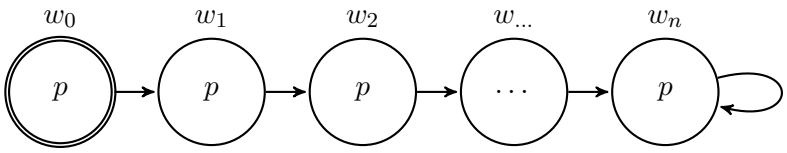
Soit α une formule de $\mathcal{L}$, α est valide dans une classe C de modèles (on écrit $\models^C \alpha$) si et seulement si α est valide dans tout modèle de C.

Il est possible de définir une logique modale comme l'ensemble des formules satisfaites par une certaine classe de modèles. Il y a d'autres façons de définir une logique modale (par ex. axiomatique). Une logique modale qui peut être caractérisée par une classe de modèles standard est appelée une logique modale normale. Une façon intéressante de constituer une famille de modèles modaux est d'ajouter des contraintes sur la relation d'accessibilité. Nous nommons *cadre* (*frame* en anglais) l'ensemble des règles qui conditionnent le modèle. Une façon simple de définir une classe de modèles consiste à spécifier une propriété du cadre sur lequel reposent les modèles. C'est déjà ce que nous avons effectué implicitement en considérant la classe des modèles universels : ce sont les modèles dont le cadre comporte une relation universelle. Le tableau 2.1 présente les principaux schémas d'axiome permettant de fixer certaines propriétés du cadre.

Axiome	**Condition sur $\mathcal{K}$**	**Contraintes sur les modèles**	**Schéma**
(K)	aucune		$\Box(\phi \Rightarrow \psi) \Rightarrow (\Box\phi \Rightarrow \Box\psi)$
(T)	réflexivité	$\forall w.R(w,w)$	$\Box\phi \Rightarrow \phi$
(B)	symétrie	$\forall w_1.\forall w_2.(R(w_1,w_2) \Rightarrow R(w_2,w_1))$	$\phi \Rightarrow \Box\Diamond\phi$
(D)	sérialité	$\forall w_1.\exists w_2.R(w_1,w_2)$	$\Box\phi \Rightarrow \Diamond\phi$
(4)	transitivité	$\forall w_1.\forall w_2.\forall w_3.((R(w_1,w_2) \wedge R(w_2,w_3)) \Rightarrow R(w_1,w_3))$	$\Box\phi \Rightarrow \Box\Box\phi$
(5)	euclidiannité	$\forall w_1.\forall w_2.\forall w_3.((R(w_1,w_2) \wedge R(w_1,w_3)) \Rightarrow R(w_2,w_3))$	$\Diamond\phi \Rightarrow \Box\Diamond\phi$

TABLE 2.1 – Schéma d'axiomes avec leurs propriétés structurelles.

Ces correspondances ont été proposées par Kripke (1959). Notons qu'une formule peut être satisfaite par une structure de Kripke qui ne satisfait pas le cadre. Considérons l'exemple suivant : soit $\phi = \Box p$ une formule de $\mathcal{L}$, nous souhaitons exhiber une structure de Kripke qui satisfait ϕ dans le cadre où les axiomes (T) et (4) sont considérés. L'axiome (4) impose que $\Box\phi \Rightarrow \Box\Box\phi$ tandis que l'axiome (T) impose que $\Box\phi \Rightarrow \phi$. Considérons la structure de Kripke suivante :



Nous pouvons observer que la structure de Kripke satisfait ϕ et respecte les axiomes du cadre. Cependant, elle n'est ni transitive ni réflexive. En fait, la notion de *bisimulation* entre modèles permet toujours d'exhiber une structure de Kripke « équivalente » qui satisfait les contraintes sur la structure du modèle imposées par un ensemble d'axiomes :

Définition 43 (Bisimulation (Blackburn et al. 2006))

Soient $M = \langle W, R, V\rangle$ et $N = \langle W', R', V'\rangle$ deux structures de Kripke et $(s,t) \in W \times W'$. Une *bisimulation* Z est une relation sur $W \times W'$. Z connecte s et t par *bisimulation*, noté sZt, si nous avons :

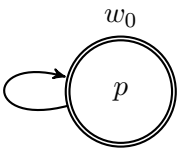
Invariance $V(s) = V'(t)$ (les deux mondes ont la même valuation) ;

Zig si $s' \in W$, $(s,s') \in R$, alors il existe $t' \in W'$ avec $(t,t') \in R'$ et $s'Zt'$;

Zag si $t' \in W'$, $(t,t') \in R'$, alors il existe $s' \in W$ avec $(s,s') \in R$ et $s'Zt'$.

Dans ce cas Z est une *bisimulation*, notée $M, s \leftrightarrow N, t$, de M et N qui connecte s et t.

La structure suivante, réflexive et transitve, est bisimilaire à la structure précédente.



Par conséquent, si nous recherchons une structure de Kripke qui satisfait les axiomes (K) et (T), il est suffisant de considérer uniquement les structures réflexives (nous avons vu que l'axiome (K) est toujours satisfait). Dans la suite, nous appelons KT-modèle un K-modèle réflexif, KD-modèle un K-modèle sériel et K5-modèle un K-modèle euclidien. En fait, la combinaison des différents axiomes donne 15 types de modèles différents présentés dans le tableau 2.2.

Pour chacune des logiques introduites dans le tableau 2.2, la notion de validité peut être définie de la manière suivante :

Type	**Propriétés**	**Type**	**Propriétés**
K	$\varnothing$	(S4) KT4 = KDT4	réflexive et transitive
KB	symétrique	KD4	sérielle et transitive
KT = KDT	réflexive	KD	sérielle
K4	transitive	KDB	sérielle et symétrique
KBT = KBDT	symétrique et réflexive	K45	transitive et euclidienne
K5	euclidien	KD5	sérielle et euclidienne
KB4 = KB5 = KB45	symétrique et transitive	KD45	sérielle, transitive et euclidienne
(S5) KT5 = KBD4 = KBD5 = KBT4 = KBT5 = KDT5 = KT45 = KBD45 = KBT45 = KDT45 = KBDT4 = KBDT5 = KBDT45			équivalence

TABLE 2.2 – Les différents types de structures de Kripke.

Définition 44 ($\star$-Validité)

Soit $\star$ une des logiques du tableau 2.2, une formule $\phi \in \mathcal{L}$ est $\star$-valide (noté $\models_\star \phi$) si et seulement si elle est satisfaite par tous les $\star$-modèles $\langle \mathcal{K}, w \rangle$. ϕ est $\star$-satisfaisable si et seulement si $\not\models_\star \neg\phi$. Un $\star$-modèle qui satisfait une formule ϕ est appelé un '$\star$-modèle de ϕ'.

Il est intéressant de noter que s'il existe un modèle pour une certaine logique, alors c'est aussi un modèle pour une logique moins contrainte. Par exemple, si la formule ϕ est KT5-satisfaisable, alors ϕ est aussi K-satisfiable, KT-satisfiable, S4-satisfaisable, *etc.* Inversement, si une formule ϕ est K-insatisfaisable, alors elle est aussi KT-insatisfiable, K4-insatisfaisable, S4-insatisfaisable, *etc.*

De manière générale, toutes les logiques ne sont pas considérées. Dans (Kripke 1959, Ladner 1977) les auteurs mettent en avant que les logiques modales normales basées sur les axiomes K + $\{(D), (T), (B), (4), (5)\}$ ont de bonnes propriétés. D'un point de vue calculatoire, Ladner (1977) a montré que le problème de la satisfaisabilité dans la logique modale K est `PSPACE`-complet et qu'il est `NP`-complet dans la logique modale S5. Halpern et Rêgo (2007) montrent que ce qui fait basculer la complexité de `NP` à `PSPACE` est la présence ou non de l'axiome (5) (*negative introspection*). Ils montrent que le problème de la satisfaisabilité pour les logiques dont le cadre contient l'axiome (5) est `NP`-complet, tandis qu'il est `PSPACE`-complet pour les autres.

Nous venons de mettre en avant le fait qu'il est possible de restreindre la recherche d'un modèle pour une formule donnée aux structures de Kripke qui satisfont une certaine structure. Cependant, puisque toutes ces logiques rentrent dans le cadre de la théorie des modèles finis (Urquhart 1981), il est possible d'aller encore plus loin et de ne considérer que les structures de Kripke d'une certaine taille. C'est-à-dire qu'il existe pour chaque $\star$-modèle un majorant, $UB(\phi)$ pour une formule ϕ, du nombre de mondes à considérer. Cette borne est calculée en fonction de la logique par rapport à la profondeur modale et le nombre d'atomes de la formule considérée. La profondeur modale d'une formule est définie formellement de la manière suivante :

Définition 45 (Profondeur modale)

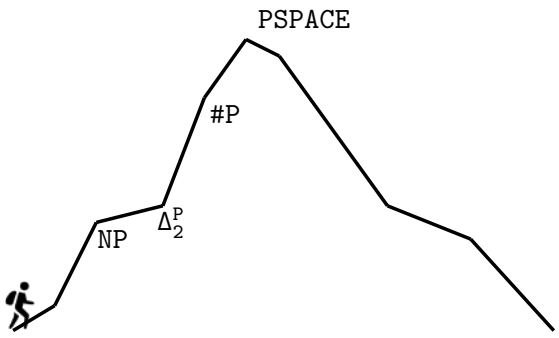
La profondeur modale d'une formule ϕ de $\mathcal{L}$, notée $\text{depth}(\phi)$, est calculée de la manière suivante :

$$\text{depth}(p) = \text{depth}(\top) = \text{depth}(\bot) = 0$$
$$\text{depth}(\neg\phi) = \text{depth}(\phi)$$
$$\text{depth}(\phi \odot \psi) = \max(\text{depth}(\phi), \text{depth}(\psi)) \ (\odot \in \{\wedge, \vee, \Leftrightarrow, \Rightarrow, \oplus\})$$
$$\text{depth}(\Box\phi) = \text{depth}(\Diamond\phi) = 1 + \text{depth}(\phi)$$

Le nombre d'atomes $\texttt{Atom}(\phi)$ d'une formule ϕ de $\mathcal{L}$ est le nombre de sous-formules dont le foncteur principal est une modalité. Par exemple $p, \Box(p_1 \vee p_2)$ et $\Box(\Box p_1 \vee p_2)$ sont des atomes. Dans leur article, Sebastiani et McAllester (1997) établissent qu'il est possible de borner le nombre de mondes de la structure de Kripke à $\texttt{Atom}(\phi)^{\text{depth}(\phi)}$. Nguyen (1999) quant à lui établit qu'il est possible de se limiter aux structures de Kripke de taille au plus $|\phi|^{2\times|\phi|+\text{depth}(\phi)}$, où $|\phi|$ est le nombre de sous formules, pour les logiques modales $\{K, K4, KD4, KD, KT, KB, KDB, KB, S4\}$. En ce qui concerne les logiques modales NP (avec l'axiome 5), Halpern et Rêgo (2007) établissent qu'il est possible de se limiter aux structures de Kripke d'une taille au maximum $|\phi|$. Ladner (1977) établit ce majorant à $\text{nm}(\phi)$, où nm est le nombre de modalités de la formule.

2.4 Conclusion

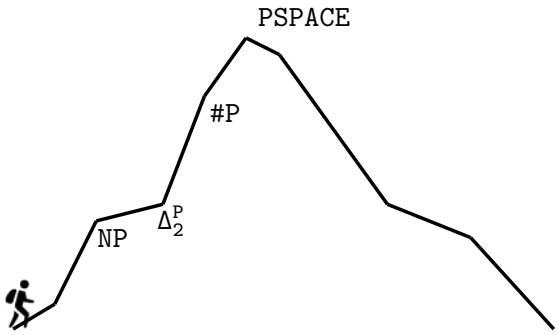
Nous sommes maintenant fin prêts pour démarrer notre voyage. Jusqu'ici nous n'avons pas encore réellement établi notre cap. La seule chose que nous savons c'est que nous souhaitons aller au delà de NP. Ainsi, il semble tout à fait naturel de commencer notre promenade par le point de départ où il s'agit d'étudier la résolution du problème de satisfaisabilité en logique propositionnelle classique.



Chapitre 3

Contributions à la résolution de SAT en parallèle

Sommaire



Nous voilà fin prêts pour commencer notre voyage au-delà de NP. La première étape dans ce voyage concerne la résolution de problèmes NP complets. Pour cela, nous allons nous intéresser au problème NP de référence : le problème SAT (Cook 1976). En plus d'être un problème central en théorie de la complexité, il est aussi très utilisé en intelligence artificielle et en démonstration automatique. En effet, cela fait de nombreuses années, et depuis l'avènement des solveurs CDCL (Marques-Silva et Sakallah 1996a, Moskewicz et al. 2001a), que le problème SAT est utilisé dans

la résolution de problèmes extrêmement divers (vérification formelle bornée (Biere et al. 1999a), gestion des dépendances dans les paquets d'une distribution linux (Argelich et al. 2010) ou encore dans les plugins du logiciel Eclipse (Le Berre et Rapicault 2009), planification (Kautz et Selman 1996), ...). Pour autant, de nombreux challenges existent encore dans la communauté, ce qui demande toujours l'élaboration de nouvelles techniques.

Avec l'avènement du parallélisme massif, il est à présent possible de mettre en place des méthodes efficaces pour la résolution de problèmes pratiques qui nécessitent l'utilisation de systèmes « intelligents ». En effet, nous voyons depuis quelque temps un engouement important autour de l'intelligence artificielle qui est porté par une méthodologie d'apprentissage automatique appelée apprentissage profond. Bien que les algorithmes d'apprentissages aient évolué, une des raisons de ce succès vient du fait qu'il y a de plus en plus de données utilisables et surtout que les machines actuelles sont en capacité de les traiter rapidement. Par exemple, la version d'AlphaGo utilisée par *Google Deepmind* pour battre Lee Sedol au jeu de Go a nécessité l'utilisation de 1 202 CPUs et de 176 GPUs (Silver et al. 2016b).

Une question que nous sommes en droit de nous poser est : pouvons-nous exploiter cette puissance de calcul pour aider à la résolution du problème SAT ? La réponse à cette question n'est pas si simple. Les solveurs SAT ont été pensés et optimisés pour fonctionner séquentiellement. Ainsi, il est nécessaire de repenser entièrement l'architecture des solveurs afin de pouvoir bénéficier de l'apport des GPUs. Quelques travaux vont dans cette direction (Beckers et al. 2011, Palù et al. 2015), mais les solveurs correspondants ne sont pas aussi efficaces que les solveurs multi-cœurs de l'état de l'art.

C'est dans ce contexte que s'inscrivent les travaux que j'ai réalisés en collaboration avec Gilles Audemard. Nous nous sommes principalement intéressés à l'utilisation de la parallélisation de `SAT` en utilisant des architectures à mémoire distribuée. L'avantage d'une telle architecture est qu'il est possible d'utiliser facilement les solveurs séquentiels de l'état de l'art. Cependant, la parallélisation n'en est pas pour autant triviale. Une approche qui consisterait à exécuter la même approche sur un millier de machines n'aurait aucune chance de fonctionner. Une autre approche pourrait être de lancer de nombreux solveurs différents en espérant que l'un d'entre eux ait de la chance. Cette approche, bien que plus intéressante que la précédente, n'est pas non plus totalement satisfaisante puisqu'il faudrait idéalement disposer d'autant de méthodes différentes que d'unités de calcul disponibles. De plus, ce type d'approche ne peut pas produire un solveur séquentiel qui est sensiblement plus efficace que le meilleur solveur que nous avons à notre disposition aujourd'hui.

Lorsque nous observons la nature, nous nous rendons rapidement compte que, pour mener à bien une tâche en collectivité, la communication est un outil très important. En effet, lorsque nous étudions certains insectes, tels que les fourmis ou les abeilles, nous sommes subjugués par leurs capacités. C'est dans ce cadre, d'échange d'informations et de partage de travail, que s'intègrent les travaux présentés dans ce chapitre. Nous verrons comment il est possible d'élaborer une approche participative pour la génération d'un arbre de guidage permettant un partage du travail entre les différentes unités de calcul. Nous nous intéresserons aussi aux problèmes qui peuvent survenir lorsqu'il est nécessaire d'échanger beaucoup d'informations parce que les solveurs ne travaillent pas sur la même machine.

Ce chapitre est divisé en trois sections. Dans la première section, nous présentons l'algorithme `CDCL` ainsi que les heuristiques qui le rendent efficace en pratique. Ensuite, nous introduisons nos contributions à la résolution du problème `SAT` dans le cadre de l'utilisation d'une architecture à mémoire distribuée. Plus précisément, nous présentons dans la seconde section le solveur

AmPharoS qui est une approche de type « diviser pour régner ». Dans la troisième section, nous présentons nos travaux autour de l'échange d'informations dans le cadre d'une version distribuée du solveur parallèle Syrup. Nous terminons ce chapitre par une conclusion et en donnant quelques perspectives.

3.1 Résolution séquentielle de SAT

Dans le cadre de ce chapitre, nous nous intéressons uniquement à la résolution du problème SAT avec des algorithmes complets. Ces algorithmes permettent, en un temps fini, de déterminer la satisfaisabilité de n'importe quelle formule CNF. Ils s'appuient généralement sur le parcours en profondeur d'un arbre de recherche, où chaque nœud correspond à l'assignation d'une variable et chaque chemin correspond à une interprétation partielle. L'objectif est alors de déterminer un chemin de la racine à une feuille - lequel représente une interprétation des variables de la formule - qui satisfait l'ensemble des contraintes du problème. Afin de ne pas explorer l'intégralité de cet arbre, la plupart des approches utilisent des méthodes de propagation de contraintes dans le but d'élaguer l'arbre de recherche en coupant les branches ne pouvant conduire à une solution.

Après avoir introduit la méthode DPLL ainsi que la propagation unitaire et quelques heuristiques de branchement, nous présentons les solveurs SAT de type CDCL et les différents composants participant à leur efficacité (redémarrage, heuristique de choix de variable VSIDS et apprentissage). Le but n'étant pas de réaliser un état de l'art complet des solveurs SAT, le lecteur intéressé pourra se référencer à (Biere et al. 2009) s'il souhaite plus de détails sur le sujet.

3.1.1 La procédure de Davis - Putnam - Logemann - Loveland

La méthode DPLL (Davis et al. 1962), décrit par l'algorithme 3.1, est en fait un algorithme de recherche arborescent de type « *depth-first search* » (recherche en profondeur d'abord) avec retour arrière. Étant donnée une formule Σ, cette procédure consiste à choisir un littéral ℓ de Σ (ligne 4) et à décomposer la formule Σ en deux sous-formules $\Sigma \wedge \ell$ et $\Sigma \wedge \neg\ell$. Ce principe, appelé *séparation*, est basé sur le fait que Σ est cohérente si et seulement si $\Sigma \wedge \ell$ ou $\Sigma \wedge \neg\ell$ est cohérente. Une fois cette décomposition matérialisée, la procédure DPLL consiste à tester la validité de la première sous-formule, si elle est cohérente alors le problème est cohérent, sinon la seconde formule est testée (ligne 5). Afin d'éviter l'exploration inutile de branches de l'arbre de recherche, certaines simplifications sont opérées. Un autre point crucial quant à l'efficacité de la procédure DPLL est le choix de la variable pour la règle de séparation. Cette variable, communément appelée variable (ou littéral) de décision, est choisie de manière heuristique.

Algorithme 3.1 : DPLL

Données : Σ un ensemble de clauses
Résultat : vrai si la formule est cohérente, faux sinon

```
1 Début
2 |  Σ ⟵ SIMPLIFICATION(Σ);
3 |  si (Σ = ∅) ou (⊥ ∈ Σ) alors retourner (Σ = ∅) ou (⊥ ∉ Σ);
4 |  ℓ ⟵ HeuristiqueDeBranchement (Σ);
5 |  retourner (DPLL(Σ ∧ ℓ) ou DPLL(Σ ∧ ¬ℓ))
6 Fin
```

Dans la suite, nous présentons la propagation unitaire (cf 3.1.1.1), qui est certainement la méthode de simplification la plus utilisée, et certaines heuristiques de choix de variables (cf 3.1.1.2).

3.1.1.1 La propagation unitaire

La propagation unitaire, notée `bcp` (pour *Boolean Constraint Propagation*), représente la forme de simplification la plus utilisée et sans doute la plus utile des approches de type `DPLL`. Ce processus s'appuie sur le fait que les seules interprétations potentiellement capables de satisfaire une formule doivent satisfaire les littéraux appartenant aux clauses unitaires. Simplifier (ou conditionner) par un littéral unitaire consiste donc à supprimer de l'ensemble de clauses, toutes celles contenant le littéral unitaire et à supprimer toutes les occurrences du littéral complémentaire, c'est-à-dire « raccourcir » les clauses contenant le littéral complémentaire. La propagation unitaire est l'application répétée de cette simplification jusqu'à ce que la base de clauses ne contienne plus de clauses unitaires (point fixe) ou jusqu'à l'obtention d'une clause vide (contradiction).

L'application de la propagation unitaire au sein de l'algorithme `DPLL` implique une modification au niveau de la gestion des retours arrière (*backtracks*). En effet, lorsqu'un *backtrack* est effectué, il est nécessaire de considérer en plus du dernier point de choix les propagations unitaires qu'il a générées. Pour effectuer cela il est nécessaire d'utiliser une notion de pile de propagations où chaque élément de cette pile représente une séquence de décisions-propagations. Étant donnée une formule Σ, $\langle x_1, x_2, \ldots, x_n \rangle$ représente la séquence de propagations $\mathcal{P}$ obtenue à partir de Σ telle que $\forall x_i \in \mathcal{P}$ la clause unitaire (x_i) appartient à $\Sigma_{|\{x_1, x_2, \ldots, x_{i-1}\}}$. Une séquence de décisions-propagations $\mathcal{S} = \langle (x), x_1, x_2, \ldots, x_n \rangle$ représente la séquence de décisions-propagations obtenue par l'application de la propagation unitaire sur la formule $(\Sigma \wedge x)$ telle que $\langle x_1, x_2, \ldots, x_n \rangle$ est une séquence de propagations obtenue à partir de $(\Sigma \wedge x)$. La pile de propagations $\mathcal{H} = \{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$ obtenue par l'application de la propagation unitaire sur la formule Σ en fonction de la séquence $\delta = \{x_1, x_2, \ldots, x_n\}$ de décisions est définie récursivement de la manière suivante :

— $\mathcal{S}_0$ est une séquence de propagations obtenue à partir de Σ ;

— $\mathcal{S}_1$ est une séquence de décisions-propagations obtenue à partir de $\texttt{bcp}(\Sigma) \wedge x_1$;

— $\forall 1 < i \leq n, \mathcal{S}_i = (\Sigma \wedge \texttt{bcp}(\bigwedge_{j=1}^{i-1} x_j)) \wedge x_i$.

Remarque 1

Lorsqu'une séquence de décisions conduit à une contradiction, le dernier niveau de la pile de propagation contient les deux littéraux complémentaires ayant conduit à celle-ci.

Dans la suite de ce manuscrit, nous ne faisons plus la distinction entre interprétation partielle issue de la propagation unitaire et la séquence de décisions-propagations associée.

À partir de la notion de pile de propagations, nous pouvons introduire pour un littéral ℓ de cette pile les notions de niveau de propagation ($niv(\ell)$), de clause raison ($\overrightarrow{cla}(\ell)$) et d'explication ($exp(\ell)$).

Définition 46 (Niveau de propagation, clause raison et explication)

Soient Σ une formule sous forme CNF, $\mathcal{H} = \{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$ une pile de propagations obtenue à partir de Σ en appliquant la séquence de décisions $\delta = [x_1, x_2, \ldots, x_n]$, $\mathcal{I}$ l'interprétation partielle associée et ℓ un littéral de Σ. Nous avons :

- Le niveau de ℓ, noté $niv(\ell)$, est tel que si $\exists \mathcal{S}_i \in \mathcal{H}$ avec $\ell \in \mathcal{S}_i$ alors $niv(\ell) = i$, $niv(\ell) = \infty$ sinon ;
- Une raison de ℓ, notée $\overrightarrow{cla}(\ell)$, est telle que si $\ell \in \delta$ alors $\overrightarrow{cla}(\ell) = \bot$, sinon $\overrightarrow{cla}(\ell) \in \Sigma$ telle que $\ell \in \overrightarrow{cla}(\ell)$ et $\forall y \in \overrightarrow{cla}(\ell) \setminus \{\ell\}, \mathcal{I}(y) = \bot$ et y précède ℓ dans l'interprétation $\mathcal{I}$. Il est possible d'associer plusieurs clauses raison à un littéral propagé. Néanmoins, il est usuel de n'en considérer qu'une seule ;
- Une explication de ℓ, notée $exp(\ell)$, est telle que si $\ell \in \delta$ alors $exp(\ell) = \varnothing$, sinon $exp(\ell) = \{\tilde{x}$ tel que $x \in \overrightarrow{cla}(\ell)$ avec $\overrightarrow{cla}(\ell)$ une clause raison de $\ell\}$. Comme pour les clauses raison il n'y a pas unicité de l'explication de la propagation d'un littéral.

Pour terminer, il est important de souligner que, en plus d'être fondamentale pour l'algorithme DPLL et d'être implémentée dans tous les solveurs de ce type, la propagation unitaire peut également être utilisée comme méthode de prétraitement. L'application de la propagation unitaire comme prétraitement permet en pratique d'extraire des instances certaines informations importantes comme les littéraux impliqués, équivalents ou encore unitaires (Le Berre 2001, Novikov 2003). De plus, elle permet aussi de déduire des formules comme des portes logiques ou des sous-clauses (Génisson et Siegel 1994, Darras et al. 2005, Grégoire et al. 2005). La collecte de ces informations est souvent obtenue par l'application d'une forme affaiblie de la conséquence logique. Cet affaiblissement se définit formellement de la manière suivante.

Définition 47 (Conséquence logique restreinte à la propagation unitaire)

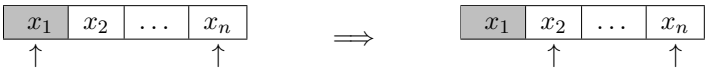
Soient Σ et Σ' deux formules sous formes CNF. Σ' est une conséquence logique restreinte à la propagation unitaire de Σ, notée $\Sigma \models^* \Sigma'$ si et seulement si, pour toute clause $\alpha \in \Sigma'$, $\texttt{bcp}(\Sigma \wedge \overline{\alpha})$ contient la clause vide.

Comme nous l'avons souligné, la propagation unitaire est un processus fondamental de l'algorithme DPLL. Généralement, le temps consacré à celui-ci est de l'ordre de 90% du temps CPU utile à la résolution d'une instance. Afin d'améliorer l'efficacité de la propagation unitaire, de nombreux progrès algorithmiques ont été effectués ces dernières années. Le plus important est incontestablement la méthode proposée par Zhang et Malik (2002). Les auteurs introduisent une structure de données dite « paresseuse » qui permet un traitement de la propagation unitaire d'une complexité moyenne sous-linéaire (bien que sa complexité dans le pire des cas soit quadratique dans le nombre de clauses). L'idée sous-jacente est d'avoir, pour chaque clause α de la formule, deux littéraux x et y appartenant à α non affectés à faux par l'interprétation courante. Ces deux littéraux sont garants du fait que la clause n'est ni unitaire ni falsifiée. Pour déterminer

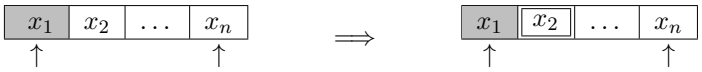
si une clause est unitaire ou non, il suffit de vérifier s'il y a plus du littéral non affecté à faux dans celle-ci. Lorsque les deux littéraux x et y de α ont été désignés pour surveiller la clause α il est nécessaire de le leur indiquer. Pour cela, chaque littéral possède une liste de clauses qu'il doit surveiller. De cette manière, lorsqu'un littéral x est propagé à vrai, il suffit de regarder la liste des clauses surveillées par le littéral complémentaire $\tilde{x}$ et à chercher une autre « sentinelle » pour chacune d'elles, d'où le nom de la structure : « *watched two literals* ».

À présent étudions, les différents cas pouvant survenir lorsqu'un littéral est propagé à faux. Pour cela, considérons le cas d'une clause α surveillée par deux littéraux x_1 et x_n. En grisant, en entourant et en désignant respectivement les littéraux falsifiés, satisfaits et marqués, les différents cas rencontrés lors de la recherche d'un nouveau littéral sentinelle, après l'affectation d'un littéral $\neg x_1$, sont les suivants :

— commençons par considérer le cas où il existe $y \in \alpha$ tel que $y \neq x_1$, $y \neq x_n$ et y est non affecté, par l'interprétation courante, alors y peut surveiller la clause α ($y = x_2$ sur la figure) ;



— supposons à présent que lors du processus de recherche d'un nouveau littéral non affecté la procédure « s'aperçoit » qu'il existe $y \in \alpha$ tel que y est satisfait par l'interprétation courante ($y = x_2$ sur la figure). Dans ce cas, il n'est pas utile de changer le rôle de sentinelle pour x_1. En effet, puisque x_1 est affecté après y, lorsque y sera de nouveau « libre » x_1 le sera aussi et par conséquent il sera de nouveau à même de remplir son rôle de sentinelle ;



— enfin, considérons le cas où il n'y a aucun littéral différent de x_n non affecté à faux. Dans cette configuration, deux situations se présentent :

1. x_n est non affecté et il est alors propagé à vrai ;

2. x_n est affecté à faux et alors la clause est falsifiée par l'interprétation courante. Dans ce cas, la propagation unitaire conduit à un conflit et un « *backtrack* » est nécessaire.

Pour les mêmes raisons que celles énoncées dans le cas où un littéral est satisfait, il n'est pas nécessaire de changer les littéraux sentinelles la clause.

Nous pouvons observer que, grâce à cette structure, lors de l'affectation d'une variable, seul un sous-ensemble des clauses contenant une occurrence de cette variable est parcouru. Pourtant ce traitement partiel est suffisant pour déclencher les propagations unitaires durant le parcours de l'espace de recherche. Ceci explique la rapidité de traitement des affectations, et en particulier de la propagation unitaire. De plus, il est important de noter qu'aucune mise à jour n'est nécessaire lors des retours arrière. Il est à noter que cette notion de « *watched literals* » peut facilement être étendue à d'autres situations où il est nécessaire de vérifier à chaque instant une condition. Par exemple, elle peut être utilisée afin de détecter efficacement lorsqu'une clause devient falsifiée lorsqu'une seule sentinelle lui est assigné.

3.1.1.2 Heuristiques de branchement

Comme nous l'avons déjà souligné, le choix de la prochaine variable à affecter est un critère déterminant pour l'efficacité d'un solveur SAT. Son incidence sur la taille de l'arbre et par conséquent les temps d'exécution est considérable. En effet, la taille de l'arbre de recherche exploré peut varier de manière exponentielle en fonction de l'ordre avec lequel les variables sont affectées (Li et Anbulagan 1997). Cependant, sélectionner à chaque point de choix la variable qui permet de conduire à l'obtention d'un arbre de taille minimale est un problème NP-difficile (Liberatore 2000). Au vu de la complexité théorique pour l'obtention de la variable de branchement optimale, il apparaît dès lors raisonnable d'estimer le plus précisément possible à l'aide d'une heuristique cette variable plutôt que de la calculer précisément. Ces dernières années beaucoup d'heuristiques de branchements différentes ont été proposées :

— les approches syntaxiques qui permettent d'estimer le nombre de propagations résultant de l'affectation d'une variable (Goldberg 1979, Jeroslow et Wang 1990) ou de décomposer la formule en différentes composantes connexes (Huang et Darwiche 2004) ;

— les approches prospectives (de type « look-ahead ») qui permettent de détecter de futures situations d'échec (Pretolani 1993, Freeman 1995, Li et Anbulagan 1997, Dequen et Dubois 2004) ;

— les approches rétrospectives (de type « look-back ») qui essayent d'apprendre à partir des situations d'échec (Brisoux et al. 1999, Zhang et al. 2001, Liang et al. 2017).

L'utilisation de structures paresseuses pour la gestion de la propagation unitaire a rendu l'utilisation d'heuristiques syntaxiques difficiles car elles requièrent une connaissance complète de l'instance après affectation(s). Seules les heuristiques sémantiques peuvent être appliquées avec les structures de données paresseuses, la structure du problème n'étant pas connue. Parmi ces heuristique, l'heuristique VSIDS (« *Variable State Independent Decaying Sum* »), proposée par Zhang et al. (2001), est l'heuristique de branchement la plus utilisée dans les implantations modernes de DPLL. Cette heuristique associe un compteur appelé *activité* à chaque variable. Lorsqu'un conflit survient, une analyse de conflits (cf. 3.1.2.1) permet de déterminer la raison du conflit et les variables en cause dans ce conflit voient leur activité augmentée. Au prochain branchement, la variable ayant la plus grande activité est alors choisie comme variable de décision. En effet, puisque celle-ci est la plus impliquée dans les conflits, elle est jugée fortement contrainte. De façon à être précis, il est important de signaler que l'activité des variables est divisée périodiquement par une certaine constante. Le fait de diminuer l'activité permet de privilégier les variables récemment intervenues dans les conflits.

Lorsqu'une variable est choisie comme point de choix il est nécessaire de lui associer une certaine valeur de vérité. À cause de sa complexité algorithmique, ce choix est effectué de manière heuristique (Eén et Sörenson 2004, Jeroslow et Wang 1990, Pipatsrisawat et Darwiche 2007, Biere 2009). L'heuristique la plus utilisée à l'heure actuelle dans les solveurs SAT est nommée ***progress saving*** (Pipatsrisawat et Darwiche 2007). Cette heuristique part de l'observation expérimentale que, dans le cadre d'une approche utilisant le *backtracking* les solveurs effectuaient beaucoup de travail redondant. En effet, lorsqu'un retour arrière est effectué, le travail accompli pour résoudre les sous-problèmes traversés avant d'atteindre le conflit est perdu. Pour éviter cela, les auteurs proposent de sauvegarder la dernière polarité obtenue pendant la recherche dans une interprétation complète notée $\mathcal{P}$. Ainsi, lorsqu'une nouvelle décision sera prise, la variable se verra attribuer la même polarité que précédemment (choisir x si $x \in \mathcal{P}$, $\neg x$ sinon). De cette manière, l'effort pour satisfaire un sous-problème déjà résolu auparavant sera moins important.

Le principal désavantage de cette approche est de ne pas assez diversifier la recherche. Afin de pallier ce problème, Biere (2009) propose d'« oublier » une partie de l'interprétation $\mathcal{P}$.

Une des caractéristiques de l'algorithme DPLL est qu'il remonte l'arbre de recherche jusqu'au nœud de décision précédant l'échec lorsqu'un conflit est atteint. Le fait d'effectuer un retour arrière chronologique peut dans certaines conditions être très problématique. En effet, la cause de l'apparition de l'échec peut être due à un autre point de choix, décidé plus haut dans l'arbre de recherche. Ne pas considérer la cause réelle de l'échec peut conduire à un phénomène nommé « *trashing* » et qui consiste à explorer de façon répétitive les mêmes sous-arbres. Afin d'y remédier un certain nombre d'approches tendent à analyser le conflit rencontré en prenant en compte les décisions à l'origine de celui-ci. Cette analyse a pour but d'extraire des informations de manière à effectuer un retour arrière non chronologique (*backjumping*) et ainsi éviter de redécouvrir les mêmes échecs.

Ce type d'approche est étudié et appliqué dans de nombreux domaines en intelligence artificielle (système de maintien de vérité ATMS (McAllester 1980, Stallman et Sussman 1977a), problèmes de satisfaction de contraintes CSP (Dechter 1990, Ginsberg 1993, Schiex et Verfaillie 1993b), ...). Ces techniques se distinguent essentiellement sur la manière dont les échecs sont analysés et les méthodes utilisées afin de ne plus se heurter aux mêmes situations par la suite. Dans le cadre de SAT, ce n'est qu'en 1996 et par l'intermédiaire de Marques-Silva et Sakallah (1996b) que l'analyse de conflits est introduite. Cette analyse de conflits permet d'extraire un terme $(x_1 \wedge x_2 \wedge \cdots \wedge x_n)$ signifiant que l'apparition du conflit est due à l'affectation conjointe des littéraux $x_1, x_2, \ldots, x_n$ à vrai (la manière d'extraire ces informations est présentée par la suite (voir 3.1.2.1)).

Même quand un retour arrière non chronologique est réalisé, il est possible qu'une même situation d'échec se répète dans le futur. Afin d'éviter cela, une approche consiste à ajouter une information sous la forme d'une clause à la formule. Cette clause, nommée « *nogood* », est obtenue par la négation du terme construit par analyse de conflits $(\neg x_1 \vee \neg x_2 \ldots \neg x_n)$. L'ajout de cette clause, connue sous le nom de *clause apprise* (*clause learning*), permet à chaque fois qu'une contradiction est détectée d'identifier et d'ajouter une clause impliquée par la formule. De plus, considérer cette clause pour la suite de la recherche permet à la propagation unitaire de découvrir de nouvelles implications lesquelles permettent d'éviter de considérer la même situation d'échec plusieurs fois.

C'est la combinaison de ces principes et leur intégration au sein de l'algorithme DPLL qui ont conduit à l'élaboration des solveurs SAT modernes (Marques-Silva et Sakallah 1996b, Zhang et al. 2001, Beame et al. 2004, Bayardo Jr. et Schrag 1997).

3.1.2 Algorithme CDCL

La procédure CDCL a été introduite par Marques-Silva et Sakallah (1996a) (et améliorée par Moskewicz et al. (2001a)). Elle constitue une extension de la procédure DPLL. Cette méthode tente de tirer parti de l'analyse de conflits afin d'apprendre de nouvelles clauses et effectuer des retour-arrières non chronologiques. L'algorithme CDCL a deux avantages majeurs par rapport à l'algorithme DPLL : (i) effectuer un *backjumping* permet de ne pas considérer une partie de l'arbre de recherche ne possédant pas de solution et (ii) l'apprentissage de clauses permet d'éviter de considérer des sous-arbres de recherche, lesquels ont été montrés sans solution lors d'une précédente analyse de conflits. L'algorithme CDCL ne pouvant pas être présenté facilement de manière récursive, une version itérative de celui-ci est donnée dans l'algorithme 3.2.

Algorithme 3.2 : Solveur CDCL

Données : Σ une formule sous CNF
Résultat : SAT or UNSAT

```
1  Δ ⟵ ∅;                                          /* ensemble de clauses apprises */
2  I_p ⟵ ∅;                                        /* interprétation partielle */
3  dl ⟵ 0;                                         /* niveau de décision */
4  tant que (true) faire
5      α ⟵ bcp(Σ ∪ Δ, I_p);
6      si (α = null) alors
7          x ⟵ heuristiqueDeBranchement(Σ ∪ Δ, I_p);
8          si (toutes les variables sont affectées) alors retourner SAT ;
9          ℓ ⟵affectePolarité(x);
10         dl ⟵ dl + 1;
11         I_p ⟵ I_p ∪ {(l^dl)};
12         si faireReduction() alors reductionClausesApprises (Δ);
13     sinon
14         β ⟵analyseConflit(Σ ∪ Δ, I_p, α);
15         bl ⟵calculRetourArrière(I_p, β);
16         si β = ⊥ alors retourner UNSAT ;
17         Δ ⟵ Δ ∪ β;
18         retourArrière(Σ ∪ Δ, I_p, bl);                  /* mise à jour de I_p */
19         si (faireRedémarrage()) alors redémarrage(I_p, dl);
```

Typiquement, le fonctionnement de l'algorithme CDCL consiste à réaliser une séquence de décisions suivie de propagations des littéraux unitaires, jusqu'à l'obtention d'un conflit. Chaque littéral choisi comme littéral de décision (ligne 7) est affecté suivant une certaine polarité (ligne 9) à un niveau de décision donné (ligne 11). Si tous les littéraux sont affectés, alors $\mathcal{I}_p$ est un modèle de Σ (ligne 8). À chaque fois qu'un conflit est atteint par propagation (lignes 13–19), une clause β est calculée en utilisant une méthode d'analyse de conflits donnée et un niveau de *backjump* est calculé en fonction de la clause β (lignes 14–15). À ce stade, il est possible de prouver l'incohérence (β est la clause vide) de la formule (ligne 16). Si ce n'est pas le cas, un retour-arrière est effectué et le niveau de décision devient égal au niveau de *backjump* (ligne 18). Ensuite, certains solveurs CDCL forcent le redémarrage et dans ce cas, un retour arrière est effectué au sommet de l'arbre de recherche (ligne 19). Finalement, la base de clauses apprises peut être réduite lorsque celle-ci est considérée comme trop volumineuse (ligne 12). Il est important de noter que pour concerver la compétude de la méthode il est nécessaire de conserver « suffisamment » de clauses apprises (Pipatsrisawat et Darwiche 2011). Dans la suite, nous donnons une brève description des différentes fonctions laissées en suspens lors de la présentation de l'algorithme.

3.1.2.1 Analyse de conflits

Les premiers travaux sur l'analyse de conflits ont été réalisés sur les algorithmes de résolution de réseaux de contraintes (Prosser 1993). Les interactions entre le monde des CSP et celui de SAT ont permis l'élaboration et le développement de ces techniques dans le cadre de SAT (Marques-Silva et Sakallah 1996b, Bayardo Jr. et Schrag 1997), qui, depuis, sont devenues incontournables.

Comme nous avons pu le souligner précédemment, le but de l'analyse de conflits est de déterminer un ensemble de littéraux responsables d'une situation d'échec. Une fois cet ensemble de littéraux localisé, une nouvelle clause est générée afin d'indiquer au solveur qu'il n'existe pas de solution dans un certain espace de recherche. Actuellement, le schéma d'analyse de conflits le plus couramment utilisé est basé sur l'analyse du graphe d'implications $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ de l'instance traitée. Ce graphe est un graphe orienté sans circuit (DAG) permettant de représenter les dépendances entre les clauses et les assignations obtenues par propagation des littéraux unitaires. Pour effectuer cela, $\mathcal{N}$ est constitué d'un nœud associé à chaque littéral affecté à vrai par l'interprétation courante. Ensuite, un arc entre un nœud x et un nœud y est dans $\mathcal{A}$ si l'affectation de x à vrai a impliqué l'affectation de y à vrai ($x \in exp(y)$).

Lorsque l'interprétation partielle est conflictuelle, le graphe d'implications généré comporte deux nœuds représentant deux littéraux complémentaires. Dans ce cas, il est possible d'ajouter un nœud supplémentaire ($\perp$) symbolisant une situation conflictuelle. Lorsqu'un tel nœud est présent il est possible d'analyser le graphe afin d'extraire les littéraux responsables de ce conflit. Généralement, cette analyse est basée sur la notion d'UIP (*Unique Implication Point*). Un UIP est un nœud du dernier niveau de décision qui domine le conflit [3].

Exemple 12

Soit la formule $\Sigma = \bigwedge_{i=1}^{13} \alpha_i$ avec :

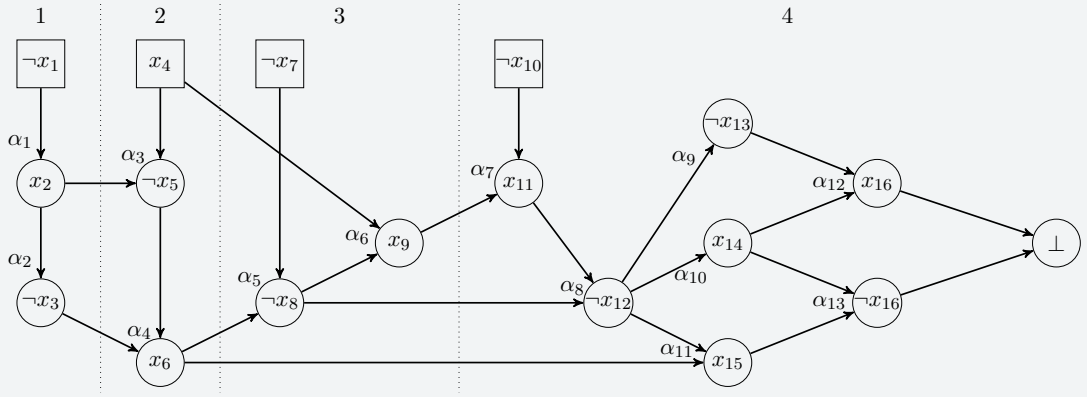
$\alpha_1 = x_1 \vee x_2$ $\quad \alpha_4 = x_3 \vee x_5 \vee x_6$ $\quad \alpha_7 = \neg x_9 \vee x_{10} \vee x_{11}$ $\quad \alpha_{10} = x_{12} \vee x_{14}$

$\alpha_2 = \neg x_2 \vee \neg x_3$ $\quad \alpha_5 = \neg x_6 \vee x_7 \vee \neg x_8$ $\quad \alpha_8 = x_8 \vee \neg x_{11} \vee \neg x_{12}$

$\alpha_3 = \neg x_2 \vee \neg x_4 \vee \neg x_5$ $\quad \alpha_6 = \neg x_4 \vee x_8 \vee x_9$ $\quad \alpha_9 = x_{12} \vee \neg x_{13}$

$\alpha_{11} = x_{12} \vee \neg x_6 \vee x_{15}$ $\quad \alpha_{12} = x_{13} \vee \neg x_{14} \vee x_{16}$ $\quad \alpha_{13} = \neg x_{14} \vee \neg x_{15} \vee \neg x_{16}$

$\mathcal{I} = \{\langle(\neg x_1^1), x_2^1, \neg x_3^1\rangle, \langle(x_4^2), \neg x_5^2, x_6^2\rangle, \langle(\neg x_7^3), \neg x_8^3, x_9^3\rangle, \langle(\neg x_{10}^4), x_{11}^4, \neg x_{12}^4, \neg x_{13}^4, x_{14}^4, x_{15}^4, x_{16}^4\rangle\}$ est l'interprétation partielle obtenue par propagation de la séquence de décisions $\langle \neg x_1, x_4, \neg x_7, x_{10}\rangle$. Nous représentons en rouge (respectivement vert) les littéraux de la formule falsifiés (respectivement satisfaits) par l'interprétation $\mathcal{I}$. L'interprétation $\mathcal{I}$ falsifie la formule (clause α_{13}). La figure suivante représente le graphe d'implications obtenu à partir de Σ et de l'interprétation $\mathcal{I}$. Le conflit porte sur la variable x_{16}.



3. Un nœud x domine un nœud y si et seulement si $niv(x) = niv(y)$ et $\forall z \in \mathcal{N}$, avec $niv(x) = niv(z)$, tous les chemins de z vers y passent par x.

La notion de point d'implication s'exprime par rapport à la notion de dominance. Plus précisément, un nœud x est un point d'implication unique si et seulement si x domine le conflit. Les UIP peuvent être ordonnés en fonction de leur distance avec le conflit. Le premier point d'implication unique (f-UIP pour « *First Unique Implication Point* ») est l'UIP le plus proche du conflit tandis que le dernier UIP (l-UIP pour « *Last Unique Implication Point* ») est le plus éloigné, c'est-à-dire le littéral de décision affecté au niveau du conflit.

Exemple 13

Reprenons le graphe d'implications $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ de l'exemple 12. Les nœuds $\neg x_{12}, x_{11}$ et $\neg x_{10}$ représentent respectivement le premier UIP, le second UIP et le dernier UIP.

La localisation d'un UIP permet de fournir une décision alternative provoquant le même conflit. À l'heure actuelle, les démonstrateurs `SAT` modernes utilisent pour la plupart la notion de f-UIP afin d'extraire un *nogood* (Zhang et al. 2001). Une clause *nogood* est générée en calculant des résolvantes entre les clauses responsables du conflit (utilisées durant la propagation unitaire) en remontant de celui-ci vers la variable de décision du dernier niveau jusqu'à obtenir une clause $\alpha \vee x$ contenant un seul littéral x du dernier niveau de décision (le UIP). C'est sur ce processus, nommé preuve par résolution basée sur les conflits, que la fonction `analyseConflit` est basée. Cette clause est nommée *clause assertive* et le littéral x est appelé *littéral assertif*.

Exemple 14

Considérons de nouveau la formule Σ et l'interprétation partielle $\mathcal{I}$ de l'exemple 12. La preuve par résolution basée sur les conflits est la suivante.

$$\begin{array}{lcll}
\sigma_1 & = & \eta[x_{16}, \alpha_{12}, \alpha_{13}] & = & x^4_{13} \vee \neg x^4_{14} \vee \neg x^4_{15} \\
\sigma_2 & = & \eta[x_{15}, \sigma_1, \alpha_{11}] & = & \neg x^2_6 \vee x^4_{12} \vee x^4_{13} \vee \neg x^4_{14} \\
\sigma_3 & = & \eta[x_{14}, \sigma_2, \alpha_{10}] & = & \neg x^2_6 \vee x^4_{12} \vee x^4_{13} \\
\sigma_4 & = & \eta[x_{13}, \sigma_3, \alpha_9] & = & \neg x^2_6 \vee x^4_{12} \\
\sigma_5 & = & \eta[x_{12}, \sigma_4, \alpha_8] & = & \neg x^2_6 \vee x^3_8 \vee \neg x^4_{11} \\
\sigma_6 & = & \eta[x_{11}, \sigma_5, \alpha_7] & = & \neg x^2_6 \vee x^3_8 \vee \neg x^3_9 \vee x^4_{10}
\end{array}$$

Les littéraux assertifs des clauses assertives σ_4, σ_5 et σ_6 représentent respectivement le premier UIP, le second UIP et le dernier UIP.

La clause assertive correspondant au f-UIP ainsi générée est apprise par le solveur et permet non seulement d'éviter à l'avenir d'atteindre à nouveau cet échec, mais aussi d'effectuer un retour-arrière. En effet, la connaissance de cette clause falsifiée permet d'affirmer que le littéral assertif doit être propagé plus tôt dans l'arbre de recherche. La hauteur du retour-arrière engendré par la clause assertive $\alpha \vee x$, calculé par la fonction `calculRetourArrière`, est le plus haut niveau de décision i des variables de α.

Exemple 15

Considérons la clause assertive $\sigma = (\neg x_6^2 \vee x_{12}^4)$ générée à partir de la formule Σ et l'interprétation partielle $\mathcal{I}$ de l'exemple 12. Le niveau de *backjump* calculé à partir de cette clause est égal à 2 et l'interprétation partielle obtenue après avoir effectué un retour-arrière et avoir propagé le littéral assertif est $\mathcal{I} = \{\langle(\neg x_1^1), x_2^1, \neg x_3^1\rangle, \langle(x_4^2), \neg x_5^2, x_6^2, x_{12}^2\rangle\}$.

Il est à noter qu'il est possible lors du processus d'analyse de conflits d'effectuer un travail supplémentaire. Par exemple, Sörensson et Biere (2009) proposent une approche qui consiste à continuer d'effectuer des résolutions si celles-ci n'augmentent pas la taille de la clause assertive. D'autres travaux ont montré comment découvrir des clauses sous-sommées durant l'analyse de conflits (Han et Somenzi 2009, Hamadi et al. 2009a). Une autre approche, proposée par Audemard et al. (2008), tente d'étendre la notion de graphe d'implications afin de considérer certaines clauses satisfaites par la formule. Une dernière approche, proposée par Nadel et Ryvchin (2010), cherche à améliorer la hauteur du saut en effectuant un retour arrière à un niveau de décision inférieur à celui proposé par la clause assertive et à affecter et propager l'ensemble des littéraux de celle-ci.

Il est important de noter que les choses commencent à changer dans le monde de SAT. En effet, de récents travaux (Nadel et Ryvchin 2018, Möhle et Biere 2019) remettent en cause l'utilisation des retours arrière non chronologiques. Ces travaux démontrent empiriquement que, dans de nombreuses situations, réaliser un retour arrière chronologique permettrait d'obtenir des meilleures performances. Ces résultats sont très récents, l'avenir nous dira si tel est le cas !

3.1.2.2 Réduction de la base de clauses apprises

L'analyse de conflits et l'apprentissage ont permis d'améliorer de manière significative l'algorithme `DPLL` (Marques-Silva et Sakallah 1996b). Cependant, comme le font justement remarquer Marques-Silva et Sakallah (1996b), il est nécessaire de gérer l'accroissement de la base de clauses apprises au risque de ralentir considérablement le processus de propagation unitaire (Eén et Sörenson 2004). Pour éviter cela, tous les solveurs `SAT` modernes réduisent la base de clauses apprises (`reductionClausesApprises`) à l'aide d'une fonction heuristique afin d'estimer la qualité des clauses. De plus, la plupart des approches conservent systématiquement les clauses de taille deux et les clauses considérées comme raisons d'un littéral propagé au moment de l'appel à la fonction réduction.

Dans la littérature, diverses approches ont été proposées pour définir quelles sont les clauses susceptibles d'être inutiles pour la suite de la recherche. La première à avoir été proposée est basée sur la notion de *first fail*. Cette stratégie, basée sur l'heuristique `VSIDS`, considère qu'une clause apparaissant peu dans la raison des conflits est inutile et qu'elle doit être supprimée. Audemard et Simon (2009b) proposent une autre mesure statique, appelée `LBD` (*Literal Block Distance*), qui correspond au nombre de niveaux différents intervenant dans la génération de la clause. Les auteurs montrent expérimentalement que les clauses ayant une faible valeur de `LBD` sont importantes pour la suite de la recherche. Dans (Audemard et al. 2011a), nous avons proposé une autre mesure, nommée `PSM`, qui représente le nombre de littéraux ayant la même polarité que le *progress saving* apparaissant dans cette clause. L'idée est de prendre en compte que l'heuristique *progress saving* est utilisée pour choisir la polarité d'une variable de décision,

afin d'estimer les chances qu'une clause sera assignée via le *progress saving*. Finalement, Jabbour et al. (2018) proposent une heuristique, nommée SRB, qui supprime aléatoirement les clauses d'une taille supérieure à une certaine limite.

Un autre point important concerne la fréquence avec laquelle la base de clauses apprises est nettoyée (Eén et Sörenson 2004, Audemard et Simon 2009a, Audemard et al. 2011a). Cette fréquence, calculée de manière heuristique (fonction `faireReduction`), si elle est mal gérée, peut conduire à rendre le solveur incomplet. En effet, contrairement à la méthode `DPLL`, l'approche `CDCL` a besoin des clauses apprises afin de se souvenir de l'espace de recherche qu'elle a déjà exploré. Ainsi supprimer une clause peut conduire le solveur à parcourir plusieurs fois le même espace de recherche et donc à visiter encore et encore le même conflit sans jamais s'arrêter (Pipatsrisawat et Darwiche 2011). Afin de résoudre ce problème et de garantir la complétude de la méthode, il est nécessaire que l'intervalle de temps atteigne une taille telle que l'ensemble des clauses apprises pouvant être généré dans cet intervalle permet d'assurer la terminaison de l'algorithme quelles que soient les clauses supprimées précédemment.

3.1.2.3 Redémarrages

Comme nous l'avons fait remarquer, les premiers choix effectués lors d'une recherche complète sont prépondérants. Gomes et al. (2000) ont montré expérimentalement qu'exécuter une même approche sur un même problème mais avec des choix initiaux différents conduit à des temps de résolution totalement hétérogènes. Ces expérimentations ont permis d'identifier un phénomène singulier nommé phénomène de longue queue (« *heavy tail* ») (Gent et Walsh 1994, Walsh 1999, Gomes et al. 2000, Chen et al. 2001). Afin d'y remédier les auteurs proposent de redémarrer la recherche au bout d'un certain temps (« *restart* »). L'idée est que si la recherche échoue depuis un certain temps (évalué en nombre de retours arrière) alors il est jugé peu probable que la recherche aboutisse en un temps raisonnable. Dans ce cas, l'algorithme est relancé avec la formule initiale tout en conservant certaines informations (les scores pour `VSIDS` par exemple) ou les clauses apprises afin d'effectuer des choix plus judicieux au début de l'arbre.

À l'heure actuelle, le constat n'est plus exactement le même, l'intérêt des redémarrages n'est plus tant d'essayer de chercher ailleurs dans l'espace de recherche, mais plutôt d'atteindre le même espace de recherche avec un chemin et des graphes d'implications différents (Biere 2008b). La fonction `redémarrage` de l'algorithme CDCL suit l'une des deux stratégies suivantes :

— **Stratégies de redémarrages statiques** : elles ont toutes en commun le fait d'être prédéterminées. Ces politiques reposent le plus souvent sur des séries mathématiques (Ryan 2004, Moskewicz et al. 2001b, Goldberg et Novikov 2002, Eén et Sörenson 2004, Walsh 1999, Luby et al. 1993, Biere 2008b). L'impact de toutes ces différentes stratégies de redémarrages sur le comportement d'un solveur `SAT` moderne (c'est-à-dire, à architecture `CDCL`) a été étudié dans (Huang 2007).

— **Stratégies de redémarrage dynamiques** : elles tentent d'exploiter différentes informations issues de la recherche (la taille des résolvantes (Pipatsrisawat et Darwiche 2009), la profondeur moyenne de l'arbre et des sauts arrière (Hamadi et al. 2010), le nombre de changements de valeur de variables (Biere 2008a)) afin de déterminer quand effectuer un redémarrage. Audemard et Simon (2009a) proposent une nouvelle approche qui dépend non seulement du nombre de conflits, mais aussi des niveaux de décision afin de déterminer si un *restart* doit être effectué. Audemard et Simon (2012) proposent d'utiliser la variation du `LBD` afin de déduire quand il faut déclencher un redémarrage. Plus précisément,

si le solveur produit des clauses avec des valeurs de LBD trop élevées pendant un certain laps de temps alors un redémarrage est réalisé. Biere et Fröhlich (2015) améliore cette dernière heuristique en utilisant une moyenne exponentielle glissante. Liang et al. (2018) proposent de réaliser un redémarrage lorsque le LBD des clauses commence à être supérieur à une certaine limite (cette limite est calculée grâce à une technique de *machine learning*). Haim et Walsh (2009) proposent aussi d'utiliser du *machine learning* mais cette fois pour choisir une stratégie de redémarrage parmi un portfolio. Dans (Nejati et al. 2017a), les auteurs proposent d'utiliser de l'apprentissage par renforcement afin de choisir l'heuristique de redémarrage qui va réduire la taille moyenne des valeurs LBD des clauses. Hamadi et al. (2010) proposent une approche basée sur l'évolution de la taille moyenne des retours arrière. L'idée est de délivrer pour de grandes (respectivement petites) fluctuations de la taille moyenne des retours arrière (entre le redémarrage courant et le précédent) une plus petite (respectivement grande) valeur de coupure.

3.2 Résolution de SAT sur une architecture à mémoire distribuée

Pendant de nombreuses années une méthode très simple pour doubler les performances d'un solveur était l'attente. En effet jusqu'en 2005, la puissance des processeurs augmentait en fonction du temps en suivant approximativement une loi exponentielle. Cette loi empirique connue sous le nom de Loi de Moore (1998) se vérifie ainsi depuis les débuts de l'informatique « moderne », c'est-à-dire depuis environ quarante ans. Mais depuis 2005, cette loi souffre d'un petit ralentissement : les dissipations thermiques empêchent une augmentation de la fréquence des processeurs. Pour contourner ce problème, et ainsi garantir une augmentation de la puissance de calcul, les constructeurs ont modifié leur stratégie de fabrication qui se tourne à présent vers l'utilisation de l'*hyperthreading* et d'une augmentation du nombre d'unités de calcul.

Une nouvelle architecture, appelée architecture multi-cœurs a été conçue ; elle peut être vue comme un ensemble de processeurs ayant la possibilité de communiquer ensemble. Et chaque processeur peut être vu comme une unité de calcul permettant d'exécuter une seule tâche à la fois. Cependant, pour des raisons de coût et aussi matériel, il n'est pas possible d'augmenter le nombre d'unités de calcul indéfiniment. Une autre manière d'augmenter le nombre d'unités de calcul est alors de considérer des architectures parallèles à mémoire distribuée. Dans de tels systèmes, le problème que nous souhaitons résoudre est exécuté sur une machine distante, mais contrairement aux architectures multi-cœurs, l'échange d'informations entre les unités de calcul est réalisé par le réseau via un « passage de messages ». Ainsi, contrairement aux architectures multi-cœurs, il est important d'être très précautionneux en matière d'échange de données. Il est aussi possible d'hybrider ces deux architectures afin de limiter autant que faire se peut la quantité de messages échangés. Nous verrons dans la suite que l'utilisation de tels systèmes permet l'obtention d'un gain substantiel dans le cadre de la résolution de SAT en parallèle (voir section 3.2.2). Dans le cadre de ce manuscrit nous choisissons de ne pas présenter les aspects technologiques liés au parallélisme. Néanmoins, le lecteur intéressé par ces aspects peut se référer à (Rajasekaran et Reif 2007) afin d'obtenir de plus amples informations.

Ces nouvelles architectures impliquent implicitement une révolution dans la manière de concevoir les solveurs SAT. Cependant, une caractéristique importante de ce problème est qu'il est très dur de prédire le temps nécessaire à l'exploration d'une branche de l'arbre de recherche. Par conséquent, il est difficile (voire impossible) de partitionner de manière uniforme l'espace de recherche avant l'exécution du programme. Pour contourner ce problème, il est nécessaire de

dégager du parallélisme. Pour réaliser cela, deux types d'approches sont principalement utilisés. D'une part, les approches de type « diviser pour régner » consistent à partager l'arbre de recherche (« *guiding path* ») (Zhang et al. 1996, Sinz et al. 2001, Blochinger et al. 2003, Chrabakh et Wolski 2003, Chu et Stuckey 2008, Vander-Swalmen et al. 2009, Ohmura et Ueda 2009, Schulz et Blochinger 2010, Audemard et al. 2014b, Heule et al. 2011b, Hyvärinen et Manthey 2012, Audemard et al. 2016, Nejati et al. 2017b, Frioux et al. 2019). D'autre part, les approches de type *portfolio* (Hamadi et al. 2009b, Guo et al. 2010, Audemard et al. 2012, Hamadi et al. 2011, Lazaar et al. 2012, Wieringa et Heljanko 2013, Audemard et Simon 2014, Biere 2017, Balyo et al. 2015, Frioux et al. 2017, Audemard et al. 2017) mettent les solveurs en concurrence, permettant ainsi de résoudre une formule à l'aide de différentes stratégies. Dans le cadre de ce manuscrit, nous n'irons pas plus loin dans la description des solveurs parallèles, le lecteur intéressé pourra se référer à (Hamadi et Sais 2018). Nous allons à présent présenter nos contributions quant à la résolution du problème SAT dans un contexte parallèle.

3.2.1 `AmPharoS` : un solveur parallèle adaptatif

La nature regorge d'exemples où un ensemble d'individus collaborent afin de mener à bien une tâche. Considérons par exemple les colonies de fourmis. Bien qu'individuellement cognitivement limitées, les fourmis sont capables de trouver efficacement des sources de nourriture et aussi d'identifier un chemin de ces dernières au nid qui soit quasiment optimal. Pour réaliser cette prouesse, les fourmis vont adopter un comportement collaboratif qui s'appuie sur un échange de messages. Lorsqu'une fourmi n'a pas identifié de source de nourriture, elle va partir en « éclaireuse », c'est-à-dire qu'elle va parcourir plus ou moins au hasard son environnement. Lorsqu'une source de nourriture est identifiée, elle en rapporte une petite quantité au nid, en laissant derrière elle une traînée de phéromones, pour marquer son chemin. Les autres fourmis tenteront ensuite de suivre ce chemin pour retrouver la source de nourriture. En passant plus nombreuses sur ce chemin, elles laissent plus de phéromones, attirant plus de fourmis, et créent ainsi un effet d'auto-renforcement. Le trajet le plus court sera donc plus marqué, ce qui conduira à la longue à la « création » d'un chemin quasiment optimal.

Cet exemple suggère le bénéfice qu'il peut y avoir à utiliser plusieurs unités de calcul pour résoudre un problème. Dans notre cas, une fourmi peut être assimilée à un solveur, l'environnement à l'espace de recherche généré par la formule à résoudre et un chemin « court » à une source de nourriture une solution de notre problème. Malheureusement, l'analogie s'arrête ici, les solveurs SAT parallèles ont en fait un comportement qui est plutôt faiblement collaboratif. En effet, dans le cas d'une approche de type diviser pour régner, la colonie de fourmis correspondante serait entièrement peuplée d'éclaireuses. Tandis que dans le cas d'approche de type *portfolio*, la colonie serait uniquement peuplée de fourmis incapables de produire des phéromones.

Afin d'améliorer la collaboration entre les solveurs, nous avons proposé une nouvelle approche, nommée `AmPharoS` (Audemard et al. 2016), pour la résolution du problème SAT en parallèle. Cette approche suit une stratégie de type maître-esclave où le maître (appelé `MANAGER`) va partager le travail entre les différentes unités de calcul via l'utilisation d'un arbre de guidage. Cependant, contrairement aux approches classiques, `AmPharoS` crée dynamiquement l'arbre de guidage en fonction des informations échangées avec les solveurs. De plus, le `MANAGER` communique avec les solveurs afin de leur assigner le chemin de guidage qui leur « convient » le mieux. Afin de mettre en place notre schéma algorithmique, nous devrons répondre aux questions suivantes :

— Comment démarrer la recherche ?

— Comment diviser le problème en sous-problèmes ?
— Comment les solveurs choisissent leurs sous-problèmes à résoudre ?
— Dans quel cas un problème est-il considéré comme résolu ?
— Quelles informations pouvons-nous partager entre les solveurs ?
— Pouvons-nous diviser la recherche en fonction des informations partagées ?

3.2.1.1 La gestion de l'arbre

La première question à laquelle il faut répondre est : comment diviser le problème en sous-problèmes ? Bien qu'`AmPharoS` soit un solveur de type « diviser pour mieux régner », il est important de noter qu'il n'utilise pas de stratégie de vol de travail (Zhang et Bonacina 1994). Il ne s'appuie pas non plus sur l'approche proposée par Heule et al. (2011b), qui répartit la charge de travail en divisant le problème en une centaine de milliers de sous-problèmes au début de la recherche. Dans notre cas, `AmPharoS` génère un ensemble de chemins de guidage, représenté de manière arborescente, qui couvre tout l'espace de recherche. Plus précisément, les nœuds de l'arbre sont des variables propositionnelles et les branches gauches (resp. droites) correspondent à l'assignement de la variable à vrai (resp. faux). Un solveur va alors résoudre la formule initiale conditionnée avec un terme correspondant au chemin de la racine à un nœud de l'arbre (généralement une feuille). Dans la suite nous ne faisons pas la différence entre un terme (qui est un chemin de la racine à un nœud de l'arbre) et le sous-problème qui lui est associé. Les feuilles de l'arbre peuvent être de deux types : `NIL` (le sous-problème n'a pas encore été résolu) ou `UNSAT` (le sous-problème a été montré incohérent). La figure 3.1 montre l'exemple d'un tel arbre. Il contient trois feuilles (les termes $[x_1, \neg x_2, x_4]$, $[x_1, \neg x_2, \neg x_4]$ et $[\neg x_1, \neg x_3]$), deux branches fermées (déjà prouvées insatisfaisables) et trois solveurs ($S_1, S2, S_3$) travaillant sur ces feuilles.

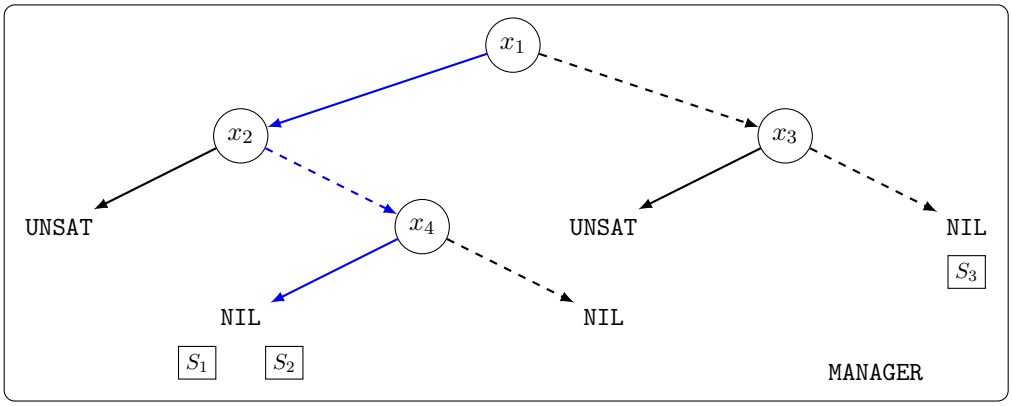


FIGURE 3.1 – Architecture globale d'`AmPharoS`.

Comme nous allons le voir dans la suite, les solveurs peuvent évoluer sur le même sous-problème (comme les solveurs S_1 et S_2 dans l'exemple de la figure 3.1). De plus, contrairement aux approches standards, les solveurs peuvent stopper leur recherche avant d'avoir trouvé une solution. Dans ce cas, le solveur communique avec le `MANAGER` afin de lui spécifier qu'il n'arrive pas à résoudre le sous-problème qui lui a été assigné. Le `MANAGER` lorsqu'il reçoit cette information vérifie si de nombreux autres solveurs ont déjà échoué à résoudre ce sous-problème. Si c'est le cas, alors le `MANAGER` et le solveur commencent une phase de communication afin d'étendre l'arbre de recherche pour casser la difficulté du sous-problème ainsi identifié (voir 3.2.1.1.3). Dans le cas où

l'arbre ne doit pas être étendu, le solveur va communiquer avec le MANAGER afin de sélectionner un sous-problème (potentiellement le même) à résoudre (voir 3.2.1.1.2). Lorsqu'un solveur prouve un sous-problème insatisfaisable, il communique avec le MANAGER afin de mettre à jour l'arbre en conséquence (voir section 3.2.1.1.4) et pour sélectionner un nouveau sous-problème. La fin du processus de résolution arrive finalement quand un sous-problème est prouvé satisfaisable ou quand chaque sous-problème de l'arbre complet est prouvé insatisfaisable.

Nous avons répondu à la question de comment est divisée la charge de travail, maintenant voyons comment démarrer la recherche, c'est-à-dire comment initialiser l'arbre de guidage.

3.2.1.1.1 Initialisation L'initialisation commence par l'exécution des solveurs sur le problème initial pendant un certain laps de temps. Dans notre cas, où nous avons utilisé des solveurs CDCL, ce laps de temps est mesuré en conflits et correspond à 10 000 conflits. Le but est de collecter des informations sur le problème, ce qui revient dans notre cas à initialiser l'activité des variables (associées à l'heuristique VSIDS) ainsi que leurs polarités. Il est à noter qu'il est nécessaire de s'assurer que les solveurs n'effectuent pas exactement le même travail. Pour cela, nous avons choisi de réaliser la première descente de chaque solveur aléatoire (*i.e.* le choix des variables et leurs polarités). De la même manière que dans (Martins et al. 2010), le premier solveur atteignant le nombre maximum de conflits communique sa meilleure variable, c'est-à-dire la variable avec la plus forte activité VSIDS dans notre cas. La variable ainsi choisie devient la racine de l'arbre et tous les solveurs arrêtent leurs recherches concurrentielles afin de demander un sous-problème au MANAGER. Ce sous-problème est choisi dans l'arbre qui contient uniquement deux feuilles, c'est-à-dire les termes disponibles sont restreints à un seul littéral (la variable choisie et son opposé). La figure 3.2 représente un tel arbre dans le cas où la variable sélectionnée est x_1.

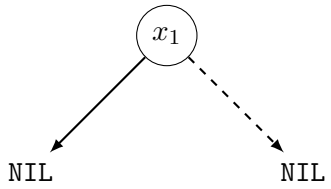
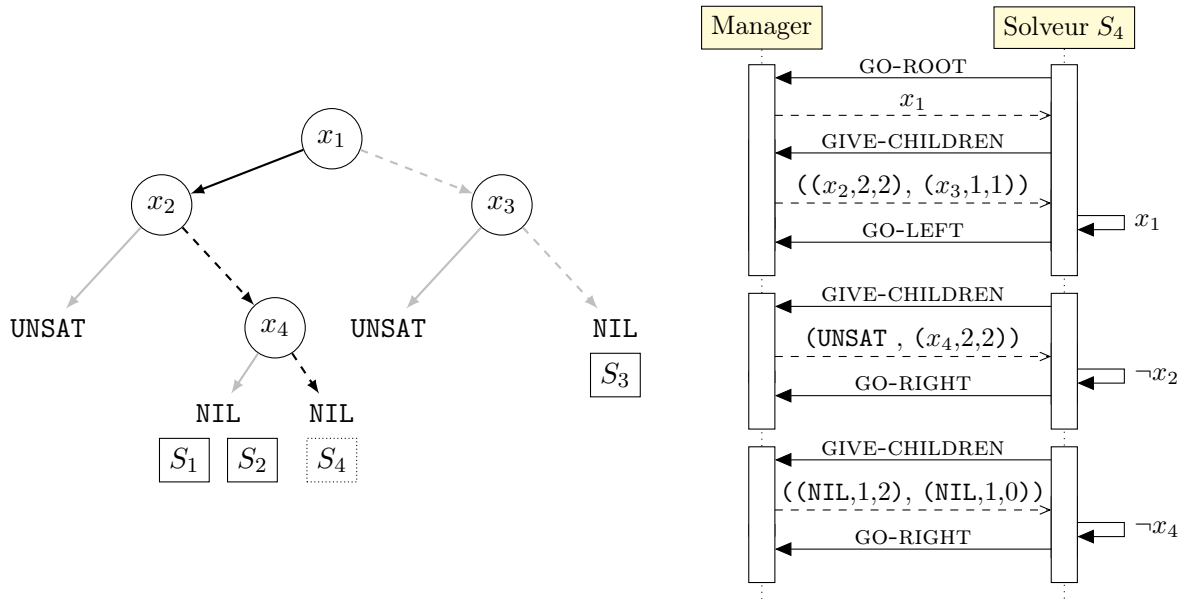


FIGURE 3.2 – Initialisation de l'arbre d'AmPharoS : la variable sélectionnée est x_1 et l'ensemble initial des cubes est $\{[x_1], [\neg x_1]\}$.

Maintenant que nous avons une petite idée de comment est construit l'arbre de guidage, considérons la question du choix du sous-problème affecté à un solveur.

3.2.1.1.2 Transmission Comme indiqué au préalable, un solveur peut arrêter sa recherche avant d'avoir fini de résoudre le sous-problème qui lui a été affecté. Dans notre cas, nous avons fixé à 10 000 le nombre de conflits alloués à un solveur pour la résolution d'un sous-problème. Lorsqu'un solveur atteint cette limite, il contacte le MANAGER afin de sélectionner un nouveau sous-problème. L'originalité de notre méthode est de laisser au solveur le choix du sous-problème qu'il aura à résoudre. La figure 3.3 montre un diagramme de séquence (figure 3.3b) qui illustre les messages échangés lorsque le solveur S_4 demande un nouveau terme au MANAGER (figure 3.3a).

Un premier message (GO-ROOT) est envoyé par le solveur pour demander la variable associée à la racine de l'arbre. Il reçoit x_1. Par la suite, à chaque étape de sélection d'une branche, le solveur demande les variables des nœuds fils de la variable précédemment reçue (message



(a) Arbre courant

(b) Diagramme de sequence

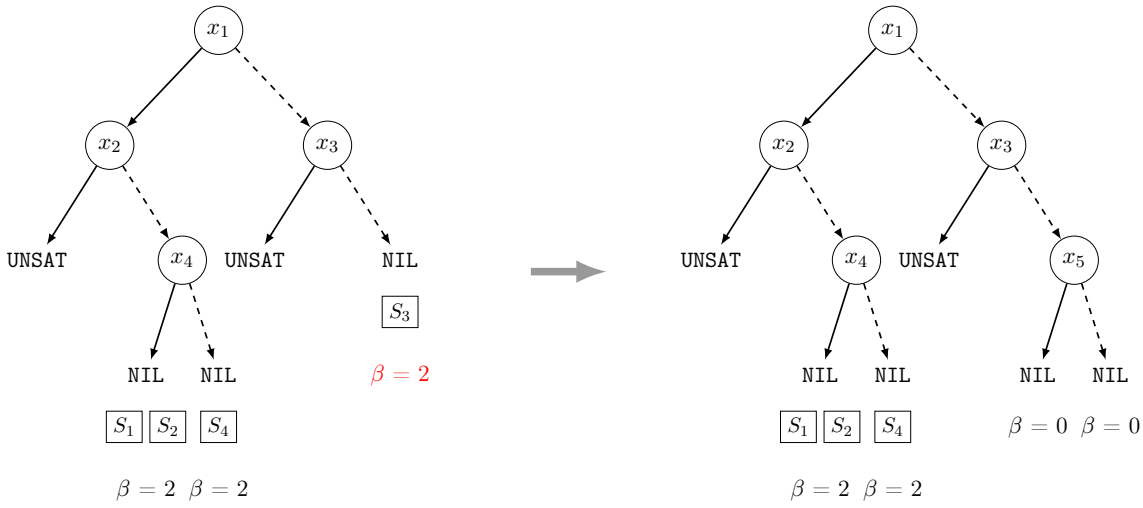
FIGURE 3.3 – Vue d'ensemble des messages échangés entre le solveurs S_4 et le `MANAGER` (Fig 3.3b) suivant une division de termes sous forme d'arbre (Fig 3.3a) (une ligne pleine (resp. en pointillés) affecte la variable à vrai (resp. faux)).

GIVE-CHILDREN). La réponse est composée de deux triplets : un pour chaque polarité du nœud courant. Chaque triplet est composé, dans cet ordre, de la variable du fils, du nombre de feuilles non résolues (nœuds `NIL`) et du nombre de solveurs travaillant à cet instant dans le sous-arbre. Sur la figure 3.3b nous voyons que le `MANAGER` retourne pour le premier message GIVE-CHILDREN le triplet $(x_2, 2, 2)$ pour la polarité positive de x_1 (la branche gauche contient deux feuilles, et deux solveurs travaillent sur ces feuilles (S_1 et S_2)) et le triplet $(x_3, 1, 1)$ pour la négative.

Lorsque le solveur obtient ces informations, il va considérer les triplets afin de sélectionner le sous-arbre à explorer. Par défaut, il choisit la branche possédant un nombre de solveurs inférieur au nombre de feuilles. L'idée est de couvrir le plus de termes dans l'arbre et ainsi de diversifier la position des solveurs. Si pour les deux branches, nous sommes dans la même configuration (c'est-à-dire les deux branches possèdent moins (respectivement plus) de solveurs que de sous-problèmes à résoudre), alors le solveur sélectionne la branche en fonction de son vecteur de polarité (*progress saving*). Après avoir sélectionné son sous-arbre, le solveur informe le `MANAGER` (via le message GO-LEFT ou GO-RIGHT) et considère le littéral associé dans la construction du terme à résoudre.

Ainsi, sur la figure 3.3, le solveur S_4 affecte x_1 (la racine) positivement (la condition mentionnée précédemment est fausse pour les deux branches). Par la suite, étant donné qu'une des branches liée à x_2 est déjà prouvée insatisfaisable, S_4 n'a pas d'autre alternative que d'affecter x_2 à faux (donc négativement). Puis, il doit mettre x_4 à faux puisque la condition précédente est prise en compte. Enfin, le solveur est arrivé sur une feuille (`NIL`) et peut commencer à résoudre le terme $[x_1, \neg x_2, \neg, x_4]$.

Voyons à présent comment l'arbre de guidage est développé en fonction des informations collectées par le `MANAGER`.



(a) Demande d'extension par le solveur S_3

(b) Extension acceptée par le `MANAGER`

FIGURE 3.4 – La figure de gauche représente l'arbre avant l'extension. Comme la valeur de β satisfait le critère d'extension, le `MANAGER` l'accepte et modifie l'arbre pour obtenir celle de droite.

3.2.1.1.3 L'extension Comme souligné dans le paragraphe sur l'initialisation de l'arbre de guidage, l'arbre contient à sa création un unique nœud et donc deux termes à résoudre (voir section 3.2.1.1.1). Pour diviser la formule originale, nous proposons d'étendre dynamiquement l'arbre de guidage pendant la recherche. Pour cela, nous associons à chaque feuille de l'arbre une variable entière β représentant la difficulté supposée du sous-problème associé. À chaque fois qu'un solveur arrête de résoudre un terme (sans trouver de solution), la variable β de la feuille associée à ce terme est incrémentée. De ce fait, plus β est grand, plus le sous-problème correspondant au terme associé est potentiellement difficile à résoudre (car beaucoup de solveurs n'ont pas réussi à le résoudre). Notons qu'un même solveur peut incrémenter plusieurs fois la même variable β. À chaque fois qu'un solveur demande un nouveau terme, la valeur β de la feuille associée est examinée. Si elle est supérieure ou égale au nombre de feuilles disponibles (c'est-à-dire les feuilles `NIL`) fois un facteur d'extension f_e, alors l'arbre est étendu au niveau de la feuille associée.

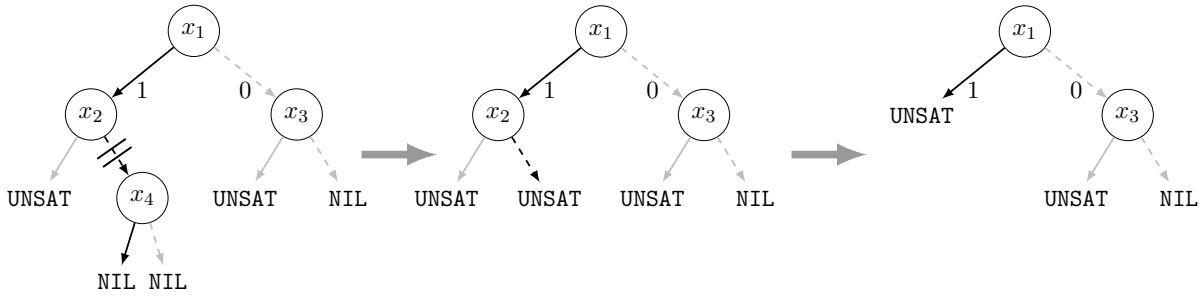
Lorsqu'un solveur arrive dans une situation où l'arbre doit être étendu, il transmet au `MANAGER` sa « meilleure » variable qui sera utilisée afin d'étendre l'arbre. La valeur β des deux nouvelles feuilles est initialisée à 0. Le fait de prendre en compte le nombre de feuilles disponibles permet de gérer le nombre de termes : plus l'arbre contient de termes, moins nous aurons d'extensions. De cette manière, et contrairement aux travaux de Heule et al. (2011b), notre approche évite de créer un trop grand nombre de termes, en prenant en compte des termes conduisant à des sous-problèmes déjà prouvés insatisfaisables. Comme une feuille peut contenir plusieurs solveurs, après une extension, plusieurs solveurs peuvent travailler sur des nœuds qui ne sont pas des feuilles.

La figure 3.4 montre l'exemple d'une extension sur l'arbre de la figure 3.3. Il contient 3 feuilles disponibles et 4 solveurs travaillent sur ces feuilles. Quand le solveur S_3 stoppe la résolution du sous-problème donné par le terme $[\neg x_1, \neg x_3]$, la variable associée β (en rouge) est incrémentée et devient égale à 3. En supposant que f_e soit égal à 1 (ce paramètre sera discuté à la section 3.2.1.3), la condition permettant l'extension est vérifiée. Dans ce cas, le solveur 3 envoie sa

meilleure variable (x_5) et le terme $[\neg x_1, \neg x_3]$ est étendu avec cette variable en générant deux nouveaux termes. Notons que la valeur β initiatrice de cette extension (en rouge) devient inutile car le nœud associé n'est plus une feuille. Le solveur S_3 est maintenant libre de demander au MANAGER un nouveau terme à traiter.

Puisque les solveurs résolvent certaines parties de l'espace de recherche représenté dans l'arbre de guidage, il est possible de simplifier l'arbre en supprimant certaines parties de ce dernier.

3.2.1.1.4 L'élagage Afin de conserver un maximum d'informations lorsqu'un solveur résout un sous-problème, nos solveurs CDCL travaillent sous hypothèses. Ce mode de fonctionnement, présenté plus en détail au chapitre 4, permet de conserver l'ensemble des clauses apprises. De plus, lorsqu'un sous-problème résolu sous hypothèses est montré incohérent, il est possible d'extraire une clause conflit qui bloque un sous-ensemble des hypothèses responsables de la non-cohérence de la formule. Afin d'élaguer l'arbre de recherche, cette information est transmisse au MANAGER. Remarquons qu'un solveur peut prouver directement l'insatisfaisabilité globale du problème lorsque la clause conflit calculée est vide. De plus, si les deux fils d'un nœud sont insatisfaisables alors ce nœud devient également insatisfaisable. Dans ce cas, ce nœud peut être supprimé et l'insatisfaisabilité est directement associée à la branche du parent. Bien sûr, ce processus est appliqué récursivement jusqu'à l'obtention d'un nœud possédant au moins un fils non insatisfaisable (figure 3.5).



(a) Conflit sur x_2 (b) Élagage sur x_2 (c) Élagage sur $\neg x_1$

FIGURE 3.5 – Élagage dans AmPharoS

Pour le moment, nous nous sommes principalement intéressés à la manière dont l'espace de recherche était divisé afin de partager le travail entre les différentes unités de calcul. Cependant, nous n'avons toujours pas répondu à la question qui consiste à déterminer le type d'informations à échanger. Dans la section suivante, nous proposons (lorsque les solveurs utilisés sont de type CDCL), d'échanger les clauses apprises.

3.2.1.2 L'échange des clauses

Dans cette section, nous examinons deux façons d'échanger les informations dans AmPharoS. Nous expliquons d'abord comment les clauses apprises par un solveur sont partagées avec les autres. Puis, nous présentons une approche originale permettant de partager des littéraux unitaires locaux en prenant avantage de la structure de l'arbre géré par le MANAGER.

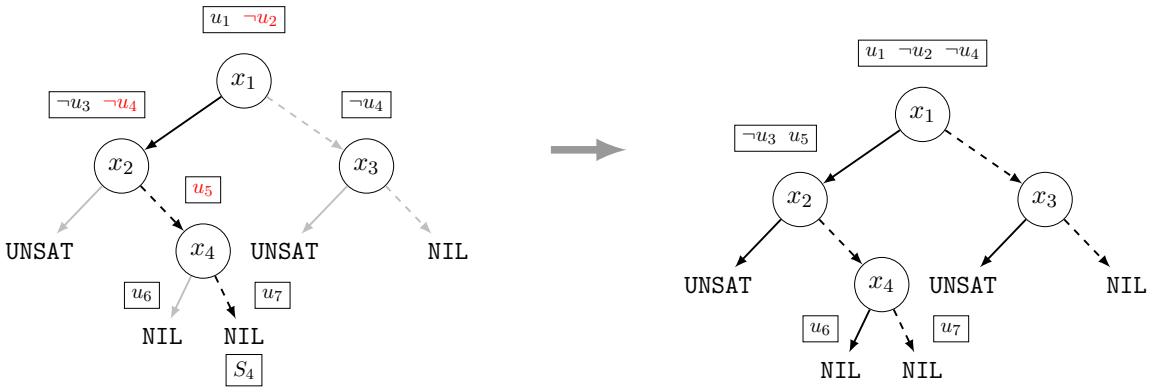
3.2.1.2.1 Le partage classique des clauses apprises Il est bien connu que le partage des clauses apprises améliore sensiblement la performance des solveurs SAT parallèles (Audemard et Simon 2014). Ainsi, dans `AmPharoS`, les solveurs ont aussi la capacité d'échanger certaines clauses apprises. Cependant, les solveurs ne s'échangent pas directement les clauses, mais ces dernières transitent via le `MANAGER`. Plus précisément, lorsqu'un solveur atteint un certain nombre de conflits (500 dans notre implémentation), il communique avec le `MANAGER` pour envoyer et/ou recevoir un ensemble de clauses. Les clauses devant être envoyées sont enregistrées dans une mémoire tampon qui est effacée après chaque communication avec le `MANAGER`. Comme il serait trop coûteux d'échanger toutes les clauses, uniquement les clauses avec une valeur de `LBD` inférieure ou égale à 2 sont partagées (cette stratégie est assez similaire à celle utilisée par Audemard et Simon (2014) dans leur solveur).

Le `MANAGER` gère les clauses apprises de tous les solveurs. Les clauses apprises sont stockées dans une file et le `MANAGER` vérifie périodiquement si elles sont subsumées ou pas. En pratique, un seul cœur de calcul est dédié au `MANAGER`. De ce fait, vérifier toutes les clauses d'une seule traite peut se révéler très coûteux et donc entraîner un blocage des communications entre le `MANAGER` et les solveurs. Afin d'éviter cette situation, le `MANAGER` calcule les clauses subsumées par paquet de 1 000 et peut ainsi s'occuper entre temps des communications avec les solveurs. Le `MANAGER` garde uniquement les clauses apprises qui ne sont pas subsumées dans sa base et il envoie à un solveur qui le demande les clauses fraîchement arrivées.

Afin de récupérer les clauses importées, les solveurs possèdent trois tampons : `standby`, `purgatory` et `learnt`. Les clauses reçues sont d'abord stockées dans le `standby` où, comme dans l'approche proposée dans (Audemard et al. 2012), les clauses ne sont pas attachées au solveur. Tous les 4 000 conflits, les clauses sont examinées afin de décider si elles doivent être transférées d'un tampon à un autre, être définitivement supprimées, ou rester dans le tampon courant. Une clause du `standby` peut être transférée vers le `purgatory`. Contrairement au `standby` les clauses du `purgatory` sont attachées au solveur et participent à la propagation. Nous discutons du critère permettant de déplacer une clause du `standby` au `purgatory` dans la section 3.2.1.3. De la même manière, une clause du `purgatory` peut être transférée dans `learnt` si elle est utilisée au moins une fois dans l'analyse d'un conflit. Le tampon temporaire `purgatory` est utilisé pour limiter l'impact des nouvelles clauses dans la stratégie de réduction des clauses apprises.

La stratégie de nettoyage de ces deux tampons supplémentaires (`standby` et `purgatory`) dépend d'un compteur associé à chaque clause. Le compteur est incrémenté à chaque fois que les clauses sont examinées. Si le compteur atteint un certain seuil (14 dans notre implémentation), la clause est supprimée. Notons que le compteur d'une clause est réinitialisé à chaque fois qu'elle change de tampon.

3.2.1.2.2 Les littéraux unitaires sous hypothèses Une autre façon d'échanger des informations entre les solveurs et le `MANAGER`, est de transférer les littéraux unitaires qui sont propagés grâce aux littéraux provenant des hypothèses. En effet, puisque chaque solveur travaille sous une hypothèse A (qui peut être vide) représentant un chemin dans l'arbre, lorsqu'un littéral $\ell \notin A$ est propagé grâce à une sous-hypothèse $A' \subseteq A$, cette information peut être transmise au `MANAGER` afin d'être diffusée aux autres solveurs. Plus précisément, le solveur communique au `MANAGER` que ℓ peut être propagé avec A'. De l'autre côté, quand un solveur sélectionne une branche associée à un littéral ℓ' durant la transmission d'un terme, il va aussi recevoir un ensemble de littéraux unitaires associé à ℓ' afin de les propager. De ce fait, la transmission d'un terme (voir section 3.2.1.1.2) contient ces messages additionnels.



(a) S_4 donne $\neg u_2$ $\neg u_4$ u_5 au `MANAGER`

(b) Gestion des littéraux unitaires

FIGURE 3.6 – Littéraux unitaires sous hypothèse dans `AmPharoS`. La figure de gauche montre un arbre décoré avec les littéraux donnés par S_4 en rouge. Ces littéraux sont remontés, en utilisant deux règles (insatisfaisabilité pour u_5 et littéraux identiques pour $\neg u_4$) afin d'obtenir celle de droite.

Par conséquent, le `MANAGER` s'occupe de décorer l'arbre contenant les chemins de guidage par des ensembles de littéraux unitaires devant être propagés à chaque branche. La figure 3.6a montre l'exemple d'un tel arbre. Lorsque S_4 demande une branche, il commence par collecter l'ensemble des littéraux unitaires $\{u_1\}$. Il propage aussi $\neg u_2$ (en rouge) et donne cette information au `MANAGER`. Il choisit la branche x_1 et récupère le littéral $\neg u_3$ afin de le propager. De la même manière, il propage aussi $\neg u_4$ et apporte ce littéral unitaire sous hypothèses au `MANAGER`.

Nous avons démontré expérimentalement dans (Audemard et al. 2016), que les littéraux obtenus sous hypothèses sont très importants, car ils permettent de réduire l'espace de recherche d'une branche donnée. Ainsi, le fait qu'un littéral ℓ peut être propagé à partir de A' est pris en compte dans le solveur concerné en ajoutant, dans une base dédiée qui n'est jamais nettoyée, une clause composée de la négation de A' et du littéral ℓ'. Remarquons que lorsque $A' = \varnothing$, la clause formée du littéral ℓ' est unitaire, et ℓ' est ajouté aux clauses unitaires du solveur.

Les informations collectées de cette manière peuvent ensuite être utilisées afin de remonter des littéraux unitaires plus haut dans l'arbre de guidage. Cette situation arrive soit quand une branche est prouvée insatisfaisable (Cas 1) ou lorsque deux branches d'un même nœud possèdent le même littéral (Cas 2) comme cela est fait dans (Le Berre 2001). Dans le premier cas, tous les littéraux de la branche satisfaisable sont remontés dans la branche père (comme le littéral u_5 sur la figure 3.6b). Dans le second cas, les littéraux apparaissant dans les deux branches d'un nœud sont remontés dans la branche père (c'est le cas du littéral $\neg u_4$ sur la figure 3.6b). Ce processus est exécuté récursivement jusqu'à l'obtention d'un point fixe. Remarquons que lorsque la branche père n'existe pas (cela arrive quand les littéraux sont déplacés à partir d'une branche de la racine), ces littéraux sont prouvés unitaires.

Nous avons pour le moment laissé en suspens le facteur d'extension f_e. Cependant, ce dernier est très important puisqu'il va contrôler la manière dont l'arbre de guidage va évoluer tout au long de la recherche. En fait, avec ce facteur d'extension nous devons essayer de régler un dilemme entre étendre fortement l'arbre, et donc éparpiller nos solveurs dans l'espace de recherche, ou au contraire ne pas étendre l'arbre et ainsi intensifier la recherche aux mêmes endroits. Dans la section suivante, nous proposons une approche dynamique pour régler ce problème qui s'appuie sur la redondance des informations échangées.

3.2.1.3 Intensification vs. diversification

Quand plusieurs solveurs résolvent en concurrence un problème, ils peuvent réaliser des recherches redondantes. Identifier une telle situation permettrait de pouvoir modifier la stratégie des solveurs afin de diversifier leurs recherches. Toutefois, à cause du partage des clauses apprises entre les solveurs, explorer différents espaces de recherche pourrait être un handicap. Dans une telle situation, focaliser tous les solveurs sur le même espace de recherche (intensification) pourrait être essentiel.

Ce paradigme, appelé dilemme d'intensification/diversification, a déjà été étudié dans le contexte des solveurs SAT de type *portfolio*. Il peut être traité soit statiquement, en utilisant plusieurs solveurs avec des stratégies opposées (Audemard et al. 2012, Guo et al. 2010), soit dynamiquement, en modifiant la stratégie des solveurs pendant la recherche. Même si beaucoup de solveurs diversifient la recherche, aucun critère ne permet d'identifier si plusieurs solveurs exécutent une recherche redondante mis à part la mesure sur la polarité proposée dans (Guo et Lagniez 2011a). Dans notre contexte, il est possible de tirer parti des clauses transitent par le `MANAGER` afin de mesurer le degré de redondance des solveurs. Plus précisément, nous proposons de mesurer le degré de redondance en prenant en compte le nombre de clauses redondantes partagées entre les solveurs. Nous utilisons une liste afin de mémoriser depuis le début le nombre de clauses reçues (st_r) et une autre afin de mémoriser le nombre de clauses gardées (st_k). Les clauses gardées sont celles qui n'ont pas été supprimées durant la vérification des clauses subsumées. Quand un solveur revient vers le `MANAGER` pour partager des clauses (tous les 1 000 clauses), le nombre de clauses reçues (resp. gardées) depuis le début est sauvegardé dans st_r (resp. st_k) par le `MANAGER`.

La *redundancy shared clauses measure* ($rscm$) est définie pour une étape t suivant un intervalle glissant de taille m (20 000 dans nos expérimentations) comme le ratio entre le nombre de clauses reçues durant les m mises à jour de st_r jusqu'à t et le nombre de clauses gardées durant ce même temps. Plus précisément, nous avons $\forall j < 0,\ st_r[j] = st_k[j] = 0$:

$$\begin{aligned} rscm_t &= \frac{st_r[t] - st_r[t-m]}{st_k[t] - st_k[t-m]},\ \text{si } st_k[t] - st_k[t-m] \neq 0 \\ rscm_t &= st_r[t] - st_r[t-m],\ \text{sinon} \end{aligned} \tag{3.1}$$

Tout d'abord, notons que lorsque plusieurs solveurs travaillent sur le même espace de recherche, il y a une forte probabilité que les clauses apprises par les différents solveurs soient identiques. Cela signifie que le nombre de clauses subsumées est important, conduisant à un $rscm$ élevé. Inversement, lorsque les solveurs sont diversifiés dans l'espace de recherche, il y a une forte probabilité d'avoir des clauses non redondantes, et donc, d'avoir un $rscm$ faible. Par conséquent, un $rscm$ faible indique que le solveur doit intensifier la recherche, tandis qu'une valeur $rscm$ élevée signifie que les solveurs doivent diversifier leur recherche.

Il y a plusieurs façons de diversifier ou d'intensifier les solveurs (clauses apprises, heuristiques des solveurs, ...). Dans `AmPharoS`, nous choisissons de résoudre le dilemme d'intensification/-diversification en contrôlant deux critères : la manière dont l'arbre est étendu et le nombre de clauses passant du `standby` aux `purgatory`. Ainsi, pour nous, diversifier (resp. intensifier) la recherche consiste à augmenter (resp. diminuer) ces deux paramètres.

Peu de clauses subsumées ($rscm$ est petit)	**Beaucoup de clauses subsumées** ($rscm$ est grand)
Réduire l'extension	Favoriser l'extension
Augmenter les clauses importées	Limiter les clauses importées
Intensification	**Diversification**

En ce qui concerne l'extension de l'arbre de guidage, remarquons que chaque chemin de la racine à une feuille représente un ensemble unique de littéraux qui divise l'espace de recherche d'une manière déterministe. Ainsi, plus l'arbre est grand, plus la probabilité d'exécuter deux solveurs dans deux sous-problèmes distincts augmente. Afin de contrôler cette croissance, nous définissons le facteur d'extension f_e comme étant égal à $\frac{1\ 000}{rscm_t{}^3}$. Rappelons que ce facteur d'extension est utilisé afin d'accroître ou de diminuer l'extension de l'arbre et est associé au nombre de termes disponibles. Par conséquent, plus la valeur $rscm_t$ est petite (resp. grande), plus la valeur f_e sera grande (resp. petite), et donc plus l'extension de l'arbre sera lente (resp. rapide). Notons que le $rscm_t{}^3$ (au cube) permet de diminuer f_e rapidement, tandis que la division (par 1 000) permet de le limiter au cas où les solveurs seraient en concurrence. Afin d'empêcher l'arbre de s'étendre trop, nous avons également limité la valeur de f_e a un maximum de 10.

Pour ce qui est du déplacement des clauses de la base `standby` à la base `purgatory`, nous allons cette fois essayer de limiter le passage de clauses déjà subsumées dans la base `purgatory`. En effet, il semble naturel que le nombre de clauses acceptées par un solveur (celles passant du `standby` au `purgatory`) augmente (resp. diminue) quand la valeur $rscm$ diminue (augmente). Autrement dit, plus (resp. moins) nous avons de clauses subsumées, moins (resp. plus) les solveurs ajoutent des clauses reçues. Pour contrôler la quantité de clauses passant du `standby` au `purgatory`, nous utilisons la notion de *psm* (voir section 3.1.2.2), déjà utilisée dans le solveur de type *portfolio* proposé par Audemard et al. (2012). Plus précisément, nous nous appuyons sur les observations réalisées dans (Audemard et al. 2011a) qui montrent qu'il est plus judicieux de conserver les clauses avec une petite valeur de *psm*. Nous combinons alors les valeurs *psm* et $rscm$ de manière à ce qu'une clause soit autorisée à passer du `standby` au `purgatory` quand sa valeur *psm* est inférieure ou égale à $\lfloor \frac{psm_{max}}{rscm_t} \rfloor$, où psm_{max} correspond au *psm* maximum accepté (fixé à 6 dans nos expérimentations). En conséquence, les clauses avec un *psm* de zéro sont systématiquement acceptées quelle que soit la valeur $rscm$. Tandis que les clauses possédant une valeur *psm* élevée sont acceptées si et seulement si il y a peu de clauses subsumées au niveau du `MANAGER`.

3.2.1.4 Discussion

Puisque l'arbre de guidage est utilisé pour assigner les sous-problèmes à envoyer aux solveurs, il est clair que l'heuristique qui aide à sa construction est importante. En fait, si nous nous plaçons du point de vue d'un solveur `SAT`, chaque chemin de guidage conditionne les premiers choix de variables réalisés par ce dernier. Il est connu que ces choix sont prépondérants quant à la faculté que le solveur a à trouver une solution. Nous avons choisi d'utiliser l'heuristique de choix de variables du dernier solveur qui demande à étendre l'arbre, et qui était l'heuristique `VSIDS`. Ce choix nous a semblé judicieux du fait que l'heuristique `VSIDS` a tendance à être très efficace en pratique. Cependant, contrairement aux solveurs `SAT`, qui utilisent cette heuristique en combinaison avec une stratégie de redémarrage, l'arbre de guidage n'est jamais réinitialisé. Le fait de redémarrer corrige les mauvais choix, ce qui dans notre cas ne peut pas être réalisé.

Afin de corriger ce problème, une solution serait d'utiliser une heuristique de type *look-ahead*. C'est ce que Nejati et al. (2017b) ont réalisé dans leur version d'`AmPharoS`, nommée MapleAmpharos. Plus précisément, quand un *worker* a atteint sa limite de conflit pour potentiellement changer le sous-problème sur lequel il travaille, le *worker* choisit d'envoyer au `MANAGER` la variable qui a causé le plus grand nombre de propagations par décision (le ratio de propagation). Plus précisément, quand une variable v est décidée, MapleAmpharos compte le nombre de propagations unitaires notées $nbPropagations(v)$ effectué à la suite de cette décision. Ainsi, durant la durée de travail d'un *worker* sur un sous-problème (pendant un certain nombre de conflits), à chaque fois que la variable v est décidée, $nbPropagations(v)$ est mis à jour en lui ajoutant les propagations unitaires associées et le nombre de décisions de la variable v noté $nbDecisions(v)$ est incrémenté. Par la suite, le ratio de propagation d'une variable $PRSH(v)$ est calculé comme suit :

$$PRSH(v) = \frac{nbPropagations(v)}{nbDecisions(v)}$$

$PRSH(v)$ est donc calculé quand un *worker* a fini de travailler sur un sous-problème ou que ce dernier est prouvé insatisfaisable. Quand une extension est effectuée, la variable ayant le plus grand ratio est choisie. Grâce à cette heuristique, les auteurs ont amélioré significativement les performances de leur solveur.

Néanmoins, si cette heuristique a permis d'améliorer les performances d'`AmPharoS`, elle reste encore très « dictatoriale ». En effet, le choix de la variable est toujours décidé par un seul *worker*. Cependant, nous savons que plusieurs solveurs ont déjà travaillé sur le sous-problème incriminé. Il est donc possible de définir une heuristique un peu plus « démocratique » et qui demande à chaque solveur sa ou ses meilleure(s) variable(s) lorsqu'il stoppe sa recherche de solution sur un sous-problème. Une fois ces informations collectées, il est alors possible d'utiliser une méthode de vote pour choisir le meilleur candidat (Brandt et al. 2016).

Nous pourrions aussi vouloir laisser au `MANAGER` le choix de la prochaine variable à utiliser pour étendre l'arbre de guidage. En effet, contrairement aux *workers*, le `MANAGER` a une vision globale de l'arbre. Ainsi, il serait possible de mettre en place des stratégies s'appuyant sur la structure du graphe de contraintes associée au problème à résoudre. Nous pourrions exploiter par exemple des mesures de centralité (Brandes et Erlebach 2005) ou encore la décomposition arborescente du graphe (Jégou et Terrioux 2017).

Une autre manière de résoudre ce problème serait de pouvoir élaguer tout ou partie de l'arbre de guidage. Dans ce cas, il pourrait être avantageux de partager avec les solveurs les informations contenues dans l'arbre et qui ont trait avec les littéraux unitaires ajoutés dans l'arbre ainsi que les sous-arbres montrés incohérents (Lecoutre et al. 2007). Il pourrait aussi être largement envisageable de maintenir plusieurs arbres de guidage et de laisser aux *workers* le choix de l'arbre à explorer.

Un autre point important concerne le choix du sous-problème sélectionné par un solveur. Dans notre version, nous avons choisi d'équilibrer la charge de travail entre les *workers*, en essayant autant que faire se peut d'assigner des solveurs dans les parties de l'arbre de guidage où il y a des problèmes non affectés. Cependant, une telle démarche casse le caractère *portfolio* de notre approche. De plus, cette stratégie ne tient pas compte de la difficulté des sous-problèmes dérivables des sous-arbres. Dans nos travaux futurs, nous souhaitons nous inspirer d'une approche de recherche de type Monte Carlo afin d'essayer de régler le problème d'intensification/diversification. La difficulté reste qu'il faudra être en mesure d'évaluer la qualité d'un sous-arbre.

Pour cela, nous pourrions considérer par exemple le nombre de littéraux unitaires attachés aux branches. De plus, dans ce contexte il faudrait être en mesure d'utiliser un nombre d'unités de calcul largement supérieur à ce que nous avons utilisé jusqu'à présent pour nos expérimentations afin d'être plus précis quant à l'évaluation de la pertinence d'un sous-arbre.

Une autre problématique concernant `AmPharoS` vient de l'utilisation d'une architecture centralisée pour l'échange d'informations entre les solveurs. En effet, nous avons montré expérimentalement que l'utilisation de cette architecture sur une approche de type *portfolio* rendait obsolète l'échange de clauses. Cette observation a conduit aux résultats présentés dans la section suivante et qui concernent l'implémentation d'une version distribuée du solveur `Syrup`.

3.2.2 `d-Syrup`

Avec l'augmentation des serveurs *cloud*, il est aujourd'hui possible d'avoir accès facilement à des milliers de machines. À ce jour, pour pouvoir tirer parti de cette puissance de calcul il est nécessaire de construire des solveurs `SAT` qui fonctionnent sur des architectures distribuées. Cependant, comme en témoigne le fait qu'il n'existe pas de compétition `SAT` de solveurs fonctionnant sur de telles architectures, les solveurs distribués sont moins performants que leurs homologues construits pour fonctionner sur des architectures multi-cœurs. Cette perte d'efficacité est due principalement au fait que l'échange d'informations se fait via le réseau et non pas directement via une zone de mémoire partagée.

Dans cette section, nous proposons une version distribuée de `Syrup` qui tente de répondre au problème de congestion réseau qui survient dans le cas d'un échange massif de clauses. Plus précisément, nous proposons un modèle de programmation hybride qui permet de limiter la quantité d'échange d'informations en considérant deux niveaux de communication : communication entre machines et communication sur la même machine. Ce schéma de programmation a aussi la particularité de ne pas utiliser de cycle de communication et d'échanger les clauses apprises dès que possible.

3.2.2.1 Description de `Syrup`

Le solveur `Syrup` est un solveur parallèle de type *portfolio* basé sur Glucose (Audemard et Simon 2009b). Il a été implémenté pour fonctionner en multi-cœur et il échange les clauses entre les solveurs en utilisant la mémoire partagée. La figure 3.7 donne une vue générale de la gestion du partage des clauses entre n solveurs séquentiels Glucose. Lorsqu'un solveur décide de partager une clause (les flèches rouges en pointillé), cette clause est ajoutée dans un *buffer* localisé dans la mémoire partagée. Syrup tente de réduire le nombre de clauses partagées entre les cœurs de calcul en envoyant uniquement les clauses unaires, binaires, « collantes » (*glue*) et celles utilisées au moins deux fois durant les analyses de conflits. Il est important de noter que le *buffer* de la mémoire partagée est limité par défaut à $0.4MB\times$ le nombre de *threads* et qu'une clause est supprimée du *buffer* quand tous les *threads* l'ont récupérée. Par conséquent, si le *buffer* est plein au moment de l'exportation d'une clause, cette dernière n'est simplement pas échangée. Cela permet de ne pas surcharger la mémoire partagée et ainsi ne pas ralentir les solveurs quand le nombre de clauses à partager est élevé.

L'importation est réalisée à chaque redémarrage d'un solveur. Comme pour le solveur PeneLoPe (Audemard et al. 2012), les clauses importées (représentées par les flèches rouges pleines) ne sont pas directement ajoutées dans la base de clauses apprises d'un solveur. Mais elles sont

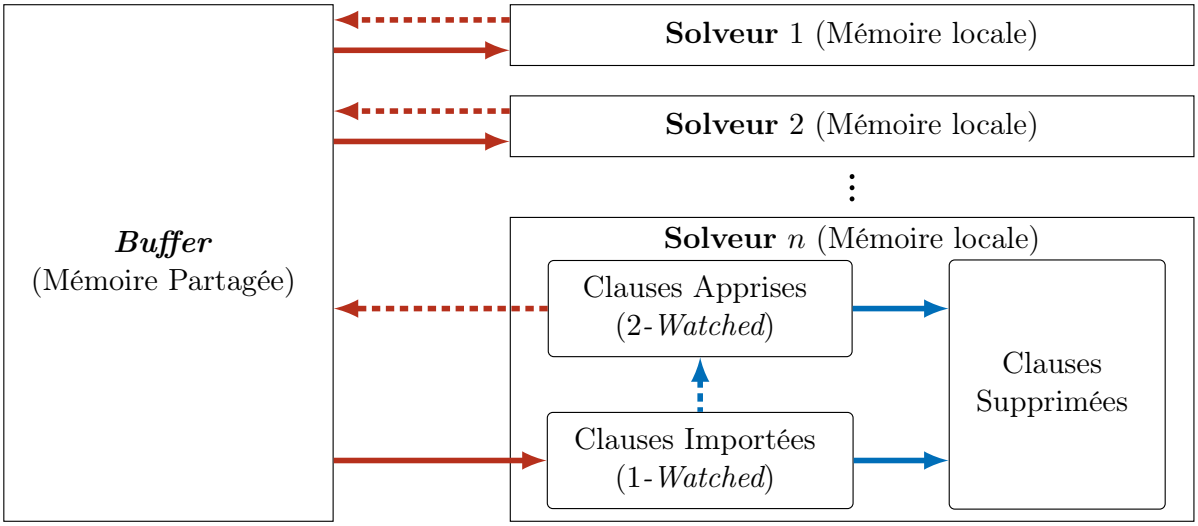


FIGURE 3.7 – Architecture du solveur *portfolio* `Syrup`

stockées dans une base transitoire où elles ne se voient assigner qu'une seule sentinelle. Ce mécanisme, alors appelé 1-*Watched*, est suffisant pour assurer la détection de toutes clauses conflictuelles pendant la propagation unitaire. Néanmoins, il n'est pas possible de détecter les clauses devenant unitaires et donc certains littéraux ne seront pas propagés. Toutefois, en détectant uniquement les conflits, la structure 1-*Watched* a l'avantage d'être beaucoup plus rapide en temps de calcul. Elle est donc un très bon candidat pour une heuristique permettant de juger la qualité d'une clause fraîchement importée. Ainsi, quand une clause importée est en conflit, elle est traitée comme une clause apprise et est intégrée dans la base des clauses apprises du solveur. Elle devient donc 2-*Watched* et peut alors propager des littéraux pendant la recherche (cela est représenté par les flèches bleues en pointillé). Cela permet de sélectionner dynamiquement les clauses utiles lors de leur réception, c'est-à-dire, qui sont celles pertinentes en fonction de l'espace de recherche courant d'un solveur receveur (comme le solveur PENELOPE avec PSM). Comme le nombre de clauses importées peut être, par moment, très élevé, SYRUP utilise une stratégie dédiée qui supprime périodiquement les clauses qui sont hypothétiquement moins utiles dans la recherche (flèches bleues pleines).

Même si `Syrup` est capable de gérer un nombre de cœurs de calcul arbitraire, il n'a pas été pensé pour fonctionner en mode distribué. En particulier, la quantité de clauses échangées entre les unités de recherche peut dans ce contexte être problématique. En effet, le partage de données sur le réseau n'est pas aussi simple et aussi efficace que sur une architecture multi-cœurs. Notamment, ces données doivent être échangées à n'importe quel moment durant la recherche. Dans (Audemard et al. 2017), nous avons étudié différents modèles de programmation parallèles afin de rendre le solveur `Syrup` efficace sur une architecture à mémoire distribuée. Ces travaux ont conduit à l'élaboration du solveur `d-Syrup` qui s'appuie sur l'utilisation d'un modèle de programmation parallèle complètement hybride.

3.2.2.2 Architecture de programmation utilisée dans `d-Syrup`

Deux méthodes peuvent être employées afin de partager des informations dans une architecture à mémoire distribuée. La première manière, dite centralisée, échange les informations en les envoyant à un processus maître qui les renvoie aux autres unités de calcul. Une telle approche

pourrait fidèlement reproduire l'architecture parallèle de `Syrup` et donc rendre le solveur fonctionnel sur une architecture distribuée. Cependant, comme nous l'avons démontré expérimentalement dans (Audemard et al. 2016), cela pose des problèmes de congestion de réseau.

Dans la seconde architecture, dite décentralisée, chacun des processus communique directement les informations avec tous les autres, sans passer par un processus maître. Même s'il y a moins de problèmes de congestion de réseau, un bouchon dans les communications peut encore survenir quand le nombre de clauses à partager est trop élevé. De plus, les communications réalisées doivent être prudemment réalisées afin d'éviter les interblocages. Notons bien ici que nous parlons des *deadlocks* engendrés par l'échange de messages via le réseau, pas ceux induits par la mémoire partagée et leurs sections critiques associées (mutex). Un *deadlock* apparaît quand un processus demande une ressource qui est déjà occupée par d'autres processus. Par exemple, considérons le cas où chaque processus est implémenté de sorte qu'il réalise respectivement une opération `send` bloquante puis une opération `receive` elle aussi bloquante. Le terme « bloquante » signifie ici que l'opération `send` (resp. `receive`) est bloquée tant que le message n'a pas été envoyé (resp. reçu) sur le réseau. Dans une telle situation, si deux processus envoient chacun un message à l'autre, alors chaque processus doit attendre que le message soit reçu par l'autre pour continuer. Par conséquent, ils sont tous les deux bloqués dans leurs opérations `send`. Plus précisément, une opération `send` n'est pas complétée tant que l'opération `receive` correspondante n'est pas exécutée.

Une manière d'éviter les problèmes d'interblocage consiste à utiliser des communications non-bloquantes et/ou collectives. Généralement, la solution retenue consiste à attendre que toutes les zones tampons contenant les messages à envoyer puissent être consultées et modifiées sans risque, donc que les messages aient été envoyés (les messages n'ont pas été nécessairement reçus). Cette solution est réalisée via ce que nous appelons un cycle de communication.

Un autre point important concerne la manière de réaliser les communications entre les solveurs. Deux solutions sont généralement considérées. La première consiste à alterner périodiquement une phase de recherche et une phase de communication. La seconde consiste à utiliser deux *thread* distincts, un qui s'occupe de la recherche et l'autre de la communication. Dans notre cas, nous avons choisi d'utiliser la seconde manière de communiquer. Cette solution nous a semblé la plus pertinente dans notre contexte où nous avons peu de prise sur la quantité d'informations qui devra être échangée.

Dans (Audemard et al. 2017), nous avons étudié différentes architectures de programmation afin de proposer une version distribuée de `Syrup`. La première architecture étudiée était assez basique. Elle consistait à voir chaque unité de calcul comme un ordinateur et à faire communiquer toutes ces unités ensemble. Une telle architecture, représentée sur la figure 3.8, associe deux *threads* par processus solveur. Le processus solveur PROCESSUS$_1$ représenté par un carré en pointillés, regroupe un *thread* de recherche S_1 (un solveur) et son propre *thread* communicateur C_1. Les données (les clauses apprises) sont partagées entre ces deux *threads* en utilisant la mémoire partagée via un tampon de données (BUFFER).

Lorsqu'un processus solveur, PROCESSUS$_1$ sur la figure, veut communiquer ses clauses, il doit d'abord les transférer du *thread* S_1 à son *thread* communicateur C_1 via un BUFFER utilisant la mémoire partagée. Ensuite, chaque *thread* communicateur C_n tel que $n \neq 1$ reçoit les clauses apprises à partir de C_1 uniquement par passage de messages et cela même quand deux *threads* sont sur la même machine. Pour finir, chaque *thread* communicateur C_n envoie les clauses apprises reçues à leur *thread* de recherche respectif S_n (associé au PROCESSUS$_n$) tel que $n \neq 1$.

Le principal désavantage de cette architecture est que chaque processus solveur se comporte

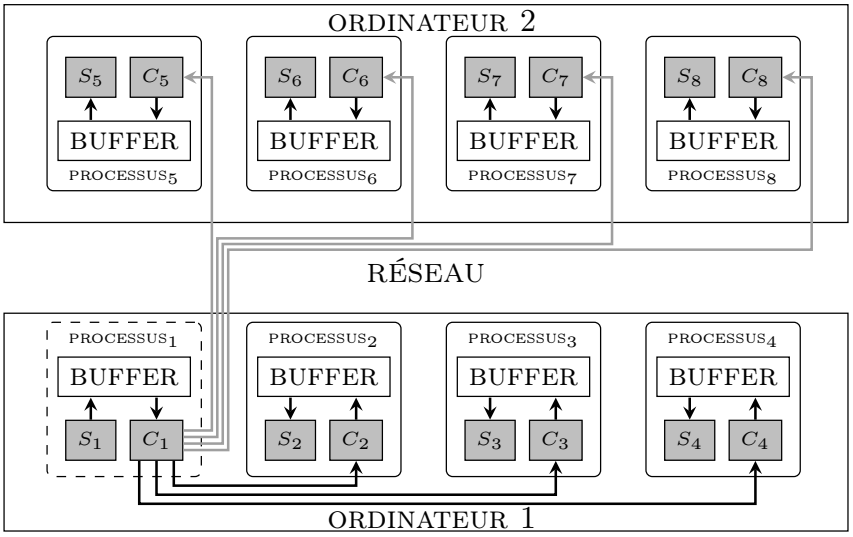


FIGURE 3.8 – Modèle de programmation pur par passage de messages : l'accent est réalisé sur les données (clauses apprises) envoyées par le solveur S_1 aux autres solveurs sur une configuration possédant deux ordinateurs.

comme un ordinateur indépendant. Même si deux processus solveurs s'exécutent sur le même ordinateur, il est nécessaire de réaliser une copie des données au lieu d'utiliser l'espace d'adressage global habituellement induit par la mémoire partagée (flèche noire). De plus, quand les données doivent être envoyées aux solveurs situés sur une autre machine, ce modèle exige d'envoyer un message à chaque solveur séquentiel (et donc à chaque processus) plutôt qu'un seul par machine (flèche grise). À titre d'exemple, dans la figure 3.8, lorsque le communicateur C_1 envoie une clause à la deuxième machine, celle-ci est envoyée 4 fois plutôt qu'une seule fois. De ce fait, sur notre exemple, le réseau est 4 fois plus encombré. Par conséquent, la même donnée est répliquée et envoyée sur le réseau autant de fois que le nombre de solveurs inclus dans la machine les réceptionnant, et cela conduit généralement à la congestion du réseau.

Afin de réduire le nombre de messages à envoyer, nous avons proposé une architecture de communication hybride qui réalise les communications par passage de message au niveau des ordinateurs. Dans ce cas, les clauses sont déposées par les solveurs dans une zone tampon et à chaque cycle de communication un *thread* communicateur est chargé de récupérer/envoyer les clauses. En fait, ce *thread* communicateur est utilisé comme une interface afin de partager les données entre des solveurs localisés sur des ordinateurs différents tandis que la mémoire partagée permet d'échanger les données entre les solveurs sur le même ordinateur. La figure 3.9 décrit ce schéma. Nous pouvons y voir comment le solveur S_1 communique ses clauses aux autres solveurs. Pour cela, le solveur S_1 envoie d'abord ses clauses apprises aux *threads* solveurs S_2, S_3, S_4 puis à son *thread* communicateur C_1 en utilisant le tampon localisé dans la mémoire partagée (BUFFER). Remarquons que les BUFFER du modèle de programmation pur par passage de messages et de ce modèle sont les mêmes au niveau de leurs implémentations, seule leur utilisation change. Par la suite, le communicateur C_1 envoie les clauses à travers le réseau au communicateur C_2. Pour finir, une fois que le communicateur C_2 a récupéré les clauses, il les distribue aux solveurs S_5, S_6, S_7 et S_8.

Le principal avantage de cette approche est la mutualisation des données échangées à la fois sur le réseau et via la mémoire partagée. L'hybridation permet d'utiliser proprement les deux

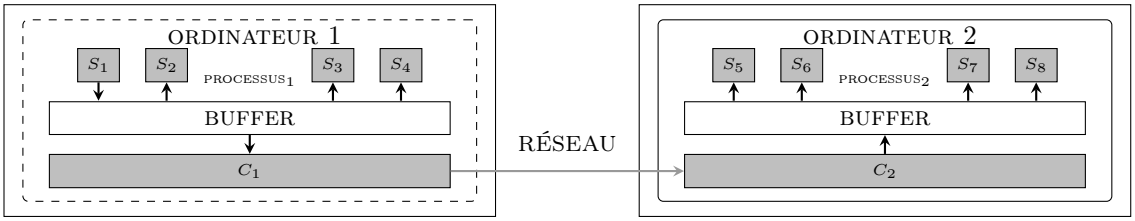


FIGURE 3.9 – Modèle de programmation partiellement hybride : une focalisation est réalisée sur les données (clauses apprises) envoyées par le solveur S_1 aux autres solveurs avec deux ordinateurs.

sortes d'échange de données en fonction de la localisation des *threads* (sur la même machine ou sur des machines différentes). Ainsi, ce modèle corrige le défaut le plus important du modèle de programmation pur par passage de messages.

Les deux architectures que nous venons de présenter s'appuient toutes les deux sur l'utilisation de cycles de communication. La question qu'il est important de se poser est : quelle doit être la durée d'un cycle de communication ? Dans (Audemard et al. 2017), nous avons démontré expérimentalement qu'il était souhaitable de partager les clauses le plus vite possible, c'est-à-dire qu'il faut éviter les cycles de communication trop longs. Cependant, nous avons aussi remarqué que, dû à la présence des sections critiques mises en place pour la récupération et l'envoi des clauses apprises, l'utilisation de cycles de communication très courts avait tendance à réduire considérablement l'efficacité du solveur. Ainsi, pour être efficace il a fallu définir une méthode qui soit capable de résoudre deux problèmes qui sont orthogonaux :

— réduire le nombre d'accès aux sections critiques ;

— envoyer les clauses aussi vite que possible.

Afin de réaliser cela, nous avons proposé une approche sans cycle de communication, qui envoie les clauses directement sur le réseau. Plus précisément, les *threads* de recherche font une partie du travail des *threads* communicateurs : l'envoi des clauses sur le réseau. Par conséquent, le *thread* communicateur d'un processus (de chaque ordinateur) devient un *thread* receveur puisque son seul travail est à présent de recevoir les clauses. Contrairement au modèle de programmation précédent, ce modèle de programmation permet à plusieurs *threads* de communiquer des données sur le réseau en même temps. Cette architecture est représentée sur la figure 3.10. Chaque ordinateur possède un seul processus qui intègre tous les *threads* de recherche et un *thread* receveur. Sur cette figure, le solveur S_1 envoie une clause aux solveurs S_2, S_3 et S_4 via la mémoire partagée (BUFFER) et au deuxième ordinateur via le réseau. Par la suite, le *thread* R_2 reçoit cette clause et l'envoie aux solveurs S_5, S_6, S_7 et S_8 via la mémoire partagée.

Notons bien que dans notre méthode, il n'y a pas de cycle de communication. À la place, les clauses sont envoyées une par une par les *threads* de recherche via une communication bloquante. Les *threads* receveurs demandent constamment aux solveurs s'ils n'ont pas de clauses à récupérer tant qu'une solution au problème n'a pas été trouvée. Lorsque qu'une clause est reçue, elle est directement placée dans le (BUFFER) en prenant soin de bloquer puis débloquer un *mutex*.

Théoriquement, un *thread* de recherche est bloqué lors de l'envoi d'une clause tant que cette dernière n'est pas reçue par les autres ordinateurs. Toutefois, en pratique, un tampon est utilisé afin de permettre aux solveurs d'envoyer des clauses sans attendre la fin de la communication. Ainsi, l'opération d'envoi est bloquée uniquement lorsque ce tampon est plein. Cela permet aux solveurs de se concentrer sur la recherche et pas sur l'envoi des clauses.

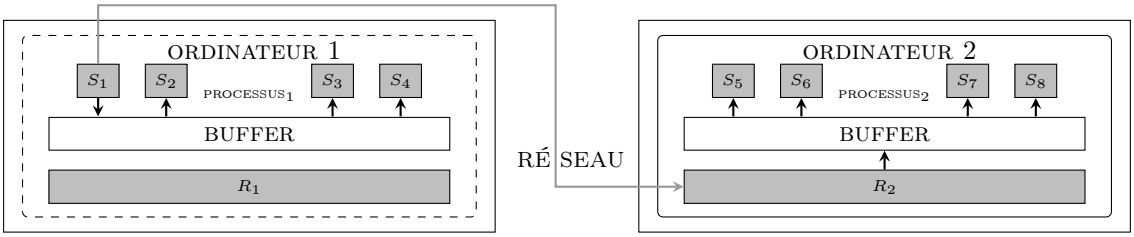


FIGURE 3.10 – Modèle de programmation complètement hybride : Une focalisation est réalisée sur les données (clauses apprises) envoyées par le solveur S_1 aux autres solveurs avec deux ordinateurs.

Le *thread* receveur de chaque ordinateur reçoit les clauses les unes après les autres. Par conséquent, ce *thread* effectue une attente active tant que le message n'a pas été reçu. L'attente active peut alors diminuer la rapidité des solveurs séquentiels associés. En effet, dans cette situation, le *thread* receveur lit dans le fichier descripteur du *socket* réseau tant qu'un message n'a pas été reçu. Le nombre d'opérations effectuées par le processeur peut alors ralentir les autres *threads*, notamment ceux dédiés à la recherche. Heureusement, nous avons observé que cette situation n'arrive quasiment jamais. En effet, le nombre de clauses à partager sur le réseau est gigantesque car elles proviennent de tous les *threads* de recherche et chacun d'entre eux peut produire un nombre de clauses apprises exponentiel par rapport à la taille de la formule initiale. Par conséquent, les *threads* receveurs ne font que recevoir des clauses et ne réalisent que très peu d'attente active. De plus, malgré son défaut d'efficacité, une attente active permet de recevoir les clauses plus rapidement qu'une communication ne la réalisant pas.

3.2.2.3 Discussion

Avec cette version distribuée de `Syrup`, nous avons pu nous rendre compte à quel point la gestion de l'échange de clauses était importante. En effet, en diminuant le nombre de messages envoyés nous avons réussi avec `d-Syrup` à augmenter la quantité d'informations échangées. Cependant, même si nous avons réalisé nos expérimentations avec 32 machines de 8 cœurs, il n'est pas clair qu'effectuer un partage de clauses avec toutes les unités de calcul soit une approche viable lorsque le nombre de machines augmente significativement. Cela peut d'une part conduire à un congestionnement du réseau, et d'autre part avoir un impact négatif sur les heuristiques des solveurs (les heuristiques du solveur `Glucose`, qui est utilisé comme base, dépendent fortement de la base de clauses apprises).

Il semble donc qu'il faille être plus précautionneux quant aux clauses échangées. Une première manière de résoudre ce problème pourrait être d'être plus regardant vis-à-vis des clauses échangées. Nous pourrions par exemple n'envoyer que les clauses qui ont été utilisées k fois, avec k défini en fonction du nombre de solveurs. Une autre manière de réduire le nombre de clauses échangées pourrait être d'utiliser des topologies d'échanges comme dans (Ehlers et al. 2014). Dans ce cas, nous pourrions rassembler certains solveurs en groupe, et faire communiquer les groupes ensemble en sélectionnant attentivement les clauses à échanger.

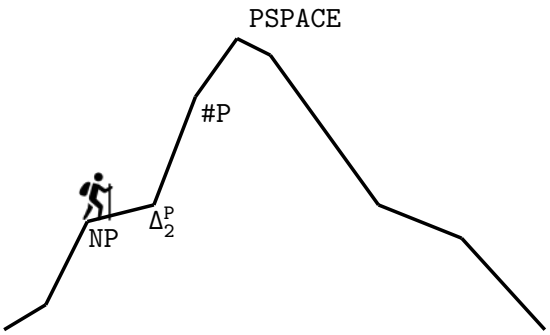
Nous pourrions aussi vouloir utiliser l'heuristique *rscm* définie dans le cadre du solveur `AmPharoS`. Cette mesure pourrait être utilisée afin de réguler l'échange d'informations. Néanmoins, il ne faut pas retomber dans les travers d'`AmPharoS` et centraliser toutes les clauses afin de calculer cette valeur. Nous pourrions simplement considérer les informations localement au

niveau d'un solveur, par exemple en choisissant de manière aléatoire l'un d'entre eux périodiquement. En réalisant assez de mesures nous pourrions être capables de prendre une photo assez fidèle du taux de redondances des clauses échangées.

Il serait également intéressant d'intégrer dans `d-Syrup` la possibilité d'utiliser une méthode de *preprocessing*. Par exemple, il est assez facile d'intégrer l'approche proposée par Luo et al. (2017) dans le solveur puisque cette dernière fonctionne avec les structures de `Glucose`.

Finalement, l'idée initiale derrière ces travaux était de comprendre pourquoi la version *portfolio* d'`AmPharoS` était inefficace. À présent que nous en avons identifié la cause, il serait intéressant d'intégrer le schéma de partage de clauses utilisé dans `d-Syrup` dans `AmPharoS`.

3.3 Conclusion



Nous voilà arrivés à la fin de notre première étape. Elle a consisté à travailler sur la résolution du problème `SAT` dans un contexte d'architectures à mémoire distribuée. Plus précisément, nous avons introduit le solveur `AmPharoS`, destiné à être exécuté sur de nombreux cœurs et qui est basé sur le paradigme « diviser pour mieux régner » s'appuyant sur une construction dynamique et collaborative d'un arbre de guidage. Ce solveur permet de partager deux sortes de clauses, les classiques clauses apprises et d'autres liées à la division du problème initial. Nous avons aussi proposé de mesurer à quel point les solveurs peuvent réaliser du travail redondant en comptant le nombre de clauses partagées et subsumées. Cette mesure nous permet d'ajuster dynamiquement la recherche afin de l'intensifier ou de la diversifier.

Lors de nos expérimentations, nous avons observé une anomalie concernant l'échange de clauses dans `AmPharoS`. Plus précisément, nous avons vu que dans la version concurrentielle d'`AmPharoS`, c'est-à-dire sans division en sous-problèmes, l'échange des clauses apprises avait un impact négatif sur les performances du solveur. En analysant le comportement du solveur, nous nous sommes aperçus que cela était dû à une congestion du réseau. En effet, `AmPharoS` partage les clauses d'une manière centralisée et cela implique un plus grand nombre de données à travers le réseau qu'un partage d'informations décentralisées.

Cette observation nous a fait prendre conscience de l'importance qu'il y avait à faire attention à la manière dont les clauses étaient échangées. Nous avons alors décidé d'étudier différents modèles de programmation parallèle pour choisir celui qui pourrait être le plus adapté pour échanger un nombre important de clauses. Ces travaux ont donné lieu à une implémentation distribuée du solveur parallèle multi-cœur `Syrup`, nommée `d-Syrup`. Cette étude nous a conduit à un schéma de partage d'informations qui utilise à la fois le passage de messages et la mémoire partagée. Ce schéma nous a permis d'exécuter `d-Syrup` sur 32 machines de 8 cœurs et nous

avons pu constater, que contrairement à AmPharoS, d-Syrup était capable de supporter la charge d'informations à échanger.

Les travaux que nous avons réalisés dans le cadre de la parallélisation de SAT ouvrent de nouvelles perspectives qui vont au-delà du problème SAT lui même. En effet, la méthode utilisée dans le cadre d'AmPharoS pourrait facilement être mise en pratique pour d'autres problèmes qui peuvent être résolus en utilisant une approche de type recherche en profondeur. Par exemple, il serait tout à fait envisageable d'étendre le schéma algorithmique d'AmPharoS pour résoudre le problème de satisfaction de contraintes (CSP) en parallèle. Afin de promouvoir notre approche, il faudra élaborer une librairie permettant la gestion des communications entre les unités de calcul, ainsi que la création et la gestion de l'arbre de guidage. Pour ce qui est de la gestion de l'arbre de guidage du côté du MANAGER, il faudra bien entendu définir des algorithmes dédiés au problème que nous souhaitons traiter afin de pouvoir appliquer les bonnes règles d'élagage et de gestion des littéraux unitaires sous hypothèses.

Nos travaux sur l'échange d'informations dans le contexte d'une architecture à mémoire distribuée nous ont permis de voir à quel point les problématiques liées à l'échange de clauses constitueraient un domaine d'étude à part entière. En effet, afin de produire une solution efficace à ce problème, il est nécessaire d'avoir une connaissance des architectures à mémoire distribuée très poussée. Cela nous conduit à penser qu'il serait intéressant de formaliser ce problème, de définir un protocole expérimental pour son étude, et d'en faire part à la communauté scientifique qui travaille sur ce sujet.

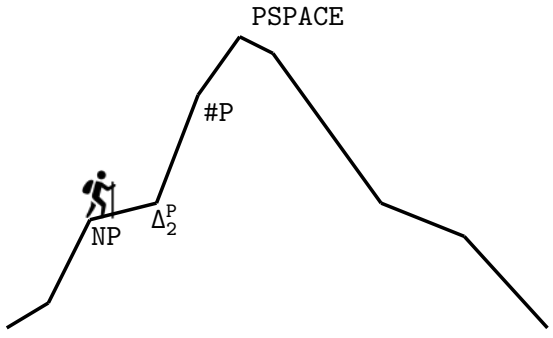
Une question importante que nous n'avons pas traitée dans le cadre de nos travaux concerne la certification de nos solveurs (Wetzler et al. 2014). Comme l'ont démontré Heule et Kullmann (2017) dans leurs travaux sur la résolution de la conjecture sur les triplets pythagoriciens, il ne suffit pas de répondre « le problème a une solution » ou « le problème n'a pas de solution » pour être satisfait. Dans le cas où le problème est satisfaisable, le certificat est très simple puisqu'il suffit de retourner un modèle que nous pourrons ensuite vérifier en temps polynomial. Cependant, dans le cas où le problème n'a pas de solution, à part si NP = coNP, il n'y a pas de certificat « court ». Dans cette situation, il est nécessaire de sauvegarder l'ensemble des clauses générées par résolution afin d'être en mesure de rejouer la preuve *a posteriori*. Néanmoins, dans le contexte qui nous intéresse, les solveurs échangent énormément de clauses et il est pour le moment impossible de retrouver la provenance de celles-ci. Afin de résoudre ce problème, une solution pourrait être de conserver pour chaque solveur une explication pour chaque clause qu'il a envoyée. Puis de marquer toutes les clauses afin que chaque solveur soit en mesure de connaître leurs provenances. Une des principales difficultés est qu'il sera nécessaire de sauvegarder une quantité élevée d'informations au niveau des solveurs, ce qui peut conduire à la longue à un dépassement de la capacité de la mémoire allouée. Le gant est jeté !

Cette première partie de voyage nous a conduit au cœur des solveurs CDCL. Dans la suite de notre aventure, nous allons tirer parti de cette expérience afin d'utiliser les solveurs SAT comme des oracles pour la résolution de problèmes plus complexes. Plus précisément, nous étudierons dans le chapitre suivant le cas particulier où un solveur SAT doit être exécuté de nombreuses fois sur un problème qui peut évoluer. Nous verrons en particulier comment utiliser ce type de solveur pour traiter le problème de calcul et d'énumération d'ensembles maximaux cohérents.

Chapitre 4

Contributions au calcul d'ensembles maximalement cohérents

Sommaire



Nous voilà arrivés au premier niveau, le chapitre précédent nous a permis d'appréhender de manière pratique le problème NP-complet de référence, c'est-à-dire le problème de satisfaisabilité propositionnelle SAT. Ce que nous avons appris lors de ce début d'ascension est que, même si SAT est un problème NP-complet, il est possible de modéliser et de résoudre de nombreux problèmes

pratiques en logique propositionnelle (Biere et al. 1999a, Argelich et al. 2010, Le Berre et Rapicault 2009, Kautz et Selman 1996, Caridroit et al. 2017, Heule et Kullmann 2017, Belahcène et al. 2018). Le fait de pouvoir résoudre des problèmes pratiques contenant des millions de variables et de clauses permet d'envisager l'utilisation des solveurs SAT comme des oracles pour attaquer des problèmes calculatoirement plus difficiles. Avant d'essayer d'attaquer des problèmes nécessitant un nombre exponentiel d'appels à un tel oracle NP, vérifions que notre bâton de marche est assez solide et essayons de résoudre des problèmes nécessitant un nombre polynomial d'appels à un oracle. Si nous nous remémorons les classes de complexité présentées dans le premier chapitre, nous voyons que la classe de complexité $\Delta_2^{\mathtt{P}} = \mathtt{P}^{\mathtt{NP}}$ est une bonne cible.

Afin d'avancer sereinement il est nécessaire de se fixer un objectif clair. En effet, la classe de complexité $\Delta_2^{\mathtt{P}}$ est un ensemble de langages permettant de caractériser des problèmes en fonction de leur difficulté intrinsèque, et, dans ce sens, elle ne peut pas être un objectif en soi. Étant donné la culture du CRIL autour de la problématique de restauration de la cohérence dans le cadre de la logique propositionnelle, c'est naturellement que je me suis orienté dans cette voie. Plus précisément, j'ai travaillé en collaboration avec Éric Grégoire, Yacine Izza et Bertrand Mazure à l'élaboration d'outils d'extraction de MCSes (*Minimal Correction Subsets*). En effet, le calcul de MCS peut être réalisé *via* un nombre polynomial d'appels à un oracle NP.

Le concept de MCS joue depuis longtemps un rôle important dans de nombreuses applications de l'I.A. basées sur la logique. En particulier, dans le domaine de la représentation des connaissances, le raisonnement à partir de bases incohérentes est souvent réalisé à l'aide de MCSes (Fermé et Hansson 2011, Grégoire et Konieczny 2006, Besnard et Hunter 2008, Ginsberg 1987, Lagniez et al. 2015). Ce concept est aussi utilisé pour le diagnostic et le débogage (Hamscher et al. 1992, Pereira et al. 1993, Felfernig et al. 2012, Marques-Silva et al. 2015, Roychoudhury et Chandra 2016), la fouille de données (Jabbour et al. 2014) ou encore pour expliquer des réseaux de neurones (Ignatiev et al. 2019). C'est autour de cette notion de MCS que s'inscrit une partie des travaux que nous présentons dans ce chapitre. Plus précisément, nous détaillons une approche pour scinder une formule booléenne CNF insatisfaisable en un MSS (*Maximal Satisfiable Subset*) et un MCS qui, contrairement aux autres approches de l'état de l'art, s'appuie sur un algorithme destructif. Nous verrons aussi comment cette approche de base peut être améliorée en considérant un certain nombre d'optimisations optionnelles.

Lors de notre voyage vers $\Delta_2^{\mathtt{P}}$, nous nous sommes aussi intéressés à l'amélioration des solveurs SAT dans un contexte où plusieurs appels étaient à envisager. En effet, les problèmes de la classe de complexité $\Delta_2^{\mathtt{P}}$ ont la particularité de pouvoir être décidés par un algorithme réalisant un nombre polynomial d'appels à un oracle NP. De manière générale, les différents appels ne sont pas forcément indépendants, et il est donc généralement possible d'utiliser des informations venant du solveur afin d'accélérer le traitement. Par exemple, dans le cas où l'objectif est de calculer un ensemble minimal (comme le problème d'extraire un MCS) ce nombre d'appels peut diminuer jusqu'à être une fonction logarithmique dépendant du nombre d'objets initial (Marques-Silva et al. 2013). Au-delà des aspects théoriques, une des particularités des méthodes d'extraction de MCSes tient au fait que l'oracle est appelé sur des formules très proches les unes des autres. Ainsi, la proximité des différents problèmes laisse entrevoir la possibilité de réutiliser les informations pouvant être collectées entre les appels successifs au démonstrateur. C'est dans ce contexte que s'inscrivent les travaux que j'ai réalisés en collaboration avec Gilles Audemard, Armin Biere et Laurent Simon sur l'amélioration des solveurs SAT dans un cadre incrémental. Nous verrons dans ce chapitre que dans cette quête de réutilisation, il est nécessaire d'ajouter des informations aux clauses lors du processus d'apprentissage, ce qui peut conduire à dérégler certaines heuristiques et de manière générale à dégrader les performances des solveurs SAT. Nous présenterons alors

deux méthodes pouvant être greffées aux solveurs SAT de l'état de l'art afin d'éviter cet écueil.

Dans le cadre de ce chapitre, nous nous intéressons aussi au problème d'énumération de MCSes. Ce problème sort un peu des limites du cadre fixé en commençant ce chapitre. En effet, le problème d'énumérer l'ensemble des MCSes est beaucoup plus difficile que celui de calculer un seul MCS. La raison à cela est assez simple : il peut y avoir un nombre exponentiel de MCS dans une formule insatisfaisable. Néanmoins, ce petit détour nous permet d'avoir un premier aperçu de la difficulté des problèmes de comptage qui seront présentés au chapitre suivant. En effet, il est clair qu'une méthode naïve qui consisterait à énumérer et à bloquer chaque modèle d'une formule afin d'en compter le nombre n'aurait que très peu de chance de fonctionner en pratique. Cela est aussi vrai pour le problème d'énumération de MCSes, une approche qui consisterait à calculer chaque MCS indépendamment des autres n'aurait que très peu de chance d'être viable en pratique. Cependant, c'est ainsi que fonctionnent les approches de l'état de l'art. Bien que certaines approches proposent de sauvegarder certaines informations afin d'éviter des appels au solveur SAT, les méthodes de l'état de l'art sont généralement basées sur une procédure incrémentale qui consiste à calculer un MCS, à ajouter une clause permettant de le bloquer et à répéter ce processus tant que la formule produite est satisfiable. Afin d'éviter de calculer un à un chaque MCS, et en collaboration avec Éric Grégoire et Yacine Izza, nous avons proposé une approche permettant de factoriser certains MCSes. Plus précisément, notre méthode utilise la notion de rotation de modèles, introduite dans (Belov et Marques-Silva 2011), afin d'extraire un ensemble de clauses de transition pouvant être utilisées de manière à que chacune d'entre elles puisse être le point de départ d'une famille de MCSes.

Ce chapitre est composé de cinq sections. Dans la première section, nous introduisons le concept de MCS et nous présentons rapidement les méthodes utilisées en pratique afin d'extraire ou d'énumérer les MCSes d'une formule. Ensuite, nous présentons nos contributions à la résolution incrémentale du problème SAT. Plus précisément, nous présentons différentes améliorations ainsi qu'une méthode de compression de clauses pouvant être greffée dans un solveur SAT de l'état de l'art de manière à en améliorer les performances quand il doit être utilisé plusieurs fois pour résoudre des instances « proches ». Dans la troisième section, un nouvel algorithme destructif d'extraction de MCS est présenté. La section quatre introduit un nouveau processus permettant d'accélérer l'énumération de MCSes en considérant des familles de MCSes. Nous terminons ce chapitre par une conclusion et en donnant quelques perspectives.

4.1 Identification et réparation de l'incohérence

Dans le cas où une instance CNF est satisfaisable, une solution peut être exhibée afin de certifier la satisfaisabilité de la formule. En revanche, lorsque la formule est insatisfaisable, aucun certificat permettant d'expliquer les raisons de ce résultat n'est généralement fourni. Extraire d'une formule insatisfaisable les raisons de sa non-satisfaisabilité peut se révéler utile dans de nombreux domaines (Hunter et Konieczny 2008, Benferhat et al. 2005, Besnard et al. 2010, Belahcène et al. 2018). Une telle information, de nature explicative, peut être fournie par la localisation de sous-formules minimalement insatisfaisables (ou MUS pour « *Minimally Unsatisfiable Subformulae*). Un tel ensemble est défini formellement de la manière suivante :

Définition 48 (CORE,MUS)

Soient Σ une formule CNF et Γ un sous-ensemble de Σ, nous avons :

- Γ est un CORE de Σ si et seulement si Γ est insatisfaisable ;
- Γ est un MUS de Σ si et seulement si Γ est un core de Σ et $\forall \alpha \in \Gamma$ nous avons $\Gamma \setminus \{\alpha\}$ est satisfaisable.

Lorsqu'un ensemble de clauses Σ est incohérent, il est toujours possible d'en extraire un sous-ensemble qui soit satisfaisable. Un tel sous-ensemble, lorsqu'il est maximal pour l'inclusion (c'est-à-dire qu'il n'est pas possible de l'étendre sans le rendre insatisfaisable), est appelé MSS (*Maximal Satisfiable Subset*) de Σ. Son complémentaire, noté MCS (*Minimal Correction Subset*), désigne un sous-ensemble minimal de clauses à retirer de Σ pour obtenir un MSS. Ces deux notions jouent un rôle central dans de nombreuses applications en intelligence artificielle (Fermé et Hansson 2011, Grégoire et Konieczny 2006, Besnard et Hunter 2008, McCarthy 1980, Ginsberg 1987, Hamscher et al. 1992, Pereira et al. 1993, Felfernig et al. 2012, Marques-Silva et al. 2015, Ignatiev et al. 2019). Formellement, nous avons :

Définition 49 (MSS/MCS)

Soit Σ une formule CNF, nous avons :

- $\Pi \subseteq \Sigma$ est un MSS de Σ si et seulement si Π est satisfaisable et $\forall \alpha \in \Sigma \setminus \Pi$ la formule $\Pi \cup \{\alpha\}$ est insatisfaisable ;
- $\Psi \subseteq \Sigma$ est un MCS de Σ si et seulement si $\Sigma \setminus \Psi$ est un MSS de Σ.

Clairement, extraire un MSS est proche du problème MaxSAT qui consiste à extraire un MSS de cardinal maximal. Chaque solution MaxSAT est un MSS, mais l'inverse n'est pas toujours vrai (un MSS n'est pas nécessairement de cardinal maximal). Comme le préconisent Marques-Silva et al. (2013), des algorithmes spécifiques pour calculer un MSS peuvent se révéler plus rapides que ceux dédiés à chercher une solution MaxSAT. À cet égard, le calcul d'un MSS peut être utilisé pour fournir une solution approchée à MaxSAT (Marques-Silva et al. 2013, Bacchus et al. 2014, Mencía et al. 2015).

Les notions de MSS et MCS peuvent être étendues respectivement aux concepts de Partial-MSS et Partial-MCS pour le cas où il est nécessaire de considérer des contraintes d'intégrité. Dans ce cas, l'instance Σ est constituée d'un couple $\langle \Sigma_1, \Sigma_2 \rangle$, où Σ_1 est l'ensemble des clauses dites « dures » de Σ et Σ_2 est l'ensemble des clauses « souples » de Σ. Les clauses dures appartiennent à tous les Partial-MSSes de Σ, tandis qu'elles n'appartiennent à aucun Partial-MCS de Σ. Formellement nous avons :

Définition 50 (Partial-MSS, Partial-MCS)

Soit la formule CNF $\Sigma = \langle \Sigma_1, \Sigma_2 \rangle$, nous avons :

— Π est un Partial-MSS de Σ si et seulement si $\Sigma_1 \subseteq \Pi$ et Π est un MSS de $\Sigma_1 \cup \Sigma_2$;

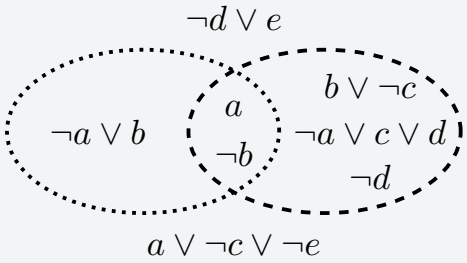
— Ψ est un Partial-MCS de Σ si et seulement si $\Psi \subseteq \Sigma_2$ et Ψ est un MCS de $\Sigma_1 \cup \Sigma_2$.

Notons que lorsque Σ_1 est insatisfaisable alors il n'existe aucun Partial-MSS pour Σ. De plus, chaque Partial-MCS de $\langle \Sigma_1, \Sigma_2 \rangle$ est un MCS de $\Sigma_1 \cup \Sigma_2$. Inversement, tout MCS Ψ de $\Sigma_1 \cup \Sigma_2$ qui ne satisfait pas $\Psi \subseteq \Sigma_2$ n'est pas un Partial-MCS de $\langle \Sigma_1, \Sigma_2 \rangle$.

Les concepts de MCS et de MUS sont des concepts duaux. Plus précisément, un MUS peut être calculé à partir d'un ensemble intersectant (« *hitting set* ») de l'ensemble des MCS d'une formule puisque chaque MCS contient au moins une clause de chaque MUS. Cette dualité a souvent été exploitée pour calculer les MUSes et/ou les MCSes (Reiter 1987, Grégoire et al. 2007, Liffiton et Sakallah 2008, Nadel et al. 2014, Bacchus et Katsirelos 2015; 2016, Previti et Marques-Silva 2013, Liffiton et al. 2016, Narodytska et al. 2018).

Exemple 16

Soit $\Sigma = \{\neg d \vee e,\ b \vee \neg c,\ \neg d,\ \neg a \vee b,\ a,\ a \vee \neg c \vee \neg e,\ \neg a \vee c \vee d,\ \neg b\}$ une formule CNF insatisfaisable composée de 8 clauses construites sur 5 variables. La figure suivante schématise, sous forme d'un diagramme de Venn, la formule et ses deux MUSes :



Cette formule contient 5 MCSes qui sont :

$$\{a\}, \{\neg b\}, \{\neg a \vee b, b \vee \neg c\}, \{\neg a \vee b, \neg a \vee c \vee d\}, \{\neg a \vee b, \neg d\}$$

Le problème de décision qui consiste à déterminer si un ensemble de clauses est un MSS d'une instance CNF donnée est DP-complet (Chen et Toda 1995). Et le problème qui consiste à calculer un MCS appartient à $\mathtt{F}\Delta_2^{\mathtt{P}} = \mathtt{FP}^{\mathtt{NP}}$ (Papadimitriou 1994). De plus, comme l'illustre l'exemple précédent, une formule peut contenir plusieurs MCSes. En fait, elle peut même en avoir un nombre exponentiel en sa taille. En effet, une CNF de n clauses peut contenir jusqu'à $C_n^{\frac{n}{2}}$ MCSes dans le pire cas. Ce nombre potentiellement élevé de MCS au sein d'une même formule rend les problèmes nécessitant la prise en compte de tous les MCSes/MUSes d'une complexité algorithmique élevée. Par exemple, tester si un ensemble de clauses fait partie de l'ensemble des MUS d'une instance

est un problème $\Sigma_2^{\mathtt{P}}$-difficile (Eiter et Gottlob 2002). Malgré le coût élevé de ce calcul dans le pire cas, plusieurs approches pour extraire ou énumérer des `MCSes` ont été proposées et se sont montrées viables pour de nombreuses instances (Liffiton et Sakallah 2008, Marques-Silva et al. 2013, Felfernig et al. 2012, Previti et al. 2018).

Généralement, les algorithmes qui cherchent à extraire un `MCS` ou un `MUS` d'une formule `CNF` sont basés sur le concept de *clause de transition*.

Définition 51 (Clause de transition)

Soit Σ une formule `CNF` insatisfaisable. Une clause $\alpha \in \Sigma$ est une clause de transition de Σ si et seulement si $\Sigma \setminus \{\alpha\}$ est satisfaisable.

Le concept de clause de transition est fondamental dans de nombreuses approches d'extraction de `MCS`. En effet, si $\alpha \in \Sigma$ est une clause de transition alors α appartient à tous les `MUS` de Σ. L'algorithme `BLS` (*Basic Linear Search*) décrit en 4.1 s'appuie sur ce principe de clause de transition pour identifier les clauses du `MCS` à retourner. Il débute par initialiser l'ensemble des clauses à tester U de Σ, et le `MSS` à construire Π à $\varnothing$ (ligne 1). Ensuite, tant que U n'est pas vide, l'algorithme retire une clause α de U et teste si $\Pi \cup \{\alpha\}$ est satisfaisable. Dans le cas vrai, α est ajouté dans le `MSS` Π en construction. Sinon, α est une clause de transition du `MSS` en construction et elle sera nécessairement dans le `MCS` retourné. À la fin de la boucle, toutes les clauses de la formule ont été parcourues. Un `MCS` $\Sigma\backslash\Pi$, le complémentaire du `MSS` Π, est fourni en sortie. Notons que l'algorithme `BLS` nécessite dans le pire cas $\mathcal{O}(n)$ appels au solveur `SAT`.

Algorithme 4.1 : `BLS` (*Basic Linear Search*)

Input : une formule `CNF` Σ
Output : un `MCS` de Σ

1 $(U, \Pi) \leftarrow (\Sigma, \varnothing)$;
2 **while** $U \neq \varnothing$ **do**
3 $\quad \alpha \leftarrow$ **choisir** $\alpha \in U$;
4 $\quad U \leftarrow U \backslash \{\alpha\}$;
5 $\quad (res,\ \mu) \leftarrow \mathrm{SAT}(\Pi \cup \{\alpha\})$;
6 $\quad$ **if** $res = true$ **then** $\Pi \leftarrow \Pi \cup \{\alpha\}$
7 **return** $\Sigma\backslash\Pi$;

De nombreuses améliorations ont été proposées par Marques-Silva et al. (2013) pour l'algorithme `BLS`. Ces dernières sont à la base de l'approche `ELS` (*Enhanced Linear Search*), qui inclut trois techniques permettant de réduire le nombre d'appels à l'oracle `SAT` et aussi de simplifier la recherche d'une solution. La première de ces améliorations consiste à considérer indépendamment des `COREs` disjoints. Plus précisément, les auteurs proposent de partitionner la formule Σ en $\langle \Pi, \{U_1, \ldots, U_n\} \rangle$ tel que Π est satisfaisable et U_i est un `CORE` de Σ. Ensuite, l'extraction d'un `MCS` se fait en considérant incrémentalement chaque U_i et en calculant un `MCS` Ψ de $\Pi \cup U_i$. Tant qu'il existe un `CORE` non considéré, Π est mis à jour en ajoutant les clauses de $U_i \setminus \Psi$ et la procédure passe au `CORE` suivant. La seconde amélioration consiste à ajouter les littéraux que

nous savons impliqués par le MSS en construction. Cette technique s'appuie sur l'observation suivante : lorsque la clause α est prouvée clause de transition du MSS Π ($\Pi \cup \{\alpha\} \models \bot$), nous avons $\Pi \models \neg\alpha$, et pour chaque littéral $\ell_i \in \alpha$, $\Pi \models \tilde{\ell_i}$. Ainsi, $\tilde{\ell_i}$ est un littéral impliqué par Π et il peut être ajouté lors de, prochains appels à l'oracle. La dernière amélioration consiste à exploiter les modèles retournés par le solveur. Cette technique, initialement proposée dans (Nöhrer et al. 2012), consiste à déplacer un maximum de clauses vers le MSS en construction. Pour ce faire, lorsque la clause α est satisfaisable avec Π, le modèle μ retourné par le solveur est utilisé afin de transférer dans Π les clauses qui sont aussi satisfaites dans Π.

En plus de ces améliorations, Marques-Silva et al. (2013) proposent une approche nommée CLD (*Computing one MCS with Clause D*). Cette approche s'inspire d'une propriété proposée par Birnbaum et Lozinskii (2003) et qui permet de considérer conjointement les clauses de la partie incohérente, au lieu d'évaluer les clauses une à une, pour vérifier si ces dernières font partie du MCS en construction. Concrètement, l'idée de CLD consiste à appeler le solveur SAT sur la formule $\Pi \cup \{D\}$ telle que Π est le MSS à augmenter et D est la disjonction de tous les littéraux appartenant à la partie incohérente $\Sigma \backslash \Pi$. Si le solveur retourne une interprétation qui satisfait $\Pi \cup \{D\}$, alors les clauses de $\Sigma \backslash \Pi$ qui sont cohérentes avec cette interprétation sont ajoutées dans Π (il est facile de voir qu'au moins une clause sera ainsi transférée). Lorsque $\Pi \cup \{D\}$ n'admet plus de modèle alors $\Pi \models \neg D$. Ainsi, il est impossible de satisfaire les clauses de $\Sigma \backslash \Pi$ sans contredire Π et donc Π est un MSS de Σ.

Cette idée de chercher à calculer l'ensemble des littéraux impliqués (*backbone*) d'une sous-formule afin d'extraire un MCS est exploitée plus finement dans l'extracteur de MCSes LBX (*Literal Based eXtractor*) proposé par Mencía et al. (2015). En effet, il est facile de voir que α est une clause du MCS $\Sigma \backslash \Pi$ si et seulement si pour tout littéral $\ell \in \alpha$, nous avons que $\Pi \models \tilde{\ell}$, c'est-à-dire, $\tilde{\ell}$ est un littéral du *backbone* de Π. En partant de cette propriété, les auteurs de (Mencía et al. 2015) proposent un algorithme consistant à détecter itérativement les littéraux du *backbone* de la partie MSS.

Algorithme 4.2 : LBX (*Literal Based eXtractor*)

input : une formule CNF Σ
output : un MCS de Σ

```
1  (Π, U) ← approximatePartition(Σ);
2  L ← literals(U);                          // L est un ensemble de littéraux.
3  Φ ← ∅;                                    // Φ est un ensemble de littéraux du backbone.
4  while L ≠ ∅ do
5  |  ℓ ← RemoveLiteral(L);
6  |  (res, μ) ← SAT(Φ ∪ Π ∪ {ℓ});
7  |  if res = true then
8  |  |  Π ← Φ ∪ {α ∈ U | ℓ ∈ α};
9  |  |  U ← {α ∈ U | μ(α) = 0};
10 |  |  L ← L ∩ literlas(U);
11 |  else Φ ← Φ ∪ {ℓ̃}
12 return Σ\Π;
```

Le schéma algorithmique de l'approche LBX est présenté dans l'algorithme 4.2. Initialement, l'algorithme commence par calculer une bi-partition (Π, U) des clauses de Σ de telle sorte que $\Pi \subseteq \Sigma$ est satisfaisable et $U = \Sigma \backslash \Pi$. Ensuite, deux ensembles $\mathcal{L}$ et Π, constituant respectivement

l'ensemble des littéraux des clauses de U et les littéraux du *backbone* de Π, sont initialisés. Puis, tant que l'ensemble $\mathcal{L}$ n'est pas vide, un littéral ℓ de $\mathcal{L}$ est sélectionné et un appel à un solveur SAT est réalisé sur $\Pi \cup \Phi \cup \{\ell\}$. Soulignons que Φ n'altère pas le résultat de l'appel au solveur mais permet simplement de simplifier la résolution à l'intérieur du solveur. Si la réponse est SAT, alors les ensembles Π et U sont mis à jour à partir du modèle calculé par le solveur (lignes $8-9$) et $\mathcal{L}$ est calculé à nouveau. Sinon, $\neg\ell$ est inséré dans l'ensemble des littéraux du *backbone* Φ. L'algorithme sort de la boucle lorsque ($\mathcal{L} = \varnothing$), et le MCS final $\Sigma \backslash \Pi$ est retourné. La complexité de LBX est en $\mathcal{O}(|\mathcal{V}|)$ où $\mathcal{V}$ est l'ensemble des variables de l'instance. Enfin, notons que LBX est actuellement la meilleure approche de l'état de l'art pour l'extraction d'un MCS.

La puissance des solveurs SAT modernes en combinaison avec les approches présentées précédemment a permis de traiter des problèmes venant d'applications réelles (Biere et al. 2009). Cependant, l'énumération des MCSes représente un défi calculatoire encore plus important. En effet, en plus de la complexité d'extraire un MCS (Δ_2^P), dans le pire des cas une instance CNF insatisfaisable peut posséder un nombre exponentiel de MCSes. Néanmoins, en pratique ce nombre est souvent relativement bas, permettant ainsi l'énumération complète des MCSes en s'appuyant sur le calcul d'un MCS. Plus précisément, l'algorithme de base d'énumération de MCSes consiste à calculer de manière itérative un MCS de l'instance CNF donnée et puis à « bloquer » ce MCS par une nouvelle clause générée à partir de la disjonction des littéraux des clauses de ce dernier. La clause bloquante assure qu'au moins une des clauses du MCS sera dans les MSSes lors des énumérations suivantes. L'algorithme 4.3 décrit un tel mécanisme, où les clauses bloquantes sont insérées dans Δ (ligne 6). Il termine lorsque ($\Sigma^S \cup \Delta$) devient UNSAT, dans ce cas tous les MCSes de Σ ont été énumérés.

Algorithme 4.3 : Enum-MCSes (Enumerate all MCSes)

input : une formule CNF Σ
output : tous les MCSes de Σ

1 $\Sigma^S \longleftarrow \{\alpha \vee \neg s_\alpha \mid \alpha \in \Sigma\}$; // avec s_α de nouvelles variables
2 $\Delta \leftarrow \varnothing$;
3 **while** SAT($\Sigma^S \cup \Delta$) = *vrai* **do**
4 $\quad \Gamma \leftarrow$ ExtractMCS($\Sigma^S \cup \Delta$);
5 $\quad$ *output*(Γ);
6 $\quad \Delta \leftarrow \Delta \cup (\bigvee_{s_\alpha \in \Gamma} s_\alpha)$; // clauses bloquantes

Clairement, l'efficacité de cet algorithme dépend fortement des performances de la méthode (Extract-MCS) utilisée pour chercher un MCS à chaque itération. À cet égard, (Previti et al. 2017) proposent d'utiliser un mécanisme de mémorisation (*caching*) dans l'algorithme de recherche linéaire ELS pour le calcul d'un MCS dans le but d'améliorer l'énumération des MCSes. L'idée consiste à sauvegarder dans une base (*cache*) les COREs retournés par le solveur SAT durant le calcul successif des différents MCSes. Ainsi, avant de vérifier si un ensemble de clauses est satisfaisable, une recherche dans le cache est effectuée. Si un CORE du cache est inclus dans la sous-formule que nous souhaitons tester, alors il n'est pas nécessaire d'appeler le solveur car le problème a déjà été montré insatisfaisable.

Comme nous pouvons l'observer, le calcul de MCSes s'appuie sur une utilisation répétée d'un solveur SAT. Dans la suite, nous présentons deux approches pour améliorer les performances des solveurs dans ce contexte.

4.2 Améliorer SAT dans le cadre incrémental

Depuis l'avènement des solveurs CDCL, des problèmes de taille de plus en plus conséquente (des millions de clauses et de variables) sont résolus quotidiennement par les approches s'appuyant sur SAT. Ainsi, ce gain de performance laisse envisager la possibilité de traiter, en pratique, à l'aide de solveurs SAT, des problèmes au-delà de NP, qui nécessitent plusieurs appels à un oracle SAT (Biere et al. 1999b, Velev et Bryant 2003).

Ainsi, depuis quelques années, une nouvelle utilisation des démonstrateurs SAT, appelée « SAT incrémental », a vu le jour afin de répondre à ce nouveau type de besoins (Schiex et Verfaillie 1993a, Eén et Sörensson 2003, Fu et Malik 2006, Nadel 2010, Bradley 2012, Soh et al. 2014, Glorian et al. 2018, Lagniez et al. 2018b). Puisque les solveurs CDCL se basent essentiellement sur le passé pour avancer dans la recherche au moyen des heuristiques, des redémarrages dynamiques et surtout de l'apprentissage de clauses, il parait assez clair que, pour viser les meilleures performances possibles, il faut laisser le démonstrateur SAT *en vie* entre deux appels successifs sur ces formules « relativement proches » et lui donner la possibilité d'ajouter des variables et d'ajouter et/ou supprimer des clauses. La figure 4.1 montre le fonctionnement général d'un démonstrateur SAT incrémental.

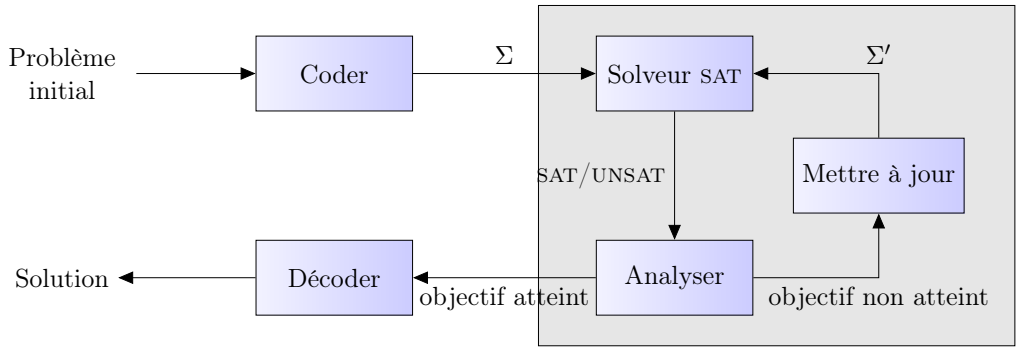


FIGURE 4.1 – Fonctionnement général d'un solveur SAT incrémental

Parmi ces applications, il n'est pas rare que l'activation et la suppression de clauses soient nécessaires. Dans ce contexte, les démonstrateurs SAT ne sont pas lancés une seule fois sur des formules potentiellement énormes, mais peuvent être appelés des milliers de fois sur des instances « proches » les unes des autres, avec quelques clauses ajoutées et/ou supprimées à chaque appel. Ainsi, la proximité des différents problèmes laisse entrevoir la possibilité de réutiliser au maximum les informations pouvant être collectées entre les appels successifs au démonstrateur. Il faut cependant faire attention, car il est clair que si des clauses sont supprimées d'un appel à l'autre, les informations ayant été dérivées (clauses apprises) à partir de ces dernières ne peuvent plus être réutilisées. Pour pallier ce problème, il est nécessaire d'ajouter un mécanisme d'utilisation particulier au démonstrateur SAT. Ce mécanisme est l'utilisation de littéraux nouveaux, appelés *hypothèses*, et qui vont contrôler les clauses ajoutées ou supprimées tout en permettant d'identifier les dépendances logiques devant être éventuellement supprimées. Ces hypothèses sont donc des littéraux spéciaux ajoutés à chaque clause et qui sont affectés avant le début de la recherche.

4.2.1 SAT incrémental avec ajouts/suppressions de clauses

L'ajout de nouvelles clauses et de nouvelles variables sont des opérations extrêmement faciles à réaliser et sont nativement incluses dans tous les démonstrateurs CDCL. La suppression de

clauses initiales est plus complexe. Lorsqu'une clause est supprimée de la formule courante, il n'est pas possible de réutiliser directement les clauses apprises générées à l'aide de cette dernière. Pour pallier ce problème, il est possible d'ajouter des hypothèses aux démonstrateurs. Celles-ci sont caractérisées par un ensemble $\mathcal{A}$ de littéraux qui sont choisis comme variables de décisions et affectés à vrai avant toute autre variable de décision. Il existe donc un niveau de décision par hypothèse, la recherche ne commençant réellement que lorsque toutes ces hypothèses ont été affectées et nous avons alors comme premier niveau de décision pour la recherche proprement dite $|\mathcal{A}| + 1$.

Quand une hypothèse est utilisée pour activer/désactiver une clause, celle-ci doit alors être associée à une nouvelle variable (s_i) du problème Σ, qui est appelée *sélecteur* pour cette clause. La variable s_i est alors ajoutée à la clause α_i de telle sorte que la clause α_i soit remplacée dans Σ par $\neg s_i \vee \alpha_i$.

Exemple 17

Considérons la formule `CNF` Σ suivante :

$$\begin{array}{lll} x \vee y \vee z & x \vee \neg y & x \vee \neg z \\ \neg x \vee y \vee z & x \vee w & w \vee z \vee \neg y \\ \neg x \vee \neg y & \neg x \vee \neg z & w \vee \neg x \vee \neg z \end{array}$$

L'ajout des sélecteurs produit la formule Σ' suivante :

$$\begin{array}{lll} \neg s_1 \vee x \vee y \vee z & \neg s_2 \vee x \vee \neg y & \neg s_3 \vee x \vee \neg z \\ \neg s_4 \vee \neg x \vee y \vee z & \neg s_5 \vee x \vee w & \neg s_6 \vee w \vee z \vee \neg y \\ \neg s_7 \vee \neg x \vee \neg y & \neg s_8 \vee \neg x \vee \neg z & \neg s_9 \vee w \vee \neg x \vee \neg z \end{array}$$

Si le sélecteur associé est vrai (resp. faux) suivant les hypothèses courantes, la clause est alors activée (resp. désactivée). Les sélecteurs n'apparaissant que négativement dans la formule, les clauses apprises obtenues en utilisant le processus d'analyse de conflits introduit au chapitre 3.1.2.1 gardent donc une empreinte (le sélecteur s_i associé) de toutes les clauses initiales utilisées pour les produire.

Exemple 18

Considérons la formule Σ' de l'exemple 17 avec comme hypothèses $\mathcal{A} = \{s_1, s_2, \ldots, s_9\}$. Supposons que le solveur décide le littéral x à vrai. Nous avons dans ce cas, $\neg y$ propagé par la clause $\neg s_7 \vee \neg x \vee \neg y$, $\neg z$ propagé par la clause $\neg s_8 \vee \neg x \vee \neg z$ et $\neg x$ propagé par la clause $\neg s_4 \vee \neg x \vee y \vee z$. Cela conduit à un conflit à partir de la clause $\neg s_4 \vee \neg x \vee y \vee z$ qui produit une clause assertive σ_2 générée de la manière suivante :

$$\begin{array}{lcll} \sigma_1 & = & \eta[z, \neg s_4^4 \vee \neg x^{10} \vee y^{10} \vee z^{10}, \neg s_8^8 \vee \neg x^{10} \vee \neg z^{10}] & = \neg s_4^4 \vee \neg s_8^8 \vee \neg x^{10} \vee y^{10} \\ \sigma_2 & = & \eta[y, \sigma_1, \neg s_7^7 \vee \neg x^{10} \vee \neg y^{10}] & = \neg s_4^4 \vee \neg s_8^8 \vee \neg s_7^7 \vee \neg x^{10} \end{array}$$

Ainsi, en affectant à vrai le sélecteur s_i, nous désactivons non seulement la clause initiale associée mais également toutes les clauses apprises dérivées à partir de celle-ci.

Si, pendant la recherche, un *nogood* nécessite d'affecter une hypothèse dans la phase inverse de celle choisie initialement, le problème devient alors insatisfaisable par rapport à ces hypothèses. Nous pouvons, en conséquence et en raisonnant à partir de la dernière clause conflictuelle, extraire le sous-ensemble des hypothèses réellement responsables du conflit en considérant le l-UIP, cet ensemble est aussi appelé *core*.

Exemple 19

Considérons une nouvelle fois la formule Σ' de l'exemple 17 avec comme hypothèses $\mathcal{A} = \{s_1, s_2, \ldots, s_9\}$. Supposons que la clause assertive générée précédemment dans l'exemple 18 soit ajoutée à la formule. L'ajout de cette clause conduit à propager $\neg x$. L'application de la propagation unitaire conduit à propager $\neg y$ par la clause $\neg s_2 \vee x \vee \neg y$, $\neg z$ par la clause $\neg s_3 \vee x \vee \neg z$ et x par la clause $\neg s_1 \vee x \vee y \vee z$. Cela conduit à un conflit au niveau de décision des hypothèses à partir de la clause $\neg s_1 \vee x \vee y \vee z$. Le l-UIP est alors généré de la manière suivante :

$$\begin{array}{lcll} \sigma_1 & = & \eta[z, \neg s_1^1 \vee x^9 \vee y^9 \vee z^9, \neg s_3^3 \vee x^9 \vee \neg z^9] & = & \neg s_1^1 \vee \neg s_3^3 \vee x^9 \vee y^9 \\ \sigma_2 & = & \eta[y, \sigma_1, \neg s_2^2 \vee x^9 \vee \neg y^{10}] & = & \neg s_1^1 \vee \neg s_3^3 \vee \neg s_2^2 \vee x^9 \\ \sigma_3 & = & \eta[x, \sigma_2, \neg s_4^4 \vee \neg s_8^8 \vee \neg s_7^7 \vee \neg x^9] & = & \neg s_1^1 \vee \neg s_3^3 \vee \neg s_2^2 \vee \neg s_4^4 \vee \neg s_8^8 \vee \neg s_7^7 \end{array}$$

L'extraction de cette clause permet de conclure qu'il n'est pas possible d'activer les clauses $1, 2, 3, 4, 7$ et 8 simultanément.

Comme nous avons pu le voir sur les deux derniers exemples, l'ajout des sélecteurs conduit à une augmentation de la taille des clauses apprises. Cela a une conséquence directe sur les performances de la procédure `bcp`. De plus, puisque les littéraux sélecteurs sont toujours assignés avant la « véritable » résolution du problème, de nombreuses informations liées à la taille des clauses sont perdues. Par exemple la clause générée dans l'exemple 18, bien qu'étant unitaire une fois les sélecteurs assignés convenablement, pourrait être supprimée lors du processus de réduction de la base de clauses apprises faute de l'utilisation d'une mesure adaptée. Dans la suite, nous présentons deux approches afin d'éviter les écueils que nous venons d'énoncer.

4.2.2 Amélioration de `Glucose`

Comme nous venons de le souligner, chaque clause apprise va posséder tous les sélecteurs associés à toutes les clauses initiales utilisées dans le processus de résolution pour la générer. Ainsi, comme nous avons pu le montrer expérimentalement dans (Audemard et al. 2013a), le nombre de sélecteurs dans une clause apprise peut être extrêmement important. Comme nous l'avons rappelé précédemment, les solveurs `SAT` associent à chaque hypothèse son propre niveau de décision (excepté lorsqu'elle est propagée par une précédente hypothèse). Ainsi, dans de nombreux cas, une mesure comme le `LBD`, qui affecte le poids d'une clause par rapport à son nombre de niveaux de décision, va avoir tendance à être dominée par le nombre de sélecteurs que possède cette clause.

Par conséquent, le LBD va très souvent valoir approximativement la taille de la clause. Si nous ajoutons à cela le fait que le score LBD est très discriminant (les clauses de LBD égal à $n+1$ sont significativement moins importantes que celles de LBD égal à n) et que toutes les heuristiques de Glucose gravitent autour de cette mesure, il est clair que le LBD doit être calculé au plus juste. Dans nos travaux (Audemard et al. 2013a), nous avons proposé d'adapter simplement ce score en ne prenant pas en compte les sélecteurs dans son calcul. Ce nouveau score LBD est simplement nommé « Nouveau LBD ».

Afin d'illustrer l'impact des sélecteurs sur la taille des clauses apprises, la table 4.1 reporte quelques statistiques relatives aux remarques précédentes dans le cadre de l'extraction d'un MUS avec l'outil MUSER (Belov et Marques-Silva 2011) (voir (Audemard et al. 2013a) pour plus de détails sur le protocole utilisé). Sur 4 instances représentatives, nous détaillons les scores LBD initiaux et les scores des nouveaux LBD. Comme nous pouvons le constater, il est clair que l'utilisation du LBD initial n'est pas du tout adapté dans le cadre de moteurs SAT utilisant les sélecteurs. La valeur des LBD se rapproche significativement de la taille des clauses apprises. En revanche, en utilisant la nouvelle définition des LBD, nous pouvons constater que la valeur des LBD n'est plus du tout reliée à la taille des clauses apprises, mais bien plus petite. De plus, le temps nécessaire à la résolution du problème est fortement amélioré.

		LBD					**Nouveau** LBD				
			Taille		LBD			Taille		LBD	
Instance	#C	T	moy	max	moy	max	T	moy	max	moy	max
fdmus_b21_96	8541	29	1145	5980	1095	5945	11	972	6391	8	71
longmult6	8853	46	694	3104	672	3013	14	627	2997	11	61
dump_vc950	360419	110	522	36309	498	35873	67	1048	36491	8	307
g7n	15110	190	1098	16338	1049	16268	75	1729	17840	27	160

TABLE 4.1 – Pour quelques instances significatives, nous reportons le nombre de clauses (#C), et pour chaque définition de LBD (initiale et nouvelle), nous donnons aussi le temps nécessaire pour calculer un MUS, la taille moyenne et max des clauses apprises et la valeur moyenne et max des LBD.

Malgré une légère amélioration des performances de Glucose, nous nous sommes aperçus que cette nouvelle version obtenait des résultats très proches de Minisat (par rapport au nombre d'instances résolues), mais plus lente. À ce niveau de nos expérimentations, les résultats étaient assez décevants : Glucose est supposé être beaucoup plus efficace que Minisat dans le cadre non incrémental. La raison de cette déconvenue vient du fait que les clauses apprises peuvent rapidement devenir extrêmement grandes. Comme le montre le tableau 4.1, les clauses apprises ont une taille moyenne de 1729 littéraux pour l'instance g7n (avec certaines clauses contenant quelque 10 000 littéraux !). Dès lors, même une opération simple comme le parcours d'une clause peut rapidement avoir un surcoût prohibitif. Ceci est très problématique, puisque de nombreuses opérations fondamentales des solveurs SAT sont justement basées sur le parcours de clauses (propagation unitaire, calcul du LBD, calcul des clauses apprises, suppression des clauses satisfaites, ...). Dans la suite nous présentons des améliorations spécialement dédiées à ces opérations simples mais cruciales.

4.2.2.1 Parcours efficaces des clauses contenant des sélecteurs

Comme nous venons de le voir, dans de nombreux cas, les clauses apprises vont contenir majoritairement des variables sélecteurs qui n'ont pas à être prises en compte dans le calcul du nouveau LBD. Comme ces littéraux peuvent être utilisés comme sentinelle (*watchers*), nous ne pouvons malheureusement pas les partitionner en deux sous-ensembles indépendants (sélecteurs/non sélecteurs) : nous devons être à même de visiter ces littéraux particuliers à volonté. Il est néanmoins possible d'enregistrer dans chaque clause sa taille et le nombre de sélecteurs qu'elle contient. De plus, lors de la création de chaque clause, nous poussons les sélecteurs à la fin. Nous pouvons espérer qu'un tel arrangement accélérera la mise à jour des LBD : nous pouvons stopper le parcours de la clause dès lors que tous les littéraux initiaux ont été visités.

4.2.2.2 Accélération de la propagation

Si les modifications des structures de données décrites précédemment permettent d'améliorer le calcul de la mise à jour du LBD, nous avons à faire face à un autre problème : la propagation unitaire nécessite également de parcourir les clauses pour détecter de nouveaux littéraux sentinelles. Supposons que, durant la propagation d'une hypothèse (sélecteur), nous cherchions une nouvelle sentinelle pour une clause α. Supposons également que la nouvelle sentinelle s_i de la clause α soit choisi parmi les autres sélecteurs. Si s_i est également choisi comme hypothèse (tous les sélecteurs le sont au fur et à mesure), alors la clause α sera à nouveau parcourue. Ceci peut facilement être évité. Lors de la propagation d'une hypothèse, nous parcourons la clause en entier pour trouver une nouvelle sentinelle qui soit vrai (la clause est alors satisfaite) ou qui ne soit pas un sélecteur. De cette manière, nous espérons limiter le nombre de clauses à visiter lors de la recherche de nouveaux littéraux sentinelles. De plus, comme Glucose effectue de nombreux redémarrages, nous limitons également le retour-arrière pour qu'il ne remonte pas au-dessus du premier niveau de décision qui n'est pas une hypothèse.

4.2.2.3 Simplification de la base de clauses

La dernière modification que nous avons introduite est reliée à la suppression définitive des clauses satisfaites. En effet, lorsqu'un sélecteur est fixé au niveau 0, toutes les clauses apprises où il apparaît sont également définitivement satisfaites. Parcourir toutes les clauses apprises pour détecter ces littéraux satisfaits peut être contre-productif. Dans la version de Glucose que nous avons proposée, nous supprimons uniquement les clauses apprises dont un des deux littéraux sentinelles est définitivement satisfait. Comme nous avons modifié le processus de propagation unitaire (voir plus haut), nous pouvons espérer que cela soit suffisant pour supprimer la majorité des clauses apprises définitivement satisfaites.

Bien que ces améliorations aient permis d'améliorer significativement les performances de Glucose, elles ne s'attaquent pas réellement au mal par la racine. En effet, le réel problème vient du fait que les clauses apprises produites sont de trop grande taille. Dans la suite, nous présentons une approche qui s'inspire de la résolution étendue (Huang 2010) afin de raccourcir les clauses.

4.2.3 Factoriser les hypothèses

Comme nous l'avons souligné précédemment, la résolution du problème SAT sous hypothèses conduit à la génération de clauses apprises contenant de nombreux littéraux. Nous décrivons

maintenant une approche, présentée dans (Lagniez et Biere 2013), qui utilise une structure dédiée afin de considérer de manière paresseuse (*lazy*) ces hypothèses. Plus précisément, nous nous inspirons de la résolution étendue (Huang 2010, Audemard et al. 2010a) afin de factoriser les hypothèses présentes dans les clauses apprises.

4.2.3.1 Graphe de définition

Lorsqu'une clause est apprise, nous proposons de remplacer la « partie hypothèse » pas un nouveau littéral, appelé *abréviation*. Cette dernière est constituée de l'ensemble des littéraux hypothèses de la clause apprise. Afin de sauvegarder les relations entre les différentes abréviations nous sauvegardons leur définition dans un *graphe de définition* de la manière suivante.

$$(p_1 \vee \cdots \vee p_n \vee \neg a_1 \vee \cdots \vee \neg a_m)$$

est factorisée en

$$(p_1 \vee \cdots \vee p_n \vee \ell) \qquad \text{et} \qquad \ell \mapsto \underbrace{\neg a_1 \vee \cdots \vee \neg a_m}_{\mathsf{G}[\ell]}$$

Formellement, soit $p_1 \vee \cdots \vee p_n \vee a_1 \vee \cdots \vee a_m$ une nouvelle clause apprise, où $p_1, \ldots, p_n$ sont des littéraux initiaux et $a_1, \ldots, a_m$ sont des sélecteurs, ou des abréviations. Au lieu de la garder telle qu'elle, un nouveau littéral ℓ est créé et la clause est modifiée de manière à incorporer le littéral ℓ à la place des hypothèses de celle-ci. Nous obtenons la clause $p_1 \vee \cdots \vee p_n \vee \ell$. Puis, nous sauvegardons $a_1 \vee \cdots \vee a_m$ comme étant une définition $\mathsf{G}[\ell]$ de ℓ dans le graphe de définition G. Remarquons que lorsque $m \leq 1$ ce remplacement n'a pas de sens et donc la clause apprise n'a pas besoin d'être modifiée.

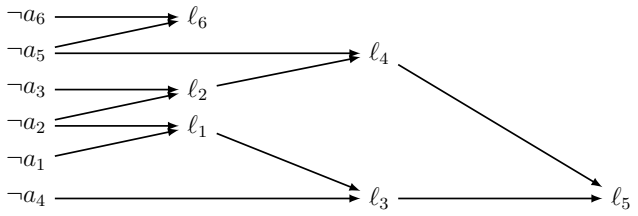
Considérons l'exemple de la figure 4.2 qui illustre le comportement de notre approche sur l'exécution d'un solveur `SAT` incrémental avec les hypothèses $a_1, \ldots, a_6$. La partie gauche de la figure 4.2(a) illustre l'ensemble des clauses apprises $\alpha_1, \ldots, \alpha_7$ par le solveur à travers ces différents appels, où $p_1, \ldots, p_7$ sont les littéraux initiaux et $a_1, \ldots, a_6$ sont les littéraux hypothèses. Afin de simplifier l'exemple, nous ne reportons pas dans les antécédents l'ensemble des clauses utilisées pour la génération des clauses apprises. Par exemple, la clause α_3 est dérivée par résolution *via* la clause α_1 et des clauses non reportées de la formule originale (les "...").

Le résultat de l'introduction d'abréviations pour la factorisation des clauses apprises est reporté dans la partie droite de la figure 4.2(a). La première clause α_1 est factorisée en α'_1 et la définition $\neg a_1 \vee \neg a_2$ du nouveau littéral abréviation ℓ_1 est ajoutée au graphe de définition reporté dans la figure 4.2(b). Cette sauvegarde se traduit par l'ajout dans le graphe G du nœud ℓ_1 et s'ils ne sont pas déjà présents, des nœuds a_1 et a_2. ℓ_1 est lié avec sa définition par les deux arcs entrants (a_1, ℓ_1) et (a_2, ℓ_1). Remarquons aussi que $\alpha'_5 = \alpha_5$, puisqu'il n'y a qu'un seul littéral non-initial, ceci (comme discuté précédemment) permet de réduire le nombre de littéraux nouveaux à introduire. Pour terminer, remarquons que comme pour ℓ_3, ℓ_4 ou ℓ_5, certaines définitions peuvent dépendre d'autres définitions et donc comme pour α_3, α_4, et α_6 nous pouvons factoriser récursivement les clauses apprises.

Rappelons que nous nous sommes placés dans le cas où les hypothèses sont *toujours* assignées. De plus, nous avons supposé que l'ensemble des **variables** utilisées dans les hypothèses ne change pas entre les différents appels aux solveurs `SAT`. Sous ces conditions, il est *toujours* possible de définir, *via* une phase d'initialisation discutée dans la section suivante, la valeur de vérité de toutes les variables abréviations. Remarquons que notre approche fonctionne toujours même si

clauses apprises	antécédents		clauses factorisées
$\alpha_1 : p_2 \vee p_7 \vee \neg a_1 \vee \neg a_2$	$\{\ldots\}$		$\alpha'_1 : p_2 \vee p_7 \vee \ell_1$
$\alpha_2 : p_2 \vee \neg a_2 \vee \neg a_3$	$\{\ldots\}$		$\alpha'_2 : p_2 \vee \ell_2$
$\alpha_3 : p_7 \vee p_4 \vee \neg p_6 \vee \neg a_1 \vee \neg a_2 \vee \neg a_4$	$\{\alpha_1, \ldots\}$	*factorisation*	$\alpha'_3 : p_7 \vee p_4 \vee \neg p_6 \vee \ell_3$
$\alpha_4 : p_6 \vee p_8 \vee \neg a_3 \vee \neg a_2 \vee \neg a_5$	$\{\alpha_2, \ldots\}$	$\Longrightarrow$	$\alpha'_4 : p_6 \vee p_8 \vee \ell_4$
$\alpha_5 : p_2 \vee p_5 \vee \neg a_2$	$\{\ldots\}$		$\alpha'_5 : p_2 \vee p_5 \vee \neg a_2$
$\alpha_6 : p_7 \vee p_4 \vee \neg a_1 \vee \neg a_2 \vee \neg a_4 \vee \neg a_5$	$\{\alpha_3, \alpha_4, \ldots\}$		$\alpha'_6 : p_7 \vee p_4 \vee \ell_5$
$\alpha_7 : \neg p_2 \vee \neg a_6 \vee \neg a_5$	$\{\ldots\}$		$\alpha'_7 : \neg p_2 \vee \ell_6$

(a) Clauses apprises (clauses originales à gauche, version factorisée à droite)



(b) Graphe de définition

FIGURE 4.2 – Factorisation des hypothèses.

cette propriété n'est pas vérifiée. Cependant, dans ce cas, elle serait moins efficace car les clauses apprises contenant des littéraux non totalement définis (c'est-à-dire un littéral qui possède un antécédent qui n'est pas dans l'ensemble des hypothèses) ne seraient plus considérées.

4.2.3.2 Initialisation

Après avoir factorisé les hypothèses et ajouté les abréviations associées, toutes les clauses apprises α contiennent *au plus une* hypothèse ou abréviation. Cette dernière sera dénoté par $r(\alpha)$ dans la suite. Remarquons que $r(\alpha)$ peut être indéfini, dans le cas où elle a été assigné au niveau zéro ou lorsqu'une clause apprise ne possède pas d'hypothèses à sa création.

Comme le graphe de définition G peut être interprété comme un circuit (acyclique), il permet de calculer la valeur de vérité de chaque abréviation une fois que toutes les hypothèses ont été assignées. Afin d'associer la bonne valeur de vérité à une abréviation, nous avons besoin d'assigner les variables hypothèses et, récursivement, toutes les abréviations présentes dans sa définition. Cette initialisation est décrite dans l'algorithme 4.4. Il prend en entrée les arguments suivants : le littéral ℓ qui doit être assigné, l'interprétation courante et le graphe de définition G. Cette interprétation $\mathcal{I}$ représente l'état des littéraux prouvés unitaires au niveau zéro ainsi que le choix de l'utilisateur concernant l'activation et la désactivation de certaine clauses. Premièrement, les clauses prouvés unitaires au niveau zéro sont supprimés de $\mathsf{G}[\ell]$. Remarquons que l'impact de l'affection des hypothèses ou des abréviations au niveau zéro (signifiant qu'elles ont été *supprimées*) est pris en considération *a priori* par le solveur durant la phase de propagation unitaire. Ensuite, tant qu'il existe un littéral ℓ' dans $\mathsf{G}[\ell]$ tel que $\mathsf{G}[\ell]$ n'est pas défini par l'interprétation courante $\mathcal{I}$, nous appelons récursivement la procédure afin d'assigner ℓ'. Lorsque la valeur de $\mathsf{G}[\ell]$ est déterminée sous l'interprétation $\mathcal{I}$, nous pouvons alors assigner ℓ à $\mathcal{I}(\mathsf{G}[\ell])$ (ligne 5).

Par construction du graphe de définition, G ne possède pas de cycle. De plus, nous supposons que toutes les hypothèses sont assignées dans $\mathcal{I}$, comme discuté précédemment. Par conséquent, notre algorithme termine et assigne la valeur de chaque abréviation ℓ en fonction de sa définition $\mathsf{G}[\ell]$.

Algorithme 4.4 : assigneAbreviation

Input : ℓ : littéral ; **var** $\mathcal{I}$: interprétation ; G : graphe de définition

```
1 supprimeUnit(G[ℓ]);
2 while I(G[ℓ]) non assigné do
3 |   choisir ℓ' ∈ G[ℓ] non assigné ;
4 |   assigneAbreviation(G, ℓ', I);
5 if I(G[ℓ]) = ⊥ then I ⟵ I ∪ {¬ℓ} else I ⟵ I ∪ {ℓ};
```

Il est important de noter que, dans le pire des cas, toutes les clauses apprises nécessitent l'ajout d'une nouvelle abréviation. Par conséquent, le graphe de définition peut croître linéairement dans le nombre de conflits réalisé par le solveur. Cela implique, d'une part, une augmentation de la consommation de la mémoire, et d'autre part, une augmentation du temps nécessaire à l'initialisation des abréviations pour chaque appel à l'oracle `SAT`. Cependant, étant donné que des clauses peuvent être inactivées ou supprimées par le solveur, certaines abréviations ne sont plus référencées dans aucune clause. Dans ce cas, l'initialisation de ces abréviations devient inutile. En fait, seules les abréviations qui apparaissent dans des clauses apprises (ou qui sont atteignables récursivement à partir de ces dernières) ont besoin d'être considérées lors du processus d'initialisation.

L'algorithme 4.5 décrit une méthode d'initialisation qui prend en compte cet argument. Il retourne une interprétation $\mathcal{I}$ qui assigne toutes les abréviations qui sont récursivement atteignables à partir d'une clause de Σ (ce qui inclut aussi les clauses apprises). Pour commencer, l'algorithme initialise l'interprétation courante avec toutes les hypothèses (ligne 1). Ensuite, toutes les clauses α de Σ contenant une référence de remplacement $r(\alpha)$ sont considérées (ligne 2 – 5). Lorsqu'une telle clause est identifiée, l'algorithme 4.4 est appelé sur tous les littéraux de α non assignés dans $\mathcal{I}$ (ligne 3–4). Ce processus assigne dans $\mathcal{I}$ toutes les abréviations atteignables en fonction de la valeur de leur définition, et cela tant qu'une clause falsifiée n'est pas identifiée (ligne 5).

Algorithme 4.5 : initialisation

Input : Σ : une formule `CNF` ; **var** $\mathcal{A}$: hypothèses ; G : graphe de définition

```
1 I ⟵ A ∪ {les littéraux unitaires};
2 foreach α ∈ Σ contenant une référence de remplacement r(α) do
3 |   if r(α) n'est pas assignée dans I then
4 |   |   assigneAbreviation(G, r(α), I)
5 |   if I(G[ℓ]) = ⊥ then break;
6 return I;
```

4.2.3.3 Extraction d'un *core*

Comme énoncé précédemment, lorsque le problème est démontré insatisfaisable sous un ensemble d'hypothèses, il est possible d'extraire un sous-ensemble d'hypothèses permettant d'expliquer la raison de cette contradiction. Dans les implémentations standards des solveurs `CDCL`, le calcul d'un tel ensemble est réalisé par la fonction « analyzeFinal » en remontant le graphe d'implication jusqu'à obtenir une raison contenant uniquement des hypothèses. Cependant, dans

le cas où des abréviations sont ajoutées, il est nécessaire d'adapter la procédure « analyzeFinal » afin de prendre en compte ces informations.

L'algorithme 4.6 décrit une procédure permettant d'extraire un *core*. Il prend en paramètres une formule CNF Σ, la séquence de décisions-propagations $\mathcal{I}$ (ou *trail*), une clause $\alpha \in \Sigma$ falsifiée par $\mathcal{I}$, le graphe de définition, et retourne l'ensemble des hypothèses $\mathcal{C}$ représentant une raison de l'incohérence de la formule. Cette procédure commence par initialiser $\mathcal{C}$ ainsi que $\mathcal{V}$ l'ensemble des littéraux déjà visités avec l'ensemble vide. Ensuite, une pile $\mathbf{T}$, contenant les littéraux qui doivent être considérés, est initialisée avec les littéraux de la clause α. Tant qu'il existe un littéral ℓ dans $\mathbf{T}$ qui n'est pas marqué, nous le marquons (ligne 4). Il y a maintenant trois cas à considérer en fonction de ℓ. Si ℓ est une hypothèse, alors ℓ est ajouté à la clause conflit $\mathcal{C}$ (ligne 5). Si ℓ est une abréviation, alors sa définition $\mathsf{G}[\ell]$ est ajoutée à $\mathbf{T}$ (ligne 6). Cette opération est en fait la seule partie qui diffère de la version originale de l'algorithme d'extraction de *core*. Finalement, si ℓ n'est ni une hypothèse ni une abréviation, alors sa raison est ajoutée à $\mathbf{T}$ (ce qui correspond à l'exploration du graphe d'implication).

Algorithme 4.6 : analyzeFinal

Input : Σ : CNF ; $\mathcal{I}$: trail ; α : clause de Σ ; G : un graphe de définitions
Result : $\mathcal{C}$: un sous-ensemble d'hypothèses

1 $\mathcal{C} \longleftarrow \varnothing$; $\mathcal{V} \longleftarrow \varnothing$;
2 $\mathbf{T} \longleftarrow \alpha$;
3 **while** $\exists \ell \in \mathbf{T} \setminus \mathcal{V}$ **do**
4 $\quad \mathcal{V} \longleftarrow \mathcal{V} \cup \{\ell\}$;
5 $\quad$ **if** ℓ *est une hypothèse* **then** $\mathcal{C} \longleftarrow \mathcal{C} \cup \{\ell\}$;
6 $\quad$ **else if** ℓ *est une abréviation* **then** $\mathbf{T} \longleftarrow \mathbf{T} \cup \mathsf{G}[\ell]$;
7 $\quad$ **else** $\mathbf{T} \longleftarrow \mathbf{T} \cup exp(\ell)$;
8 **return** $\mathcal{C}$;

Exemple 20

Considérons encore l'exemple de la figure 4.2. Soit $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ l'ensemble des hypothèses, l'apprentissage de la clause α'_7 permet de conclure que la formule est insatisfaisable. L'algorithme 4.6 produit alors :

$\mathbf{T} \setminus \mathcal{V}$	$\mathcal{V}$	$\mathcal{C}$	ℓ
$\overline{p_2}, \ell_6$	$\varnothing$	$\varnothing$	$undef$
ℓ_6, ℓ_2	$\varnothing$	$\varnothing$	$\overline{p_2}$
$\ell_2, \neg a_5, \neg a_6$	ℓ_6	$\varnothing$	ℓ_6
$\neg a_5, \neg a_6, \neg a_2, \neg a_3$	ℓ_2, ℓ_6	$\varnothing$	ℓ_2
$\neg a_6, \neg a_2, \neg a_3$	$\ell_2, \ell_6, \neg a_5$	$\neg a_5$	$\neg a_5$
$\neg a_2, \neg a_3$	$\ell_2, \ell_6, \neg a_5, \neg a_6$	$\neg a_5, \neg a_6$	$\neg a_6$
$\neg a_3$	$\ell_2, \ell_6, \neg a_5, \neg a_6, \neg a_2$	$\neg a_5, \neg a_6, \neg a_2$	$\neg a_2$
$\varnothing$	$\ell_2, \ell_6, \neg a_5, \neg a_6, \neg a_2, \neg a_3$	$\neg a_5, \neg a_6, \neg a_2, \neg a_3$	$\neg a_3$

La clause apprise est donc $(\neg a_5 \vee \neg a_6 \vee \neg a_2 \vee \neg a_3)$. Notons que ni a_1 ni a_4 n'ont été utilisés.

4.2.4 Discussion

Il est important de noter que les travaux présentés dans cette section ne concernent que l'utilisation de solveurs `SAT` incrémentaux dans le cas où les clauses sont augmentées avec des sélecteurs. Cette situation est particulière dans le sens où les clauses apprises générées par les solveurs ont des tailles inhabituelles. Cependant, certaines des améliorations apportées à `Glucose` sont aussi applicables dans le cas général des solveurs incrémentaux. En effet, les méthodes permettant une amélioration de la propagation unitaire sont applicables dans le cas général. Dans le cas du calcul du `LBD`, cela est un peu plus problématique. Notre argument principal était que les littéraux sélecteurs sont forcément assignés et donc que la partie non hypothèses de la clause représente réellement la clause que le solveur aura à prendre en compte. Cependant, dans le cas général, cela n'est pas vrai et il est donc nécessaire, faute d'assigner à une clause une mauvaise valeur de `LBD`, de supprimer cette amélioration. Une manière de contourner ce problème serait de réévaluer le `LBD` d'une clause au début d'un nouvel appel au solveur. Pour cela, il serait suffisant de sauvegarder les informations concernant les groupes de niveau de décision d'une clause lorsqu'elle est créée, et avant la recherche de recalculer le `LBD` en supposant que les littéraux hypothèses sont assignés. Il serait alors possible de « geler » (Audemard et al. 2011a) les clauses avec un mauvais `LBD`.

Les deux méthodes que nous avons présentées dans cette section sont très différentes dans le sens où elles n'ont pas la même difficulté d'intégration. D'un côté, les améliorations ajoutées à `Glucose` sont facilement implémentables dans les solveurs de l'état de l'art. De l'autre, la gestion des littéraux hypothèses avec des abréviations est un peu plus intrusive. Néanmoins, étant donné les résultats reportés dans (Audemard et al. 2014a) il semble intéressant d'ajouter notre système de compression de clauses au solveur `Glucose` (qui est connu pour être plus performant que `Minisat`). Afin de réaliser cela, il faudrait à l'avenir implémenter une librairie efficace qui permettrait une gestion transparente du graphe de définition. Une telle librairie pourrait être très utile pour la communauté.

Il est à noter que pour le moment le système de sauvegarde des définitions est un peu sommaire. Il n'est pas rare que tout ou partie d'une définition que nous souhaitons ajouter se trouve déjà représentée dans le graphe. Il pourrait donc être avantageux d'utiliser les définitions déjà présentes au lieu d'en créer une nouvelle. Le détecter pourrait permettre de générer des clauses qui dépendent de moins d'abréviations, ce qui pourrait améliorer l'efficacité de la phase d'initialisation. Afin de rechercher efficacement de telles définitions, il est envisageable d'utiliser un système proche de la propagation unitaire (et donc d'utiliser les structures de données paresseuses). Pour réaliser cela, nous pourrions dans un premier temps récupérer l'ensemble des hypothèses qui correspond à l'ensemble d'abréviations/hypothèses que nous souhaitons compresser. Ensuite, étant donné que le graphe de définition est acyclique, nous pourrions voir les définitions comme des clauses de Horn et alors propager la négation des hypothèses que nous avons précédemment collectées.

Récemment, Fazekas et al. (2019) ont étudié comment intégrer du *inprocessing* dans les solveurs incrémentaux, c'est-à-dire des simplifications sur la formule en cours de recherche. Au vu des résultats obtenus par les auteurs, il semble important de pouvoir réaliser ce processus dans les solveurs que nous avons présentés dans cette section. En ce qui concerne la version incrémentale de `Glucose`, nous ne pensons pas que cela devrait poser un problème. Pour ce qui est de l'approche qui considère des abréviations il semblerait que l'intégration soit un peu plus complexe. En effet, les méthodes de *preprocessing* nécessitent souvent de parcourir les clauses. Dans le cas des abréviations, il est possible que la récupération des informations dans le graphe

de définition ralentisse les performances de ces procédures.

Précédemment, nous avons présenté une approche pour modifier une formule CNF afin de pouvoir lui ajouter/supprimer des clauses efficacement. Dans la suite, nous montrons comment il est possible d'utiliser cette approche pour le calcul d'un MCS ou l'énumération de MCSes.

4.3 Une approche destructive pour l'extraction d'un MSS

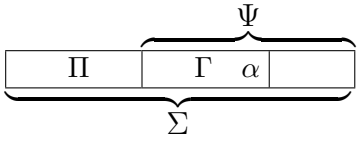
Les algorithmes d'extraction d'un MSS essaient généralement de construire incrémentalement ce dernier. Plus précisément, ils partent d'un ensemble de clauses Π de la formule Σ qui est satisfaisable (cet ensemble peut être vide) et ils vont tenter de l'augmenter en essayant de satisfaire au moins une clause α de la formule restante $\Sigma \setminus \Pi$. D'une certaine manière, ces méthodes sont orientées MSS, c'est-à-dire qu'elles vont toujours travailler avec une sous-formule satisfaisable.

Une autre manière de considérer la construction d'un MSS est de se focaliser sur sa contrepartie, c'est-à-dire la construction d'un MCS. Comme nous avons pu le voir au début de ce chapitre, un MCS représente un *hitting set* sur l'ensemble des MUSes. Il est donc possible de voir la construction d'un MCS comme la recherche et la suppression de clauses appartenant à des MUSes.

C'est sur ce principe que se base l'approche que nous présentons dans la suite. Plus précisément, contrairement aux approches standard qui opèrent à partir d'une formule satisfaisable, nous proposons de construire un MSS en supprimant des clauses d'une formule qui est insatisfaisable. Le nouvel algorithme de partition de CNF en un MSS et son MCS correspondant est appelé CMP pour *Computational Method for Partitioning*. Son ossature est décrite dans l'algorithme 4.7 de la section 4.3.1. Nous présentons aussi certaines améliorations optionnelles qui seront discutées à la section 4.3.2.

4.3.1 CMP : description

L'algorithme que nous proposons s'appuie sur la notion de clauses de transition. La différence avec l'approche de base ELS, c'est que la recherche d'une clause de transition est faite en « détruisant » la formule. Afin d'expliquer notre approche, considérons que la formule est partitionnée comme dans la figure suivante, où Σ est une CNF insatisfaisable, Σ est partitionnée en deux ensembles Π et Ψ avec Π satisfaisable, $\Gamma \subseteq \Psi$ et α est une clause de transition de $\Pi \cup \Gamma$.

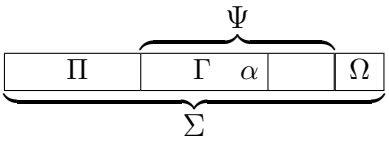


Par définition, si α est une clause de transition de $\Pi \cup \Gamma$ alors $\Pi \cup (\Gamma \setminus \{\alpha\})$ est satisfaisable. Ce qui implique qu'il est possible d'étendre la partie satisfaisable Π en considérant les clauses de $\Gamma \setminus \{\alpha\}$ (si cet ensemble n'est pas vide). Plus important, le fait que α soit une clause de transition de $\Pi \cup \Gamma$ implique qu'il ne sera jamais possible d'étendre $\Pi \cup (\Gamma \setminus \{\alpha\})$ avec α. Dit autrement, $\forall \Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$ tel que $\Theta \in \texttt{MCS}(\Sigma)$, nous avons $\alpha \in \Theta$. Ce qui implique que, quelle que soit la manière dont nous allons étendre la partie satisfaisable, la clause α sera dans le MCS retourné.

Il est important de noter que si Π est satisfaisable et $\Pi \cup \Psi$ est insatisfaisable, alors il est toujours possible de trouver un couple (Γ, α) tel que α est une clause de transition de $\Pi \cup \Gamma$.

Pour s'en convaincre il suffit de considérer un $\Theta \in \texttt{MCS}(\Pi \cup \Psi)$ tel que $\Theta \subseteq \Psi$ (un tel Θ existe puisque Π est satisfaisable et $\Pi \cup \Psi$ est insatisfaisable). Dans ce cas, nous pouvons considérer $\Gamma = \Psi \setminus (\Theta \cup \{\alpha\})$ avec $\alpha \in \Theta$.

Compliquons un peu l'exemple précédent en considérant la partition de Σ représentée sur la figure suivante, où Ω est un sous-ensemble de Σ qui satisfait la propriété que $\forall \alpha \in \Omega$ nous avons $\Pi \cup \{\alpha\}$ est insatisfaisable. Autrement dit, Ω est un MCS de $\Pi \cup \Omega$.



Comme énoncé précédemment, nous avons que $\forall \Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$ si $\Theta \in \texttt{MCS}(\Sigma)$ alors $\alpha \in \Theta$. À présent, si nous étendons l'ensemble Π avec les clauses de $\Gamma \setminus \{\alpha\}$, supprimons de Ψ les clauses de Γ et ajoutons α à Ω, alors nous sommes de nouveau dans une situation respectant toutes les propriétés de l'exemple. Ce qui implique que nous pouvons réitérer le processus en déplaçant itérativement des clauses dans Ω tant que $\Pi \cup \Psi$ est insatisfaisable. Dans le cas où $\Pi \cup \Psi$ devient satisfaisable, $\Pi \cup \Psi$ est MSS de Σ et par construction Ω est un MCS de Σ. C'est sur ce principe que s'appuie la méthode CMP (Grégoire et al. 2014d) présentée dans l'algorithme 4.7.

Algorithme 4.7 : CMP (Computational Method for Partitioning)

input : Σ une formule CNF
output : (MSS,MCS) une partition de Σ

```
1  (Π, Ψ) ⟵ ApproximatePartition(Σ);
2  Ω ⟵ ∅ ;
3  Γ ⟵ Ψ ;                                          // Working subset of Ψ
4  while Ψ ≠ ∅ do
5      extendSatPart(Π, Γ, Ω, Ψ) ;
6      if Ψ ≠ ∅ then
7          α ⟵selectClause(Γ) ;
8          (I, Δ) ⟵ solve(Π ∪ (Γ \ {α}) ∪ ᾱ);
           //Returns a model I if SAT and
           //(I = ∅ and core Δ) otherwise
9          if I ≠ ∅ then                              // Transition clause
10             Ω ⟵ Ω ∪ {α} ;
11             Π ⟵ Π ∪ {β ∈ (Ψ\{α}) s.t. I(β)=1} ∪ ᾱ ;
12             Γ ⟵ Ψ ⟵ {β ∈ (Ψ\{α}) s.t. I(β)=0} ;
13         else
14             Γ ⟵ Γ\{α};
15             exploitCore(Π, Γ, Ψ, Δ, α) ;
16 return (Π ∩ Σ,Ω);
```

L'algorithme CMP commence par calculer une approximation d'un MSS de Σ via la fonction ApproximatePartition (ligne 1) qui réalise un appel à un solveur de type CDCL pour vérifier la satisfaisabilité de Σ. Si Σ est satisfaisable, ApproximatePartition retourne la partition $(\Sigma, \varnothing)$

et comme $\Psi = \varnothing$, `CMP` n'entre pas dans la boucle principale et l'algorithme se termine. Dans le cas contraire, `ApproximatePartition` partitionne Σ en un couple (Π, Ψ), où Π est un sous-ensemble du `MSS` que va calculer l'algorithme. Ce sous-ensemble est obtenu à partir des clauses satisfaites par une interprétation retournée par le solveur `CDCL`. Nous aurions pu considérer à cet endroit une interprétation générée de manière aléatoire, mais pour des raisons de qualité de la partition, nous avons préféré utiliser l'interprétation correspondant à celle du *progress-saving*. Π sera modifié au cours de la recherche pour fournir le `MSS` final tandis que le `MCS` final Ω est d'abord initialisé à l'ensemble vide (ligne 2). Quand les parties optionnelles de l'algorithme sont activées, Π peut également être enrichi de clauses qui ne sont pas toutes issues de Ψ mais qui peuvent en être déduites, ceci afin d'accélérer le temps pris par le processus global. En fait, dans ce cas, l'invariant de boucle est $\Sigma = (\Pi \cap \Sigma) \cup \Psi \cup \Omega$.

Au début (ligne 3) et chaque fois qu'une clause est ajoutée à Ω (ligne 10), le sous-ensemble de travail Γ est initialisé à Ψ (ligne 12). Tant que chaque clause de Ψ n'a pas été répartie soit dans le `MCS`, soit dans le `MSS` en construction, l'algorithme cherche une clause $\alpha \in \Psi$ telle que cette clause soit une clause de transition de $\Pi \cup \Gamma \cup \{\alpha\}$ où $\Gamma \cup \{\alpha\} \subseteq \Psi$, c'est-à-dire telle que $\Pi \cup \Gamma \cup \{\alpha\}$ soit non satisfaisable alors que $\Pi \cup \Gamma$ l'est. Lorsque qu'une telle clause α est découverte, elle est déplacée dans le `MCS` en construction (ligne 10). Toutes les clauses de $\Psi \setminus \{\alpha\}$ qui sont satisfaites (respectivement falsifiées) par le modèle trouvé lors du dernier test de satisfaisabilité de $\Pi \cup \Gamma$ sont déplacées dans Π (ligne 11) (respectivement restent dans Ψ (ligne 12)). Le sous-ensemble de travail Γ est alors initialisé à la nouvelle valeur de l'ensemble Ψ (ligne 12). Quand α n'est pas une clause de transition (ligne 14), celle-ci est retirée du sous-ensemble de travail Γ (ligne 14). La fonction `selectClause` (ligne 7) choisit une clause α de Γ. Notre implémentation de cette fonction est basée sur l'heuristique `VSIDS`.

Il est facile de voir que la boucle se termine toujours. Gardons en mémoire que Γ est initialisé à Ψ (lignes 3 et 12). Puisque Σ n'est pas satisfaisable alors que Π est toujours satisfaisable, il existe au moins une clause $\alpha \in \Gamma$ tel que le test de satisfaisabilité (ligne 9) réussisse. Ceci grâce au fait que, précédemment, pour chaque α dont le test a été infructueux, celui-ci a été retiré de Γ (ligne 14). Lorsqu'un modèle est trouvé, Ψ est donc réduit (ligne 12). Le cas extrême est atteint lorsque la satisfaisabilité est prouvée alors que Γ est réduit à une clause $\Gamma = \{\alpha\}$.

`CMP` retourne une partition (Π, Ω) de la `CNF` Σ où Π est un `MSS` de Σ (et Ω le `MCS` correspondant). À ce niveau, deux remarques s'imposent :

1. La boucle de `CMP` effectue une recherche de clauses de transition qui diffère fondamentalement de celle de `ELS`. Dans le pire cas, `CMP` réalise $O(n^2)$ appels à un solveur `SAT` alors que `ELS` effectue seulement un nombre linéaire d'appels au solveur. Cependant, notre étude expérimentale montre que même sans les instructions optionnelles encadrées, l'algorithme de base de `CMP` est plus efficace sur de nombreuses instances.

2. La version complète de `CMP` inclut plusieurs fonctionnalités supplémentaires qui sont encadrées dans l'algorithme 4.7. Nous montrons que chacune d'elles ainsi que leur succession permet d'améliorer les performances de notre approche. Il est à noter que certaines de ces options sont déjà présentes dans d'autres approches comme `extendSatPart`, alors que d'autres sont exploitées de manière originale (c'est le cas pour le concept que nous appelons « *opposite enforcement on backbone* ») et les dernières constituent de nouveaux concepts (`exploitCore`).

4.3.2 `CMP` : améliorations optionnelles

Nous avons envisagé quatre améliorations à `CMP`. Elles sont représentées par les instructions encadrées de l'algorithme 4.7.

Tout d'abord, tout comme dans (Birnbaum et Lozinskii 2003, Marques-Silva et al. 2013), nous avons exploité l'idée que vérifier la satisfaisabilité d'une `CNF` logiquement affaiblie où certaines clauses sont remplacées par leur disjonction (notée $\bigvee \Gamma$) pouvait être utile. Dans le cas où $\Pi \cup \{\bigvee \Gamma\}$ n'est pas satisfaisable, $\Pi \cup \Gamma$ n'est pas non plus satisfaisable. Dans le cas où il existe un modèle, l'ensemble des clauses de Γ qui sont satisfaites par le modèle trouvé de $\Pi \cup \{\bigvee \Gamma\}$ peut être déplacé dans Π. C'est le rôle de la fonction `extendSatPart` à la ligne 5 de l'algorithme 4.7. Cette fonctionnalité joue un rôle important dans l'efficacité de la méthode `CLD` proposée dans (Marques-Silva et al. 2013).

La seconde amélioration (algorithme 4.7, ligne 11) concerne les littéraux du « *backbone* » (voir Monasson et al. (1999a), Kilby et al. (2005)) ; elle a également été exploitée dans les procédures d'extraction de `MCS` de Marques-Silva et al. (2013). $\overline{\alpha}$ est l'ensemble de clauses unitaires composées des littéraux opposés à ceux de α. Lorsque α est prouvé appartenir au `MCS` que l'on calcule, cela implique que $\overline{\alpha}$ est une conséquence logique de l'ensemble courant Π (et, par monotonie, de tous ses sur-ensembles). Ces clauses unitaires apprises sont ajoutées dans Π (algorithme 4.7, ligne 11) dans l'espoir d'accélérer les futurs tests de satisfaisabilité des sur-ensembles de Π.

Une autre exploitation originale de ces clauses unitaires peut être réalisée lors du test de satisfaisabilité pour déterminer si α est une clause de transition (algorithme 4.7, ligne 8). Lorsqu'un modèle est trouvé, $\Pi \cup \Gamma \setminus \{\alpha\}$ est satisfaisable alors que $\Pi \cup \Gamma$ ne l'est pas. Un résultat identique peut être obtenu en testant la satisfaisabilité de $\Pi \cup (\Gamma \setminus \{\alpha\}) \cup \{\overline{\alpha}\}$, où l'ensemble des clauses unitaires $\overline{\alpha}$ est ajouté pour accélérer le test. À notre connaissance, cette utilisation des littéraux du *backbone* dans un ensemble de prémisses plus fort qui n'est pas uniquement cohérent avec $\overline{\alpha}$ mais qui implique $\overline{\alpha}$, est nouvelle dans la recherche de `MCS`.

Nous appelons cette option « *opposite enforced* ». De manière intéressante, `CMP` autorise ce renforcement car la clause de transition est trouvée dans le cas où le test de satisfaisabilité répond positivement alors que les approches traditionnelles identifient une clause de transition lorsque l'incohérence est atteinte.

Algorithme 4.8 : exploitCore$(\Pi, \Gamma, \Psi, \Delta, \alpha)$

```
1 if Δ ∩ ᾱ = ∅ then Γ ⟵ Γ ∩ Δ ;
2 elsif Δ ∩ Γ = ∅ then
3 |   Π ⟵ Π ∪ {α} ;
4 |   Ψ ⟵ Ψ \ {α} ;
5 else Π ⟵ Π ∪ {Δ \ ᾱ ⊨ α} ;
```

La quatrième amélioration correspond à la fonction `exploitCore` (algorithme 4.7, ligne 15) qui affine de manière originale Γ étant donné Δ, le *core* calculé lorsque le test de satisfaisabilité a échoué. Trois cas peuvent être distingués.

1. Quand Δ ne contient aucune clause de $\overline{\alpha}$, nous conservons uniquement dans Γ les éléments de Δ (puisque nous pourrons toujours trouver une clause de transition dans cette intersection) ;
2. Dans le cas contraire, si aucune clause de Γ n'appartient à Δ, alors cela implique que Δ est constitué uniquement de clauses de $\overline{\alpha}$ et Π. Par conséquent, $\Pi \wedge \overline{\alpha}$ est insatisfaisable,

c'est-à-dire, α est une conséquence logique de Π. Ainsi, α peut être déplacé dans la partie satisfaisable Π.

3. Sinon, Δ est construit à partir de clauses de $\overline{\alpha}$ et Γ (et peut-être également de clauses de Π). Dans ce cas, comme au moins une clause de $\overline{\alpha}$ est dans Δ et que Δ est non satisfaisable, nous obtenons que $(\Delta \setminus \overline{\alpha}) \wedge \overline{\alpha}$ est non satisfaisable, c'est-à-dire que α est une conséquence logique de $\Delta \setminus \overline{\alpha}$. Bien que cela ne puisse être représenté sous la forme d'un ensemble de clauses de manière directe, cette information peut être exploitée et implémentée facilement (sans coût significatif en espace) en utilisant des sélecteurs de clauses comme cela a été montré dans la section précédente. Avec ces sélecteurs, $\Delta \setminus \overline{\alpha} \vDash \alpha$ peut être représenté par la clause $(\neg s_{\delta_1} \vee \cdots \vee \neg s_{\delta_m} \vee s_\alpha)$ où δ_i $(i \in [1..m])$ sont les m clauses de $\Delta \setminus \overline{\alpha}$. Lorsque toutes les clauses de $\Delta \setminus \overline{\alpha}$ sont actives, α est également activé.

Il est simple de montrer que toutes ces améliorations ne remettent pas en question la correction de CMP.

4.3.3 Discussion

Les expérimentations intensives que nous avons conduites à l'époque ont montré que CMP était plus robuste que les approches précédemment proposées et permettait de calculer une partition pour un plus grand nombre d'instances. Évidemment, cela n'impliquait pas que CMP était le plus efficace pour toutes les instances. À cet égard, il convient de rappeler que la complexité dans le pire cas de CMP est plus importante que celle des approches standard. En effet, lorsque n est le nombre de clauses de l'instance, CMP nécessite $O(n^2)$ appels à un solveur SAT dans le pire cas alors que les approches standard n'en nécessitent qu'un nombre linéaire (en l'occurrence n). Néanmoins, nous avons constaté que le nombre d'appels au solveur SAT fait par CMP reste toujours très significativement plus petit que n pour toutes les instances pour lesquelles il a réussi à calculer une partition. Cela pourrait être en partie expliqué par le fait que nous utilisons le modèle retourné par le solveur afin d'accélérer l'extraction. En effet, Janota et Marques-Silva (2016) ont montré que l'extraction d'un ensemble minimal nécessite un nombre d'appels logarithmique à un oracle dès lors que ce dernier retourne un modèle (et qu'il est utilisé). Dans leur papier, les auteurs considèrent de nombreux problèmes tels que l'extraction d'un MUS ou d'un MCS, et montrent que les approches utilisées en pratique pour l'extraction de tels ensembles ont bien une complexité logarithmique dans le nombre de clauses du problème considéré. Afin de mieux comprendre les performances de CMP, il serait intéressant d'étudier si ce résultat s'applique aussi à cet outil.

À l'heure où ce manuscrit est rédigé, CMP n'est plus l'outil d'extraction le plus efficace de l'état de l'art. En effet, l'outil proposé par Mencía et al. (2015), nommé LBX, est maintenant aussi efficace sur les instances sur lesquelles nous avons réalisé nos expérimentations dans notre papier (Grégoire et al. 2014d), et sensiblement plus efficace sur de nouvelles instances venant de la compétition MaxSAT. Faut-il conclure que l'approche destructive utilisée dans CMP est moins bonne que l'approche constructive de LBX ? La réponse à cette question n'est pas forcément affirmative. En effet, une des particularités de l'approche proposée par Mencía et al. (2015) est qu'elle ne nécessite pas l'ajout de sélecteurs aux clauses. Nous avons vu dans la section précédente que l'utilisation de sélecteurs pouvait avoir des conséquences dramatiques sur les performances des solveurs. Cette différence pourrait alors expliquer en partie le fait que LBX soit maintenant plus efficace que CMP. Une manière de vérifier cela pourrait être d'utiliser un des solveurs présentés dans la section précédente (Lagniez et Biere 2013, Audemard et al. 2013a) pour ne plus avoir ce problème de gestion de sélecteurs. En fait, pour être totalement juste il faudrait que les solveurs

utilisés dans les deux outils s'appuient sur les mêmes heuristiques. Une autre piste pour améliorer CMP pourrait être de proposer une approche qui s'appuie sur la recherche d'un *backbone* comme pour LBX, mais qui essaie de construire ce dernier de manière destructive. Ce n'est pas que nous ayons un goût particulièrement prononcé pour la destruction ... nous sommes juste un peu curieux de voir si cela est déjà possible en théorie, et si c'est le cas, de déterminer si cela est efficace.

Une autre piste pour rendre CMP plus performant pourrait être d'améliorer l'heuristique de choix de clauses (fonction selectClause). Comme notre objectif est de « casser » les MUSes de la formule, une approche telle que celle proposée par Grégoire et al. (2007) et qui s'appuie sur la recherche locale pour identifier les MUSes pourrait être appropriée. Plus précisément, il pourrait être envisageable de lancer une recherche locale, de pondérer les clauses comme cela est réalisé dans (Grégoire et al. 2007), et ensuite de considérer la clause qui a le score le plus important. Afin de ne pas rendre l'heuristique trop gourmande en ressources de calcul, il faudra bien faire attention à ne pas lancer la méthode de recherche locale trop longtemps ou de ne pas la lancer systématiquement.

Finalement, puisque d'une certaine manière nous cherchons à détecter des MUSes, nous pouvons aussi utiliser la rotation de modèles telle qu'utilisée dans l'outil MUSER (Belov et Marques-Silva 2011) (voir section 4.4.2). Plus précisément, lorsqu'une clause de transition est détectée, il pourrait être envisageable d'utiliser la rotation de modèles sur l'interprétation retournée par le solveur afin de chercher une clause de transition appartenant à Γ telle que le modèle obtenu par rotation satisfasse plus de clauses de Ψ.

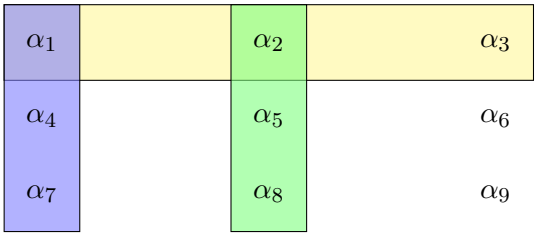
Dans cette section, nous nous sommes principalement intéressés au problème du calcul d'un seul MSS. Cependant, pour de nombreuses applications il est souvent nécessaire de calculer l'ensemble des MSSes. Nous pourrions nous appuyer sur CMP pour énumérer cet ensemble. Cependant, cela nécessiterait de calculer itérativement chaque MSS en prenant soin de le bloquer avant de continuer le calcul. Dans la suite, nous montrons qu'il est possible de factoriser certains MSSes et ainsi de ne pas les énumérer tous explicitement.

4.4 Rotation de modèles pour l'énumération de MSSes

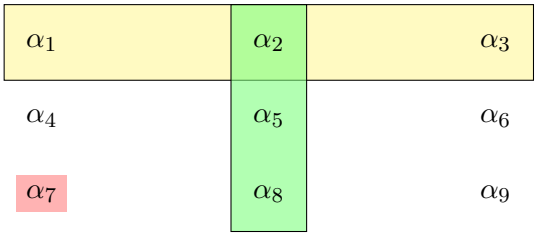
Les approches actuelles de l'état de l'art pour la recherche d'un MCS se basent généralement sur le concept de clause de transition (voir section 4.1). Dans ce travail, nous avons cherché à exploiter les clauses de transition pour détecter des ensembles de MCSes. Pour cela nous avons défini des propriétés qui nous ont permis d'élaborer une méthode originale pour calculer récursivement un ensemble de MCSes. Naturellement, cette méthode s'intègre facilement dans l'approche directe pour l'énumération des MCSes.

4.4.1 Détecter plus de MCSes grâce aux clauses de transition

Avant d'aller plus avant dans la présentation de notre méthode, commençons par considérer la figure suivante où est représentée une formule CNF Σ contenant 9 clauses $\{\alpha_1, \alpha_2, \ldots, \alpha_9\}$ et qui contient 3 MUSes identifiés par des tuiles de couleurs différentes.



Comme nous l'avons déjà souligné en préliminaire de ce chapitre, un MCS peut être vu comme un *hitting set* sur l'ensemble des MUSes, et donc énumérer l'ensemble des MCSes d'une formule revient à énumérer l'ensemble des *hitting sets* des MUSes de cette formule. Supposons à présent que nous ayons choisi de considérer la clause α_7 dans le MCS en construction, ce qui conduit à la situation suivante :



Maintenant supposons que nous souhaitions collecter l'ensemble des MCSes de la formule qui contiennent la clause α_7. De manière naïve, nous pourrions penser qu'il suffit de considérer l'ensemble des *hitting sets* sur la formule restante. Néanmoins, nous pouvons observer facilement en considérant $\{\alpha_1, \alpha_5\}$, que $\{\alpha_1, \alpha_5, \alpha_7\}$ n'est pas un MCS de la formule initiale. En effet, la clause α_1 couvre déjà le MUS bleu et donc la clause α_7 n'est pas utile. Une condition suffisante pour pouvoir étendre $\Gamma \in \texttt{MCS}(\Sigma \setminus \{\alpha_7\})$ avec α_7 est que les clauses restantes d'un MUS contenant α_7 ne soient pas présentes dans Γ. Ce qui revient à combiner α_7 avec les Partial-MCSes de $\langle \Omega \setminus \{\alpha_7\}, \Sigma \setminus \Omega \rangle$, où $\Omega \in \texttt{MUS}(\Sigma)$.

Il est facile de se convaincre qu'un MCS calculé de cette manière est bien un MCS de la formule initiale. En effet, $\Gamma \in \texttt{MCS}(\Sigma \setminus \{\alpha_7\})$ et donc $(\Sigma \setminus \{\alpha_7\}) \setminus \Gamma = \Sigma \setminus (\{\alpha_7\} \cup \Gamma)$ est satisfaisable. Ce qui implique que supprimer $\Gamma \cup \{\alpha_7\}$ de Σ permet de restaurer la cohérence de la formule. En ce qui concerne la minimalité pour l'inclusion de $\Gamma \cup \{\alpha_7\}$, il suffit de voir qu'il n'existe pas $\alpha \in \Gamma \cup \{\alpha_7\}$ telle que $\Sigma \setminus ((\{\alpha_7\} \cup \Gamma) \setminus \{\alpha\})$ est satisfaisable. Deux situations sont à considérer. Soit $\alpha \in \Gamma$ et dans ce cas $(\Sigma \setminus \{\alpha_7\}) \setminus (\Gamma \setminus \{\alpha\})$ est satisfaisable, ce qui contredit le fait que $\Gamma \in \texttt{MCS}(\Sigma \setminus \{\alpha_7\})$. Soit $\alpha = \alpha_7$ et dans ce cas $\Sigma \setminus \Gamma$ est satisfaisable, ce qui contredit le fait qu'il existe $\Omega \in \texttt{MUS}(\Sigma)$ tel que $\Omega \subseteq \Sigma \setminus \Gamma$.

Ce qui fait qu'il est possible d'étendre les MCSes de la sous-formule $\Sigma \setminus \{\alpha\}$ avec la clause α tient au fait que nous recherchons des MCSes qui ne permettent pas de restaurer la cohérence d'une sous-formule de Σ contenant α, mais que supprimer la clause α le permet. Ainsi, l'utilisation d'un MUS est une condition suffisante mais pas nécessaire, il suffit en fait de considérer un incohérent dont α est une clause de transition. La propriété suivante montre que toute clause de transition α d'une sous-formule insatisfaisable $\Sigma' \subseteq \Sigma$ peut être le point de départ d'une famille de MCSes de la formule CNF Σ, chaque membre de cette famille contenant la clause α. Cette propriété permet de capturer une famille de MCSes de Σ qui sont construits à partir de α et étendus avec tous les Partial-MCSes de $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$.

Propriété 2

Soient Σ une formule `CNF` insatisfaisable et une sous-formule $\Sigma' \subseteq \Sigma$ telle que Σ' contient au moins une clause de transition α. Pour tout `Partial-MCS` Γ de $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$, $\Gamma \cup \{\alpha\}$ est un `MCS` de Σ.

Afin de pouvoir l'appliquer récursivement, il est nécessaire de travailler un peu la propriété précédente. En effet, pour le moment, nous recherchons les `MCSes` d'une formule en considérant les `Partial-MCSes` d'une sous-formule. En fait, il est possible de modifier la propriété pour considérer en entrée un `Partial-MCS` $\langle \Sigma_1, \Sigma_2 \rangle$ tel que Σ_1 est satisfaisable et $\Sigma_1 \cup \Sigma_2$ est insatisfaisable. Dans ce cas, il faut aussi modifier la notion de clause de transition de telle manière que cette dernière soit sélectionnée dans Σ'_2 telle que $\Sigma'_2 \subseteq \Sigma_2$ et α est une clause de transition de $\Sigma_1 \cup \Sigma'_2$. En effet, la propriété 2 nous permet de conclure que $\Gamma \cup \{\alpha\}$ est un `MCS` de $\Sigma_1 \cup (\Sigma'_2 \setminus \{\alpha\}) \cup (\Sigma_2 \setminus \Sigma'_2) \cup \{\alpha\}$, et donc de $\Sigma_1 \cup \Sigma_2$. Comme $\Gamma \cup \{\alpha\} \subseteq \Sigma_2$, nous avons donc de manière directe que $\Gamma \cup \{\alpha\}$ est un `Partial-MCS` de $\langle \Sigma_1, \Sigma_2 \rangle$. Ainsi nous pouvons facilement adapter la propriété 2 pour traiter le cas où la recherche des `Partial-MCSes` est l'objectif visé.

Corollaire 1

Soit $\langle \Sigma_1, \Sigma_2 \rangle$ un couple de formules `CNF` tel que Σ_1 est satisfaisable et $\Sigma_1 \cup \Sigma_2$ est insatisfaisable. Considérons la sous-formule `CNF` $\Sigma'_2 \subseteq \Sigma_2$ telle que $\Sigma_1 \cup \Sigma'_2$ contient au moins une clause de transition α de $\Sigma_1 \cup \Sigma'_2$. Pour tous les `Partial-MCSes` Γ de $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha\}, \Sigma_2 \setminus \Sigma'_2 \rangle$, $\Gamma \cup \{\alpha\}$ est un `Partial-MCS` de $\langle \Sigma_1, \Sigma_2 \rangle$.

Nous avons vu qu'il est possible d'appliquer la propriété 2 « en profondeur ». Cependant, il est aussi possible d'utiliser cette propriété en largeur en considérant plusieurs clauses de transition. Admettons maintenant que nous calculions deux clauses de transition α_1 et α_2 pour une sous-formule `CNF` donnée $\Sigma'_2 \subseteq \Sigma_2$. Considérons à présent deux `Partial-MCSes` Γ_1 et Γ_2 qui peuvent être construits respectivement à partir de $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha_1\}, \Sigma_2 \setminus \Sigma'_2 \rangle$ et $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha_2\}, \Sigma_2 \setminus \Sigma'_2 \rangle$. À partir du corollaire 1 il est facile de montrer que $\Gamma_1 \cup \{\alpha_1\}$ et $\Gamma_2 \cup \{\alpha_2\}$ sont tous deux des `Partial-MCSes` de $\langle \Sigma_1, \Sigma_2 \rangle$. Il est important de montrer que ces deux `Partial-MCSes` sont différents. Pour cela il suffit de montrer que $\alpha_1 \notin \Gamma_2$. Par définition d'un `Partial-MCS`, nous avons $\Gamma_2 \subseteq \Sigma_2 \setminus \Sigma'_2$. Comme $\alpha_1 \in \Sigma'_2$, il est évident que $\alpha_1 \notin \Gamma_2$ et donc $\Gamma_1 \cup \{\alpha_1\} \neq \Gamma_2 \cup \{\alpha_2\}$. Par conséquent, pour chaque clause de transition, il est possible d'appliquer de manière récursive le corollaire précédent afin de construire plusieurs `Partial-MCSes` distincts.

Propriété 3

Soit $\langle \Sigma_1, \Sigma_2 \rangle$ un couple de formules CNF tel que Σ_1 est satisfaisable et $\Sigma_1 \cup \Sigma_2$ est insatisfaisable. Considérons $\Sigma'_2 \subseteq \Sigma_2$ tel que $\Sigma_1 \cup \Sigma'_2$ contient au moins deux clauses de transition α_1 et α_2 de $\Sigma_1 \cup \Sigma'_2$ ($\alpha_1 \in \Sigma'_2$ et $\alpha_2 \in \Sigma'_2$). Alors, pour tous les Partial-MCSes Γ_1 et Γ_2 qui peuvent être construits respectivement à partir de $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha_1\}, \Sigma_2 \setminus \Sigma'_2 \rangle$ et $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha_2\}, \Sigma_2 \setminus \Sigma'_2 \rangle$, $\Gamma_1 \cup \{\alpha_1\}$ et $\Gamma_2 \cup \{\alpha_2\}$ sont des Partial-MCSes différents de $\langle \Sigma_1, \Sigma_2 \rangle$.

Cette dernière propriété (3) nous permet de dériver et de justifier un algorithme original et récursif décrit en 4.9, appelé TC-MCS, pour *Transition-Clauses-Based Enumeration of MCSes*. Cet algorithme original est une extension de l'algorithme ELS pour calculer cette fois non pas un mais plusieurs MCSes de manière récursive grâce à la détection de clauses de transitions (lignes $9-14$). Afin de simplifier certains traitements énoncés dans la suite, cet algorithme effectue la recherche de MCSes en considérant une formule augmentée avec des sélecteurs. Ainsi, il prend en entrée un couple de formules CNF $\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$. Il renvoie en sortie un ensemble de Partial-MCSes de ce couple. L'ensemble U contient les sélecteurs correspondant aux clauses qui n'ont pas encore été insérées dans le MSS ou bien le MCS en construction. Nous considérons que Σ_1 est satisfaisable et donc, que $\Sigma_1 \cup \Sigma_2^S$ est satisfaisable. Σ_2^S est en effet obtenu à partir de Σ_2 en rajoutant des sélecteurs aux clauses de Σ_2. Rappelons que les clauses dures n'ont pas besoin de sélecteurs puisqu'elles doivent toujours être satisfaites et donc être toujours activées. Les Partial-MCSes de $\langle \Sigma_1, \Sigma_2 \rangle$ sont obtenus directement à partir des Partial-MCSes de $\langle \Sigma_1 \cup \Sigma_2^S, S \rangle$, où S est l'ensemble des clauses unitaires s_i des sélecteurs de la formule augmentée Σ_2^S.

Algorithme 4.9 : TC-MCS($\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$)

Input : $\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$ un couple de formules CNF
Output : Θ un ensemble de Partial-MCSes de $\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$

1 **if** $U = \varnothing$ **then return** $\varnothing$
2 $\Theta \leftarrow \varnothing$;
3 $M^+ \leftarrow \varnothing$;
4 $s_\alpha \leftarrow \top$;
5 **while** $U \neq \varnothing$ ***and*** $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ *est satisfaisable* **do**
6 $\quad M^+ \leftarrow M^+ \cup \{s_\alpha\}$;
7 $\quad s_\alpha \leftarrow$ **choose** $s_\alpha \in U$;
8 $\quad U \leftarrow U \setminus \{s_\alpha\}$;
9 **if** $s_\alpha \notin M^+$ **then**
10 $\quad \delta \leftarrow$ Core$(\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\})$;
11 $\quad T \leftarrow \{s_\alpha\} \cup$ Find-TC(δ);
12 $\quad$ **foreach** $s_\beta \in T \cap M^+$ **do**
13 $\quad\quad \Omega \leftarrow$ TC-MCS$(\langle \Sigma_1 \cup \Sigma_2^S \cup (\delta \setminus \{\neg s_\beta\}), U \cup (M^+ \setminus \delta) \rangle)$;
14 $\quad\quad \Theta \leftarrow \Theta \cup (\beta \times \Omega)$;
15 **return** Θ;

L'algorithme commence par vérifier si U est l'ensemble vide. Dans le cas positif, la procédure retourne $\varnothing$ puisque l'ensemble des Partial-MCSes de $\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$ est vide du fait que $\Sigma_1 \cup \Sigma_2^S$ est

satisfaisable par construction. Cette première étape de l'algorithme est non récursive. Dans le cas négatif, l'ensemble des `Partial-MCSes` Θ calculés et l'ensemble des sélecteurs M^+ sont initialisés à l'ensemble vide. M^+ sauvegarde les sélecteurs des clauses à activer lorsque $\Sigma_1 \cup \Sigma_2^S \cup M^+$ est satisfaisable. s_α est initialisé à $\top$ (symbole de tautologie).

Ensuite, dans la boucle *While* (lignes $5-8$), M^+ est augmenté incrémentalement par une séquence de sélecteurs s_α, qui sont retirés de U. Ce processus est itéré tant que U n'est pas vide et que α n'est pas une clause de transition de $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$. α est une clause de transition lorsque $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ devient insatisfaisable. En effet, tous les sélecteurs insérés jusqu'ici dans M^+ offrent la garantie que $\Sigma_1 \cup \Sigma_2^S \cup M^+$ demeure satisfaisable. Ainsi, lorsque le sélecteur considéré s_α rend la formule insatisfaisable, celui-ci est identifié comme étant le sélecteur d'une clause de transition. À la fin de la boucle, deux cas peuvent se produire : (1) $s_\alpha \in M^+$, dans ce cas de figure, M^+ a capturé tous les sélecteurs de U et la formule d'entrée $\Sigma_1 \cup \Sigma_2^S \cup U$ était en effet satisfaisable. Par conséquent, un ensemble vide est renvoyé en sortie ; (2) $s_\alpha \notin M^+$ et ainsi α est une clause de transition. Dans ce cas, le `CORE` δ de la formule courante insatisfaisable $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ calculé par le solveur `SAT` est récupéré (ligne 10). Évidemment, le sélecteur s_α est inclus dans le `CORE` δ et il constitue une clause de transition de δ. Comme nous l'avons déjà mentionné, le `CORE` renvoyé par un solveur `SAT` moderne est souvent plus petit que la formule prouvée incohérente. Nous préférons ainsi utiliser ce `CORE` au lieu de considérer la formule insatisfaisable en entier, car le `CORE` contient généralement plus de clauses de transition. Il est facile de prouver par contradiction que chaque clause de transition est présente dans un `CORE`. Une fois le `CORE` de $(\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\})$ récupéré, une procédure `Find-TC` est appelée afin d'identifier les clauses de transition de ce dernier. Dans la prochaine section, nous présentons une approche pour réaliser ce processus, mais pour le moment, nous supposons que les clauses de transition sont extraites par une procédure appelée `Find-TC`. Notons que nous n'avons aucune garantie que cette approche soit efficace et qu'elle détecte la clause de transition s_α ; c'est la raison pour laquelle nous insérons directement s_α dans T (T est l'ensemble des clauses de transition) dans le but d'assurer la terminaison de l'algorithme. Ensuite, pour chaque clause de transition $s_\beta \in T \cap M^+$ identifiée, un appel récursif est effectué avec $\langle \Sigma_1 \cup \Sigma_2^S \cup (\Gamma \setminus \{\neg s_\beta\}), U \cup (M^+ \setminus \Gamma)\rangle$ comme entrée. Il est facile de montrer que ces paramètres correspondent aux conditions énoncées dans le corollaire 1. Ainsi, tous les `Partial-MCSes` pouvant être calculés à partir de $\langle \Sigma_1 \cup \Sigma_2^S \cup (\Gamma \setminus \{\neg s_\beta\}), U \cup (M^+ \setminus \Gamma)\rangle$ sont étendus par s_β pour produire des `Partial-MCSes` de la formule fournie en entrée.

`TC-MCS`$(\langle \Sigma_1 \cup \Sigma_2^S, U\rangle)$ peut être inséré dans l'algorithme 4.3 pour l'énumération des `MCSes`. Le nouvel algorithme d'énumération de `MCSes` est décrit en 4.10.

Algorithme 4.10 : `Enum-ELS-RMR`

Input : une formule `CNF` insatisfaisable Σ
Output : tous les `MCSes` de Σ

1 $\Sigma^S \leftarrow \{\alpha \vee \neg s_\alpha | \alpha \in \Sigma\}$; `// avec` s_α `de nouvelles variables`
2 $S \leftarrow \{s_\alpha | \alpha \in \Sigma\}$; `// un ensemble de sélecteurs`
3 $\Delta \leftarrow \varnothing$;
4 **while** $\Sigma^S \cup \Delta$ *est satisfaisable* **do**
5 $\quad \Theta \leftarrow$`TC-MCS` $(\langle \Sigma^S \cup \Delta, S\rangle)$;
6 $\quad$ **foreach** $M^- \in \Theta$ **do**
7 $\quad\quad$ *output*(M^-);
8 $\quad\quad \Delta \leftarrow \Delta \cup (\bigvee_{s_\alpha \in M^-} s_\alpha)$; `// clauses bloquantes`

Dans la section suivante, nous expliquons comment nous pouvons utiliser la rotation de modèle pour rechercher des clauses de transition supplémentaires dans un CORE.

4.4.2 Utilisation de la rotation de modèle

La procédure Find-TC (ligne 11 de l'algorithme 4.9) implémente une méthode permettant de calculer de nouvelles clauses de transition. Une méthode naïve pour la détection de clauses de transition supplémentaires consisterait à calculer tous les MCSes singletons, c'est-à-dire de taille un, de la formule $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ inclus dans M^+. Toutefois, une telle méthode directe et complète peut être très coûteuse en terme de temps de calcul. Pour éviter ce problème, nous proposons d'utiliser une technique incomplète pour rechercher des clauses de transition supplémentaires, basée sur le paradigme de rotation de modèle récursive noté rmr (pour *recursive model rotation*) (Belov et Marques-Silva 2011).

Ce paradigme a été initialement défini dans le contexte du calcul d'un ou plusieurs ensembles minimaux insatisfaisables (MUSes) d'une formule CNF insatisfaisable, où la recherche des clauses de transition supplémentaires de manière rapide peut également avoir un rôle déterminant. Comme dans (Bacchus et Katsirelos 2015), où l'objectif est l'énumération des MUSes, nous tirons avantage de la dualité entre MUSes et MCSes. Toutefois, dans notre cas, nous appliquons le paradigme de rmr pour une tâche différente, qui consiste à énumérer des MCSes. rmr repose sur l'idée suivante : une clause de transition d'une formule CNF Σ est n'importe quelle clause $\alpha \in \Sigma$ tel qu'il existe une interprétation complète μ de Σ qui satisfait $\Sigma \setminus \{\alpha\}$ (et falsifie α, sinon la formule Σ serait satisfaisable). En partant d'un modèle μ de $\Sigma \setminus \alpha$, rmr consiste à inverser successivement les valeurs de vérité des variables de la clause α dans μ et à chaque fois vérifier si la nouvelle interprétation μ' satisfait toutes les clauses de Σ hormis une certaine clause α' de Σ. Dans le cas positif, α' est marquée comme étant une clause de transition de Σ, à condition qu'elle ne soit pas déjà marquée comme telle, et le processus est répété récursivement avec μ' et α'. Le processus rmr est dépeint dans l'algorithme 4.11.

Algorithme 4.11 : rmr (*Recursive Model Rotation*)

Input : une formule CNF Σ ;
T un ensemble de clauses de transition de Σ ;
α une clause de transition de Σ ;
μ un modèle de $(\Sigma \setminus \{\alpha\})$;

```
1 foreach x ∈ Var(α) do
2     μ' ← μ|¬x;                                   // inverser la valeur de x dans μ
3     if falsifie(μ', Σ) = {α'}  and  α' ∉ T then
4         T ← T ∪ {α'};
5         rmr(Σ, T, α', μ');
```

rmr peut ainsi calculer plusieurs clauses de transition une fois qu'une première clause de transition a été identifiée dans la formule. Dans la recherche d'un Partial-MCS, cette situation se produit lorsque nous avons montré qu'il existe un modèle μ de $\Sigma_1 \cup \Sigma_2^S \cup M^+$ et prouvé que $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ est insatisfaisable. μ représente alors l'interprétation initiale à fournir au processus rmr. Les clauses de transition à garder sont uniquement celles appartenant à $\{\beta \in \Sigma_2 \mid s_\beta \in M^+\}$. Il est important de noter que nous nous assurons que les valeurs de vérité des variables sélecteurs ne sont jamais inversées par le processus rmr.

Bien que le processus `rmr` soit en temps polynomial, en pratique cette procédure peut prendre beaucoup de temps de calcul si la base de clauses à explorer est volumineuse. Dans notre cas, une telle situation peut se produire lorsque la taille de l'ensemble des clauses bloquantes Δ devient trop grand. Pour éviter un tel inconvénient, nous montrons que certaines conditions à réunir suffisent pour ne pas prendre en considération les clauses de Δ dans le processus `rmr`.

D'abord, démontrons l'importance que peut avoir Δ dans la recherche des clauses de transition supplémentaires. Considérons $\langle \Sigma^S, S \rangle$ avec $\Sigma^S = \{\neg s_1 \vee a \vee b, \neg s_2 \vee \neg a, \neg s_3 \vee \neg b\}$. Supposons qu'un `MCS` de Σ^S, à savoir $\{s_1\}$, a déjà été calculé. À cette étape $\Delta = \{s_1\}$. Nous allons maintenant itérer le processus et calculer un `MCS` supplémentaire et appeler `TC-MCS` sur $\langle \Sigma^S \cup \Delta, S \rangle$. Admettons que nous arrêtions la boucle principale de `TC-MCS` lorsque $M^+ = \{s_1, s_2\}$ et $s_\alpha = s_3$. La condition à la ligne 9 de l'algorithme 4.9 est satisfaite. Un `CORE` qui est composé de toute la formule est calculé (ligne 10). Ensuite, nous nous mettons à chercher d'autres clauses de transition dans $\Sigma^S \cup S$ au lieu de $\Sigma^S \cup S \cup \Delta$. Ici, il est facile de montrer que s_1, s_2, s_3 sont des clauses de transition et qu'un des `MCSes` sera calculé deux fois. Évidemment, vérifier a posteriori si un `MCS` a déjà été trouvé en consultant Δ peut être coûteux en temps de calcul, et nous voulons l'éviter.

Nous savons que Σ^S et Δ ne partagent pas de littéraux. En effet, Δ est un ensemble de clauses positives composées de variables sélecteurs, tandis que ces variables sélecteurs apparaissent négativement dans Σ^S. Par conséquent, la satisfaction de $\Sigma^S \cup \Delta$ peut être divisée en deux sous-problèmes indépendants suivant une partition $\{P, N\}$ de l'ensemble des sélecteurs, où P et N désignent respectivement l'ensemble des sélecteurs assignés positivement et l'ensemble des sélecteurs assignés négativement.

En effet, si $\Sigma^S \cup \bigwedge_{s \in P} s$ et $\Delta \cup \bigwedge_{s \in N} \neg s$ sont toutes les deux satisfaisables, alors il existe un modèle μ qui satisfait $\Sigma^S \cup \bigwedge_{s \in P} s$. Par construction de Σ^S, quelle que soit l'interprétation considérée, il est toujours possible d'inverser la valeur de vérité des sélecteurs de 1 (vrai) à 0 (faux) et garder l'interprétation résultante μ_P telle que μ_P soit un modèle de Σ^S (ceci est possible car l'affectation des sélecteurs à 0 désactive des clauses de Σ^S). Puisque $P \cap N = \varnothing$, nous pouvons construire une interprétation μ_P^N qui est équivalente à μ_P excepté sur les valeurs de vérité des sélecteurs appartenant à N où nous imposons leur affectation à 0. Il est clair que μ_P^N est aussi un modèle de $\Delta \cup \bigwedge_{s \in N} \neg s$. Par construction, Δ contient uniquement des clauses positives composées de variables sélecteurs. Donc, quel que soit le modèle de $\Delta \cup \bigwedge_{s \in N} \neg s$ considéré, il est toujours possible d'inverser la valeur de vérité de certains sélecteurs de 0 à 1 et de garder cette interprétation comme modèle de Δ. Puisque tous les modèles de $\Delta \cup \bigwedge_{s \in N} \neg s$ doivent satisfaire $\bigwedge_{s \in N} \neg s$, nous pouvons construire l'interprétation $\bigwedge_{s \in N} \neg s \wedge \bigwedge_{s \in P} s$ qui satisfait $\Delta \cup \bigwedge_{s \in N} \neg s$. Comme Δ est formée uniquement de variables sélecteurs, il est facile de montrer que μ_P^N satisfait également $\Delta \cup \bigwedge_{s \in N} \neg s$. En conséquence, μ_P^N satisfait $\Sigma^S \cup \Delta \cup \bigwedge_{s \in P} s \cup \bigwedge_{s \in N} \neg s$. C'est ce principe que capture la propriété suivante :

Propriété 4

Soient $\{P, N\}$ une partition de S, Σ^S une formule `CNF` augmentée avec un ensemble de sélecteurs S et Δ une formule `CNF` composée uniquement de variables sélecteurs telle que toutes ses clauses sont positives. $\Sigma^S \cup \Delta \cup \bigwedge_{s \in P} s \cup \bigwedge_{s \in N} \neg s$ est satisfaisable si et seulement si $\Sigma^S \cup \bigwedge_{s \in P} s$ et $\Delta \cup \bigwedge_{s \in N} \neg s$ est satisfaisable.

La construction d'un MCS peut se voir de manière implicite comme un calcul de bi-partition de l'ensemble des sélecteurs S en $\{M^+, M^-\}$ tel que $\Sigma^S \cup \bigwedge_{s\in M^+} s$ et $\Delta \cup \bigwedge_{s\in M^-} \neg s$ soient satisfaisables. Le principe est de déplacer autant que possible des sélecteurs de M^- à M^+ tout en maintenant la satisfaisabilité dans $\Sigma^S \cup \bigwedge_{s\in M^+} s$ et $\Delta \cup \bigwedge_{s\in M^-} \neg s$. D'après la propriété précédente, il est facile de prouver que si la bi-partition $\{M^+, M^-\}$ de S vérifie que $\Sigma^S \cup \bigwedge_{s\in M^+} s$ et $\Delta \cup \bigwedge_{s\in M^-} \neg s$ sont satisfaisables, alors si nous déplaçons un élément s' de M^- à M^+ tel que $\Sigma^S \cup \bigwedge_{s\in M^+\cup\{s'\}} s$ est satisfaisable, alors $\Delta \cup \bigwedge_{s\in M^-\setminus\{s'\}} \neg s$ l'est aussi (comme toutes les clauses de Δ sont positives, l'affectation d'un sélecteur positif ne peut pas rendre la formule Δ insatisfaisable).

Maintenant, afin d'éviter de considérer Δ dans la procédure rmr, nous proposons le processus suivant. D'abord, calculer une bi-partition $\{M^+, M^-\}$ de S qui assure que $\Sigma^S \cup \bigwedge_{s\in M^+} s$ et $\Delta \cup \bigwedge_{s\in M^-} \neg s$ sont satisfaisables. M^+ est utilisé comme ensemble initial à étendre pour avoir un MSS et toutes les clauses de M^- sont marquées comme étant candidates pour être identifiées comme des clauses de transition. Ensuite, à chaque fois qu'un nouveau sélecteur est ajouté dans M^+, $\Sigma^S \cup \bigwedge_{s\in M^+} s$ est satisfaisable. Notons qu'ajouter un élément dans M^+ c'est dans un certain sens « déplacer » cet élément de M^- à M^+. Par conséquent, $\Sigma^S \cup \bigwedge_{s\in M^+} s$ et $\Delta \cup \bigwedge_{s\in M^-} \neg s$ sont tous les deux satisfaisables. Ainsi, les sélecteurs qui ont été marqués sont garants de la satisfaisablité de Δ. Puisque M^+ est construit de manière à ce que $\Sigma^S \cup \bigwedge_{s\in M^+} s$ soit satisfaisable, il devient inutile de vérifier la satisfaisabilité de Δ pendant le déroulement du processus rmr quand nous interdisons que les clauses marquées dans M^+ soient sélectionnées comme clauses de transition.

4.4.3 Discussion

Avec cette contribution, nous avons introduit une nouvelle approche qui permet d'accélerer l'énumération de MCSes en factorisant certains parmi eux. Les expérimentations que nous avons conduites dans (Grégoire et al. 2018a) ont montré que l'utilisation de cette technique permet d'énumérer plus de MCSes et cela plus rapidement. Nous avons aussi montré que cette approche pouvait être combinée avec la méthode de *caching* proposée par Previti et al. (2017). Les expérimentations que nous avons conduites dans ce contexte ont démontré que ces deux approches pouvaient être combinées avec succès.

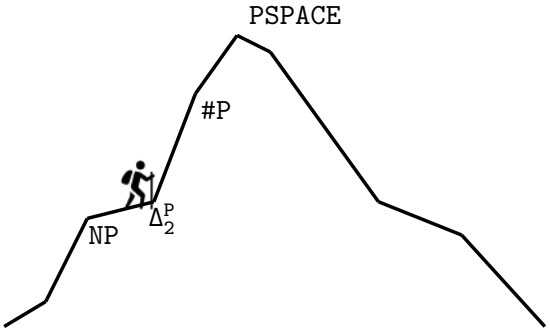
En parallèle de nos travaux, Previti et al. (2018) ont proposé une nouvelle méthode d'énumération de MCSes qui s'appuie sur le modèle d'extraction de LBX (Mencía et al. 2015). Comme nous l'avons déjà mentionné dans la section précédente, le schéma algorithmique utilisé dans LBX est sensiblement différent de celui utilisé dans les approches d'extraction de MCSes classiques, lesquelles s'appuient sur la notion de clause de transition. Cependant, il est tout de même possible de retrouver la notion de clauses de transitions en considérant les clauses qui sont identifiées comme faisant partie du MCS pendant le processus d'extraction de LBX. En effet, comme décrit en préliminaire de ce chapitre (voir l'algorithme 4.2), LBX cherche un ensemble de littéraux Π impliqués par une sous-formule $\Sigma' \subseteq \Sigma$ en considérant incrémentalement un littéral $\ell \in \alpha$ avec $\alpha \in \Sigma \setminus \Sigma'$ et en vérifiant si ℓ est impliqué par Σ'. Pour cela, un solveur SAT est utilisé pour tester la satisfaisabilité de $\Sigma' \wedge \tilde{\ell}$. Si $\Sigma' \wedge \tilde{\ell}$ est insatisfaisable alors ℓ est ajouté à Π et les clauses falsifiées par Π sont supprimées de Σ. En fait, ces clauses feront partie du MCS retourné. Il est facile de voir que chacune des clauses ainsi supprimée est une clause de transition de Σ'. Nous pouvons donc considérer un modèle de Σ' et utiliser la rotation de modèles afin de construire une famille de MCSes en utilisant notre approche.

Dans sa version actuelle, notre approche d'extraction de `MCSes` n'utilise pas complètement les résultats énoncés dans les propriétés 2 et 3. En effet, pour le moment nous nous sommes focalisés sur une recherche en largeur des `MCSes` en considérant les clauses de transitions. Cependant, il est aussi possible d'exploiter la propriété 2 pour réaliser une recherche en profondeur de bas en haut. Une telle approche pourrait permettre de réduire sensiblement le nombre d'appels à un extracteur de `MCS`. De plus, il pourrait être envisageable de bloquer des familles de `MCSes` partageant un même préfixe au lieu de les bloquer indépendamment. Pour cela, il suffirait d'ajouter de nouvelles variables au problème qui seraient utilisées pour combiner les différents niveaux de `MCSes`. Par exemple, supposons que nous souhaitions combiner le début d'un `MCS` identifié par les sélecteurs $\{s_1, s_4, s_7\}$ avec deux `MCSes` $\{\{s_2, s_3\}, \{s_3, s_4\}\}$ représentés par leurs sélecteurs. Il est facile de voir que $\{\neg s_1 \vee \neg s_4 \vee \neg s_7 \vee a, \neg a \vee \neg s_2 \vee \neg s_3, \neg a \vee \neg s_3 \vee \neg s_4\}$ exprime la même information que $\{\neg s_1 \vee \neg s_4 \vee \neg s_7 \vee \neg s_2 \vee \neg s_3, \neg s_1 \vee \neg s_4 \vee \neg s_7 \vee \neg s_3 \vee \neg s_4\}$, mais avec moins de littéraux.

Toujours dans cette idée de combiner des `MCSes`, nous pourrions exploiter la décomposition en composantes connexes pour accélérer l'extraction des `MCSes` de la formule donnée. Deux stratégies s'offrent alors à nous. Soit nous utilisons la décomposition de manière opportuniste, c'est-à-dire que nous vérifions si la formule courante ne possède pas de composantes connexes. Dans ce cas nous n'avons rien à faire. Soit nous utilisons une stratégie plus agressive en considérant par exemple un arbre de décomposition pour guider l'heuristique de choix de clauses. Le fait de pouvoir exploiter la décomposition n'est pas anodin car les `MCSes` de la conjonction de deux formules Σ_1 et Σ_2 ne partageant pas de variables sont précisément les unions des `MCSes` de Σ_1 avec les `MCSes` de Σ_2. Si nous voulons faire une analogie avec l'arithmétique élémentaire, c'est comme si les approches actuelles ne pouvaient utiliser que l'addition pour réaliser des opérations. Ce qui en pratique est très limitant, et surtout moins compact.

Finalement, la propriété 4 ouvre la voie à une certaine parallélisation de la procédure d'énumération. Lorsqu'un ensemble de clauses de transitions est détecté, la propriété 4 nous assure qu'il est possible de générer des familles de `MCSes` différents en marquant certaines clauses (sélecteurs). Ainsi, il est tout à fait envisageable, à partir de chaque clause de transition identifiée, de distribuer le travail à différentes unités de calcul. L'un des désavantages de cette approche est que nous ne pouvons distribuer le travail qu'à partir de clauses de transition, ce qui pourrait limiter le nombre de familles de `MCSes` pouvant être calculées en parallèle. Cependant, cette procédure peut être réalisée de manière récursive. Nous espérons donc pouvoir augmenter le nombre de `MCSes` pouvant être calculés en parallèle, tout en limitant au maximum le nombre d'unités de calcul qui sont en situation de famine.

4.5 Conclusion



Ainsi s'achève cette seconde étape. Elle a permis de montrer qu'il est possible d'utiliser les solveurs SAT comme oracle afin de résoudre des problèmes au-delà de NP. Plus précisément, nous nous sommes focalisés dans ce chapitre sur le problème d'extraction d'un MCS d'une formule donnée, problème qui est dans la classe de complexité $\Delta_2^{\mathtt{P}}$. Cela a été l'occasion d'étudier le comportement des solveurs SAT dans un contexte où ils devaient être utilisés de manière incrémentale. Nous avons observé qu'ajouter des sélecteurs, afin de permettre une réutilisation des clauses apprises entre les différents appels du solveur, pouvait perturber le solveur et ainsi réduire ses performances. Pour éviter cela, nous avons proposé différentes améliorations pouvant être greffées aux solveurs SAT.

Dans ce chapitre, nous avons aussi proposé CMP un nouveau schéma algorithmique pour l'extraction d'un MCS qui, contrairement aux approches de l'état de l'art, se focalise sur la recherche de sous-formules insatisfaisables à restaurer. Les expérimentations que nous avons conduites dans le cadre de l'évaluation de CMP ont montré que, bien qu'ayant une complexité algorithmique quadratique en fonction du nombre n de clauses, CMP était très compétitif sur un large ensemble de problèmes. De plus, notre étude expérimentale a montré que CMP est plus efficace que beaucoup d'autres approches basées sur des schémas algorithmiques ayant des complexités linéaires en la taille de la formule d'entrée. Une analyse en profondeur du comportement de notre méthode a permis de montrer qu'en pratique, le nombre d'appels à l'oracle SAT faits par CMP reste toujours significativement plus petit que n pour toutes les instances pour lesquelles nous avons réussi à calculer un MCS.

Cette étape a aussi été l'occasion d'aller un peu plus loin que $\Delta_2^{\mathtt{P}}$, en considérant le problème d'énumération de MCSes. Pour réaliser cela, nous avons démontré deux propriétés faisant intervenir la notion de clause de transition, et qui permettent de créer des familles de MCSes. Ces propriétés ont servi à élaborer une approche qui, contrairement aux méthodes de l'état de l'art qui calculent indépendamment chaque MCS, tente d'accélérer l'énumération en regroupant certains MCSes. Nos expérimentations ont permis de montrer que cette technique améliore les performances des meilleures approches pour l'énumération des ensembles minimaux rectificatifs et qu'elle est compatible avec la technique de *caching* proposée par Previti et al. (2017).

Nos travaux autour de l'amélioration des solveurs SAT pour l'extraction de MCSes ont mis en évidence à quel point il est important de considérer la nature du problème à traiter afin d'améliorer les performances des solveurs. Néanmoins, l'utilisation de sélecteurs est vraiment spécifique aux problèmes d'identification d'un sous-ensemble de clauses de la formule qui respectent une certaine propriété. De manière générale, les approches qui se basent sur une utilisation intensive des solveurs SAT s'appuient sur le schéma algorithmique CEGAR (*Counter-Example Guided Abstraction Refinement*) (Clarke et al. 2003), où la problématique soulevée par l'utilisation de sélecteurs n'apparaît pas. Dans ce type d'approche une « approximation » du problème initial est donnée à résoudre au solveur et, en fonction de la réponse retournée par ce dernier, le problème est résolu dans son ensemble ou il est nécessaire d'affiner l'approximation et de recommencer le processus. Il est à noter que les problèmes ainsi générés au fil des itérations seront généralement de plus en plus difficiles.

Il est connu que les performances des solveurs SAT sont très dépendantes des paramètres utilisés (Hutter et al. 2017). Ainsi, dans le contexte de SAT incrémental, le fait que les problèmes soient de même nature laisse entrevoir la possibilité d'utiliser les problèmes « plus faciles » afin de préparer le solveur à résoudre les problèmes les plus difficiles. Afin d'exploiter cette information, nous pourrions allouer une certaine quantité de ressources à la résolution d'un problème. Ensuite, tant que le processus de résolution nous fournit des instances qui peuvent être résolues avec la quantité de ressource attribuée, nous ne modifions pas le solveur. Si nous obtenons une instance

plus difficile, alors nous reconsidérons l'ensemble des problèmes déjà résolus par le solveur et nous essayons d'optimiser les paramètres afin de minimiser le temps de résolution de l'ensemble des problèmes. De cette manière, nous espérons que la configuration ainsi trouvée sera adaptée a la résolution des problèmes difficiles.

Dans le cadre de ce chapitre, nous nous sommes principalement intéressés au problème d'extraction de `MCSes`. Cette notion est généralement utilisée pour modéliser le processus de raisonnement d'un agent crédule. Pour modéliser le processus de raisonnement d'un agent sceptique, il faut aller un peu plus loin et considérer les conséquences logiques pouvant être déduites à partir des informations n'appartenant à aucun `MCS`. Cependant, tester si une clause fait partie d'un `MCS` d'une formule donnée est un problème $\Sigma_2^{\mathtt{P}}$-difficile (Eiter et Gottlob 2002). Bien que ce problème soit central en intelligence artificielle, il n'existe pas à notre connaissance d'outils dédiés à ce type de problème. Néanmoins, dans (Lagniez et al. 2015) nous avons proposé une méthode naïve consistant à énumérer l'ensemble des `MCSes` afin de vérifier si un argument était sceptiquement accepté.

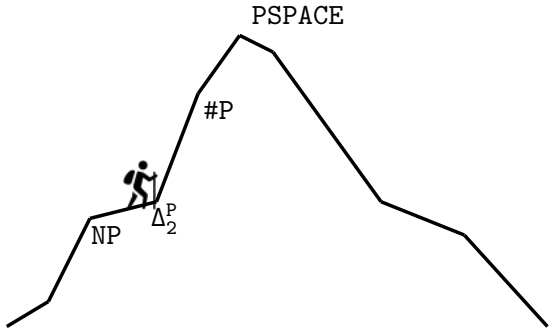
Le fait d'utiliser l'énumération afin de vérifier si une clause est sceptiquement acceptée est assez problématique d'un point de vue pratique. En effet, puisque ce problème est $\Sigma_2^{\mathtt{P}}$-difficile, il existe un algorithme qui fonctionne en espace polynomial et qui est capable de décider le problème. Néanmoins, lorsque nous observons en détail les approches d'énumération de `MCSes`, il apparaît que ces approches nécessitent un espace mémoire qui est au moins égal au nombre de `MCSes` (il faut bloquer chaque `MCS` indépendamment). Puisque le nombre de `MCSes` peut être exponentiel en fonction de la taille de la formule, une telle approche ne fonctionne pas en espace polynomial. Une de nos perspectives est ainsi de développer un outil permettant de réaliser l'inférence sceptique tout en essayant de limiter l'espace mémoire nécessaire à cette procédure. Pour cela, nous pourrions dans un premier temps essayer de réduire l'espace de recherche à explorer en bloquant certains ensembles de clauses satisfaisables, et ainsi éviter de considérer certains `MSSes`, qui permettent d'inférer un littéral de la clause que nous souhaitons tester.

Avec le problème d'énumération de `MCSes`, nous avons pu apercevoir les difficultés qui pouvait survenir dans la suite de notre voyage. En effet, il semble assez clair qu'une approche naïve qui consisterait à calculer chaque modèle d'une formule `CNF` pour en compter le nombre serait inefficace. Comme le laisse envisager nos travaux, il est nécessaire dans un tel contexte d'essayer de regrouper des informations et d'exploiter la structure du problème. Nous présentons dans le chapitre suivant différents langages de représentation permettant de réaliser de manière « efficace » le comptage et l'énumération de modèles d'une formule `CNF`. Plus précisément, nous y étudions en détail les mécanismes permettant de compiler efficacement dans ces langages une formule représentée en `CNF`.

Chapitre 5

Contributions à la compilation de connaissances et au comptage de modèles

Sommaire



La prochaine étape de notre ascension semble difficile. En effet, d'un point de vue théorique le théorème de Toda (1989) indique que $\mathtt{PH} \subseteq \mathtt{P}^{\#\mathtt{P}}$, et donc que tout problème de PH peut être résolu en temps polynomial (déterministe) pour autant qu'un oracle de comptage #P existe. Ceci laisse penser que le comptage de modèles est une requête particulièrement difficile du point de vue du calcul. Ainsi, il semblerait que les difficultés que nous avons rencontrées jusqu'à présent pour atteindre ce palier ne soient rien par rapport à ce qui nous attend. Faut-il s'en effrayer ? Ma réponse à cette question est : pas forcément. En fait, la théorie nous dit aussi qu'il n'existe pas d'algorithmes efficaces pour traiter tous les problèmes de NP (à part si $\mathtt{P} = \mathtt{NP}$, et dans ce cas il faudrait encore se poser la question de la difficulté du problème de comptage). Cependant, nous sommes actuellement capables de traiter de manière pratique des problèmes industriels dans NP contenant des millions de variables et de clauses.[4] Ce succès est principalement dû au fait que les problèmes considérés ont une bonne structure. Qu'en est-il du problème de comptage de modèles ? En fait, il existe de nombreuses familles d'instances où le problème SAT est connu pour être facile (formules 2-SAT ou Horn-SAT par exemple) et où le problème de comptage associé est aussi difficile que le cas général. Il semblerait que ce ne soit pas dans cette direction qu'il faille se diriger. Si nous voulons nous en sortir, il faut sans doute aller plus loin dans l'analyse de la structure des instances.

Une manière de quantifier à quel point une instance donnée est « facile » pour le comptage consiste à analyser la structure du graphe (où de l'hypergraphe) de contraintes associé au problème. À partir de ce graphe il est possible de mesurer certains paramètres issus de la théorie des graphes (Samer et Szeider 2010, Slivovsky et Szeider 2013, Capelli et al. 2014, Sæther et al. 2015, Paulusma et al. 2016), comme la largeur d'arbre, afin de décider s'il est envisageable d'attaquer le problème de manière pratique. En fait, pour de nombreux paramètres, le problème de comptage est FPT (*Fixed-Parameter Tractable*), c'est-à-dire qu'il existe un algorithme exponentiel dans la valeur du paramètre et non plus exponentiel dans le nombre de variables de l'instance (Samer et Szeider 2009). Ainsi, si l'instance considérée a une « petite » largeur d'arbre, nous savons qu'il sera possible de compter son nombre de modèles dans des temps acceptables.

Le fait que le comptage de modèles soit FPT pour de nombreuses mesures de graphe permet-il de décider si une instance donnée pourra être résolue ? Malheureusement, pour de nombreuses raisons cela n'est pas le cas. Tout d'abord, ces mesures ne prennent pas du tout en compte la sémantique du problème. Par exemple, il est facile de construire une formule dont l'incohérence peut être prouvée par propagation unitaire et qui a une très grande largeur d'arbre. Dans ce cas, une procédure qui s'appuierait uniquement sur une mesure de la structure du graphe ne serait pas capable de donner le nombre de modèles, qui est simplement 0. Une autre problématique est que même si un graphe à une petite largeur d'arbre, il n'est pas toujours possible d'appliquer la bonne machinerie permettant de ne pas exploser en mémoire. En effet, en supposant que le nombre de variables de l'instance soit 1000 et qu'une des mesures de graphe donne 10, il est clair qu'un algorithme avec 10^{30} étapes de calcul n'est pas applicable en pratique. Finalement, l'algorithmique FPT utilisée pour compter le nombre de modèles d'une formule ne permet pas de réutiliser l'historique de calcul. En effet, la sémantique de la formule conduit de manière générale à reconsidérer plusieurs fois des formules équivalentes sous des conditionnements différents. Considérons par exemple un problème où nous avons à choisir entre plusieurs camions $\{c_1, c_2, \ldots, c_k\}$ pour réaliser un ensemble de tâches $\{t_1, t_2, \ldots, t_k\}$, et que ces véhicules soient interchangeables. Dans ce cas, si nous assignons t_1 à c_1 et t_2 à c_2 ou t_1 à c_2 et t_2 à c_1, nous avons les mêmes sous-problèmes résultants et il n'est donc pas nécessaire de calculer le nombre de modèles pour chacun des sous-problèmes.

4. Voir les résultats des dernières compétitions SAT : http://www.satcompetition.org/.

En fait, en pratique, les compteurs de l'état de l'art ne sont généralement pas basés sur ces algorithmes FPT. Ils s'appuient principalement sur l'exploration de l'espace de recherche de la formule *via* l'utilisation d'une procédure de type DPLL. En plus de la propagation unitaire, ces approches exploitent l'analyse de conflits afin de réduire l'espace de recherche à visiter, sauvegardent chaque sous-formule calculée dans une table de hachage (*caching*), et finalement essaient de décomposer la formule en composantes connexes disjointes de manière à considérer chaque sous-formule indépendamment des autres. Le fait que nous n'utilisions pas la machinerie FPT ne signifie pas forcément que nous nous éloignons des mesures de graphes. En fait, la possibilité de calculer le nombre de modèles de différentes composantes connexes disjointes permet de produire une méthode basée sur le parcours d'un arbre DPLL où l'heuristique utilisée s'appuie sur un arbre de décomposition (cette décomposition permet de définir la largeur d'arbre du graphe). Ainsi, nous avons les mêmes garanties que les approches FPT, mais nous pouvons aussi exploiter la propagation unitaire, le *caching* et l'analyse de conflits.

Le fait que nous puissions construire une heuristique s'appuyant sur un arbre de décomposition n'implique pas forcément que c'est la bonne approche. D'une part, la largeur d'arbre peut être élevée et donc l'arbre de décomposition associé sans grand intérêt, et d'autre part, l'utilisation d'une telle approche nécessite l'utilisation d'une heuristique partiellement statique (une fois que l'arbre de décomposition est choisi, il est nécessaire de le suivre, faute de perdre toute garantie quant à la taille de l'espace de recherche visité). Une manière de traiter le premier problème consiste à retravailler la formule afin de lui donner une meilleure « tête ». Pour cela, nous avons proposé dans (Lagniez et al. 2016b, Lagniez et Marquis 2017b), avec Pierre Marquis et Emmanuel Lonca, différentes approches de pré-traitements qui vont dans ce sens. Ces prétraitements, présentés dans la section 5.1, permettent soit de réduire la taille de la formule en supprimant des clauses ou des littéraux de ces dernières, soit d'éliminer des variables lorsque celles-ci sont définissables. En ce qui concerne le problème ayant trait à l'heuristique, avec Pierre Marquis, nous avons proposé dans (Lagniez et Marquis 2017a), une heuristique dynamique dont l'objectif est la décomposition de la formule. Cette heuristique, ainsi qu'une nouvelle méthode de *caching* représentant finement et de manière compacte les entrées du *cache*, est présentée à la section 5.3.

Dans le cadre de nos travaux, nous avons aussi exploré d'autres pistes autour de la problématique du comptage de modèles. Dans la section 5.4, nous présentons le problème de compter le nombre de modèles d'une formule où certaines variables doivent être oubliées (appelé *projected model counting*) et un nouvel algorithme récursif, réalisé en collaboration avec Pierre Marquis et publié dans (Lagniez et Marquis 2019a), pour attaquer ce problème de manière pratique. Avant de conclure ce chapitre, nous présentons aussi un nouveau langage de compilation, élaboré en collaboration avec Frédéric Koriche, Pierre Marquis et Samuel Thomas et publié dans (Koriche et al. 2013), permettant le comptage de modèles en temps polynomial par rapport à la taille de la structure. L'avantage de l'approche à base de compilation, par rapport à l'approche directe, est qu'il est possible de réaliser plusieurs requêtes de comptage « rapidement », c'est-à-dire si la forme compilée est concise, une fois que la formule est compilée (ce qui n'est pas le cas avec une approche directe).

Avant de présenter les travaux sus-cités, nous commençons ce chapitre par de brefs rappels au sujet du comptage de modèles et de la compilation de connaissances.

5.1 Préliminaires sur le comptage de modèles et la compilation de connaissances

Dans le cadre de ce chapitre nous nous intéressons principalement au problème de comptage de modèles d'une formule représentée en logique propositionnelle. Ainsi, nous commençons par présenter l'algorithmique permettant de calculer cette valeur de manière directe, c'est-à-dire que le compteur de modèles ne sauvegarde rien une fois qu'il a terminé son processus de comptage. Ensuite, nous présentons une approche un peu différente qui s'appuie sur la compilation de connaissances. Nous nous focalisons alors sur les critères à prendre en compte pour comparer les différents langages cibles possibles pour la compilation et dresser ainsi une carte de compilation pour les formules propositionnelles. Il s'agit d'une part de l'efficacité temporelle du langage, c'est-à-dire l'existence ou la non-existence (prouvée absolument ou sous les hypothèses usuelles de la théorie de la complexité) d'algorithmes en temps polynomial pour réaliser différents traitements, principalement de répondre à telle ou telle requête (la requête qui nous intéresse ici principalement est la requête de comptage) et effectuer telle ou telle transformation sur des représentations (la transformation qui va surtout nous intéresser est le conditionnement) issue du langage cible considéré. Il s'agit aussi de l'efficacité spatiale (relative) du langage, c'est-à-dire la possibilité et/ou l'impossibilité (là encore, prouvée absolument ou sous les hypothèses usuelles de la théorie de la complexité) de traduire les représentations d'un langage en représentation d'un autre en espace polynomial. Nous rappelons ensuite les définitions de plusieurs langages cibles introduits jusqu'ici, en particulier ceux de la famille `d-DNNF` mais aussi les formules affines `AFF` et les disjonctions de formules affines `AFF`$[\vee]$ et précisons leur efficacité temporelle et spatiale telle qu'elles ont été identifiées dans la littérature.

5.1.1 Le comptage de modèles

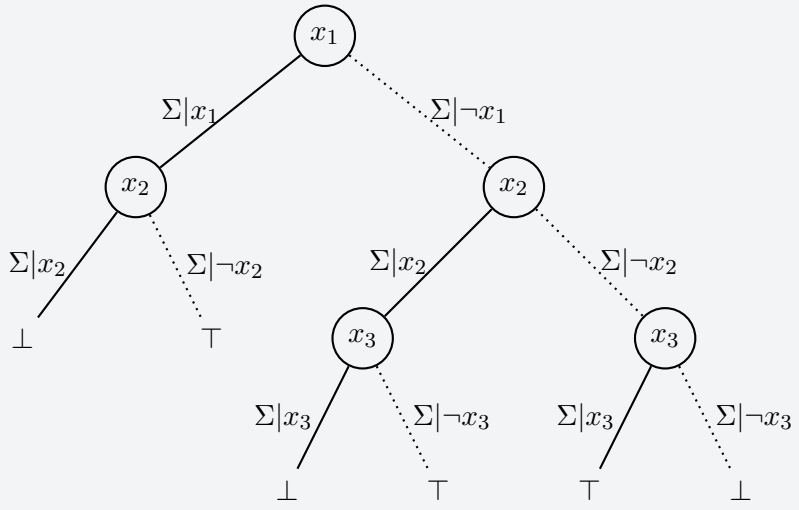
Le problème du comptage de modèles, aussi appelé #SAT, consiste à déterminer le nombre de modèles d'une formule propositionnelle quelconque. Autrement dit, de calculer le nombre d'interprétations distinctes qui évaluent la formule à vrai. Comme nous l'avons déjà souligné au chapitre 2.1.2, ce problème est le problème #P-complet de référence. Pour plus d'informations sur le sujet, le lecteur pourra consulter (Gomes et al. 2009). Il est important de noter que nous ne traitons pas du problème de comptage de modèles approchés dans ce manuscrit, le lecteur intéressé pourra consulter (Wei et al. 2004, Wei et Selman 2005, Gomes et al. 2007, Meel et al. 2016, Soos et Meel 2019, Sharma et al. 2019).

Le comptage de modèles est un problème très intéressant car il apparaît dans beaucoup d'applications, telles la planification et la configuration. Cependant comme déjà évoqué, le théorème de Toda (1989) donne une bonne indication quant à la difficulté des problèmes #P-complets. Bien que ce problème soit en théorie très difficile à appréhender, il existe tout de même des approches permettant de compter le nombre de modèles d'une formule `CNF` de manière exacte (Sang et al. 2004, Thurley 2006). Ces approches s'appuient généralement sur l'un des algorithmes les plus populaires pour `SAT`, l'algorithme `DPLL`. Comme nous avons pu le voir à la section 3.1.1, c'est un algorithme d'exploration systématique qui recherche dans l'espace des interprétations jusqu'à trouver un modèle à la formule ou montrer qu'il n'existe pas de modèle pour cette formule. En particulier, étant donnée une formule `CNF` Σ, l'algorithme choisit une variable x_i de Σ et est appelé récursivement en considérant la formule $\Sigma_{|x_i}$, puis $\Sigma_{|\neg x_i}$. Enfin, il décide si la formule est cohérente ou non lorsque toutes les variables de Σ sont assignées à une valeur de vérité. Cette approche peut facilement être étendue pour calculer le nombre exact de modèles d'une formule

CNF. Il suffit de ne pas arrêter la recherche au premier modèle trouvé. En continuant la recherche ainsi, nous cherchons l'ensemble des modèles de la formule. Il suffit d'augmenter un compteur à chaque feuille $\top$ rencontrée dans l'arbre de recherche pour obtenir le nombre de modèles de la formule CNF. Ceci nous donne un algorithme naïf pour compter exactement le nombre de modèles d'une formule.

Exemple 21

La figure suivante représente un arbre de recherche complet pour la formule CNF $\Sigma = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$ portant sur les trois variables $\{x_1, x_2, x_3\}$. Cet arbre de recherche montre l'ensemble des chemins parcourus lors de la recherche exhaustive par l'algorithme DPLL sur la formule Σ.



Ainsi, un chemin entre la racine et un nœud représente une interprétation (partielle) des variables de Σ. Chaque feuille représente le résultat de la recherche lorsque l'interprétation représentée par le chemin entre la racine et la feuille est appliquée à la formule booléenne. Quand la feuille est étiquetée par $\bot$, l'interprétation partielle n'est pas un modèle de la formule, et est un modèle quand elle est étiquetée par $\top$. Le nombre de modèles d'une feuille $\top$ est de 2^{n-t} où n est le nombre de variables de Σ et t le nombre de variables assignées dans l'interprétation partielle calculée. Le nombre de modèles de la CNF Σ correspond donc à la somme des nombres obtenus à chaque feuille $\top$, c'est-à-dire 4 modèles pour l'exemple précédent. Nous appelons cet arbre la trace de la recherche exhaustive effectuée par l'algorithme DPLL.

L'algorithme DPLL étant conçu pour la recherche d'une solution, et non de l'ensemble des solutions, il ne peut pas être utilisé tel quel. Par exemple l'heuristique de choix de variables d'un algorithme DPLL va chercher à obtenir le plus rapidement possible un modèle, tandis qu'un compteur de modèles doit parcourir tout l'arbre de recherche. Ceci réduit fortement l'efficacité du solveur. Plusieurs méthodes ont été développées afin d'améliorer cet algorithme. Les principales techniques utilisées par les algorithmes pour compter le nombre de modèles de Σ résident dans la conservation de la trace du DPLL, l'utilisation d'une heuristique adaptée (voir la sous-section

5.3.2), la technique de *caching* (voir la sous-section 5.3.3) et enfin, la décomposition de la formule pendant la recherche `DPLL`.

L'analyse de composantes connexes permet de découper la formule en plusieurs sous-formules indépendantes, sur lesquelles l'algorithme peut travailler séparément. Puisque la recherche est effectuée sur une sous-formule avec un plus petit nombre de variables, cela permet de réduire fortement les temps de calcul. La recherche de composantes connexes se fait depuis le graphe de contraintes (ou graphe primal) de la formule `CNF`. Soit $\mathcal{G}$ le graphe de contraintes d'une formule `CNF` Σ. Les sommets de $\mathcal{G}$ sont les variables de Σ et une arête entre deux sommets signifie que les deux variables correspondantes apparaissent dans une même clause de Σ. Pour découper la formule Σ en sous-formules indépendantes, il suffit de calculer les composantes connexes $\mathcal{G}_1, \ldots, \mathcal{G}_n$ du graphe $\mathcal{G}$. Autrement dit, $\mathcal{G}$ est découpé en plusieurs sous-graphes tel que pour chaque sommet d'une composante connexe $\mathcal{G}_i$ il n'existe pas d'arête entre ce sommet et une autre composante connexe $\mathcal{G}_j$ avec $i \neq j$. De cette manière nous découpons Σ en plusieurs sous-formules $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$, de telle sorte que deux variables de deux composantes connexes différentes n'apparaissent pas dans une même clause. Chaque sous-formule Σ_i ne contient que les clauses contenant des variables apparaissant dans les sommets de la composante connexe correspondante $\mathcal{G}_i$. Comme les composantes connexes sont disjointes, une clause n'apparaît que dans une seule sous-formule. Ainsi, nous pouvons calculer séparément le nombre de modèles de chaque sous-formule Σ_i. Le nombre de modèles de la formule initiale Σ peut alors être calculé facilement en multipliant le nombre de modèles obtenu pour chaque sous-formule, c'est-à-dire $\#\Sigma = \#\Sigma_1 \times \Sigma_2 \times \cdots \times \#\Sigma_n$. Avec l'approche `DPLL`, après chaque affectation d'une variable, des simplifications peuvent être effectuées. En effet, chaque clause satisfaite par les variables affectées peut être supprimée de la formule, ce qui permet de simplifier le graphe de contrainte. La recherche de composantes connexes peut ainsi être effectuée à chaque nœud du parcours de l'espace de recherche.

Algorithme 5.1 : $\texttt{MC}(\Sigma)$

input : Σ une formule `CNF`
output : le nombre de modèles de Σ

```
1  S ← solve(Σ);
2  if S = {∅} then return 0;
3  if Var(Σ|S) = ∅ then return 2^(Var(Σ)−|S|);
4  if cache(Σ) ≠ nil then return [cache(Σ)];
5  comps ← connectedComponents(Σ);
6  cpt ← 0;
7  foreach Σ_c ∈ comps do
8  |   ℓ ⟵ un littéral tel que var(ℓ) ∈ Var(Σ_c) ;
9  |   cpt ⟵ cpt × (MC(Σ_c|ℓ) + MC(Σ_c|¬ℓ));
10 cache(Σ) ← cpt;
11 return cpt;
```

L'algorithme 5.1 donne le pseudo-code d'un compteur de modèles qui prend en entrée une formule `CNF` Σ et retourne son nombre de modèles. À la ligne 1, `solve` est appelé sur la fomule Σ. Cette procédure peut aller jusqu'à être un solveur `SAT`, la procédure `bcp`, ou juste une approche qui vérifie que la formule ne contient pas de clause vide (cette dernière est la forme de vérification de la cohérence de la formule la plus faible que nous pouvons utiliser afin d'assurer la terminaison

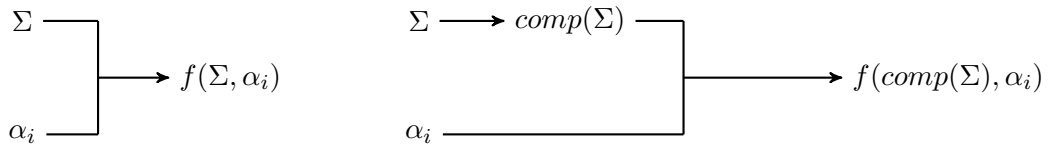
de l'algorithme). S est égal à $\{\varnothing\}$ si Σ est incohérent, et dans ce cas le nombre de modèles est 0 (ligne 2). Sinon S est l'ensemble des littéraux unitaires pouvant être propagés à partir de Σ (il faut garder à l'esprit que `bcp` (Σ) « simplifie » la formule et que cette procédure est toujours appelée dans le processus `solve`). Le second cas de base est considéré à la ligne 3 : si Σ une fois simplifiée par `bcp` ne contient plus de variables, alors le nombre de modèles de la formule est donné $2^{Var(\Sigma)-|S|}$ (nous devons considérer les variables libres de Σ). À la ligne 4, nous testons si la formule courante Σ a déjà été rencontrée ou pas par le passé. Si cela est le cas, alors nous retournons le nombre de modèles stocké. Sinon, à la ligne 5 la fonction connectedComponents retourne un ensemble *comps* de formules `CNF`, représentant les composantes connexes de Σ. À la ligne 6 nous initialisons la variable accumulateur *cpt* où sera stocké le nombre de modèles de la formule Σ. La boucle à la ligne 7 considère alors chaque élément Σ_c de *comps* successivement. Pour chaque composante, nous commençons par sélectionner un littéral ℓ tel que la variable associée appartient à Σ_c. Alors nous calculons récursivement la somme des nombres de modèles des deux sous-formules engendrées par le conditionnement de Σ_c par ℓ et $\neg\ell$, ce résultat est alors multiplié avec la valeur de l'accumulateur *cpt*. Une fois que toutes les composantes ont été traitées, nous associons dans la structure de *cache* la formule Σ au nombre de modèles *cpt* que nous avons trouvé (ligne 10). Finalement, nous retournons *cpt* à la ligne 11.

Une autre approche pour le comptage exact de modèles consiste à compiler la formule `CNF` vers une représentation dans un autre langage permettant de compter plus facilement le nombre de modèles. L'idée principale de la compilation de connaissances est de passer beaucoup de temps sur la partie difficile, qui est la traduction de la formule écrite dans un langage vers un autre langage, et de compenser ce temps grâce à l'exécution de nombreuses requêtes qui seront beaucoup plus simples à résoudre à partir de la représentation obtenue (la forme compilée) dans ce nouveau langage.

5.1.2 Compilation de connaissances

Le formalisme logique offre un cadre puissant pour construire des bases de connaissances modélisant de nombreuses applications concrètes. L'intérêt d'utiliser une telle représentation déclarative est que l'information stockée existe indépendamment des traitements pouvant être mis en œuvre pour l'exploiter. Malheureusement, les requêtes d'interrogation que nous souhaitons réaliser sont le plus souvent calculatoirement difficiles, c'est le cas en particulier quand la base de connaissances prend la forme d'une formule propositionnelle ou d'un réseau de contraintes (vérifier si une formule est une conséquence logique de la base est coNP-complet dans cette situation). Pour pallier cette complexité de calcul, la compilation de connaissances est une approche possible. Elle consiste à traduire la base de connaissances dans un autre formalisme afin de répondre plus efficacement à différents traitements, par exemple à plusieurs requêtes d'interrogation. Ainsi, en logique propositionnelle, si la base de connaissances considérée est décrite par l'ensemble de ses solutions (le langage `MODS`), il est alors facile de déterminer si la formule est cohérente ou encore si une formule construite sur les mêmes variables que la base en est une conséquence logique.

L'apport de la compilation du point de vue computationnel vient du fait que les différents traitements à réaliser sur les connaissances portent souvent sur deux parties distinctes : une partie « fixe » (par exemple la base de connaissances) et une partie « variable » (par exemple les requêtes d'interrogation à prendre en compte). L'idée est simplement de pré-traiter (compiler) la partie « fixe » une bonne fois pour toutes, pour ensuite, quand la partie « variable » est connue, réaliser plus efficacement le traitement voulu. La phase de compilation est appelée la phase *off-line*, elle n'est effectuée qu'une seule fois (nous pouvons donc y consacrer un effort



(a) Approche directe (b) Approche compilée

FIGURE 5.1 – Représentation schématique de l'approche directe et de l'approche compilée.

calculatoire important) et doit préserver les informations utiles à la résolution du problème traité (par exemple, l'ensemble des solutions). Puis les traitements sont effectivement réalisés, en utilisant la forme compilée de la base de connaissances en lieu et place de la base originale. Cette phase est appelée la phase *on-line* (ou phase de requêtes). Plus formellement, la compilation de connaissances porte sur des traitements f à réaliser (f est une certaine fonction), pour lesquels chaque entrée possible prend la forme d'un couple (Σ, α_i) :

— Σ est la partie fixe de l'instance (typiquement, commune à plusieurs parties variables) ;

— α_i est la partie variable de l'instance.

La figure 5.1a schématise l'évaluation directe de f sur l'entrée (Σ, α_i). $f(\Sigma, \alpha_i)$ est le résultat du traitement. La figure 5.1b schématise l'évaluation de f sur l'entrée (Σ, α_i) après compilation de Σ. La partie fixe Σ est d'abord traduite en une forme compilée $comp(\Sigma)$ et l'évaluation de f est réalisée sur l'entrée $(comp(\Sigma), \alpha_i)$. Les résultats retournés par $f(\Sigma, \alpha_i)$ et $f(comp(\Sigma), \alpha_i)$ sont les mêmes quand Σ et $comp(\Sigma)$ sont équivalents.

Pour que l'approche par compilation ait un sens, il faut que le temps mis pour évaluer l'ensemble des $f(comp(\Sigma), \alpha_i)$ quand α_i varie soit au final plus petit que le temps mis pour évaluer l'ensemble des $f(\Sigma, \alpha_i)$ quand α_i varie. En pratique, cela pourra être le cas quand le calcul de $f(\Sigma, \alpha_i)$ est NP-difficile alors que le calcul de $f(comp(\Sigma), \alpha_i)$ peut être réalisé en temps polynomial par rapport à la représentation sous forme compilée. Toutefois, le fait de faire baisser la complexité de l'évaluation de f lors de la phase en ligne quand nous avons compilé Σ ne suffit pas à garantir que l'approche par compilation a de l'intérêt au niveau du problème considéré (i.e., pour ce qui concerne le calcul de f en toute généralité, sans préciser les instances considérées). En effet, la complexité d'un algorithme s'exprime en toute généralité comme une fonction de la taille de son entrée. Si le meilleur algorithme connu pour calculer f selon l'approche directe (donc sans compilation) a une complexité temporelle en $\mathcal{O}(2^n)$ et que le meilleur algorithme connu pour calculer f selon l'approche par compilation a une complexité temporelle en $O(n)$, nous pouvons pas forcément conclure à l'intérêt de cette dernière approche. Ainsi, si la taille $|comp(\Sigma)|$ de la forme compilée de $comp(\Sigma)$ est exponentielle en la taille de Σ, le temps mis en pratique pour calculer $f(comp(\Sigma), \alpha_i)$ (avec un algorithme en temps linéaire) peut très bien excéder celui mis pour calculer $f(\Sigma, \alpha_i)$ (avec un algorithme pourtant en temps exponentiel). Pour cette raison, compiler une base de connaissances donnée par une formule propositionnelle vers le langage cible MODS pour pouvoir l'interroger plus efficacement n'est, en général, pas une bonne idée (le nombre de modèles d'une formule propositionnelle pouvant être exponentiel dans la taille de sa représentation dans des langages usuels, comme le langage CNF).

En théorie, beaucoup de traitements intéressants mais calculatoirement difficiles f ne sont pas compilables dans P, i.e., nous ne connaissons pas de fonction de compilation *comp* qui garantirait d'une part que $|comp(\Sigma)|$ soit toujours polynomiale dans $|\Sigma|$ pour toute entrée Σ possible et d'autre part pour lesquels l'évaluation de f pourrait être réalisée en temps polynomial à partir

de la forme compilée alors qu'elle est NP-difficile sinon P = NP. Pour divers f (comme l'interrogation, même quand les requêtes d'interrogation se limitent à des formules « simples » comme des clauses), nous pouvons même montrer que de telles fonctions *comp* ne peuvent pas exister sauf si NP $\subseteq$ P/poly (une hypothèse jugée peu vraisemblable en théorie de la complexité) (Cadoli et Donini 1997, Cadoli et al. 2002). Cela n'enlève toutefois rien à l'apport que la compilation peut fournir en pratique : ainsi, même lorsqu'il ne peut être assuré que $|comp(\Sigma)|$ soit toujours polynomiale en $|\Sigma|$, si pour Σ donné, $|comp(\Sigma)|$ reste « relativement petit », le temps mis à compiler Σ pourra être compensé. En outre, nous pourrons souvent majorer finement le temps nécessaire pour calculer $f(comp(\Sigma), \alpha_i)$ (et assurer ainsi des garanties de temps de réponse s'il est réduit) alors que le temps nécessaire pour calculer $f(\Sigma, \alpha_i)$ sera très élevé dans le pire des cas. Pour poursuivre l'exemple ci-dessus, si le nombre de modèles de Σ est réduit, compiler Σ dans le langage MODS pour pouvoir interroger Σ plus efficacement a parfaitement du sens : le temps nécessaire pour répondre à une requête d'interrogation est linéaire dans la taille de sa représentation en MODS et la taille de la requête alors qu'aucun algorithme en temps polynomial n'existe pour cela si Σ est représentée dans le langage CNF (et qu'il n'en existe pas, sauf si P = NP). De telles garanties de temps de réponse sont attendues dans diverses applications, en particulier quand il s'agit de fournir des réponses aux sollicitations d'un utilisateur qui interagit à distance avec un système informatique, par exemple via le Web (Astesana et al. 2010).

Différents critères, proposés par Darwiche et Marquis (2002), peuvent être utilisés afin d'évaluer les langages cibles pour la compilation de formules propositionnelles et, sur cette base, de choisir un « bon » langage cible selon l'application traitée. Il s'agit d'une part de critères portant sur l'efficacité temporelle du langage cible considéré $\mathcal{L}$, c'est-à-dire l'existence ou la non-existence d'algorithmes en temps polynomial pour réaliser différents traitements. Ces traitements sont classés en requêtes - les traitements qui extraient de l'information de la forme compilée, par exemple son nombre de solutions - et en transformations - les traitements qui permettent de produire ou de modifier une représentation sous forme compilée. Il s'agit aussi de l'efficacité spatiale (relative) du langage $\mathcal{L}$ par rapport à d'autres langages cibles possibles $\mathcal{L}'$, c'est-à-dire la possibilité et/ou l'impossibilité de traduire les représentations de $\mathcal{L}$ en représentations de $\mathcal{L}'$ équivalentes et de tailles polynomiales, et vice-versa.

Darwiche et Marquis (2002) considèrent huit requêtes de base dans la carte de compilation, à savoir le test de cohérence, le test de validité, le test d'implication, le test d'impliquant, le test d'équivalence, le test d'implication générale, le test de comptage du nombre de modèles et l'énumération des modèles d'une base de connaissances. Plus précisément, soit $\mathcal{L}$ un langage cible de compilation. Les requêtes listées ci-dessus sont définies formellement de la manière suivante :

- **Cohérence (CO) :** $\mathcal{L}$ satisfait la propriété du test de cohérence **CO** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne *vrai* si Σ est cohérente, *faux* sinon ;
- **Validité (VA) :** $\mathcal{L}$ satisfait la propriété du test de validité **VA** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne *vrai* si Σ est valide, *faux* sinon ;
- **Implication clausale (CE) :** $\mathcal{L}$ vérifie la propriété du test d'implication clausale **CE** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et toute clause γ retourne *vrai* si $\Sigma \models \gamma$ et *faux* sinon ;
- **Impliquant (IM) :** $\mathcal{L}$ vérifie la propriété du test d'impliquant **IM** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout terme γ retourne *vrai* si $\gamma \models \Sigma$ et *faux* sinon ;

— **Équivalence (EQ) :** $\mathcal{L}$ satisfait la propriété du test d'équivalence **EQ** si et seulement s'il existe un algorithme en temps polynomial qui pour toutes formules Σ_i et Σ_j de $\mathcal{L}$ retourne *vrai* si $\Sigma_i \equiv \Sigma_j$, *faux* sinon ;

— **Implication générale (SE) :** $\mathcal{L}$ satisfait la propriété du test d'implication générale **SE** si et seulement s'il existe un algorithme en temps polynomial qui pour toutes formules Σ_i et Σ_j de $\mathcal{L}$ retourne *vrai* si $\Sigma_i \models \Sigma_j$ et *faux* sinon ;

— **Comptage de modèles (CT) :** $\mathcal{L}$ satisfait la propriété **CT** de comptage de modèles si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne un entier naturel qui est le nombre de modèles de Σ ;

— **Énumération de modèles (ME) :** $\mathcal{L}$ satisfait la propriété **ME** d'énumération de modèles si et seulement s'il existe un polynôme p et un algorithme qui retourne tous les modèles d'une formule Σ de $\mathcal{L}$ en temps $p(n, m)$ où n est la taille de Σ et m son nombre de modèles (sur les variables de Σ).

En ce qui concerne les transformations, Darwiche et Marquis (2002) proposent de considérer le conditionnement, l'oubli (possiblement borné) et les clôtures (possiblement bornées) par les connecteurs logiques $\wedge$ et $\vee$, ainsi que la négation $\neg$. Formellement, nous avons :

— **Conditionnement (CD) :** $\mathcal{L}$ vérifie la propriété de conditionnement **CD** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout terme cohérent γ retourne une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_{|\gamma}$;

— **Oubli (FO) :** $\mathcal{L}$ vérifie la propriété de l'oubli **FO** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout sous-ensemble de variables X retourne une formule de $\mathcal{L}$ logiquement équivalente à $\exists X.\Sigma$;

L'oubli des variables de X dans Σ consiste à supprimer toute occurrence de X dans Σ tout en maintenant les informations ne concernant pas X que Σ capture. Formellement, il s'agit de la formule notée $\exists X.\Sigma$ qui est la conséquence logique la plus générale de Σ qui soit indépendante de X (i.e., elle équivaut à une formule dans laquelle aucune variable de X n'apparaît). $\exists X.\Sigma$ est unique à l'équivalence logique près ;

— **Oubli d'une variable (SFO) :** $\mathcal{L}$ vérifie la propriété de l'oubli d'une variable **SFO** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout singleton X retourne une formule de $\mathcal{L}$ logiquement équivalente à $\exists X.\Sigma$;

— **Conjonction ($\wedge$C) :** $\mathcal{L}$ vérifie la propriété de la conjonction $\wedge$**C** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à tout ensemble fini de formules $\{\Sigma_1, \ldots, \Sigma_n\}$ de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_1 \wedge \cdots \wedge \Sigma_n$;

— **Conjonction bornée ($\wedge$BC) :** $\mathcal{L}$ vérifie la propriété de la conjonction bornée $\wedge$**BC** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à toute paire de formules Σ_i, Σ_j de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_i \wedge \Sigma_j$;

— **Disjonction ($\vee$C) :** $\mathcal{L}$ vérifie la propriété de la disjonction $\vee$**C** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à tout ensemble fini de formules $\{\Sigma_1, \ldots, \Sigma_n\}$ de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_1 \vee \cdots \vee \Sigma_n$;

— **Disjonction bornée ($\vee$BC) :** $\mathcal{L}$ vérifie la propriété de la disjonction bornée $\vee$**BC** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à toute paire de formules Σ_i, Σ_j de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_i \vee \Sigma_j$;

— **Négation ($\neg$C) :** $\mathcal{L}$ vérifie la propriété de la négation $\neg$**C** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne une formule de $\mathcal{L}$ logiquement équivalente à $\neg\Sigma$.

Des liens formels entre requêtes et transformations (vues comme des propriétés des langages cibles) existent et ont été identifiés dans (Darwiche et Marquis 2002). Ainsi, lorsque certaines propriétés sont satisfaites par un langage, d'autres propriétés sont automatiquement offertes par celui-ci.

Le dernier critère d'évaluation qui nous intéresse concerne l'efficacité spatiale (relative) d'un langage (Gogic et al. 1995). Comme nous l'avons évoqué, elle indique la capacité (relative) d'un langage à représenter succinctement de l'information en comparaison à une autre. Cette efficacité se définit comme suit :

Définition 52 (Efficacité spatiale)

Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages propositionnels. Le langage $\mathcal{L}_1$ est au moins aussi succinct que $\mathcal{L}_2$, noté $\mathcal{L}_1 \leq_s \mathcal{L}_2$, si et seulement si il existe un polynôme p tel que pour tout $\alpha \in \mathcal{L}_2$, il existe $\beta \in \mathcal{L}_1$ qui est logiquement équivalente à α et telle que $\mid \beta \mid \leq p(\mid \alpha \mid)$.

Cette notion d'efficacité spatiale n'impose pas qu'il existe une traduction en temps polynomial de $\mathcal{L}_2$ vers $\mathcal{L}_1$, mais (ce qui est moins restrictif) qu'une traduction en espace polynomial de $\mathcal{L}_2$ vers $\mathcal{L}_1$ existe. En particulier, aucune hypothèse n'a besoin d'être faite sur le temps nécessaire à la traduction. Il est aisé de prouver que la relation $\leq_s$ est réflexive et transitive, elle constitue donc un pré-ordre. Nous pouvons ainsi définir sa partie stricte, i.e., la relation $<_s$ telle que $\mathcal{L}_1 <_s \mathcal{L}_2$ si et seulement si $\mathcal{L}_1 \leq_s \mathcal{L}_2$ et $\mathcal{L}_2 \not\leq_s \mathcal{L}_1$. $\mathcal{L}_1 <_s \mathcal{L}_2$ signifie que $\mathcal{L}_1$ est strictement plus succinct que $\mathcal{L}_2$.

Dans leur papier, Darwiche et Marquis (2002) ont étudié de nombreux langages cibles de compilation développés jusqu'ici en la logique propositionnelle. Néanmoins, dans le cadre de ce manuscrit nous nous limitons à présenter quelques langages qui satisfont **CT**. Pour plus de détails sur les autres langages, le lecteur intéressé pourra se référer, en particulier, à (Darwiche et Marquis 2002, Fargier et Marquis 2008b;a).

Les premiers langages auxquels nous nous intéressons sont des sous-ensembles du langage `NNF` des circuits (ou graphes) dans lesquels les portes sont de type $\wedge$ ou $\vee$ et les négations ne portent que sur les entrées du circuit. Ces dernières sont des variables propositionnelles ou des constantes booléennes ($\top$ ou $\bot$). Formellement :

Définition 53 (`NNF`)

Un formule `NNF` est un graphe fini, non vide, orienté et sans circuit (DAG), ayant une racine unique et où chaque feuille est étiquetée par $\top$, $\bot$ ou une variable propositionnelle x possiblement niée ($\neg x$). Chaque nœud interne est étiqueté par une conjonction $\wedge$ ou une disjonction $\vee$ et peut avoir un nombre quelconque (fini) de fils.

La taille d'une formule Σ `NNF`, notée $|\Sigma|$, est le nombre d'arcs du graphe. Sa hauteur correspond au nombre maximum d'arcs entre la racine et l'une des feuilles du graphe. Pour alléger

l'écriture et quand cela ne crée aucune ambiguité, nous nous autoriserons dans la suite à identifier tout nœud N d'une formule NNF avec la formule enracinée à ce nœud.

Bien que complet pour la logique propositionnelle, le langage NNF ne permet aucune requête intéressante du point de vue de la compilation (ni **CO**, ni a fortiori **CT**). En revanche, l'ajout de restrictions permet de définir des langages sous-ensembles de NNF, offrant d'intéressantes propriétés pour la compilation. Ainsi, le langage DNNF est le sous-ensemble du langage NNF qui satisfait la propriété de décomposabilité.

Définition 54 (Décomposabilité)

Un nœud de conjonction $N_\wedge$ d'une formule NNF est décomposable si et seulement si ses fils $N_1, \ldots, N_k$ ne partagent aucune variable en commun. Formellement, $Var(N_i) \cap Var(N_j) = \varnothing$ pour tout $i \neq j$.

Définition 55 (DNNF)

Une formule DNNF est une formule NNF dans laquelle chaque nœud de conjonction est décomposable.

Le langage DNNF se montre plus attractif que NNF puisqu'il satisfait trois des requêtes qui nous intéressent tout en préservant la transformation **CD**. Ainsi, DNNF satisfait le test de cohérence (**CO**), le test d'implication clausale (**CE**) et l'énumération des modèles (**ME**). Il ne permet cependant pas de compter le nombre de modèles d'une formule en temps polynomial sauf si P = NP. L'ajout d'une restriction supplémentaire permet de rendre ce langage encore plus attractif. Le langage d-DNNF est le sous-ensemble du langage DNNF qui satisfait la propriété de déterminisme.

Définition 56 (Déterminisme)

Un nœud de disjonction $N_\vee$ d'une formule NNF est déterministe si et seulement si ses fils $N_1, \ldots, N_k$ sont deux à deux contradictoires. Formellement, $N_i \wedge N_j \models \bot$ pour tout $i \neq j$.

Définition 57 (d-DNNF)

Une formule d-DNNF Σ est une formule DNNF dans laquelle chaque nœud de disjonction $\vee$ est déterministe. Chaque nœud interne d'une formule d-DNNF est donc soit un nœud de disjonction déterministe, soit un nœud de conjonction décomposable.

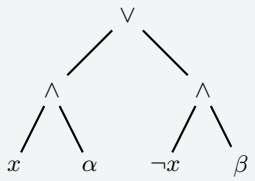
Le langage `d-DNNF` est beaucoup plus intéressant que `DNNF` du point de vue des requêtes offertes. En effet, ce langage satisfait toutes les propriétés correspondant aux requêtes proposées, sauf deux, le test d'implication générale (**SE**) et le test d'équivalence (**EQ**). En particulier, la propriété **CT** est offerte par `d-DNNF`. En effet, la propriété de la décomposabilité stipule que les fils de chaque nœud $\wedge$ n'ont aucune variable en commun. Par conséquent, le nombre de modèles d'un nœud $\wedge$ décomposable correspond au produit du nombre de modèles de ses fils. De plus, la propriété de déterminisme du langage `d-DNNF` stipule que les fils de chaque nœud de disjonction d'une formule `d-DNNF` ne possèdent aucun modèle en commun. Ainsi, le nombre de modèles d'un nœud $\vee$ déterministe représente la somme du nombre de modèles de ses fils (à un facteur de normalisation près par fils, lié au fait que les fils peuvent être construits sur des ensembles de variables qui diffèrent).

Le calcul du nombre de modèles d'une formule `d-DNNF` peut s'effectuer en un seul parcours récursif du graphe. Le principe est de partir de la racine, puis de parcourir l'ensemble de ses fils jusqu'aux feuilles, puis de remonter les valeurs d'un attribut synthétisé (le nombre de modèles de la formule enracinée à ce nœud) jusqu'à la racine. Si la feuille considérée est étiquetée par $\top$, la valeur retournée est 1. Si elle est étiquetée par $\bot$, la valeur retournée est 0. Sinon, la feuille représente un littéral et la valeur retournée est 1. Enfin, les nœuds $\wedge$ décomposables retournent le produit des valeurs de leur fils et les nœuds $\vee$ déterministes retournent la somme des valeurs de leurs fils (après produit par un facteur de cadrage défini à partir du nombre de variables figurant dans les frères du fils traité mais pas dans le fils lui-même).

Un autre sous-ensemble intéressant de `d-DNNF`, le langage `Decision-DNNF`, est défini en imposant une restriction supplémentaire sur les nœuds $\vee$ des formules. Ce langage est basé sur la notion de nœud de décision, où un nœud de décision est un nœud $\vee$ déterministe particulier.

Définition 58 (Nœud de décision)

Un nœud de disjonction $N_\vee$ d'une formule `NNF` est un nœud de décision s'il a la forme suivante :



La feuille x représente une variable, $\neg x$ sa négation, et α et β sont les racines de formules `NNF`. On dit que le nœud de décision $N_\vee$ porte sur la variable x et que x est une variable de décision.

Pour simplifier, nous représentons souvent graphiquement les nœuds de décision comme sur la partie gauche de la figure 5.2. L'arête en pointillés (respectivement, l'arête pleine) indique que la variable de décision x doit être affectée à faux (respectivement, vrai) dans tout chemin auquel elle participe. Chaque nœud de décision peut être vu comme l'occurrence d'un connecteur ternaire *ite* (pour *if ... then ... else ...*) dont la sémantique est $ite(x, \alpha, \beta) \equiv (x \wedge \alpha) \vee (\neg x \wedge \beta)$. Pour que $ite(x, \alpha, \beta)$ soit vrai, il faut et il suffit que x soit vrai et alors α doit l'être aussi, et sinon (donc quand $\neg x$ est vrai), β doit être vrai.

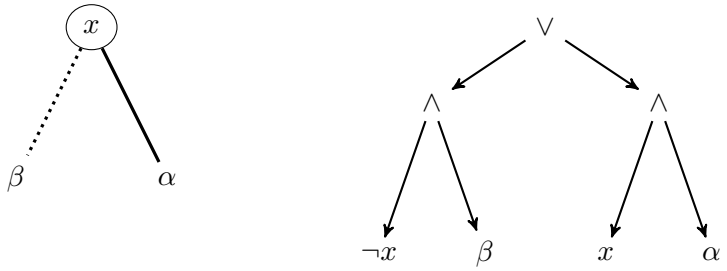


FIGURE 5.2 – Représentation des nœuds de décision

Définition 59 (`Decision-DNNF`)

Une formule `Decision-DNNF` Σ est une formule `d-DNNF` dans laquelle chaque nœud de disjonction est un nœud de décision. Chaque nœud interne d'une formule `Decision-DNNF` est donc soit un nœud $\wedge$ décomposable, soit un nœud $\vee$ de décision.

Clairement, pour chaque nœud de décision d'une formule `Decision-DNNF` Σ où la décision porte sur x, les formules α et β qui lui sont associées ne peuvent pas contenir x. En effet, dans le cas contraire, les nœuds de conjonction de la forme $x \wedge \alpha$ et $\neg x \wedge \beta$ ne seraient pas décomposables, donc Σ ne serait pas une formule `DNNF`.

Le langage `Decision-DNNF` satisfait les mêmes propriétés que le langage `d-DNNF` (quand nous nous limitons à celles listées plus haut). Les compilateurs existants, tels que `C2D` (Darwiche 2004) ou `Dsharp` (Muise et al. 2012), ciblant le langage `d-DNNF` ciblent en fait plus spécifiquement le langage `Decision-DNNF`. Par ailleurs, le supplément de structure apportée par les nœuds de décision fait qu'il est possible d'évaluer plus finement la taille des formules `Decision-DNNF` et le temps mis pour les calculer, en comparaison aux formules `d-DNNF` « générales » (Oztok et Darwiche 2014).

L'algorithme 5.2 donne le pseudo-code d'un compilateur `Decision-DNNF` d'une formule `CNF` Σ passée en paramètre. Il ressemble fortement à l'algorithme de comptage présenté dans la sous-section précédente, il s'appuie donc aussi sur la trace d'un solveur `DPLL`. À la ligne 1, `solve` est appelé sur la formule Σ (voir l'algorithme 5.1 pour plus de détails sur cette procédure). S est égal à $\{\varnothing\}$ si Σ est incohérent, et dans ce cas la `Decision-DNNF` représentant l'incohérence est générée ($\mathsf{leaf}(\bot)$) à la ligne 2. Le second cas de base est considéré à la ligne 3 : si Σ une fois simplifiée par `bcp` ne contient plus de variables, alors un nœud $\wedge$ est généré via l'appel à la routine $\mathsf{aNode}(\{\ell_1, \ldots, \ell_k\}, [N_1, \ldots, N_m])$ qui génère un nœud $\wedge$ avec $k + m$ enfants : les k littéraux de l'ensemble $\{\ell_1, \ldots, \ell_k\}$ et les m nœuds de la liste $[N_1, \ldots, N_m]$. Ainsi, $\mathsf{aNode}(\mathsf{S}, [\mathsf{leaf}(\top)])$ retourne un nœud $\wedge$ avec les littéraux de S (complété par une feuille $\top$ pour gérer le cas où S est vide). À la ligne 4, si la formule courante Σ avait déjà été rencontrée par le passé (elle est stockée dans le cache), alors l'algorithme retourne simplement le nœud racine correspondant à la représentation en `Decision-DNNF` de la formule courante. À la ligne 5 la fonction $\mathsf{connectedComponents}$ retourne un ensemble *comps* de formules `CNF`, représentant les composantes connexes de Σ. La boucle à la ligne 7 considère alors chaque élément Σ_c de *comps* successivement. Une variable v de $Var(\Sigma_c)$ est d'abord sélectionnée (ligne 8), puis un noœud de décision N_d avec la variable de décision v est créée, *via* deux appels récursifs à `Decision-DNNF` correspondant aux deux manières de

conditionner v dans c (ligne 9). Finalement, N_d est ajouté à la liste de nœuds en construction LN_d à la ligne 10, laquelle a été initialisée à vide avant de commencer la boucle (ligne 6). Une fois que toutes les composantes ont été traitées, un nœud $\wedge$ $N_\wedge$ regroupant les littéraux unitaires de S et les nœuds de décision de LN_d est créé (ligne 11), et ajouté à la structure de *cache* (en lui assignant comme clé Σ) (ligne 12), et finalement ce nœud $\wedge$ est retourné comme étant le résultat de la fonction principale (ligne 13).

Algorithme 5.2 : `Decision-DNNF` (Σ)

input : Σ une formule `CNF`
output : le nœud racine N de la `Decision-DNNF` représentant Σ

1 $\mathsf{S} \leftarrow \mathsf{solve}(\Sigma)$;
2 if $\mathsf{S} = \{\varnothing\}$ then return $\mathsf{leaf}(\bot)$;
3 if $Var(\Sigma) = \varnothing$ then return $\mathsf{aNode}(\mathsf{S}, [\mathsf{leaf}(\top)])$;
4 if $\mathsf{cache}(\Sigma) \neq \mathsf{nil}$ then return $\mathsf{aNode}(\mathsf{S}, [\mathsf{cache}(\Sigma)])$;
5 $comps \leftarrow \mathsf{connectedComponents}(\Sigma)$;
6 $LN_d \leftarrow []$;
7 foreach $\Sigma_c \in comps$ do
8 choisir une variable v de Σ_c ;
9 $N_d \leftarrow ite(v, \texttt{Decision-DNNF}(\Sigma_c|\neg v), \texttt{Decision-DNNF}(\Sigma_c|v))$;
10 $LN_d \leftarrow \mathsf{add}(N_d, LN_d)$;
11 $N_\wedge \leftarrow \mathsf{aNode}(S, LN_d)$;
12 $\mathsf{cache}(\Sigma) \leftarrow N_\wedge$;
13 return $N_\wedge$;

En imposant différentes restrictions sur le langage des `Decision-DNNF` sur l'utilisation ou pas du partage de structures (le non-partage conduit à construire des arbres) et de l'utilisation de nœuds $\wedge$ décomposables, il est possible de définir d'autres langages cibles permettant le comptage, comme les langages : `DT`, `EDT`, ou `FBDD` (Darwiche et Marquis 2002, Koriche et al. 2013). En imposant en plus un ordre strict des nœuds des chemins de la racine aux feuilles, nous obtenons les langages $\texttt{ODT}_<$ et $\texttt{OBDD}_<$. Plus précisément nous avons :

Langage	Partage de structures	$\wedge$ décomposable	Contrainte d'ordre
`DT`	✗	✗	✗
$\texttt{ODT}_<$	✗	✗	✓
`EDT`	✗	✓	✗
`FBDD`	✓	✗	✗
$\texttt{OBDD}_<$	✓	✗	✓

Nous avons considéré jusqu'ici des langages cibles construits à partir des connecteurs $\wedge$, $\vee$, $\neg$ et des constantes booléennes $\top$ et $\bot$. Nous présentons maintenant un langage qui s'appuie sur une autre ensemble de connecteurs logiques, que nous retrouverons à la section 5.5. Ce langage, appelé `AFF`, inclut des formules dites affines, construites à partir des connecteurs $\wedge$, $\oplus$, $\neg$, $\top$ et $\bot$. On a donc en quelque sorte laissé de côté la disjonction usuelle $\vee$ (inclusive), pour considérer à sa place la disjonction exclusive $\oplus$. Une formule affine est simplement une conjonction finie de clauses affines. Comme on va le montrer, les formules affines peuvent aussi être considérées comme des systèmes d'équations linéaires sur $\{0, 1\}$, et donc les clauses affines comme des équations linéaires.

Définition 60 (Clause affine (ou XOR-clause) et formule affine AFF)

Une clause affine, ou XOR-clause, est une disjonction exclusive (finie) de littéraux (incluant possiblement $\top$ et $\bot$). Une formule affine est une conjonction (finie) de clauses affines (de XOR-clauses).

Dans le but de simplifier l'écriture et de ne pas surcharger les formules, il est possible de normaliser l'écriture de chaque clause affine (on le fera assez systématiquement dans la suite) :

Définition 61 (Forme normale des clauses affines)

Une clause affine simplifiée, ou clause affine sous forme normale, est une clause affine constituée de littéraux positifs portant sur des variables distinctes et d'au plus une occurrence de $\top$.

Toute clause affine peut être transformée en une clause affine simplifiée en temps linéaire, en utilisant les équivalences $\neg x \equiv x \oplus \top$, $x \oplus x \equiv \bot$, $\top \oplus \top \equiv \bot$, $x \oplus \bot \equiv x$ et en profitant du fait que $\oplus$ est un connecteur associatif et commutatif.

Chaque clause affine simplifiée de la forme $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ (resp. $x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus \top$) peut aussi être vue comme une équation linéaire de la forme $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ (respectivement, $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 0$). Il suffit en effet de voir $=$ comme le symbole d'équivalence $\Leftrightarrow$, 1 comme la constante $\top$ et 0 comme la constante $\bot$ et de se souvenir que $\alpha \Leftrightarrow \beta$ équivaut à $\alpha \oplus \beta \oplus \top$: les affectations de valeur (de vérité) aux variables qui satisfont l'équation $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ (resp. $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 0$) sont exactement celles qui satisfont la clause associée.

Notons qu'une clause affine peut être représentée de façon canonique sous forme CNF par la conjonction de toutes les clauses formées sur l'ensemble de ses variables, contenant un nombre pair (resp. un nombre impair) de littéraux négatifs si la clause affine contient un nombre pair (resp. un nombre impair) de littéraux négatifs. Ainsi une clause affine initiale est beaucoup plus concise que n'importe quelle forme CNF équivalente. En effet, toute représentation d'une clause affine sous forme CNF est exponentielle en la taille de la clause affine. Plus précisément, une clause affine simplifiée de taille n ne possède que des représentations CNF qui contiennent au moins 2^{n-1} clauses (ce qui montre au passage que CNF $\not\leq_s$ AFF).

Un atout important du langage AFF est qu'il satisfait **CT** (Zanuttini 2003). En fait, l'ensemble des requêtes considérées dans la carte de compilation peut être résolu efficacement sur les formules affines. Tout est basé sur un même algorithme : l'algorithme d'élimination de Gauss/Jordan, aussi appelé pivot de Gauss. Le pivot de Gauss est un algorithme permettant de déterminer l'ensemble des solutions d'un système d'équations linéaires. Il consiste ici à réduire le nombre de clauses affines de la formule affine considérée Σ pour faire en sorte d'obtenir une formule affine logiquement équivalente, possédant un nombre minimum de clauses affines. L'algorithme du pivot de Gauss permet ainsi de garantir que la formule affine résultante Σ_t contient au maximum $\mid Var(\Sigma) \mid$ clauses affines. Le principe de l'algorithme est de trianguler la formule affine Σ selon un ordre total et strict $<$ sur les variables. Par exemple, la formule AFF $(x \oplus y \oplus u \oplus w) \wedge (y \oplus$

$z \oplus w \oplus \top) \wedge (z \oplus u)$ est triangulée puisque la première clause affine contient x, la clause affine suivante contient y mais pas x, et la dernière clause affine contient z mais pas les variables x et y. Nous pouvons ainsi représenter la formule sous forme d'une matrice triangulaire de la forme :

$$\begin{pmatrix} x & y & & u & w \\ & y & z & & w \\ & & z & u & \end{pmatrix}$$

Intuitivement, l'idée de l'algorithme est de faire en sorte d'ordonner les clauses de sorte qu'une variable (la plus petite selon $<$) de la première clause affine n'apparaisse pas dans les clauses affines suivantes. Puis, que la seconde clause contienne une variable (la seconde plus petite selon $<$) qui n'apparaisse pas dans les prochaines clauses affines, et ainsi de suite pour toutes les variables de la formule affine. À la fin de l'exécution nous obtenons donc une formule affine qui possède au plus n clauses affines, avec n le nombre de variables de la formule en entrée.

L'algorithme du pivot de Gauss est en temps polynomial en la taille de la formule en entrée (Zanuttini 2003). Cet algorithme calcule une formule affine triangulée Σ_t logiquement équivalente à Σ en temps $\mathcal{O}(\mid \Sigma \mid^2 . \mid Var(\Sigma) \mid)$. De plus, la formule affine Σ_t possède au plus $\mid Var(\Sigma) \mid$ clauses affines. Nous notons que le plus petit système d'équations linéaires ne possédant aucune solution est $0 = 1$, qui correspond à la formule $\bot$. Si la formule affine Σ en entrée ne possède aucun modèle, l'algorithme du pivot de Gauss retourne donc en sortie la formule $\bot$. L'algorithme du pivot de Gauss permet donc de décider la cohérence d'une formule affine Σ en temps $\mathcal{O}(\mid \Sigma \mid^2 \mid Var(\Sigma) \mid)$. Si la formule Σ est cohérente, nous pouvons facilement calculer un modèle de celle-ci à partir de la formule triangulée Σ_t. En effet, comme nous l'avons remarqué précédemment, dans chaque clause de la formule affine triangulée obtenue, il y a une variable liée, dont la valeur de vérité ne dépend que des autres variables de la formule. L'idée est donc de procéder de façon gloutonne à partir de la dernière clause affine de la formule. Nous affectons les variables libres de cette clause, puis nous remontons à la clause suivante, et ainsi de suite. Lorsque toutes les variables libres sont assignées, les variables liées sont assignées par propagation.

Plus généralement, pour connaître le nombre de modèles d'une formule affine triangulée, il suffit de connaître le nombre de ses variables libres. Celui-ci est facilement calculable. En effet, il y a exactement une variable liée pour chaque clause de la formule affine triangulée et celle-ci contient un nombre minimal de clauses. Par conséquent, le nombre de variables libres de Σ_t vaut le nombre de variables de Σ_t moins le nombre de clauses affines de celle-ci.

Définition 62 (Nombre de modèles d'une formule affine)

Soit Σ une formule affine et Σ_t une formule affine triangulée logiquement équivalente à Σ. Si Σ_t contient k variables libres, alors Σ possède 2^k modèles (sur $Var(\Sigma)$).

Nous disposons donc d'un algorithme en temps polynomial de calculer le nombre de modèles d'une formule affine donnée. D'autres requêtes liées au comptage de modèles (le test de cohérence, celui de validité) sont évidemment offertes en prime, à titre de conséquence. Ceci rend le langage `AFF` très compétitif vu nos objectifs.

Cependant, le langage `AFF` des formules affines n'est pas complet pour la logique propositionnelle. En effet, certaines formules propositionnelles ne peuvent pas être représentées par des

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
NNF	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
DNNF	√	○	√	○	○	○	○	√
d-DNNF	√	√	√	√	?	○	√	√
FBDD	√	√	√	√	?	○	√	√
$\text{OBDD}_<$	√	√	√	√	√	√	√	√
AFF	√	√	√	√	√	√	√	√
AFF[∨]	√	○	√	○	○	○	○	√

TABLE 5.1 – Requêtes. √ signifie "satisfait" et ○ signifie "ne satisfait pas sauf si P = NP."

conjonctions de formules affines. Par exemple, la formule $x_1 \vee x_2$ ne peut pas être représentée par une conjonction de clauses affines. Ce langage n'est donc pas utilisable dans de nombreux cas, à cause de son expressivité réduite.

Pour pallier ce problème, il faut généraliser le langage AFF. Nous pouvons alors par exemple le généraliser pour autoriser la présence de certaines disjonctions (inclusives) dans les formules. Ainsi, un langage constitué de disjonctions de formules affines AFF[∨] a-t-il été proposé dans (Fargier et Marquis 2008b). Il a été montré qu'un tel langage est complet pour la logique propositionnelle. Cependant, l'ajout de la disjonction à ce langage implique la perte de quelques transformations telles $\wedge\mathbf{C}$ et $\neg\mathbf{C}$ ainsi que la perte de beaucoup de requêtes, telles **VA**, **IM**, **EQ**, **SE** et le comptage de modèles **CT** qui est la principale requête qui nous intéresse.

Nous présenterons dans la section 5.5 d'autres généralisations de AFF permettant de pallier son incomplétude, tout en préservant **CT**.

Attardons-nous maintenant sur les langages que nous venons de décrire, et regardons quelles sont les requêtes et transformations satisfaites par ces derniers. Pour commencer, le tableau 5.1 résume les requêtes satisfaites par les langages que nous avons présentés. Nous remarquons que le langage NNF ne satisfait aucune requête considérée sauf si P=NP. Le langage CNF ne vérifie que le test de validité et le test d'impliquant. Ces deux langages ne sont donc pas intéressants en tant que langages cibles pour la compilation. Le langage AFF satisfait toutes les requêtes, cependant il n'est pas complet pour la logique propositionnelle et ne peut donc pas être considéré comme un langage cible pour la compilation exacte de certaines formules. Les langages d-DNNF et $\text{OBDD}_<$ sont deux langages satisfaisant le comptage de modèles (**CT**) et donc les plus intéressants (parmi ceux présentés plus haut) quand on se focalise sur cette requête.

Le tableau 5.2 résume les transformations que vérifient les langages que nous avons présentés. Nous remarquons que tous ces langages satisfont le test de conditionnement (**CD**). Le langage NNF vérifie toutes les transformations sauf l'oubli (**FO**). Le langage d-DNNF ne vérifie que le conditionnement, tandis que le langage $\text{OBDD}_<$ vérifie toutes les transformations sauf l'oubli, la conjonction et la disjonction.

$\mathcal{L}$	CD	FO	SFO	$\wedge$C	$\wedge$BC	$\vee$C	$\vee$BC	$\neg$C
NNF	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
CNF	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\bullet$	$\surd$	$\bullet$
DNF	$\surd$	$\surd$	$\surd$	$\bullet$	$\surd$	$\surd$	$\surd$	$\bullet$
DNNF	$\surd$	$\surd$	$\surd$	$\circ$	$\circ$	$\surd$	$\surd$	$\circ$
d-DNNF	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?
FBDD	$\surd$	$\bullet$	$\circ$	$\bullet$	$\circ$	$\bullet$	$\circ$	$\surd$
$\text{OBDD}_{<}$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
AFF	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	!	!	!
AFF[$\vee$]	$\surd$	$\circ$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$

TABLE 5.2 – Transformations. $\surd$ signifie "satisfait", $\bullet$ signifie "ne satisfait pas" et $\circ$ signifie "ne satisfait pas sauf si P = NP".

5.2 Pré-traitement pour la compilation de connaissances et le comptage de modèles

Pré-traiter une formule de la logique propositionnelle consiste à transformer cette dernière en une autre formule propositionnelle, de sorte que la formule produite conserve certaines propriétés intéressantes (la satisfaisabilité par exemple). De manière générale, un pré-traitement est considéré comme utile lorsqu'il rend le problème pré-traité plus « facile » à résoudre que le problème initial. Dans le cadre de la résolution de SAT et QBF, certains pré-traitements sont devenus incontournables (Bacchus et Winter 2003, Subbarayan et Pradhan 2005, Lynce et Marques-Silva 2003, Eén et Biere 2005, Piette et al. 2008, Han et Somenzi 2007, Heule et al. 2010, Järvisalo et al. 2012a, Heule et al. 2011a). Le succès de ces techniques est tel que de nombreux solveurs SAT de l'état de l'art les considèrent nativement : Glucose utilise SatElite (Eén et Biere 2005), Lingeling (Biere 2017) utilise une méthode de pré-traitement interne et Riss (Manthey 2012) utilise Coprocessor.

Étant donné le succès des méthodes de pré-traitement pour les problèmes de satisfaisabilité, nous nous sommes posés la question, dans des travaux en collaboration avec Pierre Marquis et Emmanuel Lonca dans (Lagniez et Marquis 2014, Lagniez et al. 2016b, Lagniez et Marquis 2017b), de l'efficacité de telles méthodes pour le comptage de modèles (exact et approché) et la compilation de connaissances. Le fait de considérer le comptage de modèles plutôt que la satisfaisabilité a un impact important sur le type de méthodes de pré-traitement qu'il est possible d'utiliser. D'une part, les méthodes de pré-traitement qui préservent la satisfaisabilité ne préservent pas forcément le nombre de modèles. Ainsi, certaines méthodes de pré-traitement utilisées dans le cadre de SAT ne peuvent pas être utilisées pour le comptage de modèles : par exemple l'élimination des clauses contenant des littéraux purs, les méthodes d'élimination de variables basées sur la règle de résolution utilisée dans SatElite, l'élimination des clauses bloquées (Kullmann 1999, Heule et al. 2015). En effet, chacune de ces méthodes conserve la satisfaisabilité, mais aucune ne garantit de conserver le nombre de modèles. D'autre part, le fait que le problème de compter le nombre de modèles d'une formule propositionnelle soit beaucoup plus difficile que le problème de décider la satisfaisabilité de cette dernière, permet d'envisager des méthodes de pré-traitement beaucoup plus gourmandes en ressource de calcul. Par exemple, il peut être utile de calculer l'ensemble des littéraux impliqués par une formule avant de compter son nombre

de modèles, tandis que cela n'a pas de sens si nous souhaitons uniquement déterminer si cette formule est satisfaisable. Finalement, puisque beaucoup de méthodes de pré-traitement existent, il peut aussi être intéressant de les combiner. Dans ce cas, il paraît important d'étudier si l'ordre dans lequel elles sont combinées a de l'importance.

Un autre aspect important, pour le choix des méthodes de pré-traitement utilisables, est la nature du compteur de modèles employé. Si c'est un compteur de modèles classique, conserver le nombre de modèles est suffisant. Cependant, si le comptage de modèles est réalisé à partir d'une formule compilée, il est nécessaire d'utiliser des méthodes qui permettent de conserver l'équivalence de la formule si nous souhaitons réaliser des requêtes *a posteriori*.

La suite de cette section introduit d'abord les méthodes de pré-traitements permettant de préserver l'équivalence et ensuite celles permettant uniquement de conserver le nombre de modèles. Nous concluons cette section par une discussion sur les possibles évolutions de ces travaux.

5.2.1 Pré-traitement préservant l'équivalence

Nous avons étudié et validé expérimentalement les méthodes de pré-traitement élémentaires suivantes : détection du *backbone*, réduction d'occurrences et vivification. Le *backbone* (Monasson et al. 1999b) d'une formule `CNF` est l'ensemble des littéraux qui sont impliqués par la formule. Le but de cette méthode de pré-traitement est de calculer le *backbone* de la formule et d'utiliser la propagation unitaire pour simplifier la formule en considérant les littéraux de ce dernier. L'algorithme 5.3 décrit une procédure d'extraction du *backbone*. La recherche des littéraux du *backbone* est limitée aux littéraux ℓ satisfaits par un modèle $\mathcal{I}$ de Σ. Si $\Sigma \wedge \sim\ell$ est contradiction, alors tous les modèles de Σ satisfont ℓ, et donc ℓ fait partie du *backbone*. Sinon, $\Sigma \wedge \sim\ell$ a un modèle $\mathcal{I}'$ et seulement les littéraux qui satisfont $\mathcal{I}$ et $\mathcal{I}'$ peuvent faire partie du *backbone*.

Il est clair que cette méthode de pré-traitement permet de conserver l'équivalence. Cependant, puisqu'il est nécessaire de faire des appels à un solveur `SAT`, cette fonction peut être gourmande en temps. Afin d'accélérer cette procédure, nous avons utilisé le solveur `SAT` incrémental présenté au chapitre précédent (voir 4.2.2).

Algorithme 5.3 : `backboneSimpl`

input : Σ une formule `CNF`
output : la `CNF` `bcp`$(\Sigma \cup \mathcal{B})$, où $\mathcal{B}$ est le *backbone* de Σ

```
1 B ⟵ ∅;
2 I ⟵ solve(Σ);
3 while ∃ℓ ∈ I tel que ℓ ∉ B do
4 |   I' ⟵ solve(Σ ∪ {ℓ̃});
5 |   if I' = ∅ then B ⟵ B ∪ {ℓ} else I ⟵ I ∩ I';
6 return bcp(Σ ∪ B)
```

La méthode de réduction d'occurrences, présentée dans l'algorithme 5.4, est une procédure que nous avons développée afin de supprimer des littéraux d'une formule `CNF` Σ en remplaçant une clause $\alpha = \ell_1 \vee \ldots \vee \ell_j \vee \ell_{j+1}$ par une clause qui la subsume $\alpha \setminus \{\ell_{j+1}\}$. Afin de déterminer si un littéral ℓ_{j+1} peut être supprimé d'une clause α de Σ, l'approche que nous proposons consiste à vérifier si la clause α où nous avons remplacé ℓ_{j+1} par $\sim\ell_{j+1}$ est une conséquence logique de Σ. Pour réaliser cela, nous vérifions si $\Sigma \wedge \ell_{j+1} \wedge \sim\ell_1 \wedge \ldots \wedge \sim\ell_j$ est contradictoire. Quand cela est le cas, ℓ_{j+1} peut être supprimé de α tout en conservant l'équivalence. Afin de limiter le coût de cette

approche, au lieu d'appeler un oracle pour chaque test de satisfaction, nous utilisons la procédure `bcp`. De manière à maximiser le nombre d'occurrences de littéraux éliminés, nous considérons en priorité les littéraux qui apparaissent le plus fréquemment dans Σ. Il est important de noter que, contrairement à la méthode précédente, ce pré-traitement n'est pas confluent (c'est-à-dire que l'ordre dans lequel les éliminations sont réalisées change le résultat de la procédure). De plus, cette méthode n'est pas *idempotente* (c'est-à-dire $p(p(\Sigma))$ est potentiellement plus réduit que $p(\Sigma)$), ce qui implique qu'il peut être intéressant de l'appliquer plusieurs fois.

Algorithme 5.4 : `occurrenceSimpl`

input : Σ une formule `CNF`
output : une formula `CNF` équivalente à Σ

1 $\mathcal{L} \longleftarrow sort(Lit(\Sigma))$;
2 **foreach** $\ell \in \mathcal{L}$ **do**
3 **foreach** $\alpha \in \Sigma$ *tel que* $\ell \in \alpha$ **do**
4 **if** $\varnothing \in$ `bcp`$(\Sigma \cup \{\ell\} \cup \{\sim(\alpha \setminus \{\ell\})\})$ **then** $\Sigma \longleftarrow (\Sigma \setminus \{\alpha\}) \cup \{\alpha \setminus \{\ell\}\}$;
5 **return** Σ

Dans le pire des cas, la complexité de cette procédure est cubique dans la taille de la formule. La méthode d'élimination d'occurrences de littéraux peut être vue comme une version plus faible de la vivification présentée juste après (ici, nous cherchons uniquement à supprimer certains littéraux). Cependant, comparée à la vivification, cette procédure permet toujours de produire une formule qui sera plus puissante du point de vue du la propagation unitaire (cela s'explique par le fait que nous ne pouvons que raccourcir des clauses, tandis que la vivification peut en supprimer).

La vivification est une méthode proposée par Piette et al. (2008) et dont l'objectif est de réduire la formule `CNF` Σ donnée en entrée, c'est-à-dire supprimer certaines clauses et certains littéraux de Σ tout en conservant l'équivalence. Cette procédure, décrite dans l'algorithme 5.5, a comme la méthode précédente une complexité cubique dans la taille de la formule d'entrée.

Cette méthode considère incrémentalement chaque clause $\alpha = \ell_1 \vee \ldots \vee \ell_k$ de Σ de manière à vérifier si elle doit être réduite ou supprimée. Pour cela, les littéraux ℓ_{j+1} de α sont considérés successivement afin de déterminer s'ils doivent être ajoutés ou pas dans la sous-clause $\alpha' = \ell_1 \vee \ldots \vee \ell_j$ de α (α' est initialisée à vide à la ligne 3) en considérant les règles suivantes :

— s'il existe $j \in 0, \ldots, k-1$ tel que nous avons $\Sigma \setminus \{\alpha\} \models \alpha'$, alors nous savons que la clause α' est une conséquence logique de $\Sigma \setminus \{\alpha\}$ et nous pouvons donc la supprimer (dans ce cas $\alpha' \leftarrow \top$ ligne 8) ;

— s'il est possible de prouver par propagation unitaire que $\Sigma \setminus \{\alpha\} \models \alpha' \vee \sim\ell_{j+1}$, alors ℓ_{j+1} peut être supprimé de α sans remettre en cause l'équivalence.

Dans notre implémentation, nous avons considéré en priorité les clauses de plus grande taille. Dans le cas où le *backbone* a été calculé par l'algorithme 5.3, les littéraux de la clause sont considérés dans l'ordre inverse de leur activité `VSIDS`, sinon c'est l'ordre lexicographique qui est utilisé.

Comme pour la méthode d'élimination d'occurrences, la vivification n'est ni confluente ni idempotente, ce qui implique qu'il y a un intérêt à appliquer cette procédure plusieurs fois.

Algorithme 5.5 : `vivificationSimpl`

input : Σ une formule `CNF`
output : une formule `CNF` équivalente à Σ

foreach $\alpha \in \Sigma$ **do**
  $\Sigma \longleftarrow \Sigma \setminus \{\alpha\}$;
  $\alpha' \longleftarrow \bot$;
  $\mathcal{I} \longleftarrow \texttt{bcp}(\Sigma)$;
  while $\exists \ell \in \alpha$ *tel que* $\sim\ell \notin \mathcal{I}$ *et* $\alpha' \neq \top$ **do**
    $\alpha' \longleftarrow \alpha' \vee \ell$;
    $\mathcal{I} \longleftarrow \texttt{bcp}(\Sigma \cup \{\sim\alpha'\})$;
    if $\varnothing \in \mathcal{I}$ **then** $\alpha' \longleftarrow \top$;
  $\Sigma \longleftarrow \Sigma \cup \{\alpha'\}$;
return Σ

5.2.2 Pré-traitement préservant le nombre de modèles

Nous présentons ici une technique de pré-traitement pour le comptage de modèles propositionnels qui exploite les définitions de variables, c'est-à-dire la détection de portes logiques impliquées par la formule Σ fournie en entrée. De telles portes peuvent être utilisées pour simplifier la formule Σ lors d'une phase de pré-traitement sans modifier son nombre de modèles, mais en rendant le comptage de ceux-ci plus aisé. L'idée générale de cette approche est que toute variable y de Σ définie dans Σ en fonction d'un ensemble d'autres variables $X = \{x_1, \ldots, x_k\}$ peut être remplacée par sa définition Φ_X, noté $\Sigma[y \leftarrow \Phi_X]$, sans modifier le nombre de modèles de Σ. Plus précisément, pour toute affectation partielle des variables de X, soit cette dernière affectation contredit Σ, soit tout modèle étendant cette affectation donne la même valeur de vérité à y.

Commençons par quelques préliminaires. Étant donné un sous-ensemble de variables $X \subseteq PS$, un terme canonique γ_X sur X est un terme cohérent dans lequel toutes les variables de X apparaissent (sous la forme d'un littéral positif ou négatif, c'est-à-dire une variable niée). Nous notons $\exists X.\Sigma$ l'oubli des variables de X dans Σ, c'est-à-dire toute formule équivalente à la conséquence logique la plus forte de Σ qui est indépendante des variables de X (Lang et al. 2002). Nous rappelons maintenant les deux formes (équivalentes) sous lesquelles le concept de définissabilité propositionnelle peut être rencontré dans la littérature.

Définition 63 (Définition implicite)

Soit $\Sigma \in \mathcal{CPL}$, $X \subseteq PS$ et $y \in PS$. Σ *définit de manière implicite* y à partir de X si et seulement si pour tout terme canonique γ_X sur X, on a $\gamma_X \wedge \Sigma \models y$ ou $\gamma_X \wedge \Sigma \models \neg y$.

Définition 64 (Définition explicite)

Soit $\Sigma \in \mathcal{CPL}$, $X \subseteq PS$ et $y \in PS$. Σ *définit explicitement* y à partir de X si et seulement s'il existe une formule $\Phi_X \in \mathcal{CPL}$ telle que $\Sigma \models \Phi_X \leftrightarrow y$. Dans ce cas, Φ_X est appelée *définition (ou porte) de* y *sur* X *dans* Σ, y est la variable de sortie de la porte, et X est l'ensemble des variables d'entrée.

Exemple 22

Soit Σ la formule `CNF` constituée de la conjonction des clauses suivantes :

$a \vee b$,	$\neg a \vee \neg b \vee d$,	$a \vee e$,
$a \vee c \vee \neg e$,	$\neg a \vee \neg c \vee d$,	$b \vee c \vee e$,
$a \vee \neg d$,	$\neg a \vee \neg b \vee c \vee \neg e$,	$\neg b \vee \neg c \vee e$,
$b \vee c \vee \neg d$,	$\neg a \vee b \vee \neg c \vee \neg e$.	

Les variables d et e sont définies implicitement dans Σ à partir de $X = \{a, b, c\}$. Nous pouvons, par exemple, vérifier que pour le terme canonique $\gamma_X = a \wedge b \wedge \neg c$, on $\gamma_X \wedge \Sigma \models d \wedge \neg e$; pour $\gamma'_X = \neg a \wedge \neg b \wedge \neg c$, $\gamma'_X \wedge \Sigma$ est incohérent. On peut aussi vérifier que d et e sont définies explicitement dans Σ à partir de $X = \{a, b, c\}$; plus précisément Σ implique les équivalences suivantes :

$$d \leftrightarrow (a \wedge (b \vee c)) \text{ et } e \leftrightarrow (\neg a \vee (b \leftrightarrow c)).$$

Le fait que les mêmes variables soient définies à la fois implicitement et explicitement n'est pas fortuit, comme l'explique le théorème suivant, dû à Beth (1953), et restreint ici à la logique propositionnelle.

Théoreme 2

Soit $\Sigma \in \mathcal{CPL}$, $X \subseteq PS$ et $y \in PS$. Σ définit implicitement y à partir de X si et seulement si Σ définit explicitement y à partir de X.

Puisque ces deux notions de définissabilité coïncident, nous dirons simplement dans la suite que y est défini à partir de X dans Σ, sans préciser s'il s'agit de définissabilité implicite ou explicite.

5.2.2.1 `pmc` : une exploitation explicite de la définition

Une manière de dégager des définitions explicites d'une formule est de rechercher des portes logiques (Ostrowski et al. 2002) en particulier des portes très simples comme des équivalences entre littéraux (Bacchus et Winter 2003). Une fois ces informations collectées, il est possible de remplacer dans la formule les littéraux définis par leurs définitions. Bien que cette méthode de pré-traitement ne conserve pas l'équivalence logique, elle préserve le nombre de modèles de la

formule : étant données deux formules Σ et Φ et un littéral ℓ, si nous avons $\Sigma \models \ell \leftrightarrow \Phi$, alors $\Sigma[\ell \leftarrow \Phi]$ a le même nombre de modèles que Σ.

L'implémentation de ce pré-traitement nécessite dans un premier temps de détecter les définitions de type $\ell \leftrightarrow \Phi$ dans Σ, de réaliser le remplacement et de réécrire la formule sous forme `CNF` si besoin. Dans notre approche, le remplacement est uniquement réalisé si ce dernier ne fait pas croître la taille de la formule (à la NIVER (Subbarayan et Pradhan 2005)). Cela est garanti dans le cas où nous remplaçons des littéraux par d'autres, mais pas dans le cas où nous les remplaçons par des portes logiques plus complexes.

De manière générale, la recherche de portes logiques (et d'équivalences de littéraux) est réalisé de « manière syntaxique » ou en utilisant la propagation unitaire. La recherche de littéraux équivalents et leur remplacement dans la formule est une méthode de pré-traitement déjà à l'œuvre dans le solveur `SAT` de Eén et Biere (2005). L'algorithme 5.6 presente comment ce pré-traitement est réalisé.

Certaines techniques pour la détection et le remplacement de littéraux équivalents sont basées sur la recherche de *patterns*, c'est-à-dire dans la formule `CNF` Σ des clauses binaires permettant d'encoder des équivalences, ou plus généralement en recherchant des composantes fortement connexes dans le graphe d'implication de Σ (Gelder 2005). Dans notre approche nous considérons une technique légèrement différente qui utilise `bcp` afin de détecter des équivalences entre littéraux. Ensuite, pour chaque littéral ℓ, tous les littéraux ℓ' pouvant être trouvés équivalents par propagation unitaire sont remplacés par ℓ dans Σ. Dans le pire des cas, la complexité de `equivSimpl` est cubique dans la taille de la formule d'entrée.

Algorithme 5.6 : `equivSimpl`

input : Σ une formule `CNF`
output : Φ une formule `CNF` telle que $\|\Phi\| = \|\Sigma\|$

$\Phi \longleftarrow \Sigma$;
Démarquer toutes les variables de Φ;
while $\exists \ell \in Lit(\Phi)$ *tel que* $var(\ell)$ *n'est pas marqué* **do**
    `// détection`
    marquer $var(\ell)$;
    $\mathcal{P}_\ell \longleftarrow$ `bcp`$(\Phi \cup \{\ell\})$;
    $\mathcal{N}_\ell \longleftarrow$ `bcp`$(\Phi \cup \{\sim\ell\})$;
    $\Gamma \longleftarrow \{\ell \leftrightarrow \ell' | \ell' \neq \ell$ et $\ell' \in \mathcal{P}_\ell$ et $\sim\ell' \in \mathcal{N}_\ell\}$;
    `// remplacement`
    foreach $\ell \leftrightarrow \ell' \in \Gamma$ **do**
        remplacer ℓ' par ℓ dans Φ;
return Φ

Dans `equivSimpl`, les littéraux de la formule `CNF` Σ sont considérés dans l'ordre lexicographique des variables. Il est important de noter que, puisque le choix du représentant du littéral choisi pour représenter chaque littéral de sa classe d'équivalence dépend de la représentation syntaxique de Σ, la méthode `equivSimpl` n'est ni confluente ni idempotente.

Avant de présenter la méthode de pré-traitement qui recherche et remplace des portes logiques de type `AND`, rappelons que les lois de De Morgan permettent d'établir que toute porte `AND` (de la forme $\ell \leftrightarrow (\ell_1 \wedge \ldots \wedge \ell_k)$) peut être vue comme une porte `OR` (de la forme $\sim\ell \leftrightarrow (\sim\ell_1 \vee \ldots \vee \sim\ell_k)$).

Cela explique pourquoi nous n'avons traité que le cas de la détection de portes de type `AND`.

Notre algorithme de détection de portes `AND`/`OR` est présenté dans l'algorithme 5.7. Dans le pire des cas sa complexité est cubique dans la taille de la formule d'entrée. Contrairement aux travaux de Ostrowski (2004), où les portes sont recherchées de manière syntaxique, notre approche utilise la propagation unitaire.

Algorithme 5.7 : `ANDgateSimpl`

input : Σ une formule `CNF`
output : Φ une formule `CNF` telle que $\|\Phi\| = \|\Sigma\|$

1 $\Phi \longleftarrow \Sigma$;
// `détection`
2 $\Gamma \longleftarrow \varnothing$;
3 démarquer tous les littéraux de Φ;
4 **while** $\exists \ell \in Lit(\Phi)$ *tel que* ℓ *n'est pas marqué* **do**
5 marquer ℓ;
6 $\mathcal{P}_\ell \longleftarrow (\texttt{bcp}(\Phi \cup \{\ell\}) \setminus (\texttt{bcp}(\Phi) \cup \{\ell\})) \cup \{\sim\ell\}$;
7 **if** $\varnothing \in \texttt{bcp}(\Phi \cup \mathcal{P}_\ell)$ **then**
8 soit $\mathcal{C}_\ell \subseteq \mathcal{P}_\ell$ tel que $\varnothing \in \texttt{bcp}(\Phi \cup \mathcal{C}_\ell)$ et $\sim\ell \in \mathcal{C}_\ell$;
9 $\Gamma \longleftarrow \Gamma \cup \{\ell \leftrightarrow \bigwedge_{\ell' \in \mathcal{C}_\ell \setminus \{\sim\ell\}} \ell'\}$;
// `remplacement`
10 **while** $\exists \ell \leftrightarrow \beta \in \Gamma$ *tel que* $|\beta| < \texttt{maxA}$ *et* $|\Phi[\ell \longleftarrow \beta]| \leq |\Phi|$ **do**
11 $\Phi \longleftarrow \Phi[\ell \longleftarrow \beta]$;
12 $\Gamma \longleftarrow \Gamma[\ell \longleftarrow \beta]$;
13 $\Gamma \longleftarrow \Gamma \setminus \{\ell' \leftrightarrow \zeta \in \Gamma | \ell' \in \zeta\}$
14 **return** Φ

L'algorithme commence par une phase de détection des portes. À la ligne 4, les littéraux ℓ de Σ sont considérés dans un certain ordre de telle sorte que le littéral positif de la variable x est considéré avant le littéral négatif $\neg x$. Comme pour la méthode `equivSimpl`, nous considérons dans `ANDgateSimpl` les littéraux dans l'ordre lexicographique par rapport aux variables de Σ. À la ligne 6, $\mathcal{P}_\ell$ contient $\sim\ell$ et l'ensemble des littéraux pouvant être dérivés en plus par propagation unitaire en considérant ℓ. Ainsi, après cette étape $\mathcal{P}_\ell$ est tel que $\Sigma \models \ell \rightarrow (\bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell')$. Le test de la ligne 7 permet de décider si une porte `AND` β avec comme sortie ℓ peut être extraite de Σ. En effet, si $\Sigma \wedge \bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell' \wedge \sim\ell$ est prouvé insatisfaisable par propagation unitaire, nous avons $\Sigma \models (\wedge \bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell') \rightarrow \ell$. Et donc il y a dans Σ une porte `AND` $\beta = \bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell'$ avec comme sortie ℓ. Dans notre implémentation nous avons essayé de minimiser le nombre de variables dans les portes générées (lignes 8 et 9) en exploitant le graphe d'implication (Marques-Silva et Sakallah 1996b, Bayardo Jr. et Schrag 1997, Moskewicz et al. 2001a) en remontant les décisions responsables de l'incohérence obtenue par propagation unitaire (voir 4.2.3.3).

Lorsque nous avons calculé l'ensemble des portes, nous réalisons la phase de remplacement (lignes 10 à 13). Les définition $\ell \leftrightarrow \beta$ sont triées dans l'ordre croissant de leur taille. Le remplacement d'un littéral ℓ par sa définition β est réalisé si et seulement si le nombre de clauses générées par le remplacement reste « assez petit » (*ie.* $\leq$ `maxA` – dans nos expérimentations nous avons considéré `maxA` = 10). Le processus de remplacement s'arrête dès lors qu'il n'est plus possible de réaliser un remplacement valide étant donné les critères que nous venons de définir (ligne 10). Il est imporant de noter que le remplacement est réalisé à la fois sur la formule `CNF` et dans

l'ensemble Γ des définitions (lignes 11 and 12). Ainsi, les définitions qui deviennent valides sont supprimées de Γ (ligne 13). Notons qu'une diminution de la taille de la formule peut aussi survenir lors du processus de remplacement, cela arrive lorsque des clauses valides sont générées. Il est facile de voir, que comme pour `equivSimpl`, le pré-traitement `ANDgateSimpl` n'est ni confluent ni projectif.

La dernière procédure de pré-traitement que nous avons exploitée réalise une détection « syntaxique » de portes `XOR`. Cela est représenté dans l'algorithme 5.8 à la ligne 2, où des portes `XOR` $\ell_i \leftrightarrow \chi_i$ sont recherchées dans Σ en examinant l'ensemble des clauses de Σ représentant explicitement $\sim\ell_i \oplus \chi_i$ (dans nos expérimentations nous nous sommes limités à le recherche de portes de taille inférieure à `maxX` $= 5$). Une fois détectée, l'élimination de Gauss est réalisée afin de trianguler la formule. L'ensemble des portes `XOR` ainsi calculé peut alors être vu comme un ensemble de portes car $\ell_i \leftrightarrow \chi_i$ est équivalent à $\sim\ell_i \oplus \chi_i$. La dernière phase consiste à remplacer les littéraux ℓ_i par leurs définitions χ_i dans Σ, tout en faisant attention à ne pas trop faire augmenter la taille de la formule en ne considérant que les `XOR` qui ont une taille intérieure à `maxX`. Étant donné cette condition et le fait que la détection est « syntaxique », la complexité de l'algorithme est quadratique dans la taille de la formule d'entrée.

Algorithme 5.8 : `XORgateSimpl`

input : Σ une formule `CNF`
output : Φ une formule `CNF` tel que $\|\Phi\| = \|\Sigma\|$

1 $\Phi \longleftarrow \Sigma$;
`// détection`
2 $\Gamma \longleftarrow$ {clauses XOR détectées syntaxiquement};
`// élimination de Gauss`
3 $\Gamma \longleftarrow \text{Gauss}(\{\ell_1 \leftrightarrow \chi_1, \ell_2 \leftrightarrow \chi_2, \ldots, \ell_k \leftrightarrow \chi_k\})$
`// replacement`
4 **for** $i \longleftarrow 1$ *to* k **do**
5 ⌊ **if** $\nexists \alpha \in \Phi[\ell_i \longleftarrow \chi_i] \setminus \Phi$ *tel que* $|\alpha| >$ `maxX` **then** $\Phi \longleftarrow$ `CNF`$(\Phi[\ell_i \longleftarrow \chi_i])$;
6 **return** Φ

Il est facile de voir que, comme les méthodes `equivSimpl` et `ANDgateSimpl`, `XORgateSimpl` n'est ni confluent ni projectif.

C'est l'utilisation de ces méthodes de pré-traitement en combinaison avec celles préservant l'équivalence qui a donné naissance à `pmc`. Cet algorithme de pré-traitement, décrit dans 5.9 se base sur un système d'options pour activer ou pas les différentes techniques de pré-traitement élémentaires : `optV` (vivification), `optB` (identification du *backbone*), `optO` (réduction des occurrences), `optG` (détection et remplacement de portes). `gatesSimpl(Φ)` est un raccourci pour `XORgateSimpl(ANDgateSimpl(equivSimpl(Φ)))`.

`pmc` est un processus itératif. En effet, il peut être utile d'appliquer plusieurs fois les différentes méthodes de pré-traitement élémentaires puisque la formule `CNF` change et que les méthodes ne sont pas généralement idempotentes. Néanmoins, comme nous l'avons déjà souligné, au départ il n'est pas utile d'appeler plusieurs fois la méthode d'identification du *backbone*.

Une fois que les littéraux du *backbone* ont été identifiés et propagés via la méthode `bcp`, nous utilisons la méthode `occurrenceSimpl` afin de réduire la taille des clauses, et ainsi augmenter le pouvoir de la propagation unitaire afin d'augmenter le nombre de portes pouvant être détectées.

Nous aurions aussi pu utiliser `vivificationSimpl`, mais puisque cette méthode peut conduire à la suppression de clauses, cette méthode peut affaiblir `bcp`. Nous utilisons `vivificationSimpl` à la fin de chaque itération afin de « nettoyer » la formule après l'application du remplacement de portes.

Dans `pmc`, les méthodes de pré-traitement élémentaires (excepté `backboneSimpl`) peuvent être appliquées plusieurs fois. En fait, nous sortons de la boucle lorsqu'il n'y a pas eu de modification sur la formule après l'application de toutes les règles ou lorsque nous atteignons un certain nombre d'itérations `numTries` fixé au départ (dans nos expérimentations `numTries` = 10).

Algorithme 5.9 : `pmc`

```
   input  : Σ une formule CNF
   output : Φ une formule CNF telle que ‖Φ‖ = ‖Σ‖
 1 Φ ⟵ Σ;
 2 if optB then Φ ⟵ backboneSimpl(Φ);
 3 i ⟵ 0;
 4 while i < numTries do
 5 |   i ⟵ i + 1;
 6 |   if optO then Φ ⟵ occurrenceSimpl(Φ);
 7 |   if optG then Φ ⟵ gatesSimpl(Φ);
 8 |   if optV then Φ ⟵ vivificationSimpl(Φ);
 9 |   if fixpoint then break;
10 return Φ
```

Dans nos travaux, nous avons considéré deux types de méthode :

— *eq* qui correspond au pré-traitement qui conserve l'équivalence et où nous avons dans `pmc` `optV` = `optB` = `optO` = 1 et `optG` = 0 ;

— *#eq* qui correspond au pré-traitement qui assure de générer une formule avec le même nombre de modèle (mais qui n'est pas forcément équivalente à la formule d'entrée) et où nous avons dans `pmc` `optV` = `optB` = `optO` = `optG` = 1.

5.2.2.2 `B+E` : une exploitation implicite de la définition

Contrairement à l'approche précédente, nous exploitons dans la suite simplement l'existence de portes, sans besoin de les expliciter. Pour cela nous nous appuyons sur une conséquence intéressante du théorème de Beth, qui est qu'il n'est pas nécessaire d'expliciter une définition Φ_X de y à partir de X pour montrer qu'elle existe ; il suffit de démontrer que Σ définit implicitement y à partir de X, ce qui est un problème « seulement » coNP-complet (Lang et Marquis 2008). Ceci résulte du théorème suivant, dû à Padoa Padoa (1903), restreint à la logique propositionnelle et rappelé dans (Lang et Marquis 2008).

Théoreme 3

Étant donné $\Sigma \in \mathcal{CPL}$ et $X \subseteq PS$, soit Σ'_X la formule obtenue en remplaçant de manière uniforme dans Σ toute occurrence de symbole propositionnel x de $Var(\Sigma) \setminus X$ par un nouveau symbole propositionnel x'. Si $y \notin X$ alors Σ définit (de manière implicite) y à partir de X si et seulement si $\Sigma \wedge \Sigma'_X \wedge y \wedge \neg y'$ est incohérent [a].

a. Trivialement, dans le cas contraire où $y \in X$, Σ définit y à partir de X.

Ainsi notre pré-traitement se décompose en deux phases : dans un premier temps, nous calculons une bipartition $\langle I, O\rangle$ des variables de Σ telles que les variables de O soient définies dans Σ à partir des variables de I ; dans un second temps, nous éliminons des variables de O dans Σ. Cette technique de pré-traitement tend à s'attaquer au problème de manière beaucoup plus agressive, avec l'idée *qu'il n'est pas nécessaire d'identifier les portes logiques mais qu'il peut être suffisant de déterminer que ces portes existent.* Plus précisément, il se révèle suffisant de déterminer qu'il existe des relations de définition entre variables ; il n'est pas nécessaire de déterminer comment sont définies ces relations. Cette distinction est de première importance dans la mesure où, bien que l'espace de recherche des définitions potentielles Φ_X soit très important (2^{2^k} éléments à l'équivalence logique près, quand X contient k variables), la taille d'une définition explicite Φ_X de y dans Σ n'est pas, dans le cas général, bornée par un polynôme en $|\Sigma|+|X|$ sous l'hypothèse $\mathsf{NP} \cap \mathsf{coNP} \not\subseteq \mathsf{P/poly}$, considérée usuellement adoptée en théorie de la complexité (Lang et Marquis 2008).

Ainsi, plutôt que de détecter des portes et de les remplacer dans Σ pour éliminer des variables de l'ensemble des variables de sortie, notre méthode de pré-traitement recherche des variables de sortie et les oublie dans Σ. Plus précisément, l'objectif est de déterminer dans un premier temps une bipartition de définition $\langle I, O\rangle$ de Σ.

Définition 65 (Bipartition de définition)

Soit Σ une formule propositionnelle de $\mathcal{CPL}$. Une *bipartition de définition* de Σ est un couple $\langle I, O\rangle$ tel que $I \cup O = Var(\Sigma)$, $I \cap O = \varnothing$, et toute variable $o \in O$ est définie à partir de I dans Σ.

Algorithme 5.10 : B + E

input : une formule CNF Σ
output : une formule CNF Φ telle que $\|\Phi\| = \|\Sigma\|$

```
1 O ⟵ B(Σ);
2 Φ ⟵ E(O, Σ);
3 return Φ
```

Dans un second temps, des variables de O sont oubliées de la formule Σ pour la simplifier, ce qui donne le prétraitement B + E (*B(ipartition), et É(limination)*) illustré à l'algorithme 5.10.

Un atout important de cet algorithme est que chacune des deux phases peut être adaptée dans le but de garder le pré-traitement le plus léger possible du point de vue du temps de calcul. D'un côté, il n'est pas nécessaire de déterminer une bipartition $\langle I, O\rangle$ de Σ telle que le cardinal de O soit maximal (il peut être suffisant de déterminer un ensemble O de cardinal « raisonnable »). D'un autre côté, il n'est pas non plus nécessaire d'oublier dans Σ toutes les variables de O, il peut suffire de se focaliser sur un sous-ensemble $E \subseteq O$. Plus formellement, la correction de `B + E` est établie par la proposition suivante (Lagniez et Marquis 2017a) :

Proposition 1

Soit Σ une formule propositionnelle de $\mathcal{CPL}$. Soit $\langle I, O\rangle$ une bipartition de définition de $Var(\Sigma)$. Soit $E \subseteq O$. Alors $\|\Sigma\| = \|\exists E.\Sigma\|$.

La conjugaison du fait de n'identifier qu'un sous-ensemble O de variables de sortie lors de l'étape de bipartition avec le fait de ne considérer qu'un sous-ensemble $E \subseteq O$ de ces variables lors de la phase d'élimination apporte un réel gain. En effet, le calcul d'une base minimale (c'est-à-dire un sous-ensemble I de cardinal minimal tel que toute variable de $O = Var(\Sigma) \setminus I$ soit définissable dans Σ à partir de I) (Lang et Marquis 2008) serait beaucoup trop coûteux ; dans le pire des cas, un algorithme de type *branch-and-bound* pour calculer une telle base nécessiterait un nombre de tests de définissabilité exponentiel dans le nombre de variables de Σ. De plus, bien qu'oublier des variables dans Σ diminue évidemment le nombre de variables qui y apparaissent, cela peut aussi conduire à une augmentation exponentielle de sa taille. De ce fait, n'éliminer de Σ qu'un sous-ensemble E des variables de O permet de sélectionner des variables pour lesquelles la phase d'élimination n'augmente pas de manière trop importante la taille de Σ (à la NiVER (Subbarayan et Pradhan 2005)). Plus précisément, l'élimination d'une variable de sortie ne sera effectuée que si la taille de Σ après l'élimination reste suffisamment faible, une fois l'application de pré-traitements supplémentaires achevée. Parmi ces prétraitements additionnels qui préservent l'équivalence, nous trouvons la simplification d'occurrences et la vivification (déja présentées en section 5.2.2).

Exemple 23 (Suite de l'exemple 22)

Aucune équivalence de littéraux, portes `AND`, `OR`, ou `XOR` n'est conséquence logique de Σ. Cependant, dans la mesure où Σ implique

$$d \leftrightarrow (a \wedge (b \vee c)) \text{ et } e \leftrightarrow (\neg a \vee (b \leftrightarrow c))$$

$\langle\{a, b, c\}, \{d, e\}\rangle$ est une bipartition de définition de $Var(\Sigma)$. En oubliant d et e dans Σ, les deux clauses non tautologiques $a \vee c$ et $a \vee b \vee c$ sont générées, ce qui conduit à une formule `CNF` équivalente à $\exists\{d, e\}.\Sigma$ donnée par :

$$a \vee b, \quad a \vee c, \quad a \vee b \vee c,$$

qui peut être simplifiée en $(a \vee b) \wedge (a \vee c)$. Cette formule `CNF` admet 5 modèles, ce qui est donc aussi le cas de Σ.

Algorithme 5.11 : `B`

input : une formule `CNF` Σ
output : un ensemble O de variables de sortie, *i.e.*, des variables définies dans Σ à partir de $I = Var(\Sigma) \setminus O$

```
1 ⟨Σ, O⟩ ⟵ backbone(Σ);
2 V ⟵ trier(Var(Σ));
3 I ⟵ ∅;
4 foreach x ∈ V do
5     if défini?(x, Σ, I ∪ succ(x, V), max#C) then
6         O ⟵ O ∪ {x};
7     else
8         I ⟵ I ∪ {x};
9 return O ;
```

L'algorithme 5.11 montre comment une bipartition $\langle I, O\rangle$ de $Var(\Sigma)$ est calculée de façon gloutonne par `B`. À la ligne 1, `backbone`(Σ) calcule le *backbone* de Σ et initialise O avec ses variables. La propagation unitaire des contraintes est ensuite appliquée à Σ conjoint aux littéraux de son *backbone*, ce qui conduit à simplifier la formule et à éliminer de celle-ci les variables des littéraux du *backbone*. À la ligne 2, nous ordonnons les variables de Σ de celles possédant le moins d'occurrences dans Σ à celles en possédant le plus. À la ligne 5, `défini`? utilise la méthode de Padoa (Théorème 3) afin de déterminer si x est définie dans Σ à partir de $I \cup$ `succ`$(x, \mathcal{V})$, où `succ`$(x, \mathcal{V})$ est l'ensemble des variables de V qui apparaissent après x dans V. La fonction `défini`? utilise un prouveur `SAT solve` basé sur l'architecture CDCL pour effectuer le test d'incohérence requis par la méthode de Padoa. Dans notre implémentation, la formule `CNF` d'entrée de `solve` est $\Sigma \wedge \Sigma'_{\varnothing} \wedge \bigwedge_{z \in Var(\Sigma)}((\neg s_z \vee \neg z \vee z') \wedge (\neg s_z \vee z \vee \neg z'))$, où figurent des littéraux hypothèses (« *assumptions* ») : pour chaque z appartenant à $I \cup$ `succ`$(x, \mathcal{V})$, nous introduisons une variable hypothèse s_z, c'est-à-dire une clause unitaire qui joue un rôle de sélecteur (quand elle est affectée à vrai, elle crée une équivalence entre z et sa copie z') ; x et $\neg x'$ sont elles-aussi ajoutées en tant qu'hypothèses de façon à pouvoir réduire le test de définissabilité considéré sur x à un test d'incohérence. L'intérêt d'utiliser des hypothèses est grand, dans la mesure où cela permet au prouveur `SAT` de conserver les clauses qu'il apprend, pour aider à la recherche lors des appels suivants. `défini`? admet un paramètre `max#C`, qui borne le nombre de clauses que le prouveur peut apprendre lors de cet appel ; si ce nombre est atteint alors qu'aucune contradiction n'a pu être détectée, la fonction retourne *false* (x n'est pas considérée comme définie dans Σ à partir de $I \cup$ `succ`$(x, \mathcal{V})$, alors que nous aurions pu potentiellement démontrer l'inverse avec une valeur de `max#C` supérieure). Nous voyons de ce fait que le nombre de variables de sortie déterminées par `B` n'est pas nécessairement maximal ; ceci est réalisé dans une optique d'efficacité. Enfin, on peut observer que le nombre d'appels à `solve` n'excède pas le nombre de variables de Σ.

L'algorithme 5.12 montre comment les variables de O sont éliminées de Σ par `E`. P est l'ensemble des variables de E dont l'élimination est possiblement reportée, et est initialisé à la ligne 2 avec l'ensemble O. La boucle principale à la ligne 3 est répétée tant que l'élimination d'une des variables a été réalisée (ligne 16). À la ligne 4, l'ensemble E des variables à tenter d'éliminer durant l'itération courante est initialisé avec P, qui est réinitialisé à $\varnothing$. À la ligne 5, les clauses de Φ sont successivement vivifiées, en utilisant une légère variante de l'algorithme de Lagniez et Marquis (2014) ; le paramètre additionnel E est utilisé pour ordonner les clauses de Σ de sorte que les littéraux portant sur les variables de E soient traités en premier (c'est-à-dire que

Algorithme 5.12 : `E`

input : une formule `CNF` Σ et un ensemble de variables de sortie $O \subseteq Var(\Sigma)$
output : une formule `CNF` Φ telle que $\Phi \equiv \exists E.\Sigma$ avec $E \subseteq O$

```
1  Φ ⟵ Σ;
2  itérer ⟵ true ; P ⟵ O;
3  while itérer do
4      E ⟵ P ; P ⟵ ∅ ; itérer ⟵ false;
5      Φ ⟵ vivificationSimpl(Φ, E);
6      while E ≠ ∅ do
7          x ⟵ select(E, Φ);
8          E ⟵ E \ {x};
9          Φ ⟵ occurrenceSimpl(Φ, x);
10         if #(Φ_x) × #(Φ_¬x) > max#Res then
11             P ⟵ P ∪ {x}
12         else
13             R ⟵ removeSub(Res(x, Φ), Φ);
14             if #((Φ \ Φ_{x,¬x}) ∪ R) ≤ #(Φ) then
15                 Φ ⟵ (Φ \ Φ_{x,¬x}) ∪ R;
16                 itérer ⟵ true;
17             else
18                 P ⟵ P ∪ {x}
19 return Φ ;
```

nous allons tenter d'éliminer prioritairement les occurrences des littéraux de E). Nous entrons à la ligne 6 dans la boucle interne qui va procéder à une itération par variable de E. Une des variables x de cet ensemble est sélectionnée à la ligne 7 en comptant le nombre $\#(\Phi_x)$ (resp. $\#(\Phi_{\neg x})$) de clauses où x apparaît sous la forme de son littéral positif (resp. négatif) ; la variable x sélectionnée est l'une de celles qui minimisent $\#(\Phi_x) \times \#(\Phi_{\neg x})$, valeur qui majore le nombre de résolvantes possiblement générées par l'élimination de x dans Φ. Puis x est finalement retirée de E à la ligne 8. Ensuite, avant de générer l'ensemble R des résolvantes non tautologiques des clauses de Φ sur x (calculées ligne 13, par la fonction `Res`), nous tentons dans un premier temps d'éliminer des occurrences de x en utilisant la fonction `occurrenceSimpl` (voir l'algorithme 5.4) dans le but de diminuer le cardinal de R (ligne 9). Nous vérifions ensuite à la ligne 10 que le majorant du cardinal de R recalculé après simplification de Φ ne dépasse pas une constante prédéfinie `max#Res`. Si tel est le cas, alors l'élimination de x dans Φ est possiblement remise à une itération future de la boucle principale, ce qui est réalisé en la réintégrant à l'ensemble P (ligne 11). Dans le cas contraire, l'ensemble R est simplifié en lui retirant les clauses sous-sommées par une autre clause de R ou une clause de Φ (`removeSub`, ligne 13). Nous vérifions ensuite, à la ligne 14, que l'élimination de x dans Φ (en retirant de Φ les clauses mentionnant x et en lui ajoutant les résolvantes de R) n'augmente pas le nombre de clauses de Φ. S'il y a augmentation alors l'élimination de x dans Φ est possiblement reportée (ligne 18), sinon l'élimination de x dans Φ est actée. Nous voyons qu'il n'y a aucune assurance que toutes les variables de O soient effectivement éliminées par `E` ; encore une fois, ce comportement est dicté par un souci d'efficacité.

5.2.3 Discussion

Nous avons défini deux nouveaux préprocesseurs `B+E` et `pmc`, qui associent à une formule `CNF` Σ donnée une formule `CNF` qui admet le même nombre de modèles que Σ, mais qui est souvent plus simple à traiter d'un point de vue du comptage de ses modèles ou de sa compilation. `pmc` permet deux modes de pré-traitement : mode équivalent et mode préservation du nombre de modèles. En ce qui concerne le mode équivalent de `pmc`, les expérimentations que nous avons conduites dans (Lagniez et Marquis 2017b) ont montré que l'application de ce pré-traitement à Σ permet un gain important en terme de temps de calcul pour le comptage de modèles et la compilation en `d-DNNF` via les approches de l'état de l'art. De plus, dans le cadre de la compilation, l'application de `pmc` permet de construire des formes compilées qui sont plus petites, permettant ainsi de réaliser des requêtes sur la forme compilée de manière plus efficace.

Les deux algorithmes de pré-traitement s'appuient tous les deux sur la notions de définissabilité pour éliminer des variables tout en conservant le nombre de modèles de la formule. La différence est que `pmc` s'appuie sur une recherche explicite de portes logiques, tandis que `B+E` s'appuie sur une recherche implicite de variables définies dans Σ pour simplifier cette formule. Nous avons expérimenté ces deux algorithmes de pré-traitement en mode préservation du nombre de modèles sur de nombreuses instances provenant de différentes familles de jeux d'essai (Lagniez et Marquis 2017b, Lagniez et al. 2016b). Nos expérimentations montrent que l'application de ce type de pré-traitement à Σ permet un gain important en terme de temps de calcul pour le comptage de modèles via des compteurs exacts de l'état de l'art, comparé au temps nécessaire à ces compteurs lorsqu'aucun prétraitement n'est appliqué ou lorsque le pré-processeur `pmc` est employé.

Ce travail ouvre plusieurs perspectives pour des recherches futures. Tout d'abord, en ce qui concerne `B` + `E`, il serait intéressant de considérer d'autres heuristiques dans `B` pour calculer la bi-partition, ou encore d'intégrer directement la recherche de portes logiques. Comme la taille de la formule produite est un critère très important, il serait intéressant de déterminer comment adapter la valeur des constantes `max#C` et `max#Res` en fonction de l'instance considérée. Puisqu'elle n'augmente pas la taille de la formule, nous pensons aussi que la méthode de pré-traitement qui recherche et remplace des équivalences entre littéraux devrait être réalisée en pré-traitement de `B+E`.

D'autres perspectives concernent la notion de comptage de modèles projetés (Aziz et al. 2015), dont le but est le calcul de $\|\exists E.\Sigma\|$, pour un ensemble E et une formule Σ donnés. Un tel calcul est utile, par exemple pour l'analyse et la quantification de flots d'informations dans les programmes et des compteurs de modèles projetés ont ainsi été développés à cet effet (Klebanov et al. 2013). Au lieu d'appliquer `B` + `E` suivi d'un appel à un compteur de modèles pour calculer $\|\Sigma\|$, nous pourrions appliquer `B` puis invoquer un compteur de modèles projetés (où la projection se ferait sur I). Réciproquement, lorsqu'un comptage de modèles projetés est requis, nous pourrions appliquer `E` sur E et Σ comme un pré-processeur du compteur de modèles projetés considéré. Il serait donc intéressant d'implémenter ces deux approches afin d'évaluer leur intérêt pratique.

En ce qui concerne `pmc`, comme pour `B+E`, il serait intéressant de déterminer comment adapter les constantes `maxX`, `maxA` et `numTries` en fonction de l'instance considérée. Une autre piste d'amélioration concerne le remplacement des variables par leurs définitions. Puisque dans `pmc` les définitions sont données explicitement, il serait possible d'effectuer le remplacement sans transformer systématiquement la formule en `CNF`. Plus précisément, nous pourrions réaliser les remplacements en considérant un langage hybride à mi-chemin entre circuit et `CNF`. Ensuite,

nous pourrions soit transformer la formule en CNF ou modifier un solveur SAT afin de travailler avec ce type de formule. Cependant, avant de réaliser cela, il serait possible de travailler sur les clauses hybrides afin de les réduire au maximum (ce qui ne peut pas être fait si nous réalisons les remplacements de manière incrémentale). Il est important de noter que, si nous choisissons la solution qui consiste à travailler avec un langage hybride, il faudra bien faire attention à repenser les heuristiques de choix de variables ainsi que la méthode de *caching*.

Il est bien connu qu'utiliser une méthode de pré-traitement durant le processus de recherche d'une solution permet d'améliorer les performances des solveurs SAT de l'état de l'art (Järvisalo et al. 2012b, Luo et al. 2017). Dans nos travaux futurs, nous souhaitons intégrer dans un compteur de modèles (et/ou un compilateur) du *in-processing*. Au vu du nombre de nœuds à explorer, il est clair qu'il ne sera pas possible d'appeler une méthode de pré-traitement à chaque fois que nous séparons l'espace de recherche suivant une variable. De plus, il faudra s'assurer que ces pré-traitement n'ont pas un impact néfaste sur le processus de *caching*. Cependant, étant donné les gains de performance obtenus grâce à nos méthodes de pré-traitement sur les problèmes que nous avons considérés dans nos expérimentations, il semble que l'exploitation de telles méthodes durant la recherche peut permettre de traiter des problèmes qui sont pour le moment hors de portée des compteurs de modèles actuels.

Enfin, dans (Lagniez et Marquis 2017b) nous avons étudié l'impact des méthodes de pré-traitement sur la largeur d'arbre du graphe de contraintes associé à la formule CNF prétraitée. Dans la section suivante, nous présentons une heuristique de choix de variables qui s'appuie sur la recherche d'un ensemble de variables permettant de couper la formule en différentes composantes connexes (Lagniez et Marquis 2017a). La particularité de cette approche étant de ne recalculer cet ensemble que lorsqu'il a été épuisé par l'heuristique de choix de variables, nous pourrions envisager d'utiliser les méthodes de pré-traitement élémentaires permettant de réduire la largeur d'arbre du graphe de contrainte afin de favoriser la recherche d'ensembles coupants de petites tailles.

5.3 Description du compilateur d4

d4 est un compilateur [5] fonctionnant en *top-down* qui associe à n'importe quelle formule CNF une représentation équivalente en Decision-DNNF. Comme les compilateurs de l'état de l'art C2D et Dsharp qui ciblent le même langage, d4 explore l'espace des interprétations propositionnelles en s'appuyant sur le parcours d'un arbre. d4 tire aussi avantage des techniques déjà utilisées dans C2D et Dsharp, c'est-à-dire l'analyse en composantes connexes disjointes, l'analyse de conflit, les retours en arrière non chronologiques et la mémorisation dans une table de hachage des formules déjà compilées. La principale différence entre d4 et les compilateurs de l'état de l'art C2D et Dsharp tient à l'heuristique de décomposition utilisée afin de guider l'exploration de l'espace de recherche et la manière dont les formules sont stockées dans la table de *cache*.

Comme Dsharp, d4 utilise une heuristique de branchement dynamique sur la formule CNF conditionnée par l'interprétation partielle courante, tandis qu'une heuristique statique est utilisée dans C2D (voir section 5.1.2). Cependant, comme C2D, d4 cherche à décomposer en composantes connexes l'hyper-graphe dual associé à la formule CNF conditionnée par l'interprétation partielle courante, tandis que l'heuristique de branchement utilisée dans Dsharp est basée sur le score VSADS (*Variable State Aware Decaying Sum*) (Sang et al. 2005) qui n'a pas forcément cet objectif

5. À ne pas confondre avec le Partagas D4 qui est, d'après Pierre Marquis, un excellent cigare.

en ligne de mire (voir section 5.1.2). Ce qui fait l'efficacité de `d4` est que la décomposition est réalisée parcimonieusement (l'outil de partitionnement n'est pas utilisé à chaque décision) et que certaines simplifications ont été réalisées sur la formule afin de minimiser le temps de calcul de l'outil de partitionnement tout en permettant de trouver des décompositions de bonne qualité. C'est cette heuristique et ces simplifications que nous présentons à la section 5.3.1.

Comme nous l'avons vu au début de ce chapitre, le *caching* consiste simplement à stocker dans une table de hachage des couples clef-valeur, où la clef est une formule `CNF` Σ et la valeur est la formule φ compilée correspondant à Σ. La qualité du *caching* peut être mesurée par l'efficacité avec laquelle il est possible de déterminer si une formule a une représentation équivalente dans la table, et elle est aussi déterminée par la taille de la table de *cache*. En fait, les deux aspects (temps et espace) sont clairement liés et ont un impact direct sur la complexité (en temps et espace) du compteur de modèles ou du compilateur dans lequel cette structure est exploitée. À la section 5.3.3, nous présentons un nouveau schéma de *caching* qui permet de réduire la quantité d'informations nécessaires à la représentation des clefs tout en conservant le maximum d'informations sur la formule afin d'augmenter le nombre de formules équivalentes détectées.

Avant de commencer la présentation de ces deux améliorations, nous présentons dans la section suivante l'architecture du compilateur `d4`.

5.3.1 L'architecture de `d4`

Comme pour les compilateurs `C2D` et `Dsharp` (Darwiche 2004, Muise et al. 2012), la représentation en `Decision-DNNF` de Σ calculée par `d4` correspond à la trace d'un solveur `SAT`. Nous avons utilisé notre propre solveur `solve`, lequel est basé sur le solveur de l'état de l'art `MiniSAT 2.2` (Eén et Sörenson 2004). `d4` est un algorithme récursif où chaque appel correspond à un nœud de la `Decision-DNNF` en construction. Avant de commencer la compilation du problème passé en paramètre, un appel à `solve` est réalisé sur la sous-formule qui correspond à Σ conditionné par l'ensemble des littéraux sur le chemin qui va de la racine de la `Decision-DNNF` au nœud de décision courant. Cependant, au lieu de considérer à chaque appel une nouvelle formule, nous avons choisi de travailler sous hypothèses (voir section 4.2 pour plus de détails sur le mode de fonctionnement de ce type d'approche). De cette manière, au lieu d'effectuer un redémarrage du solveur pour chaque point de décision, nous effectuons uniquement un retour arrière jusqu'au point de décision courant. De plus, ce mode de fonctionnement nous permet de conserver les clauses apprises, donc d'augmenter le nombre de littéraux pouvant être déduits par propagation unitaire et donc de ne pas considérer certaines variables déjà décidées. Pour des raisons d'efficacité évidentes, `d4` exploite les mêmes techniques que celles utilisées dans `C2D` et `Dsharp` (l'analyse en composantes connexes disjointes, l'analyse de conflit, les retours arrière non chronologiques et le *caching*). Cependant, l'heuristique utilisée par `d4` pour explorer l'espace de recherche ainsi que la méthode de *caching* implémentée sont différentes de celles utilisées dans `C2D` et `Dsharp`.

L'algorithme 5.13 donne le pseudo-code de notre compilateur `d4`. La compilation d'une formule `CNF` Σ est réalisée en appelant `d4`(Σ, $\varnothing$). Cet algorithme est très similaire à celui présenté dans les préliminaires de ce chapitre (voir algorithme 5.2), les parties qui diffèrent de l'approche de base sont colorées en bleu. La première différence tient au fait que cet algorithme prend en entrée un ensemble de variables LV qui sera utilisé ensuite pour restreindre le choix de la prochaine variable de décision réalisé à la ligne 10. La seconde différence apparaît à la ligne 4, elle correspond à la gestion de la structure de *cache* qui sera discutée dans la sous-section 5.3.3. La dernière différence est plus flagrante et concerne l'heuristique de choix de variables de décision. Lors du parcours des composantes connexes (lignes 7 – 14), pour un c donné, les variables LV

Algorithme 5.13 : d4 (Σ, LV)

input : Σ une formule CNF
input : une liste de variables LV
output : le nœud racine N de la Decision-DNNF représentant Σ

```
1  S ← solve(Σ);
2  if S = {∅} then return leaf(⊥);
3  if Var(Σ) = ∅ then return aNode(S, [leaf(⊤)]) ;
4  if cache(Σ) ≠ nil then return aNode(S, [cache(Σ)]) ;
5  comps ← connectedComponents(Σ);
6  LN_d ← [];
7  foreach c ∈ comps do
8      LV_c ← restrict(LV, Var(c));
9      if LV_c = ∅ et #(Var(c)) ∈ [10, 5000] then
10         LV_c ← sort(HGP(c));
11     v ← head(LV_c);
12     LV_c ← tail(LV_c);
13     N_d ← ite(v, d4(c|¬v, LV_c), d4(c|v, LV_c));
14     LN_d ← add(N_d, LN_d);
15 N_∧ ← aNode(S, LN_d);
16 cache(Σ) ← N_∧;
17 return N_∧;
```

sont filtrées de manière à ne conserver que des variables de c (ligne 7). Nous testons ensuite à la ligne 9 si la liste résultante LV_c est vide, et si le nombre de variables de c est suffisant sans être trop gros (dans nos expérimentations nous avons considéré $|c| \in [10, 5000]$). Si ces conditions sont satisfaites, alors nous mettons à jour la liste de variables LV_c. Pour cela, nous appelons la fonction $\mathsf{HGP}(c)$ qui calcule un *cutset* sur une version simplifiée de l'hypergraphe dual de la formule Σ restreinte aux variables de c (ce point sera discuté dans la sous-section suivante). Les variables de ce *cutset* sont triées suivant une heuristique *via* un appel à la fonction $\mathsf{sort}(\mathsf{HGP}(c))$ (nous avons utilisé le score VSADS dans nos expérimentations) pour obtenir une liste mise à jour. Ensuite, la première variable v de LV_c est selectionnée (ligne 11) et supprimée de LV_c (ligne 12). L'algorithme va alors construire un nœud de décision en réalisant un appel récursif à d4 comme cela a déjà été discuté dans les préliminaires.

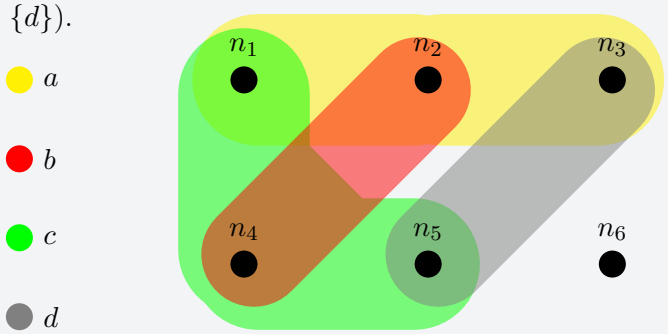
5.3.2 Une heuristique dynamique favorisant la décomposition (DECOMP)

Comme nous l'avons déjà souligné en introduction de ce chapitre, les méthodes de compilation en Decision-DNNF (et les compteurs de modèles) suivent la trace d'un solveur de type DPLL. Cependant, contrairement au cas où une seule solution est recherchée, le processus de compilation nécessite de parcourir l'ensemble de l'arbre de recherche. De la même manière que pour le problème de décision, l'heuristique de choix de variables a un impact très important sur la taille de la formule compilée. En fait, cette heuristique est encore plus cruciale dans le cadre de la compilation car il n'est pas envisageable pour des raisons d'efficacité d'effectuer des redémarrages afin de reconsidérer des mauvais choix que nous aurions pu réaliser au début de la recherche.

Comme nous avons pu le montrer dans le cadre de la compilation en `MDDG`[6] de formules exprimées à l'aide de contraintes (Lagniez et al. 2017b), les heuristiques initialement définies pour rechercher une solution ne sont pas nécessairement adaptées pour la compilation. Ainsi, l'heuristique `VSADS` (Sang et al. 2004), qui est très similaire à l'heuristique `dom/wdeg` (Boussemart et al. 2004) utilisée dans le cadre du problème de satisfaction de contraintes, a été initialement prévue pour traiter en particulier les parties difficiles du problème et non pour la compilation de connaissances. En fait, cette heuristique n'exploite pas la principale spécificité du langage des `Decision-DNNF`, c'est-à-dire la possibilité de considérer des nœuds $\wedge$ décomposables. En effet, un des leviers pouvant être activé afin de réduire la taille de l'espace de recherche parcouru lors du processus de compilation consiste à identifier des composantes connexes de la formule courante et de les compiler séparément en les combinant *via* un nœud $\wedge$ décomposable. Ce concept s'appuie sur la notion d'hypergraphe dual de la formule `CNF`, noté est $\texttt{DH}(\Sigma) = (N_d, H_d)$ où $N_d = \{\delta | \delta \in \Sigma\}$ et $H_d = \{\{\delta \in \Sigma \mid x \in Var(\delta)\} \mid x \in Var(\Sigma)\}$. Les nœuds de N_d correspondent aux clauses de Σ et H_d est une famille de parties de N_d qui associe à chaque variable de Σ les clauses dans lesquelles elles apparaissent (pour chaque variable x, l'hyper-arête correspondante est l'ensemble des clauses $Cls_\Sigma(x)$ de Σ contenant x).

Exemple 24

Soit $\Sigma = (a \vee b) \wedge (a \vee \neg c) \wedge (a \vee \neg d) \wedge (b \vee \neg c) \wedge (b \vee \neg d)$. Nous avons $\texttt{DH}(\Sigma) = (N_d, H_d)$, avec $N_d = \{n_1, n_2, n_3, n_4, n_5\}$ (n_1 (resp. n_2, n_3, n_4, n_5) correspondant à $a \vee b$ (resp. $a \vee \neg c$, $a \vee \neg d$, $b \vee \neg c$, $b \vee \neg d$) et $H_d = \{\{n_1, n_2, n_3\}, \{n_1, n_4, n_5\}, \{n_2, n_4\}, \{n_3, n_5\}\}$ (étiqueté respectivement par $\{a\}$, $\{b\}$, $\{c\}$, et $\{d\}$).



Ce concept de décomposition est fondamental car nous savons qu'il est possible de produire une `Decision-DNNF` de « petite » taille lorsque nous pouvons trouver une décomposition arborescente de l'hypergraphe dual de la formule telles que les séparateurs sont « petits » (Amarilli et al. 2018). Cependant, ces décompositions ne prennent pas en compte la sémantique du problème et elles conduisent généralement à des heuristiques statiques pour le choix de variable. Dans (Lagniez et Marquis 2017a), en collaboration avec Pierre Marquis, nous avons proposé dans le compilateur `d4` une approche permettant de combiner ces deux types d'heuristiques, c'est-à-dire que nous favorisons la décomposition en composantes connexes tout en proposant une heuristique dynamique.

Comme `Dsharp` (Muise et al. 2012), et contrairement à `C2D` (Darwiche 2004), `d4` utilise donc une heuristique de choix de variables dynamique. Cependant, comme `C2D` et contrairement à

6. Le langage `MDDG` étant le langage `decision-DNNF` aux domaines multi-valués.

Dsharp, l'heuristique de choix de variables de d4 est orientée d'une manière à sélectionner en priorité des variables permettant de séparer la formule courante en au moins deux composantes connexes. Pour cela, nous utilisons une procédure de partitionnement d'hypergraphe (HGP, pour *HyperGraph Partionning*) s'appuyant sur l'outil de partitionnement PaToH[7] (Çatalyürek et Aykanat 2011) afin de trouver des *cutsets* de l'hypergraphe dual de la formule courante.

Étant donné un tel hypergraphe, PaToH recherche de manière heuristique un sous-ensemble C de H_d contenant un nombre aussi restreint que possible d'éléments tel que la suppression des hyper-arêtes de C de l'hypergraphe H_d conduit à un hypergraphe contenant au moins deux composantes disjointes ayant une taille à peu près similaire (voir le papier de Çatalyürek et Aykanat (2011) pour plus de détails). Lorsque les variables correspondant aux éléments de C sont assignées (et cela quelle que soit la valeur de vérité des variables) il est garanti que la formule courante conditionnée par cette affectation contient au moins deux composantes connexes, c'est-à-dire que nous allons pouvoir générer un nœud $\wedge$ décomposable dans la forme compilée.

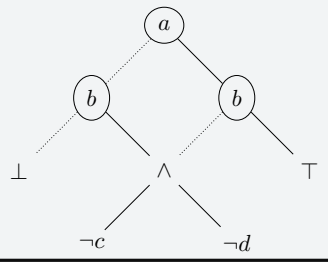
Lorsque C2D est utilisée avec son option `-dt_method 1`, un dtree (Darwiche et Hopkins 2001) sur l'hypergraphe de la formule initiale est calculé avant de commencer la compilation. Pour cela, l'outil de partitionnement hMETIS[8] est utilisé afin de calculer un *cutset* C de l'hypergraphe dual (N_d, H_d) de la formule Σ, puis chacune des composantes connexes de l'hypergraphe, obtenues en supprimant les hyper-arêtes de C, sont considérées récursivement afin d'être de nouveau partitionnées à leur tour et cela tant que l'hypergraphe considéré possède plus d'un nœud. Dans d4, nous avons choisi d'utiliser une stratégie similaire mais dynamique : pour chaque affectation des variables correspondant aux hyper-arêtes contenues dans C, nous considérons l'hypergraphe associé à la formule Σ conditionnée (et simplifiée par bcp) par cette affectation comme un candidat possible pour être décomposé. Afin de conserver le meilleur des deux heuristiques utilisées dans C2D and Dsharp, nous utilisons l'heuristique VSADS afin de choisir la prochaine variable à sélectionner dans l'ensemble des variables représentant les hyper-arêtes de C.

Les deux stratégies de décomposition ont des avantages et des inconvénients. D'une part, calculer une décomposition statique au début de la recherche comme dans C2D permet de limiter le nombre d'appels à l'outil de partitionnement. Cela n'est pas vrai pour la stratégie dynamique qui a besoin d'être appelée pour chaque affectation, ce qui peut nécessiter une quantité de ressources trop importante relativement au temps nécessaire pour réaliser le processus de compilation. En effet, en profilant le code de d4, nous avons observé qu'en moyenne, pour les instances considérées dans nos expérimentations, au moins 50% du temps de compilation utilisé par d4 est utilisé par PaToH (et pour certaines instances ce temps peut dépasser 90% du temps de calcul). D'un autre côté, la stratégie de décomposition statique ne permet pas en général d'obtenir la meilleure décomposition. En effet, pour chaque *cutset* C, l'affectation des variables correspondant aux hyper-arêtes de C (et les propagations unitaires pouvant résulter de cette affectation) peut avoir un impact important sur la structure et la taille de l'hypergraphe associé à la formule simplifiée. Ainsi, considérer ces affectations permet de considérer la sémantique de la formule, et permet donc de trouver des décomposition différentes et meilleures par rapport à celles pouvant être obtenues uniquement en coupant le graphe par rapport à C.

7. PaToH – *Partitioning Tools for Hypergraph*, v. 3.2 (http://bmi.osu.edu/~umit/software.html)
8. hMETIS est similaire à PaToH. Cependant, nous avons choisi d'utiliser PaToH car la librairie nous permettant d'utiliser hMETIS dans notre programme est compilée en 32 bits, tandis que notre programme est compilé pour fonctionner en 64 bits.

Exemple 25

Considérons l'hypergraphe de l'exemple 24. Soit $C = \{\{n_1, n_2, n_3\}, \{n_1, n_4, n_5\}\}$ le *cutset* calculé par $\texttt{DH}(\Sigma)$: supprimer les éléments de C dans H_d conduit à la création de deux hypergraphes disjoints car les hyper-arêtes restantes $\{n_2, n_4\}$, $\{n_3, n_5\}$ ne partagent aucun nœud. $\{n_1, n_2, n_3\}$ (resp. $\{n_1, n_4, n_5\}$) sont étiquetées par l'ensemble de variables $\{a\}$ (resp. $\{b\}$), ainsi assigner ces deux variables a et b dans Σ (quelle que soit l'affectation considérée) conduit à la création de deux composantes disjointes (l'une d'elles est la clause $b \vee \neg c$ simplifiée par l'affectation de la variable b et l'autre est la clause $b \vee \neg d$ aussi simplifiée par l'affectation de la variable b). La figure suivante illustre la construction d'une `Decision-DNNF` où l'ensemble des affectations possibles sur l'ensemble de variables $\{a, b\}$ est considéré en priorité :



Puisque le calcul du *cutset* est une étape critique et coûteuse en temps, `d4` n'appelle pas l'algorithme de partitionnement à chaque fois qu'une variable est sélectionnée pour créer un nœud de décision. Plus précisément, une nouvelle partition est calculée seulement si la condition à la ligne 9 est satisfaite, c'est-à-dire que la liste des variables courantes LV_c est vide, ou que le nombre de variables de la formule Σ obtenue après l'application de la propagation unitaire (projeté sur les variables de la composante courante c) n'est ni trop petit, ni trop grand (entre 10 et 5000 dans notre implémentation).

Afin d'améliorer les temps de calcul de l'algorithme HGP sur notre problème, nous avons aussi défini plusieurs règles afin de simplifier l'hypergraphe (que ce soit en terme de nombre de nœuds ou d'hyper-arêtes) associé à la sous-formule de Σ définie sur les variables de c. Nous avons montré que nos simplifications permettaient de calculer un *cutset* LV_c sur l'hypergraphe réduit qui pouvait être étendu à l'hypergraphe de la formule courante, c'est-à-dire que tout affectation complet γ_c des variables de LV_c (calculé à la ligne 10) est tel que $\texttt{bcp}\ (c \mid \gamma_c)$ permet de produire au moins deux composantes connexes. Notre première simplification exploite la détection de littéraux équivalents pouvant être obtenus *via* l'utilisation de la fonction `bcp`. Plus précisément, nous utilisons une méthode proche de celle implémentée pour `equivSimpl` (voir algorithme 5.3) afin de construire un ensemble de classes d'équivalence disjointes dont l'union couvre toutes les variables de c. Pour cela, nous allons considérer chaque variable v de c de manière consécutive et dans l'ordre lexicographique. Ensuite, nous associons à v sa classe d'équivalence $\{\{v\} \cup (var(\texttt{bcp}(\Sigma_{|\ell}) \cap \texttt{bcp}(\Sigma_{|\neg\ell})) \cap c)\}$ construite en utilisant la propagation unitaire, où ℓ est le littéral positif associé à v.[9] L'algorithme 5.14 donne le pseudo-code d'une telle procédure.

Une fois que l'ensemble des classes d'équivalence $E = \{e_1, e_2, \ldots, e_k\}$ est calculé, pour chaque classe d'équivalence $\{v, v_1, v_2, \ldots, v_m\}$ où v est le littéral utilisé pour construire la classe d'équivalence, les hyper-arêtes étiquetées par l'ensemble des variables $\{v_1, v_2, \ldots, v_m\}$ sont supprimées

9. Si nous détectons par propagation unitaire que $\texttt{bcp}(\Sigma_{|\ell})$ (resp. $\texttt{bcp}(\Sigma_{|\neg\ell})$) conduit à une incohérence, alors nous sauvegardons le fait que $\neg\ell$ (resp. ℓ) doit être propagé.

Algorithme 5.14 : `equivClass`

input : Σ une formule `CNF`
input : c un sous-ensemble des variables de Σ
output : $E = \{e_1, e_2, \ldots, e_k\}$ un ensemble de classe d'équivalence tel que $\forall i, j, e_i \cap e_j = \varnothing$ et $c = e_1 \cup e_2 \cup \ldots \cup e_k$

$E \longleftarrow \varnothing$;
$c' \longleftarrow c$;
while $\exists v \in c'$ **do**
  $\ell \longleftarrow$ the positive literal of v;
  $e \longleftarrow \{\{v\} \cup (var(\texttt{bcp}(\Sigma_{|\ell}) \cap \texttt{bcp}(\Sigma_{|\neg\ell})) \cap c')\}$;
  $E \longleftarrow E \cup \{e\}$;
  $c' \longleftarrow c' \setminus e$;
return E

de $\texttt{DH}(\Sigma_c)$, et l'hyper-arête $Cls_{\Sigma c}(v)$ est remplacée par $Cls_{\Sigma_c}(var(\ell)) \cup \bigcup_{i=1}^{m} Cls_{\Sigma_c}(v_i)$.[10] Démontrons à présent que tout *cutset* représenté par l'ensemble d'étiquettes LV_c obtenu sur cet hypergraphe $HG_{\Leftrightarrow}$ produira nécessairement une coupure dans le graphe $\texttt{DH}(\Sigma_c)$ une fois que nous aurons réalisé la propagation unitaire sur la formule Σ_c. Pour commencer, considérons une partition $\{V_1, V_2, LV_c\}$ des étiquettes des hyper-arêtes de $HG_{\Leftrightarrow}$ qui permette de séparer $HG_{\Leftrightarrow}$ en au moins deux composantes connexes (dont l'une est construite sur les étiquettes de V_1 et l'autre sur les étiquettes de V_2) lorsque nous supprimons les hyper-arêtes étiquetées par LV_c (cela est possible puisque l'ensemble des hyper-arêtes étiquetées par LV_c est un *cutset* de $HG_{\Leftrightarrow}$). Raisonnons par l'absurde et supposons qu'il existe une interprétation complète γ_c des variables de LV_c qui ne sépare pas la formule Σ_c en au moins deux composantes connexes. Étant donné la manière dont les classes d'équivalences ont été calculées, nous savons que l'affectation du représentant d'une classe d'équivalence propage systématiquement les littéraux contenus dans leur classe d'équivalence. Dit autrement, les variables de $LV'_c = \bigcup_{e_i \in E | e_i \cap LV_c \neq \varnothing} e_i$ seront toujours assignées après l'application de $\texttt{bcp}(\Sigma_{|\gamma_c})$, et cela quel que soit γ_c. Par conséquent, si LV_c ne permet pas de séparer la formule en au moins deux composantes connexes, alors LV'_c ne peut pas séparer la formule en au moins deux composantes connexes. Considérons $\{V'_1, V'_2, LV'_c\}$ une partition de c telle que $V'_1 = \bigcup_{e_i \in V_1 | e_i \cap V_1 \neq \varnothing} e_i$, $V'_2 = \bigcup_{e_i \in V_2 | e_i \cap V_2 \neq \varnothing} e_i$ et $c = V'_1 \cup V'_2 \cup LV'_c$ (il est simple de voir que c'est une partition, car $\{V_1, V_2, LV_c\}$ est une partition et E est une partition). Puisque les hyper-arêtes étiquetées par LV'_c ne forment pas un *cutset* qui permet de séparer $\texttt{DH}(\Sigma_c)$ en deux composantes disjointes, il existe un nœud n, et deux hyper-arêtes $x_1 \in V'_1$ et $x_2 \in V'_2$ avec $n \in Cls_{\Sigma_c}(x_1) \cap Cls_{\Sigma_c}(x_2)$. Considérons à présent $HG_{\Leftrightarrow}$ et focalisons-nous sur le nœud n. Tout d'abord, considérons x'_1 (resp. x'_2) le représentant de la classe contenant x_1 (resp. x_2). Il est facile de voir que, par construction de V'_1 et V'_2, nous avons $x'_1 \in V_1$ et $x'_2 \in V_2$. Puisque nous avons transféré les nœuds de l'hyper-arête étiquetée par x_1 (resp. x_2) à x'_1 (resp. x'_2) dans la phase de remplacement, n appartient aux deux hyper-arêtes étiquetées par x'_1 et x'_2. Ce qui contredit le fait que l'ensemble des hyper-arêtes étiquetées par LV_c est un *cutset* de $HG_{\Leftrightarrow}$ et donc conclut la preuve.

Lorsque nous avons considéré l'ensemble des classes d'équivalence, deux règles supplémentaires sont systématiquement appliquées sur l'hypergraphe résultant afin de le simplifier. Le

10. Il est facile de voir que l'hypergraphe que nous obtenons de cette manière correspond exactement à l'hypergraphe de la formule que nous aurions obtenu si nous avions effectué les remplacements des variables équivalentes par leur représentant dans leur classe d'équivalence.

processus de simplification se termine lorsqu'aucune des deux règles ne peut plus être appliquée. La première règle s'applique sur les nœuds de l'hypergraphe. Supposons que l'ensemble de nœuds courants contient deux nœuds n_i et n_j tels que toutes les hyper-arêtes qui contiennent n_i contiennent aussi n_j. Dans un tel cas, il est facile de voir que n_i peut être supprimé de l'ensemble des nœuds, et de l'ensemble des hyper-arêtes qui le contiennent. Les étiquettes des hyper-arêtes contenant n_i sont alors marquées et ajoutées aux étiquettes des hyper-arêtes contenant n_j. En fait cette règle modélise le fait que les variables de la clause étiquetée par n_i sont incluses dans l'ensemble des variables de la clause étiquetée par n_j. La seconde règle concerne la suppression de étiquettes des hyper-arêtes de taille 1 qui ne sont pas marquées. Cette suppression est possible car ces variables ne pourront jamais être utilisées dans le *cutset*. Cette situation modélise le cas où une variable n'apparaît que dans une seule clause : il va de soi que brancher sur cette variable ne permettra pas de réduire le nombre de connexions entre les variables de la partition que nous cherchons à construire.

Exemple 26

Soit $\Sigma = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee a) \wedge (a \vee c \vee d)$ une formule `CNF`. Nous avons $\texttt{DH}(\Sigma) = (N_d, H_d)$, avec $N_d = \{n_1, n_2, n_3, n_4\}$ (n_1 (resp. n_2, n_3, n_4) correspond à $\neg a \vee b$ (resp. $\neg b \vee c$, $\neg c \vee a$, $a \vee c \vee d$) et $H_d = \{\{n_1, n_3, n_4\}, \{n_1, n_2\}, \{n_2, n_3, n_4\}, \{n_4\}\}$ (étiquetées respectivement par $\{a\}$, $\{b\}$, $\{c\}$, et $\{d\}$). La première étape conduit à identifier la classe d'équivalence de a comme étant $\{a, b, c\}$. L'hypergraphe résultant après la première phase de remplacement est $(\{n_1, n_2, n_3, n_4\}, \{\{n_1, n_2, n_3, n_4\}, \{n_4\}\})$. L'ensemble des étiquettes de $\{n_1, n_2, n_3, n_4\}$ est $\{a\}$ et l'ensemble des étiquettes de $\{n_4\}$ est $\{d\}$. Alors les nœuds n_1, n_2, n_3 sont supprimés, a est marqué et ajouté à l'ensemble des étiquettes de $\{n_4\}$ qui devient alors $\{a, d\}$. Finalement, l'étiquette d est supprimé de cet ensemble. L'hypergraphe résultant est donc $(\{n_4\}, \{\{n_4\}\})$ où $\{n_4\}$ est étiqueté par $\{a\}$.

5.3.3 Un nouveau schéma de *caching*

Si nous nous plaçons d'un point de vue de la compilation de connaissances, nous nous rendons rapidement compte que le *caching* permet de construire des structures qui sont des graphes, tandis que ne rien cacher conduit à des formes compilées de types `EDT` (voir la section 5.5 pour une description de ce langage) qui est en théorie et en pratique moins succinct que `Decision-DNNF`. Ainsi, la majorité des compilateurs ou des compteurs de modèles de l'état de l'art utilisent une méthode de *caching* : `Cachet` (Sang et al. 2004), `SharpSAT` (Thurley 2006), `C2D` (Darwiche 2004), et `Dsharp` (Muise et al. 2012).

Dans `d4`, nous avons implémenté et évalué un schéma de *caching* qui apparaît empiriquement efficace comparé aux schémas de *caching* utilisés dans la littérature. Dans ce schéma, les clauses de la formule `CNF` courante sont représentées explicitement comme dans le schéma standard utilisé par `Cachet` (Sang et al. 2004), permettant ainsi de reconnaître certaines entrées qui ne sont pas reconnues par le schéma hybride utilisé dans `SharpSAT` (Thurley 2006). De plus, la taille du *cache* est minimisée en considérant une représentation compacte de l'ensemble des variables s'appuyant sur un encodage en binaire, et en ne considérant qu'un sous-ensemble de clauses de la formule comme clef. Plus précisément, cet ensemble de clauses ne contient pas les clauses satisfaites

(ce qui eststandard), ni les clauses binaires de la formule initiale (comme cela est fait dans SharpSAT), et *plus généralement toute clause qui n'a pas été raccourcie par rapport à sa version dans la formule initiale.* Nous verrons dans la suite que cette restriction est correcte, c'est-à-dire que si deux formules ont la même clef alors elles sont équivalentes, nous montrons aussi que nous retrouvons dans notre structure de *caching* au moins autant de formules équivalentes que dans le cas de l'utilisation du schéma de *caching* utilisé dans Cachet, et nous montrons aussi expérimentalement que ce schéma est efficace en pratique.

Avant d'aller plus en avant dans la description des schémas de *caching*, nous allons définir ce que sont les clefs de notre table de hachage. Considérons, l'assignation γ, augmentée avec les littéraux pouvant être déduits, par propagation unitaire, des variables des nœuds de décision qui vont de la racine jusqu'à la formule courante. La formule Σ que nous souhaitons utiliser comme clef sera alors définie comme un sous-ensemble des clauses de la formule initiale Σ_i, où certaines des clauses ont été réduites par l'assignation γ, où les clauses de Σ_i satisfaites par γ ne sont pas représentées, et où nous ne considérons que les clauses de Σ_i qui sont dans la composante courante. Ainsi la clef que nous souhaitons rechercher dans notre structure de *caching* concerne les éléments de $S(\Sigma) = \{\delta' \subseteq \delta \mid \delta \in \Sigma\}$.

Un schéma de *caching* δ associe une formule $\Sigma \in S(\Sigma_i)$ à une représentation $r_c(\Sigma)$ de Σ qui sera utilisée comme clef (cette représentation peut être de n'importe quelle nature). Un *cache* associe alors à chaque entrée $r_c(\Sigma)$ le nombre de modèles Σ si nous sommes dans le cas du comptage de modèles de Σ_i, ou une formule Decision-DNNF représentant Σ dans le cas de la compilation en Decision-DNNF. La taille du *cache*, étant donné un schéma de *caching* δ, correspond à la somme de la taille des entrées $r_c(\Sigma)$ qui y sont stockées plus la taille des éléments qui y sont associés ($\|\Sigma\|$ (en notation binaire) ou la taille de la représentation en Decision-DNNF associée à $r_c(\Sigma)$).

Une formule CNF $\Sigma_1 \in S(\Sigma_i)$ est considérée comme faisant partie de la structure de *cache* lorsque nous pouvons trouver une entrée $r_c(\Sigma_2)$ du cache telle que nous avons $r_c(\Sigma_1) = r_c(\Sigma_2)$. Un schéma de *caching* δ est correct lorsque pour n'importe quelle formule CNF Σ_i, $\Sigma_1, \Sigma_2 \in S(\Sigma_i)$, si $r_c(\Sigma_1) = r_c(\Sigma_2)$ alors $\Sigma_1 \equiv \Sigma_2$. Dit autrement, quand les représentations de Σ_1 et Σ_2 coïncident, les deux formules CNF Σ_1 et Σ_2 sont équivalentes. Puisque l'égalité des deux représentations de Σ_1 et Σ_2 étant donné un schéma de *caching* δ est syntaxique, cette opération peut être réalisée efficacement, c'est-à-dire en temps linéaire dans la taille de $r_c(\Sigma_1)$ plus la taille de $r_c(\Sigma_2)$. Lorsque le schéma de *caching* est correct, alors cette égalité peut être vue comme une sous-approximation de la relation d'équivalence entre formules CNF. Il est évident, que d'un point de vue théorique, considérer l'équivalence logique entre formules CNF conduirait à reconnaître plus de formules et donc à éviter de nombreux calculs, cependant une telle approche n'est pas envisageable en pratique (il faut conserver à l'esprit que le nombre de formules que nous allons avoir à considérer durant le processus de comptage (ou de compilation) est généralement très important, et puisque vérifier l'équivalence entre deux formules est un problème coNP-complet, il est clair que procéder ainsi serait trop gourmand en temps). Pour les mêmes raisons, il est aussi nécessaire que la fonction de *mapping* r_c soit calculable en temps polynomial (sans cette supposition, il pourrait être envisageable de représenter chaque entrée de manière canonique – en utilisant une représentation en ROBDD par exemple – et donc de réaliser le test d'équivalence, ce qui conduirait nécessairement à une augmentation de la taille des clefs).

Étant donné sa nature syntaxique, les performances d'un schéma de *caching* dépendent seulement des deux facteurs suivants : la taille cumulée des entrées et la qualité de l'approximation de la relation d'équivalence. En effet, la taille cumulée des entrées donne la taille du *cache*, qui est l'ingrédient le plus consommateur en mémoire des compteurs de modèles et des compilateurs en

`Decision-DNNF` fonctionnant en *top-down*. Puisque le temps nécessaire à rechercher ou à ajouter un élément dans la table de *cache* dépend essentiellement de la taille du cache, cette dernière a un impact direct sur l'efficacité en temps de ces compteurs et compilateurs. La qualité de l'approximation de la relation d'équivalence a aussi une grande influence sur les performances du schéma de *caching*. Un schéma de *caching* δ_1 qui permettrait de représenter plus succinctement l'information qu'un autre schéma de *caching* δ_2, mais où l'approximation de la relation d'équivalence de δ_1 serait beaucoup moins fine que celle de δ_2, ne serait pas forcément le meilleur. En effet, le temps additionnel requis par le schéma δ_2 pourrait être contre-balancé par le fait de pouvoir éviter de calculer plusieurs fois le nombre de modèles (ou compiler) de la même formule `CNF` (qui ne pourrait pas être reconnue par le schéma δ_1). Si cela n'était pas le cas, alors nous pourrions envisager que ne pas utiliser de *caching* – le schéma n dans la suite – est la meilleure stratégie.

Présentons à présent certains des schémas de *caching* de l'état de l'art. Nous supposons que les clauses unitaires (même celles obtenues par l'application de `bcp`) ne sont pas représentées dans le *cache* (il ne faut pas les représenter faute de quoi nous ne pouvons rien cacher).

Dans le schéma standard de *caching* s (utilisé dans `Cachet`), toute formule `CNF` Σ rencontrée durant la recherche est représentée comme une chaîne de caractères contenant les identifiants (des entiers) des littéraux des clauses de cette dernière, séparés par des zéros. L'identifiant d'un littéral positif x_i est son indice dans l'énumération $x_1 < x_2 < \ldots < x_{n-1} < x_n$, et l'identifiant d'un littéral négatif $\neg x_i$ est $-i$.

Dans le schéma de *caching* hybride h considéré dans `SharpSAT`, les clauses de la formule initiale Σ_i sont indicées d'une manière arbitraire (l'indice de la première clause de Σ_i est 1, l'indice de la seconde clause est 2, etc.). Toute formule `CNF` Σ rencontrée durant la recherche est représentée par un couple qui consiste en l'ensemble des identifiants de variables de Σ triés dans l'ordre croissant, et de l'ensemble des indices des clauses de Σ_i qui sont « encore en vie » (non satisfaites) et qui contiennent au moins deux littéraux non assignés, triés en l'ordre croissant. Une variante de h, appelée o, consiste à omettre dans l'ensemble des indices les indices des clauses binaires. En effet, comme cela a été démontré par Thurley (2006), cet ensemble est redondant puisqu'il peut être reconstruit simplement en considérant les indices des variables. Cette omission permet de réduire la taille des clefs, et conduit donc à une gestion plus efficace de la structure de *caching*.

Exemple 27

Illustrons ces trois schémas de *caching* sur un exemple simple. Soit $\Sigma_i = (x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$. Soit $\gamma = x_2$, et soit Σ l'unique composante résultante de $\Sigma_{i|\gamma}$ simplifiée en utilisant `bcp`. Nous avons $\Sigma = (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$. Cette formule Σ est représentée par la chaîne de caractères $r_s(\Sigma) = 3, 4, 5, 0, 1, -6, 0$ si le schéma standard de *caching* est utilisé. Elle est représentée par $r_h(\Sigma) = (\{1, 3, 4, 5, 6\}, \{3, 4\})$ en utilisant le schéma de *caching* hybride. Elle est représentée par $r_o(\Sigma) = (\{1, 3, 4, 5, 6\}, \{3\})$ en utilisant la variante du schéma de *caching* hybride où les indices des clauses binaires initiales sont omises.

Afin de réduire la taille des entrées de la structure de *cache*, les indices sont compressés, c'est-à-dire qu'un nombre minimum des bits est utilisé (au lieu d'utiliser une représentation fixe

pour les entiers). Le schéma o où les indices sont compressés est nommé schéma p dans le papier de Thurley (2006) (p pour *packing*).

Il est clair que représenter les clauses de Σ par leurs indices (comme dans les schémas h, o ou p) est plus efficace en terme de taille d'encodage que de les représenter de manière explicite (comme cela est fait dans le schéma s). Cependant, représenter de manière explicite les clauses permet de considérer une approximation de la relation d'équivalence qui est plus fine. Considérons de nouveau l'exemple précédent, et considérons les deux termes $\gamma_1 = \neg x_1 \wedge x_3$ et $\gamma_2 = x_1 \wedge \neg x_3 \wedge x_5$, nous pouvons observer que $\Sigma_{i|\gamma_1}$ est équivalent à $\Sigma_{i|\gamma_2}$ (la `CNF` résultante consiste en l'unique clause $x_2 \vee x_4$). $\Sigma_{i|\gamma_1}$ et $\Sigma_{i|\gamma_2}$ partagent la même r_s représentation : $2, 4, 0$. D'un autre côté, nous avons $r_h(\Sigma_{i|\gamma_1}) = (\{2, 4\}, \{1\})$ tandis que $r_h(\Sigma_{i|\gamma_2}) = (\{2, 4\}, \{2\})$. Ainsi, $\Sigma_{i|\gamma_1}$ n'est pas vu comme étant équivalent à $\Sigma_{i|\gamma_2}$ quand le schéma h est utilisé (le même constat peut être fait pour les schémas o et p).

Nous allons maintenant introduire notre schéma de *caching*, nommé i (i insiste sur le fait que, comparé aux schémas standard, certaines clauses de Σ_i sont gérées implicitement, c'est-à-dire qu'elles n'apparaissent pas dans la composition de la clef).

Avant de présenter notre schéma de *caching*, commençons progressivement en présentant le schéma de *caching* b, lequel est tel que $r_b(\Sigma) = (V, C)$ où V est l'ensemble des indices des variables de $Var(\Sigma)$ triés dans l'ordre croissant et C est l'ensemble des clauses résiduelles de Σ étant donné $\Sigma_{i|\gamma}$ (les clauses qui ne sont pas satisfaites par l'assignation γ). Dans le cadre du schéma b, les clauses de C sont triées dans l'ordre croissant d'abord par rapport à leur taille et ensuite en fonction de l'ordre lexicographique sur les indices des littéraux des clauses tels que $x_1 < \neg x_1 < x_2 < \neg x_2 < \ldots < x_{n-1} < \neg x_{n-1} < x_n < \neg x_n$ quand les variables sont ordonnées comme suit $x_1 < x_2 < \ldots < x_{n-1} < x_n$. Ainsi, les clauses restantes sont représentées d'une manière explicite, comme dans le schéma de *caching* s (i n'utilise pas les indices contrairement aux schémas h, o et p), mais les clauses sont ordonnées de manière telle que deux ensembles de clauses résiduelles, qui diffèrent seulement sur l'ordre des clauses et/ou l'ordre des littéraux dans ces dernières, ont la même représentation C. Comme pour le schéma p, les clauses de C sont représentées de manière succincte en limitant autant que faire se peut le nombre de bits utilisés pour la représentation des littéraux. Une variante de b, notée 2, est obtenue en évitant d'enregistrer les clauses binaires comme cela est fait dans le schéma de *caching* o.

Le schéma i reprend le principe de b mais il diffère quant à la notion de clauses résiduelles à considérer. Avec le schéma de *caching* i, en plus des clauses binaires de Σ_i qui ne sont pas considérées, les clauses de Σ_i qui n'ont pas été réduites par l'affectation courant ne sont pas représentées non plus. Nous considérons aussi une variante i' de i qui considère le même ensemble de clauses résiduelles que i, mais qui représente les clauses par leurs indices.

Exemple 28

Considérons l'exemple précédent, soit Σ l'unique composante résultante de $\Sigma_{i|\neg x_2}$. Nous avons $\Sigma = (x_1 \vee x_4) \wedge (x_3 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_1 \vee \neg x_6)$, qui est représentée par :

- $r_b(\varphi) = (\{1,3,4,5,6\}, \{\{1,4\}, \{1,-6\}, \{3,4\}, \{3,4,5\}\})$,
- $r_2(\varphi) = (\{1,3,4,5,6\}, \{\{1,4\}, \{3,4\}, \{3,4,5\}\})$,
- $r_i(\varphi) = (\{1,3,4,5,6\}, \{\{1,4\}, \{3,4\}\})$.
- $r_{i'}(\varphi) = (\{1,3,4,5,6\}, \{1,2\})$.

Nous avons que $x_3 \vee x_4 \vee x_5$ et $x_1 \vee \neg x_6$ ne sont pas considérées avec les schémas i et i'.

Bien que nous n'enregistrions pas les clauses de Σ_i qui n'ont pas été réduites dans la clef, que les clauses soient stockées de manière implicite ou explicite, **les schémas i et i' sont corrects**. En effet, considérons les deux schémas de *caching* séparément et étudions ce qu'il se passe :

- Afin de voir que le schéma de *caching* i est correct, il est suffisant de prouver que pour n'importe quelle formule `CNF` Σ_i et $\Sigma \in S(\Sigma_i)$, Σ est complètement déterminée par $r_i(\varphi) = (V, C)$ étant donnée Σ_i. La preuve est constructive : elle consiste à considérer toute clause α de Σ_i et de décider en utilisant $r_i(\Sigma)$ si α (ou dans certains cas, une sous-clause de α) appartient à Σ. Le premier cas est quand $Var(\alpha) \cap V = \varnothing$. Dans ce cas, α n'appartient pas à Σ puisque ses variables ne sont pas présentes dans la composante courante. Dans le cas restant, il y a deux possibilités : (1) soit $Var(\alpha) \subseteq V$ ou (2) $V \setminus Var(\alpha) \neq \varnothing$. Dans le cas (1), nous concluons que α appartient à Σ_i (même si elle n'est pas représentée explicitement dans C). Dans le cas (2), il y a encore deux possibilités : s'il n'existe pas de clause α' représentée dans C telle que $Var(\alpha') \subseteq V$ et α' est une sous-clause de α, alors α ne doit pas apparaître dans Σ (α est satisfaite par l'affectation courante), sinon α' appartient à Σ (α ou une autre clause de Σ_i a été réduite pour donner la clause α') ;
- En ce qui concerne le schéma de *caching* i', nous pouvons utiliser la même approche que pour i. La seule différence concerne le cas (2). Dans ce cas, si l'indice de α appartient à la partie C de $r_{i'}(\varphi)$, alors une sous-clause de α contenant seulement ses littéraux dans la partie V appartient à Σ, sinon α n'appartient pas à Σ (α est satisfaite par l'affectation courante).

Afin de mesurer l'impact réel de la structure de *caching* sur la consommation de mémoire, nous nous sommes limités dans le cadre de nos expérimentations au problème de comptage de modèles. Il est clair que ces résultats peuvent être étendus au cadre de la compilation en `Decision-DNNF`. Dans notre évaluation expérimentale, nous avons considéré deux heuristiques de branchement : `VSADS` qui est l'heuristique utilisée par défaut dans `Cachet` et `SharpSAT`, et l'heuristique `DECOMP` que nous avons présentée dans la sous-section 5.3.2. Nous avons aussi considéré l'ensemble des schémas de *caching* présentés précédemment et qui sont résumés dans la table 5.3.

Pour chaque combinaison heuristique de branchement/schéma de *caching*, nous avons évalué les performances de `d4` sur un ensemble de formules `CNF` en considérant le temps nécessaire pour compter le nombre de modèles et la quantité de mémoire consommée (qui est sensiblement équivalente à la taille du *cache*) par le compteur de modèles pour réaliser sa tâche. Nous avons considéré 703 instances `CNF` de la `SAT` *LIBrary*[11], qui se décomposent en 8 catégories : BN

11. www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html

Nom	Explicite / Implicite	Les clauses restantes	Pas les binaires	Pas les réduites
n	-	-	-	-
b	E	✓		
2	E		✓	
i	E			✓
h'	I	✓		
p	I		✓	
i'	I			✓

TABLE 5.3 – Les sept schémas de *caching* que nous avons considérés dans nos expérimentations.

(*Bayesian networks*) (192), BMC (*Bounded Model Checking*) (18), Circuit (41), Configuration (35), Handmade (58), Planning (248), Random (104), Qif (7) (*Quantitative Information Flow analysis - security*). Les expérimentations ont été conduites sur un cluster de processeurs Intel Xeon E5-2643 (3.30 GHz) quad core avec 32 GiB de RAM. L'*hyperthreading* a été désactivé, et le partage de mémoire au sein du même cœur n'a pas été permis. Pour chaque instance, une limite de temps de 3600s et de mémoire de 7.6 GiB a été considérée.

La table 5.4 reporte pour chaque combinaison heuristique de branchement/schéma de *caching* le nombre d'instances résolues x (sur les 703) et le nombre dépassements de la capacité en mémoire y sous une expression de la forme $x(y)$. La colonne la plus à gauche (n) correspond au cas où le *caching* a été désactivé. Les trois colonnes qui suivent (b, 2, i) correspondent aux schémas de *caching* pour lesquels les clauses résiduelles sont représentées par leurs littéraux. Les trois dernières colonnes (h', p, i') correspondent respectivement à b, 2, i vis-à-vis de la manière dont les clauses sont représentées, mais pour ces trois schémas, les clauses sont représentées de manière implicite par leurs indices.

Heuristique	n	b	2	i	h'	p	i'
`DECOMP`	521 (0)	561 (122)	574 (91)	581 (83)	568 (115)	576 (68)	577 (50)
`VSADS`	485 (0)	511 (182)	517 (172)	521 (174)	516 (176)	523 (145)	523 (130)

TABLE 5.4 – Nombre d'instances résolues par `d4` et nombre de *memory-out* atteints pour plusieurs combinaisons d'heuristique de branchement/schéma de *caching*.

Comme nous pouvons l'observer, la meilleure combinaison est `DECOMP`+i. Commençons tout d'abord par analyser les résultats quand l'heuristique `DECOMP` est considérée. Nous pouvons observer que la taille réduite de la clef générée par i (lorsque nous comparons à b et 2) permet de résoudre plus d'instances et conduit à réduire le nombre de dépassements de la capacité de mémoire. Une observation similaire peut être faite lorsque nous comparons i' avec h' et p. Ces trois schémas sont, comme nous nous en doutions, moins gourmands en taille de *cache* que leur version respective qui considère une représentation explicite. En ce qui concerne les résultats avec l'heuristique `VSADS`, les performances relatives de b, 2, i d'un côté, et de h', p, i' de l'autre, suivent un modèle similaire à celui observé dans le cas de l'utilisation de l'heuristique `DECOMP`. L'exception est que le nombre de dépassements de la capacité mémoire pour 2 est légèrement inférieur (deux instances) à celui du schéma i. Cela peut s'expliquer par le fait que l'espace de recherche parcouru par `d4` lorsque 2 est utilisé est plus petit que celui parcouru par `d4` equipé du schéma i. En effet, `d4` équipé avec 2 a été stoppé par la limite de temps sur ces deux instances et le fait de poursuivre le processus de comptage aurait indubitablement conduit à un dépassement

de la capacité en mémoire. Lorsque l'heuristique `VSADS` est considérée, les schémas de *caching* h', p, i' basés sur une représentation implicite des clauses apparaissent plus intéressants que ceux basés sur une représentation explicite de ces dernières, et i' est pour cette heuristique le meilleur schéma de *caching*. Cela contraste avec ce qui se passe dans le cas où l'heuristique `DECOMP` est considérée. Cela peut s'expliquer par le fait que l'utilisation de `VSADS` conduit à gérer des sous-formules résiduelles plus grosses que celles générées *via* l'utilisation d'une politique agressive de décomposition telle que `DECOMP`. Ainsi, dans cette situation il apparaît plus intéressant de réduire au maximum la taille de la clef au détriment de l'efficacité de l'approximation de la relation d'équivalence. Finalement, quelle que soit l'heuristique utilisée, nous pouvons observer que le *caching* est utile pour le comptage de modèles (dans les deux cas, utiliser n conduit à résoudre moins d'instances).

5.3.4 Discussion

Dans cette section, nous avons décrit `d4` qui est un compilateur de l'état de l'art s'appuyant sur le parcours d'un arbre de recherche et qui cible le langage `Decision-DNNF`. Dans `d4`, nous avons utilisé une heuristique dynamique qui a pour objectif premier de décomposer en composantes disjointes la formule `CNF` donnée en entrée. Cette décomposition est basée sur l'utilisation parcimonieuse d'un outil de partitionnement d'hypergraphe. Afin de réduire le temps de `CPU` alloué au partitionnement, nous avons aussi proposé plusieurs règles de simplification afin de minimiser la taille de l'hypergraphe tout en assurant que les décompositions calculées sont correctes et de bonne qualité. Les expérimentations que nous avons conduites dans (Lagniez et Marquis 2017a) montrent clairement que l'utilisation de cette heuristique permet à `d4` d'être plus efficace en temps et de produire des formules `Decision-DNNF` de plus petites tailles.

Nous avons aussi présenté dans cette section un schéma de *caching* i qui peut être exploité pour le comptage de modèles et la compilation en `Decision-DNNF`. Ce schéma consiste à enregistrer pour chaque entrée (représentant une formule `CNF` formée à partir d'une composante connexe donnée par une assignation γ, et de son nombre de modèles ou d'une représentation en `Decision-DNNF`) l'ensemble de variables correspondantes et un ensemble de clauses représentées de manière explicite, exceptés celles qui sont satisfaites ou qui n'ont pas été réduites par l'affectation γ. Nous avons aussi proposé une variante de i, nommée i', qui au lieu de représenter explicitement les clauses, les représente implicitement par leurs indices. Nous avons montré que ces deux schémas de *caching* sont corrects, dans le sens où ils conduisent à des approximations par défaut de la relation d'équivalence entre formules `CNF`.

Ces travaux ouvrent de nombreuses perspectives pour de futures recherches. Les résultats que nous avons obtenus grâce à l'utilisation de notre heuristique de décomposition, nous font penser qu'il serait avantageux d'aller plus loin dans l'exploitation de la sémantique pour le calcul des *cutset*. En effet, ce qui fait la puissance de cette heuristique, si nous la comparons avec l'heuristique de `C2D`, est que nous considérons l'état de la formule après l'application de `bcp`. Une manière d'aller plus loin dans cette direction pourrait être de rechercher des nœuds $\wedge$ décomposables de manière sémantique. Pour cela, nous pourrions nous inspirer des travaux de Chen et Marques-Silva (2012) sur la recherche de portes logiques décomposables. Bien que l'approche proposée par les auteurs s'appuie sur le calcul de `MUSes`, la détection en haut dans l'arbre de recherche de telles portes pourrait sensiblement réduire la taille de la `Decision-DNNF` produite. Une autre manière d'ajouter un peu de sémantique dans la recherche d'un *cutset*, sans aller jusqu'à appeler un extracteur de `MUSes`, pourrait être de considérer d'autres représentations

de la formule CNF en hypergraphe. Pour cela, nous pourrions nous inspirer des travaux de Heule et Kullmann (2006) où cela est étudié pour la résolution du problème SAT.

Une autre piste, pour incorporer un peu de sémantique dans la recherche de *cutset*, pourrait être de regrouper certaines variables en *clusters* quand elles ont un lien sémantique. Par exemple, nous pourrions rechercher des contraintes de domaines impliquées par la formule et regrouper l'ensemble des variables qui les composent. Cela a du sens, car de nombreux problèmes modélisent des réseaux de contraintes où les variables ne sont pas binaires. Nous avons déjà démontré expérimentalement dans (Koriche et al. 2015), que beaucoup d'informations sur la structure d'un réseau de contraintes étaient perdues lors de son encodage en CNF et que celles-ci sont difficilement récupérables lors du processus de compilation en Decision-DNNF. Ainsi, nous pourrions espérer récupérer ces informations tout en réduisant la taille du graphe à décomposer. En ce qui concerne le branchement sur ces « super » variables, il suffirait de considérer une DNF orthogonale construite sur ces variables qui est équivalente à la contrainte impliquée. Il suffirait, dans le cas de la détection de contraintes de domaines, de considérer les n interprétations possibles de la contrainte.

En ce qui concerne le *caching*, nous pourrions envisager de combiner les différents schémas, c'est-à-dire de combiner les schémas pour lesquels les représentations implicite et explicite peuvent être utilisées. Le choix entre les deux représentations pourrait être réalisé en fonction de la taille des clauses considérées et elle pourrait changer durant la recherche (en fonction par exemple du nombre de fois qu'une formule a été retrouvée dans le *cache*).

Une autre manière de choisir entre les différents schémas de *caching* consisterait à s'inspirer des travaux de Balafrej et al. (2015) qui utilisent un bandit manchot pour la sélection du propagateur utilisé pour réaliser le filtrage dans un solveur CP. Dans notre cas, nous pourrions utiliser le fait que nous avons à parcourir l'intégralité de l'espace de recherche afin d'apprendre quel schéma de *caching* est le mieux adapté. Ici, la fonction de récompense pourrait être calculée en fonction du nombre de formules équivalentes détectées par rapport à la taille des entrées du *cache* (ce qui représente l'effort nécessaire pour utiliser le schéma de *caching* associé).

Dans la section suivante, nous montrons comment il est possible d'utiliser d4 afin de réaliser un comptage de modèles projetés.

5.4 Un algorithme récursif pour le comptage de modèles projetés

Dans cette section, nous présentons nos travaux au sujet du comptage de modèles d'une formule CNF où certaines variables sont oubliées. Ces travaux ont été réalisés en collaboration avec Pierre Marquis et ont été publiés dans (Lagniez et Marquis 2019a). Étant donné une formule Σ et un ensemble de variables X à oublier, nous cherchons à calculer le nombre d'assignations des variables de Σ privées de X qui peuvent être étendues en une interprétation de Σ. Dit autrement, notre objectif est compter le nombre de modèles de la formule booléenne quantifiée $\exists X.\Sigma$ sur son ensemble de variables (c'est-à-dire les variables de Σ privées de X). Le comptage de modèles projetés est un problème central en intelligence artificielle (par exemple en planification quand l'objectif est d'évaluer la robustesse d'un plan donné en fonction des états initiaux pour lesquels le plan permet d'atteindre l'état but (Aziz et al. 2015)), mais aussi en dehors de l'intelligence artificielle (spécifiquement dans le cadre de certains problèmes de vérification formelle (Klebanov et al. 2013)).

Puisqu'il généralise le problème de comptage de modèles standard (qui correspond au cas où $X = \varnothing$), le comptage de modèles projetés est au moins aussi difficile que ce dernier (#P-difficile).

Néanmoins, la présence de variables X à oublier peut rendre le problème plus facile dans certains cas (si toutes les variables de Σ sont considérées dans X, alors le problème revient à décider si la satisfaisabilité). Cependant, une méthode naïve qui consisterait à oublier toutes les variables et à calculer le nombre de modèles de la formule résultante, serait en général non applicable en pratique (en particulier lorsque $|X|$ est élevé). En effet, oublier les variables de X dans Σ de manière systématique conduit souvent à générer une formule d'une taille largement supérieure à celle de Σ (dans le pire des cas elle sera exponentiellement plus grande).

Plusieurs algorithmes ont été proposés par Aziz et al. (2015), afin de déterminer le nombre de modèles d'une formule où certaines variables ont été oubliées. Plus précisément, trois approches, nommées dSharp$_\mathsf{P}$, #clasp, et d2c, peuvent être utilisées. Bien qu'ayant le même objectif en ligne de mire, ces trois approches s'appuient sur des schémas algorithmiques totalement différents :

— dSharp$_\mathsf{P}$ est une adaptation du compteur de modèles dSharp (Muise et al. 2012), lequel calcule une représentation en `Decision-DNNF` de la formule d'entrée Σ afin d'en déterminer le nombre de modèles $\|\Sigma\|$. Cependant, dSharp$_\mathsf{P}$ considère une autre entrée P représentant un ensemble de variables protégées (c'est-à-dire les variables de Σ qui ne seront pas oubliées). L'espace de recherche exploré par dSharp est alors restreint d'une manière à ce que les variables de décision choisies par le compilateur soient prises en priorité dans P. Lorsqu'il n'y a plus de variables de la formule courante disponibles dans P (c'est-à-dire que cette formule contient uniquement des variables de X), une solveur `SAT` est utilisé pour déterminer si la formule résultante est satisfaisable ou pas. Si cela est le cas, alors son nombre de modèles est 1, sinon son nombre de modèles est 0. Cette technique a aussi été utilisée dans (Klebanov et al. 2013). Notons que la contrainte imposée sur l'ordre des variables impacte énormément l'heuristique de choix de variables du compilateur, et rend très difficile la détection de *cutset* de petites tailles. Elle peut avoir un impact très négatif sur le nombre de composantes connexes disjointes de Σ pouvant être détectées par dSharp ;

— #clasp est une extension de clasp, un algorithme d'énumération de modèles projetés sur un ensemble de variables (Gebser et al. 2009), qui consiste grosso modo à calculer une `DNF` orthogonale qui représente $\exists X.\Sigma$. Chaque fois qu'un impliquant de $\exists X.\Sigma$ est trouvé, une clause équivalente à la négation de l'impliquant est ajoutée à Σ afin d'éviter de reconsidérer ce dernier par la suite. Dans #clasp, lorsqu'un impliquant de $\exists X.\Sigma$ est trouvé, un impliquant premier est d'abord calculé de manière gloutonne (ce qui rappelle les approches proposées dans (Schrag 1996, Castell 1996)). Comparé à clasp, cette étape de réduction de l'impliquant permet souvent de réduire le nombre de clauses nécessaires à ajouter pour obtenir le résultat, ce qui implique que de manière générale #clasp est meilleur que clasp ;

— d2c s'appuie sur la construction d'une `Decision-DNNF` de Σ afin d'oublier les variables de X, en fait il est connu (Darwiche et Marquis 2002) que l'oubli peut être réalisé en temps polynomial sur les `DNNF` (les `Decision-DNNF` sont des `DNNF` particulières). Ensuite, la `DNNF` résultante est transformée en `CNF`. Cela peut être réalisé en espace et temps linéaire en ajoutant des variables, tout en conservant le nombre de modèles, *via* l'encodage de Tseitin (1968). Finallement, le nombre de modèles de la `CNF` (qui coïncide maintenant avec la formule initiale où les variables de X ont été oubliées) est calculé en utilisant le compteur de modèles `SharpSAT` (Thurley 2006).

Contrairement à ces algorithmes, notre approche pour compter le nombre de modèles d'une formule où les variables de X sont oubliées, appelée projMC, est basée sur un algorithme récursif exploitant une décomposition disjonctive de $\exists X.\Sigma$ afin de calculer $\|\exists X.\Sigma\|$. Plus précisément, $\exists X.\Sigma$ est découpé en un ensemble équivalent (interprété disjonctivement) de formules qui sont deux à deux incohérentes, telles que $\|\exists X.\Sigma\|$ peut être obtenu en sommant le nombre de modèles

de ces formules où les variables de X sont oubliées. projMC exploite aussi la décomposition en composantes disjointes afin de réduire les temps de calcul.

5.4.1 projMC : un nouvel algorithme pour le comptage de modèles projetés

projMC calcule $\|\exists X.\Sigma\|$ où Σ est une formule CNF [12] et X un ensemble de variables propositionnelles. Contrairement aux approches présentées précédemment pour réaliser cette opération, projMC est un algorithme récursif guidé par une disjonction déterministe construite sur les variables de $Var(\Sigma) \setminus X$ et représentant la formule Σ. Cette disjonction peut être vue comme un ensemble de formules $\{\varphi_1, \ldots, \varphi_{k+1}\}$ (interprétées disjonctivement) sur $Var(\Sigma)$ qui satisfait les conditions suivantes :

— $\forall i \in \{1, \ldots, k+1\},\ Var(\varphi_i) \cap X = \varnothing$;

— $\forall i, j \in \{1, \ldots, k+1\}$, si $i \neq j$ alors $\varphi_i \wedge \varphi_j \models \bot$;

— $\bigvee_{i=1}^{k+1} \varphi_i \equiv \top$.

Puisque $\bigvee_{i=1}^{k+1} \varphi_i$ est valide, nous avons $\exists X.\Sigma \equiv (\exists X.\Sigma) \wedge (\bigvee_{i=1}^{k+1} \varphi_i) \equiv \bigvee_{i=1}^{k+1} (\exists X.\Sigma) \wedge \varphi_i \equiv \bigvee_{i=1}^{k+1} \exists X.(\Sigma \wedge \varphi_i)$ puisqu'il n'y a pas d'éléments de la disjonction déterministe de Σ qui contiennent des variables de X. De plus, puisque les éléments de la disjonction sont deux à deux conflictuels, nous avons que $\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|\exists X.(\Sigma \wedge \varphi_i)\|$. Nous notons par la suite $\{\Sigma \wedge \varphi_i \mid i \in \{1, \ldots, k+1\}\}$ l'ensemble des formules interprétées disjonctivement de la conjonction de Σ avec chaque élément de $\{\varphi_1, \ldots, \varphi_{k+1}\}$. Il est clair qu'une telle décomposition est utile seulement dans le cas où les formules $\Sigma \wedge \varphi_i$ peuvent être traitées plus efficacement que Σ. Pour s'assurer de cela et pour garantir la terminaison de projMC, nous ne calculons pas directement une telle disjonction déterministe Σ, mais nous la construisons à partir d'un modèle ω de Σ (s'il n'existe pas de tel modèle, alors $\exists X.\Sigma$ n'est pas satisfaisable et donc $\|\exists X.\Sigma\| = 0$). Plus précisément, nous commençons par considérer la formule CNF $\Sigma_{|\omega[X]} = \bigwedge_{i=1}^{k} \delta_i$, appelée *core* [13] φ_1 de la disjonction déterministe $\{\varphi_1, \ldots, \varphi_{k+1}\}$ de Σ induite par ω. Par construction, $\varphi_1 = \bigwedge_{i=1}^{k} \delta_i$ ne contient que des variables de $Var(\Sigma)$ qui n'apparaissent pas dans X. Ensuite nous générons l'ensemble (interprété disjonctivement) de formules CNF $\{\varphi_2, \ldots, \varphi_{k+1}\}$ qui est équivalent à la négation de φ_1 en définissant pour chaque $i \in \{1, \ldots, k\}$, $\varphi_{i+1} = (\bigwedge_{j=1}^{i-1} \delta_j) \wedge {\sim}\delta_i$. Par construction, les éléments de $\{\varphi_1, \ldots, \varphi_{k+1}\}$ satisfont bien la condition de déterminisme. Notons que $\exists X.(\Sigma \wedge \varphi_1)$ est équivalent à φ_1. En effet, $\exists X.(\Sigma \wedge \varphi_1)$ est équivalent à $(\exists X.\Sigma) \wedge \varphi_1$ puisque φ_1 ne contient pas de variables de X. De plus, puisque $\omega[X] \wedge \Sigma \models \Sigma$, nous avons que $\exists X.(\omega[X] \wedge \Sigma) \models \exists X.\Sigma$. Mais $\exists X.(\omega[X] \wedge \Sigma)$ est équivalent à $\Sigma_{|\omega[X]}$, et donc à φ_1, ce qui implique que $\varphi_1 \models \exists X.\Sigma$, et ce qui conduit au fait que $(\exists X.\Sigma) \wedge \varphi_1$ est équivalent à φ_1. Par conséquent, lorsque nous calculons le nombre de modèles de la disjonction déterministe $\{\Sigma \wedge \varphi_i \mid i \in \{1, \ldots, k+1\}\}$ associée à $\{\varphi_1, \ldots, \varphi_{k+1}\}$, nous pouvons remplacer $\Sigma \wedge \varphi_1$ par φ_1. Puisque $Var(\varphi_1) \cap X = \varnothing$, nous pouvons alors utiliser un compteur de modèles « standard » pour calculer $\|\varphi_1\|$ et dans ce cas il n'y a pas besoin d'appeler récursivement projMC (cela définit le cas de base de la récursion).

projMC tire aussi avantage de la décomposition en composantes disjointes pouvant être induite par l'assignation de certaines variables de Σ. En effet, lorsque Σ peut être séparée en deux ensembles de clauses disjoints Δ et Γ qui ne partagent pas de variables tels que $\Sigma \equiv \Delta \wedge \Gamma$, alors

12. Supposer que Σ est en CNF n'est pas indispensable car il est toujours possible d'associer à n'importe quelle formule de la logique propositionnelle une formule qui possède exactement le même nombre de modèle en utilisant l'encodage de Tseitin (1968).

13. Attention, ici la notion de *core* est différente de celle définie à la section 4.1, et qui a trait à l'identification de l'incohérence dans les formules CNF.

le calcul de $\exists X.\Sigma$ peut être réalisé en calculant $\exists X.\Delta$ et $\exists X.\Gamma$ puisque $\exists X.\Sigma \equiv \exists X.(\Delta \wedge \Gamma) \equiv (\exists X.\Delta) \wedge (\exists X.\Gamma)$. Et puisque $\exists X.\Delta$ et $\exists X.\Gamma$ ne partagent pas de variables, nous pouvons calculer $\|\exists X.\Sigma\| = \|\exists X.\Delta\| \times \|\exists X.\Gamma\|$. Comme cela est le cas dans le cadre du comptage de modèles, l'utilisation de la décomposition permet d'obtenir un gain significatif en terme de temps de calcul.

Algorithme 5.15 : projMC (Σ, X)

input : Σ une formule CNF, un ensemble de variables X à oublier
output : le nombre de modèles de $\exists X.\Sigma$ sur $Var(\Sigma) \setminus X$

```
1  Σ ← bcp(Σ) ;
2  if Var(Σ) ∩ X = ∅ then returnMC(Σ);
3  if cache(Σ) ≠ nil then return cache(Σ);
4  comps ← connectedComponents(Σ);
5  if #(comps) ≠ 1 then
6  |   cpt ← 1 ;
7  |   foreach φ ∈ comps do
8  |   |   cpt ← cpt× projMC (φ, X) ;
9  else
10 |   ω ← sat(Σ) ;
11 |   if ω = ∅ then return 0;
12 |   dd ← DD(Σ, ω, X);
13 |   cpt ← 0 ;
14 |   foreach φ ∈ dd do
15 |   |   cpt ← cpt+projMC (φ, X) ×2^#(Var(dd)\(Var(φ)∪X)) ;
16 cache(Σ) ← cpt ;
17 return cpt ;
```

L'algorithme 5.15 donne le pseudo-code de projMC. Le nombre de modèles d'une formule CNF Σ où les variables de X doivent être oubliées est obtenu en appelant la fonction projMC (Σ, X). À la ligne 1, Σ est simplifiée en appliquant la propagation unitaire. À la ligne 2, nous considérons le cas où Σ ne contient pas de variables de X. Dans ce cas, il n'y a pas de variables à oublier et il est suffisant de calculer le nombre de modèles de Σ en utilisant n'importe quel compteur de modèles MC. C'est le cas de base de notre algorithme (il n'y a pas d'appels récursifs). À la ligne 3, nous recherchons dans la structure de *cache* si la formule courante Σ a déjà été rencontrée précédemment (voir la sous-section 5.3.3 pour plus de détail sur le *caching*). Le cache, initialisé à vide, stocke des couples de formules CNF avec le nombre de modèles obtenu en projetant les variables de X sur cette dernière. Chaque fois que Σ est détectée comme faisant partie du *cache*, au lieu de re-calculer $\|\exists X.\Sigma\|$, il est suffisant de considérer cache(Σ). Dans notre implémentation, nous ne réinitialisons pas le *cache* et il est partagé avec le compteur de modèles MC. À la ligne 4, la formule Σ est partitionnée en un ensemble de formules CNF qui sont indépendantes (c'est-à-dire construites sur des ensembles de variables disjoints). connectedComponents est une procédure standard utilisée dans tous les compteurs de modèles. Elle est basée sur une recherche de composantes connexes dans le graphe primal de Σ et elle retourne un ensemble *comps* de formules CNF composées de clauses de Σ, telles que toute paire de formules distinctes de *comps* n'ont pas de variables en commun. À la ligne 5, nous regardons combien de composantes connexes ont été identifiées dans Σ. S'il y a plus d'une composante connexe, alors à la ligne 6, 7, et 8, nous calculons récursivement le nombre de modèles de chaque composante où les variables de X

seront oubliées, et nous enregistrons le produit dans la variable *cpt* qui correspond au nombre de modèles recherché. Pour des raisons d'efficacité, dans notre implémentation, nous utilisons une amélioration que nous n'avons pas décrite dans l'algorithme pour des raisons de lisibilité : nous groupons systématiquement tous les clauses unitaires dans une même composante et nous assignons directement 1 comme nombre de modèles à cette dernière. Par exemple, si la formule Σ est égale à $x_1 \wedge \neg x_2 \wedge (x_3 \vee x_4)$, nous combinons les deux composantes connexes x_1 et $\neg x_2$ pour ne conserver que la composante $x_3 \vee x_4$. En effet, pour la composante qui regroupe les clauses unitaires $\{x_1, \neg x_2\}$, il est facile de voir que le nombre de modèles lorsque les variables de X sont projetées est toujours égal à 1 (et cela quelles que soient les variables apparaissant dans X). Ainsi projMC retourne directement 1 dans ce cas, sans avoir besoin de considérer les littéraux indépendamment. À la ligne 9, nous considérons le cas où il n'y a qu'une seule composante connexe (c'est-à-dire qu'il n'y a pas de décomposition). À la ligne 10, nous recherchons un modèle ω de Σ sur $Var(\Sigma)$. Si un tel modèle n'existe pas ($\omega = \varnothing$), alors Σ est insatisfaisable, et par la même occasion $\exists X.\Sigma$, et l'algorithme retourne 0 (ligne 11). L'heuristique utilisée par le solveur `SAT` essaie de satisfaire Σ en assignant en priorité les variables de X. Une conséquence intéressante de cette heuristique est qu'elle conduit généralement à ne pas considérer dans la suite de la recherche les clauses de Σ contenant un littéral pur ℓ tel que $var(\ell) \in X$. En effet, lorsque ℓ est pur dans Σ, ℓ sera assigné à 1 par ω et donc aucune clause de Σ contenant ℓ ne sera présente dans $\Sigma_{|\omega[X]}$. Cela n'impacte pas la correction de l'approche puisque de telle clauses peuvent être supprimées de Σ sans changer le nombre de modèles de la formule projetée sur X. Ainsi, tandis que la règle « classique » des littéraux purs (c'est-à-dire la règle consistant à supprimer de Σ toutes les clauses contenant une variable pure dans Σ) ne peut pas être appliquée (puisqu'elle ne préserve pas le nombre de modèles dans le cas général), sa restriction où seulement les littéraux purs sur X sont considérés est correcte : soit Δ (resp. Γ) le sous-ensemble des clauses de Σ contenant ℓ (resp. ne contenant pas ℓ). Nous avons $\exists X.\Sigma \equiv \exists X.(\exists\{x\}.(\Delta \wedge \Gamma))$. Quand ℓ est pur dans Σ, x n'apparaît pas dans Γ. Ainsi $\exists\{x\}.(\Delta \wedge \Gamma) \equiv (\exists\{x\}.\Delta) \wedge \Gamma$. Maintenant, puisque toutes les clauses de Δ contiennent ℓ, $\exists\{x\}.\Delta$ est valide, donc $\exists X.\Sigma \equiv \exists X.\Gamma$. À la ligne 12, nous calculons la décomposition disjonctive *dd* associée à la disjonction déterministe de Σ induite par ω. À la ligne 13, 14, et 15, nous additionnons le nombre de modèles des formules de *dd* projetées sur X (notons que quand les éléments φ de *dd* ne contiennent pas le même ensemble de variables que la formule Σ, une étape de normalisation est réalisée – nous multiplions tous les termes de l'addition par 2 à la puissance le nombre de variables de *dd* qui n'apparaissent ni dans φ ni dans X). Finalement, à la ligne 16, nous ajoutons Σ dans le *cache*, en associant la formule correspondante au nombre *cpt* de modèles que nous avons déterminé, et nous retournons *cpt*.

Algorithme 5.16 : $\mathsf{DD}(\Sigma, \omega, X)$

input : Σ une formule `CNF`, ω un modèle de Σ, X les variables à oublier
output : *dd* une disjonction déterministe de Σ induite par ω

1 $\varphi \leftarrow \mathsf{bcp}(\Sigma_{|\omega[X]})$;
2 $dd \leftarrow \{\varphi\}$;
3 foreach $\delta_j \in \varphi = \bigwedge_{i=1}^{k} \delta_i$ do
4 $\quad dd \leftarrow dd \cup \{\Sigma \wedge (\bigwedge_{l=1}^{j-1} \delta_l) \wedge \sim\delta_j\}$;
5 return *dd*;

L'algorithme 5.16 présente notre approche afin de générer une disjonction déterministe de Σ induite par ω de Σ utilisée pour guider la recherche. À la ligne 1, nous conditionnons la

formule Σ par le terme $\omega[X]$, c'est-à-dire que toutes les clauses contenant un littéral de $\omega[X]$ sont supprimées de Σ, et dans toutes les clauses restantes, nous supprimons les littéraux ℓ de X tels que $\sim\ell$ est un littéral de $\omega[X]$, et finalement nous appliquons la propagation unitaire sur la formule résultante afin de réduire au maximum sa taille. Il est clair, par construction, qu'aucune variable de X n'apparaît dans la formule résultante φ, qui est le *core* de la disjonction déterministe *dd* que nous sommes en train de construire. À la ligne 2, nous commençons par ajouter dans *dd* le *core* φ calculé juste avant. Aux lignes 3 et 4, nous ajoutons à *dd* k formules `CNF` lorsque $\varphi = \bigwedge_{i=1}^{k} \delta_i$ contient k clauses. À la ligne 4, les clauses de Σ qui sont subsumées par les clauses δ_l sont supprimées (cela n'est pas détaillé dans le pseudo-code afin de ne pas rendre sa lecture trop difficile). Par construction, les formules de *dd* sont deux à deux incohérentes. De plus, tout δ_j du *core* φ est une sous-clause d'une clause de Σ puisque φ est obtenue en conditionnant Σ avec un terme cohérent. Ainsi, dans notre implémentation, lorsque δ_j est égal à une clause de Σ, δ_j n'est pas pris en considération dans la boucle (en effet, dans ce cas, $\Sigma \wedge \delta_j$ est équivalent à Σ et $\Sigma \wedge \sim\delta_j$ est incohérent, et donc sa contribution au nombre de modèles est nulle). Finalement, à la ligne 4, nous retournons la disjonction déterministe *dd* que nous avons calculée.

Exemple 29

Voici un exemple illustrant comment projMC fonctionne. Soient $\Sigma = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_5)$ une formule `CNF` et $X = \{x_2, x_3\}$ l'ensemble des variables à oublier. La formule $\exists X.\Sigma$ est équivalente à $x_1 \vee x_4 \vee x_5$, et donc $\|\exists X.\Sigma\| = 7$.
Lorsque nous appelons projMC avec comme paramètres Σ et X, la première instruction n'a pas d'effet ($\texttt{bcp}(\Sigma) = \Sigma$). Ensuite, puisque le graphe primal de Σ n'a qu'une composante, nous considérons directement la partie `else` à la ligne 9. Supposons que le modèle ω de Σ retourné à la ligne 10 assigne à 1 toutes les variables. Nous appelons alors la fonction DD (ligne 12), qui commence à ajouter le *core* $\Sigma_{|\omega[X]} = x_5$ à *dd* (ligne 2 de l'algorithme 5.16). Ensuite, nous ajoutons une seule formule $\Sigma \wedge \neg x_5$ à *dd* dans la boucle de l'algorithme 5.16.
L'appel récursif à projMC sur le *core* x_5 et X conduit à appeler le compteur de modèles `MC` sur x_5 (aucune variable de X n'apparaît dans le *core*). Alors `MC` retourne 1, et finalement la contribution de cet appel récursif au nombre de modèles est $1 \times 2^2 = 4$ puisque les deux variables x_1 et x_4 appartiennent à l'ensemble des variables de dd mais ne sont pas présentes dans l'ensemble de variables du *core* (cette normalisation est réalisée à la ligne 15).
L'appel récursif de projMC sur $\Sigma \wedge \neg x_5$ et X conduit dans un premier temps à simplifier $\Sigma \wedge \neg x_5$ avec bcp, nous obtenons donc $\neg x_5 \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_3$. Puisque les deux composantes $\neg x_5$ et $\neg x_3$ forment un terme cohérent, nous ne considérons qu'une seule composante qui est $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4)$. Supposons encore que le modèle ω de Σ assigne à 1 toutes les variables. Alors le *core* résultant est x_4, et le résultat de l'appel de la procédure DD est $\{x_4, (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_4\}$. Puisque x_4 ne contient pas de variables de X, au prochain appel récursif de projMC le résultat de `MC` sur cette formule est 1, et puisque seule la variable x_1 appartient à dd mais pas à X, la normalisation donnera $1 \times 2^1 = 2$ comme contribution de cette formule au nombre de modèles. L'appel récursif de projMC sur $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_4$ et X conduit dans un premier temps a simplifier (avec bcp) la formule d'entrée en $x_1 \wedge \neg x_2 \wedge \neg x_4$, qui est un terme cohérent et son nombre de modèles est 1. Nous obtenons ainsi que $\|\exists X.((x_1 \vee x_2) \wedge (\neg x_2 \vee x_4))\| = 2 + 1 = 3$. Finalement en additionnant les différents retours de projMC nous obtenons $4 + 3 = 7$ modèles, qui est le nombre de modèles attendu.

Il est intéressant de noter que projMC fonctionne de manière adéquate lorsque $X = \varnothing$ or $X = Var(\Sigma)$, dans le sens où il ne réalisera pas d'appels récursifs superflus dans ces deux cas extrêmes. En effet, lorsque $X = \varnothing$, projMC appelle directement un compteur de modèles standard à la ligne 2. Quand $X = Var(\Sigma)$, si Σ est insatisfaisable, alors cela sera détecté lors du premier appel à projMC (ligne 11) et le nombre de modèles retourné sera 0. Dans le cas restant, un modèle ω de Σ sera trouvé. Mais puisque $\omega[X]$ coïncide avec ω lorsque $X = Var(\Sigma)$, le *core* de la disjonction déterministe de Σ induite par ω sera égal à un ensemble de clauses vide ($\Sigma_{|\omega}$ est valide lorsque ω est un modèle de Σ), donc le résultat retourné par la procédure DD sera cet unique *core*. Puisque ce *core* ne contient pas de variables, le prochain appel à projMC (c'est-à-dire celui avec le *core* comme paramètre) consistera à appeler `MC` (ligne 2) sur ce dernier, et `MC` retournera 1 dans ce cas.

Il est facile de montrer que projMC est correct, c'est-à-dire que le résultat retourné par ce dernier est $\|\exists X.\Sigma\|$ sur les entrées Σ et X, et qu'il termine. Cela vient du fait que chaque règle de l'algorithme est correcte dans le cas du comptage de modèles projetés, comme expliqué précédemment. La clef est que $\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|\exists X.(\Sigma \wedge \varphi_i)\|$ pour n'importe quelle disjonction déterministe $\{\varphi_1, \ldots, \varphi_{k+1}\}$ de Σ par rapport à X, et que $\|\exists X.(\alpha \wedge \beta)\| = \|\exists X.\alpha\| \times \|\exists X.\beta\|$ quand $Var(\alpha) \cap Var(\beta) = \varnothing$. La terminaison de projMC vient du fait que le *core* de la disjonction déterministe de Σ est calculé par rapport à X (nous n'avons plus de variables à oublier), ainsi l'appel récursif à projMC le concernant conduira au cas de base de la récursion (ligne 2). Comme les autres $\Sigma \wedge \varphi_i$ de la disjonction déterministe correspondante sont construits de sorte que φ_i contient la négation d'une sous-clause δ_{i-1} de Σ, au prochain appel à projMC concernant l'une de ces formules, les littéraux du terme $\sim\delta_{i-1}$ sont assignés et les variables correspondantes seront supprimées par propagation unitaire (ligne 1). Ainsi le nombre de variables de l'entrée diminue strictement à chaque appel, ce qui assure la terminaison de l'algorithme. Nous avons donc la proposition suivante :

Proposition 2

L'algorithme 5.15 est correct et termine.

Notons que la détection des composantes disjointes (lignes 4 – 9 de l'algorithme 5.15) peut être désactivée dans projMC sans remettre en question la correction et la terminaison de cet algorithme (cependant, cette détection a un impact significatif sur l'efficacité de l'approche). De manière similaire, l'utilisation du *cache* n'a pas non plus d'impact sur la correction et la terminaison de cet algorithme, ainsi les instructions aux lignes 3 et 16 peuvent aussi être supprimées (néanmoins en pratique l'utilisation du *cache* permet d'améliorer l'efficacité de l'approche en terme de temps de calcul).

5.4.2 Discussion

Dans cette section nous avons proposé projMC, un nouvel algorithme pour le comptage de modèles de $\|\exists X.\Sigma\|$, *i.e.* d'une formule propositionnelle Σ après avoir oublié les variables de l'ensemble X. Contrairement aux approches précédentes de l'état de l'art, projMC tire avantage de la décomposition en disjonction déterministe de $\exists X.\Sigma$ pour calculer $\|\exists X.\Sigma\|$. Il considère aussi la décomposition en composantes disjointes de la formule d'entrée ainsi que l'utilisation d'un

schéma de *caching* afin d'être plus performant en pratique. Les expérimentations que nous avons conduites dans (Lagniez et Marquis 2019a) ont montré que projMC peut être significativement plus efficace que les approches dSharp$_\mathsf{P}$, #clasp, d2c, et d4$_p$ (une version de dSharp$_\mathsf{P}$ construit sur d4). Nous avons aussi démontré empiriquement que projMC était meilleur que d4$_p$ sur de nombreuses instances, montrant ainsi que le gain de performance offert par projMC n'était pas dû uniquement au fait que d4 est en général plus efficace que dSharp.

Il existe de nombreuses pistes afin d'améliorer les performances de projMC. Tout d'abord, il serait envisageable d'utiliser un compilateur `Decision-DNNF` à la place du compteur de modèles `MC` utilisé à la ligne 2. En effet, l'une des particularités de notre approche est que nous connaissons la formule au départ et qu'à chaque appel de projMC, nous considérons systématiquement la conjonction de Σ, d'un terme et de certaines de ses sous-clauses obtenues en supprimant les variables de X. Ainsi, lorsque nous arrivons au cas de base où le compteur de modèles est appelé, c'est-à-dire lorsque la formule Σ considérée ne possède plus de variables de X, il est possible de la retrouver en conditionnant la formule suivante :

$$\Sigma_s = \Big(\bigwedge_{\alpha\in\Sigma | var(\alpha)\cap X=\varnothing} \alpha\Big) \wedge \Big(\bigwedge_{\alpha\in\Sigma | var(\alpha)\cap X\neq\varnothing} \neg s_\alpha \vee \Big(\bigvee_{\ell\in\alpha | var(\ell)\notin X} \ell\Big)\Big)$$

où les s_α sont des variables propositionnelles permettant de sélectionner les sous-clauses que nous souhaitons considérer. Cette formule contient toutes les clauses de Σ ne contenant pas de variables de X, et où l'ensemble des clauses restantes $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ de Σ sont telles que dans chaque α_i nous avons supprimé les littéraux associés aux variables de X que nous avons remplacés par un littéral sélecteur $\neg s_{\alpha_i}$. À partir de Σ_s il est possible de sélectionner les sous-clauses qui nous intéressent en assignant les bons sélecteurs à vrai et en assignant à faux les autres. Par conséquent, en compilant Σ_s en `Decision-DNNF` et en conditionnant correctement la forme compilée, nous pourrions obtenir le nombre de modèles en temps polynomial (voir la section 5.1) au lieu de faire appel à un compteur de modèles. En ce qui concerne la gestion de la décomposition en composante connexes, il est facile de voir qu'il suffit de considérer un modèle de la formule et de conditionner les variables qui ne sont pas dans la composante connexe courante par les valeurs retenues dans le modèle. Il est important de noter qu'une telle approche n'est possible que si nous utilisons la méthode DD pour construire la disjonction déterministe (il est nécessaire de travailler avec des sous-clauses bien définies au départ pour que cela fonctionne).

Un autre point critique de notre approche concerne le nombre d'appels récursifs réalisés. Une manière de les limiter pourrait être de chercher à minimiser le nombre de clauses restantes dans le *core* obtenu à partir de ω, dit autrement, de rechercher un modèle ω de Σ qui maximise le nombre de clauses satisfaites contenant des variables de X. En fait, nous avons déjà expérimenté cela dans projMC et il s'est avéré que le *core* calculé à partir de ω était d'une taille proche de l'optimal. Néanmoins, la taille n'est peut être pas le facteur prépondérant, et il faudrait peut être envisager d'autres mesures afin de déterminer si un *core* est plus intéressant qu'un autre. Par exemple, le fait que les clauses du *core* soient de grandes tailles pourrait être un critère d'évaluation intéressant. En effet, plus la contribution du *core* à la quantité de modèles du problème initial est élevée, plus nous couvrons rapidement les modèles de $\exists X.\Sigma$. De la même manière, essayer de minimiser le nombre de variables du *core* pourrait conduire à paver plus rapidement les modèles de $\exists X.\Sigma$. Finalement, puisque les différents appels récursifs sont indépendants, il est tout à fait envisageable de les réaliser en parallèle. Pour cela, nous pourrions nous appuyer sur nos travaux sur le comptage de modèles sur architectures à mémoire distribuée (Lagniez et al. 2018d).

Une autre perspective consisterait à tirer avantage des deux procédures B et E que nous avons présentées à la section 5.2.2.2. En effet, comme nous l'avons déjà discuté à la section 5.2.3, B et E

pourraient être exploitées comme une méthode de pré-traitement dans projMC. Nous pourrions aussi utiliser la procédure B afin de déterminer les variables X_b de X qui sont définies dans Σ à partir des variables de $Var(\Sigma) \setminus X$ et ainsi changer le test du cas de base en if $(X_b = X)$. En effet, si toutes les variables de X sont définies, il est possible d'utiliser directement un compteur de modèles sur Σ sans se soucier des variables à oublier (nous exploitons ici en fait la proposition 1).

Pour le moment nous n'avons considéré que des approches pour le comptage ou la compilation de connaissances qui s'appuient sur l'utilisation de nœuds de décision « classiques ». Dans la section suivante, nous allons un peu plus loin et nous présentons un nouveau langage de compilation qui s'appuie sur l'utilisation de nœuds de décision affine.

5.5 Les arbres de décision affine

Comme nous avons pu le remarquer en préliminaire, peu de langages cibles offrent la requête de comptage de modèles. Parmi eux figurent principalement le langage `d-DNNF` et ses sous-ensembles, `Decision-DNNF`, $\texttt{OBDD}_<$, mais aussi `DT`, ainsi que le langage représentant l'ensemble des modèles `MODS`. Clairement `MODS` n'est pas utilisable en pratique, il est en effet un langage peu efficace du point de vue de l'efficacité spatiale. Un langage intéressant pour le comptage de modèles est `DT`, mais il est lui aussi peu efficace d'un point de vue spatial. Un autre langage offrant **CT** est le langage `AFF` des formules affines, mais il n'est pas complet pour la logique propositionnelle.

L'idée directrice qui a conduit à la définition du langage `ADT` des arbres de décision affine a été de combiner les principes qui sont à la base des formules affines (l'utilisation de clauses affines) et des arbres de décision (la prise en compte de nœuds de décision organisés en arbre). Pour généraliser les arbres de décision, nous avons étendu les nœuds de décision standard à des nœuds de décision affine. Nous avons aussi proposé le langage `EADT` des arbres de décision affine étendus qui est une extension des `ADT` autorisant la présence de nœuds de conjonction vérifiant une forme forte de décomposabilité.

Commençons par définir un langage très général, le langage `ADN` des réseaux de décision affine, qui va servir de langage de base pour définir la sémantique des formules de ses sous-ensembles qui vont plus particulièrement nous intéresser, `ADT`, `EADT`, et `EDT`. Tout comme `NNF`, `ADN` n'est pas un langage cible intéressant car il ne vérifie pas **CT**. Son introduction sert essentiellement à fixer un cadre et des notions utiles. `ADN` (et ses sous-ensembles `ADT` et `EADT`) se base sur un nouveau type de nœud, les nœuds de décision affine, définis comme suit :

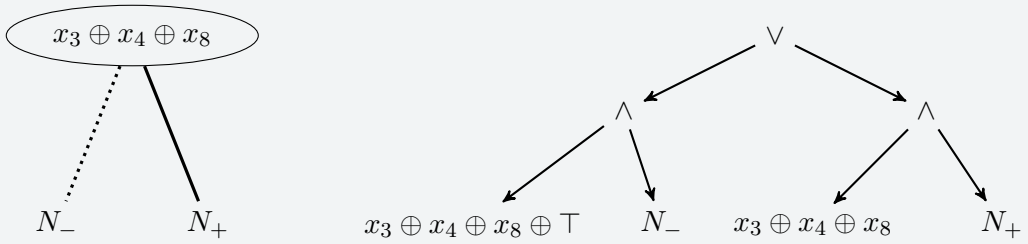
Définition 66 (Nœud de décision affine)

Un nœud de décision affine est un nœud de décision binaire de la forme $N = \langle \delta, N_-, N_+ \rangle$ où δ est la clause affine étiquetant N et N_- (respectivement N_+) est le fils gauche (respectivement droit) de N. N_- représente la branche dans laquelle la clause affine $\delta \oplus \top$ est considérée comme fausse, et N_+ la branche dans laquelle la clause affine δ est considérée comme vraie.

Comme pour les nœuds de décision habituels, un nœud de décision affine correspond à une disjonction déterministe de deux conjonctions.

Exemple 30 (Nœud de décision affine et sa forme propositionnelle)

Soit $N = \langle x_3 \oplus x_4 \oplus x_8, N_-, N_+ \rangle$ un nœud de décision affine. La figure de gauche représente N et la figure de droite représente sa forme propositionnelle.



Un réseau de décision affine est un graphe dont les nœuds internes sont des nœuds de disjonction, de conjonction ou de décision affine. Le langage contenant ces réseaux est défini formellement de la manière suivante :

Définition 67 (ADN)

Un réseau de décision affine (ADN) est un graphe orienté fini sans circuit (DAG) ne comportant qu'une racine, où les feuilles sont étiquetées par $\top$ ou $\bot$, et les nœuds internes sont des nœuds de conjonction ou de disjonction (avec un nombre arbitraire mais fini de fils) ou des nœuds de décision affine.

Afin de comparer les différents langages en terme d'efficacité spatiale, il est nécessaire de définir la notion de taille d'une formule. Alors que la taille d'une formule NNF est donnée par son nombre d'arcs, la taille $|\Delta|$ d'une formule ADN Δ est définie de la manière suivante :

Définition 68 (Taille d'un ADN)

La taille $|\Delta|$ d'une formule ADN Δ est la somme de son nombre d'arcs et de la taille cumulée des clauses affines utilisées pour étiqueter les nœuds de décision affine.

Nous notons R_Δ la racine du graphe représentant une formule ADN Δ. Soit Δ une formule ADN et N un nœud de Δ, nous notons Δ_N la sous-formule de Δ représentée par la formule ADN ayant pour nœud racine N. Pour tout nœud N d'une formule ADN Δ, l'ensemble $Var(N)$ des variables constituant la formule ADN Δ_N enracinée en N, est défini de manière inductive comme :

— si N est une feuille, alors $Var(N) = \varnothing$;

— si N est un nœud de décision affine $N = \langle \delta, N_-, N_+ \rangle$, alors $Var(N) = var(\delta) \cup Var(N_-) \cup Var(N_+)$;

— si N est un nœud $\wedge$ (ou $\vee$) ayant comme fils $N_1, \ldots, N_k$, alors $Var(N) = \cup_{i=1}^{k} Var(N_i)$.

L'ensemble $Var(\Delta)$ des variables d'une formule ADN Δ est ainsi égal à l'ensemble des variables de la racine R_Δ de Δ. $Var(R_\Delta)$ peut être calculé en temps linéaire en la taille de Δ. En effet, ce calcul peut être effectué par un simple parcours du graphe Δ.

Toute formule ADN Δ est interprétée comme une formule propositionnelle $I(\Delta)$ sur les variables $Var(\Delta)$, avec $I(\Delta) = I(R_\Delta)$ définie de manière inductive par :

— si N est une feuille étiquetée par $\top$ (resp. $\bot$), alors $I(N) = \top$ (resp. $\bot$) ;

— si N est un nœud de décision affine $N = \langle \delta, N_-, N_+ \rangle$, $I(N) = ((\delta \oplus \top) \wedge I(N_-)) \vee (\delta \wedge I(N_+))$;

— si N est un nœud $\wedge$ (resp. $\vee$) ayant pour fils $N_1, \ldots, N_k$, alors $I(N) = \bigwedge_{i=1}^{k} I(N_i)$ (resp. $\bigvee_{i=1}^{k} I(N_i)$).

Notons qu'un nœud de décision affine étiqueté par la clause affine $\delta = x_1 \oplus \cdots \oplus x_n$ divise les interprétations possibles d'une formule en deux parties égales. En effet, nous aurons d'un côté un fils dans lequel les seules interprétations possibles seront celles dont un nombre pair de variables de $\{x_1, \ldots, x_n\}$ seront assignées à vrai, et de l'autre coté un fils dans lequel ce seront celles dont un nombre impair de variables de $\{x_1, \ldots, x_n\}$ seront assignées à vrai.

Nous rappelons que le nombre de modèles d'une formule ADN Δ sur $Var(\Delta)$ est noté $\|\Delta\|$. Nous pouvons observer que le langage DAG-NNF est traduisible en temps polynomial en un sous-ensemble de ADN, dans lequel les nœuds de décision affine ne possèdent que des feuilles comme fils. En effet, chaque nœud étiqueté par un littéral positif x (resp. négatif $\neg x$) d'une formule DAG-NNF est équivalent à un nœud de décision affine N étiqueté par x, tel que $N_- = \bot$ (resp. $N_- = \top$) et $N_+ = \top$ (resp. $N_+ = \bot$).

En conséquence, le langage ADN est un langage de représentation très succinct. A contrario, il offre peu de requêtes parmi celles considérées dans la carte de compilation. En effet, comme il est possible de transformer en temps polynomial n'importe quelle formule DAG-NNF en formule ADN, si une requête n'est pas offerte par DAG-NNF, elle ne l'est pas non plus pour ADN. Or, nous avons vu en préliminaire de ce chapitre que DAG-NNF n'offre aucune requête de la carte de compilation. Ainsi, ADN ne satisfait pas le comptage de modèles **CT** (sauf si P = NP) ni même le test de cohérence **CO**. C'est pourquoi nous allons par la suite nous concentrer sur des sous-ensembles du langage ADN.

Comme nous avons pu le remarquer en préliminaire, les langages qui satisfont le plus de requêtes sont souvent les langages qui ont une structure très contrainte. Considérons alors les deux sous-langages de ADN définis de la manière suivante :

Définition 69 (ADG et GEADT)

Une formule ADG est une formule ADN dans laquelle les nœuds internes sont des nœuds de décision affine et des nœuds de conjonction décomposables. Une formule GEADT est une formule ADG dont la forme est arborescente.

La restriction qui consiste à ne considérer dans les formules que des nœuds de décision et des nœuds de conjonction décomposables conduit, quand nous l'appliquons au langage NNF, au langage Decision-DNNF, qui est un langage cible intéressant. Cependant comme nous l'avons montré dans (Koriche et al. 2013), elle ne suffit pas à définir un langage cible qui a de l'intérêt

quand nous les appliquons au langage ADN. En effet, ni le langage ADG, ni son sous-ensemble GEADT ne satisfait le test de cohérence. A fortiori, le langage ADG ne satisfait pas la requête **CO**. Par conséquent, ni GEADT ni ADG ne satisfont **CT**.

Pour obtenir un langage cible intéressant, il est donc nécessaire d'imposer à ADN des restrictions supplémentaires à celles considérées pour définir le langage GEADT. L'une d'entre elles consiste à renforcer la condition de décomposabilité portant sur les nœuds de conjonction et elle conduit au langage EADT des arbres de décision affine étendus.

5.5.1 Les arbres de décision affine étendus (EADT)

Le langage EADT s'appuie sur la notion de nœud de conjonction affine décomposable, définie comme suit :

Définition 70 (Nœud affine décomposable)

Soit un nœud de conjonction N ayant pour fils $N_1, \ldots, N_k$ dans une formule ADN Δ. N est affine décomposable si et seulement si :

(1) pour tout $i, j \in 1, \ldots, k$, si $i \neq j$, alors $Var(N_i) \cap Var(N_j) = \varnothing$, et

(2) il existe au plus un fils N_i de N tel que pour tout nœud de décision affine $N' = \langle \delta, N_-, N_+ \rangle$ de Δ qui est un ancêtre de N et pour lequel la clause affine δ n'est pas réduite à un littéral, nous avons $Var(N_i) \cap var(\delta) \neq \varnothing$.

Si nous ne prenons en compte que la première condition de la définition des nœuds affines décomposables, alors le nœud N est (classiquement) décomposable.

Définition 71 (EADT)

Une formule EADT est une formule GEADT dont les nœuds de conjonction (qui ne participent pas aux nœuds de décision) sont affines décomposables.

À partir de EADT nous définissons aussi deux sous-langages. Le premier ADT, a pour formules les arbres de décision affine définis de la manière suivante :

Définition 72 (ADT)

Une formule ADT est une formule EADT dans laquelle nous ne trouvons aucun nœud de conjonction, sauf ceux issus des nœuds de décision.

Ainsi, il est possible de considérer ce langage comme une extension du langage DT dans laquelle les nœuds de décision standard sont généralisés en des nœuds de décision affine. Chaque chemin entre la racine R_Δ d'une formule ADT Δ et une feuille de Δ est associé à une formule affine, la conjonction des clauses affines rencontrées sur le chemin.

Le second langage que nous considérons est le langage EDT des arbres de décision étendu ; dans lequel les nœuds internes sont soit des nœuds de décision affine unitaire (i.e., des nœuds de décision classiques) soit des nœuds de conjonction classiquement décomposables :

Définition 73 (EDT)

Une formule EDT est une formule EADT dans laquelle tous les nœuds de décision sont des nœuds de décision standard.

Le langage EDT peut donc être vu comme une extension du langage DT dans laquelle les nœuds internes des formules peuvent aussi être des nœuds de conjonction décomposables. Le langage EDT peut donc aussi être vu comme une restriction du langage Decision-DNNF aux formules dont la forme est arborescente.

Il est important de noter que le langage AFF est polynomialement traduisible vers ADT, donc vers EADT. En effet, tout chemin entre la racine R_Δ d'un ADT Δ et une feuille de Δ représente une formule affine. Donc une formule affine $\Sigma = \bigwedge_{i=1}^{k} \delta_i$ où chaque δ_i ($i \in 1, \ldots, k$) est une clause affine, est équivalente à une formule ADT sous forme de « peigne ». Pour construire une telle formule ADT à partir de Σ, il suffit, en effet, de parcourir la formule affine et pour chaque clause affine δ_i, de créer un nœud $N = \langle \delta_i, N_-, N_+ \rangle$ où N_- est la feuille $\bot$ et N_+ est la racine d'une représentation ADT de la formule affine obtenue en retirant δ_i de Σ (et la conjonction vide de clauses affines est représentée par la feuille $\top$ comme il se doit).

Contrairement au langage AFF, le langage DT et ses sur-ensembles ADT, EDT et EADT sont des langages complets pour la logique propositionnelle. Cela vient simplement du fait que le langage des arbres de décision binaire « standard » DT est un langage complet et qu'il est un sous-ensemble de ADT, de EDT, et donc de EADT.

La propriété de complétude est aussi satisfaite par le langage des arbres de décision ordonnés ($\text{ODT}_<$), le sous-ensemble de DT des formules Δ où chaque chemin de la racine aux feuilles de Δ respecte un ordre strict et total $<$ sur l'ensemble des variables. Il apparaît clairement que $\text{ODT}_<$ est également un sous-ensemble de $\text{OBDD}_<$ (plus précisément, $\text{ODT}_<$ est l'intersection des langages DT et $\text{OBDD}_<$). Comme $\text{ODT}_<$ est un sous-ensemble de $\text{OBDD}_<$, le langage des arbres de décision réduits possède une propriété intéressante de canonicité : deux formules logiquement équivalentes ont la même représentation en $\text{ODT}_<$, une fois réduites (où seule la règle d'élimination des variables redondantes est appliquée). Cette propriété de canonicité n'est pas offerte par les langages DT, ADT, EDT et EADT.

5.5.2 Extension de la carte de compilation et efficacité spatiale

Dans (Koriche et al. 2013), nous avons analysé les langages introduits dans la section précédente, i.e., le langage des arbres de décision affine (ADT), celui des arbres de décision étendus (EDT)

et celui des arbres de décision affine étendus (EADT), selon les critères utilisés dans la carte de compilation des formules propositionnelles. Nous nous sommes efforcés de déterminer si ces langages satisfont ou non les requêtes et les transformations proposées dans la carte de compilation (Darwiche et Marquis 2002). Nous avons évalué aussi l'efficacité spatiale de ces langages.

Dans la suite, nous commençons par résumer l'ensemble des requêtes définies dans la carte de compilation et nous comparons les langages introduits aux principaux langages déjà étudiés et situés dans la carte de compilation. Le tableau 5.5 indique les requêtes offertes par les différents langages.

$\mathcal{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
EADT	√	√	√	√	?	?	√	√
EDT	√	√	√	√	?	?	√	√
ADT	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√
$\text{ODT}_<$	√	√	√	√	√	√	√	√
d-DNNF	√	√	√	√	?	○	√	√
$\text{OBDD}_<$	√	√	√	√	√	√	√	√

TABLE 5.5 – Requêtes. √ signifie "satisfait" et ○ signifie "ne satisfait pas sauf si P = NP."

Il apparaît que le langage ADT des arbres de décision affine et ses sous-ensembles DT et $\text{ODT}_<$ satisfont l'ensemble des requêtes proposées dans la carte de compilation. Ainsi, ADT et ses sous-ensembles sont équivalents au langage $\text{OBDD}_<$ du point de vue des requêtes offertes. De même, EADT et son sous-ensemble EDT sont équivalents au langage d-DNNF pour ce qui est des requêtes pour lesquelles nous connaissons des algorithmes en temps polynomial pour les réaliser.

En ce qui concerne les transformations, le tableau 5.6 résume les transformations offertes par les nouveaux langages introduits. Nous remarquons qu'à l'instar des requêtes, les langages $\text{ODT}_<$, DT et ADT sont équivalents au langage $\text{OBDD}_<$ du point de vue des transformations offertes. De même, EADT et son sous-ensemble EDT sont équivalents au langage d-DNNF pour ce qui est des requêtes pour lesquelles nous connaissons des algorithmes en temps polynomial pour les réaliser.

La figure 5.3 représente les différents résultats que nous avons obtenus concernant l'efficacité spatiale des nouveaux langages introduits, relativement à plusieurs langages cibles déjà insérés dans la carte de compilation. Il s'agit des langages sd-DNNF, d-DNNF_T, d-DNNF et $\text{OBDD}_<$. Le langage d-DNNF_T est le sous-ensemble de d-DNNF introduit par Pipatsrisawat et Darwiche (2008),

$\mathcal{L}$	CD	FO	SFO	∧C	∧BC	∨C	∨BC	¬C
EADT	√	○	?	○	○	○	?	?
EDT	√	○	?	○	○	○	?	?
ADT	√	○	√	○	√	○	√	√
DT	√	○	√	○	√	○	√	√
$\text{ODT}_<$	√	○	√	○	√	○	√	√
d-DNNF	√	○	○	○	○	○	○	•
$\text{OBDD}_<$	√	•	√	•	√	•	√	√

TABLE 5.6 – Transformations. √ signifie "satisfait", • signifie "ne satisfait pas" et ○ signifie "ne satisfait pas sauf si P = NP".

qui contient les formules d-DNNF qui respectent un arbre donné T de décomposition des variables. sd-DNNF est l'union de tous ces langages quand T varie. Chaque langage $\text{OBDD}_<$ est un sous-ensemble de sd-DNNF.

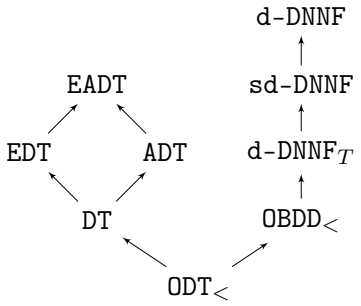


FIGURE 5.3 – Graphe d'inclusion. $\mathcal{L}_1 \to \mathcal{L}_2$ indique que $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

Étant donné ce graphe d'inclusion et du fait que DT (resp. EDT) ne satisfait pas plus de requêtes ou de transformations que ADT (resp. EADT), nous pouvons déduire que DT (resp. EDT) ne peut pas être un meilleur choix que ADT (resp. EADT) du point de vue de la carte de compilation. C'est pourquoi nous avons choisi de nous concentrer sur les langages ADT et EADT. Nous avons obtenu les résultats d'efficacité spatiale supplémentaires suivants :

Proposition 3

- CNF $\not\leq_s$ ADT
- DNF $\not\leq_s$ ADT
- $\text{OBDD}_<$ $\not\leq_s$ ADT
- d-DNNF$_T$ $\not\leq_s$ ADT
- EADT $<_s$ ADT

Sur la base de ces résultats, nous pouvons conclure que $\text{OBDD}_<$ ne domine pas ADT du point de vue de la carte de compilation. De plus, comme ADT $\subseteq$ EADT, $\text{OBDD}_<$ ne domine pas EADT. Nos résultats de concision montrent qu'aucun des langages « plats » CNF et DNF n'est au moins aussi succinct que ADT ou EADT. Bien que nous ne connaissions pas comment d-DNNF et EADT se comparent du point de vue de l'efficacité spatiale, nous pouvons néanmoins conclure que la sous-classe sd-DNNF de d-DNNF ne domine ni ADT, ni EADT.

5.5.3 Un compilateur CNF vers EADT

Dans cette section, nous présentons un algorithme de compilation du langage CNF vers le langage EADT. Tout d'abord, nous introduisons le langage des formules propositionnelles sous forme normale conjonctive étendue (ECNF) qui sera utilisé dans notre algorithme de compilation.

Définition 74 (Clause étendue et forme normale conjonctive étendue)

Une clause étendue est une disjonction finie de clauses affines. Le langage des formules sous forme normale conjonctive étendue (`ECNF`) est l'ensemble de toutes les conjonctions finies de clauses étendues.

Notons que la conversion d'une formule `CNF` en `ECNF` s'effectue en temps linéaire de manière évidente. En effet, toute clause d'une `CNF` est une disjonction finie de clauses affines unitaires. Donc toute clause est une clause étendue et par extension, toute formule `CNF` est une formule `ECNF` particulière.

Une approche naturelle pour compiler une formule `CNF` en `EADT` repose sur l'exploitation d'une forme généralisée de la décomposition de Shannon. Cette forme généralisée de la décomposition de Shannon permet d'introduire des nœuds de décision étiquetés non plus par des variables mais par des formules propositionnelles. Nous définissons la décomposition de Shannon généralisée de la manière suivante :

Définition 75 (Décomposition de Shannon généralisée)

Pour toutes formules propositionnelles Σ et δ, la décomposition de Shannon généralisée de Σ selon δ est définie par :

$$\Sigma \equiv (\delta \wedge \Sigma) \vee (\neg\delta \wedge \Sigma)$$

Comme la décomposition de Shannon classique, le principe de la décomposition de Shannon généralisée est de séparer les modèles possibles de Σ en deux sous-ensembles disjoints : ceux qui satisfont δ et ceux qui satisfont $\neg\delta$. Le nœud $\vee$ introduit dans $(\delta \wedge \Sigma) \vee (\neg\delta \wedge \Sigma)$ est un nœud de décomposition de Shannon généralisée. Nous remarquons ainsi que les nœuds de décision affine des langages `ADT`, `EDT` et `EADT` sont en fait des nœuds de décomposition de Shannon généralisé dans lesquels Σ est une formule `ADT`, `EDT` ou `EADT` et δ est une clause affine.

Cependant, utilisée directement, la décomposition de Shannon généralisée est peu exploitable pour la compilation puisque l'un des avantages de la décomposition de Shannon classique est qu'une variable est éliminée de Σ à chaque nœud. Ceci garantit la terminaison d'un algorithme de compilation utilisant cette méthode. En effet, comme au moins une variable est supprimée à chaque itération, après un nombre fini d'itérations, toutes les variables de Σ sont considérées. Nous restreignons la décomposition de Shannon généralisée afin d'assurer que son application élimine à chaque étape au moins une variable dans Σ.

Soient deux formules propositionnelles Σ et δ, et soit x une variable. Nous notons $\Sigma_{|x \leftarrow \delta}$ la formule obtenue en remplaçant toute occurrence de x dans la formule Σ par la formule δ. Avec cette notation, nous définissons la décomposition de Shannon généralisée restreinte de la façon suivante :

Définition 76 (Décomposition de Shannon généralisée restreinte)

Pour toutes formules propositionnelles Σ et δ et toute variable x telle que $x \notin Var(\delta)$, la décomposition de Shannon généralisée restreinte est définie par :

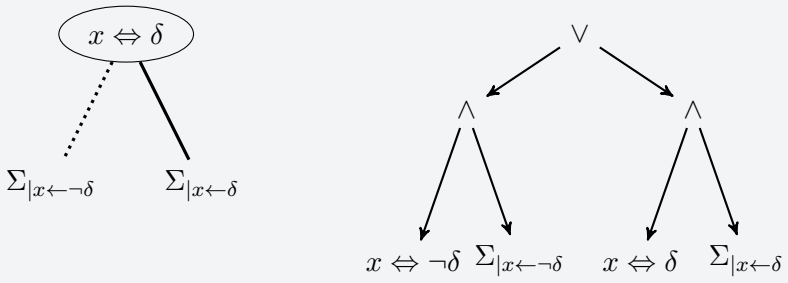
$$\Sigma \equiv ((x \Leftrightarrow \neg\delta) \wedge \Sigma_{|x \leftarrow \neg\delta}) \vee ((x \Leftrightarrow \delta) \wedge \Sigma_{|x \leftarrow \delta})$$

Cette équivalence montre qu'en utilisant la décomposition de Shannon généralisée restreinte, les modèles de Σ sont séparés en deux sous-ensembles disjoints : ceux qui satisfont $x \Leftrightarrow \delta$ (i.e., ceux où x et δ prennent la même valeur de vérité) et ceux qui satisfont la négation de cette formule, qui équivaut à $x \Leftrightarrow \neg\delta$.

Utiliser la forme généralisée de la décomposition de Shannon restreinte consiste à exploiter cette équivalence comme une règle de réécriture, orientée de la gauche vers la droite. Nous allons ainsi remplacer à chaque étape une variable de la formule de départ par une formule propositionnelle ne contenant pas cette variable. La décomposition de Shannon standard est retrouvée dans le cas où la formule $\delta = \top$ est considérée. Effectivement, dans ce cas cela revient à remplacer x par vrai d'un côté et faux de l'autre. La validité de la décomposition de Shannon généralisée restreinte vient du fait que la formule Σ est logiquement équivalente à $((x \Leftrightarrow \neg\delta) \wedge \Sigma) \vee ((x \Leftrightarrow \delta) \wedge \Sigma)$, et du fait que pour toute formule propositionnelle δ (ou sa négation), l'expression $(x \Leftrightarrow \delta) \wedge \Sigma$ est logiquement équivalente à $(x \Leftrightarrow \delta) \wedge \Sigma_{|x \leftarrow \delta}$.

Exemple 31

Soit une formule propositionnelle Σ. La décomposition de Shannon généralisée restreinte sur la variable x de Σ selon la formule δ est donnée par : $\Sigma \equiv (x \Leftrightarrow \neg\delta \wedge \Sigma_{|x \leftarrow \neg\delta}) \vee (x \Leftrightarrow \delta \wedge \Sigma_{|x \leftarrow \delta})$



Appliquer cette décomposition revient à créer un nœud de décision étiqueté par la formule $x \Leftrightarrow \delta$, comme montré sur la figure ci-avant. Par construction, dans les formules $\Sigma_{|x \leftarrow \delta}$ et $\Sigma_{|x \leftarrow \neg\delta}$ la variable x n'apparaît pas.

Pour construire une formule `EADT` équivalente à une formule `ECNF` Σ donnée, nous appliquons la décomposition de Shannon généralisée restreinte en choisissant pour δ une clause affine qui ne contient pas la variable de décision x.

Algorithme 5.17 : eadt(Σ)

entrée : une formule ECNF Σ
sortie : une formule EADT Σ_{eadt} équivalente à Σ

1 if $\Sigma \equiv \top$ then return leaf($\top$);
2 if solve(Σ) $= \bot$ then return leaf($\bot$);
3 soient $\Sigma_1, \cdots, \Sigma_k$ les composantes connexes de Σ;
4 if $k>1$ *et la décomposition* $\Sigma_1, \cdots, \Sigma_k$ *est affine* then
5 return aNode(eadt(Σ_1), $\cdots$, eadt(Σ_k));
6 choisir une clause affine simplifiée $x \oplus \delta$ telle que $Var(x \oplus \delta) \subseteq Var(\delta')$ avec $\delta' \in \Sigma$;
7 return dNode($x \oplus \delta$, eadt($\Sigma_{|x \leftarrow \delta \oplus \top}$), eadt($\Sigma_{|x \leftarrow \delta}$));

Comme $x \Leftrightarrow \delta$ équivaut à $x \oplus \delta \oplus \top$ et $x \Leftrightarrow \neg\delta$ équivaut à $x \oplus \delta$, nous pouvons reformuler la décomposition de Shannon généralisée restreinte en utilisant $\oplus$ au lieu de $\Leftrightarrow$:

$$\Delta \equiv ((x \oplus \delta) \wedge \Delta_{|x \leftarrow \delta \oplus \top}) \vee ((x \oplus \delta \oplus \top) \wedge \Delta_{|x \leftarrow \delta})$$

Par ailleurs, quand δ est une clause affine, $x \oplus \delta \oplus \top$ et $x \oplus \delta$ sont aussi des clauses affines. De ce fait, sous cette forme, appliquer la décomposition de Shannon généralisée restreinte consiste à construire des nœuds de décision affine étiquetés par des clauses de la forme $x \oplus \delta$ et à supprimer à chaque étape la variable x de Σ qui a été sélectionnée.

L'algorithme 5.17 décrit en pseudo-code notre compilateur eadt qui se fonde sur la décomposition généralisée restreinte de Shannon. Cet algorithme prend en entrée une formule ECNF Σ et retourne une formule EADT logiquement équivalente à Σ. Les deux premières lignes gèrent les deux cas particuliers où Σ est valide ou contradictoire. Dans les deux cas, la feuille correspondante ($\top$ si Σ est valide, $\bot$ si Σ est contradictoire) est retournée.

Nous notons au passage que le problème de l'incohérence (resp. de la validité) pour une formule sous forme ECNF a la même complexité que pour son sous-ensemble CNF, c'est-à-dire qu'il est coNP-complet (resp. dans P). Cela est évident pour le problème de l'incohérence. En effet, comme toute formule CNF est une formule ECNF, s'il existait un algorithme en temps polynomial vérifiant qu'une formule ECNF est incohérente, nous aurions un algorithme en temps polynomial pour cette requête restreinte aux formules CNF, or ce problème est coNP-complet pour les formules CNF. Pour le problème de validité, une formule ECNF est valide si et seulement si chaque clause étendue de la formule est valide. Une clause étendue $\delta_1 \vee \ldots \vee \delta_k$ (où chaque δ_i, $i \in \{1, \ldots, k\}$ est une clause affine) est valide si est seulement si la formule affine $(\delta_1 \oplus \top) \wedge \ldots \wedge (\delta_k \oplus \top)$ est contradictoire. La négation d'une clause étendue représente donc une formule affine. Or, nous savons que tester si une formule affine est incohérente est faisable en temps polynomial dans sa taille. Donc la validité d'une clause étendue, et au-delà d'une ECNF est décidable en temps polynomial en la taille de la formule donnée.

À la ligne 3, nous recherchons les composantes connexes $\Sigma_1, \cdots, \Sigma_k$ de Σ en parcourant son graphe primal. S'il existe plusieurs composantes et que la conjonction obtenue est affine décomposable [14], les formules Σ_i ($i \in \{1, \ldots, k\}$) sont compilées séparément de manière récursive en formules EADT. Puis les formules obtenues sont jointes en utilisant la fonction aNode (ligne 4) par un nœud $\wedge$ qui est alors, par construction, affine décomposable. Notons que la recherche

14. Nous conservons pour cela l'ensemble des clauses affines correspondant aux choix effectués sur la branche courante ; cela n'est pas explicité dans l'algorithme dans un souci de garder celui-ci lisible.

de la décomposition s'effectue avant un branchement. Lorsque Σ ne possède qu'une composante connexe ou que la décomposition n'est pas affine, le compilateur choisit une clause affine $x \oplus \delta$ dans laquelle toute variable possède au moins une occurrence dans Σ (ligne 5). Ensuite, nous effectuons le branchement sur cette clause en utilisant la décomposition généralisée restreinte de Shannon (ligne 6). L'algorithme est appelé récursivement avec les formules conditionnées, dans lesquelles x est remplacé par $\delta \oplus \top$ ou par δ. Enfin, la fonction dNode retourne un nœud de décision correspondant. Nous remarquons que, privé des lignes 3 et 4 le compilateur eadt se réduit à un compilateur de ECNF vers ADT.

La terminaison de l'algorithme eadt est garantie par le fait que par définition, la clause affine simplifiée δ dans $x \oplus \delta$ est une clause affine qui ne contient pas x. Comme ni $\Sigma_{|x \leftarrow \delta \oplus \top}$ ni $\Sigma_{|x \leftarrow \delta}$ ne contiennent x, les étapes 5 et 6 ne peuvent donc être appliquées qu'un nombre fini de fois. Comme la décomposition généralisée de Shannon exprime une équivalence valide, l'algorithme eadt est correct (la décomposabilité affine des nœuds $\wedge$ éventuellement introduits est garantie par construction).

L'algorithme eadt a été implémenté sur la base du solveur état de l'art MiniSat (Eén et Sörenson 2004), qui s'appuie sur une architecture CDCL (*Conflict Driven Clause Learning*) (Moskewicz et al. 2001b). Nous avons étendu MiniSat aux formules ECNF. Pour cela, nous avons modifié la façon dont sont gérés les *watched* littéraux. Nous avons adapté la propagation aux clauses étendues et la gestion des conflits pour prendre en compte des clauses affines.

Pour pouvoir utiliser eadt, il reste à préciser les choix effectués à la ligne 6 : quelle variable x sélectionner, quelle clause affine δ retenir. Dans notre compilateur, l'heuristique utilisée pour choisir les clauses affines de la forme $x \oplus \delta$ à la ligne 6 est basée sur le concept d'activité des variables (VSIDS, *Variable State Independant Decaying Sum*) (Moskewicz et al. 2001b). L'idée de cette heuristique est d'éliminer les variables qui apparaissent le plus dans les conflits lors de la recherche de solution. Ces variables étant généralement des variables clés du problème, les considérer le plus haut possible lors de la recherche permet en général de réduire la taille de la forme compilée. Plus précisément, pour chaque clause étendue ϵ de Σ, le score de ϵ est calculé comme la somme des scores de chaque clause affine qu'elle contient, où le score d'une clause affine est la somme des scores VSIDS de ses variables, $\texttt{VSIDS}(\epsilon) = \Sigma_{l_i \in \epsilon} \texttt{VSIDS}(var(l_i))$. Nous sélectionnons une clause étendue ϵ de Σ de score maximal, et les variables de ϵ sont triées par ordre décroissant selon leur score VSIDS. La variable x sélectionnée est la première variable de la liste ainsi créée. La clause affine δ que nous créons est formée des $k-1$ variables suivantes de la liste. Notons que sélectionner uniquement les variables de la même clause étendue ϵ de Σ permet d'éviter de générer des connexions supplémentaires entre les variables qui ne sont pas déjà connectées dans le graphe primal de Σ. Ceci est donc fait dans le but de ne pas diminuer le nombre de composantes connexes qu'il est possible de trouver.

Nous nous efforçons aussi de réduire la formule ECNF courante à chaque étape de remplacement de x par δ ou par $\delta \oplus \top$: nous supprimons les littéraux redondants qui ont pu apparaître suite à l'ajout de la clause affine δ pour remplacer la variable x. Nous supprimons aussi des clauses étendues construites les clauses affines rendues contradictoires par le remplacement de x par δ (respectivement $\delta \oplus \top$).

Nous tirons enfin avantage d'une méthode simple de filtrage utilisée uniquement dans les premiers nœuds de l'arbre de recherche. Cette méthode consiste à calculer les clauses affines impliquées par la formule ECNF donnée initialement Σ. Une méthode pour détecter des clauses affines « simples » dans une ECNF est d'utiliser la propagation unitaire (voir Lagniez et Marquis (2017a) et la section 5.2.2.1). L'idée est de rechercher des équivalences $\ell \Leftrightarrow \ell'$ entre les littéraux

de Σ. Reconnaître une clause affine $\delta\ \ell \oplus \ell'$ impliquée par la formule Σ permet de couper l'une des branches de l'arbre de recherche. En effet, si $\Sigma \models \delta$, nous avons $\Sigma \wedge \neg\delta \models \bot$. Il est donc inutile d'explorer la branche $\delta \oplus \top$ de l'arbre. Plus une clause affine est trouvée haut dans l'arbre, plus cela permet de réduire l'espace de recherche et, par conséquent, de réduire la taille de la représentation en `EADT` finale.

Une fois la compilation terminée, la formule $\mathsf{eadt}(\Sigma)$ obtenue peut être utilisée pour compter directement le nombre de modèles de Σ. En effet, notre approche consistant à créer une clause affine $x \oplus \delta$ avec laquelle nous remplaçons toute occurrence de la variable x par δ (ou $\delta \oplus \top$) dans la formule Σ, la variable x peut être considérée comme variable pivot de la clause affine $x \oplus \delta$ puisqu'elle n'apparaît plus dans les clauses affines suivantes. Cela est vrai pour l'ensemble des clauses affines créées sur le chemin entre la racine et toute feuille. Ainsi, les formules affines engendrées par notre algorithme sont triangulées par construction. Il suffit donc de connaître la profondeur p du chemin entre la racine et la feuille, ainsi que le nombre de variables de la composante connexe courante pour déterminer son nombre de modèles.

5.5.4 Discussion

Nous avons introduit des langages cibles de compilation s'appuyant sur la notion de nœuds de décision affine. Nous avons montré que le langage `EADT` est un langage de compilation attractif quand le comptage de modèles est une requête clé. En particulier, le langage `EADT` offre les mêmes requêtes que `d-DNNF`. Le sous-ensemble `ADT` de `EADT` satisfait toutes les requêtes et les mêmes transformations que celles offertes par le langage $\texttt{OBDD}_<$. De plus, $\texttt{OBDD}_<$ n'est pas au moins aussi succinct que `ADT`, ce qui place `ADT` comme un challenger possible du langage $\texttt{OBDD}_<$.

Ce travail ouvre plusieurs perspectives pour de futures recherches. D'un point de vue théorique, une extension naturelle de `ADT` est l'ensemble des DAG finis ne possédant qu'un nœud racine, où les feuilles sont étiquetées par $\top$ ou $\bot$, et les nœuds internes sont des nœuds de décision affine. Cependant, ce langage n'est pas un langage intéressant du point de vue de la compilation, car il contient le langage des diagrammes de décision binaire (`BDD`) (Bryant 1986) comme sous-ensemble, et `BDD` ne satisfait *aucune* requête proposée dans la carte de compilation (Darwiche et Marquis 2002), sauf si $\mathsf{P} = \mathsf{NP}$. Ainsi, le problème de déterminer des classes intéressantes de graphes de décision affine offrant le comptage de modèles semble un problème intéressant à creuser.

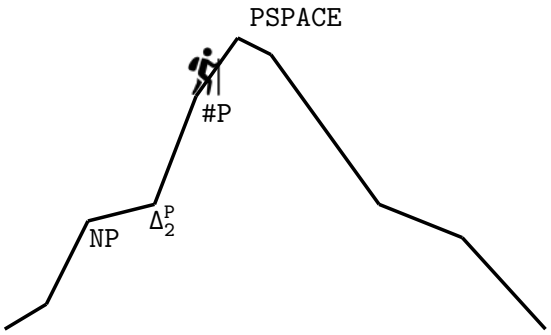
Il serait aussi possible d'exploiter le *caching* dans le cas où il est juste nécessaire de compter le nombre de modèles de l'entrée. Dans ce cas, il serait possible d'élaborer un compteur de modèle exploitant la décomposition de Shannon généralisée avec des clauses affines. Un tel compteur de modèles serait par nature strictement plus puissant que les compteurs de l'état de l'art, dans le sens il pourrait compter le nombre de modèles d'une formule affine en temps polynomial. De plus, il serait facile de voir qu'il est aussi capable de simuler les compteurs de modèles classiques.

Le comptage de modèles approché est souvent suffisant pour de nombreuses applications. Ainsi, il serait aussi intéressant d'étudier si le langage `EADT` peut être exploité afin de compter de manière approchée le nombre de modèles d'une formule `CNF`. En effet, de nombreux compteurs de modèles approchés exploitent la notion de clause affine afin de partitionner l'espace de recherche de manière uniforme (Stockmeyer 1983, Meel et al. 2016, Soos et Meel 2019). Il serait donc intéressant de déterminer s'il est possible d'exploiter la forme arborescente des `ADT` afin de réduire le nombre de cellules à considérer. Il serait aussi intéressant de voir s'il est possible d'exploiter la décomposition en composantes connexes des `EADT`.

D'un point de vue pratique, différentes pistes pour améliorer notre compilateur existent. Il serait intéressant de profiter des techniques de pré-traitement développées pour SAT (Piette et al. 2008, Järvisalo et al. 2012b, Lagniez et Marquis 2017a) afin de simplifier la formule `CNF` en entrée avant de la compiler. De plus, il pourrait être utile d'utiliser l'élimination de Gauss pour traiter plus efficacement (voir par exemple (Li 2003, Chen 2007, Soos et al. 2009)) les instances contenant des sous-problèmes correspondant à des clauses affines, comme celles données dans (Crawford et Kearns 1995, Cannière 2006). Finalement, tenir compte d'autres heuristiques pour choisir les clauses affines de branchement (par exemple, utiliser l'heuristique que nous avons mise en place dans notre compilateur `d4` afin de favoriser la création de nœuds $\wedge$ décomposables (Lagniez et Marquis 2017a)) pourrait rendre plus efficace la compilation.

Dans les expérimentations que nous avons conduites (Koriche et al. 2013), les clauses affines étaient généralement de taille 2 (excepté pour celles qui étaient calculées explicitement *via* la méthode de pré-traitement). Le fait de limiter la taille des clauses affines restreint le pouvoir d'expression des `EADT`, et donc l'efficacité du langage. L'un de nos objectifs est donc de proposer une heuristique de choix de clauses affines qui permettrait d'exploiter la puissance du langage `EADT` tout en permettant de construire des `EADT` de petites tailles. Pour cela, nous pourrions nous inspirer des travaux de Prcovic (2016) qui traitent d'une généralisation de la résolution classique s'appuyant sur l'introduction de clauses hybrides contenant des clauses affines. Plus précisément, l'auteur propose de modifier la règle de résolution standard dans le cas où la clause générée est tautologique, de manière à produire une clause non triviale qui est en fait une clause affine étendue. Partant de ces travaux, nous pourrions dans un premier temps réaliser un ensemble de résolutions de ce type afin de créer un ensemble de clauses affines étendues. Puis, nous pourrions voir chaque clause affine comme une variable et alors modifier l'heuristique en conséquence.

5.6 Conclusion



Cette partie de l'ascension a été l'occasion d'appliquer tout ce que nous avons vu pendant la première partie de notre voyage. En fait, la classe de complexité $\#P$ est une classe très intéressante et pour laquelle il y a encore de nombreux défis pratiques et théoriques à relever (Marquis 2015). Le fait que `#SAT` soit calculatoirement très difficile, nous a permis d'utiliser des méthodes de pré-traitement d'une complexité élevée. En effet, nous avons vu qu'il était possible de calculer le *backbone* de la formule ou encore rechercher des définitions dans cette dernière en pré-traitement, et que même si les problèmes associées sont coNP-difficiles, cela permet de réduire fortement le temps nécessaire au calcul du nombre de modèles de l'instance ou le temps de compilation de celle-ci et la taille de sa forme compilée.

Cette étape nous a aussi permis de mieux comprendre le fonctionnement des compteurs de modèles et des compilateurs de l'état de l'art, ce qui a donné naissance à l'outil `d4`. Dans ce contexte, nous avons actionné deux leviers existants afin de limiter l'espace de recherche exploré lors du processus de compilation (ou de comptage) : l'utilisation de nœuds $\wedge$ décomposables et le partage de structure. Pour ce qui est de l'utilisation de la décomposabilité, nous avons proposé une nouvelle heuristique qui s'appuie sur l'utilisation parcimonieuse d'un algorithme de partitionnement d'hypergraphe afin de couper le plus rapidement possible la formule en plusieurs composantes connexes. Pour ce qui est du partage de structure, il est très difficile de prévoir de manière heuristique qu'une suite d'affectations conduira à une sous-formule déjà compilée. Ici notre approche est un peu plus opportuniste, dans le sens où le partage de structure est réalisé *via* le *caching*. Dans ce contexte, nous avons proposé un nouveau schéma de *caching* qui permet d'approcher correctement la relation d'équivalence entre formules et qui a aussi l'avantage d'être plus compact que les schémas ayant un pouvoir d'approximation similaire.

L'élaboration de notre propre compteur de modèles nous a permis d'attaquer un problème connexe : le comptage de modèles projetés. Bien qu'il puisse « facilement » être traité avec une version légèrement modifiée d'un compteur de modèles de l'état de l'art, le fait de devoir oublier des variables rend ce problème plus difficile en pratique que le problème de comptage de modèles classique. En fait, pour considérer les variables à oublier X, il est nécessaire de modifier l'heuristique de choix de variables en imposant de ne pas brancher sur les variables de X. Cette contrainte impacte fortement l'heuristique de choix de variables et réduit fortement l'apparition de composantes connexes disjointes dans la formule. Afin d'éviter cet écueil, nous avons proposé une nouvelle approche qui va découper récursivement, sous la forme d'une disjonction déterministe, la formule d'entrée Σ de manière à ne considérer que des formules sans variables à oublier, c'est-à-dire des formules construites sur les variables de $Var(\Sigma) \setminus X$ pour lesquelles nous sommes capables d'utiliser toute la puissance des compteurs de l'état de l'art.

Dans le cadre de nos travaux, nous nous sommes aussi intéressés à la compilation de connaissances, et, plus précisément, à l'utilisation des formules affines. Il existe très peu de langages permettant le comptage de modèles en temps polynomial, et le langage `AFF` des formules affines fait partie de ceux-là. Cependant, ce langage n'est pas complet pour la logique propositionnelle, ce qui le rend de manière générale inutilisable en pratique. Pour pallier ce problème nous avons proposé de nouveaux langages cibles pour la compilation de formules propositionnelles. En particulier, les langages `EDT` des arbres de décision étendus, `ADT` des arbres de décision affine et `EADT` des arbres de décision affine étendus, ont été introduits et analysés selon les critères mis en œuvre dans la carte de compilation. Nous avons ainsi déterminé les requêtes et les transformations de la carte de compilation réalisables en temps polynomial à partir de représentations dans ces langages, et montré que certaines ne le sont pas sauf si $\mathsf{P} = \mathsf{NP}$. Nous avons également comparé l'efficacité spatiale relative de ces langages à celles d'autres langages cibles existants. Il ressort de cette étude que le langage `EADT` est intéressant lorsque la requête du comptage de modèles est une question cruciale. Nous avons aussi développé un compilateur du langage `CNF` vers `EADT`, et nous avons montré que ce langage est réellement attractif en pratique.

Dans ce chapitre nous n'avons pas relaté l'ensemble de nos travaux autour du comptage de modèles. En fait, les travaux réalisés en collaboration avec Anicet Bart, Frédéric Koriche et Pierre Marquis ne sont pas repris ici. Ces travaux concernent l'élaboration d'un langage de compilation permettant d'utiliser une nouvelle forme de partage de structures s'appuyant sur la notion de symétrie (Bart et al. 2014), et l'élaboration d'un nouvel encodage `CNF` pour la modélisation de réseaux bayésiens (Bart et al. 2016). Ce n'est pas que ces travaux ne sont pas intéressants, au contraire, c'est juste que ce chapitre est déjà suffisamment volumineux et que l'ajout de nouvelles

sections peut rendre sa lecture difficile. Pour les mêmes raisons, j'ai choisi de ne pas présenter les travaux réalisés en collaboration avec Pierre Marquis et Szczepanski sur le comptage de modèles dans un environnement à mémoire distribuée (Lagniez et al. 2018d).

Les travaux que nous avons réalisés depuis six ans sur les sujets de la compilation de connaissances et du comptage de modèles nous permettent aujourd'hui d'envisager d'attaquer des problèmes connexes. L'un de ces problèmes concerne la résolution pratique du problème `E-MAJSAT` (Littman et al. 1998), qui est une extension du problème `SAT` permettant de modéliser des problèmes réels où il est nécessaire de considérer une certaine forme d'incertitude (Drummond et Bresina 1990, Dechter 1998, Park et Darwiche 2004). C'est aussi un cas spécial de la classe plus générale `SSAT` (*stochastic satisfiability*) (Littman et al. 2001). Plus précisément, étant donnée une formule `CNF` Σ construite sur l'ensemble de variables $E \cup R$, le problème `E-MAJSAT` est le problème de décision qui consiste à déterminer s'il existe une affectation ω des variables de E telle que le nombre de modèles de la formule Σ conditionnée par ω soit supérieur à la moitié du nombre des modèles pouvant être construits sur R. Puisque nous avons à rechercher dans un espace exponentiel une affectation des variables de E (problème `NP`) et, à partir de cette dernière, à compter le nombre de modèles de la formule obtenue par conditionnement (problème `PP`), il est assez facile de comprendre que ce problème est $\texttt{NP}^{\texttt{PP}}$-complet (Littman et al. 1998, Park 2002).

Pour résoudre ces problèmes, une première piste que nous souhaitons explorer concerne l'utilisation de l'algorithme récursif développé dans le cadre du comptage de modèles projetés. Plus précisément, nous pourrions découper l'espace de recherche en une disjonction déterministe sur les variables de E, nous permettant ainsi de considérer des sous-formules construites sur les variables de R. De cette manière, nous pourrions élaguer certaines de ces sous-formules lorsque le nombre de modèles n'est pas suffisant (il n'y a pas forcément d'interprétation des variables E qui satisfasse la condition de majorité) ou bien développer certaines branches de la disjonction qui ont suffisamment de modèles afin de décider si une affectation des variables de E satisfaisant la condition de majorité existe. Une autre question qui nous intéresse concerne l'utilisation des méthodes de pré-traitement dans ce contexte, et plus précisément la notion de définissabilité. Nous aimerions déterminer des pré-traitements qui permettraient d'avoir des garanties quant au problème `E-MAJSAT`. Par exemple, il est facile de voir que si un littéral ℓ de E est pur dans Σ, il est possible de supprimer toutes les clauses de la formule contenant ce littéral (en fait, si ω est une assignation de E contenant $\tilde{\ell}$ qui satisfait la contrainte de majorité, alors assigner ℓ à vrai dans ω ne peut pas faire diminuer le nombre de modèles de la formule induite par conditionnement). Il serait intéressant d'établir si d'autres règles, telle que l'élimination de clauses bloquées (Järvisalo et al. 2010), peuvent être appliquées.

Nous aimerions aussi aller plus loin dans l'élaboration d'outils pour la compilation de connaissances. Dans un premier temps, nous envisageons d'étendre nos travaux sur le comptage de modèles dans un environnement à mémoire distribuée, au problème de compilation de connaissances en `Decision-DNNF`, aussi dans un contexte de calcul à mémoire distribuée. Le principal écueil ici est qu'il est difficile, voire même impossible, de rapatrier *via* un réseau informatique les `Decision-DNNF` construites sur des machines distantes afin de réaliser les requêtes et transformations sur une même machine. Ainsi, si nous souhaitons profiter de la puissance de calcul obtenu en utilisant plusieurs machines, il est nécessaire de laisser les informations concernant les `Decision-DNNF` à l'endroit où elles ont été calculées et de paralléliser le processus d'interrogation de ces formules. Il n'y a pas vraiment de défis intellectuels ici, il faut juste définir proprement le protocole de communication afin de proposer à la communauté un outil facilement utilisable. Le but final de ce projet étant de pouvoir réaliser la compilation dans le *cloud* et de proposer une interface *web* pour l'interrogation.

Nous aimerions aussi travailler à la définition d'autres types de langages ne satisfaisant pas nécessairement le comptage de modèles, mais permettant par exemple de satisfaire le test de cohérence. Comme nous l'avons déjà mentionné, le comptage de modèles est une requête beaucoup plus difficile que la requête de cohérence. Cependant, pour de nombreuses applications le comptage de modèles n'est pas utile, ce que l'utilisateur recherche en premier lieu, c'est un langage assez compact sur lequel il pourra réaliser le conditionnement et le test de cohérence efficacement. Par exemple, en configuration automatique ou en planification nous voulons juste décider, avec des garanties de temps de réponse, si un ensemble de choix conduit à une solution. Nous sommes actuellement, en collaboration avec Petr Kučera, en train d'étudier des langages basés sur l'utilisation de formes traitables pour la cohérence. Plus précisément, nous souhaitons à court terme proposer un compilateur qui exploite la notion de formule complète pour la propagation unitaire (Babka et al. 2013). Nous souhaitons dans un premier temps nous focaliser sur le langage des formules de Horn pour réaliser cela. L'idée est de nous inspirer de nos travaux sur les arbres de décision affine, et l'utilisation de la décomposition de Shannon généralisée. Nous aimerions même aller plus loin, et proposer un outil qui, s'appuyant sur cette notion, permettrait de construire un arbre de décision pour une formule quelconque (mais dont la conjonction conduit à considérer des formules qui ont des propriétés intéressantes – par exemple en ne branchant que sur des clauses contenant au maximum un littéral positif, nous obtenons une disjonction de formules de `Horn`).

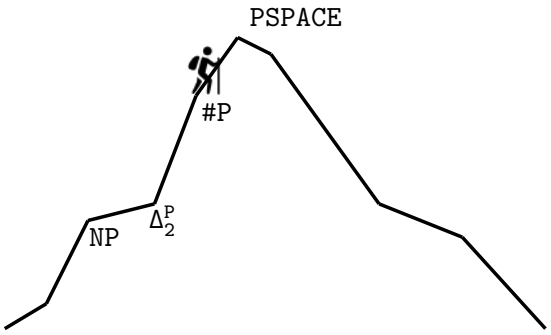
Un autre point qui nous intéresse concerne l'énumération et la représentation de modèles préférés. D'une certaine manière, ce point a déjà été discuté dans le chapitre précédent lorsque nous avons présenté le problème d'énumération de `MCSes`/`MSSes`. En effet, si nous considérons une formule `CNF` Σ augmentée avec des sélecteurs S, un `MCS` de Σ peut être obtenue en considérant les modèles de Σ_S (Σ avec sélecteurs) pour lesquels le nombre de sélecteurs à faux est minimal pour l'inclusion. C'est-à-dire que nous considérons les modèles de Σ_S assujettis à une contrainte. Ce type de problèmes se retrouve aussi dans d'autres applications. Par exemple, en fouille de données, le problème de recherche d'*item sets* fréquents peut être aussi vu comme la recherche de modèles particuliers (Jabbour et al. 2014). Le principal défi ici est que la contrainte de préférence est une contrainte qui porte sur toutes les variables, ce qui réduit fortement la possibilité de décomposer la formule en plusieurs composantes connexes, et pose donc une chape de plomb sur nos compilateurs. Comme cela a été démontré par Koriche et al. (2016), il n'est pas vraiment envisageable de compiler la formule séparément de la contrainte de minimalité. Ainsi, d'un outil efficace pour ce type de problèmes n'est pas facile à élaborer, ce qui rend le problème plus excitant encore !

Nous avons maintenant à notre disposition tous les outils nous permettant d'attaquer la dernière étape de notre périple, c'est-à-dire la résolution pratique de problèmes dans `PSPACE`.

Chapitre 6

Contributions à la résolution du problème de la satisfaisabilité des formules en logique modale

Sommaire



C'est la dernière ligne droite de notre ascension. Nous terminons ce manuscrit par la résolution pratique de problèmes PSPACE-complets. L'ensemble des problèmes que nous avons considérés jusqu'à présent sont des problèmes qui peuvent être résolus facilement en théorie quand nous disposons d'un oracle PSPACE. Alors, pourquoi ne pas avoir commencé par ce chapitre ? À quoi servent les travaux présentés jusqu'à présent ? La réponse à cette question est assez simple : en prenant en compte les spécificités des problèmes que nous souhaitions résoudre, nous avons été en mesure de proposer des solutions plus performantes en pratique. C'est un peu comme si vous aviez à enlever un boulon de 13 et que vous ayez à votre disposition une clef de 13 (un oracle dédié) et une pince multi-prise (un oracle PSPACE). Il est clair que la pince permet de traiter tous les types de boulons, mais je pense que nous serons tous d'accord pour dire que la clef est plus facile à manipuler et aussi plus efficace dans cette situation. En fait, le Graal pour notre problème de boulons s'apparenterait plutôt à un clef à molette. En effet, bien qu'étant un peu moins efficace que la clef, elle reste très agréable d'utilisation et elle permet de considérer tous les types de boulons.

L'analogie avec la clef à molette est intéressante, dans le sens où elle illustre le fait qu'il est possible d'étendre le concept de clef de serrage de manière à s'adapter à tous les types de

boulons, et donc de concevoir une « bonne » solution de manière générale pour attaquer plusieurs situations. Dans notre cas, le concept de clef de serrage peut être joué par un oracle NP, et plus précisément par un solveur SAT. La théorie est assez claire sur le fait qu'il n'est pas possible de traiter efficacement tous les problèmes PSPACE avec un oracle NP. C'est un peu comme avec notre clef de serrage, il n'est pas possible de traiter tous les types de boulons avec une unique clef. En fait, si nous considérons la notion de clef à molette, nous nous rendons compte que l'idée directrice derrière son concept est d'une certaine manière d'approcher la notion de boulon. C'est dans cette idée d'approximation que s'intègrent les travaux présentés dans ce chapitre.

Il existe plusieurs manières d'approcher un problème : nous pouvons par exemple considérer une version du problème où nous avons enlevé ou ajouté des éléments. Dans notre contexte, le concept d'approximation consiste à considérer une version du problème où nous avons ajouté des contraintes (ce qui implique que si l'approximation est satisfaisable alors le problème initial l'est aussi) ou une version du problème où nous avons supprimé des contraintes (ce qui implique que si l'approximation est insatisfaisable alors le problème initial l'est aussi). L'ajout de contraintes conduit alors à une sur-approximation du problème, tandis que la suppression de contraintes conduit à une sous-approximation du problème.

L'idée directrice de ce type d'approche est de considérer des approximations pouvant être traitées (traduites) dans un formalisme moins « puissant » (ici la logique propositionnelle) tout en évitant l'explosion combinatoire pouvant survenir lors d'une transformation directe. Ce type d'approche est connu dans la littérature sous le nom d'approche CEGAR (*Counter-Example-Guided Abstraction Refinement*) (Clarke et al. 2003).

Comme illustration du fonctionnement d'une approche CEGAR, considérons le problème de planification STRIPS (Bylander 1994). Grossièrement, ce problème consiste, étant donné un état initial, un ensemble de buts élémentaire, et un ensemble d'actions, à trouver une suite d'actions « cohérente » qui à partir de l'état initial permet de réaliser la conjonction des buts. Dans ce contexte, une sur-approximation pourrait être de se limiter à rechercher un plan parmi les plans qui ont une taille inférieure à une certaine valeur, et où une sous-approximation pourrait consister à supprimer des buts ou à ajouter de nouvelles actions.

Une fois que le problème a été approché et traité par notre oracle de plus bas niveau, il faut encore interpréter le résultat obtenu quant à la satisfaisabilité de l'approximation. Le cas où la sur-approximation (resp. sous-approximation) est SAT (resp. UNSAT) est triviale, puisque nous pouvons directement conclure à la satisfaisabilité (resp. l'insatisfaisabilité) de la formule initiale. Une difficulté survient lorsque la sur-approximation (resp. sous-approximation) est UNSAT (resp. SAT). Dans ce cas il est nécessaire de rélâcher (resp. ajouter) certaines contraintes (*Abstraction Refinement*).

Ce que nous souhaitons ne pas faire autant que possible, lorsque nous utilisons une approche CEGAR, c'est de traduire le problème dans son ensemble. Cependant, il y a certaines situations où un mauvais choix de type d'approximation conduit irrémédiablement à cette situation. Considérons par exemple une instance de problème PSPACE incohérente, et supposons que nous choisissions d'attaquer le problème avec un solveur SAT en considérant des sur-approximations. Dans cette configuration, nous ne pouvons conclure à l'incohérence de la formule initiale que lorsque nous sommes sûrs que la CNF qui correspond à la sur-approximation est équisatisfaisable au problème initial. Ce qui conduit généralement à une explosion de la taille de la formule CNF générée. Dans cette situation nous aurions été plus avisés de commencer par rechercher une sous-approximation.

Afin d'éviter l'écueil que nous venons de mentionner, nous avons proposé un nouveau schéma

algorithmique, nommé RECAR (*Recursive Explore and Check Abstraction Refinement*), qui permet de jongler entre les deux types d'approximation. Une manière simple de réaliser cela aurait pu être de considérer chacune des procédures séparément pendant un certain laps de temps. Par exemple, sur notre problème de planification, nous aurions pu allouer 50% du temps disponible pour traiter des plans d'une taille fixe, et les 50% restants pour considérer des versions du problème à planifier où nous avions supprimé des buts ou ajouté de nouvelles actions. Cependant, avec l'approche RECAR nous avons souhaité aller plus loin en considérant la possibilité d'appeler récursivement la procédure, c'est-à-dire que nous utilisons un oracle PSPACE pour décider un problème PSPACE. Si nous poursuivons avec le problème de planification, c'est comme si nous pouvions enlever des buts et alors considérer sur cette sous-approximation des plans d'une taille fixe. Cela permet plus de flexibilité qu'une approche qui simplement commute entre les deux paradigmes de résolution.

Afin de valider expérimentalement notre nouveau schéma de résolution, nous avons considéré le problème de décision qui consiste à déterminer si une formule de la logique propositionnelle modale est satisfaisable (voir la section 2.3 pour plus de détails sur cette logique). Notre première étape a donc été de définir les notions de sur-approximation et de sous-approximation pour une formule de la logique modale. En ce qui concerne les sous-approximations, nous avons considéré un affaiblissement structurel, c'est-à-dire que nous remplaçons certaines parties de la formule par $\top$ (en nous assurant bien sûr que les sous-formules remplacées ne se trouvent pas sous des négations). Pour ce qui est des sur-approximations, nous nous sommes largement inspirés de la planification (Seipp et Helmert 2018), en focalisant la recherche de solutions parmi l'ensemble des structures de Kripke de taille inférieure à une certaine valeur. Nous verrons à la section 6.3 que l'utilisation de ces approximations permet aussi d'exploiter finement le retour du solveur SAT afin d'orienter plus efficacement le processus de raffinement d'approximations.

Ce chapitre est divisé en quatre sections. Dans la première, nous passons rapidement en revue les méthodes de l'état de l'art utilisées pour résoudre pratiquement des problèmes exprimées sous la formule d'une formule de la logique modale. Ensuite, nous présentons nos travaux sur la résolution pratique de fragments NP-complets (section 6.2) et PSPACE-complets (section 6.3) du problème de la satisfaisabilité en la logique modale. Nous concluons enfin ce chapitre et nous donnons quelques perspectives quant aux futures applications et développements de la méthode RECAR.

6.1 Préliminaires

Contrairement à la résolution du problème SAT (en logique classique), où il est clair que les approches CDCL sont aujourd'hui les mieux adaptées pour attaquer en pratique des problèmes structurés, il n'existe pas une approche qui supplante toutes les autres pour résoudre une instance modélisée en logique modale. Il y a principalement deux manières d'envisager la résolution de telles instances : utiliser la méthode des tableaux ou bien s'appuyer sur une transformation incrémentale du problème en logique propositionnelle.

Nous supposons que le lecteur est familier avec la méthode des tableaux sémantiques proposée par Smullyan (1966) pour le cadre de la logique propositionnelle. La méthode des tableaux fonctionne par réfutation, c'est-à-dire que le but n'est pas de montrer la satisfaisalité de la formule donnée, mais de montrer que la négation de cette dernière ne peut pas être satisfaite. Plus précisément, la méthode s'appuie sur une recherche arborescente réalisée par l'application de différentes règles permettant de traiter chaque opérateur ou foncteur. L'application de ces règles ouvre des branches. Ces branches peuvent conduire à des contractions évidentes, et dans

ce cas, la branche est fermée. Si toutes les branches sont fermées alors la preuve est terminée et la formule d'origine est valide. Sinon, quand l'ensemble des règles a été appliqué sur toutes les formules du tableau et qu'il n'est pas possible de le fermer, alors la formule que nous souhaitons réfuter est montrée non valide. En particulier, les branches que nous ne pouvons pas fermer forment un modèle pour l'ensemble de départ. Du point de vue de la réfutation, ces branches peuvent être vues comme des contre-exemples à la validité de la formule de départ.

Parmi les méthodes de déduction par tableaux sémantiques, il est assez commun d'utiliser la méthode de Massacci (2000) qui utilise la notion de formule préfixée, définie une expression de la forme $\sigma : \phi$ avec σ un préfixe (suite finie d'entiers positifs) et ϕ une formule de la logique modale. Nous notons $\sigma_0.\sigma_1$ le préfixe obtenu par la concaténation des préfixes σ_0 et σ_1. Soit un entier n, nous notons $\sigma.n$ la suite σ suivie de n. $\sigma.n$ nomme un monde accessible par l'un des mondes nommé par σ. Pour construire une preuve de ϕ par tableaux sémantiques, nous construisons un arbre dont la racine est étiquetée par la formule préfixée $1 : \neg\phi$, ce qui signifie que le monde nommé par 1 ne satisfait pas la formule ϕ, et les branches de l'arbre sont construites par l'application des règles d'expansion suivantes :

$$\frac{\sigma : (\phi \vee \psi)}{\sigma : \phi \mid \sigma : \psi} \qquad \frac{\sigma : (\phi \wedge \psi)}{\begin{array}{c}\sigma : \psi \\ \sigma : \phi\end{array}} \qquad \frac{\sigma : \neg\neg\phi}{\sigma : \phi} \qquad \frac{\sigma : \Box\phi}{\sigma.n : \phi} \qquad \frac{\sigma : \Diamond\phi}{\sigma.n : \phi \ (n \text{ nouveau})}$$

La sémantique de l'opérateur $\Diamond$ (resp. $\Box$) spécifie que pour la formule $\sigma : \Diamond\phi$ (resp. $\sigma : \Box\phi$), le monde nommé par σ satisfait $\Diamond\phi$ (resp. $\Box\phi$) et il existe au moins un monde accessible (resp. tous les mondes accessibles) à partir du monde dont le nom est σ qui satisfait ϕ (resp. satisfont ϕ). D'après la définition des préfixes, dans le cas du foncteur $\Diamond$, le monde accessible à partir de σ a un nom de la forme $\sigma.n$ à condition que ce nom n'ait jamais été utilisé préalablement (nouveau). Pour ce qui est du foncteur $\Box$, d'après la définition des préfixes, si le préfixe $\sigma.n$ a déjà été utilisé, alors il est le nom d'un monde accessible à partir du monde dont le nom est σ.

L'avantage de la méthode des tableaux est qu'elle est très modulaire. En effet, pour considérer un ensemble d'axiomes spécifiant un cadre, il suffit d'appliquer certaines des règles suivantes :

$$(K)\ \frac{\sigma : \Box\phi}{\sigma.n : \phi} \qquad (T)\ \frac{\sigma : \Box\phi}{\sigma : \phi} \qquad (4)\ \frac{\sigma : \Box\phi}{\sigma.n : \Box\phi}$$

$$(B)\ \frac{\sigma.n : \Box\phi}{\sigma : \phi} \qquad (D)\ \frac{\sigma : \Box\phi}{\sigma : \Diamond\phi} \qquad (5)\ \frac{\sigma : \Diamond\phi}{\sigma.n : \Diamond\phi \ (n \text{ nouveau})}$$

Il n'est pas évident de se rendre compte à première vue que l'application de ces règles permet d'obtenir une méthode qui termine et qui est correcte. Puisque cela ne rentre pas vraiment dans le cadre des travaux qui seront présentés par la suite, nous n'entrerons pas plus dans ces détails dans le cadre de ce manuscrit. Le lecteur intéressé pourra se référer à l'ouvrage de Goré (1999).

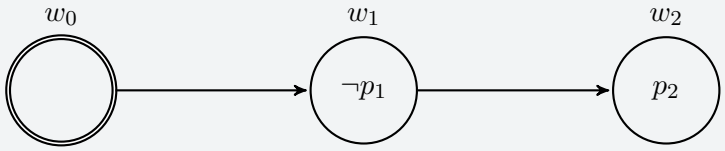
Considérons l'exemple suivant comme illustration du fonctionnement de la méthode des tableaux.

Exemple 32

Démontrons que la formule $\phi = (\Box\neg p_1 \wedge \Diamond(p_1 \vee \Diamond p_2))$ est satisfaisable dans la logique modale K. Par l'application de la méthode des tableaux, nous obtenons :

$1 : \Box\neg p_1 \wedge \Diamond(p_1 \vee \Diamond p_2)$	(1)				
$1 : \Box\neg p_1$	(2) de (1)				
$1 : \Diamond(p_1 \vee \Diamond p_2)$	(3) de (1)				
$1.1 : p_1 \vee \Diamond p_2$	(4) de (3)				
$1.1 : p_1$	(5) de (4)		$1.1 : \Diamond p_2$	(7) de (4)	
$1.1 : \neg p_1$	(6) de (2)		$1.2 : \neg p_1$	(8) de (2)	
$1.1 : \dagger$	(5) et (6)		$1.2.1 : p_2$	(9) de (7)	

Comme il existe une branche que nous ne pouvons pas fermer, nous pouvons conclure que la formule ϕ est satisfaisable. En fait, en considérant les préfixes nous pouvons construire le modèle de Kripke suivant qui satisfait la formule.



Il existe de nombreux solveurs de la littérature qui s'appuient sur l'utilisation de la méthode des tableaux pour résoudre des instances du problème de la satisfaisalité en logique modale (Götzmann et al. 2010, Tsarkov et Horrocks 2006, Gasquet et al. 2005, Schmidt et Tishkovsky 2008, Baader et Hollunder 1991, Balsiger et al. 1998, Abate et al. 2007). L'un des inconvénients de la méthode des tableaux est lié au fait qu'un arbre est développé, et que cet arbre peut être très grand. Afin d'éviter en partie cet écueil, Goré et al. (2014) ont proposé une approche nommée `BDDTab` qui essaie de factoriser certaines parties de l'arbre *via* l'utilisation de `BDD` (voir la section 5.1.2 pour plus d'informations sur ce type de structure).

Une autre manière de résoudre une instance de logique modale consiste à s'appuyer sur le fait que c'est un problème `PSPACE`-complet, et d'utiliser une traduction dans un problème qui est aussi `PSPACE`-complet. Nous avons par exemple `QMRES` (Pan et Vardi 2004) qui transforme l'instance considérée en `QBF`, ou encore `(M)SPASS` (Hustadt et al. 1999, Weidenbach et al. 2009) et `OFT` (Horrocks et al. 2006) qui la transforment en logique du premier ordre.

Bien que `SAT` en logique propositionnelle classique soit `NP`-complet (ce qui implique une explosion de la taille de la transformation si celle-ci est réalisée directement), il existe de nombreuses approches qui s'appuient sur la puissance des solveurs `SAT` afin de résoudre des problèmes modélisés en logique modale. Il y a d'un côté les approches qui utilisent la technologie `SAT` pour guider la recherche d'une solution. C'est le cas de la méthode `InKreSAT` (Kaminski et Tebbi 2013) qui utilise un solveur `SAT` pour guider le développement d'un tableau. Ou encore la méthode `⋆SAT` (Giunchiglia et al. 2002a), qui peut être vue comme l'ancêtre des solveurs `SMT` (Barrett et al. 2009), qui combine raisonnement en logique propositionnelle avec un raisonnement orienté logique modale (la théorie). Cette dernière méthode a été récemment remise au goût du jour par les auteurs du solveur `Moloss` (Delmas et al. 2018), qui utilisent l'encodage `SMT` proposé par Areces et al. (2015).

D'un autre côté, il y a les approches qui essaient de transformer une instance de logique modale dans un formalisme NP-complet de manière directe. C'est par exemple le cas de la méthode KCSP (Brand et al. 2003) qui modélise le problème comme un problème de satisfaction de contraintes (Rossi et al. 2006, Mackworth 1977, Stallman et Sussman 1977b), ou de la méthode $\mathtt{K}_m\mathtt{2SAT}$ (Sebastiani et Vescovi 2009a) qui utilise le formalisme CNF.

Pour terminer, il existe aussi des approches qui s'appuient sur un système de preuve proche de la résolution, c'est par exemple le cas de la méthode K_SP proposée par Nalon et al. (2016b) ou encore de la méthode KX proposée par Voronkov (1999).

Parmi les approches que nous venons de citer, celles qui obtenaient les meilleures performances, lorsque nous avons commencé nos travaux autour de la résolution pratique de problèmes modélisés en logique modale, étaient basées sur la méthode des tableaux. Comme nous avons pu le voir, la méthode des tableaux va implicitement construire un modèle de Kripke durant son processus de résolution, et ce modèle a une structure arborescente. Le fait que les structures générées soient arborescentes est à notre avis quelque chose de très restrictif, ce qui nous a conduit aux travaux présentés dans la suite, qui s'efforcent de construire de manière explicite des modèles de Kripke qui ne sont pas soumis à cette restriction.

6.2 Résolution pratique de S5

Avant d'attaquer les logiques modales PSPACE, attardons-nous quelques instants sur la résolution pratique de problèmes modélisés à l'aide de la logique modale S5, pour laquelle le problème de la satisfaisalité (S5-SAT) d'une formule est NP-complet. Nous avons tenté de résoudre ce type de problèmes en recherchant un modèle dans l'espace des structures de Kripke respectant les axiomes de cadre de la logique S5. Une question s'est tout de suite posée : combien faut-il considérer de mondes pour satisfaire une formule S5 ? En fait, la véritable question est : à partir de quelle valeur de n pouvons-nous être assurés que s'il n'y a pas de structures de Kripke avec n mondes qui satisfait la formule, alors c'est que cette dernière est insatisfaisable ?

Les travaux de Ladner (1977) mettent cette limite au nombre de modalités de la formule. Cette borne peut rapidement être très large et, elle ne prend pas du tout en compte la structure de la formule d'entrée. Afin de pallier ce problème, nous avons proposé, en collaboration avec Thomas Caridroit, Daniel Le Berre, Tiago de Lima et Valentin Montmirail, une nouvelle borne, appelée *diamond degree*, qui est plus fine et qui permet d'envisager la résolution pratique de problèmes de grande taille encodés en logique modale S5.

S'est ensuite posée la question de déterminer si cette borne est assez proche de l'optimum, c'est-à-dire du nombre de mondes minimal nécessaire pour satisfaire une formule de la logique modale S5 satisfaisable. Pour cela, nous avons défini et étudié dans (Lagniez et al. 2018b), en collaboration avec Daniel Le Berre, Tiago de Lima et Valentin Montmirail, le problème MinS5-SAT qui répond à cette question. Nous avons démontré que ce problème pouvait être résolu en considérant un encodage spécifique en CNF de S5 (décrit ci-après), et qu'il est suffisant d'extraire un MSS pour répondre à la question (ce qui n'est pas forcément très intuitif car nous cherchons à minimiser une fonction).

6.2.1 Un encodage CNF pour résoudre S5

Il a été montré dans (Ladner 1977) que si une formule S5 ϕ contenant n modalités est satisfaisable, alors il existe une structure de Kripke satisfaisant ϕ contenant au plus $n+1$ mondes.

Puisque S5-SAT est NP-complet (Ladner 1977), nous savons qu'il existe un algorithme s'exécutant en temps polynomial capable de transformer un problème S5-SAT en une formule CNF qui lui est équisatisfaisable. Partant de cela, nous avons proposé et évalué dans (Caridroit et al. 2017), en collaboration avec Thomas Caridroit, Daniel Le Berre, Tiago de Lima et Valentin Montmirail, un nouvel encodage CNF de S5 qui a donné naissance au solveur S52SAT.

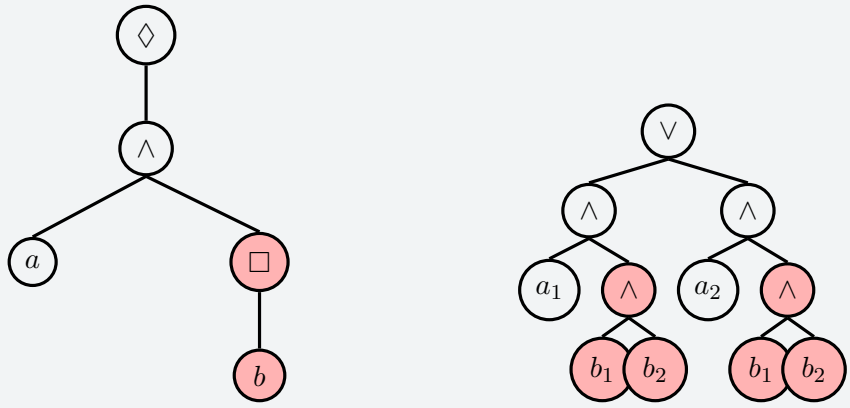
À cet effet, nous nous sommes inspirés de la traduction standard d'une formule modale vers la logique du premier ordre (Blackburn et al. 2006) (voir la section 2.3 pour plus de détails). Cependant, puisque la relation d'accessibilité est similaire à une relation d'équivalence lorsque nous considérons les axiomes de S5, il n'est pas nécessaire de représenter explicitement le cadre. La relation d'équivalence est retranscrite dans le modèle de Kripke par le fait que tous les mondes sont connectés. Ainsi, lorsqu'une formule ϕ est enracinée en un opérateur $\Box$ et que $\Box\phi$ doit être satisfait, ϕ doit être vraie dans tous les mondes. Il est donc suffisant d'interpréter conjonctivement l'opérateur $\Box$, c'est-à-dire que pour chaque monde w du modèle il est suffisant d'encoder le fait que la formule ϕ est vraie dans w. Nous pouvons utiliser un raisonnement similaire pour l'opérateur $\Diamond$, à la différence que dans ce cas nous l'interprétons comme une disjonction. La fonction de traduction tr suivante s'appuie sur ce raisonnement pour encoder une formule de S5 en CNF lorsque le nombre de mondes du modèle de Kripke dans lequel nous recherchons une solution est n.

$$
\begin{aligned}
&\mathrm{tr}(\phi, n) = \mathrm{tr}'(\mathtt{NNF}(\phi), 1, n)\\
&\mathrm{tr}'(\top, i, n) = \top\\
&\mathrm{tr}'(\neg\top, i, n) = \neg\top\\
&\mathrm{tr}'(p, i, n) = p_i\\
&\mathrm{tr}'(\neg p, i, n) = \neg p_i\\
&\mathrm{tr}'((\phi \wedge \cdots \wedge \delta), i, n) = \mathrm{tr}'(\phi, i, n) \wedge \cdots \wedge \mathrm{tr}'(\delta, i, n)\\
&\mathrm{tr}'((\phi \vee \cdots \vee \delta), i, n) = \mathrm{tr}'(\phi, i, n) \vee \cdots \vee \mathrm{tr}'(\delta, i, n)\\
&\mathrm{tr}'(\Box\phi, i, n) = \bigwedge_{j=1}^{n} \mathrm{tr}'(\phi, j, n)\\
&\mathrm{tr}'(\Diamond\phi, i, n) = \bigvee_{j=1}^{n} \mathrm{tr}'(\phi, j, n)
\end{aligned}
$$

La traduction tr′ ajoute de nouvelles variables booléennes p_i à la formule, désignant la valeur de vérité de p dans le monde w_i. Dans cette fonction, le paramètre i représente l'indice du monde. La fonction tr est définie sur une formule en forme normale négative (NNF) par souci de simplicité.

Exemple 33

Soit la formule $\phi = \Diamond(a \wedge \Box b)$, la figure suivante montre la traduction (à droite) de la formule NNF ϕ (à gauche) avec $n = 2$.



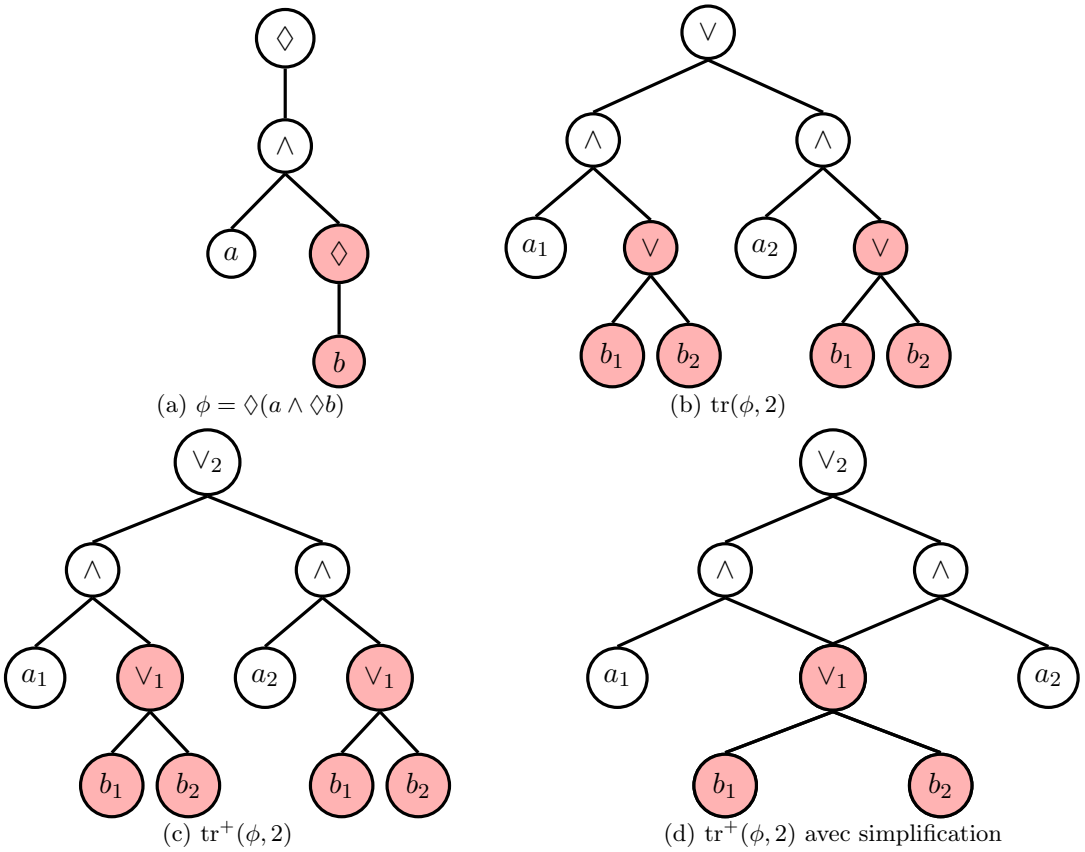
Si la valeur de n est un majorant du nombre de mondes dans le modèle, alors la traduction et la formule S5 d'origine sont équisatisfaisables. C'est en particulier le cas pour la borne $\mathrm{nm}(\phi)$ d'une formule ϕ montrée dans (Ladner 1977), et qui correspond au nombre de connecteurs modaux contenus dans le formule ϕ. Nous avons donc le théorème suivant :

Théoreme 4

Une formule ϕ de S5 est satisfaisable si et seulement si $tr(\phi, \mathrm{nm}(\phi) + 1)$ est satisfaisable.

Le théorème 4 est prouvé de la même façon que la traduction standard vers la logique du premier ordre, en utilisant en plus le lemme 6.1 de Ladner (1977). Notons que le résultat de la traduction n'est pas en CNF. Ainsi, la traduction classique en CNF utilisant l'algorithme de Tseitin (1968) (ou celui de Plaisted et Greenbaum (1986)) est nécessaire pour utiliser ensuite un solveur SAT.

En observant la figure de l'exemple 33, nous pouvons observer que la sous-formule $(b_1 \wedge b_2)$ apparaît deux fois. En fait, la traduction du premier diamant crée deux sous-formules $(a_1 \wedge \Box b)$ et $(a_2 \wedge \Box b)$, où chaque $\Box b$ doit être traduit. Dans cette situation, nous aimerions ne pas avoir à représenter deux fois la même information et nous aimerions donc exploiter toute la puissance de représentation du langage NNF en partageant certaines sous-formules. Une manière de réaliser cela est d'utiliser une table de hachage afin de mémoriser les sous-formules déjà calculées. Ce type de procédure, appelée *caching*, est déjà utilisée dans le solveur de formules de la logique modale *SAT qui considère une « matrice de bits » pour sauvegarder les sous-formules déjà traduites en SAT (Giunchiglia et Tacchella 2001). Comme nous l'avons vu au chapitre précédent, la compilation est aussi un domaine où le *caching* est largement utilisé afin de représenter plus succinctement l'information. Par exemple, les implémentations efficaces de compilateur ROBDD (Bryant 1986) utilisent un processus de *caching* complet, c'est-à-dire que les



(a) $\phi = \Diamond(a \wedge \Diamond b)$ (b) $\mathrm{tr}(\phi, 2)$

(c) $\mathrm{tr}^+(\phi, 2)$ (d) $\mathrm{tr}^+(\phi, 2)$ avec simplification

FIGURE 6.1 – Traduction de $\Diamond(a \wedge \Diamond b)$ (a), traduction initiale (b), formule marquée (c), traduction finale (d)

formules sont partagées à l'équivalence logique près. Sans aller jusque là, nous proposons une technique « simple mais efficace » qui permet de réduire la taille de la formule obtenue après codage.

Notre approche tire parti du fait qu'en S5 tous les mondes sont connectés, ce qui implique que les traductions de $\Box b$ sur des mondes différents sont équivalentes, et donc que nous pouvons réutiliser la même traduction. Cela vient du fait que lorsque nous traduisons en CNF les modalités le résultat est indépendant du monde dans lequel « nous nous trouvons ». En effet, nous avons $\mathrm{tr}'(\Box\phi, i, n) = \bigwedge_{k=1}^{n} \mathrm{tr}'(\phi, k, n))$ et $\mathrm{tr}'(\Diamond\phi, i, n) = \bigvee_{k=1}^{n} \mathrm{tr}'(\phi, k, n)$ qui ne font pas intervenir i. Ainsi, quelle que soit la façon dont est incorporée la sous-formule modale considérée dans la formule entière, sa traduction donnera toujours le même résultat (indépendamment de l'indice i). Par conséquent, nous pouvons commencer par traduire une sous-formule enracinée à une profondeur maximale dans la formule entière, marquer le nœud correspondant puis revenir en arrière. La formule résultante peut contenir plusieurs nœuds ayant le même marquage. Cela signifie que ces sous-formules sont syntaxiquement identiques (voir la figure 6.1.c). Ensuite, nous ne conservons qu'une occurrence de la sous-formule, transformant l'arbre en un DAG (voir figure 6.1.d). Le *caching* structurel est donc effectué à la volée avant de traduire en CNF. La fonction de traduction utilisant cette technique est notée tr^+.

Un autre élément important concernant la taille de la traduction est le nombre de mondes que va contenir la structure de Kripke que nous souhaitons construire. Dans notre traduction,

cette taille dépend du paramètre $\mathrm{nm}(\phi)$. Généralement, ce majorant produit des formules CNF très grandes, qui ne peuvent pas être obtenues dans des temps raisonnables. En fait, le nombre de modalités ne reflète pas complètement la structure de l'instance et surtout est sans rapport vis-à-vis de la sémantique de S5. Par exemple, l'interprétation d'un $\Box$ ne nécessite pas forcément la création d'un nouveau monde, puisque la formule qui est enracinée en cette modalité doit être vraie dans tous les mondes et que ces derniers sont connectés. De même, lorsque nous considérons une disjonction entre deux formules ϕ_1 et ϕ_2, il n'est pas nécessaire de considérer suffisamment de mondes pour satisfaire ϕ_1 **et** ϕ_2 (interprétés comme une somme) mais pour satisfaire ϕ_1 **ou** ϕ_2 (interprété comme un min). De ces observations, nous avons défini une nouvelle mesure, appelée *diamond degree*, définie de la manière suivante :

Définition 77 (*Diamond degree*)

Le *diamond degree* d'une formule ϕ de $\mathcal{L}$, noté $\mathtt{dd}(\phi)$, est défini récursivement comme suit :

$$\begin{aligned}
\mathtt{dd}(\phi) &= \mathtt{dd}'(\mathtt{NNF}(\phi)) \\
\mathtt{dd}'(\top) &= 0 \\
\mathtt{dd}'(\neg\top) &= 0 \\
\mathtt{dd}'(p) &= 0 \\
\mathtt{dd}'(\neg p) &= 0 \\
\mathtt{dd}'(\phi \wedge \psi) &= \mathtt{dd}'(\phi) + \mathtt{dd}'(\psi) \\
\mathtt{dd}'(\phi \vee \psi) &= \max(\mathtt{dd}'(\phi), \mathtt{dd}'(\psi)) \\
\mathtt{dd}'(\Box\phi) &= \mathtt{dd}'(\phi) \\
\mathtt{dd}'(\Diamond\phi) &= 1 + \mathtt{dd}'(\phi)
\end{aligned}$$

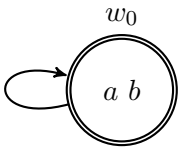
Cette mesure nécessite dans un premier temps de convertir ϕ en NNF. Cette opération est nécessaire afin de s'assurer que les opérateurs et modalités que nous sommes en train de considérer ne sont pas altérés par une négation. Comme nous l'avons mentionné à la section 2.3, cette opération peut être effectuée en temps polynomial. Ainsi, le *diamond degree* de toute formule peut être calculé en temps polynomial. Sans perte de généralité, nous supposerons que ϕ est en NNF et considérerons $\mathtt{dd}'$ au lieu de dd. La preuve que le *diamond degree* de ϕ est bien un majorant du nombre de mondes nécessaires à la satisfaction est donnée dans (Caridroit et al. 2017).

Une question que nous sommes en droit de nous poser est alors : est-ce que le *diamond degree* représente bien le nombre de mondes minimal nécessaires afin de satisfaire une formule S5 ? Afin de répondre à cette question, nous définissons et étudions dans la sous-section suivante le problème d'optimisation qui consiste à déterminer combien de mondes il est nécessaire de considérer pour satisfaire une formule S5.

6.2.2 Le problème MinS5-SAT

Le majorant que nous avons proposé à la sous-section précédente, souvent bien que plus fin que celui proposée par Ladner (1977), reste assez « basique », dans le sens où il peut être loin de la valeur minimale. Considérons par exemple la formule modale $\phi = ((\Box\neg a \vee \Diamond b) \wedge \Diamond a \wedge \Box b)$.

D'après le résultat énoncé précédemment, au sujet du *diamond degree*, cette formule peut être satisfaite avec 3 mondes. Cependant, il est facile de vérifier que le modèle suivant, contenant un seul monde, est un modèle de ϕ :



Afin d'étudier empiriquement combien il est nécessaire de considérer de mondes pour satisfaire une formule S5, nous avons défini et étudié dans (Lagniez et al. 2018b), en collaboration avec Daniel Le Berre, Tiago de Lima et Valentin Montmirail, le problème MinS5-SAT qui répond à cette question. Formellement :

Définition 78 (Satisfaisabilité minimale en S5)

Une formule ϕ est MinS5-satisfaite par une structure $\langle M, w\rangle$ (notée $\langle M, w\rangle \models_{\min} \phi$) si et seulement si $\langle M, w\rangle \models \phi$ et ϕ n'a pas de modèle $\langle M', w'\rangle$ tel que $|M'| < |M|$.

Définition 79 (Problème de satisfaisabilité minimale en S5)

Le problème de satisfaisabilité minimale en S5 (MinS5-SAT) est le suivant : étant donné ϕ une formule de $\mathcal{L}$, trouver une structure $\langle M, w\rangle$ telle que $\langle M, w\rangle \models_{\min} \phi$.

Remarquons que calculer un modèle minimal de ϕ n'est pas aussi simple que de fusionner les mondes avec les mêmes valuations en un seul monde dans n'importe quel modèle de ϕ. La minimalité ne peut être garantie de cette façon. Revenons à l'exemple précédent, où $\phi = ((\Box\neg a \vee \Diamond b) \wedge \Diamond a \wedge \Box b)$, pour illustrer cela. La structure de Kripke $\langle M, w_0\rangle$, telle que $W = \{w_0, w_1, w_2\}$, $\mathcal{I} = \{\langle a, \{w_0, w_2\}\rangle, \langle b, \{w_0, w_1, w_2\}\rangle\}$, est un modèle de ϕ avec $\mathtt{dd}(\phi) + 1 = 3$ mondes. Si nous supprimons la redondance, nous ne conservons que les deux mondes w_0 et w_1, conduisant à un modèle de taille 2. Cependant, nous avons vu que ϕ est également satisfaite par une structure avec un seul monde : $W = \{w_0\}$, $\mathcal{I} = \{\langle a, \{w_0\}\rangle, \langle b, \{w_0\}\rangle\}$, qui est un modèle minimal.

Un moyen très simple de résoudre le problème MinS5-SAT consiste à utiliser le codage $\mathrm{tr}'(\phi, b)$ défini dans la sous-section précédente avec une stratégie de recherche linéaire. Plus précisément, la procédure commence par essayer des structures de taille $b = 1$. Si aucun modèle n'est trouvé pour $\mathrm{tr}'(\phi, b)$, nous incrémentons la valeur de b. Nous itérons ainsi jusqu'à ce qu'un modèle de ϕ soit trouvé ou que la borne $\mathtt{dd}(\phi) + 1$ soit atteinte. Cette stratégie est appelée 1toN dans la suite. Il est bien sûr également possible d'effectuer la recherche dans l'ordre inverse : la procédure commence par $b = \mathtt{dd}(\phi) + 1$ et décrémente la valeur de b. Une autre possibilité consiste à utiliser une recherche dichotomique.

Cependant, ces approches sont très naïves. Prenons par exemple 1toN. Lorsque la solution optimale est un modèle de taille m, l'approche effectuera m traductions de S5 vers SAT puis m

appels à un solveur SAT. Le problème est qu'une telle stratégie ne profite pas de la réponse UNSAT précédente du solveur pour résoudre la formule à l'étape suivante. Pour pallier ce problème, nous proposons d'exploiter la notion de CORE présentée à la section 4.2, et qui permet à un solveur d'expliquer la raison de l'incohérence d'une formule CNF. Pour réaliser cela, nous proposons de « décorer » avec des variables-sélecteurs la NNF produite lors de la traduction.

Nous modifions ainsi la fonction de traduction de manière à considérer des variables-sélecteurs s_i pour activer ou désactiver l'accessibilité aux mondes w_i. Le changement concerne principalement la traduction de $\Box\psi$, qui devient $\bigwedge_{i=1}^{n}(s_i \Rightarrow \mathrm{tr}(\psi, i, n))$, et celle de $\Diamond\psi$, qui devient $\bigvee_{i=1}^{n}(s_i \wedge \mathrm{tr}(\psi, i, n))$. La taille du modèle S5 sera alors déterminée par le nombre de variables-sélecteurs à vrai. La fonction de traduction complète est donnée ci-dessous :

Définition 80 (Traduction sous hypothèses d'une formule de S5)

Soit $\phi \in \mathcal{L}$. Nous avons :

$$\begin{aligned}
&\mathrm{tr_s}(\phi, n) = \mathrm{tr_s}'(\phi, 1, n) \quad \mathrm{tr_s}'(p, i, n) = p_i \\
&\mathrm{tr_s}'(\neg\psi, i, n) = \neg\,\mathrm{tr_s}'(\psi, i, n) \\
&\mathrm{tr_s}'(\psi \wedge \delta, i, n) = \mathrm{tr_s}'(\psi, i, n) \wedge \mathrm{tr_s}'(\delta, i, n) \\
&\mathrm{tr_s}'(\psi \vee \delta, i, n) = \mathrm{tr_s}'(\psi, i, n) \vee \mathrm{tr_s}'(\delta, i, n) \\
&\mathrm{tr_s}'(\Box\psi, i, n) = \bigwedge_{j=1}^{n}(\neg s_j \vee (\mathrm{tr_s}'(\psi, j, n)) \\
&\mathrm{tr_s}'(\Diamond\psi, i, n) = \bigvee_{j=1}^{n}(s_j \wedge (\mathrm{tr_s}'(\psi, j, n))
\end{aligned}$$

Un modèle S5 doit contenir au moins un monde possible (le monde courant). Sans perte de généralité, nous considérons que le monde courant est le numéro 1, donc s_1 est toujours vraie. Dans la suite, nous notons $\mathrm{tr_s}(\phi)$ la formule $\mathrm{tr_s}(\phi, \mathtt{dd}(\phi) + 1) \wedge s_1$ et l'ensemble de tous les sélecteurs de $\mathrm{tr_s}(\phi)$ est donné par $\mathcal{S}(\phi)$, c'est-à-dire, $\mathcal{S}(\phi) = \{s_i \mid 1 \leq i \leq \mathtt{dd}(\phi) + 1\}$.

Exemple 34 (Exemple de '$\mathrm{tr_s}$')

Considérons à nouveau la formule $\phi = ((\Box\neg a \vee \Diamond b) \wedge \Diamond a \wedge \Box b)$. Sa traduction $\mathrm{tr_s}(\phi, 3)$ est :

$$\begin{aligned}
&(\neg s_1 \vee (\neg a_1 \vee (s_1 \wedge b_1) \vee (s_2 \wedge b_2) \vee (s_3 \wedge b_3))) \wedge \\
&(\neg s_2 \vee (\neg a_2 \vee (s_1 \wedge b_1) \vee (s_2 \wedge b_2) \vee (s_3 \wedge b_3))) \wedge \\
&(\neg s_3 \vee (\neg a_3 \vee (s_1 \wedge b_1) \vee (s_2 \wedge b_2) \vee (s_3 \wedge b_3))) \wedge \\
&((s_1 \wedge a_1) \vee (s_2 \wedge a_2) \vee (s_3 \wedge a_3)) \ \wedge ((\neg s_1 \vee b_1) \wedge \\
&(\neg s_2 \vee b_2) \wedge (\neg s_3 \vee b_3)) \ \wedge s_1
\end{aligned}$$

Intuitivement, chaque formule avec un indice i est une formule qui est vraie dans le monde possible i. Si la variable-sélecteur s_i est fausse, alors le monde i n'est pas présent dans le modèle. Ci-dessous, une formule équivalente à $\mathrm{tr_s}(\phi, 3)$ mais avec s_1 et s_2 activées et s_3 désactivée.

$$\begin{aligned}
&(\neg\top \vee (\neg a_1 \vee (\top \wedge b_1) \vee (\top \wedge b_2) \vee (\bot \wedge b_3))) \wedge \\
&(\neg\top \vee (\neg a_2 \vee (\top \wedge b_1) \vee (\top \wedge b_2) \vee (\bot \wedge b_3))) \wedge \\
&(\neg\bot \vee (\neg a_3 \vee (\top \wedge b_1) \vee (\top \wedge b_2) \vee (\bot \wedge b_3))) \wedge \\
&((\top \wedge a_1) \vee (\top \wedge a_2) \vee (\bot \wedge a_3)) \ \wedge ((\neg\top \vee b_1) \wedge \\
&(\neg\top \vee b_2) \wedge (\neg\bot \vee b_3)) \ \wedge \top
\end{aligned}$$

Cette formule est équivalente à $(\neg a_1 \vee b_1 \vee b_2) \ \wedge (\neg a_2 \vee b_1 \vee b_2) \ \wedge (a_1 \vee a_2) \ \wedge (b_1 \wedge b_2)$. Cela correspond au problème de décider si ϕ est satisfaisable dans un modèle avec 2 mondes.

Comme nous pouvons le comprendre, le problème de résoudre le problème de satisfaisabilité minimale en logique modale S5 est maintenant équivalent au problème de satisfaire $\mathrm{tr_s}(\phi, n)$ et de minimiser le nombre de s_i, pour $i > 1$, affectées à vrai (ou, de manière équivalente, maximiser le nombre de s_i affectées à faux). Évidemment, cela peut être vu comme un problème d'optimisation pseudo-booléenne (PBO) (Biere et al. 2009), où la fonction d'optimisation à minimiser est le nombre de variables-sélecteurs assignées à vrai. Ce problème est également souvent résolu de nos jours sous la forme d'une instance du problème MaxSAT partiel (Biere et al. 2009), qui consiste à satisfaire toutes les *hard clauses* et le nombre maximum de *soft clauses*. Dans notre cas, les *hard clauses* sont celles générées par la fonction de traduction, et les *soft clauses* sont les clauses unitaires $\{\neg s_i \mid s_i \in \mathcal{S}(\phi) \text{ et } i > 1\}$ construites à partir des variables-sélecteurs.

Ainsi, nous pouvons utiliser des solveurs MaxSAT partiels et des solveurs PBO pour résoudre MinS5-SAT. Cependant, le fait que les mondes soient interchangeables nous permet de nous focaliser sur la recherche d'un MSS maximal pour l'inclusion et non pas maximal pour la taille (voir section 4.1 pour plus de détails sur cette notion). Cette interchangeabilité des mondes s'exprime au niveau de la formule générée par $\mathrm{tr_s}(\phi, n)$ à travers la proposition suivante :

Proposition 4

Soit $\psi = \mathrm{tr_s}(\phi, n)$, et soit χ une formule du type $\bigwedge_{s_i \in \mathcal{S}'} \neg s_i$, où $\mathcal{S}' \subseteq \mathcal{S}(\phi)$. Si $(\Sigma \wedge \chi)$ est satisfaisable alors la formule $(\Sigma \wedge \chi')$ l'est aussi, où χ' est obtenu depuis χ en remplaçant les occurrences d'une variable-sélecteur $s \in \mathcal{S}'$ par une autre $s' \in \mathcal{S}(\phi) \setminus \mathcal{S}'$.

Il est facile de prouver par récurrence sur la longueur de la formule ϕ que cette proposition est correcte. Le cas de base est du type, $\phi = p$, avec p une variable propositionnelle. Nous avons alors $\psi = p_1$, $\chi = \neg s_1$ et $\mathcal{S}(\phi) = \{s_1\}$, ce qui implique que l'affirmation est vraie (car $\mathcal{S}(\phi) \setminus \mathcal{S}' = \varnothing$). Nous avons plusieurs cas d'hérédité de récurrence. Puisque leurs preuves sont toutes semblables, nous n'en montrons qu'une seule ici. Soit $\phi = \Box\phi'$. Nous avons $\psi = \bigwedge_{i=1}^{n}(\neg s_i \vee \mathrm{tr_s}(\phi', i, n))$, $\chi = \bigwedge_{s_i \in S'} \neg s_i$, et $\mathcal{S}(\phi) = \{s_1, \ldots, s_n\}$. Soit χ', obtenue par χ où s_i est remplacée par $s_j \in \mathcal{S}(\phi) \setminus S'$. Si $(\psi \wedge \chi)$ est satisfaite par un modèle M alors nous construisons un nouveau modèle M'. Ce modèle est égal à M sauf que les valeurs de vérité de toutes les variables propositionnelles avec l'indice i sont les mêmes que celles avec l'indice j. Nous avons immédiatement que si $M \models \chi$ alors $M' \models \chi'$. Nous avons aussi que si $M \models \neg s_i$ alors $M' \models \neg s_j$. Enfin, pour chaque $1 \leq i \leq n$, si $M \models \mathrm{tr_s}(\phi', i, n))$ alors $M' \models \mathrm{tr_s}(\phi', j, n)$, par hypothèse de récurrence (puisque la longueur de ϕ' est strictement inférieure à celle de ϕ). Par conséquent, $M' \models \psi \wedge \chi'$.

Ainsi, bien qu'en général, un `MSS` partiel n'est pas une solution à un problème `MaxSAT` partiel, nous pouvons montrer que dans le cas spécifique de `MinS5-SAT`, un `MSS` partiel est également une solution au problème `MaxSAT` partiel correspondant, ce qui implique que l'optimalité pour l'inclusion (`MSS`) est équivalente à l'optimalité pour la taille (`MaxSAT`). Cela est exprimé à travers la proposition suivante :

Proposition 5

Soit Σ une formule en `CNF` équisatisfaisable à $\mathrm{tr_s}(\phi, \mathtt{dd}(\phi) + 1)$, et soit χ une formule du type $\bigwedge_{i=2}^{\mathtt{dd}(\phi)+1} \neg s_i$. Un `MSS` de $(\Sigma \wedge \chi)$, où Σ est l'ensemble des *hard clauses*, est aussi solution du problème `MaxSAT` partiel pour l'entrée $\Sigma \wedge \chi$.

En nous appuyant sur la proposition 4 et en raisonnant par l'absurde nous pouvons aisément montrer que cette proposition 5 est correcte. En effet, supposons qu'il existe un `MSS` $\delta_1 = (\psi \wedge \chi_1)$, où $\chi_1 = \bigwedge_{s \in S_1} \neg s$, qui n'est pas le plus grand. Alors il existe un autre `MSS` $\delta_2 = (\psi \wedge \chi_2)$, où $\chi_2 = \bigwedge_{s \in S_2} \neg s$ et tel que $|S_1| < |S_2|$. Maintenant, soit $S_3 = S_2 \setminus \{s\} \cup \{s'\}$, où $s \in S_2$ et $s' \in S_1$. Par la proposition 4, la formule $\delta_3 = (\psi \wedge \chi_3)$, où $\chi_3 = \bigwedge_{s \in S_3} \neg s$ est satisfaisable, car elle est équivalente à δ_2 avec l'une des variables-sélecteurs de S_2 dans χ_2 remplacée par une autre variable-sélecteur. Nous pouvons continuer à remplacer itérativement les sélecteurs dans cet ensemble jusqu'à ce que nous ayons obtenu l'ensemble S_k, tel que $S_1 \subseteq S_k$. La formule $\delta_k = (\psi \wedge \chi_k)$, où $\chi_k = \bigwedge_{s \in S_k} \neg s$ est satisfaisable, en appliquant la proposition 4 $|S_1|$ fois. Par conséquent δ_k est un `MSS` qui contient δ_1, ce qui contredit l'hypothèse. Cela signifie que chaque `MSS` de la formule initiale est maximal. Par conséquent, chaque `MSS` de $(\psi \wedge \chi)$ est aussi solution du problème `MaxSAT` partiel.

La proposition 5 implique que nous pouvons toujours trouver un MSS de $\Sigma \wedge \chi$ tel que les indices des variables-sélecteurs utilisés dans cette formule sont contigus. En conséquence, nous pouvons envisager une optimisation qui réduit l'espace de recherche (en cassant les symétries), en ajoutant la contrainte qui suit :

$$(\bigwedge_{i=1}^{n-1} \neg s_i \Rightarrow \neg s_{i+1}) \tag{6.1}$$

En donnant en entrée $\mathrm{tr}_{\mathrm{s}}(\phi, n)$ ainsi que $\mathcal{S}(\phi)$, nous pouvons résoudre le problème MinS5-SAT avec un solveur MaxSAT ou PBO. Si nous ajoutons également la contrainte 6.1 à l'entrée, nous pouvons utiliser un extracteur de MSS. Cependant, il est possible d'aller encore plus loin dans l'amélioration des performances de la méthode de résolution du problème MinS5-SAT en considérant une approche dédiée, utilisant un solveur SAT incrémental et en exploitant les noyaux incohérents.

Considérons l'exemple suivant : soit ϕ la formule en entrée et soit $\mathtt{dd}(\phi) + 1 = 10$. Nous traduisons ϕ en utilisant les variables-sélecteurs et nous cherchons un modèle pour cette formule. Supposons que, après un certain temps de calcul, nous concluons que 4 mondes ne peuvent pas être désactivés complètement, c'est-à-dire que si les variables-sélecteurs s_i, s_j, s_k et s_l sont affectées à faux, une incohérence apparaît. Nous pouvons déduire de cette information que nous avons besoin d'au moins 7 mondes pour trouver un modèle S5 à ϕ. Cela vient du fait que les « 4 mondes qui ne peuvent pas être complètement désactivés » peuvent correspondre, en fait, à n'importe quel groupe de 4 mondes. Cette situation est exprimée dans la proposition suivante :

Proposition 6

Si C est un noyau incohérent de $\mathrm{tr}_{\mathrm{s}}(\phi)$ sous les hypothèses $\mathcal{S}(\phi)$ alors n'importe quel ensemble de littéraux $C' = \{\neg s \mid s \in \mathcal{S}(\phi)\}$ tel que $|C'| = |C|$ est un noyau incohérent de ϕ.

La proposition 6 peut facilement être pouvée en raisonnant par l'absurde. En effet, supposons que C soit un noyau incohérent de $\mathrm{tr}_{\mathrm{s}}(\phi)$ sous les hypothèses $\mathcal{S}(\phi)$. Donc $(\phi \wedge \bigwedge_{l \in C} l)$ est incohérent. Maintenant, raisonnons par l'absurde, supposons aussi qu'il existe un ensemble $C' = \{\neg s \mid s \in \mathcal{S}(\phi)\}$ tel que $|C'| = |C|$ et $(\phi \wedge \bigwedge_{s \in C} \neg s)$ est satisfaisable. Par la proposition 4, nous obtenons un ensemble D à partir de C' en remplaçant les variables-sélecteurs dans C' par celles de C telles que $(\phi \wedge \bigwedge_{s \in D} \neg s)$ est satisfaisable. Comme $D = C$, nous obtenons une contradiction. Par conséquent, tout ensemble de littéraux C' obtenu ainsi est un noyau incohérent de ϕ.

De cette proposition, nous dérivons la proposition 7, qui exprime le fait que si un groupe de m sélecteurs forme un noyau incohérent et que la borne est égale à n, alors tout modèle S5 de la formule d'entrée a au moins $(n - m + 1)$ mondes.

Proposition 7

Soit $\phi \in \mathcal{L}$ tel que $\mathtt{dd}(\phi) + 1 = n$. Si C est un noyau incohérent de $\mathrm{tr}_{\mathrm{s}}(\phi)$ sous les hypothèses $\mathcal{S}(\phi)$ alors $\mathrm{tr}_{\mathrm{s}}(\phi, n')$ est insatisfaisable pour tout $n' \in \{1, \ldots, (n - |C|)\}$.

La preuve de cette proposition est comme suit. Supposons que la formule est satisfaisable avec n mondes. Le solveur SAT renvoie un noyau incohérent C de taille m. Donc l'une des variables-sélecteurs doit être affectée à vrai. Mais en raison de la proposition 6, nous devons mettre au moins une variable-sélecteur à vrai pour tous les noyaux incohérents possibles de taille m. Dit autrement, nous devons avoir $(n - m + 1)$ variables-sélecteurs vraies ensemble faute de quoi la formule est nécessairement incohérente. Cela signifie aussi que $\forall b' \in [1 \dots (n - m)]$ $\mathrm{tr_s}(\phi, b')$ est incohérent.

En utilisant cette propriété, il est donc possible de construire un algorithme itératif basé sur un solveur `SAT` incrémental. Le solveur `SAT` va être capable de retourner un noyau incohérent, et en interprétant ce noyau dans le cadre de la logique modale `S5` en exploitant la proposition 7, nous pourrons affiner la borne utilisée dans la traduction. Une telle technique est présentée dans l'algorithme 6.1. La procédure commence par essayer des structures de taille $b = 1$. Si aucun modèle n'est trouvé, le processus est itéré en augmentant chaque fois la valeur de b qui est remplacée par $(\mathtt{dd}(\phi) + 1 - |s| + 1)$ (où $|s|$ est la taille du noyau incohérent). Nous itérons jusqu'à ce qu'un modèle de ϕ soit trouvé ou que la borne supérieure $\mathtt{dd}(\phi) + 1$ soit atteinte. Notez que $|s|$ diminue strictement à chaque étape, car nous augmentons strictement le nombre de variables-sélecteurs satisfaites. La procédure `solver` qui apparaît dans l'algorithme est le solveur `SAT Glucose` présenté dans (Audemard et al. 2013a). La procédure `getS5Model`() est une procédure qui génère le modèle S5 à partir du modèle propositionnel donné en argument. Nous présentons seulement l'approche $1toN_c$, l'approche $Dicho_c$ est similaire.

Algorithme 6.1 : `S52SAT 1toN`$_c$

Data : $\phi \in \mathcal{L}$
Result : $\langle M, w\rangle$ t.q $\langle M, w\rangle \models_{min} \phi$, sinon `UNSAT`

```
1 b ← 1 ;
2 ⟨r, s⟩ ← solver(tr_s(φ, b)) ;
3 n ← dd(φ) + 1 ;
4 while (r ≠ SAT ∧ (b ≤ n)) do
5 |   b ← (n − |s| + 1) ;
6 |   ⟨r, s⟩ ← ( tr_s(φ, b)) ;
7 if (r ≠ SAT) then return UNSAT else
8 |   ⟨M, w⟩ ← getS5Model(s) ;
9 |   return ⟨M, w⟩;
```

Nous avons commencé cette section en critiquant la qualité de la borne utilisée pour notre transformation en `CNF` d'une formule de `S5`. Mais qu'en est-il en pratique ? Afin de mesurer cela empiriquement, nous avons considéré un ensemble de problèmes standards généralement utilisés pour évaluer les solveurs de logique modale. Cependant, malgré nos recherches, nous n'avons pas pu trouver de *benchmarks* pour la logique modale `S5`. Donc, nous avons choisi d'utiliser les *benchmarks* suivants pour les logiques modales `K`, `KT` et `S4` : `TANCS-2000 QBF` (`MQBF`) (Massacci et Donini 2000) ; `LWB K`, `KT` et `S4` (Balsiger et al. 2000), avec 56 formules choisies parmi chacune des 18 classes paramétrées, générées à partir du script donné par les auteurs de (Nalon et al. 2016a) ; et les benchmarks `3CNF`$_{\mathtt{KSP}}$ générés aléatoirement (Patel-Schneider et Sebastiani 2003) de profondeur modale égale à 1 et 2. Nous avons gardé seulement les benchmarks satisfaits dans leur logique d'origine.

La figure 6.2 illustre sur un nuage de dispersion, la différence de nombre de mondes minimal

nécessaire pour satisfaire une formule S5 (axe des ordonnées) par rapport à la valeur retournée par dd (axe des abscisses). Comme nous pouvons l'observer pour de nombreuses instances la valeur dd est largement supérieure à la valeur MinS5-SAT.

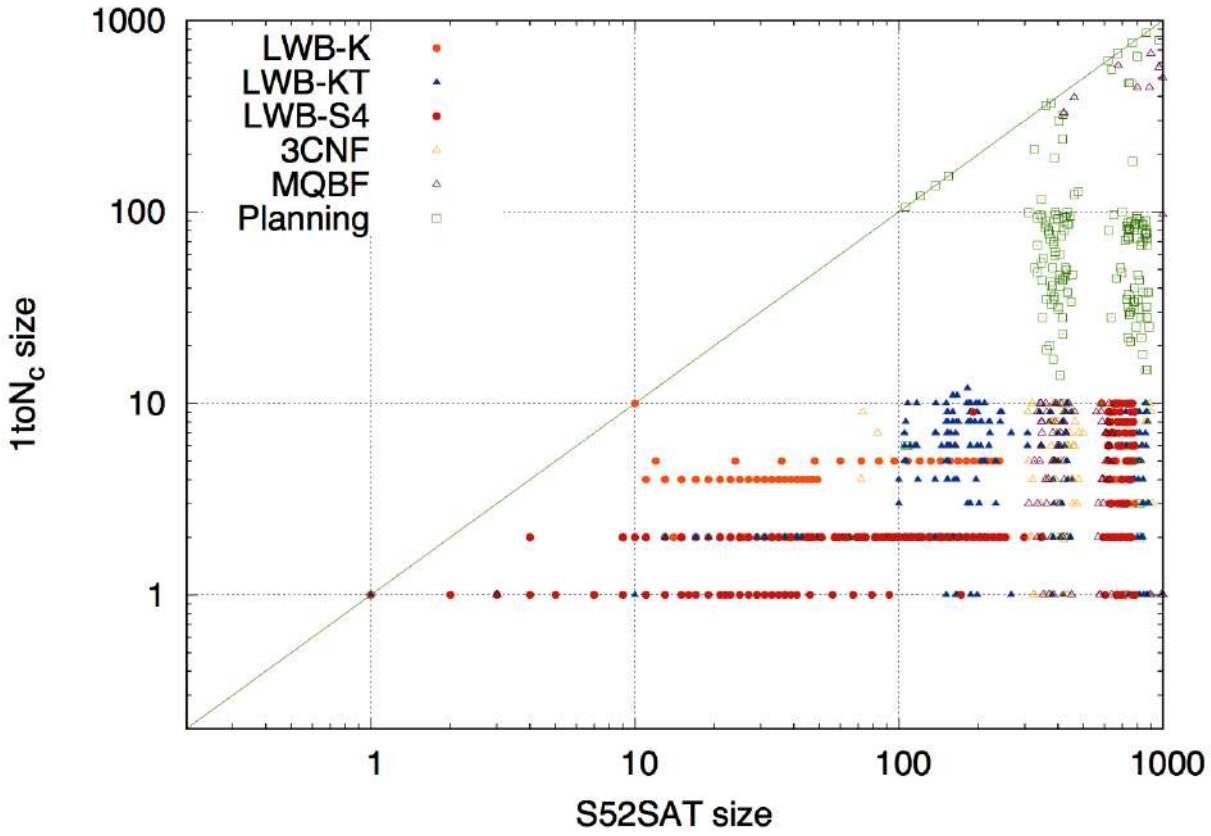


FIGURE 6.2 – Nuage de dispersion permettant de comparer le nombre de mondes minimal nécessaires pour satisfaire une formule S5 (axe des ordonnées) par rapport à la valeur retournée par dd (axe des abscisses).

6.2.3 Discussion

Dans cette section, nous avons étudié la résolution pratique du problème S5-SAT. Plus précisément, nous avons présenté un nouvel encodage pour résoudre ce problème en utilisant un solveur SAT. Cet encodage est basé sur une réduction de S5-SAT à SAT avec deux améliorations par rapport à l'existant : une meilleure borne sur le nombre de mondes requis et un *caching* structurel. Les expérimentations que nous avons conduites dans (Caridroit et al. 2017) ont permis de montrer que notre approche était efficace, comparativement aux approches de l'état de l'art, sur un large éventail de *benchmarks*.

La définition de cette nouvelle borne nous a conduit à réfléchir au problème d'optimisation, que nous avons nommé satisfaisabilité minimale en S5 (MinS5-SAT), et qui consiste à déterminer combien de mondes au minimum il est nécessaire de considérer pour satisfaire une formule S5 satisfaisable. Nous avons démontré que ce problème peut être réduit au problème de l'extraction d'une ensemble de clauses de satisfaction maximale (MSS) et peut donc être résolu avec un extracteur de MSS ou un solveur PBO ou MaxSAT. Nous avons également montré que, grâce à une

propriété inhérente à la logique modale S5, ce problème peut aussi être résolu en utilisant des noyaux incohérents dans une procédure SAT incrémentale. Les expérimentations que nous avons conduites dans (Lagniez et al. 2018b) ont montré que cette dernière approche est la plus efficace en pratique.

Des travaux récents, réalisés par Huang et al. (2019), ont permis d'améliorer la borne que nous avons proposée. Pour l'obtenir, les auteurs commencent par réécrire la formule modale sous une forme clausale. Puis, après certaines simplifications, ils représentent sous la forme d'un graphe les interactions entre les différentes modalités d'une formule ϕ, et montrent que s'il existe un coloriage du graphe avec k couleurs alors il est suffisant de considérer k mondes pour satisfaire la formule ϕ. Les auteurs ont montré dans leur papier qu'utiliser cette borne permet d'améliorer sensiblement les performances de notre approche. Le fait que les auteurs aient réussi à améliorer la borne ne nous surprend pas vraiment. En fait, les résultats que nous avons reportés dans la figure 6.2 montrent clairement que le *diamond degree* n'est pas assez fin. Il serait intéressant de comparer les bornes obtenues par les auteurs aux valeurs optimales que nous avons trouvées. Cela pourrait indiquer s'il est encore possible de définir une meilleure borne ou si cette nouvelle borne est suffisante.

Bien que l'amélioration de la borne puisse permettre de résoudre plus d'instances en pratique, cela ne permet pas d'attaquer de très gros problèmes, qui, par construction, nécessitent beaucoup de mondes. Pour pallier ce problème, une approche pourrait être d'exploiter la structure de la formule S5 considérée. Plus précisément, il pourrait être envisageable de résoudre des sous-formules et en fonction de leurs satisfaisabilités remplacer certaines parties par $\bot$ ou $\top$. Afin de choisir au mieux les parties à tester, nous pourrions sélectionner en priorité les sous-formules qui conduisent à une forte augmentation du nombre de mondes nécessaire pour coder de manière équisatisfaisable le problème S5 en CNF.

Un réel problème concernant nos expérimentations est le fait qu'il n'y a pas vraiment de *benchmarks* disponibles pour évaluer au mieux nos solveurs. Même si nous avons proposé dans (Lagniez et al. 2018b) des *benchmarks* de planification encodés en S5, cela reste très insuffisant. Bien que les *benchmarks* de logiques modales K, KT et S4 aient une structure intéressante, ils ont été proposés pour évaluer les solveurs pour des logiques modales PSPACE. Cependant, il y a dans la littérature des travaux qui traitent de la modélisation de problèmes en S5 (Salhi et al. 2012, Salhi et Sioutis 2015).[15] Ainsi, dans l'optique de constuire une librairie de problèmes S5, nous envisageons dans un futur proche d'encoder certains de ces problèmes et de les rendre disponibles à la communauté.

Même si les *benchmarks* que nous avons considérés peuvent ne pas être représentatifs de l'utilisation de S5 en pratique, puisqu'ils proviennent d'autres logiques modales (K, KT et S4), les résultats ouvrent des perspectives intéressantes quant à l'utilisation de notre solveur en prétraitement. En effet, prouver la satisfaisabilité d'une formule de la logique modale S5 implique que la formule est également satisfaisable dans des systèmes moins restrictifs (c'est-à-dire dans K, KT et S4). Puisque notre solveur S5 fournit un modèle en quelques secondes (temps médian de 2,06 s dans nos expérimentations), nous pourrions parfaitement l'utiliser comme une étape de prétraitement pour améliorer les performances de l'approche que nous présentons à la section suivante, et qui porte sur la résolution pratique du problème de la satisfaisabilité dans les logiques modales K, KT et S4.

15. Ces travaux sont issus du CRIL, c'est pour cette raison qu'ils ont été portés à ma connaissances, mais je suis persuadé qu'il doit en exister d'autres.

6.3 RECAR : *Recursive Explore and Check Abstraction Refinement*

Comme nous avons pu le voir tout au long de ce manuscrit, utiliser un solveur SAT pour attaquer en pratique des problèmes NP peut être une solution très efficace (Biere et al. 2009). Un des principaux problèmes est alors de trouver le « bon » encodage pour le problème, c'est-à-dire de trouver la réduction polynomiale du problème original vers une formule de la logique propositionnelle en CNF qui peut être efficacement résolue par un solveur SAT (voir le chapitre – CNF *encoding* – dans (Biere et al. 2009)).

La faible expressivité du langage CNF conduit souvent à des situations où le temps nécessaire pour générer et lire la formule est plus grand que le temps nécessaire pour la résoudre. Ceci est typiquement l'un des freins les plus importants à une utilisation massive des solveurs SAT. Cependant, pour les situations où la représentation en CNF est d'une taille trop importante, des approches spécifiques ont été proposées dans la littérature. Dans ce type d'approches, le solveur SAT est utilisé tel un oracle dans une procédure plus complexe. Une de ces procédures est appelée *Counter-Example-Guided Abstraction Refinement* (CEGAR) (Clarke et al. 2003). L'idée est d'abstraire le problème de manière à considérer une version traitable par un solveur SAT.

Il y a deux manières d'abstraire un problème, soit en considérant une sur-abstraction de celui-ci, soit en considérant une sous-abstraction. Lorsqu'une sous-abstraction est utilisée, elle a plus de modèles que le problème original. Par conséquent, si cette sous-abstraction est incohérente, alors le problème original l'est aussi (court-circuit UNSAT). Sinon la procédure est capable de vérifier si le modèle trouvé pour la sous-abstraction est une solution pour le problème original. Dans ce cas, nous avons un court-circuit SAT additionnel capable de décider la cohérence de la formule. Si ce n'est pas le cas, de nouvelles contraintes sont ajoutées pour éviter au solveur de trouver un tel exemple fallacieux (étape de raffinement) et la procédure se répète. Éventuellement une formule équisatisfaisable est détectée, et le solveur SAT peut décider ainsi le problème. Le cas de l'utilisation d'une approche CEGAR avec une sur-abstraction est en miroir, pour plus d'informations voir la sous-section suivante.

L'utilisation pratique d'une approche CEGAR a du sens lorsqu'il est difficile de considérer la formule complète. Ainsi, afin d'éviter cela, il faut soit « avoir de la chance » (court-circuit SAT), soit être en mesure de tirer avantage de la structure du problème afin d'identifier une sous-partie qui est insatisfaisable (court-circuit UNSAT). Cette approche est élégante et a été appliquée à de nombreux problèmes : la cohérence modulo théorie (SMT) (Brummayer et Biere 2009), la planification (Seipp et Helmert 2018) ou encore QBF (Janota et al. 2016).

Le but des travaux que nous avons réalisés, en collaboration avec Daniel Le Berre, Tiago de Lima et Valentin Montmirail dans (Lagniez et al. 2017a; 2018c), était d'attaquer la résolution pratique de certains fragments PSPACE de la logique modale en utilisant une approche de type CEGAR. Il y avait déjà des travaux dans ce sens (Giunchiglia et al. 2002b), cependant nous avons proposé un nouveau schéma algorithmique qui s'adapte bien au problème de résolution de formules de la logique modale. Plus précisément, nous avons introduit une extension à l'approche CEGAR qui propose une étape récursive pour introduire un nouveau court-circuit dans la procédure CEGAR originale. Dans notre contexte, la boucle CEGAR principale contient un court-circuit SAT (↯SAT), pendant que l'étape récursive permet à la procédure de fournir un court-circuit UNSAT (↯UNSAT). Nous appelons cette extension « *Recursive Explore and Check Abstraction Refinement* » (RECAR [16]).

16. Il faut bien entendu voir dans ce nom une référence à une marque d'apéritif très célèbre sur la Canebière.

Puisqu'elle n'est pas spécifique à un domaine, dans la suite, nous commençons par présenter le schéma `RECAR` de manière générique. Nous montrons que pour qu'il soit applicable, il est uniquement nécessaire que certaines conditions soient respectées quant aux fonctions d'abstraction et de raffinement utilisées. Nous avons prouvé que si ces conditions sont respectées, alors l'approche est correcte et complète. Une fois ce `RECAR` servi, nous montrons comment il est possible de l'instancier pour déterminer la cohérence d'une formule en logiques modales `K`, `KT` et `S4`, en fournissant des fonctions d'abstractions pour ces problèmes.

6.3.1 Le schéma `RECAR`

Avant de présenter le schéma de résolution `RECAR`, attardons-nous un instant sur le schéma de résolution `CEGAR` (*Counter-Example Guided Abstraction Refinement*). Ce schéma a été conçu à l'origine pour la question de vérification de modèle (Clarke et al. 2003), c'est-à-dire pour répondre aux questions du type "Est-ce que $S \models P$?" ou de manière équivalente, "Est-ce que $(S \wedge \neg P)$ est incohérent ?", où S décrit un système et P une propriété. Dans les problèmes très structurés, il est souvent le cas que seule une petite partie de la formule soit nécessaire pour répondre à la question. L'idée de la procédure `CEGAR` est alors de remplacer $\phi = (S \wedge \neg P)$ par une abstraction ϕ' de ϕ, où ϕ' est plus simple à résoudre en pratique que ϕ. Deux types d'abstractions sont considérés :

- **sur-abstraction** : $\hat{\phi}$ est une sur-abstraction de ϕ si $\hat{\phi} \models \phi$, c'est-à-dire que $\hat{\phi}$ a au plus autant de modèles que ϕ ;
- **sous-abstraction** : $\check{\phi}$ est une sous-abstraction de ϕ si $\phi \models \check{\phi}$, c'est-à-dire que $\check{\phi}$ a au moins autant de modèles que ϕ.

Les termes sous et sur peuvent d'une certaine manière être interprétés dans le sens « sous-contraint » et « sur-contraint ». Un exemple de problème où la technique `CEGAR` est largement utilisée est la planification `STRIPS` dans le cas propositionnel (Seipp et Helmert 2018). L'idée est en fait de modéliser en `CNF` une version du problème qui consiste à rechercher un plan dans l'espace des plans de longueur k, avec $k \leq n$ et où n est la longueur du plan à partir duquel nous savons que s'il n'existe pas de plans de cette longueur alors il n'existe pas de plans avec une longueur supérieure (c'est une borne sur la longueur des plans). Ainsi, un modèle de cette sur-abstraction $\hat{\phi}$ peut être étendu pour devenir un modèle de ϕ, mais l'incohérence de $\hat{\phi}$ signifie que la borne k a besoin d'être augmentée et la procédure doit être répétée.

La figure 6.3 illustre le déroulement d'une approche `CEGAR` fonctionnant avec des sur-abstractions. L'algorithme décrit par l'organigramme présenté dans la figure reçoit en entrée une formule ϕ et calcule une sur-abstraction ψ de ϕ. Ensuite, il utilise un oracle pour vérifier si ψ est cohérent, si c'est le cas, il conclut que ϕ est cohérent. Sinon ψ est raffinée, c'est-à-dire, qu'elle se rapproche de ϕ, jusqu'à ce qu'elle soit satisfaisable ou jusqu'à ce que le raffinement de la sur-abstraction soit détecté comme étant équisatisfaisable à ϕ, noté $\psi \equiv^{?}_{\text{sat}} \phi$ (c'est-à-dire $\exists M, M \models_1 \psi$ si et seulement si $\exists M', M' \models_2 \phi$) [17] où il conclut que ϕ est incohérent. Dans la suite, $\phi \equiv^{?}_{\text{sat}} \psi$ représente un test incomplet mais efficace pour l'équisatisfaisabilité, test qui répond **oui** ou **inconnu**.

Reprenons notre exemple au sujet de la résolution de problèmes de planification à l'aide d'une approche `CEGAR`. Essayons de comprendre comment une telle approche fonctionne en pratique. Le problème de planification `STRIPS` dans le cas propositionnel est connu pour être `PSPACE`-complet (Bylander 1994), cela implique que généralement il est impossible de transformer un

17. $\models_1$ et $\models_2$ désignent éventuellement différentes relations de conséquence (en logique propositionnelle et en logique modale par exemple).

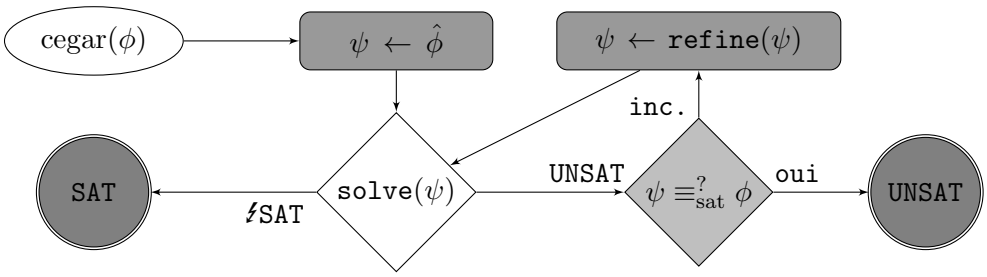


FIGURE 6.3 – Le *framework* CEGAR avec des sur-abstractions.

problème de planification en une formule CNF de taille polynomiale. Le fait de fixer la longueur du plan va en fait permettre de faire descendre la complexité de PSPACE à NP, et donc de pouvoir utiliser la technologie SAT. Cependant, une approche CEGAR de la sorte ne fonctionne que lorsque le problème de planification en entrée possède un plan. En effet, si tel n'est pas le cas, il est nécessaire d'encoder entièrement le problème de planification en CNF, ce qui de manière générale n'est pas envisageable en pratique puisque cette approche conduit à une explosion exponentielle de la taille de la formule.

Cette observation est générale et peut tout aussi bien être appliquée dans le cas d'une approche CEGAR considérant des sous-abstractions. En fait, une approche CEGAR classique avec une sur-abstraction (resp. sous-abstraction) fonctionne bien quand l'entrée est cohérente (resp. incohérente). La raison à cela est qu'il est nécessaire de continuer à raffiner jusqu'à obtenir l'équisatisfaisabilité avec le problème original, ce qui est généralement à éviter si l'oracle utilisé est moins puissant (au sens de la théorie de la complexité) que l'oracle nécessaire pour résoudre le problème d'entrée.

Une manière d'éviter ce problème est de mixer les court-circuits SAT et UNSAT, comme dans (Brummayer et Biere 2009) et (Wang et al. 2007). Dans ces approches, les méthodes alternent les sur et sous abstractions. Nous souhaitons aller un peu plus en loin en permettant l'utilisation d'une approche CEGAR à l'intérieur d'une approche CEGAR. C'est cette procédure que nous avons nommée RECAR pour *Recursive Explore and Check Abstraction Refinement*, et qui est illustrée dans l'organigramme de la figure 6.4. Elle intercale les deux types d'abstraction : chaque abstraction est réalisée avec les informations retournées par le solveur sur le résultat de l'abstraction précédente. Le court-circuit UNSAT est réalisé en utilisant un appel récursif à la procédure principale quand une sous-abstraction stricte $\check{\phi}$ peut être construite. Il convient également de noter que l'approche proposée permet des abstractions sur deux niveaux : une utilisée pour simplifier le problème au niveau du domaine (l'appel récursif), tandis que l'autre permet de simplifier le problème au niveau de l'oracle.

Dans (Lagniez et al. 2017a), nous avons montré que si la sous-abstraction $\check{\phi}$ et la sur-abstraction $\hat{\phi}$ satisfont certaines conditions, alors la procédure RECAR est juste, complète et termine. Dans ce qui suit estSAT(ϕ) signifie que ϕ est cohérente ($\not\models_1 \neg\phi$) et estUNSAT(ϕ) signifie ($\models_2 \neg\phi$), possiblement avec différentes relations de conséquence $\models_1$ et $\models_2$. $RC(\phi, \check{\phi})$ désigne une fonction booléenne décidant si un appel récursif (*Recursive Call*) doit être effectué. Les conditions sont alors les suivantes :

1. La fonction '`solve`' est une implémentation de '`estSAT`' qui est correcte, complète et qui se termine.
2. estSAT($\hat{\phi}$) implique estSAT(refine($\hat{\phi}$)).

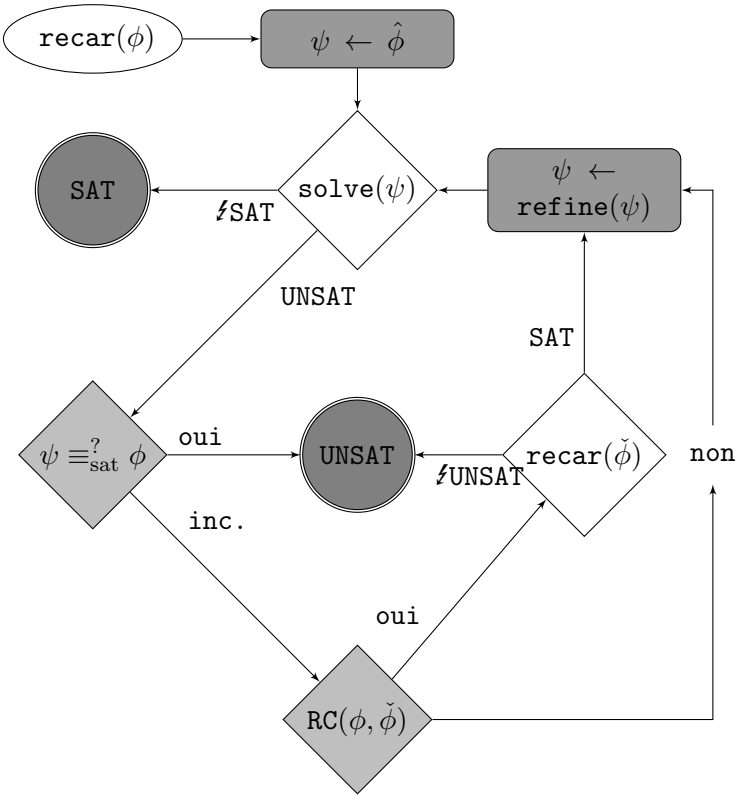


FIGURE 6.4 – L'approche RECAR

3. Il existe $n \in \mathbb{N}$ tel que $\texttt{refine}^n(\hat{\phi}) \equiv^{?}_{\text{sat}} \phi$.
4. $\texttt{estUNSAT}(\check{\phi})$ implique $\texttt{estUNSAT}(\phi)$.
5. Soit $\text{under}(\phi) = \check{\phi}$. Il existe $n \in \mathbb{N}$ tel que $RC(\text{under}^n(\phi), \text{under}^{n+1}(\phi))$ s'évalue à faux.

Notons que nous avons $\texttt{estSAT}(\hat{\phi})$ implique ϕ est cohérent sous la conjonction des Conditions 2 et 3.

Dans la suite, nous implémentons le schéma de résolution RECAR pour résoudre des instances de la logique propositionnelle modale PSPACE.

6.3.2 Le solveur de logique modale MoSaiC

Comme pour la logique modale S5, la fonction de traduction en logique propositionnelle s'appuie sur la traduction standard d'une formule modale vers la logique du premier ordre (Blackburn et al. 2006). Cependant, contrairement à ce que nous avons pu réaliser pour la traduction de S5 en CNF, nous avons besoin ici de représenter les relations entre les mondes. Ainsi, la traduction ajoute de nouvelles variables p_i et $r^a_{i,j}$ à la formule. p_i signifie que la variable p est vraie dans le monde w_i alors que $r^a_{i,j}$ signifie que w_j est accessible depuis w_i par la relation a. La fonction de traduction proposée est la suivante :

Définition 81 (Traduction d'une formule de K)

$$
\begin{aligned}
\mathrm{tr}(\phi, n) &= \mathrm{tr}'(\mathtt{NNF}(\phi), 0, n) \\
\mathrm{tr}'(p, i, n) &= \mathsf{p}_i \\
\mathrm{tr}'(\neg p, i, n) &= \neg p_i \\
\mathrm{tr}'(\phi \wedge \psi, i, n) &= \mathrm{tr}'(\phi, i, n) \wedge \mathrm{tr}'(\psi, i, n) \\
\mathrm{tr}'(\phi \vee \psi), i, n) &= \mathrm{tr}'(\phi, i, n) \vee \mathrm{tr}'(\psi, i, n) \\
\mathrm{tr}'(\Box_a \phi, i, n) &= \bigwedge_{j=0}^{n} (r_{i,j}^a \rightarrow \mathrm{tr}'(\phi, j, n)) \\
\mathrm{tr}'(\Diamond_a \phi, i, n) &= \bigvee_{j=0}^{n} (r_{i,j}^a \wedge \mathrm{tr}'(\phi, j, n))
\end{aligned}
$$

Puisqu'il n'y a pas de contraintes sur le cadre de la structure de Kripke, cette fonction de traduction n'est utilisable que pour traiter la logique modale K. Dans (Lagniez et al. 2018b), toujours en collaboration avec Daniel Le Berre, Tiago de Lima et Valentin Montmirail, nous avons étendu cette traduction aux autres logiques modales PSPACE. Pour cela, nous ajoutons pour chaque axiome requis un ensemble de contraintes qui limite l'espace des structures de Kripke possibles à celles respectant les axiomes de cadre. Nous avons donc, pour chaque axiome, les règles additionnelles suivantes :

$$
\begin{aligned}
\mathrm{tr}((T), n) &= \bigwedge\nolimits_{a=0}^{m} \bigwedge\nolimits_{i=0}^{n} (r_{i,i}^a) \\
\mathrm{tr}((D), n) &= \bigwedge\nolimits_{a=0}^{m} \bigwedge\nolimits_{i=0}^{n} \bigvee\nolimits_{j=0}^{n} (r_{i,j}^a) \\
\mathrm{tr}((B), n) &= \bigwedge\nolimits_{a=0}^{m} \bigwedge\nolimits_{i=0}^{n} \bigwedge\nolimits_{j=0}^{n} (r_{i,j}^a \Rightarrow r_{j,i}^a) \\
\mathrm{tr}((4), n) &= \bigwedge\nolimits_{a=0}^{m} \bigwedge\nolimits_{i=0}^{n} \bigwedge\nolimits_{j=0}^{n} \bigwedge\nolimits_{k=0}^{n} ((r_{i,j}^a \wedge r_{j,k}^a) \Rightarrow r_{i,k}^a) \\
\mathrm{tr}((5), n) &= \bigwedge\nolimits_{a=0}^{m} \bigwedge\nolimits_{i=0}^{n} \bigwedge\nolimits_{j=0}^{n} \bigwedge\nolimits_{k=0}^{n} ((r_{i,j}^a \wedge r_{i,k}^a) \Rightarrow r_{j,k}^a)
\end{aligned}
$$

À partir de cette fonction de traduction, il est alors possible de décider la cohérence d'une formule modale ϕ dans K en appelant un solveur SAT sur la formule $\mathrm{tr}(\phi, b(\phi))$, où $nm(\phi) + 1$ donne le nombre de mondes nécessaires pour satisfaire ϕ (voir la section 2.3 pour plus de détails). Le principal problème avec cela est que la traduction peut générer une formule exponentiellement plus grande, ce que nous voulons éviter si possible en utilisant le *framework* RECAR.

Comme nous l'avons mentionné précédemment, afin d'appliquer le *framework* RECAR, nous avons besoin de garantir que les fonctions de sur-abstraction, de sous-abstraction et de raffinement respectent certaines conditions. Commençons par définir formellement les fonctions de sur-abstraction et de raffinement et regardons si elles vérifient bien les conditions attendues.

Définition 82 (Sur-abstraction)

Soit $\phi \in \mathcal{L}$. La sur-abstraction de ϕ, noté $\hat{\phi}$, est la formule $\mathrm{tr}(\phi, 1)$.

Définition 83 (Raffinement)

Soit $1 \leq n \leq nm(\phi) + 1$. Le raffinement de $\mathrm{tr}(\phi, n)$, noté $\texttt{refine}(\mathrm{tr}(\phi, n))$ est la formule $\mathrm{tr}(\phi, n+1)$.

Il est facile de montrer que la condition 2 (si $\texttt{estSAT}(\mathrm{tr}(\phi, n))$ alors $\texttt{estSAT}(\mathrm{tr}(\phi, n+1))$ pour tous $1 \leq n < nm(\phi)+1$) est satisfaite. Si ϕ est satisfaite par un modèle M avec n mondes, alors nous pouvons trouver un modèle M' avec $n+1$ mondes satisfaisant ϕ, en ajoutant à M un monde additionnel qui n'est juste pas accessible depuis les mondes déjà dans M. Il est aussi facile de montrer que la condition 3 est aussi satisfaite, puisqu'il suffit de raffiner $nm(\phi)$ fois pour arriver à l'équisatisfaisabilité.

En ce qui concerne la fonction de sous-abstraction, nous allons travailler sur la structure de la représentation `NNF` de la formule. Considérons un exemple. Soit $\phi = (\Diamond p \wedge \Box\neg p \wedge \chi)$ pour un $\chi \in \mathcal{L}$, et où $b(\chi)$ est grand. ϕ est incohérente car $(\Diamond p \wedge \Box\neg p)$ est incohérente. Cependant sur une telle formule, une approche `CEGAR` utilisant la sur-abstraction et le raffinement définis plus tôt va prendre un temps considérable avant d'être capable de conclure à l'incohérence. La raison étant que chaque raffinement $\mathrm{tr}(\phi, n+1)$ de la formule originale va être montré incohérent et la procédure ne va s'arrêter que lorsque la borne $b(\phi)$ sera atteinte.

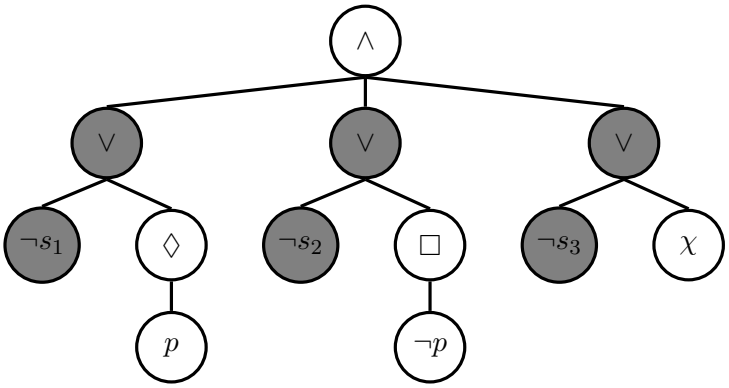


FIGURE 6.5 – Application de sélecteurs pour l'identification d'une sous-formule incohérente.

Aussi, afin d'éviter ces cas pathologiques, notre fonction de sous-abstraction va essayer d'identifier une sous-formule incohérente $\check{\phi}$ de ϕ qui soit « sous-contrainte ». Comme nous l'avons déjà spécifié, la formule ϕ est sous forme `NNF`. Ainsi, lorsque nous remplaçons par $\top$ une branche d'un nœud $\wedge$, nous rendons moins contrainte la formule résultante. La difficulté est alors de bien choisir les sous-formules à occulter. Pour réaliser cela, nous avons proposé une approche qui exploite le résultat du solveur `SAT` lors de l'appel sur la sur-abstraction $\hat{\phi}$. Plus précisément,

nous étiquetons chaque branche des nœuds avec un sélecteur lors de la transformation en CNF de la sur-abstraction, de manière à ce que l'affectation à vrai de tous les sélecteurs conduise à activer chaque branche des nœuds $\wedge$ (et donc à obtenir une formule équivalente à $\hat{\phi}$). Ainsi, si nous appelons un solveur SAT avec comme hypothèses l'ensemble des sélecteurs à vrai, alors une réponse SAT du solveur indique que $\hat{\phi}$ est satisfaisable, ce qui nous permet de conclure. Sinon, dans le cas où le solveur retourne UNSAT, nous pouvons exploiter le noyau retourné par le solveur afin d'identifier la partie de la sur-abstraction qui est incohérente, et donc nous focaliser sur cette dernière.

Afin de mieux cerner la manière dont fonctionne notre procédure, reprenons notre exemple pathologique. Tout d'abord, nous ajoutons à chaque conjonction dans ϕ, une nouvelle variable s_i (un sélecteur) qui va être supposée vraie par le solveur SAT, comme illustré par la figure 6.5. Ensuite, nous construisons notre première sur-abstraction $\text{tr}(\phi, 1)$ et nous alimentons le solveur SAT avec. Le solveur va répondre incohérent en fournissant un cœur incohérent exprimé avec les s_i. De ce cœur incohérent, nous pouvons donc extraire un ensemble de sélecteurs *core*. Supposons que, dans notre exemple, $core = \{s_1, s_2\}$. Cela signifie que la formule $\check{\phi} = (\Diamond p \wedge \Box\neg p)$, étiquetée par les sélecteurs est suffisante pour montrer l'incohérence de ϕ avec seulement 1 monde possible. Prouver l'incohérence de $\check{\phi}$ va impliquer que ϕ est incohérente. Il est à noter que, dans ce cas, $b(\check{\phi})$ est bien plus petit que $b(\phi)$. Ainsi, l'approche CEGAR appliquée à $\check{\phi}$ va répondre bien plus rapidement que si elle avait été appliquée sur ϕ. Formellement, nous avons :

Définition 84 (Sous-Abstraction)

$$
\begin{aligned}
&\text{under}(p, core) = p\\
&\text{under}(\neg p, core) = \neg p\\
&\text{under}(\Box_a\phi, core) = \Box_a(\text{under}(\phi, core))\\
&\text{under}(\Diamond_a\phi, core) = \Diamond_a(\text{under}(\phi, core))\\
&\text{under}((\phi \wedge \psi), core) = \text{under}(\phi, core) \wedge \text{under}(\psi, core)\\
&\text{under}((\psi \vee \chi), core) = \begin{cases} \text{under}(\chi, core) & \text{si } (\psi = \neg s_i, s_i \in core)\\ \top & \text{si } (\psi = \neg s_i, s_i \notin core)\\ (\text{under}(\psi, core) \vee \text{under}(\chi, core)) & \text{sinon} \end{cases}
\end{aligned}
$$

Dans (Lagniez et al. 2017a), nous avons montré que la fonction 'under' satisfait les conditions 4 et 5 de RECAR, et donc en considérant ensemble les fonctions de sur-abstraction, de raffinement et de sous-abstraction nous pouvons construire une approche de résolution pour les logiques modales PSPACE. Le solveur résultant, nommé MoSaiC, est présenté sous forme d'organigramme dans la figure 6.6.

MoSaiC possède aussi différentes fonctionnalités qui figurent dans les solveurs de l'état de l'art. Comme dans Km2SAT (Sebastiani et Vescovi 2009b), il simplifie la formule en entrée en appliquant différentes règles : le *Box Lifting*, le *Flattening* et la règle de *Truth Propagation* sur les opérateurs booléens et modaux (voir (Sebastiani et Vescovi 2009b) pour plus d'informations).

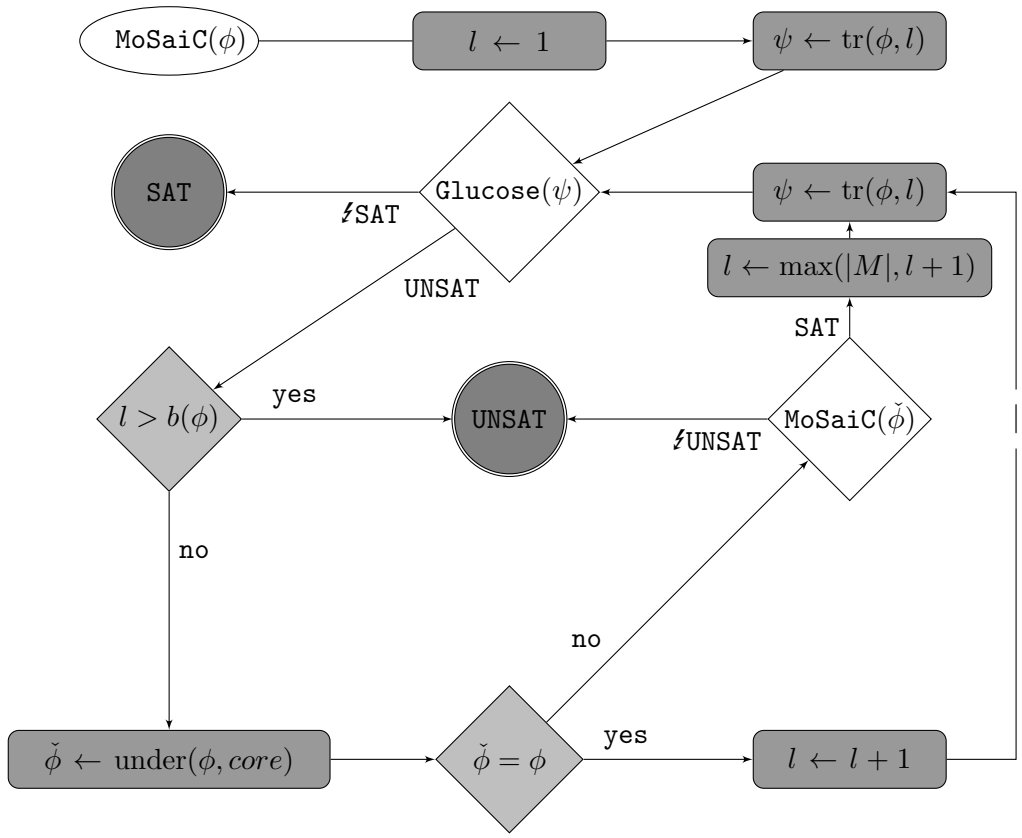


FIGURE 6.6 – `MoSaiC` : `RECAR` pour la logique modale K

`MoSaiC` utilise aussi le solveur `Glucose` en mode incrémental Audemard et al. (2013a) (voir section 4.2 pour plus détails au sujet de cette version de `Glucose`) pour décider la cohérence de chaque sur-abstraction (ψ).

Notons que l'implémentation réelle de `MoSaiC` diffère légèrement de celle présentée à la figure 6.6 (l'implémentation réelle n'est pas décrite en détail dans un souci de lisibilité). Par exemple, nous n'appelons pas `Glucose` sur ψ mais sur ψ sur laquelle nous avons appliqué les sélecteurs ; nous n'avons pas besoin de générer la sous-abstraction $\check{\phi}$ pour tester la condition ($\check{\phi} = \phi$) : il suffit de connaître le nombre de sélecteurs impliqués dans l'incohérence de la formule. Nous retournons aussi un modèle de Kripke dans la procédure principale, pas simplement `SAT`/`UNSAT`. Nous tirons avantage d'une telle information pour fournir une nouvelle borne l.

6.3.3 Discussion

Dans cette section, nous avons proposé une nouvelle approche, nommée `RECAR`, pour résoudre des problèmes de décision en utilisant une approche basée sur du raffinement d'abstractions récursives. Nous avons montré la correction de cette approche, et nous l'avons instanciée pour le problème de la cohérence d'une formule représentée en logique modale. Ce solveur, nommé `MoSaiC`, montre clairement l'intérêt pratique de mixer les court-circuits `SAT` et `UNSAT`. En effet, les expérimentations que nous avons conduites dans (Lagniez et al. 2017a; 2018b), ont montré que `MoSaiC` surpasse les autres solveurs de l'état de l'art sur les *benchmarks* considérés.

Il existe de nombreuses améliorations à apporter au solveur `MoSaiC` afin de le rendre plus

efficace. En effet, la version de MoSaiC présentée dans cette section ne prend pas complètement en compte les informations portées par les axiomes de cadre. Afin de les intégrer, il est possible de généraliser le *framework* RECAR pour utiliser des compositions de fonctions d'abstraction. L'idée est de considérer un ensemble d'abstractions qui, considérées ensemble, permettent de réduire de manière plus agressive la formule originale que nous aurons à encoder, tout en préservant sa satisfaisabilité. De manière générale, nous pouvons toujours couper une branche enracinée dans un nœud $\vee$ lorsque la formule est représentée en NNF pour obtenir une sur-abstraction. Cependant, si nous voulons aller plus loin il faut aussi considérer la sémantique des axiomes. Par exemple, considérons une formule $\phi \in \mathcal{L}$ en NNF dans une logique modale satisfaisant (T). Par l'axiome (T) nous avons $\Box\phi \Rightarrow \phi$, et donc si nous remplaçons une sous-formule $\Box_a\psi$ par ψ, alors la formule résultante ϕ' est une sous-abstraction de ϕ. Nous avons déjà réalisé des expérimentations préliminaires combinant différentes abstractions, montrant qu'il est possible d'obtenir un gain de performances significatif.

Afin de booster les performances de notre approche, nous pourrions aussi travailler directement sur la représentation de la structure de Kripke. En effet, pour le moment, nous parcourons « bêtement » l'ensemble des structures contenant un certain nombre de mondes. Cependant, il serait facile d'ajouter des contraintes permettant de ne pas considérer certaines structures qui ne peuvent pas être solution. Par exemple, si nous savons qu'une formule ϕ n'est pas satisfaisable avec n mondes, alors il est nécessaire que le monde courant (w_0) soit connecté à au moins $n+1$ mondes (ce qui peut s'exprimer par un ensemble de clauses). Une autre manière de réduire l'espace de recherche est de tirer parti du fait que les mondes sont interchangeables afin de casser certaines symétries. Pour cela, nous pourrions simplement ajouter des contraintes spécifiant un ordre d'atteignabilité sur les mondes. Par exemple, il est facile de contraindre le problème de sorte que le monde w_0 ne peut pas être connecté au monde w_{k+1} s'il n'est pas connecté au monde w_k. Une question intéressante consiste à déterminer s'il est possible de casser ce type de symétrie de manière générale, et de déterminer si contraindre l'espace des structures ne rend pas les CNF générées plus difficiles à résoudre en pratique.

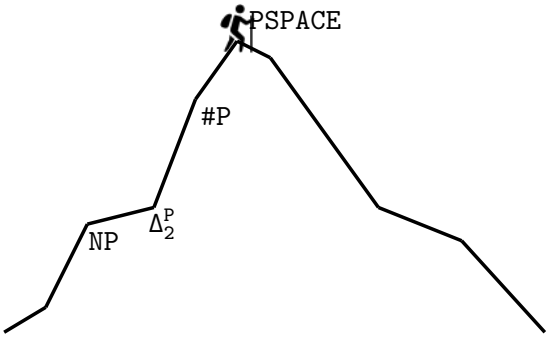
Un point critique concernant l'efficacité de notre approche concerne le nombre de mondes des structures de Kripke que nous devons modéliser en CNF. En effet, contrairement au cas de la transformation de S5 en CNF, cette valeur est exponentielle par rapport à la taille de la formule d'entrée. Une question est : pouvons-nous, comme avec le *diamond degree*, prendre en compte la structure de la formule afin de réduire cette valeur ? C'est une question très intéressante d'un point de vue théorique dont la réponse aurait des répercussions pratiques immédiates. Mon intuition est qu'il est possible de considérer le nombre de mondes maximal par rapport à une sous-formule ϕ' d'une formule modale ϕ exprimée en NNF, où nous avons remplacé certaines sous-formules enracinées en des nœuds $\vee$ par $\bot$. L'idée est que si la formule ϕ' est satisfaisable, alors ce modèle doit aussi satisfaire ϕ. Par conséquent, dans le pire des cas, nous devons considérer chaque sous-formule de ϕ où les nœuds $\vee$ ne possèdent qu'un fils. Cela est juste une intuition, et il reste encore pas mal de travail pour décider si elle est correcte. Avant de faire cela, nous pourrions commencer par déterminer si en pratique le nombre de mondes nécessaires pour satisfaire une formule modale cohérente est largement inférieur à la borne théorique. Nous pourrions à cet effet étendre la notion de modèle minimal définie dans le cadre S5 (voir section précédente 6.2.2) aux logiques modales PSPACE.

Ce que nous souhaitons éviter à tout prix avec notre approche, c'est de considérer trop de mondes dans les structures de Kripke que nous encodons en CNF. Il est clair que dans la situation actuelle, lorsque nous augmentons le nombre de mondes, nous ne pouvons jamais le diminuer ensuite. L'augmentation réalisée peut être très importante au retour de l'appel récursif, puisque

nous considérons le maximum entre la taille du modèle $|M|$ calculé par RECAR et $l+1$. Cependant, il peut exister une petite sous-formule qui est incohérente que le cœur retourné par le solveur SAT ne permet pas de détecter (à cause de l'heuristique utilisée par exemple).

Afin d'éviter cette situation, nous souhaitons mieux exploiter le retour du solveur de manière à réaliser d'autres appels récursifs avant d'augmenter la borne. En effet, pour le moment, nous exploitons principalement le cœur retourné par le solveur SAT afin de sélectionner une sous-formule que nous allons considérer lors de l'appel récursif à RECAR. Cependant, nous n'exploitons pas le retour du solveur RECAR lorsque la sous-abstraction est satisfaisable. Il pourrait être intéressant, avant d'augmenter le nombre de mondes, de vérifier si le modèle calculé par l'appel récursif ne pourrait pas être utilisé pour satisfaire la formule dans son ensemble. Cela pourrait par exemple être réalisé en utilisant le vérificateur de modèle que nous avons proposé dans (Lagniez et al. 2016a). Si le modèle calculé est bien un modèle de la formule, alors nous avons résolu le problème dans son ensemble (nous pourrions mettre des guillemets ici car nous ne vérifions que la formule passée lors de l'appel récursif, qui peut être différente de la formule que nous souhaitons résoudre en général). L'avantage est que ce modèle peut remonter en cascade les appels récursifs et donc permettre de réduire sensiblement le nombre d'appels à un oracle SAT. Si le modèle calculé n'est pas un modèle de la formule, alors nous pouvons extraire la sous-formule qui conduit à cette situation et réaliser un autre appel récursif sur cette dernière. Dans ce cas, soit nous arrivons à trouver une sous-abstraction incohérente, soit nous n'arrivons pas à extraire une sous-formule « stricte » et donc nous augmentons la borne afin de considérer un nouvelle sur-abstraction (ce qui correspond à ce qui a été implémenté).

6.4 Conclusion



Nous avons atteint notre objectif qui était la résolution pratique de problèmes PSPACE-complets. Plus précisément, dans ce chapitre, nous nous sommes principalement concentrés sur la résolution pratique de problèmes modélisés avec le formalisme de la logique propositionnelle modale.

Dans ce contexte, nous avons dans un premier temps proposé une approche pour la résolution d'instances modélisées avec un fragment de la logique modale qui est NP-complet : S5. Pour réaliser cela, nous avons proposé une nouvelle traduction en logique propositionnelle permettant de tirer parti de l'efficacité des solveurs SAT. Nous avons montré dans (Caridroit et al. 2017) qu'il est possible de réduire sensiblement la taille de la traduction en considérant des informations sur la structure de la formule modale que nous souhaitons traiter (voir section 6.2.1). Nous sommes même allés un peu plus loin en considérant le problème d'optimisation qui consiste à

rechercher le modèle de Kripke qui satisfait une formule S5 avec le moins de mondes possible (voir section 6.2.2). Nous avons pu constater dans (Lagniez et al. 2018b) que ce nombre pouvait être significativement plus petit que celui que nous étions en mesure de calculer en prenant en compte uniquement la structure de la formule. Ce résultat laisse donc envisager de futures améliorations quant à la performance de notre solveur.

Les travaux que nous avons réalisés dans le cadre de la logique modale S5 nous ont confortés dans l'idée qu'il est envisageable de rechercher une solution dans l'ensemble des structures de Kripke d'une certaine taille. De cette observation, nous nous sommes intéressés à la résolution pratique de fragments PSPACE de la logique propositionnelle modale. Plus précisément, nous avons proposé une nouvelle procédure de raffinement, que nous avons nommée RECAR. Contrairement aux approches CEGAR, cette nouvelle approche permet de faire « communiquer » plusieurs types d'approximation. Nous avons montré expérimentalement dans (Lagniez et al. 2017a; 2018c) que le schéma de résolution RECAR pouvait être utilisé efficacement pour traiter en pratique des problèmes modélisés en logique modale. Ce qui est remarquable est que MoSaiC, le solveur qui implémente le schéma RECAR, résout significativement plus d'instances que les approches de l'état de l'art sur les instances que nous avons considérées.

Ces travaux, autour de la résolution pratique de problèmes modélisés en logique propositionnelle modale, ouvrent de nombreuses perspectives de recherche. L'une d'entre elles consiste à prendre en compte des applications réelles. Les problèmes actuellement considérés dans nos expérimentations ne sont pas satisfaisants, dans le sens où ils sont généralement générés de manière aléatoire et ne correspondent donc pas à des cas réels. Il est connu, dans le contexte de la résolution pratique de SAT, que la structure du problème considéré a un impact important sur les performances du solveur (par exemple, dans le cadre de SAT, les solveurs de type CDCL ne sont efficaces que sur les instances structurées). Ainsi, nous avons tout intérêt, pour promouvoir nos solveurs et aussi afin de valider leur efficacité pratique, de chercher à modéliser de nouveaux problèmes qui ont « du sens ». Pour cela, nous pouvons par exemple nous inspirer des travaux présentés dans (der Hoek et Pauly 2007) afin de modéliser des problèmes issus de la théorie des jeux, ou encore de nous inspirer des travaux de Bolander et Andersen (2011) afin de modéliser des problèmes de planification.

Une autre piste de recherche que nous souhaitons explorer consiste à tirer parti du paradigme de résolution RECAR pour attaquer en pratique d'autres types de problèmes. Dans ce contexte, et en collaboration avec Philippe Dague et Lina Ye, nous envisageons d'étendre le schéma RECAR pour attaquer en pratique des modèles à événements discrets (automates et automates temporisés) ou hybrides (automates hybrides) abstraits par les précédents. Plus précisément, nous souhaitons proposer une solution pour l'analyse de la diagnosticabilité (et d'autres propriétés formelles voisines, plus faibles comme la manifestabilité, plus fortes comme la prédictabilité, faisant intervenir des actions sur le système comme la contrôlabilité). L'étude de la diagnosticabilité d'un système consiste à concevoir et implémenter des algorithmes de vérification de propriétés formelles du système garantissant qu'un modèle, dont on connaît à l'avance les événements ou variables observables, permet la détection et la discrimination d'un ensemble de défaillances possibles connues a priori et incorporées au modèle du système. Cette propriété est de première importance et doit être vérifiée formellement en amont dès la phase de conception ou dans les premières phases de réalisation. La diagnosticabilité, une fois vérifiée, permet d'engendrer automatiquement un diagnostiqueur qui, sur la foi des uniques observations disponibles, sera capable d'identifier en ligne la survenue de fautes dans le système en fonctionnement. Les travaux sur la diagnosticabilité ont essentiellement porté sur des modèles de systèmes à événements discrets. Il a été montré que toute paire de trajectoires suffisamment longues partageant la même

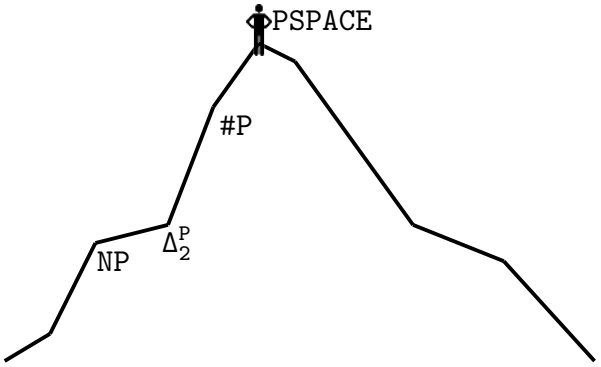
observation, l'une fautive et l'autre normale (appelée paire critique), constitue un témoin de non-diagnosticabilité. De là, il a été prouvé que le problème de vérification de la diagnosticabilité était dans la classe `P` pour les automates et devenait `PSPACE`-complet pour les automates temporisés. Le défi est d'étendre les méthodes précédentes des systèmes à événements discrets aux systèmes hybrides. Mais, en raison de la présence d'espaces d'état continus en chaque mode, la quasi-totalité des propriétés (comme la propriété de base d'atteignabilité d'états et *a fortiori* la diagnosticabilité) est non décidable pour les automates hybrides.

C'est pourquoi en général des abstractions qualitatives discrètes de ces automates hybrides sont utilisées (en partitionnant l'espace continu en chaque mode en un nombre fini de régions géométriques), préservant la propriété à prouver (ou sa négation selon la propriété) de façon à ce qu'elle soit décidable au niveau d'une telle abstraction (Zaatiti et al. 2018, Zaatiti 2018). Ce raffinement est donc classiquement guidé par le contre-exemple à la propriété trouvé au niveau abstrait (et jugé fallacieux au niveau concret), de façon à l'éviter ainsi que ceux qui lui ressemblent au niveau plus fin (Zaatiti et al. 2017) : c'est la technique `CEGAR`. Il s'agira donc d'étudier l'application du schéma `RECAR` au problème de diagnosticabilité, ce qui comprendra à la fois l'extension de ce schéma et son instanciation à ce problème. Le verrou principal va être de caractériser les sur- et sous-approximations pertinentes pour l'application de `RECAR`.

Finalement, nous aimerions aussi paralléliser le schéma `RECAR`. Il existe deux manières de réaliser cela. Soit nous parallélisons l'oracle qui est utilisé pour vérifier la cohérence des approximations. Soit nous parallélisons la procédure `RECAR`. La première forme de parallélisme est clairement facilement réalisable, et pour cela, elle n'a pas beaucoup d'intérêt. La seconde forme de parallélisation est plus intéressante, dans le sens où elle intervient au niveau de l'algorithme `RECAR` et offre donc plus de possibilités. Une manière simple de réaliser un tel parallélisme pourrait, par exemple, consister à considérer plusieurs sous-approximations qui seraient réalisées en parallèle, ou encore à tester différentes sur-approximations avec différentes tailles pour la structure de Kripke recherchée.

Chapitre 7

Conclusion



Ce manuscrit raconte le voyage d'un ch'ti chercheur en informatique qui s'efforce de résoudre en pratique des problèmes au-delà de NP. Étant donné la difficulté de la tâche, il est clair que cette ascension n'a été possible qu'avec le concours d'un ensemble d'amis et collègues (le nous dans le manuscrit), qui sont : Gilles Audemard, Anicet Bart, Daniel Le Berre, Philippe Besnard, Armin Biere, Frédéric Boussemart, Thomas Caridroit, Gaël Glorian, Éric Grégoire, Long Guo, Katsutoshi Hirayama, Benoît Hoessen, Katsumi Inoue, Yacine Izza, Saïd Jabbour, Sébastien Konieczny, Frédéric Koriche, Christophe Lecoutre, Tiago de Lima, Emmanuel Lonca, Jean-Guy Mailly, Pierre Marquis, Bertrand Mazure, Valentin Montmirail, Tenda Okimoto, Anastasia Paparrizou, Cédric Piette, Lakhdar Saïs, Nicolas Schwind, Laurent Simon, Michael Sioutis, Nicolas Szczepanski, Sébastien Tabary, Samuel Thomas et Du Zhang. Parmi ces compagnons de route, il y en a deux particulièrement qui, grâce à leurs conseils et encouragements, ont rendu ce voyage possible et intéressant : Éric Grégoire et Pierre Marquis.

Pour atteindre le sommet PSPACE, nous avons choisi de suivre le versant de la logique propositionnelle. Plus précisément, nous avons choisi tout au long de ce manuscrit de nous appuyer sur une utilisation fine des solveurs SAT. C'est donc tout naturellement que notre première étape a été dirigée vers la résolution pratique de SAT. Ainsi, au chapitre 3, nous avons présenté deux contributions autour de l'exploitation des architectures à mémoire distribuée pour décider la satisfaisabilité d'une formule de la logique propositionnelle exprimée en CNF. Dans ce contexte, nous avons présenté le solveur AmPharoS, qui est basé sur le paradigme « diviser pour mieux régner », et qui s'appuie sur une construction dynamique et collaborative d'un arbre de guidage. Nous avons montré qu'en monitorant l'échange de clauses *via* l'utilisation d'un processus maître, il est possible de mesurer la quantité de travail redondant réalisée par les processus solveurs et, en fonction de cette information, d'ajuster dynamiquement la recherche afin de l'intensifier ou de la diversifier.

Bien que les résultats obtenus par AmPharoS étaient plutôt bons, nous nous sommes aperçus en analysant le comportement du solveur que l'échange de clauses entre les unités de calcul avait

tendance à congestionner le réseau et donc à réduire les performances de l'approche. Nous avons alors proposé notre deuxième contribution à la résolution parallèle de SAT, qui traite de l'échange de clauses dans un environnement à mémoire distribuée. Afin de réaliser cela, nous avons dans un premier temps décidé d'étudier différents modèles de programmation parallèle pour choisir celui qui pourrait être le plus adapté pour échanger un nombre important de clauses. Ces travaux ont donné lieu à une implémentation distribuée du solveur parallèle multi-cœurs Syrup, nommée d-Syrup. Cette étude nous a conduits à un schéma de partage d'informations qui utilise à la fois le passage de messages et la mémoire partagée.

Nous avons poursuivi notre route au chapitre 4, vers la résolution du problème $\Delta_2^{\mathtt{P}}$ d'extraction d'un MCS d'une formule donnée. Ce problème pouvant être résolu par une procédure consistant à appeler un nombre polynomial de fois un solveur SAT, nous avons d'abord décidé d'étudier le comportement des solveurs SAT dans ce contexte incrémental. Nous avons observé qu'ajouter des sélecteurs, afin de permettre une réutilisation des clauses apprises entre les différents appels au solveur, pouvait perturber son fonctionnement et ainsi réduire ses performances. Ceci nous a conduit à proposer et à expérimenter différentes améliorations pouvant être greffées aux solveurs SAT.

Une fois cette étape préliminaire achevée, nous avons présenté CMP un nouveau schéma algorithmique pour l'extraction d'un MCS. Nous avons montré expérimentalement que, bien qu'ayant une complexité en temps quadratique (contrairement aux approches de l'état de l'art qui ont une complexité généralement linéaire) en fonction du nombre de clauses, CMP est très compétitif sur un large ensemble de problèmes. Ce chapitre autour de la résolution de problèmes $\Delta_2^{\mathtt{P}}$ a aussi été l'occasion d'aller un peu plus loin en considérant le problème d'énumération de MCSes. L'algorithme que nous avons proposé à cet effet s'appuie sur deux propriétés faisant intervenir la notion de clause de transition, et qui permettent de créer des familles de MCSes. En s'appuyant sur ces propriétés, nous avons pu élaborer une approche qui, contrairement aux méthodes de l'état de l'art qui calculent indépendamment chaque MCS, tente d'accélérer l'énumération en regroupant certains MCSes. Nos expérimentations ont permis de montrer que notre approche pouvait être combinée efficacement avec la technique de *caching* proposée par Previti et al. (2017).

Ce petit écart autour de l'énumération de MCSes nous a préparé à la suite de notre ascension qui nous a orienté au chapitre 5 vers le problème #P, et plus précisément vers la résolution du problème #SAT. Pour résoudre ce problème, il existe deux approches principales : l'approche directe et l'approche par compilation. Néanmoins, quelle que soit l'approche utilisée, le théorème de Toda (1989) ($\mathtt{PH} \subseteq \mathtt{P}^{\#\mathtt{P}}$), et montre que cette tâche est calculatoirement difficile. Partant de ce constat, nous nous sommes un temps aventurés sur un chemin nous menant aux techniques de pré-traitement de formules CNF. En effet, puisque #SAT est #P, nous pouvons largement utiliser des procédures de pré-traitement qui s'appuient sur l'utilisation d'un solveur SAT. Ainsi, nous avons vu qu'il est possible de calculer le *backbone* de la formule ou encore de rechercher des définitions dans cette dernière en pré-traitement. L'utilisation de ces techniques de réécriture permettent souvent une réduction du temps CPU nécessaire au calcul du nombre de modèles de l'instance ou du temps de compilation de celle-ci et de la taille de sa forme compilée.

Nous avons ensuite continué notre ascension vers #SAT en travaillant directement à l'élaboration de compteurs de modèles et de compilateurs. Ces travaux ont donné naissance à l'outil d4, qui tente d'exploiter au maximum la structure de l'instance considérée en cherchant le plus rapidement à générer des nœuds $\wedge$ décomposables et en améliorant les schémas de *caching* utilisés pour le partage de structure. Plus précisément, nous avons proposé une nouvelle heuristique qui s'appuie sur l'utilisation parcimonieuse d'un algorithme de partitionnement d'hypergraphes et un schéma de *caching* qui permet d'approcher correctement la relation d'équivalence entre formules

et qui a aussi l'avantage d'être plus compact que les schémas ayant un pouvoir d'approximation similaire.

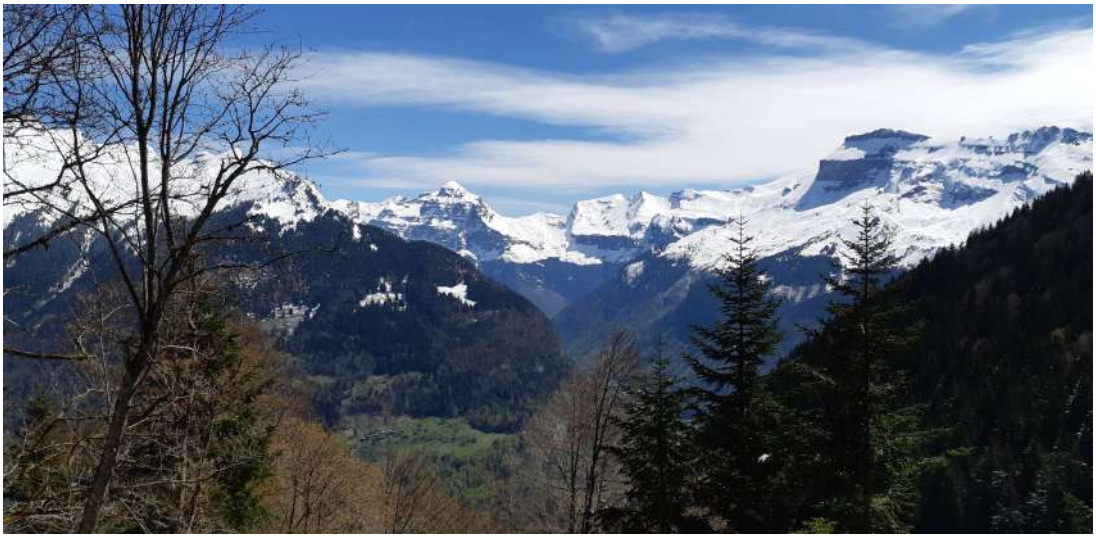
Avant de nous élever vers PSPACE, nous avons exploré la problématique du comptage de modèles d'une formule CNF où certaines variables doivent être oubliées. Pour réaliser cela, nous avons proposé une nouvelle approche qui va découper récursivement, sous la forme d'une disjonction déterministe, la formule d'entrée Σ de manière à ne considérer que des formules sans variables à oublier, c'est-à-dire des formules construites sur les variables de $Var(\Sigma) \setminus X$ pour lesquelles nous sommes capables d'utiliser toute la puissance des compteurs de modèles de l'état de l'art.

Nous avons terminé ce chapitre par la présentation d'un nouveau langage de compilation de connaissances permettant le comptage de modèles, qui s'appuie sur le langage AFF des formules affines. Puisque ce langage n'est pas complet pour la logique propositionnelle, nous avons proposé de nouveaux langages cibles pour la compilation de formules propositionnelles s'appuyant sur l'utilisation d'une forme restreinte de la décomposition de Shannon faisant intervenir des clauses affines. Plus précisément, ces travaux ont donné naissance aux langages EDT des arbres de décision étendus, ADT des arbres de décision affines et EADT des arbres de décision affines étendus. L'analyse de ces langages selon les critères mis en œuvre dans la carte de compilation ont permis de déterminer les requêtes et les transformations de la carte de compilation réalisables en temps polynomial à partir de représentations dans ces langages, et montré que certaines ne le sont pas sauf si $\mathsf{P} = \mathsf{NP}$. Nous avons aussi pu comparer l'efficacité spatiale relative de ces langages à celles d'autres langages cibles existants, et nous avons montré que le langage EADT est intéressant lorsque la requête du comptage de modèles est cruciale. Le développement d'un compilateur du langage CNF vers EADT, nous a aussi permis de montrer que ce langage est réellement attractif en pratique.

Pour arriver à PSPACE, nous sommes passés, au chapitre 6, par la résolution pratique de problèmes modélisés avec le formalisme de la logique propositionnelle modale. Avant d'attaquer la partie difficile, nous avons étudié la résolution d'instances modélisées avec un fragment de la logique modale qui est NP-complet : S5. Nous avons proposé une traduction en logique propositionnelle qui s'appuie sur une représentation explicite d'un modèle de Kripke. Afin de réduire la taille de la traduction, nous avons exploité des informations sur la structure de la formule modale nous permettant de réduire le nombre de mondes de la structure de Kripke que nous souhaitons représenter. De manière à estimer combien de mondes il est nécessaire d'utiliser pour satisfaire une formule S5, nous avons aussi étudié le problème d'optimisation qui consiste à rechercher un plus « petit » modèle de Kripke. Nous avons pu constater que de manière générale ce nombre pouvait être significativement plus petit que le majorant que nous étions en mesure de donner en prenant en compte uniquement la structure de la formule.

Après cette légère « pause », nous avons repris notre ascension en nous intéressant à la résolution pratique de fragments PSPACE de la logique propositionnelle modale. Nous avons proposé une nouvelle procédure de raffinement, nommée RECAR. L'idée est d'exploiter en même temps des sous-approximations et des sur-approximations du problème considéré en entrée, et de les faire « communiquer ». Afin de valider ce nouveau schéma de résolution, nous l'avons mis en œuvre pour traiter en pratique des problèmes modélisés en logique modale. Nos résultats expérimentaux ont montré que le solveur résultant, nommé MoSaiC, est très performant et qu'il surpasse les approches de l'état de l'art sur les instances que nous avons considérées.

Bien que notre ch'ti chercheur ait de nombreuses perspectives quant aux travaux présentés dans ce manuscrit, il est aussi stupéfait par la beauté et la richesse des contrées qui lui restent à explorer ...



Certains pessimistes prédisent un futur hiver de l'intelligence artificielle. Cependant, lorsque nous observons le paysage de la recherche, nous nous rendons compte qu'il y a encore de nombreuses régions à explorer. Il est difficile de prédire dans quel état sera l'intelligence artificielle dans dix ans, mais une chose est certaine : le futur nous offre plein de défis super palpitants.

Parmi ces défis, il est certain que l'intelligence artificielle explicable fera couler beaucoup d'encre dans les prochaines années. Il ne faut pas forcément associer le terme explicable au terme apprentissage. Il faut plutôt considérer ce terme d'une manière plus large au sens d'explication humainement compréhensible. Le rapport de l'être humain à la machine a toujours été difficile, et plus nous essaierons d'ajouter de l'intelligence à un système et plus ce dernier fera peur. Un utilisateur peut être effrayé par un véhicule autonome utilisant un système de navigation principalement basé sur l'utilisation d'un réseau de neurones, et pour lequel nous sommes incapables d'expliquer pourquoi dans une certaine situation c'est telle ou telle décision qui a été choisie. Cependant, puisqu'un réseau de neurones n'est rien d'autre qu'un circuit particulier, il est largement envisageable de rechercher des informations permettant d'expliquer une décision (Shih et al. 2019, Ignatiev et al. 2019, Narodytska et al. 2019). La difficulté sera alors, étant donné ces informations, d'apporter une explication du comportement du système qu'un humain pourra comprendre. En effet, nous avons déjà toute la machinerie pour vérifier de manière formelle une propriété, mais nous n'avons rien pour l'expliquer à un humain. Considérons par exemple le cas de la recherche d'une explication pour l'insatisfaisabilité d'une formule représentée en `CNF`. Dans ce contexte, il n'est pas rare de retourner à titre d'explication un sous-ensemble de la formule qui est minimalement insatisfaisable (`MUS`). Cependant, mettez-vous à la place d'un utilisateur qui reçoit comme explication un ensemble de clauses. Comment réagiriez-vous ? Le « mont » intelligence artificielle explicable est assez élevé et son sommet est encore dans le brouillard. De nombreux chercheurs sont déjà en route vers le sommet, ces pionniers nous permettront peut-être de voir plus clair dans un avenir proche, en traçant de nouvelles voies.

Un autre défi sera de faire cohabiter réseaux de neurones et problème de décision `NP`-difficile. Certaines tentatives ont déjà été réalisées dans ce sens, mais, de manière générale, elles conduisent à des procédures qui sont bien moins efficaces que les méthodes de l'état de l'art qui ne s'appuient pas sur l'apprentissage (Selsam et al. 2019). Il existe des cas où cette combinaison permet de produire une approche qui obtient de bons résultats, comme par exemple dans les travaux de Prates et al. (2019) sur le problème du voyageur de commerce, Mais alors pourquoi cela fonctionne dans

certains cas et pas dans d'autres ? De mon point de vue, le fait que ce type d'approche fonctionne pour le problème du voyageur de commerce est principalement dû au fait que c'est un problème bien spécifique pour lequel nous sommes capables d'extraire une information pertinente et exploitable du processus d'apprentissage. Dans le cas général nous n'avons pas cette connaissance, et par conséquent, il est souvent difficile d'extraire du réseau de neurones des informations utiles. Cet autre « mont » est aussi très élevé et pour atteindre son sommet il sera nécessaire de trouver quoi apprendre pour aider les solveurs.

La richesse de la recherche ne me permet pas de conclure par « un dernier défi ». En fait, il reste tant de défis techniques et intellectuels que je ne crois pas à un prochain hiver de l'intelligence artificielle. Peut-être que le terme sera moins à la mode, et que l'apprentissage sera moins au centre de l'attention, mais nous sommes, depuis l'avènement d'internet, entrés dans une ère du numérique. Nous avons de plus en plus de données, nous sommes de plus en plus connectés, et nous avons à résoudre des problèmes combinatoires de plus en plus difficiles. Avec l'arrivée de la `5G` cela va encore s'accélérer. Tout ceci laisse présager un futur scientifique passionnant ...

Deuxième partie
Curriculum vitae

Chapitre 1

Notice Individuelle

Sommaire

Affiliation

I currently hold a position of associate professor at Artois University.

Personal Data

Jean-Marie Lagniez

Born on November 29th, 1983 in Lens (Pas-de-Calais, France).

French national, single.

Contact information

CRIL-CNRS UMR8188
Institut Universitaire Technologique
rue de l'université, SP18
F-62307 Lens

+33 (0)3 21 79 32 77

langiez@cril.fr

http://www.cril.fr/~lagniez

Higher Education

2011 Doctorat en informatique, CRIL, Artois University (Lens, France), December 5th 2011

- Title : *Satisfiabilité propositionnelle et raisonnement par contraintes : modèles et algorithmes*
- Supervisors : Pr. Gilles Audemard, Pr. Bertrand Mazure and Pr. Lakhdar Saïs
- Funded by a doctoral fellowship from the French Ministry of Research

2008 Master 2 recherche en informatique (**mention très bien**), Artois University, Lens, France

2006 Licence en informatique (**mention bien**), Artois University, Lens, France

Professional Experience

From Sep 19 Maître de conférences, Artois University

Mar 13 – Sep 14 Postdoctoral position, CRIL research center, Artois University, Lens, France

Feb 12 – Mar 13 Postdoctoral position, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Sep 11 – Jan 12 Temporary Researcher and Teaching Assistant (ATER), Artois University, Lens, France

2008 – 2011 Doctoral position and higher education monitor, Artois University, Lens, France

Participation in Research Projects

2019 – 2023 ANR PING/ACK : Pre-traitement d'informations pour la résolution de tâches complexes/Compilation avancée de connaissances

2015 – 2019 ANR SATAS : SAT as Service

2018 PEPS INSMI-INS2I I3A SAT4EX : Deep Learning For Deep Solving

2018 PEPS S2IH INS2I SAT4EX : SAT based approach to extract EXplanation

2017 – 2018 Bilateral project France/Czech Republic on knowledge compilation for constraint programming

2017 PEPS JCJC INS2I DISCO : DIStributed COmpilation

2012 – 2013 RiSE project : Rigorous Systems Engineering

2008 – 2011 ANR Blanc UNLOC : Local Search and Unsatisfiability

Awards

2018 Gold medal of the CSP Minitrack at the 2018 XCSP3 Competition http://xcsp.org/competition

2015 Winner of the First International Competition on Computational Models of Argumentation http://argumentationcompetition.org/2015/results.html

2012 ICTAI Best Paper Award

2011 SAT Best Paper Award

Research Activities

I am mainly interested in developing, implementing and evaluating algorithms for the practical solving of instances of "Beyond NP" problems. This includes knowledge compilers (used for efficient model counting, a key issue for many forms of probabilistic reasoning), preprocessors (for simplifying the inputs), reasoners suited to modal logics, and other tools for generating minimal unsatisfiable subset/maximal consistent subsets, which are fundamental tasks for a number of artificial intelligence problems. I am also interested in developing efficient SAT solvers and other constraint programming engines.

Main topics : AI, knowledge representation, automated deduction, knowledge compilation, argumentation, constraint satisfaction problems (SAT and CSP), consistency restoration (MUS and and MCS)

Publications : 6 articles in international journals, 46 communications in the proceedings of international conferences. Among them are 33 papers in the following A* or A-ranked conferences : IJCAI, AAAI, ECAI, IJCAR, LPAR, SAT, CP

Design, implementation and evaluation of pieces of software (those which are still state-of-the-art today are spotted) :

- Compilers : EADT (www.cril.fr/KC/eadt.html), piFBDD, D4 (www.cril.fr/KC/d4.html : state-of-the-art), MDDG (www.cril.fr/KC/mddg.html state-of-the-art), and d-DNNF reasoner (www.cril.fr/KC/d-DNNF-reasoner.html) :
- Projected Model Counter : projMC (www.cril.fr/KC/projmc.html : state-of-the-art)
- Parallel Model Counter : DMC (www.cril.fr/KC/dmc.html : state-of-the-art)
- MCS Extractor and Enumerator : CMP (www.cril.fr/documents/cmp/) and Enum-MCSes (www.cril.fr/enumcs/ : state-of-the-art)
- Preprocessors for model counting : PMC (www.cril.fr/KC/pmc.html) and B+E (www.cril.fr/KC/bpe2.html : state-of-the-art)
- Incremental SAT Solvers : MiniAbbreviation and Glucose (www.labri.fr/perso/lsimon/glucose/ : state-of-the-art)
- Local Search Solvers : SAT-WalkSat, CSP-WalkSat and PB-WalkSat
- Backtracking Solvers : Nacre (www.cril.fr/~glorian/), RCL-SAT, MinisatPsmDyn and RCL-CSP

- Hybrid Solvers : SATHYS and FAC-Solver (CSP)
- Parallel Solvers : D-Syrup (www.cril.fr/dsyrup/ : state-of-the-art) and Ampharos (www.cril.fr/ampharos/)
- Modal Logic Solver : MoSaic (www.cril.fr/~montmirail/mosaic/ : state-of-the-art)
- Abstract Argumentation Solver : CoQuiAAS (www.cril.fr/coquiaas/)
- Bayesian Network Encoder : bn2cnf (www.cril.fr/KC/bn2cnf.html : state-of-the-art)
- Graph Coloring : Picasso (www.cril.fr/Picasso : state-of-the-art)
- Belief Revision and Belief Merging : br2cnf (www.cril.fr/KC/br2cnf.html : state-of-the-art) and bm2cnf (www.cril.fr/KC/bm2cnf.html : state-of-the-art)

Co-supervision of Ph.D. Students

2019 –	Marie Micelli "Explication via les « weak cores » " (on-going)
2016 –	Gaël Glorian "Optimisation distribuée pour les problèmes combinatoires sous contraintes de grande taille" (on-going)
2015 – 2018	Yacine Izza "Informatique ubiquitaire : techniques de curage d'informations perverties" (defended)
2015 – 2018	Valentin Montmirail "Résolution pratique de la cohérence de formules en logique modale" (defended)
2014 – 2017	Nicolas Szczepanski "SAT en Parallèle" (defended)
2012 – 2016	Samuel Thomas "Compilation de connaissances pour le comptage de modèles" (defended)

Elective Positions

2019–2020	Invited member of the AFIA board (Association Française pour l'Intelligence Artificielle)
2017–2021	Elected member of the AFPC board (Association Française pour la Programmation par Contraintes)
2008–2011	Representative of PhD students and postdoc, laboratory council (CRIL, France)
2008–2009	Engineering Sciences graduate school council (Lille, France)

Research Assessment

- Coordinator of the working group of the 'GDR IA' (CNRS research group) on Representation and Algorithms in Practice (RAP)
- Participation in 18 program committees of international conferences or workshops (the last ones are AAAI'18, ICTAI'19 (co-chair of the SAT/CSP track), CP'19 and IJCAI'19)
- Reviewer for international conferences and journal : ICTAI, SAT, CP, DATE, CAV, SAT, ICALP, JSAT, Constraint and IJAIT

Publications

— Audemard, Gilles ; Lagniez, Jean-Marie ; Mazure, Bertrand ; Sais, Lakhdar: Integrating Conflict Driven Clause Learning to Local Search. In: *Proceedings 6th International Workshop on Local Search Techniques in Constraint Satisfaction, LSCS 2009, Lisbon, Portugal, 20 September 2009.*, 2009, p. 55–68

— Audemard, Gilles ; Lagniez, Jean-Marie ; Mazure, Bertrand ; Sais, Lakhdar: Learning in Local Search. In: *ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, 2-4 November 2009*, 2009, p. 417–424

— Audemard, Gilles ; Lagniez, Jean-Marie ; Mazure, Bertrand ; Sais, Lakhdar: Boosting Local Search Thanks to CDCL. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, 2010, p. 474–488

— Audemard, Gilles ; Lagniez, Jean-Marie ; Mazure, Bertrand ; Sais, Lakhdar: On Freezing and Reactivating Learnt Clauses. In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, 2011, p. 188–200

— Guo, Long ; Lagniez, Jean-Marie: Dynamic Polarity Adjustment in a Parallel SAT Solver. In: *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, 2011, p. 67–73

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: A CSP Solver Focusing on FAC Variables. In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, 2011, p. 493–507

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Relax! In: *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, 2012, p. 146–153

— Audemard, Gilles ; Hoessen, Benoît ; Jabbour, Saïd ; Lagniez, Jean-Marie ; Piette, Cédric: Revisiting Clause Exchange in Parallel SAT Solving. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, p. 200–213

— Audemard, Gilles ; Lagniez, Jean-Marie ; Simon, Laurent: Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 309–317

— Lagniez, Jean-Marie ; Biere, Armin: Factoring Out Assumptions to Speed Up MUS Extraction. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 276–292

— Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre ; Thomas, Samuel: Knowledge Compilation for Model Counting: Affine Decision Trees. In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 947–953

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Preserving Partial Solutions While Relaxing Constraint Networks. In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 552–558

— Audemard, Gilles ; Lagniez, Jean-Marie ; Simon, Laurent: Just-In-Time Compilation of Knowledge Bases. In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 447–453

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Questioning the Importance of WCORE-Like Minimization Steps in MUC-Finding Algorithms. In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, 2013, p. 923–930

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Improving MUC extraction thanks to local search. In: *CoRR* abs/1307.3585 (2013)

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Multiple Contraction through Partial-Max-SAT. In: *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, 2014, p. 321–327

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Enforcing Solutions in Constraint Networks. In: *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic*, 2014, p. 1017–1018

— Bart, Anicet ; Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre: Symmetry-Driven Decision Diagrams for Knowledge Compilation. In: *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, 2014, p. 51–56

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: A General Artificial Intelligence Approach for Skeptical Reasoning. In: *Artificial General Intelligence - 7th International Conference, AGI 2014, Quebec City, QC, Canada, August 1-4, 2014. Proceedings*, 2014, p. 33–42

— Lagniez, Jean-Marie ; Marquis, Pierre: Preprocessing for Propositional Model Counting. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 2688–2694

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: An Experimentally Efficient Method for (MSS, CoMSS) Partitioning. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 2666–2673

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: Boosting MUC extraction in unsatisfiable constraint networks. In: *Appl. Intell.* 41 (2014), Nr. 4, p. 1012–1023

— Grégoire, Éric ; Lagniez, Jean-Marie: On Anti-subsumptive Knowledge Enforcement. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, 2015, p. 48–62

— Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre ; Thomas, Samuel: Compiling Constraint Networks into Multivalued Decomposable Decision Graphs. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, p. 332–338

— Lagniez, Jean-Marie ; Lonca, Emmanuel ; Mailly, Jean-Guy: CoQuiAAS: A Constraint-Based Quick Abstract Argumentation Solver. In: *27th IEEE International Conference on*

Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015, 2015, p. 928–935

— Besnard, Philippe ; Grégoire, Éric ; Lagniez, Jean-Marie: On Computing Maximal Subsets of Clauses that Must Be Satisfiable with Possibly Mutually-Contradictory Assumptive Contexts. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 2015, p. 3710–3716

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: On getting rid of the preprocessing minimization step in MUC-finding algorithms. In: *Constraints* 20 (2015), Nr. 4, p. 414–432

— Grégoire, Éric ; Konieczny, Sébastien ; Lagniez, Jean-Marie: On Consensus Extraction. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, p. 1095–1101

— Lagniez, Jean-Marie ; Lonca, Emmanuel ; Marquis, Pierre: Improving Model Counting by Leveraging Definability. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, p. 751–757

— Grégoire, Éric ; Lagniez, Jean-Marie: A Computational Approach to Consensus-Finding. In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands*, 2016, p. 795–801

— Bart, Anicet ; Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre: An Improved CNF Encoding Scheme for Probabilistic Inference. In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, 2016, p. 613–621

— Grégoire, Éric ; Lagniez, Jean-Marie ; Zhang, Du: On Computing Non-hypocritical Consensuses in Standard Logic. In: *27th International Workshop on Database and Expert Systems Applications, DEXA 2016 Workshops, Porto, Portugal, September 5-8, 2016*, 2016, p. 97–101

— Audemard, Gilles ; Lagniez, Jean-Marie ; Szczepanski, Nicolas ; Tabary, Sébastien: An Adaptive Parallel SAT Solver. In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, 2016, p. 30–48

— Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: On Checking Kripke Models for Modal Logic K. In: *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016.*, 2016, p. 69–81

— Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie: On the Extraction of One Maximal Information Subset That Does Not Conflict with Multiple Contexts. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, p. 3404–3410

— Grégoire, Éric ; Lagniez, Jean-Marie ; Zhang, Du: Logical consensuses for case-based reasoning and for mathematical engineering of AI. In: *15th IEEE International Conference on Cognitive Informatics & Cognitive Computing ,ICCI*CC 2016, Palo Alto, CA, USA, August 22-23, 2016*, 2016, p. 29–33

— Grégoire, Éric ; Lagniez, Jean-Marie: RCL: An A. I. Tool for Computing Maximal Consensuses. In: *International Journal on Artificial Intelligence Tools* 25 (2016), Nr. 4, p. 1–10

— Grégoire, Éric ; Lagniez, Jean-Marie: A Computational Method for Enforcing Knowledge that Cannot be Subsumed. In: *International Journal on Artificial Intelligence Tools* 25 (2016), Nr. 4, p. 1–19

— Audemard, Gilles ; Lagniez, Jean-Marie ; Szczepanski, Nicolas ; Tabary, Sébastien: A Distributed Version of Syrup. In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 215–232

— Konieczny, Sébastien ; Lagniez, Jean-Marie ; Marquis, Pierre: Boosting Distance-Based Revision Using SAT Encodings. In: *Logic, Rationality, and Interaction - 6th International Workshop, LORI 2017, Sapporo, Japan, September 11-14, 2017, Proceedings*, 2017, p. 480–496

— Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: A Recursive Shortcut for CEGAR: Application To The Modal Logic K Satisfiability Problem. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, p. 674–680

— Lagniez, Jean-Marie ; Marquis, Pierre: An Improved Decision-DNNF Compiler. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, p. 667–673

— Lagniez, Jean-Marie ; Marquis, Pierre ; Paparrizou, Anastasia: Defining and Evaluating Heuristics for the Compilation of Constraint Networks. In: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 172–188

— Glorian, Gael ; Boussemart, Frédéric ; Lagniez, Jean-Marie ; Lecoutre, Christophe ; Mazure, Bertrand: Combining Nogoods in Restart-Based Search. In: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 129–138

— Caridroit, Thomas ; Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: A SAT-Based Approach for Solving the Modal Logic S5-Satisfiability Problem. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, p. 3864–3870

— Konieczny, Sébastien ; Lagniez, Jean-Marie ; Marquis, Pierre: SAT Encodings for Distance-Based Belief Merging Operators. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, p. 1163–1169

— Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie: On Computing One Max_Subset Inclusion Consensus. In: *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, 2017, p. 838–845

— Lagniez, Jean-Marie ; Marquis, Pierre: On Preprocessing Techniques and Their Impact on Propositional Model Counting. In: *J. Autom. Reasoning* 58 (2017), Nr. 4, p. 413–481

— Lagniez, Jean-Marie ; Marquis, Pierre ; Szczepanski, Nicolas: DMC: A Distributed Model Counter. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, p. 1331–1338

— Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie: Boosting MCSes Enumeration. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, p. 1309–1315

— Glorian, Gael ; Lagniez, Jean-Marie ; Montmirail, Valentin ; Sioutis, Michael: An Incremental SAT-Based Approach to Reason Efficiently on Qualitative Constraint Networks. In: *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, 2018, p. 160–178

— Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: An Assumption-Based Approach for Solving the Minimal S5-Satisfiability Problem. In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, 2018, p. 1–18

— Grégoire, Éric ; Lagniez, Jean-Marie ; Zhang, Du: Consensus-finding that preserves mutually conflicting hypothetical information from a same agent. In: *AI Commun.* 31 (2018), Nr. 3, p. 303–317

— Schwind, Nicolas ; Okimoto, Tenda ; Inoue, Katsumi ; Hirayama, Katsutoshi ; Lagniez, Jean-Marie ; Marquis, Pierre: Probabilistic Coalition Structure Generation. In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018.*, 2018, p. 663–664

— Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: A SAT-Based Approach For PSPACE Modal Logics. In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018.*, 2018, p. 651–652

— Schwind, Nicolas ; Inoue, Katsumi ; Konieczny, Sébastien ; Lagniez, Jean-Marie ; Marquis, Pierre: What Has Been Said? Identifying the Change Formula in a Belief Revision Scenario. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, p. 1865–1871

— Lagniez, Jean-Marie ; Marquis, Pierre: A Recursive Algorithm for Projected Model Counting. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 1536–1543

— Glorian, Gaël ; Lagniez, Jean-Marie ; Montmirail, Valentin ; Szczepanski, Nicolas: An Incremental SAT-Based Approach for Graph Colouring Problem. In: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, Connecticut, U.S., September 30–4 October, Proceedings*, 2019

Troisième partie
Sélection de publications

Chapitre 1

Liste des articles

Contributions à la résolution de SAT en parallèle

— Audemard, Gilles ; Lagniez, Jean-Marie ; Szczepanski, Nicolas ; Tabary, Sébastien: An Adaptive Parallel SAT Solver. In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, 2016, p. 30–48

— Lagniez, Jean-Marie ; Biere, Armin: Factoring Out Assumptions to Speed Up MUS Extraction. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 276–292

Contributions au calcul d'ensembles maximalement cohérents

— Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand: An Experimentally Efficient Method for (MSS, CoMSS) Partitioning. In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 2666–2673

— Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie: Boosting MCSes Enumeration. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, p. 1309–1315

Contributions à la compilation de connaissances et au comptage de modèles

— Lagniez, Jean-Marie ; Lonca, Emmanuel ; Marquis, Pierre: Improving Model Counting by Leveraging Definability. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, p. 751–757

—

— Lagniez, Jean-Marie ; Marquis, Pierre: A Recursive Algorithm for Projected Model Counting. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 1536–1543

— Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre ; Thomas, Samuel: Knowledge Compilation for Model Counting: Affine Decision Trees. In: *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 947–953

Contributions à la résolution du problème de la satisfaisabilité des formules en logique modale

— Caridroit, Thomas ; Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: A SAT-Based Approach for Solving the Modal Logic S5-Satisfiability Problem. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, p. 3864–3870

— Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin: A Recursive Shortcut for CEGAR: Application To The Modal Logic K Satisfiability Problem. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, p. 674–680

Chapitre 2

An Adaptive Parallel SAT Solver

Article paru dans les actes de « *the 22nd International Conference on Principles and Practice of Constraint Programming* » (CP'16), pages 30 – 48, septembre 2016.

Co-écrit avec Gilles Audemard, Nicolas Szczepanski et Sébastien Tabary.

An adaptive parallel SAT solver

Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary

Univ. Lille-Nord de France. CRIL/CNRS UMR 8188, Lens, {name}@cril.fr

Abstract. We present and evaluate AMPHAROS, a new parallel SAT solver based on the divide and conquer paradigm. This solver, designed to work on a great number of cores, runs workers on sub-formulas restricted to cubes. In addition to classical clause sharing, it also exchange extra information associated to the cubes. Furthermore, we propose a new criterion to dynamically adapt both the amount of shared clauses and the number of cubes. Experiments show that, in general, AMPHAROS correctly adjusts its strategy to the nature of the problem. Thus, we show that our new parallel approach works well and opens a broad range of possibilities to boost parallel SAT solver performances.

1 Introduction

Papers dealing with SAT solvers usually begin by recalling the tremendous progress achieved on problems coming from industry. Recent results are indeed very impressive, and a large number of industrial problems, *e.g.* from planning [35], formal verification [11] and cryptography [40] are nowadays solved using a reduction to SAT instead of ad-hoc solvers. However, playing the devil's advocate, one can observe that progress has slowed down noticeably. It has become harder and harder to improve solvers dramatically. Furthermore, SAT suffers from its own success, since formulas to solve are more and more difficult.

At the same time, cloud computing is changing the landscape of computing science: it is now possible to request a virtually unlimited number of computing units that can be used within a few seconds. However, as it was pointed out during the last competition [37], parallel SAT solvers are not well scalable. Indeed, the winner of the parallel SAT track chose to only use half of the available cores. Thus, to benefit from the huge number of computing units, as in a cloud context, one must design new solvers architectures.

In the case of SAT solving, solvers can be divided into two categories. First and foremost, portfolio based approaches [1,8,13,23,24,36] run different strategies/heuristics concurrently, each on the whole formula. While computing the processes exchange information (generally in the form of learnt clauses) to help each other [1,7,23,24]. The second category of solvers uses the well known divided and conquer paradigm [2,15,16,25,26,39,41,42]. In such solvers, the search space is divided into sub-spaces, which are successively sent to SAT solvers running on different processors, so called workers. In general, each time a solver finishes its job (while the others are still working), a load balancing strategy is invoked, which dynamically transfers sub-spaces to this idle worker [15,16]. The sub-spaces can be defined using the guiding path concept [42], generated statically, *i.e.*, before the search [25,39], or dynamically, *i.e.*, during the search process [2,26,41]. As in portfolio solvers, learnt clauses can also be shared [18].

Even though the winners of the parallel track of the last SAT competitions are based on the portfolio paradigm, solvers based on the divide and conquer approach become increasingly more efficient (TREENGELING [12] a solver based on this paradigm was ranked second in the last competition). It is in this context that we propose AMPHAROS, a new parallel SAT solver, which follows the divide and conquer approach. Our long term objective is to develop a SAT solver for the cloud and this paper is a first step in this direction. In our approach, the formula is partitioned using cubes (as in [41]). One process, named MANAGER, is dedicated to managing these cubes. Then, solvers work on the formulas induced by those cubes. In constrast to other divide and conquer approaches, several solver may work on the same sub-problem and they can stop working before finding a solution or a contradiction. The latter is to avoid solvers being stuck on instances that turn out to be too hard for them. In that case, the solver asks the manager for another sub-problem. This sub-problem can either originate from an existing cube or from refining the current sub-problem. In our approach, the solvers select by themselves the dynamically generated cubes they try to solve. Additionally, two types of learnt clauses are shared: the classical shared clauses and others that are dependent on the cubes.

Since our goal is to solve SAT with a great number of computing units, it is important to propose a parallel architecture which adapts its strategy to the number of workers and the nature of the problem. To this end, we propose an approach which uses an adaptive algorithm that adjusts simultaneously and dynamically the number of clauses that are shared and the number of new cubes. This is possible thanks to a new measure that estimates if the search process has to be intensified or diversified. As we demonstrate in experiments, this measure works well and aligns with the stated goal. We show that when the search space needs to be diversified (resp. intensified), the proposed measure detects that the number of cubes must be increased (resp. decreased) and the number of shared clauses decreased (resp. increased).

2 Preliminaries

Due to lack of space, we assume the reader to be familiar with the essentials of propositional logic and SAT solving. Let us just recall some aspects of CDCL SAT solvers [32,30]. CDCL solving is a branching search process, where at each step a literal is selected for branching. Usually, the variable is picked w.r.t. the VSIDS heuristic [32] and its value is taken in a vector, called polarity vector, in which the previous value assigned to the variable is stored [34]. Afterwards, Boolean constraint propagation is performed. When a literal and its opposite are propagated, a conflict is reached, a clause is learnt from this conflict [30] and a backjump is executed. These operations are repeated until a solution is found or the empty clause is derived.

CDCL SAT solvers can be enhanced by considering restart strategies [20] and deletion policies for learnt clauses [3,6,19]. Among the measures proposed to identify the relevant clauses, the literal blocked distance measure (in short LBD) [6] is one of the most efficient. The clause's LBD corresponds to the number of different levels involved in a given learnt clause. Then, as experimentally shown by the authors of [6], clauses with smaller LBD should be considered more relevant.

It is well known that for several applications it is necessary to solve many similar instances [5,9,17]. To make solvers more effective in such a context, it is particularly useful to use assumptions to keep track of learnt clauses during the whole search. A set of assumptions is defined as a set of literals that are assumed to be true [17]. This set can be viewed as a cube, i.e. a cunjunction of literals (in the remainder of this paper, we denote cubes using square brackets, also we sometimes identify cubes with the formulas they imply), and the search is restricted to this cube. If during the search process, one needs to flip the assignment of one of these assumptions to false, the problem is unsatisfiable under the initial assumptions. In such a situation, it is possible to recursively traverse the implication graph to extract a clause that explains the reason of the conflict. Even if this problem seems close to the classical SAT problem, a special track of the last SAT competition has been dedicated to this issue [37] and several existing studies attempt to improve SAT solvers to deal with assumptions [4,28,33].

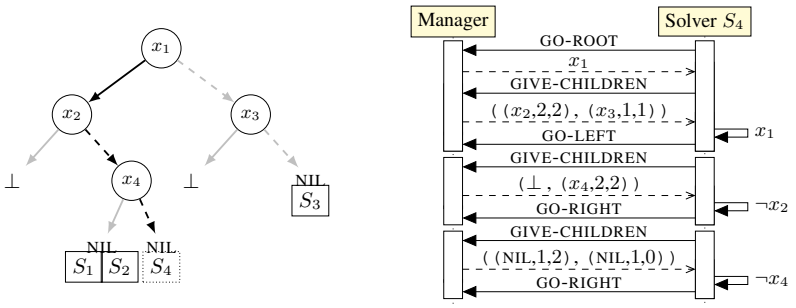
3 Tree management

The performance of divide and conquer approaches depends on both, the quality of the search space splitting, and how the sub-spaces are assigned to the solvers.

Even if AMPHAROS is a divide and conquer based solver, it is important to stress that, contrary to [38], it does not use the work stealing strategy. In our case, the division is done in a classical way as in [2,16]. More precisely, our approach generates guiding paths, restricted to cubes, that cover all the search space. This way, the outcome of the division is a tree where nodes are variables and the left (resp. right) edge corresponds to the assignment of the variable to true (resp. false). Then, solvers operate on leaves (represented by the symbol NIL) and solve (under assumptions) the initial formula restricted to a cube which corresponds to the path from the root to the related leaf. Fig. 1a shows an example of a tree containing three open leaves (cubes $[x_1,\neg x_2,x_4]$, $[x_1,\neg x_2,\neg x_4]$ and $[\neg x_1,\neg x_3]$), two closed branches (already proven unsatisfiable) and four solvers ($S_1 \dots S_4$) working on these leaves.

As we will see in Sect. 3.2, in our architecture, solvers can work on the same cube (as solvers S_1 and S_2 in Fig. 1a) and can stop working before finding a solution or a contradiction. In AMPHAROS, each time a solver shares information or asks to solve a new cube, it communicates with a dedicated worker, called MANAGER. Its main mission is to manage the cubes and the communication between the solvers (here CDCL solvers). Thus, when a solver decides to stop solving a given cube (without having solved the instance), it can ask the MANAGER to enlarge this one (see Sect. 3.3). Another situation where a solver stops, is once a branch is proved to be unsatisfiable. In this case, a message informs the MANAGER and the tree is updated in consequence (see section 3.4). In both cases, when a solver stops it goes through the tree and starts solving a new cube (potentially the same, see Sect. 3.2). The end of the solving process finally occurs either when a cube is proved to be satisfiable or when the tree is proved to be unsatisfiable.

This section describes the overall picture of our solver. First, in Sect. 3.1, the way the tree is initialized is presented. Then, the transmission and extension processes are respectively explained in Sect. 3.2 and Sect. 3.3. Finally a tree pruning rule is introduced in Sect. 3.4.



(a) Current tree (b) Sequence diagram

Fig. 1: Schematic overview of how the solver S_4 and the MANAGER interact to select a cube in the current tree represented Fig. 1a (a plain (resp. dotted) line means that the variable is assigned to true (resp. false)). On the sequence diagram, in Fig. 1b, we can see that seven messages are exchanged between the MANAGER and the solver before S_4 starts to solve the sub-problem induced by $[x_1, \neg x_2, \neg x_4]$. The path selectioned by S_4 is represented with black lines in the left picture.

3.1 Initialization

At the beginning of the search process, we initialize the workers. This step is required to setup the activity (related to VSIDS heuristic [32]), the polarity of variables and to create the root of the tree. To this end, all solvers try to solve the whole formula concurrently until a given amount of conflicts is reached (10,000 in our implementation). Note that this corresponds to solve an empty cube. In order to avoid performing the same search, the first descent of each solver (*i.e.* the choice of the variables and their polarity on the first branch) is randomized. Then, in the same manner as [31], the first solver reaching the maximum number of conflicts communicates its best variable with respect to the VSIDS heuristic to the MANAGER. This variable becomes the root of the tree. Consequently, the tree only contains two leaves, *i.e.* cubes are restricted to a single literal (a variable and its opposite). Regarding Fig. 1a, the selected variable was x_1 and the set of initial cubes was $\{[x_1], [\neg x_1]\}$.

3.2 Transmission

As already mentioned, a solver may stop the search before solving its instance. This situation occurs when it cannot solve the sub-problem associated to the cube with a number of conflicts less than a certain limit (10,000 in our implementation). The solver then contacts the MANAGER in order to select a potentially new cube to solve. The originality of our method is that a solver selects by itself one cube among all unsolved ones in the tree (corresponding to NIL leaves).

Fig. 1 shows a diagram sequence (Fig. 1b) that illustrates the exchanged messages when the solver S_4 requests a new cube from the MANAGER's tree (Fig. 1a).

A first message (GO-ROOT) is sent by the solver to ask for the root of the tree. It receives x_1. Then, at each step of the cube selection, the solver asks for the children of the previously received variable (with message GIVE-CHILDREN). The answer is composed of two triplets: one for each polarity of the current node. Each triplet is composed of

the child variable, the number of available leaves (NIL nodes) and the number of solvers working on these leaves, in that order. Considering Fig. 1b, the first message returns the triplet $(x_2,2,2)$ for the positive polarity of x_1 (the left branch contains two leaves and two solvers (S_1 and S_2)) and $(x_3,1,1)$ for the negative one.

The solver decides to go down either on the left (assigning positively the current variable) or on the right (assigning negatively the current variable) branch according to the values returned in these triplets. By default, it selects the branch where the number of working solvers is lower than the number of leaves. The idea is to cover the most of cubes and to dispatch solvers all over the tree. If this condition is true or false for both branches, the solver selects the branch according to its polarity vector [34]. Note that in this implementation, we do not know if some cubes do not contain solvers. After selecting its branch, the solver informs the MANAGER (with messages GO-LEFT or GO-RIGHT) and assigns the related literals using assumptions.

Thus, in our example of Fig. 1, the solver S_4 assigns x_1 (the root) positively using its polarity (as the condition previously mentioned is false for both branches). Since the branch related to x_2 is already proven unsatisfiable, S_4 does not have other alternatives to setting the literal x_2 to false. Finally, it has to set x_4 to false since the previous condition holds. Arriving at a leaf, the solver starts to solve the cube $[x_1,\neg x_2,\neg, x_4]$.

3.3 Extension

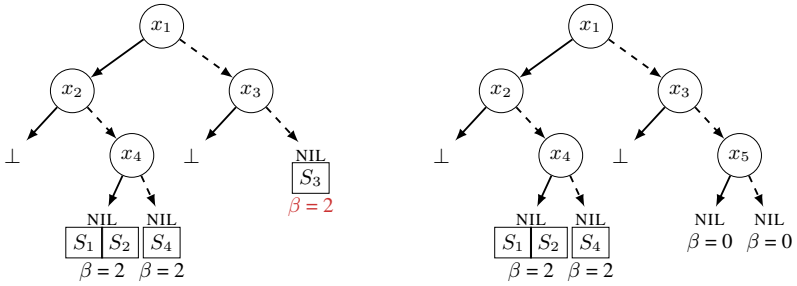
Initially, the tree contains only one variable and then two cubes to solve (see Sect. 3.1). To divide the original formula into a substantial number of cubes, we propose to dynamically extend the tree during the search. Recall that we do not use the work stealing strategy.

One associates to each leaf an integer variable β representing the presumed difficulty of a subproblem (cube). Each time a solver cancels its search on a given cube (associated with a leaf of the tree), the variable β of this leaf is incremented. Then, a large value of β expresses that a cube is potentially hard to solve. Note that a solver can increase several times the same variable β. When a solver stops its search and requests a new cube, the MANAGER increments the value β associated to the leaf on which the solver was working. When the β value of a leaf is greater or equal than the number of open leaves (*i.e.* NIL leaves) times an extension factor f_e then the tree is expanded on the given leaf. [1]

The extension is done in the following way. The last solver increasing the variable β returns its best boolean variable w.r.t. to VSIDS heuristic and two new leaves are created, extending the related cube. The β values of the two leaves are initialized to 0. Taking into account the number of open leaves, the more unsolved cubes the tree contains, the less extensions are performed. In this way and contrary to Cube And Conquer [41], our approach does not create too many cubes, regardless of the number of cubes already proven unsatisfiable. Since a leaf can contain many solvers, note that after extension, some solvers can work on a node that is not a leaf.

Fig. 2 shows an example of an extension. The tree (the same as in Fig. 1) contains 3 open leaves and some solvers work on these leaves. When, solver S_3 stop working on

[1] We will discuss about the definition of the extension factor in section 5.2.



(a) Extension requested by the solver S_3

(b) Extension accepted by the MANAGER

Fig. 2: The left picture represents the tree before the S_3's extension request was accepted. Since the value of β associated to the node satisfied the extension criterion, the MANAGER accepts this extension and modified the left tree to obtain the right one.

cube $[\neg x_1, \neg x_3]$ the associated β (in red) is incremented and becomes 3. The condition allowing an extension holds (we suppose that f_e is equal to one) and thus extension is performed. Solver S_3 that is responsible for the extension provides its best variable (x_5) to the MANAGER and the cube $[\neg x_1, \neg x_3]$ is expanded with the variable x_5 generating two new cubes. Note that the (red) β value initiator of this extension becomes useless since its associated node is not a leaf anymore. Solver S_3 is now free to ask the MANAGER a new cube to solve (see Sect. 3.2). Furthermore, in the next step, no matter which solver ask for extension, it will not be performed since the number of leaves is now equal to 4 (we suppose here that f_e remains unchanged and is still equal to 1).

3.4 Pruning

Because each sub-problem is solved under assumptions, when a cube is proved to be unsatisfiable, the solver (from which unsatisfiability is proved) computes a conflict clause (which is the negation of a subset of the literal assumptions). This information is transfered to the MANAGER which is able to compute a cutoff level in the tree search. The tree is simplified in consequence. Let us remark that a solver can directly prove the global unsatisfiability of the problem when the computed conflict clause is empty.

Moreover, if both children of a node are unsatisfiable then this node also becomes unsatisfiable. In that case, the node can be safely removed and the unsatisfiability is directly associated to the edge of its parent. Of course, this process is recursively applied until each node has at least one non-unsatisfiable child.

4 Clause exchange

In this section, we discuss the two ways of exchanging information in our solver AMPHAROS. We first explain how the clauses learnt by a solver are shared with the others and then we present an original approach to sharing local unit literals by taking advantage of our tree.

4.1 Classical Clause Sharing

It is well known that clause sharing noticeably improves the performance of parallel SAT solvers [24]. In our framework, solvers also share learnt clauses. However, contrary to the classical behavior where the clauses are directly shared between workers, for us information passes through the MANAGER.

Clause sharing from the solver side Once a solver reaches a threshold of conflicts (500 in our implementation), it communicates with the MANAGER to send and/or receive a set of clauses. Clauses to be sent are saved in a buffer which is cleared after each communication with the MANAGER. Good clauses with respect to initial LBD (less or equal to 2) are directly added to the buffer. Other clauses are also added, as in [7], if they participate in the conflict analysis. However, because we cannot share as many clauses as SYRUP, only clauses which obtain a dynamic LBD less or equals to 2 before being used twice in the conflict analysis procedure are shared.

In order to deal with imported clauses, solvers manage three buffers: `standby`, `purgatory` and `learnt`. Received clauses are stored in `standby`. In this buffer, clauses are not attached to the solver [3]. Every $4{,}000$ conflicts, clauses are reviewed: they can be transfered from a buffer to another, or be definitively deleted or kept in the current buffer.

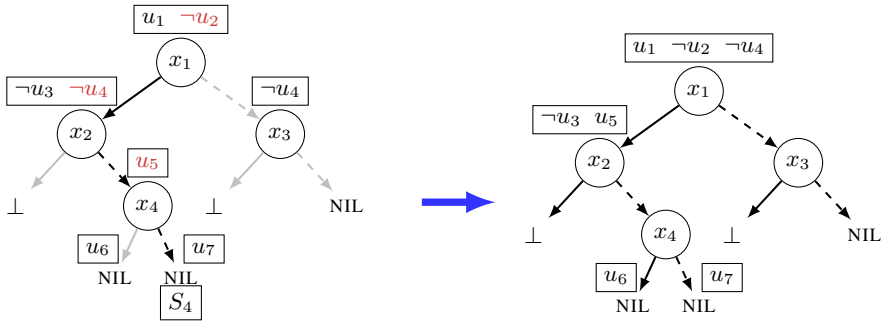
A clause from the `standby` buffer can be transfered to the `purgatory` buffer. Contrary to the `standby` buffer, clauses in the `purgatory` are attached to the solver and then participate to the unit propagation process. We discuss the criterion allowing a clause to be moved from `standby` to `purgatory`in Sect. 5.2.

In the same manner, a clause from the `purgatory` can be transfered to the third buffer `learnt` when it is used at least once in the conflict analysis process. The temporary buffer `purgatory` is used to limit the impact of new clauses on the learnt strategy reduction.

The reduction strategy used to clean these two additional buffers depends on a counter associated with each clause. The counter is incremented each time the associated clause remains in the same buffer. If the counter reaches a threshold (14 in our implementation), the clause is deleted. Note that the counter is reset each time a clause is moved from one buffer to another.

Clause sharing from the MANAGER side MANAGER collects learnt clauses from every solver and manages them. Learnt clauses are stored in a queue and the MANAGER periodically checks if they are subsumed or not. In practice, a single core is dedicated to the MANAGER. Thus, processing all clauses in the queue at once can be time consuming and can block communications between MANAGER and solvers. To avoid this situation, MANAGER checks subsumption by batches of $1{,}000$ clauses each.

MANAGER stores the learnt clauses that are not subsumed in a database and sends them each time a solver requests them. Of course, sent clauses are those that have not been already sent to the solver and that are not coming from it.



(a) S_4 gives $\neg u_2$ $\neg u_4$ u_5 to MANAGER (b) Unit literals management

Fig. 3: The left picture represents a decorated tree with the additional literals (in red) given by S_4. These additional literals are pull up, using the unsat (pull up u_5) and identical literals (pull up u_4) rules, to obtain the right tree.

4.2 Assumptive Unit Literals

A second way of exchanging information in our approach is transferring unit literals (which are propagated under some assumptive literals) between solvers and the MANAGER. In the following, we present where these literals originate from and how they are exchanged and managed.

Assumptive Unit Literals from the solver side Let us first recall that each solver works under an assumption A (this assumption can be empty) representing the cube to solve. When a literal $\ell \notin A$ is propagated thanks to a sub-assumption $A' \subseteq A$, this information can be spread to the MANAGER in order to be broadcasted to other solvers. More precisely, the solver communicates to the MANAGER that ℓ can be propagated with A'. From the other side, when a solver selects a branch (ie a literal ℓ') during cube transmission, it also receives the set of unit literals associated with ℓ' that can be propagated. Thus, the transmission of a cube (see Sect. 3.2) contains these additional messages. Hence, MANAGER takes care of a decorated tree containing guiding paths and the set of unit literals that have to be propagated at each branch. Figure 3.a shows an example of such a tree. When solver S_4 asks for a branch, it starts by recovering the set of unit literals $\{u_1\}$. It also propagates $\neg u_2$ (in red) giving this information to the MANAGER. It selects the branch x_1 and then, retrieves the literal $\neg u_3$ to propagate. in the same way, it also propagates $\neg u_4$, providing such assumptive unit literal to the MANAGER and so on.

As we will see in the experiments later, assumptive literals are very important. They are special clauses that clearly reduce the search space of a given branch. Consequently, the fact that a literal ℓ can be propagated from A' is taken into account in the solver by adding, in a dedicated database (this database is different from the aforementioned `learnt` buffer and is never cleaned up), a clause built with the negation of A' and the literal ℓ'. Remark that when $A' = \emptyset$ the literal ℓ' is unit and is added to the unit literals of the solver.

Assumptive Unit Literals from the MANAGER side When the MANAGER learns that a literal can be propagated from a subset of literals coming from an assumption, this information is communicated during cube's transmission and can be added in the last branch of the node associated with this sub-assumption. From this decorated tree, one can pull up unit literals from a branch to higher branches. This situation occurs either when a branch is proved unsatisfiable or when both branches of a node contain the same literal [29] (as highlighted Fig. 3). In the first case, all the literals of the non-unsatisfiable branch are pulled up to the father branch (as literal u_5). In the second case, only literals occurring in both branches are transmitted to the father branch (this is the case for literal $\neg u_4$). This process loops recursively until a fix point is reached. Remark that when no father branch exists (occurring when literals are moved from branches of the root node) then these literals are proved unit.

5 The Intensifcation/Diversification Dilemma

When several solvers run concurrently on a problem, they can perform redundant work. Identifying such a situation, it would be beneficial to modify the solvers' strategies in order to diversify the search. Nevertheless, due to clause sharing between solvers, exploring too different search spaces is also a handicap. Thus, in some situation focusing several solvers on the same part is required (intensification).

This paradigm, called intensification/diversification dilemma, has already been studied in the context of portfolio-based parallel SAT solvers. This issue can be addressed either statically, by using several solvers with orthogonal strategies [1,24,36], or dynamically, by modifying the solvers' strategies during the search. However, deciding when a solver must intensify or diversify its search is not easy and only few publications tried to deal with this problem [21,22]. Thus, in [21], a master/slave architecture is proposed, where masters try to solve the original problem (ensuring diversification), whereas slaves intensify their master's strategy. In [22], a measure to estimate the degree of redundancy between two solvers is presented. It considers that two solvers are closed when they have approximately the same polarity vector. The diversification process consists in modifying the way the phase of the next decisions is realized.

To the best of our knowledge, no criterion has been established to identify that several solvers execute redundant work except the measure based on the polarity mentioned before [22]. Unfortunately, this criterion is not applicable with many solvers (this measure has been initially proposed for a portfolio of four solvers). That is why a more scalable criterion is required.

5.1 Evaluating the Degree of Redundancy

We propose to measure the degree of redundancy by taking into account how many clauses that are shared between solvers are redundant. We use a list to store from the beginning the number of received clauses (st_r) and a second to store the number of kept clauses (st_k). Kept clauses are those which have not been removed during the subsumption process. Each time a solver comes back to the MANAGER (every $1{,}000$ conflicts in our implementation), it shares its clauses. The number of received (resp. kept) clauses since the beginning is pushed to st_r (resp. st_k) by the MANAGER.

The *redundancy shared clauses measure*, in short $rscm$, is defined for a step t w.r.t. a sliding window of size m (20,000 in our experiments) as the ratio between the number of clauses received during the last $t - m$ updates of st_r and the number of clauses kept during the same time. More precisely, we have $\forall j < 0$, $st_r[j] = st_k[j] = 0$:

$$\begin{aligned} rscm_t &= \frac{st_r[t] - st_r[t-m]}{st_k[t] - st_k[t-m]}, \text{ if } st_k[t] - st_k[t-m] \neq 0 \\ rscm_t &= st_r[t] - st_r[t-m], \text{ otherwise} \end{aligned} \tag{1}$$

First, note that when several solvers work on the same part of the search space, there is a high likelihood that learnt clauses by the different solvers are redundant. This means that the number of subsumed clauses is important and therefore the $rscm$ value is high. Conversely, when solvers are sparsed in the search space, there is a high probability that shared clauses are not redundant and then the $rscm$ value tends to be low. Consequently, a small value of the $rscm$ indicates that the solver needs to intensify the search, whereas a high value signifies that the solvers have to diversify their search space.

5.2 Intensification/Diversification Mechanisms based on the $rscm$ measure

It is possible to control in several ways how solvers explore the search space (shared clauses, solvers' heuristics, ...). In AMPHAROS, we choose to solve the intensification/diversification dilemma by controlling two criteria: the way the tree is extended (see Sect. 3.3) and the number of clauses which are transferred from the `standby` to the `purgatory` buffers (see Sect. 4.1). Thus, for us, diversifying (resp. intensifying) the search consists in increasing (resp. decreasing) these two parameters. Before introducing them, let us a summarize:

Few subsumed Clauses ($rscm$ is low)	**Many subsumed clauses** ($rscm$ is high)
Reduce extension	Favour extension
Increase the number of imported clauses	Limit the number of imported clauses
Intensification	**Diversification**

Extension guiding by the $rscm$ First, let us remark that each path from the root to a leaf represents a unique set of literals that splits the search space in a deterministic way. Thus, the bigger the tree, the higher the probability to run two solvers in totally different sub-problems. To control the tree grows, we define the extension factor fe introduced in the section 3.3 in the following way:

$$fe_t = \frac{1{,}000}{{rscm_t}^3} \tag{2}$$

Let us recall that this extension factor is used to define the threshold of misses that a cube can encounter before an extension is accepted. Hence, the smaller (resp. bigger) the $rscm_t$ value is, the bigger (resp. smaller) the fe value is and then the slower (faster) the tree extension is. Note that the cubic factor allows to decrease fe rapidly while fe is bounded (by 1,000) since when fe is high solvers run in concurrently and the tree is never updated. To prevent the tree from growing too quickly, we also bound the minimum value that fe can take by 10.

Condition to move from the `standby` to the `purgatory` When a clause is received by a solver it is possible that it is subsumed by a clause already present. This becomes highly probable when almost all shared clauses are found to be subsumed during the clause subsumption process. Thus, it seems natural that the number of accepted clauses (ie. the number of clauses transferred from the `standby` to the `purgatory`) increases (resp. decreases) when the $rscm$ value decreases (resp. increases).

As already mentioned (Sect. 4.1), in AMPHAROS the clauses freshly received are not directly attached to the solver. Thus, it is important to choose a clause selection criterion independant from the activation of clauses. To control the amount of clauses moved from the `standby` to the `purgatory` we use the notion of psm introduced in [3] and already used in the portfolio based SAT solver PENELOPE [1]. Recall that the psm of a clause represents the number of literals which are assigned to true by the polarity vector. Thus, a clause can be scored even if it is not used by the solver.

Then, in order to increase/decrease the number of clauses attached (and then transferred) in the `purgatory`, a criterion based on both the psm and $rscm$ values is proposed. This criterion is motivated by the observation from [3] that the clauses with a small psm value have a great potential to enter in conflict or be used during the search. Thus, a clause will be authorized to move from the `standby` buffer to the `purgatory` buffer when its psm value is less or equals than $\lfloor \frac{psm_{max}}{rscm_t} \rfloor$, where psm_{max} corresponds to the psm maximum limit accepted (set to 6 in our experiments). Consequently, clauses with a psm value of zero will be systematically accepted whatever the value of $rscm$. Whereas, clauses with a high psm value will be accepted if and only if they are probably not subsumed.

6 Experiments

We now evaluate AMPHAROS on the 100 benchmarks from the SAT-RACE 2015, parallel track [37]. During the last competition 53 (resp. 33) instances have been proved satisfiable (resp. unsatisfiable) by at least one solver and 14 instances remained unsolved. All experimentations have been conducted on 2 Dell R910 with 4 Intel Xeon X7550. Each node has 32 cores, a gigabit ethernet controller and 256GB of RAM. Time limit was set to 1,200 seconds per test (wall clock time). Then, for experiments executed with 64 cores, we use two different computers. All log files and additional pictures are available in http://www.cril.univ-artois.fr/ampharos/.

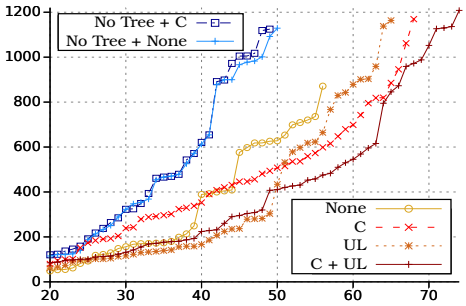
6.1 Communication management

Since in AMPHAROS a lot of messages have to be exchanged between the MANAGER and solvers, the management of the communications has to be very effective. Thus, we have opted for the open source Message Passing Interface implementation (Open MPI) to manage the communication on a low level.

The bottleneck imposed by the fact that the MANAGER has to all at once compute the subsumed clauses and communicate with the solvers, was a major problem. To avoid that solvers wait too long without work, a round robin architecture with non-blocking listening of solvers was put in place. Moreover, because the subsumption process can be time consuming, the clauses received to be checked are not treated at once (in our

Tree	Exchanges	SAT	UNS	Total
Yes	C + UL	**49**	**25**	**74**
Yes	C	47	21	68
Yes	UL	47	18	65
Yes	None	41	15	56
No	C	43	6	49
No	None	44	6	50

(a) Overview table



(b) Number of instances solved w.r.t. the time

Fig. 4: Comparing several versions of AMPHAROS on 64 cores. Tab. 4a gives the results of each version in term of solved instances. The columns represent, in that order, the fact that the tree decomposition is activated (**yes/no**), the kind of information exchanged (clauses (**C**) or/and unit literal (**UL**)), the number of SAT/UNSAT instances solved. Fig. 4b shows the number of solved instances (x-axis) w.r.t. the time (y-axis).

implementation packet of 1,000 clauses are considered). Thus, the MANAGER communicates with a solver, then checks a set of clauses, and so on, until the time limit is reached or the problem is proved satisfiable/unsatisfiable.

6.2 Setup

AMPHAROS is a modular framework that allows to add easily new types of solvers. For these experiments three sequential SAT solvers have been used: GLUCOSE [6], MINISAT [17] and MINISATPSM [3]. Only a couple small changes have been implemented in these solvers. In order to manage the interactions with the MANAGER, all solvers implement a C++ interface. This interface grouped communication routines and methods used to deal with `standby` and `purgatory` buffers. The core of solvers has also been modified in order to avoid resetting everything at each call to SAT solver (restart, learnt deletion policies, ...). Moreover, as for the version of GLUCOSE presented in [4], when a solver restarts it does not go to decision level 0 but to the level of the last assumption. The clauses moved from the `purgatory` to the `learnt` buffer are simply incorporated into the learnt clauses database as if they were learnt by solvers themselves.

6.3 Results

The experimental evaluation is divided into four parts. First, we evaluate the different ingredients of AMPHAROS. Then, we study the scalability of our solver. Finally, we compare AMPHAROS to the state-of-the-art and study the impact of the $rscm$ measure.

On the impact of each component The benefit of the three optional components (tree decomposition (**Tree**), clauses (**C**) and unit literals exchange (**UL**)) of AMPHAROS has been studied experimentally. To this end, several versions of AMPHAROS have been executed on 64 cores. These experiments, reported in Fig. 4, show gradual improvements

when each of these options was taken into account in a cumulative way. From the table (4a) and the cactus plot (4b) several observations can be made.

First, Fig. 4a shows that whatever the combination of options is used, AMPHAROS is more efficient when the tree decomposition is used (Tree sets to true in the first column). The versions working on the initial problem in a competitive way, which could be regarded as portfolio parallel SAT solvers (with (C) or without (none) clause sharing), solve systematically less instances than the others running on sub-problem obtained from cubes. This shows the importance of how AMPHAROS solves the intensification/diversification dilemma using a tree decomposition.
Second, results show the importance of exchanging information between solvers. AMPHAROS that does not exchange information systematically solved less problems than the others which share clauses or unit literals. When we separately compare the two exchange options (C and UL), we observe that sharing clauses allows (as expected) to improve the solver on unsatisfiable problems. However, as pointed out in Fig. 4b, activating this option makes the solver slower on easy problems (solved with less than 600 seconds). This can be explained by the fact that the communication engendered to share clauses and manage them is significant and slows down the solvers on 'easy' problems.

Finally, as highlighted by these experiments, there is a synergy between the exchange options. Even if clause sharing drastically reduces the solver performance on easy benchmarks, combining this component with the unit literals allows one to deliver the most significant improvement in terms of number of successfully solved instances. From now, AMPHAROS is reported as the version using all components.

Scalability evaluation To evaluate the scalability of AMPHAROS we run it on 8, 16, 32 and 64 cores. Fig. 5 gives the number of solved instances w.r.t. the time by the different versions of AMPHAROS. It clearly demonstrates that our approach is highly scalable. The version running on 64 cores solves 49 SAT and 25 UNSAT benchmarks, that is 15%, 45% and 70% more benchmarks than the one running on 32 (44 SAT and 20 UNSAT), 16 (36 SAT and 15 UNSAT) and 8 (33 SAT and 11 UNSAT) cores, respectively. In order to show the efficiency of our approach with more computers linked over the network, we also ran it with 64 cores using 8 computers with 8 cores each (see the curve 8×8 on Fig. 5.). This version solves 46 SAT and 24 UNSAT benchmarks. Since we restrict the number of messages, we obtain similar results. Differences can be explain by the indeterminism of our approach.

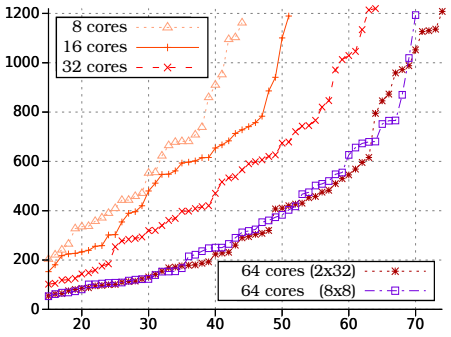


Fig. 5: Number of instances solved

AMPHAROS versus the state-of-the-art In order to evaluate AMPHAROS with respect to existing work, we choose to compare our approach with the three best solvers of the last SAT competition that ran in the parallel track. These solvers are (in their

rank order): SYRUP [7], TREENGELING and PLINGELING [12]. Because they do not run with MPI, and we have no processor with 64 cores, we execute them on 32 cores. We also compare our solver on 64 cores with the work stealing parallel SAT solver DOLIUS [2] and the portfolio solver HORDESAT [8]. Fig. 6 reports results obtained by these solvers.

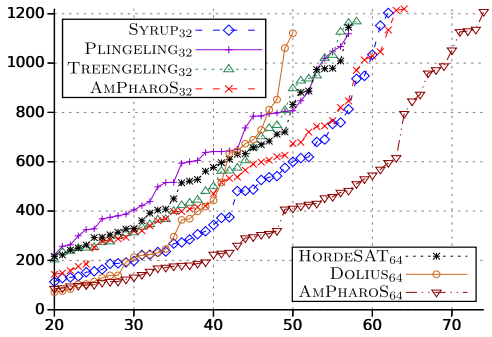
Let us first consider the experiments launched on 32 cores. As reported in Tab. 6a, AMPHAROS solves more instances than the other solvers. It is the best solver on the satisfiable benchmarks and solves the same number of unsatisfiable problems as TREENGELING (which is also a divide and conquer based method). Comparing to SYRUP and PLINGELING, we can see that our solver significantly outperforms both on satisfiable problems but it is less efficient on unsatisfiable ones. This can be partially explained by the fact that AMPHAROS essentially solves the unsatisfiable problems by totally refuting the tree (*i.e.* by closing all branches). Consequently, it seems that on 32 cores we do not have enough workers to achieve this goal within the time limit.

When considering the running time of the solvers, reported in Fig. 6b, we can observe that AMPHAROS is faster than TREENGELING and PLINGELING but it is slower than SYRUP. This can be explained by the fact that SYRUP solves several unsatisfiable problems in short time (the `6s` family for instance).

If we consider the experiments run on 64 cores, we can see that our approach is highly competitive. AMPHAROS is significantly better than DOLIUS and HORDESAT. Moreover, it is important to notice that during the competition SYRUP (the winner of the parallel track) only used 32 cores instead of the 64 cores available. Consequently, it is possible to conclude that AMPHAROS is more efficient than both TREENGELING and PLINGELING on 64 cores. More importantly, as reported in Fig. 6b, AMPHAROS is very effective since it solves more instances and faster.

Solver	**#thr.**	**SAT**	**UNS**	**Total**
AMPHAROS	32	**44**	20	**64**
SYRUP	32	36	**26**	62
TREENGELING	32	38	20	58
PLINGELING	32	31	**26**	57
AMPHAROS	64	**49**	**25**	**74**
HORDESAT	64	33	24	57
DOLIUS	64	33	17	50

(a) Overview table



(b) Number of solved instances w.r.t. the time

Fig. 6: Comparing AMPHAROS versus the state-of-the-art parallel SAT solver. Tab. 6a gives results of each solver in term of solved instances w.r.t. the number of threads (**#thr.**). Fig. 6b shows the number of solved instances (x-axis) w.r.t. the time (y-axis).

Let us stress that none of these solvers are deterministic. To be fair, we ran all solvers just once and report the obtained results (as it was done in SAT competition 2015).

Benchmarks Information		Time w.r.t. rscm					rscm statistics			
name	**sol.**	**1**	**3**	**5**	**10**	**D**	**min**	**max**	**avg**	**med**
hitag2-10-60-0-65	UNS	563	173	120	127	304	1.20	2.36	2.11	2.28
jgiraldezlevy.109	UNS	544	243	214	154	260	1.60	4.32	3.76	4.12
minandmaxor128	UNS	788	IN	IN	IN	972	1.09	1.37	1.25	1.23
jgiraldezlevy.33	SAT	776	339	386	169	288	1.63	4.58	3.32	3.82
56bits-12.dimacs	SAT	114	168	180	395	101	1.25	1.60	1.43	1.46
004-80-8	SAT	248	412	16	264	110	1.41	4.64	3.10	3.09

Table 1: This table presents the obtained results on a representative set of benchmarks. Each line corresponds to an instance, with its satisfiability, identified by the leftmost column. The next four columns give the WC time (reported in seconds) to solve the instance w.r.t. the value of $rscm$ (static (set to 1, 3, 5 and 10) or dynamic (D)). The rightmost reports stastictics on the value obtained by the dynamic computation of $rscm$.

Impact of the $rscm$ value To conclude this section, we evaluate the impact of the $rscm$ value on our solver's performance. To this end, we selected a representative set of benchmarks and ran four versions of AMPHAROS with different values of $rscm$ (1, 3, 5 and 10) and compared them with the dynamically chosen value. Let us recall that $rscm$ has an impact on both the tree extension and the amount of exchanged clauses. Moreover, recall that, as mentioned in Sect. 5.2, the extending factor fe is fixed to 10 when $rscm > \sqrt[3]{100}$. Thus, the difference between $rscm = 5$ and $rscm = 10$ is only the amount of shared clauses. Tab. 1 shows that these instances do not have the same comportment with respect to the $rscm$ value. Some problems need to extend more (`jgiraldezlevy`) and others need to extend less and exchange more (`minandmaxor128`). It is also important to note some benchmarks are unpredictable (`004-80-8`). As regards the dynamic adjustment, we observe that it is in average often close to the best value. The performance differences often come from the fact that the solver needs time to find the right $rscm$ value.

7 Conclusion

We proposed a new parallel SAT solver, designed to work on many cores, based on the divide and conquer paradigm. Our solver allows two kinds of clause sharing, the classical one and one more linked to the division of the initial formula. Furthermore, we proposed to measure the degree of redundancy of the search by counting the number of subsumed shared clauses. With this measure, we are able to adjust dynamically the search, resulting in a new way of controlling the dilemma of intensification/diversification of the search. Experiments show promising results.

Our main objective is to deploy a SAT solver among the cloud. Thus, this paper is a first step towards this goal and leads to many perspectives. We plan to run our solver on a cloud architecture using grid computing. For that, we plan to run several MANAGERs in parallel letting solvers go from a MANAGER to another one. For that, we need to choose more carefully variables used for the division. Many possibilities arise like the notion of blocked literals recently used for SAT solving [27,14]. Finally, we also need to improve performances on unsatisfiable instances by paying more attention on shared clauses.

References

1. Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 200–213, 2012.
2. Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette. An effective distributed d&c approach for the satisfiability problem. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 183–187, 2014.
3. Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On freezing and reactivating learnt clauses. In *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, pages 188–200, 2011.
4. Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pages 309–317, 2013.
5. Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Just-in-time compilation of knowledge bases. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.
6. Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.
7. Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 197–205, 2014.
8. Tomas Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pages 156–172, 2015.
9. Anton Belov, Inês Lynce, and João Marques-Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, 2012.
10. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.
11. Armin Biere. *Bounded Model Checking*, chapter 14, pages 455–481. Volume 185 of Biere et al. [10], February 2009.
12. Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. *Proceedings of SAT Competition 2013*, page 51, 2013.
13. Armin Biere. Yet another local search solver and lingeling and friends entering the sat competition 2014. In *proceedings of SAT competition.*, 2014.
14. Jingchao Chen. Minisat bcd and abcdsat: Solvers based on blocked clause decomposition. In *SAT RACE 2015 solvers description*, 2015.
15. Wahid Chrabakh and Richard Wolski. The gridsat portal: a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure. *Concurrency and Computation: Practice and Experience*, 19(6):795–808, 2007.
16. Geoffrey Chu, Peter J. Stuckey, and AAron Harwood. Pminisat: a parallelization of minisat 2.0. Technical report, SAT Race, 2008.
17. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

18. Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3):75–90, 2005.
19. Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.
20. Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 431–437, 1998.
21. Long Guo, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pages 252–265, 2010.
22. Long Guo and Jean-Marie Lagniez. Dynamic polarity adjustment in a parallel SAT solver. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pages 67–73, 2011.
23. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 499–504, 2009.
24. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.
25. Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 50–65, 2011.
26. Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, pages 372–386, 2010.
27. Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.
28. Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pages 276–292, 2013.
29. Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.
30. Joao Marques-Silva and Karem. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, 1996.
31. Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Improving search space splitting for parallel SAT solving. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 336–343, 2010.
32. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

33. Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 242–255, 2012.
34. Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, pages 294–299, 2007.
35. Jussi Rintanen. *Planning and SAT*, chapter 15, pages 483–504. Volume 185 of Biere et al. [10], February 2009.
36. Olivier Roussel. ppfolio. http://www.cril.univ-artois.fr/ roussel/ppfolio.
37. SAT-race, 2015. http://baldur.iti.kit.edu/sat-race-2015/.
38. Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2009.
39. Alexander Semenov and Oleg Zaikin. Using monte carlo method for searching partitionings of hard variants of boolean satisfiability problem. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 222–230, 2015.
40. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference*, pages 244–257, 2009.
41. Peter van der Tak, Marijn Heule, and Armin Biere. Concurrent cube-and-conquer. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, pages 475–476, 2012.
42. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

Chapitre 3

Factoring Out Assumptionsto Speed Up MUS Extraction

Article paru dans les actes de « *the 16th International Conference on Theory and Applications of Satisfiability Testing* » (SAT'13), pages 276 – 292, juillet 2013.

Co-écrit avec Armin Biere.

Factoring Out Assumptions to Speed Up MUS Extraction*

Jean-Marie Lagniez and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

Abstract. In earlier work on a limited form of extended resolution for CDCL based SAT solving, new literals were introduced to factor out parts of learned clauses. The main goal was to shorten clauses, reduce proof size and memory usage and thus speed up propagation and conflict analysis. Even though some reduction was achieved, the effectiveness of this technique was rather modest for generic SAT solving. In this paper we show that factoring out literals is particularly useful for incremental SAT solving, based on assumptions. This is the most common approach for incremental SAT solving and was pioneered by the authors of MINISAT. Our first contribution is to focus on factoring out only assumptions, and actually all eagerly. This enables the use of compact dedicated data structures, and naturally suggests a new form of clause minimization, our second contribution. As last main contribution, we propose to use these data structures to maintain a partial proof trace for learned clauses with assumptions, which gives us a cheap way to flush useless learned clauses. In order to evaluate the effectiveness of our techniques we implemented them within the version of MINISAT used in the publically available state-of-the-art MUS extractor MUSer. An extensive experimental evaluation shows that factoring out assumptions in combination with our novel clause minimization procedure and eager clause removal is particularly effective in reducing average clause size, improves running time and in general the state-of-the-art in MUS extraction.

1 Introduction

The currently most widespread approach for *incremental* SAT was pioneered by the authors of MINISAT [1] in context of bounded model checking [2] and finite model finding [3], and has seen many other important practical applications since then. It can easily be implemented on top of a standard SAT solver based on the *conflict driven clause learning* (CDCL) paradigm [4], as described in [1], by modifying the heuristics for picking decisions, to branch on literals assumed to be true first. In this paper we refer with *assumptions* to this set of literals assumed to be true.

Another important application, which makes use of incremental SAT, is the extraction of a *minimal unsatisfiable set* (MUS) of clauses from a propositional

* Supported by FWF, NFN Grant S11408-N23 (RiSE).

formula in *conjunctive normal form* (CNF). The current state-of-the-art in MUS extraction [5] is based on incremental SAT. In the context of MUS extraction [6,7,8,9,10], the focus of this paper, and in similar or related applications of incremental SAT [3,11,12,13,14,15,16], an additional analysis is required, which learns sub-sets of assumptions, under which the formula is proven to be unsatisfiable. In these applications, the number of assumptions is usually not only quite large, e.g. similar in size to the number of original variables and clauses in the CNF, but also the SAT solver is called many times, while the set of assumptions almost stays the same.

As it turns out, current SAT solvers have not been optimized for this actually rather common usage scenario. We propose a new technique for compressing incremental proofs for problems with many assumptions. Our technique is based on the idea of factoring out literals of learned clauses by extended resolution steps, which also forms the basis of related work on speeding up SAT solving in general [17,18]. Clauses *learned* in those applications we are interested in typically contain many literals which are the negation of original assumptions. We call these negations of originally assumed literals also assumptions or more precisely *assumption literals*, if the context requires to distinguish between originally assumed literals ("assumptions") used as decisions and their negations occurring in learned clauses ("assumption literals").

In our approach we factor out these assumption literals in order to shrink learned clauses and reduce the number of literals traversed, particularly during boolean constraint propagation (BCP). This idea, if implemented correctly, does not change the search at all, but it is still quite effective in reducing the time needed for MUS extraction. Further, factoring out assumptions enables the use of compact dedicated data structures, and naturally suggests a new form of clause minimization, which gives another substantial improvement. Recording factored out assumptions explicitly, also gives us simple way to maintain a partial proof trace for learned clauses with assumptions. The trace can be used to compute an approximation of a "clausal core". We can then discard learned clauses out-side this clausal core eagerly, which empirically seems to be a useful strategy.

The authors of [19] observed a similar deficiency when using the assumption based approach for incremental SAT solving in the context of bounded model checking. They propose to use an additional SAT solver, to which assumptions are added as unit clauses. This in turn allows to improve efficiency of preprocessing and inprocessing under assumptions, but prohibits to reuse in the main solver clauses learned by the additional solver. However, according to [19] it is possible, by selectively adding assumptions, to extract "pervasive clauses" from the resolution proofs of clauses learned in the additional solver, with the objective that adding these "pervasive clauses" to the main solver is sound.

While in [19] as in our approach some sort of resolution proof has to be maintained, the main solver in [19] still uses the classical assumption based approach and thus will benefit from our proposed techniques. Finally, the motivations as well as the application characteristics considered in the experimental part differ.

2 Factoring Out Assumptions

In incremental SAT with *many* assumptions, learned clauses contain many assumption literals too (see previous section for the definition of this terminology). Accordingly the average size of learned clauses can become very large (as we will see in Fig. 6). This effect increases the size of the working set (used memory), or more specifically, the average number of traversed literals per visited clause during BCP. The same argument applies to visited clauses during conflict analysis. As a consequence, SAT solver performance degrades.

For every learned clause we propose to replace the "assumption part" by a new fresh literal, called *abbreviation* literal. The replaced part consists of all assumptions and previously added abbreviations. The connection between the abbreviation and the replaced literals is stored in a *definition map* as follows.

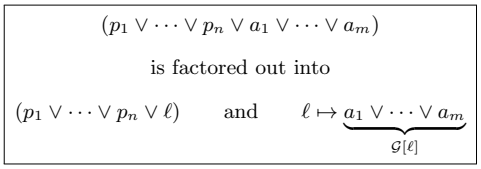


Fig. 1. Factoring out assumptions by introducing a new abbreviation literal ℓ.

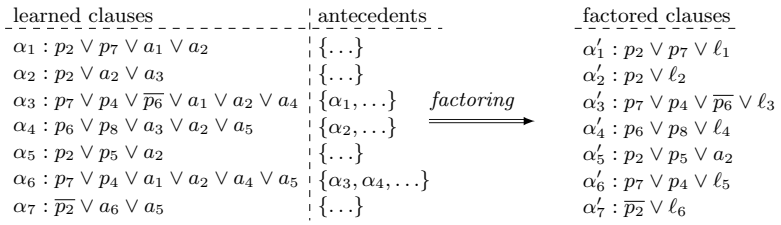
Let $p_1 \vee \cdots \vee p_n \vee a_1 \vee \cdots \vee a_m$ be a new learned clause, where $p_1, \ldots, p_n$ are original literals and $a_1, \ldots, a_m$ are either assumptions or abbreviations. We pick a fresh abbreviation literal ℓ and instead of the originally learned clause add the clause $p_1 \vee \cdots \vee p_n \vee \ell$ to the clause data base. Then we record $a_1 \vee \cdots \vee a_m$ as the *definition* $\mathcal{G}[\ell]$ of ℓ in the definition map $\mathcal{G}$ (see Fig. 1). For $m \leq 1$ this replacement does not make sense and the original learned clause is kept instead.

Consider the example in Fig. 2 for an (incremental) SAT run under the assumptions $\overline{a_1}, \ldots, \overline{a_6}$. Conflict analysis might learn clauses α_1, ..., α_7 depicted on the left of Fig. 2(a), where $p_1, \ldots, p_7$ are original literals and $a_1, \ldots, a_6$ assumption literals.[1] Note that the run is not supposed to be complete. Only some clauses are shown together with their antecedent clauses, and original clauses are ignored too (to simplify the example). For instance α_3 is derived through resolution from α_1 and from some other original clauses not shown (the "...").

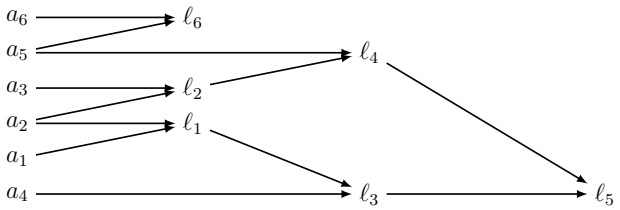
The result of introducing abbreviations to factor out assumptions is shown on the right. The first clause α_1 is factored into α'_1 and the definition $a_1 \vee a_2$ of the new abbreviation literal ℓ_1. The definition is recorded in the definition map, as shown in Fig. 2(b), where ℓ_1 has two incoming arcs, one from a_1 and one from a_2. Further let us point out, that $\alpha'_5 = \alpha_5$, because it *keeps* a_2 as single non-original literal, which (as discussed above) reduces the overall number of introduced abbreviations. Finally, note how definitions might recursively depend on other definitions as for ℓ_3, ℓ_4 or ℓ_5, while factoring α_3, α_4, and α_6 respectively.

As briefly discussed above, assumptions are always assigned first and thus assigning them can actually be seen as a preprocessing resp. initialization step

[1] Assumption literals are literals made of a variable which is currently used or was used in an assumption. See again the introduction section for a precise definition.



(a) Learned clauses (original version left, factored version right)



(b) Definition Map

Fig. 2. Factoring out assumptions

before the actual solving starts. Furthermore, the algorithm for MUS extraction, as implemented in MUSer [9], to which we applied our technique, has the following property: *the set of variables used in assumptions stays the same over all incremental calls*, with the exception of variables assigned at the top-level. The techniques presented in this paper are sound, even if this property does not hold, i.e. the set of assumptions changes (substantially) from one incremental call to the next. However, if the property does not hold they are probably less effective. We focus on the important problem of MUS extraction here and leave it to future work to apply our techniques to other scenarios of incremental SAT.

Assigning in every incremental call the current set of assumptions during an initialization phase, will imply a unique value for all the (previously introduced) abbreviation literals, unless the set of assumptions turns out to be inconsistent, in which case the solver returns immediately. For that reason we do not have to encode definitions as part of the CNF. Abbreviations are assigned during an initialization phase, as described in the next Section (see also Alg. 2).

2.1 Initialization

After factoring out assumptions and adding abbreviations instead, every learned clause α contains *at most one* assumption or abbreviation. In this case we denote by $r(\alpha)$ this *replacement* literal. For other clauses we assume $r(\alpha)$ to be undefined. The graph represented by the definition map $\mathcal{G}$ can be interpreted as a (non-cyclic) circuit, which computes consistent values for abbreviations after all the assumption variables have been assigned. Special care has to be taken to handle assumptions and abbreviations, which are fixed by the user in between incremental calls. For instance, in MUS extraction, they are used to permanently select transition clauses [9] to be part of the extracted MUS.

Algorithm 1: assignAbbreviation

Input: ℓ: literal; **var** $\mathcal{I}$: interpretation; $\mathcal{G}$: definition map

```
removeUnit(G[ℓ]);
while I(G[ℓ]) unassigned do
    pick unassigned ℓ' ∈ G[ℓ];
    assignAbbreviation(G, ℓ', I);
if I(G[ℓ]) = ⊥ then I ← I ∪ {¬ℓ} else I ← I ∪ {ℓ};
```

In order to assign an abbreviation, we need to assign assumption variables and, recursively, every abbreviation in its definition. This is formulated in Alg. 1, which has the following arguments: the literal ℓ to be assigned, and (by reference) the current interpretation $\mathcal{I}$ and the definition map $\mathcal{G}$. First, literals assigned at the top-level (units), are removed from $\mathcal{G}[\ell]$. Next, while there is an unassigned literal ℓ' in $\mathcal{G}[\ell]$ and $\mathcal{G}[\ell]$ is itself unassigned by the current interpretation $\mathcal{I}$, we assign ℓ', using the same algorithm recursively. As soon as the value of $\mathcal{G}[\ell]$ under $\mathcal{I}$ is determined, we can also assign ℓ to $\mathcal{I}(\mathcal{G}[\ell])$.

By construction the definitions in the definition map $\mathcal{G}$ are non-cyclic. Further, we assume that every assumption is assigned by $\mathcal{I}$, as discussed in the previous section. Then this algorithm terminates and consistently assigns the value of each abbreviation ℓ to the value of its definition $\mathcal{G}[\ell]$.

2.2 Assigning the Set of Necessary Abbreviations

In the worst case, every learned clause resp. conflict requires a new abbreviation to be added. Therefore, in principle, the definition map grows linearly in the number of conflicts. This not only requires a huge amount of memory, but also needs substantial running time to initialize all the abbreviations of the definition map during incremental SAT calls.

However, since inactive [20] resp. less useful learned [21,22] clauses are frequently collected during the main CDCL loop of the SAT solver anyhow, many abbreviations turn out not to be referenced anymore after a certain point. They become *garbage abbreviations* and could be collected too. Actually, only the assignments of those abbreviations have to be initialized, which are still referenced in learned clauses (recursively). Assigning additional abbreviations is not harmful, but useless.

Algorithm 2 implements an initialization of abbreviations taking this argument into account. It returns an interpretation $\mathcal{I}$, which assigns all abbreviations recursively reachable from the clauses in the CNF Σ (which includes learned clauses). First, the algorithm initializes $\mathcal{I}$ by assigning all assumptions. Next, it traverses all clauses α to which a replacement $r(\alpha)$ has been added and then calls Alg. 1 to assign the replacement literal. The resulting $\mathcal{I}$ consistently assigns reachable abbreviations to the value of their definition in the definition map $\mathcal{G}$, unless a clause is found that has all its literals assigned to false.

Algorithm 2: initialization

Input: Σ: CNF formula; $\mathcal{A}$: assumptions; $\mathcal{G}$: a definition map
Result: $\mathcal{I}$ a partial interpretation

```
1 I ← A ∪ {top-level units};
2 foreach α ∈ Σ with r(α) defined do
3     if r(α) is unassigned by I then
4         assignAbbreviation(G, r(α), I);
5     if I(α) = ⊥ then break;
6 return I;
```

2.3 Assumption Core Analysis

As discussed in the introduction, applications of incremental SAT with assumptions often make use of the SAT solver's ability to return an *assumption core*, i.e., a subset of the given assumptions, which in combination with the given CNF can not to be satisfied. Intuitively, the assumption core exactly contains the assumptions "used" by the SAT solver to derive the inconsistency. In contrast to the concept of MUS, these assumption cores are typically not required to be minimal. As implemented in MINISAT [1] such an assumption core can be computed by a separate conflict analysis routine called "analyzeFinal", which recursively goes through the implication graph to only collect assumptions in contrast to the usual analysis routine of CDCL solvers which cuts off the search for a learned clause as soon as possible, e.g., following the 1st UIP scheme [23].

After factoring out assumptions and adding abbreviations the "analyzeFinal" procedure has to be adapted to care for abbreviations, which is described in Alg. 3. The algorithm takes as input a CNF formula Σ, the current unsatisfiable trail[2] $\mathcal{I}$, a clause α falsified under $\mathcal{I}$, the definition map $\mathcal{G}$, and returns the set of assumptions $\mathcal{C}$ "used" to establish the unsatisfiability proof. It starts by initializing $\mathcal{C}$ and the literals $\mathcal{V}$ already visited with the empty set. Next, the stack $\mathcal{T}$, containing the set of literals that must be further visited, is initialized with the conflict clause α. Then, while there is still an unvisited literal $\ell \in \mathcal{T}$, it is marked. Depending on its type three different cases have to be distinguished. First in line 5, if ℓ is an assumption, then ℓ is added to the conflict clause $\mathcal{C}$. Second in line 6, if ℓ is an abbreviation its definition $\mathcal{G}[\ell]$ is added to $\mathcal{T}$. This is actually the only part where the algorithm has to be adapted to recursively explore the definition map. Third in line 7, ℓ is neither an assumption nor an abbreviation and the reason of its propagation is added to $\mathcal{T}$ (implication graph exploration). Decision literals are assumed to have an empty set of antecedents.

[2] Every literal assigned to true, particularly those found during BCP, are added to a stack, called *trail*, to record the order of assignments. The reason, also called antecedent, of a forced assignment is saved too. Please refer to [1] for more details.

Algorithm 3: analyzeFinal

Input: Σ: CNF; $\mathcal{I}$: trail; α: clause; $\mathcal{G}$: a definition map
Result: $\mathcal{C}$, a subset of the assumptions

$\mathcal{C} = \emptyset$; $\mathcal{V} = \emptyset$;
$\mathcal{T} \leftarrow \alpha$;
while $\exists \ell \in \mathcal{T} \setminus \mathcal{V}$ **do**
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{\ell\}$;
  if ℓ *is an assumption* **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{\ell\}$;
  else if ℓ *is an abbreviation* **then** $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{G}[\ell]$;
  else $\mathcal{T} \leftarrow \mathcal{T} \cup reason(\ell, \mathcal{I})$;
return $\mathcal{C}$;

Example 1. Consider again the example in Fig. 2. Given $\{\overline{a_1}, \overline{a_2}, \overline{a_3}, \overline{a_4}, \overline{a_5}, \overline{a_6}\}$, learning α'_7 allows to conclude that the formula is unsatisfiable. Alg. 3 produces:

$\mathcal{T}$	$\mathcal{V}$	$\mathcal{C}$	ℓ
$\overline{p_2}, \ell_6$	$\emptyset$	$\emptyset$	$undef$
ℓ_6, ℓ_2	$\emptyset$	$\emptyset$	$\overline{p_2}$
ℓ_2, a_5, a_6	ℓ_5	$\emptyset$	ℓ_6
a_5, a_6, a_2, a_3	ℓ_5, ℓ_6	$\emptyset$	ℓ_2
a_6, a_2, a_3	ℓ_5, ℓ_6, a_5	a_5	a_5
a_2, a_3	ℓ_5, ℓ_6, a_5, a_6	a_5, a_6	a_6
a_3	$\ell_5, \ell_6, a_5, a_6, a_2$	a_5, a_6, a_2	a_2
$\emptyset$	$\ell_5, \ell_6, a_5, a_6, a_2, a_3$	a_5, a_6, a_2, a_3	a_3

The resulting learned clause is $(a_5 \vee a_6 \vee a_2 \vee a_3)$. Note, neither a_1 nor a_4 were actually "used" in deriving it. In the next section will make use of such an analysis to eagerly reduce the learned clause data base.

2.4 Reduce Learned Clause Database

Keeping all learned clauses slows down the SAT solver considerably. Thus heuristics to determine which learned clauses to keep resp. how and when to reduce the learned clause database are an essential part of state-of-the-art SAT solvers [20,21,22]. After an incremental SAT call returned "unsatisfiable", we propose to only keep those learned clauses, which were used to show that the assumed assumptions in this SAT call are inconsistent and discard all others.

Experiments in Sect. 3.2 will give empirical evidence for the effectiveness of these heuristics. Even though it is not a solid argument, an intuitive explanation could be that learned clauses are removed quite frequently anyhow. Further, most likely exactly those learned clauses related to the last set of assumptions are useful in the next SAT call too. This particularly applies to MUS extraction where the assumptions do not change much.

However, in order to apply these heuristics we need to be able to determine whether a certain clause was used in deriving the inconsistency. As it turns out, our definition map can be interpreted as partial proof trace for learned clauses (with assumptions) and thus gives us a cheap way to flush learned clauses and definitions not required to show that the given set of assumptions is inconsistent.

Algorithm 4: eagerLearnedClauseDatabaseReduction

```
Input: var Δ: set of learned clauses; var G: a definition map; V: literals;
foreach α ∈ Δ do
    if r(α) is an abbreviation and r(α) ∉ V then
        Δ ← Δ \ α;
        remove r(α) and its definition from G;
```

Focusing on the remaining relevant learned clauses and definitions in this "core" reduces run time, as our experiments in Sect. 3.2 will show.

Let us continue with Example 1 after learning α'_7. Only α'_2 and α'_7 are required to show unsatisfiability under the given set of assumptions, while α'_4 is not required and thus according to our heuristic should be removed. This eager reduction of the learned clause database can be easily implemented as a post-processing phase using $\mathcal{V}$ computed by analyzeFinal, which is shown in Alg. 4.

2.5 Assumption Aware Clause Minimization

New learned clauses can often be minimized by applying additional resolution steps with antecedent clauses in the implication graph. Two approaches are currently used to achieve this minimization: applying self-subsuming resolution, also called local minimization, or applying recursive minimization[24]. In recursive minimization several resolution steps are tried to determine whether a literal can be removed from the learned clause. In both cases resolutions are only applied if the resulting clause is a strict sub-clause. Sörensson and Biere [24] demonstrated that clause minimization usually improves SAT solver performance. In the following we will either apply this classical recursive minimization, no minimization at all, or a new form of recursive minimization, and thus do not consider local minimization further.

In the incremental setting with many assumptions, our preliminary experiments showed that classical clause minimization is not very effective. Usually the number of literals deleted in classical clause minimizations is rather small. As reason we identified the fact that assumptions are not obtained by unit propagation, and thus cannot be removed from learned clauses through additional resolution steps. Furthermore, non-assumption literals are often blocked by at least one assumption pulled in by resolution steps. The classical minimization algorithm requires that the resulting clause is a strict sub-clause. It is not allowed to contain more assumptions.

This situation is not optimal since assumptions, during one call of the incremental SAT algorithm, are assigned to false and can thus be considered to be irrelevant, at least for this call. Our new minimization procedure makes use of this observation and simply ignores additionally pulled in assumptions during minimization. The resulting "minimized" clause might even increase in size. However, it will never have more non-assumption literals than the original clause.

3 Experiments

The algorithms described above have been implemented within the SAT solver MINISAT [1], starting from the original version, used in the current version of the state-of-the-art MUS extractor MUSer [9]. It heavily makes use of incremental SAT solving with many assumptions following the selector variable-based approach [25]. Our modified version of [1] is called MINISAT$_{abb}$ (MINISAT with abbreviation). We focus on MUS extraction and compare the performance of MUSer for different versions of MINISAT.

For our experiments we used all 295 benchmarks from the MUS track of the SAT Competition 2011 [3] after removing 5 duplicates from the original 300 benchmarks. These benchmarks[4] have their origin in various industrial applications of SAT, including hardware bounded model checking, hardware and software verification, FPGA routing, equivalence checking, abstraction refinement, design debugging, functional decomposition, and bioinformatics. The experiments were performed on machines with Intel® Core^{TM}2 Quad Processor Q9550 with 2.83 GHz CPU frequency with 8 GB memory and running Ubuntu 12.04. Resource limits are the same as in the competition: time limit of 1800 seconds, memory limit of 7680 MB.

In the first experiment we apply our new approach of factoring out assumptions *without* changing clause learning. We then evaluate the impact of our new learned clause reduction scheme and our new clause minimization procedure. The experimental part concludes with more details on memory consumption.

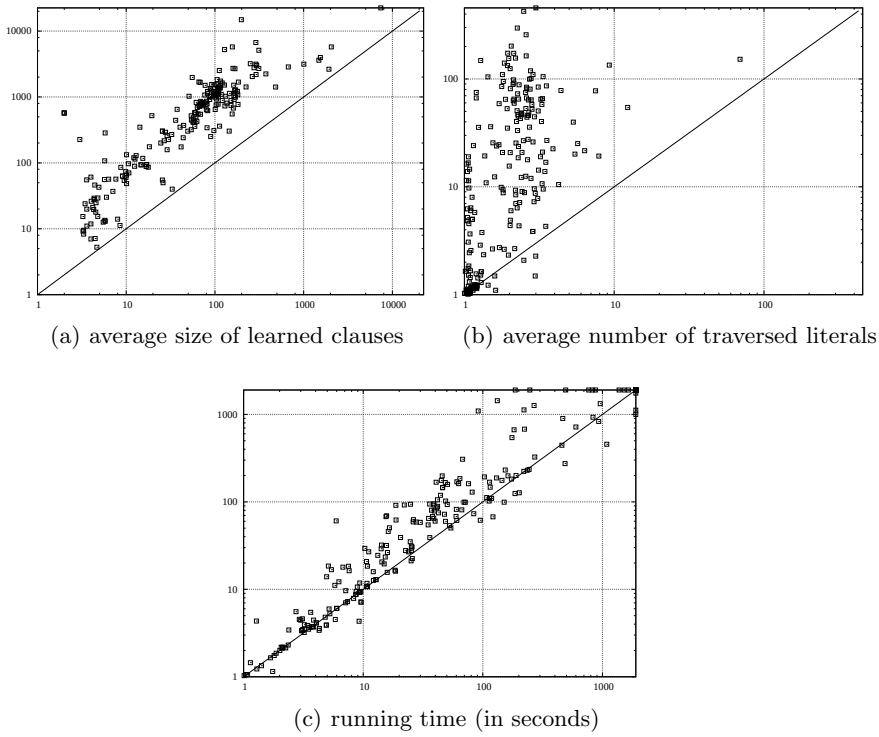
3.1 Factoring Out Assumptions

Fig. 3 shows a comparison between MUSer with our new approach based on factoring out assumptions, called MINISAT$_{abb}$, and the original version of MINISAT. First, in Fig. 3(a) the average size of learned clauses is compared. For many problems, adding clause abbreviations reduces the average size of learned clauses by an order of magnitude.

The main effect of our new technique is to reduce the size of learned clauses. This should also decrease the number of literals traversed while visiting learned clauses during BCP. In the scatter plot in Fig. 3(b) we focus on this metric and compare the average number of traversed literals while running both versions on the same instance. This includes the literals traversed in clauses visited during BCP, also including original clauses, but of course ignores clauses that are skipped due to satisfied blocking literals [26]. As the plot shows, the reduction in terms of the number of traversed literals is even more than the reduction of the average size of learned clauses. Consequently also the running time reduces considerably, see Fig. 3(c), but of course not in the same scale as in the previous plots. Note that in essence the "same clauses" are learned and thus the number of conflicts and learned clauses does not change.

[3] `http://www.satcompetition.org/2011`

[4] The set of benchmarks is available at `http://www.cril.univ-artois.fr/SAT11/`.



(a) average size of learned clauses (b) average number of traversed literals

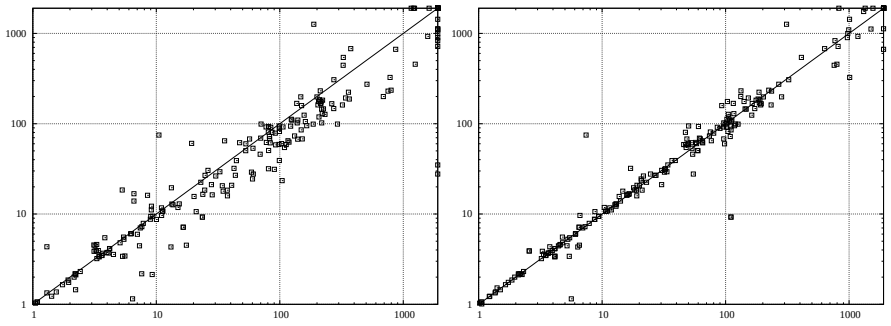
(c) running time (in seconds)

Fig. 3. Comparing MUSer on the 2011 competition instances from the MUS track, using the original MINISAT without abbreviations (y axis) vs. using our new version MINISAT$_{abb}$ with abbreviations (x axis) w.r.t. three different criteria.

The net effect of using abbreviations to factor out assumptions is that MUSer based on MINISAT$_{abb}$ solves 272 out of the 295 instances, and runs out of memory on 3 instances, whereas the version with the original MINISAT solves only 261 instances and runs out of memory in 13 cases. Our approach solves more instances, but not, at least primarily, because it runs out of memory less often.

As it turns out in the context of MUS extraction, definition clauses actually do not have to be watched. Further, abbreviation literals never have to be considered as decision and thus also do not have to be added to the priority queue (implemented as heap in MINISAT) for picking decisions. Thus we need initialization, by assigning all assumptions and abbreviations, the latter in incremental calls only, at the first decision level.

In order to make sure that the improvement observed in the previous experiment is independent from using our new optimized initialization phase, we report in Fig. 4(a) the run times of MUSer using the original version of MINISAT compared to the run times using a modified version of MINISAT, in which the assumption variables are assigned up-front and removed from the priority queue

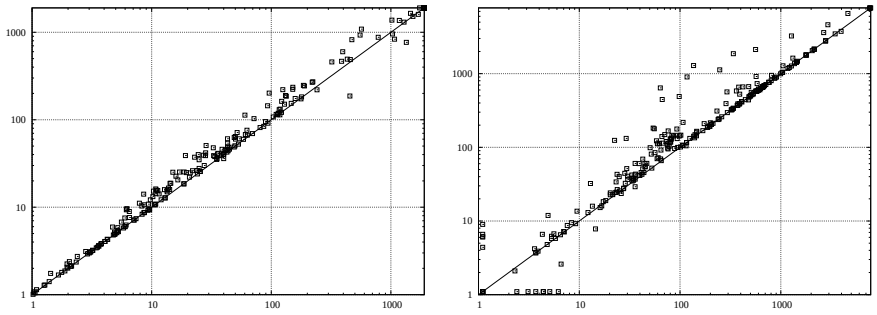


(a) MINISAT using initialization (b) MINISAT assumption-level-restarting

Fig. 4. On the left we show the running time of MUSer using MINISAT+init, a version of MINISAT, which initializes assumptions explicitly (x axis) vs. the original MINISAT version, which does not initialize them explicitly before search (y axis), both *without* abbreviations. Visiting each learned clause during initialization is time consuming without abbreviations. In the experiment shown on the right we only modified the restart mechanism to backtrack to the decision level of the last assigned assumption instead of backtracking to the root level. The modified version MINISAT+assumption-level-restarting (x axis) performs equally well as the original version of MINISAT (y axis). Running time is measured in seconds with a time limit of 1800 seconds as always.

initially too, called MINISAT+init. The results show that using this modified initialization scheme in the original version of MINISAT actually has a negative effect on the performance of MUSer (MUSer using MINISAT solves 261 instances whereas MUSer using MINISAT+init solves 257 instances) and thus can not be considered to be the main reason for the witnessed improvements in the first experiment. Our explanation for this effect is, that our initialization algorithm in essence needs only one pass over the learned clauses, even just a subset of all learned clauses, while initializing up-front BCP in MINISAT+init needs to visit lots of clauses during initialization. Note, again, that initialization has to be performed at the start of every incremental SAT call and might contribute a substantial part to the overall running time.

Modern SAT solvers based on the CDCL paradigm restart often by frequently backtracking to the root-level (also called top-level) [27,28,29,30] using a specific restart schedule [31,32,33,34]. With assumptions it seems however to be more natural to backtrack to the highest decision level, where the last assumption was assigned, which we call *assumption-level*. This technique is implemented in Lingeling [35], since it can naturally be combined with the technique of reusing the trail [36], but is not part of MINISAT. It might be conceivable, that forcing MINISAT to backtrack to the assumption-level during restarts can give the same improvement as initialization up-front. However, the experiment reported in Fig. 4(b) shows, that at least in MUS extraction, this "optimization" is useless.



(a) running time (in seconds) (b) memory used (in Mega Bytes)

Fig. 5. Comparison MUSer using MINISAT$_{abb}$ (y axis) vs. MINISAT$_{abb}$+g (x axis).

3.2 Learned Clauses Database Reduction

In this section, we study the impact on the performance of MINISAT$_{abb}$ w.r.t our new reduction algorithm for the learned clause database presented in Sect. 2.4. Fig. 5 compares MUSer using MINISAT$_{abb}$ with and without this more "eager garbage collector", which we denote by MINISAT$_{abb}$+g resp. MINISAT$_{abb}$. According to Fig. 5(b) eager garbage collection reduces memory consumption. Moreover, as shown in Fig. 5(a), this memory reduction does not hurt performance, since three more instances are solved (275 vs. 272) and only 1 instance (instead of 3) runs out of memory (see also Tab. 1).

3.3 Minimization of the Learned Clauses

In this section, we compare our new clause minimization procedure to existing variants of clause minimization. We consider three versions of MINISAT and MINISAT$_{abb}$ as back-end in MUSer [9]:

- without clause minimization (called *without*);
- the classical recursive clause minimization (*classic*) [24];
- our new clause minimization procedure (*full*) described in Sect. 2.5.

From the cactus plot in Fig. 6, which compares average size of learned clauses, we can draw the following conclusions. First, classical minimization is not effective in terms of reducing the average size of learned clauses, neither for MINISAT nor for MINISAT$_{abb}$, because it cannot remove assumptions during clause minimization. Classical minimization is slightly more effective with abbreviations than without. However, abbreviations might block self-subsumption during recursive resolution steps and thus prevent further minimization.

Next, we study the impact of our new full clause minimization described in Sect. 2.5 with and without using abbreviations. As reported in [24] for SAT solving *without* assumptions, recursive clause minimization typically is able to

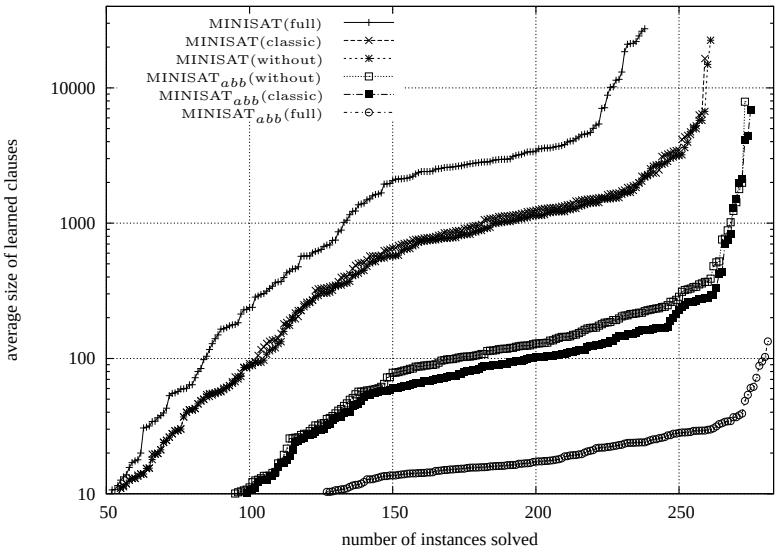


Fig. 6. Cactus plot reporting the average size of learned clauses produced by MUSer using MINISAT without abbreviations and MINISAT$_{abb}$ with abbreviations, and different clause minimization approaches. Factoring out assumptions (lower three curves) is always better than the original scheme (upper three curves). Further, full minimization gives a large improvement but only if assumptions are factored out. Without abbreviations full minimization actually turns out to be detrimental. Classic minimization only gives a small advantage over not using any minimization.

reduce the average size of learned clauses by one third. In the SAT solving *with* assumptions, as previously noted, assumptions prevent this reduction. With full clause minimization, however, we get back to the same reduction ratio of around 30% considering only literals that are neither assumptions nor abbreviations. Nevertheless, since deleting one literal is often necessary to apply additional resolutions, many new assumptions are added to the minimized clause. Using full minimization in MINISAT without abbreviations increases the average size of learned clauses by an order of magnitude, whereas MINISAT$_{abb}$ does not have this problem, since assumptions and abbreviations are factored out.

Actually, our new full clause minimization procedure in combination with MINISAT$_{abb}$ is able to reduce the average size of learned clauses by two orders of magnitude w.r.t the best version of MINISAT without abbreviations, while already one order of magnitude is obtained by MINISAT$_{abb}$ just by using abbreviations alone (with or without using classical clause minimization procedure).

In another experiment we measured the effect of our new garbage collection procedure Alg. 4. As it turns out, the average size is not influenced by adding this procedure, but as Tab. 1 shows, it has a positive impact on the number solved instances independent from the minimization algorithm used. Finally, this

	MINISAT	MINISAT$_{abb}$	MINISAT$_{abb}$+g
	#solved(MO)	#solved(MO)	#solved(MO)
without minimization	259(15)	272(3)	273(3)
classic minimization	261(13)	272(3)	275(1)
full minimization	238(25)	276(0)	**281(0)**

Table 1. The table shows the number of solved instances by MUSer within a time limit of 1800 seconds and a memory limit of 7680 MB, for different back-end SAT solver: the original MINISAT, then MINISAT$_{abb}$ with abbreviations, and finally MINISAT$_{abb}$+g with abbreviations and eager learned clause garbage collection. For each version of these three SAT solvers we further use three variants of learned clause minimization. The approach with abbreviations, eager garbage collection and full learned clause minimization, e.g. using all of our suggested techniques, works best and improves the state-of-the-art in MUS extraction from 261 solved instances to **281**.

tables also shows that the reduction of the average size of learned clauses directly translates into an increase of the number of solved instances. The combination of our new techniques improves the state-of-the-art of MUS extraction considerably.

3.4 Memory Usage

We conclude the experiments with a more detailed analysis of memory usage for the various considered versions of MUSer. As expected, Fig. 7 shows that shorter clauses need less memory. However, the effect in using our new techniques on overall memory usage is less pronounced than their effect w.r.t. to reducing average learned clause length. The main reason is that definitions have to be stored too. However, MINISAT with full clause minimization but without abbreviations produces a huge increase in memory consumption by an order of magnitude. This shows that factoring out assumptions is the key to make full clause minimization actually work. Also note, that our current implementation for storing definitions is not optimized for memory usage yet, and we believe that it is possible to further reduce memory consumption considerably.

4 Conclusion

In this paper we introduced the idea of factoring out assumptions, in the context of incremental SAT solving under assumptions. We developed techniques that work particularly well for large numbers of assumptions and many incremental SAT calls, as it is common, for instance, in MUS extraction. We implemented these techniques in the SAT solver MINISAT$_{abb}$ and showed that they lead to a substantial reduction in solving time if used in the SAT solver back-end of the state-of-the-art MUS extractor MUSer [9].

More specifically, experimental results show that factoring out assumptions by introducing abbreviations is particularly effective in reducing the average learned clause length, which in turn improves BCP speed. Even though memory

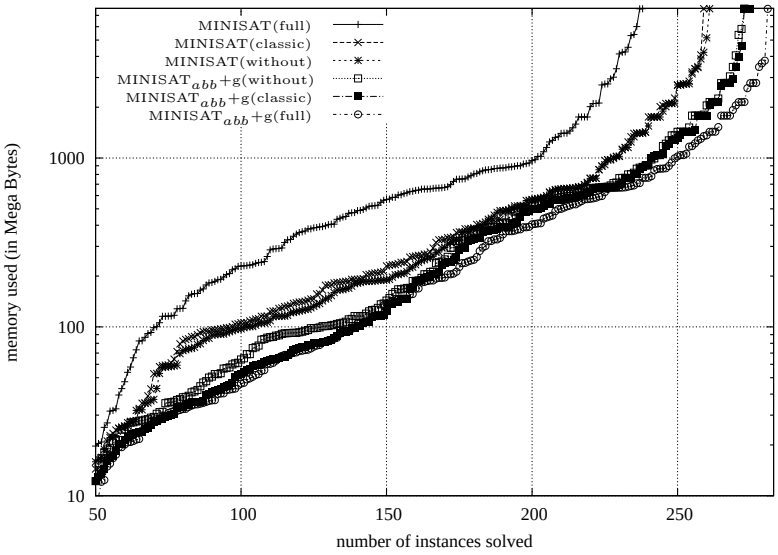


Fig. 7. Memory usage of MUSer based on the original MINISAT without abbreviations and MINISAT$_{abb}$+g with both abbreviations and eager garbage collection. Both versions of the MINISAT are combined with three different clause minimization strategies. Note, that even with eager garbage collection, which reduces memory consumption, e.g., see Fig. 5(b), the effect of our techniques on overall memory usage is not particularly impressive and leaves room for further optimization.

usage is not reduced at the same level as average learned clause lengths, using abbreviations leads to shorter running time. Furthermore, the ability to factor out assumptions is crucial for a new form of clause minimization, which gave another substantial improvement. In general, we improved the state-of-the-art in MUS extraction considerably.

Our prototype MINISAT$_{abb}$ uses rather basic data structures, which can be improved in several ways. Memory usage could be reduced by a more sophisticated implementation of managing abbreviations. Further, in the current implementation, identical definitions are not shared. A hashing scheme could cheaply detect this situation and would allow to reuse already existing definitions instead of introducing new ones. This should reduce memory usage further and also speed up the initialization phase.

Finally, it would be interesting to combine the techniques presented in this paper with more recent results on MUS preprocessing [10] and preprocessing under assumptions [19,37] resp. inprocessing [38]. We also want to apply our approach to high-level MUS extraction [7,8,16].

Software and more details about the experiments including log files are available at `http://fmv.jku.at/musaddlit`.

References

1. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT'04. Volume 2919 of LNCS., Springer (2004) 502–518
2. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. ENTCS **89**(4) (2003) 543 – 560
3. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: Proc. CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications. (2003)
4. Marques-Silva, J., Lynce, I., Malik, S.: 4. In: Conflict-Driven Clause Learning SAT Solvers. Volume 185 of Frontiers in Artificial Intel. and Applications. IOS Press (February 2009) 131–153
5. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2) (2012) 97–116
6. Grégoire, É., Mazure, B., Piette, C.: Extracting MUSes. In: Proc. ECAI'06. Volume 141 of Frontiers in Artificial Intel. and Applications., IOS Press (2006) 387–391
7. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Proc. FMCAD'10, IEEE (2010) 221–229
8. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Proc. SAT'11. Volume 6695 of LNCS. (2011) 174–187
9. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: Proc. FMCAD'11, FMCAD (2011) 37–40
10. Belov, A., Järvisalo, M., Marques-Silva, J.: Formula preprocessing in MUS extraction. In Piterman, N., Smolka, S., eds.: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013). Volume 7795 of Lecture Notes in Computer Science., Springer (2013) 110–125
11. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Proc. SAT'06. Volume 4121 of LNCS. Springer (2006) 252–265
12. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Refinement strategies for verification methods based on datapath abstraction. In: Proc. ASP-DAC'06, IEEE (2006) 19–24
13. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. CoRR **abs/0712.1097** (2007)
14. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Proc. EUROCAST'09. Volume 5717 of LNCS. (2009) 304–311
15. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof - and counterexample - based abstraction. In: Proc. FMCAD'10. (2010) 181–188
16. Nöhrer, A., Biere, A., Egyed, A.: Managing SAT inconsistencies with HUMUS. In: Proc. VaMoS'12, ACM (2012) 83–91
17. Huang, J.: Extended clause learning. AI **174**(15) (2010) 1277–1284
18. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: Proc. AAAI'10. (2010)
19. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Proc. SAT'12. (2012) 242–255
20. Goldberg, E.I., Novikov, Y.: BerkMin: A fast and robust Sat-solver. In: Proc. DATE'02, IEEE (2002) 142–149
21. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proc. IJCAI'09, Morgan Kaufmann (2009) 399–404

22. Audemard, G., Lagniez, J.M., Mazure, B., Saïs, L.: On freezing and reactivating learnt clauses. In: Proc. SAT'11. Volume 7317 of LNCS., Springer (2011) 188–200
23. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: Proc. ICCAD'01. (2001) 279–285
24. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Proc. SAT'09, Springer (2009) 237–243
25. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: a minimally-unsatisfiable subformula extractor. In: Proc. DAC'04, ACM (2004) 518–523
26. Chu, G., Harwood, A., Stuckey, P.J.: Cache conscious data structures for boolean satisfiability solvers. JSAT **6**(1-3) (2009) 99–120
27. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Proc. AAAI/IAAI'98. (1998) 431–437
28. Huang, J.: The effect of restarts on the effectiveness of clause learning. In: Proc. IJCAI'07. (2007)
29. Pipatsrisawat, K., Darwiche, A.: RSat!2.0: SAT solver description. Technical Report Technical Report D153, Automated Reasoning Group, Comp. Scienc. Dept., UCLA (2007)
30. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
31. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47** (1993)
32. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Proc. SAT'08. Volume 4996 of LNCS., Springer (2008) 28–33
33. Ryvchin, V., Strichman, O.: Local restarts. In: Proc. SAT'08. Volume 4996 of LNCS., Springer (2008) 271–276
34. Audemard, G., Simon, L.: Refining restarts strategies for SAT and UNSAT. In: Proc. CP'12. Volume 7514 of LNCS., Springer (2012) 118–126
35. Biere, A.: Lingeling and friends at the SAT Competition 2011. FMV Report Series Technical Report 11/1, Johannes Kepler University, Linz, Austria (2011)
36. van der Tak, P., Ramos, A., Heule, M.: Reusing the assignment trail in CDCL solvers. JSAT **7**(4) (2011) 133–138
37. Nadel, A., Ryvchin, V., Strichman, O.: Preprocessing in incremental SAT. In: Proc. SAT'12. (2012) 256–269
38. Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Proc. IJCAR'12. (2012) 355–370

Chapitre 4

An Experimentally Efficient Method for (MSS, CoMSS) Partitioning

Article paru dans les actes de « *the 28th AAAI Conference on Artificial Intelligence* » (AAAI'14), pages 2666 – 2673, juillet 2014.

Co-écrit avec Éric Grégoire et Bertrand Mazure.

Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence

An Experimentally Efficient Method for (MSS,CoMSS) Partitioning

Éric Grégoire, Jean-Marie Lagniez, Bertrand Mazure
CRIL
Université d'Artois & CNRS
rue Jean Souvraz F-62307 Lens, France
{gregoire,lagniez,mazure}@cril.fr

Abstract

The concepts of MSS (Maximal Satisfiable Subset) and CoMSS (also called Minimal Correction Subset) play a key role in many A.I. approaches and techniques. In this paper, a novel algorithm for partitioning a Boolean CNF formula into one MSS and the corresponding CoMSS is introduced. Extensive empirical evaluation shows that it is more robust and more efficient on most instances than currently available techniques.

Introduction

The concept of MSS (Maximal Satisfiable Subset) has long been playing a key role in many logic-based A.I. approaches and techniques. Especially, in the knowledge representation domain, reasoning on the basis of inconsistent premises is often modeled using MSSes as basic building blocks (in e.g. belief revision, knowledge fusion, argumentation theory or nonmonotonic reasoning (see surveys and edited volumes of early seminal contributions in each of these fields in e.g. (Fermé and Hansson 2011), (Grégoire and Konieczny 2006), (Besnard and Hunter 2008) and (Ginsberg 1987)).

The complement of one MSS in an inconsistent set of formulas, noted CoMSS, is often called Minimal Correction Subset: it forms a minimal subset of formulas to be dropped in order to restore consistency. Consequently, the concept of CoMSS is a crucial paradigm in both model-based diagnosis (see seminal works in (Hamscher, Console, and de Kleer 1992)) and consistency-restoring techniques in knowledge-bases.

Beyond logic-based frameworks, MSSes and CoMSSes find similar roles in constraint reasoning when a problem is over-constrained and thus yields no solution (see for example in constraint networks (Rossi, van Beek, and Walsh 2006; Lecoutre 2009) or in optimisation (Chinneck 2008)).

In this paper, we are interested in the ubiquitous problem within the aforementioned domains that consists in partitioning an unsatisfiable Boolean CNF formula into one MSS and the corresponding CoMSS, when maximality is considered with respect to set-theoretical inclusion (vs. cardinality). Computing one such partition is in $\Delta_2^P = P^{NP}$ (Papadimitriou 1993). Despite this heavy cost in the worst case, several computational approaches to extract one (MSS,CoMSS) partition have been proposed that prove empirically viable for many instances.

Clearly, extracting one (MSS,CoMSS) pair is close to the basic version of the MAX-SAT problem that involves soft clauses only and consists in extracting one maximal (with respect to cardinality) satisfiable subset of a Boolean CNF formula. Every solution to MAX-SAT is one MSS whereas every MSS is not necessarily a solution to MAX-SAT. Interestingly, as advocated by (Marques-Silva et al. 2013), specific algorithms to compute one MSS can prove faster than MAX-SAT-dedicated ones; in this respect, computing one MSS can be used to deliver one approximate solution to MAX-SAT when a solution to this latter problem is out of reach.

Noticeably, CoMSS and MUS (namely, Minimal Unsatisfiable Subset) are dual concepts. One MUS is an unsatisfiable subset such that dropping any one of its clauses leads to satisfiability. MUSes can be computed as hitting sets of CoMSSes since any CoMSS contains one clause of each MUS (Bailey and Stuckey 2005; Grégoire, Mazure, and Piette 2007a; 2007b; Liffiton and Sakallah 2008).

In the paper, a novel algorithm to compute one (MSS,CoMSS) partition of an unsatisfiable Boolean CNF formula is thus introduced. Extensive empirical evaluation shows that it is more robust and more efficient on most instances than currently available techniques to extract one MSS or one CoMSS.

Logical preliminaries and basic concepts

Let $\mathcal{L}$ be a standard Boolean logical language built on a finite set of Boolean variables and usual connectives (namely, $\wedge$, $\vee$, $\neg$ and $\rightarrow$ standing for conjunction, disjunction, negation and material implication, respectively). Any formula in $\mathcal{L}$ can be represented (while preserving satisfiability) in clausal normal form (in short, a CNF) using a set (interpreted conjunctively) of clauses, where a clause is a formula made of a finite disjunction of literals and where a literal is Boolean variable that can negated. Formulas will be noted using lower-case Greek letters such as α, β, etc. Sets of formulas will be represented using letters like Γ, Δ, Σ, etc. We note by $\overline{\alpha}$ the opposite of the clause α, i.e. the set of unit clauses formed of the opposite of the literals of α.

An interpretation $\mathcal{I}$ assigns values from $\{0, 1\}$ to every

Boolean variable, and, following usual compositional rules, to all formulas of $\mathcal{L}$. A formula α is consistent or satisfiable when there exists at least one interpretation $\mathcal{I}$ that satisfies it, i.e. such that $\mathcal{I}(\alpha) = 1$; $\mathcal{I}$ is then called a model of α and is represented by the set of variables that it satisfies. α can be deduced from Σ, noted $\Sigma \models \alpha$, when α is satisfied in all models of Σ.

SAT is the NP-complete problem that consists in checking whether or not a CNF is satisfiable, i.e., whether or not there exists a model of all clauses in the CNF. In the paper, we often refer to CDCL-SAT solvers, which are currently the most efficient logically complete SAT solvers and that exploit so-called Conflict-Direct Clause Learning features (see e.g. (Marques-Silva and Sakallah 1996; Moskewicz et al. 2001; Zhang et al. 2001; Zhang and Malik 2002; Eén and Sörensson 2004; Audemard and Simon 2009; 2012)).

Let Σ be a CNF.

Definition 1 (MSS) $\Gamma \subseteq \Sigma$ *is a Maximal Satisfiable Subset (*MSS*) of* Σ *iff* Γ *is satisfiable and* $\forall \alpha \in \Sigma \setminus \Gamma$, $\Gamma \cup \{\alpha\}$ *is unsatisfiable.*

Definition 2 (CoMSS) $\Gamma \subseteq \Sigma$ *is a Minimal Correction Subset (*MCS *or* CoMSS*) of* Σ *iff* $\Sigma \setminus \Gamma$ *is satisfiable and* $\forall \alpha \in \Gamma$, $\Sigma \setminus (\Gamma \setminus \{\alpha\})$ *is unsatisfiable.*

Accordingly, Σ can always be partitioned into a pair made of one MSS and one CoMSS. Obviously, such a partition needs not be unique.

A *core* of Σ is a subset of Σ that is unsatisfiable. Minimal cores, with respect to set-theoretical inclusion, are called MUSes.

Definition 3 (MUS) $\Gamma \subseteq \Sigma$ *is a Minimal Unsatisfiable Subset (*MUS*) of* Σ *iff* Γ *is unsatisfiable and* $\forall \alpha \in \Gamma$, $\Gamma \setminus \{\alpha\}$ *is satisfiable.*

Under its basic form where all clauses are considered as being soft, i.e., not compulsory satisfiable, MAX-SAT consists in delivering one maximal (with respect to cardinality) subset of Σ. As emphasized earlier, every solution to MAX-SAT is one MSS and there exists a hitting set duality between CoMSSes and MUSes.

Our algorithm to partition a CNF is based on the *transition clause* concept.

Definition 4 (Transition clause) *Let* Σ *be an unsatisfiable* CNF. *A clause* $\alpha \in \Sigma$ *is a transition clause of* Σ *iff* $\Sigma \setminus \{\alpha\}$ *is satisfiable.*

The transition clause concept has proved crucial in approaches to MUS extraction. Especially, (Belov and Marques-Silva 2011) takes advantage of the following property in order to enhance the performance of a MUS extractor.

Property 1 *Let* Σ *be an unsatisfiable* CNF. *When* $\alpha \in \Sigma$ *is a transition clause,* α *belongs to every* MUS *of* Σ.

Likewise, a concept of transition constraint has proved powerful for the extraction of minimal unsatisfiable sets of constraints in the general setting of constraint networks (Hemery et al. 2006; Grégoire, Lagniez, and Mazure 2013a; 2013b). The transition clause concept will be a key concept in our partitioning algorithm, too.

Algorithm 1: BLS (Basic Linear Search)

input : one CNF Σ
output: one (MSS,CoMSS) partition of Σ

$\Pi \leftarrow \Omega \leftarrow \emptyset$;
foreach $\alpha \in \Sigma$ **do**
  if SAT $(\Pi \cup \alpha)$ **then** $\Pi \leftarrow \Pi \cup \{\alpha\}$;
  else $\Omega \leftarrow \Omega \cup \{\alpha\}$;
return (Π,Ω) ;

The simplest algorithm to partition a CNF into one (MSS,CoMSS) pair is given in Algorithm 1 and called Basic Linear Search (in short, BLS). It inserts within the MSS under construction every clause that is satisfiable together with this subset. Clearly, when the number of clauses in Σ is n, BLS requires n calls to a SAT-solver. This basic algorithm is only given here because we believe that it is the best starting point to understand the new approach that we are going to introduce. A survey of the currently more efficient approaches is provided for example in (Marques-Silva et al. 2013).

CMP: a novel partitioning algorithm

Algorithm 2: CMP
(Computational Method for Partitioning)

input : one CNF Σ
output: one (MSS,CoMSS) partition of Σ

$(\Pi, \Psi) \leftarrow$ `ApproximatePartition` (Σ) ;
$\Omega \leftarrow \emptyset$;
$\Gamma \leftarrow \Psi$; `// Working subset of` Ψ
while $\Psi \neq \emptyset$ **do**
  `extendSatPart` $(\Pi, \Gamma, \Omega, \Psi)$;
  if $\Psi \neq \emptyset$ **then**
    $\alpha \leftarrow$ `selectClause` (Γ) ;
    $(\mathcal{I}, \Delta) \leftarrow$ `solve` $(\Pi \cup (\Gamma \setminus \{\alpha\}) \cup \overline{\alpha})$;
    `//Returns a model` $\mathcal{I}$ `if SAT and`
    `//` $(\mathcal{I} = \emptyset$ `and core` $\Delta)$ `otherwise`
    if $\mathcal{I} \neq \emptyset$ **then** `// Transition clause`
      $\Omega \leftarrow \Omega \cup \{\alpha\}$;
      $\Pi \leftarrow \Pi \cup \{\beta \in (\Psi \setminus \{\alpha\})$ s.t. $\mathcal{I}(\beta) = 1\} \cup \overline{\alpha}$;
      $\Gamma \leftarrow \Psi \leftarrow \{\beta \in (\Psi \setminus \{\alpha\})$ s.t. $\mathcal{I}(\beta) = 0\}$;
    else
      $\Gamma \leftarrow \Gamma \setminus \{\alpha\}$;
      `exploitCore` $(\Pi, \Gamma, \Psi, \Delta, \alpha)$;
return ($\Pi \cap \Sigma$,Ω);

The novel algorithm for partitioning an unsatisfiable CNF into one MSS and its corresponding CoMSS is called CMP for Computational Method for Partitioning. Its skeleton is

Procedure `ApproximatePartition(Σ)`

```
1 if SAT(Σ) then        // Satisfiable instance
2 |   return (Σ, ∅);
3 else
4 |   Let P be the progress-saving interpretation delivered
  |   by the previous call to the CDCL-SAT solver ;
5 |   return({β∈Σ s.t. P(β)=1},{β∈Σ s.t. P(β)=0});
```

depicted in Algorithm 2, where framed boxes in the algorithm are optional and will be discussed in a subsequent section.

CMP delivers a partition (Π, Ω) of a CNF Σ where Π is one MSS of Σ (and, consequently, where Ω is the corresponding CoMSS). At this point, it is important to stress that:

1. The loop structure of CMP makes a search for transition clauses that differs from BLS in a fundamental original way: the iteration loops in both algorithms are very different. In the worst case, CMP makes $O(n^2)$ calls to a SAT-solver whereas BLS makes a linear number of such calls, only. However, our extensive experimental studies show that even without the optional features given in the framed boxes, the basic structure of CMP is more efficient than every current competitor on many instances.
2. The full version of CMP thus includes several additional optional features encapsulated within framed boxes in Algorithm 2. We will show that each of them incrementally increases the performance of the approach: some of the options were already included in other approaches (`extendSatPart`), other ones are exploited in an original way (especially a concept that we will call "opposite enforcement on backbones") and the last ones are new concepts (parts of `exploitCore`).

CMP: step-by-step presentation

CMP starts by computing one MSS approximation of Σ using the `ApproximatePartition` procedure (Alg.2:line1), which calls a CDCL-SAT solver to check the satisfiability of Σ. In the positive case, `ApproximatePartition` delivers the partition $(\Sigma, \emptyset)$ and since $\Psi = \emptyset$, CMP does not enter its main loop and ends. In the negative case, `ApproximatePartition` partitions Σ into a couple (Π, Ψ), where Π is a subset of the MSS under construction that is obtained (Proc.`ApproximatePartition`:line5) by making use of the so-called *progress-saving interpretation* (Pipatsrisawat and Darwiche 2007) $\mathcal{P}$ of Σ that has been delivered by the call to the CDCL-SAT solver (Proc.`ApproximatePartition`:line1): all clauses of Σ satisfied by $\mathcal{P}$ are recorded inside Π. In CDCL-SAT solvers, the progress-saving interpretation captures the final state of variables which can hopefully exhibit a "good" approximation of an MSS of Σ. Moreover, the facilities provided by CDCL-SAT solvers to record useful information like the final state of variables ordering, the polarity cache and clause deletion/restart strategies for subsequent calls to the solver, can be exploited as well (Audemard et al. 2011; Guo and Lagniez 2011).

Π will evolve to deliver the final MSS whereas the final CoMSS Ω under construction is initialized to the empty set (Alg.2:line 2). When the optional parts of the algorithm are taken into account, Π might not only be enriched with clauses that are moved from Ψ but also with additional (entailed) clauses that are expected to speed the whole process. Actually, in this case, $\Sigma = (\Pi \cap \Sigma) \cup \Psi \cup \Omega$ will be an invariant of the loop of the algorithm.

Let us leave apart the optional framed boxes for the moment and see also Figure 1. At the beginning (Alg.2:line 3) and whenever a clause α is added into Ω (Alg.2:line 10), a working subset Γ of Ψ is initialized to Ψ (Alg.2:line 12). While every clause of Ψ has not been dispatched either in the CoMSS or in the MSS under construction, the algorithm searches for a clause $\alpha \in \Psi$ that is a transition clause of $\Pi \cup \Gamma \cup \{\alpha\}$ where $\Gamma \cup \{\alpha\} \subseteq \Psi$, i.e., such that $\Pi \cup \Gamma \cup \{\alpha\}$ is unsatisfiable whereas $\Pi \cup \Gamma$ is satisfiable. When such an α is discovered, it is moved inside the CoMSS under construction Ω (Alg.2:line 10). All clauses of $\Pi \cup \Psi$ that are satisfied (resp. falsified) by the exhibited model $\mathcal{I}$ from the last satisfiability test of $\Pi \cup \Gamma$ are moved inside Π (Alg.2:line 11) (resp. remain in Ψ (Alg.2:line 12)). The working subset Γ is then initialized to Ψ according to the new value of this latter set (Alg.2:line 12). When α is not such a transition clause (Alg.2:line 14), α is expelled from the working subset $\Gamma \in \Psi$ (Alg.2:line 15). The `selectClause` function (line 7) selects one clause α from Γ. Our implementation of this function is based on the well-known VSIDS (Zhang and Malik 2002) heuristic.

CMP: proof of correctness

For space reasons, we only give here the main keys of the proof, together with useful observations that help to understand why CMP delivers the intended results.

First, it is easy to see that the loop always terminates. Remember that Γ is initialized to Ψ (alg.2:lines 3 &12). Since Σ is unsatisfiable whereas Π is always satisfiable, there always exists at least one $\alpha \in \Gamma$ such that the satisfiability test (Alg.2:line 9) is positive thanks to the fact the previously negatively tested α are expelled from Γ, successively (Alg.2:line 14). When this positive test occurs, Ψ decreases (Alg.2:line 12). An extreme case occurs when satisfiability is only found when $\Gamma = \{\alpha\}$.

Now, let us explain why the final Ω is a CoMSS of Σ. First, consider the first time when some α is going to be moved inside Ω (Alg.2:line 10) because α has been recognized as a transition clause of $\Pi \cup \Gamma$. This corresponds to Figure 1, where $\Omega = \emptyset$. The following property ensures that α can be inserted within Ω since when α is a transition clause of $\Pi \cup \Gamma$, α belongs to any Θ that is a CoMSS of Σ and that is such that $\Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$.

Property 2 *Let Σ be an unsatisfiable* CNF. *When $\Sigma = \Pi \cup \Psi$ with Π satisfiable, $\Gamma \subseteq \Psi$ and $\alpha \in \Gamma$ is a transition clause of $\Pi \cup \Gamma$, we have that $\forall \Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$ s.t. $\Theta \in$ CoMSS(Σ): $\alpha \in \Theta$.*

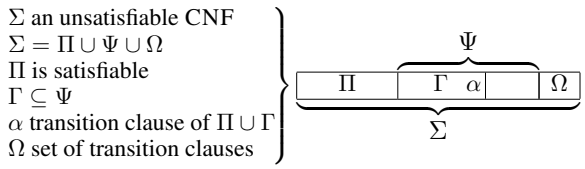


Figure 1: The various sets of clauses

Proof. By contradiction. Assume that $\exists\ \Theta \in \text{CoMSS}(\Sigma)$ s.t. $\Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$ and $\alpha \notin \Theta$, i.e. $\Theta \subseteq \Psi \setminus \Gamma$. If $\Theta \in \text{CoMSS}(\Sigma)$ then $\Sigma \setminus \Theta$ is satisfiable. Moreover, since $\Gamma \subseteq \Psi$ and $\Theta \subseteq \Psi \setminus \Gamma$, we have: $\Psi = (\Psi \setminus \Gamma) \cup \Gamma = ((\Psi \setminus \Gamma) \setminus \Theta) \cup \Theta \cup \Gamma$. Thus, $\Sigma = \Pi \cup \Psi = \Pi \cup \Gamma \cup ((\Psi \setminus \Gamma) \setminus \Theta) \cup \Theta$. By consequence, $(\Sigma \setminus \Theta) \supseteq (\Pi \cup \Gamma)$. As $\Sigma \setminus \Theta$ is satisfiable, we have also $\Pi \cup \Gamma$ satisfiable which is in contradiction with $\Pi \cup \Gamma$ being unsatisfiable (because α is a transition clause of $\Pi \cup \Gamma$). □

Now, thanks to the following property it is easy to prove by induction that every time a clause α is inserted within Ω, α belongs to a same CoMSS than the clauses that were previously inserted within Ω.

Property 3 *Let Σ be an unsatisfiable* CNF. *Assume $\Sigma = \Pi \cup \Psi$, Π satisfiable, $\Gamma \subseteq \Psi$ and that $\alpha \in \Gamma$ is a transition clause of $\Pi \cup \Gamma$. If $\Theta \subseteq (\Psi \setminus \Gamma)$ is a* CoMSS *of $\Sigma \setminus \{\alpha\}$ then $\Theta \cup \{\alpha\}$ is a* CoMSS *of Σ.*

Proof. By contraposition. Assume that $(\Theta \cup \{\alpha\})$ is not a CoMSS of Σ. Let us prove that Θ is not a CoMSS of $\Sigma \setminus \{\alpha\}$. $(\Theta \cup \{\alpha\})$ is not a CoMSS of Σ means that either $\Sigma \setminus (\Theta \cup \{\alpha\})$ is unsatisfiable (1) or $(\Theta \cup \{\alpha\})$ is not minimal, i.e. $(\Theta \cup \{\alpha\})$ is an upper-approximation of a CoMSS of Σ (2).

(1) $\Sigma \setminus (\Theta \cup \{\alpha\})$ unsatisfiable entails that $(\Sigma \setminus \{\alpha\}) \setminus \Theta$ is unsatisfiable and consequently Θ is not a CoMSS of $\Sigma \setminus \{\alpha\}$.

(2) $(\Theta \cup \{\alpha\})$ is an upper-approximation of a CoMSS of Σ implies that there exists a CoMSS Φ of Σ s.t. $\Phi \subset (\Theta \cup \{\alpha\})$. In other words, there exists $\beta \in (\Theta \cup \{\alpha\})$, s.t. $\beta \notin \Phi$. Property 2 ensures that $\alpha \in \Phi$, we obtain that $\alpha \neq \beta$.

$\Sigma \setminus (\Theta \cup \{\alpha\})$ is satisfiable (because $(\Theta \cup \{\alpha\})$ is an upper-approximation of a CoMSS of Σ). Thus, $\Sigma \setminus (\Theta \cup \{\alpha\} \cup \{\beta\})$ is also satisfiable. Therefore, $(\Sigma \setminus \{\alpha\}) \setminus (\Theta \setminus \{\beta\})$ is satisfiable and thus Θ is not a CoMSS of $\Sigma \setminus \{\alpha\}$. □

CMP: optional improvements

We have investigated four candidate improvements of CMP. They are encapsulated in the framed boxes of Algorithm 2.

First, following (Birnbaum and Lozinskii 2003) and (Marques-Silva et al. 2013), we have exploited the idea that checking the satisfiability of a logically weaker (and thus hopefully easier) CNF where some clauses (here, the clauses belonging to Γ) are replaced by their disjunction (here, noted $\bigvee \Gamma$) can prove informative. In case of unsatisfiability of $\Pi \cup \{\bigvee \Gamma\}$, $\Pi \cup \Gamma$ is also unsatisfiable. In case of satisfiability, the set of clauses from Γ that are containing literals satisfied in the exhibited model, is satisfiable together with Π and can thus be moved inside Π. This is the role of

Procedure `extendSatPart` $(\Pi, \Gamma, \Omega, \Psi)$

1 $(\mathcal{I}, \Delta) \leftarrow$ `solve` $(\Pi \cup \{\bigvee \Gamma\})$;
2 **if** $\mathcal{I} \neq \emptyset$ **then**
3 $\quad \Pi \leftarrow \Pi \cup \{\beta \in \Psi$ s.t. $\mathcal{I}(\beta) = 1\}$;
4 $\quad \Gamma \leftarrow \{\beta \in \Psi$ s.t. $\mathcal{I}(\beta) = 0\}$;
5 **else**
6 $\quad \Omega \leftarrow \Omega \cup \Gamma$;
7 $\quad \Gamma \leftarrow \Psi \leftarrow \Psi \setminus \Gamma$;

the `extendSatPart` function in line 5 of Algorithm 2. This feature proved important for the efficiency of the CLD method from (Marques-Silva et al. 2013).

The second tentative improvement (Algo2:line 11) is related to backbone literals ((Monasson et al. 1999) and more recently e.g. (Kilby et al. 2005)); it has been investigated in CoMSS extraction procedures by (Marques-Silva et al. 2013), too. $\overline{\alpha}$ is the set of unit clauses made of the opposite literals of α. As α will here belong to the intended CoMSS, this entails that $\overline{\alpha}$ is a deductive consequence of the current value of Π (and any superset of it by monotonicity). These learnt unit clauses are put inside Π in the hope to speed up the future satisfiability tests of supersets of Π.

An original tentative improvement occurs when it is tested whether or not α is a transition clause (Algo2:line 8). In the positive case, $\Pi \cup \Gamma \setminus \{\alpha\}$ is satisfiable whereas $\Pi \cup \Gamma$ is unsatisfiable. Accordingly, a same result will be obtained by checking the satisfiability of $\Pi \cup (\Gamma \setminus \{\alpha\}) \cup \{\overline{\alpha}\}$, where the set of unit clauses $\overline{\alpha}$ is expected to speed up the test. To the best of our knowledge, this use of backbones literals in a logically stronger set of premises that is not only consistent with $\overline{\alpha}$ but entails $\overline{\alpha}$ is novel in the search for CoMSS. We call this feature *opposite enforced*, in short *oe*. Interestingly, CMP allows this additional feature because a transition clause is found in case of satisfiability whereas other approaches find transition constraints when unsatisfiability is encountered.

Procedure `exploitCore` $(\Pi, \Gamma, \Psi, \Delta, \alpha)$

1 **if** $\Delta \cap \overline{\alpha} = \emptyset$ **then** $\Gamma \leftarrow \Gamma \cap \Delta$;
2 **elsif** $\Delta \cap \Gamma = \emptyset$ **then**
3 $\quad \Pi \leftarrow \Pi \cup \{\alpha\}$;
4 $\quad \Psi \leftarrow \Psi \setminus \{\alpha\}$;
5 **else** $\Pi \leftarrow \Pi \cup \{\Delta \setminus \overline{\alpha} \vDash \alpha\}$;

A fourth improvement is the `exploitCore` function (Algo2:line 15) that refines in the following original way Γ thanks to the computed core Δ. Three cases need be distinguished.

1. When Δ does not contain any clause of $\overline{\alpha}$, Γ can be assigned its set-theoretic intersection with Δ as a transition constraint can always be found within this intersection.
2. Otherwise, if no clause of Γ belongs to Δ then this entails that Δ is only made of clauses of $\overline{\alpha}$ and Π. Consequently, $\Pi \wedge \overline{\alpha}$ is unsatisfiable, i.e., α is a deductive consequence of Π. Thus, α can be moved into the satisfiable part Π.

3. Else, Δ is built from clauses of $\overline{\alpha}$ and Γ (and possibly from clauses of Π, too). In this case, since at least one clause of $\overline{\alpha}$ is in Δ and since Δ is unsatisfiable, we have that $(\Delta \setminus \overline{\alpha}) \wedge \overline{\alpha}$ is unsatisfiable, i.e., α is a deductive consequence of $\Delta \setminus \overline{\alpha}$. Although it cannot be represented by a set of clauses in a direct manner, this information can be exploited and implemented easily (and without significant additional space cost) using clause selectors markers *à la* (Oh et al. 2004), as follows. Each clause β of Σ is associated to a new corresponding dedicated literal noted λ_β, called selector, and β is replaced by $\beta \vee \neg\lambda_\beta$ in Σ. A clause is active in (i.e., belongs to) a set of clauses if its selector is assigned to 1. With these selectors, $\Delta \setminus \overline{\alpha} \vDash \alpha$ can be represented by the clause $(\neg\lambda_{\delta_1} \vee \cdots \vee \neg\lambda_{\delta_m} \vee \lambda_\alpha)$ where δ_i ($i \in [1..m]$) are the m clauses of $\Delta \setminus \overline{\alpha}$. When all clauses of $\Delta \setminus \overline{\alpha}$ are active then α must be also active.

It is easy to show that all those improvements do not alter the correctness of the algorithm.

Related work

Handling over-constrained systems though maximal satisfiable or minimal correction subsets has long been an active subject of research in A.I. (see pioneering work for example in (Meseguer et al. 2003)). In the general constraint networks setting, a seminal approach called QUICKXPLAIN has been described in (Junker 2004). In the same framework, improved techniques can be found in e.g. (Hemery et al. 2006).

In order to solve a constraint network instance or extract its MSSes and CoMSSes, it is often more efficient to encode the network through a set of Boolean clauses and benefit from SAT technology (provided that the size of the Boolean instance does not blow-up). In the SAT domain, many approaches have been proposed to compute MSSes and CoMSSes. The earliest approaches were based on DPLL-based SAT solvers (see e.g. (Birnbaum and Lozinskii 2003)) but are quite inefficient compared to more recent approaches (Liffiton and Sakallah 2008) based on MAX-SAT. As stressed in (Marques-Silva et al. 2013), the latter approaches also proved more efficient than various algorithms based on iterative calls to a SAT solver like (Bailey and Stuckey 2005). Much research has also been conducted in model-based diagnosis. Noticeably, a recent approach, called FastDiag (hereafter noted BFD for Basic FastDiag), (Felfernig, Schubert, and Zehentner 2012) has adapted QUICKXPLAIN from (Junker 2004) to the Boolean case and proves often very competitive.

Recently (Marques-Silva et al. 2013) experimentally compared the best MSS and CoMSS extraction approaches in the Boolean setting, namely the above BFD algorithm, BLS (depicted in Algorithm 1), EFD and ELS for Enhanced FastDiag and Enhanced Linear Search through additional features discussed in (Marques-Silva et al. 2013). The same authors also proposed a novel algorithm for computing CoMSSes that they experimentally showed to be the most efficient and robust one for CoMSSes computation, compared with the aforementioned list of approaches. Roughly, this algorithm, called CLD, extracts one CoMSS by using the `extendSatPart` principle iteratively (and also using backbone considerations, among other things). The authors showed that CLD experimentally outperforms all the competing approaches.

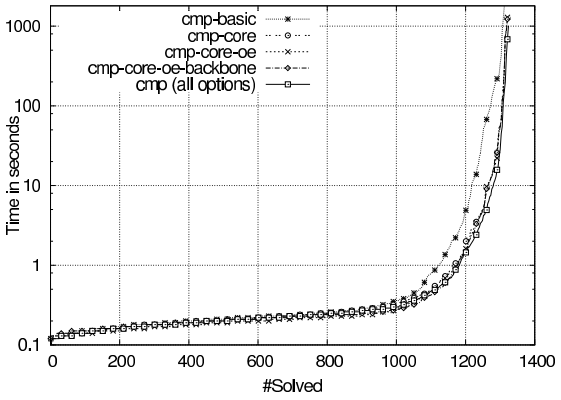


Figure 2: CMP variants on SAT/MAX-SAT instances

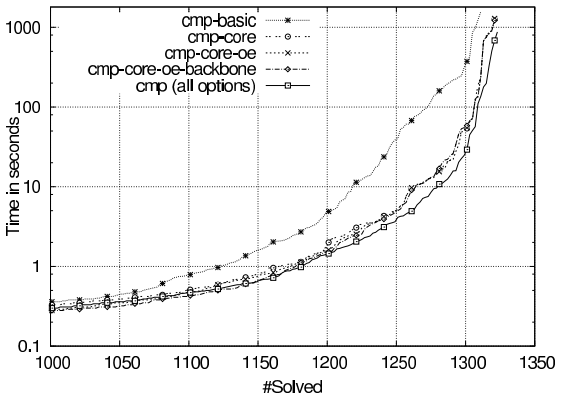


Figure 3: Zoom in on Figure 2

Although it implements the backbones and `extendSatPart` features, CMP is very different in nature as its key principle is the search for transition constraints rather than iterating on `extendSatPart`. Moreover, it includes the novel *opposite enforced* and `exploitCore` features.

Experimental results

All experimentations have been conducted on Intel Xeon E5-2643 (3.30GHz) processors with 7.6Gb RAM on Linux CentOS. Time limit was set to 30 minutes.

Two series of benchmarks have been considered. The first one was made of the 1343 benchmarks used and referred to in (Marques-Silva et al. 2013): they are small-sized industrial-based instances from SAT competitions www.satcompetition.org and structured instances from the MAX-SAT evaluations maxsat.ia.udl.cat:81. We enriched this experimentation setting by also considering a second series of benchmarks, made of *all* the 295 instances used for the 2011 MUS competition organized in parallel with the SAT one. Note that in all Figures that we are going to present,

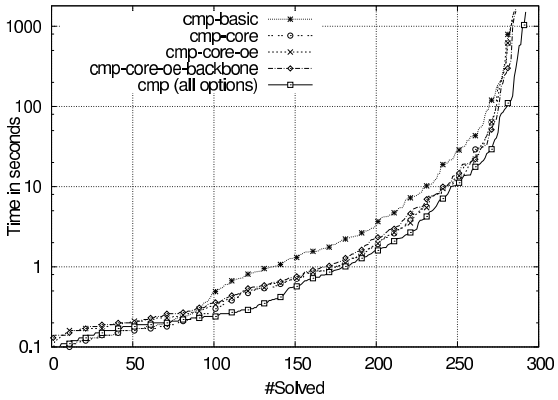


Figure 4: CMP variants on MUS instances

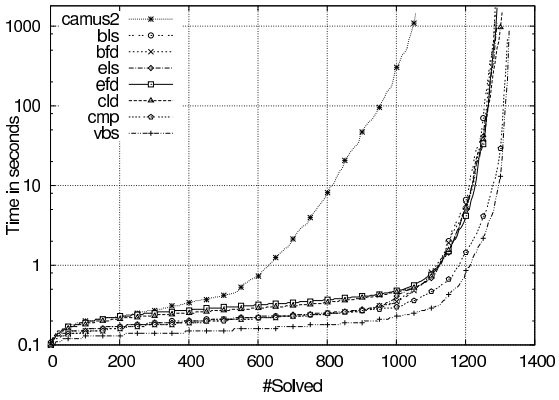


Figure 5: Comparison on SAT and MAX-SAT instances

	SAT/MAX-SAT inst.	MUS inst.
Number of instances	1343	295
CMP-*basic*	1312	285
CMP-*core*	1321	285
CMP-*core-oe*	1321	286
CMP-*core-oe-backbone*	1322	286
CMP (all options)	1327	292

Table 1: Number of successfully partitioned instances

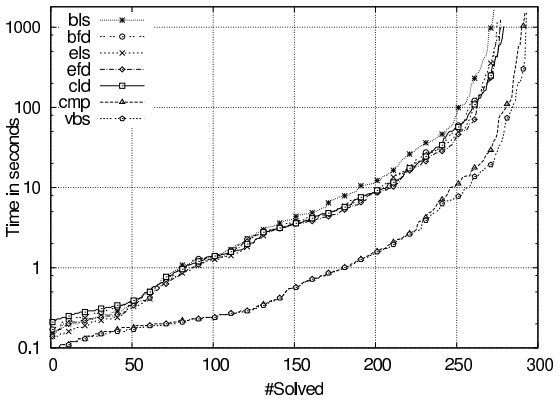


Figure 6: Comparison on MUS instances

when the y-values represent the CPU times to partition a given number x of instances through different approaches, this does not mean that the x partitioned instances are necessarily identical and that each instance is partitioned in an identical way by all the considered approaches.

CMP is implemented in C++. MINISAT (Eén and Sörensson 2004) was selected as the CDCL SAT-solver. CMP and all experimentation data are available from www.cril.univ-artois.fr/documents/cmp/.

First, the actual benefits of the four optional parts of CMP have been assessed experimentally.

The basic version of CMP, i.e. Algorithm 2 without any of the options, was re-named CMP-*basic*. CMP-*basic* was then tentatively enhanced by taking the options into account in an incremental way. Only adding `exploitCore` (Algo2:line 15) gave rise to CMP-*core*; adding also the opposite-enforced feature (Algo2:line 8) yielded CMP-*core-oe*. CMP-*core-oe-backbone* was obtained by also activating the backbone literals tentative enhancement (Algo2:line 11). From now on, CMP denotes Algorithm 2 with all options, which thus included `extendSatPart` (Algo2:line 5) as well.

Figures 2 and 4 show gradual improvements when each of the options was taken into account in a cumulative way: each additional option allowed for some additional efficiency gain. Figure 3 is a focus on the rightest part of curves of Figure 2. Table 1 shows that the number of successful partitionings also increased according to the considered range of options. Noticeably, the `exploitCore` (*core*) option delivered the most significant improvement in terms of both computing time and number of successfully partitioned instances. Other ways to combine the options together allowed similar observations to be made.

Then, we compared CMP with existing approaches that we mentioned earlier: namely, the BFD, BLS, CLD, EFD and ELS algorithms, which are described in (Marques-Silva et al. 2013) and implemented in the MCS_{LS} tool logos.ucd.ie/wiki/doku.php?id=mcsls. The version 2 of CAMUS (sun.iwu.edu/~mliffito/camus/ (Liffiton and Sakallah 2008; 2009)), named CAMUS2, was also tested on SAT/MAX-SAT benchmarks but this earlier system allowed a significantly smaller number of instances to be partitioned, only. Table 2 shows that CMP-*basic* itself allows more instances to be partitioned than any of the competitors. Figures 5 and 6 also compare the investigated approaches. We have also drawn the VBS (*Virtual Best Solver*) curve, which represents for each instance the best computing time amongst the tested methods. Clearly, CMP appeared best performing and was very close to VBS. For each method, Table 2 gives the number of instances that were successfully partitioned, with the highest score for CMP, too.

Discussion and perspectives

The extensive experimentations that we have conducted show that CMP is more robust than previous approaches and allows more Boolean instances to be successfully par-

	SAT/MAX-SAT inst.	MUS inst.
Number of instances	1343	295
BLS	1287	273
BFD	1287	276
ELS	1293	277
EFD	1291	277
CLD	1307	279
CMP-*basic*	1312	285
CMP (all options)	1327	292
VBS	1327	293

Table 2: Number of successfully partitioned instances

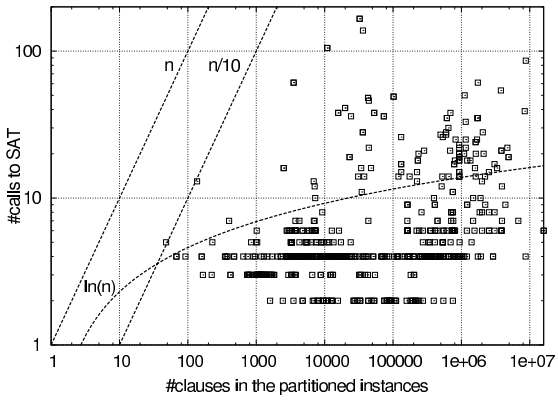


Figure 7: Number of calls by CMP to the SAT solver

titioned. Obviously, this does not entail that CMP would be more efficient for *every* instance. In this respect, it is worth mentioning that CMP exhibits a higher worst-case complexity than for example BLS, namely the basic linear search approach. Indeed, when n is the number of clauses in the instance, CMP requires $O(n^2)$ calls to a SAT solver in the worst case whereas BLS requires a linear number of such calls, only. Note however that the number of calls by CMP to the SAT solver always remains very significantly lower than n for all successfully partitioned instances (Figure 7).

We envision several paths for further research. First, the method could be enhanced by making use of incremental SAT solvers (Lagniez and Biere 2013; Audemard, Lagniez, and Simon 2013). Second, CMP can offer a way to approximate MAX-SAT when solving this latter problem is out of reach for hard instances. In this respect, although the features in CMP are not directly exportable to MAX-SAT algorithms, we believe that the way MSSes are computed in CMP can lead to novel ways to compute MAX-SAT. Also, CMP delivers approximate solutions for a basic version of MAX-SAT: in the future, we plan to study how to push the envelope in order to address *weighted* MAX-SAT. Finally, it would also be interesting to extend CMP to address the problem of enumerating CoMSS (or MSS) and study whether or not this could improve recent practical computational results about this issue (Marques-Silva et al. 2013).

References

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, IJCAI'09, 399–404.

Audemard, G., and Simon, L. 2012. Refining restarts strategies for sat and unsat. In *Principles and Practice of Constraint Programming - 18th International Conference (CP'12)*, volume 7514 of *Lecture Notes in Computer Science*, 118–126. Springer.

Audemard, G.; Lagniez, J.-M.; Mazure, B.; and Saïs, L. 2011. On freezing and reactivating learnt clauses. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing (SAT'11)*, 188–200. Springer.

Audemard, G.; Lagniez, J.-M.; and Simon, L. 2013. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT'13)*, volume 7962 of *Lecture Notes in Computer Science*, 309–317. Springer.

Bailey, J., and Stuckey, P. J. 2005. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'2005)*, volume 3350 of *Lecture Notes in Computer Science*, 174–186. Springer.

Belov, A., and Marques-Silva, J. 2011. Accelerating MUS extraction with recursive model rotation. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'11)*, 37–40. FMCAD Inc.

Besnard, P., and Hunter, A. 2008. *Elements of Argumentation*. The MIT Press.

Birnbaum, E., and Lozinskii, E. L. 2003. Consistent subsets of inconsistent systems: structure and behaviour. *J. Exp. Theor. Artif. Intell.* 15(1):25–46.

Chinneck, J. W. 2008. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*, volume 118 of *International Series in Operations Research & Management Science*. Springer.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03). Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Felfernig, A.; Schubert, M.; and Zehentner, C. 2012. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM* 26(1):53–62.

Fermé, E. L., and Hansson, S. O. 2011. AGM 25 years - twenty-five years of research in belief change. *J. Philosophical Logic* 40(2):295–331.

Ginsberg, M. 1987. *Readings in nonmonotonic reasoning*. M. Kaufmann Publishers.

Grégoire, É., and Konieczny, S. 2006. Logic-based approaches to information fusion. *Information Fusion* 7(1):4–18.

Grégoire, É.; Lagniez, J.-M.; and Mazure, B. 2013a. Improving MUC extraction thanks to local search. *CoRR* abs/1307.3585.

Grégoire, É.; Lagniez, J.-M.; and Mazure, B. 2013b. Questioning the importance of WCORE-like minimization steps in MUC-finding algorithms. In *Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'13)*, 923–930.

Grégoire, É.; Mazure, B.; and Piette, C. 2007a. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2300–2305.

Grégoire, É.; Mazure, B.; and Piette, C. 2007b. Local-search extraction of MUSes. *Constraints* 12(3):325–344.

Guo, L., and Lagniez, J.-M. 2011. Dynamic polarity adjustment in a parallel SAT solver. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'11)*, 67–73.

Hamscher, W.; Console, L.; and de Kleer, J., eds. 1992. *Readings in Model-Based Diagnosis*. Morgan Kaufmann.

Hemery, F.; Lecoutre, C.; Saïs, L.; and Boussemart, F. 2006. Extracting MUCs from constraint networks. In *Proc. of the 17th European Conference on Artificial Intelligence (ECAI'06)*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, 113–117. IOS Press.

Junker, U. 2004. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'04)*, 167–172. AAAI Press.

Kilby, P.; Slaney, J. K.; Thiébaux, S.; and Walsh, T. 2005. Backbones and backdoors in satisfiability. In *Proceedings, The 20th National Conference on Artificial Intelligence (AAAI'05)*, 1368–1373. AAAI Press / The MIT Press.

Lagniez, J.-M., and Biere, A. 2013. Factoring out assumptions to speed up mus extraction. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT'13)*, volume 7962 of *Lecture Notes in Computer Science*, 276–292. Springer.

Lecoutre, C. 2009. *Constraint Networks: Techniques and Algorithms*. Wiley.

Liffiton, M. H., and Sakallah, K. A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning* 40(1):1–33.

Liffiton, M. H., and Sakallah, K. A. 2009. Generalizing core-guided Max-SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, volume 5584 of *Lecture Notes in Computer Science*, 481–494. Springer.

Marques-Silva, J., and Sakallah, K. A. 1996. GRASP: a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design (ICCAD'96)*, 220–227. IEEE Computer Society.

Marques-Silva, J.; Heras, F.; Janota, M.; Previti, A.; and Belov, A. 2013. On computing minimal correction subsets. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*.

Meseguer, P.; Bouhmala, N.; Bouzoubaa, T.; Irgens, M.; and Sánchez, M. 2003. Current approaches for solving over-constrained problems. *Constraints* 8(1):9–39.

Monasson, R.; Zecchina, R.; Kirkpatrick, S.; Selman, B.; and Troyansky, L. 1999. Determining computational complexity from characteristic 'phase transitions'. *Nature* 400:133.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 530–535. ACM.

Oh, Y.; Mneimneh, M. N.; Andraus, Z. S.; Sakallah, K. A.; and Markov, I. L. 2004. AMUSE: A minimally-unsatisfiable subformula extractor. In *Proceedings of the 41th Design Automation Conference (DAC'04)*, 518–523. ACM.

Papadimitriou, C. H. 1993. *Computational Complexity*. Addison-Wesley.

Pipatsrisawat, K., and Darwiche, A. 2007. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, 294–299. Springer.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*. Elsevier.

Zhang, L., and Malik, S. 2002. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, 17–36. Springer.

Zhang, L.; Madigan, C.; Moskewicz, M.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. of the Proceedings of IEEE/ACM International Conference on Computer Design (ICCAD'01)*, 279–285.

Chapitre 5

Boosting MCSes Enumeration

Article paru dans les actes de « *the 27th International Joint Conference on Artificial Intelligence* » (IJCAI'18), pages 1309 – 1315, juillet 2018.

Co-écrit avec Éric Grégoire et Yacine Izza.

Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Boosting MCSes Enumeration

Éric Grégoire, Yacine Izza, Jean-Marie Lagniez
CRIL - Université d'Artois & CNRS, Lens, France
{gregoire,izza,lagniez}@cril.fr

Abstract

The enumeration of all Maximal Satisfiable Subsets (MSSes) or all Minimal Correction Subsets (MCSes) of an unsatisfiable CNF Boolean formula is a useful and sometimes necessary step for solving a variety of important A.I. issues. Although the number of different MCSes of a CNF Boolean formula is exponential in the worst case, it remains low in many practical situations; this makes the tentative enumeration possibly successful in these latter cases. In the paper, a technique is introduced that boosts the currently most efficient practical approaches to enumerate MCSes. It implements a model rotation paradigm that allows the set of MCSes to be computed in an heuristically efficient way.

1 Introduction

Computing one *maximal*[1] *satisfiable subset* of clauses, noted MSS, within an unsatisfiable CNF Boolean formula is a cornerstone task in various A.I. domains ranging from model-based diagnosis (see e.g., the seminal work in [Reiter, 1987], a readings book [Hamscher *et al.*, 1992] or some more recent research work in [Felfernig *et al.*, 2012; Marques-Silva *et al.*, 2015]) to various paradigms of belief change (see e.g., [Fermé and Hansson, 2011] for a survey of the field). In the Boolean setting, reasoning in a credulous [Reiter, 1980] way about contradictory information represented by an unsatisfiable CNF Σ can amount to reasoning about one MSS of Σ. Interpreted as a set of constraints, an unsatisfiable CNF formula Σ represents an over-constrained problem [Meseguer *et al.*, 2003] for which no solution exists. When Φ is one maximal satisfiable subset of Σ and when Ψ is defined as $\Sigma \setminus \Phi$, Ψ is one minimal subset of Σ such that dropping Ψ from the problem makes this one become feasible. Accordingly, Ψ is sometimes called a *minimal correction subset* of Σ (hence the interchangeable notations MCS and CO-MSS for this concept). In the worst case, the computation of an MSS or an MCS is a hard computational task since the basic problem of checking whether a set of clauses forms one MCS is DP-complete [Chen and Toda, 1995], which is also the complexity of the SAT-UNSAT problem. However, in many practical situations, computing one MSS is routinely performed by SAT-based tools (see e.g., [Marques-Silva *et al.*, 2013; Grégoire *et al.*, 2014; Bacchus *et al.*, 2014; Mencía *et al.*, 2015; 2016] among others).

The enumeration of all MCSes or all MSSes of Σ is even more computationally challenging. This task is a useful and sometimes necessary step in order to implement some forms of skeptical reasoning in abstract argumentation [Lagniez *et al.*, 2015] or in the presence of contradictions, which is a general issue that can be traced back to early seminal works about nonmonotonic logics [Bobrow, 1980]. It plays also a role in infeasibility analysis of a set of clauses, as the computation of all *minimal unsatisfiable subsets*, in short MUSes, can rely on the success of this enumeration (see e.g., [Liffiton and Sakallah, 2008; Grégoire *et al.*, 2007; Nadel *et al.*, 2014; Bacchus and Katsirelos, 2015; 2016; Previti and Marques-Silva, 2013; Liffiton *et al.*, 2016] among others). Although the number of different MCSes of the same formula is exponential in the worst case, it remains low in many practical situations; this makes the tentative enumeration of all MCSes possibly successful in these latter cases.

In the paper, a technique is introduced that boosts the currently most efficient practical techniques to enumerate the MCSes of an unsatisfiable Boolean formula Σ. It is based on a form of so-called model rotation paradigm [Belov and Marques-Silva, 2011a; Nadel *et al.*, 2014; Bacchus and Katsirelos, 2015]. We show that it allows the set of MCSes of Σ to be computed in an heuristically efficient way.

The paper is organized as follows. The technical background and the well-known MSS and MCS concepts are briefly reviewed in the preliminaries. In section 3, the key paradigms of transition clauses and clause selectors are recalled; their roles are illustrated through the basic linear search for one MCS. Section 4 presents a basic MCSes enumeration algorithm before an original advanced one is step-by-step described in section 5. In section 6, the use of model rotation in this latter algorithm is explained. Then, we present our extensive experimental study. Promising paths for further research are briefly introduced in the conclusion.

2 Technical Background

We consider a standard language of formulas $\mathcal{L}$ of Boolean logic. $\neg$, $\vee$, $\wedge$ and $\Rightarrow$ represent the negation, disjunction, conjunction and material implication connectives, respectively.

[1] We always consider set-inclusion maximality in this paper.

A literal is either a Boolean variable or its negation. A clause is a formula that consists of a disjunction of literals. A CNF (clausal normal form) formula is a conjunction (also represented as a set) of clauses. $\alpha, \beta, \ldots$ and $\Sigma, \Delta, \Phi, \Psi, \ldots$ denote formulas and sets of formulas, respectively. An interpretation μ assigns values from $\{0, 1\}$ to every Boolean variable, and, following usual compositional rules, to all formulas of $\mathcal{L}$. A formula α is satisfiable iff there exists at least one interpretation μ that satisfies α; μ is then called a model of α. Accordingly, μ satisfies a non-empty clause α when there exists at least one literal of α that is assigned 1 by μ; μ satisfies a CNF Σ iff μ satisfies all clauses of Σ. Any formula of $\mathcal{L}$ can be rewritten under an equisatisfiable CNF formula.

The core, MUS, MSS and MCS cross-related concepts are defined as follows. Let Σ be a CNF formula.

Definition 1. (Core) A CNF Σ' is a *core* of Σ iff $\Sigma' \subseteq \Sigma$ and Σ' is unsatisfiable.

Definition 2. (MUS) A *minimal unsatisfiable subset* (in short, MUS) Υ of Σ is a core of Σ such that $\forall \alpha \in \Upsilon$, $\Upsilon \setminus \{\alpha\}$ is satisfiable.

When a CNF Σ is unsatisfiable, it can be split into one MSS, i.e., one *maximal satisfiable subset* of Σ, and its set-theoretical complement in Σ, namely, one *minimal correction subset*, for short one MCS, of Σ.

Definition 3. (MSS) A *maximal satisfiable subset* (in short, MSS) Φ of Σ is a subset $\Phi \subseteq \Sigma$ that is satisfiable and such that $\forall \alpha \in \Sigma \setminus \Phi$, $\Phi \cup \{\alpha\}$ is unsatisfiable.

Definition 4. (MCS) A *minimal correction subset* (in short MCS, also called CO-MSS) Ψ of Σ is a set $\Psi \subseteq \Sigma$ whose complement in Σ, i.e., $\Sigma \setminus \Psi$, is an MSS of Σ.

Example 1. *Let Σ be an unsatisfiable* CNF *formed by a set of clauses $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$, where $\alpha_1 = a \vee b$, $\alpha_2 = \neg a \vee b$, $\alpha_3 = a \vee \neg b$, $\alpha_4 = \neg a \vee \neg b$, $\alpha_5 = \neg b$, $\alpha_6 = b$. The* MCS*es of Σ are $\{\alpha_1, \alpha_6\}$, $\{\alpha_2, \alpha_6\}$, $\{\alpha_3, \alpha_5\}$ and $\{\alpha_4, \alpha_5\}$. The* MUS*es of Σ are $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$, $\{\alpha_1, \alpha_2, \alpha_5\}$, $\{\alpha_3, \alpha_4, \alpha_6\}$ and $\{\alpha_5, \alpha_6\}$.*

Notice that one MSS of Σ can be interpreted as one approximate solution of Max-SAT on Σ, since maximality in MSS is about set-inclusion vs. cardinality. The currently most efficient approaches and tools to compute one MSS are described in [Grégoire *et al.*, 2014] and [Mencía *et al.*, 2015; 2016].

The MSS and MCS paradigms can be extended into the so-called Partial-MSS and Partial-MCS concepts in order to handle the situation where $\Sigma = \langle \Sigma_1, \Sigma_2 \rangle$, where Σ_1 and Σ_2 are sets of hard and soft clauses, respectively: hard clauses are required to belong to any Partial-MSS of Σ; they thus do not belong to any Partial-MCS of Σ. In the sequel, Σ_1 and Σ_2 always denote sets of hard and soft clauses, respectively. Notice that when Σ_1 is unsatisfiable, Partial-MSSes of Σ do not exist. In all the other cases, there always exists at least one Partial-MCS of Σ that is included in Σ_2. Moreover, every Partial-MCS of $\langle \Sigma_1, \Sigma_2 \rangle$ is an MCS of $\Sigma_1 \cup \Sigma_2$. Conversely, every MCS of $\Sigma_1 \cup \Sigma_2$ that does not satisfy Σ_1 is not a Partial-MCS of $\langle \Sigma_1, \Sigma_2 \rangle$.

3 Basic Linear Search for one MCS

State-of-the-art MCS extractors often make use of so-called *clause selectors* as follows. Each clause α of Σ is augmented with its own (negated) selector, namely a fresh new literal $\neg s_\alpha$. This yields a new (relaxed) CNF formula Σ^S. A clause $\alpha \vee \neg s_\alpha$ in Σ^S is thus activated (resp., deactivated) when the literal s_α (resp., $\neg s_\alpha$) is set to 1. Selectors play the role of assumptions that can be activated/deactivated during the same search while useful information can be recorded at each step. Especially, when Σ' is shown unsatisfiable under some such assumptions, modern SAT solvers can often extract a subset of the assumptions that causes Σ' to be unsatisfiable [Eén and Sörensson, 2003; Lagniez and Biere, 2013; Audemard *et al.*, 2013]. We will exploit this feature in the MCSes enumeration algorithm. In the rest of the paper, we often implicitly refer to Σ^S and make no ontological difference between the selectors and the other literals in Σ^S.

The concept of transition clause (in short, TC) is also often a key paradigm for the currently most efficient approaches to compute one MUS [Grégoire *et al.*, 2007; Previti and Marques-Silva, 2013], one MSS or one MCS [Grégoire *et al.*, 2014].

Definition 5. (Transition Clause) A clause $\alpha \in \Sigma$ is a transition clause (in short, TC) of Σ when, at the same time, Σ is unsatisfiable and $\Sigma \setminus \{\alpha\}$ is satisfiable.

Thus, when a clause α is a TC of Σ, α (resp., $\Sigma \setminus \{\alpha\}$) is an MCS (resp., MSS) of Σ. Moreover, if α is a TC of Σ then α belongs to every MUS of Σ.

Although Σ might not contain any TC, state-of-the-art MCS-finding tools often take advantage of TCes in the following way. Starting from the empty set, they iteratively and greedily construct a set Σ': at each step, clauses from Σ that have not been considered so far are inserted one by one in Σ' until Σ' becomes unsatisfiable. The last considered clause is a TC for Σ' and is introduced in the MCS under construction. The corresponding linear search algorithm for computing one MCS is called `BLS`. State-of-the-art MCS-finding tools [Grégoire *et al.*, 2014; Bacchus *et al.*, 2014; Marques-Silva *et al.*, 2013] have grafted various improvements to this algorithm skeleton; they take advantage of disjoint cores, computed models, the exploitation of the disjunction of the clauses of the MCS under construction and backbone literals, mainly. In the rest of the paper, we refer to improved versions of `BLS` that include subsets of these additional features: they are noted `ELS` for Enhanced Linear Search.

`ELS` is easily adapted in such a way that it computes one Partial-MCS: the hard clauses are initially inserted in the MSS without selectors. An initial call to a SAT solver is necessary to check whether the set of the hard clauses is unsatisfiable: in the positive case, an empty set needs to be returned. In the following, when we refer to a Partial-MCS-finding algorithm, we consider a procedure with a set of hard clauses Σ_1 and a set of soft clauses Σ_2 as input parameters: namely, `ExtractPartialMCS`(Σ_1, Σ_2).

4 MCSes Enumeration

Alg. 1 describes the typical skeleton of usual current approaches [Liffiton and Sakallah, 2008; Liffiton *et al.*, 2016; Marques-Silva *et al.*, 2013] to enumerate all MCSes: each time one MCS is found, an additional clause is created. Its role is to prevent the same MCS from being computed again. It is made of the disjunction of all the selectors of the clauses in the discovered MCS. In Alg. 1, the blocking clauses are inserted in a set Δ. Partial-MCSes of $(\Sigma^S \cup \Delta, S)$ are computed while MCSes still exist, or, equivalently, while $\Sigma^S \cup \Delta$ is satisfiable.

Clearly, the time-consuming part of this algorithm lies in the multiple calls to a SAT oracle in the routine extracting one Partial-MCS. In this respect, [Previti *et al.*, 2017] has proposed to enhance the algorithm by recording, for caching purpose, the cores discovered during the successive computations of the different MCSes: this yields the currently most efficient MCSes enumeration algorithm, noted `mcscache-els` in this paper. In the next section, we propose another approach to decrease the number of calls to a SAT oracle; it is compatible with the caching technique. In the experimental section, we show that the proposed approach outperforms `mcscache-els`. Moreover, when combined with the caching technique, it yields an even more efficient algorithm. It is based on properties of transition clauses and on the so-called recursive model rotation paradigm; both are described in the next two sections.

Algorithm 1: `Enum-ELS` (Enumerate All MCSes Computed with the `ExtractPartialMCS` procedure);

Input : an unsatisfiable CNF formula Σ
Output : all MCSes of Σ

1 $\Sigma^S \leftarrow \{\alpha \vee \neg s_\alpha | \alpha \in \Sigma\}$; // with s_α fresh variables
2 $S \leftarrow \{s_\alpha | \alpha \in \Sigma\}$; // a set of selectors
3 $\Delta \leftarrow \emptyset$;
4 **while** $\Sigma^S \cup \Delta$ *is satisfiable* **do**
5 $M^- \leftarrow$ `ExtractPartialMCS`$(\Sigma^S \cup \Delta, S)$;
6 $output(M^-)$;
7 $\Delta \leftarrow \Delta \cup (\bigvee_{s_\alpha \in M^-} s_\alpha)$; // blocking clauses

5 More MCSes Thanks to Transition Clauses

First, the next property shows that any transition clause (in short, TC) α of an unsatisfiable subset $\Sigma' \subseteq \Sigma$ can be the starting point of a family of MCSes of Σ, where each member of this family contains α. This family is shown to capture the MCSes of Σ that are made of α together with any Partial-MCS of $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$,

Property 1. *Let Σ be an unsatisfiable* CNF *formula and let $\Sigma' \subseteq \Sigma$ such that Σ' contains at least one TC α.*
*For all Partial-*MCS*es Γ that can be built from $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$, $\Gamma \cup \{\alpha\}$ is an* MCS *of Σ.*

The previous property is easily adapted to address the case where Partial-MCSes are targeted, as the following corollary shows.

Corollary 1. *Let $\langle \Sigma_1, \Sigma_2 \rangle$ be a couple of* CNF *formulas such that Σ_1 is satisfiable and $\Sigma_1 \cup \Sigma_2$ is unsatisfiable. Assume that $\Sigma'_2 \subseteq \Sigma_2$ is such that $\Sigma_1 \cup \Sigma'_2$ contains at least one TC α of Σ'_2. For all Partial-*MCS*es Γ that can be built from $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha\}, \Sigma_2 \setminus \Sigma'_2 \rangle$, we have that $\Gamma \cup \{\alpha\}$ is a Partial-*MCS *of $\langle \Sigma_1, \Sigma_2 \rangle$.*

Now assume that we compute several TCes for a given subformula CNF $\Sigma' \subseteq \Sigma$. For each TC, it is possible to recursively apply the previous corollary in order to compute several Partial-MCSes. Moreover, the following property ensures that all these Partial-MCSes are different.

Property 2. *Let $\langle \Sigma_1, \Sigma_2 \rangle$ be a couple of* CNF *formulas such that Σ_1 is satisfiable and $\Sigma_1 \cup \Sigma_2$ is unsatisfiable. Assume that $\Sigma'_2 \subseteq \Sigma_2$ is such that $\Sigma_1 \cup \Sigma'_2$ contains at least two different TCes α_1 and α_2 that occur in Σ'_2. Then, for all Partial-*MCS*es Γ_1 and Γ_2 that can be built from $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha_1\}, \Sigma_2 \setminus \Sigma'_2 \rangle$ and $\langle \Sigma_1 \cup \Sigma'_2 \setminus \{\alpha_2\}, \Sigma_2 \setminus \Sigma'_2 \rangle$, respectively, we have that $\Gamma_1 \cup \{\alpha_1\}$ and $\Gamma_2 \cup \{\alpha_2\}$ are different Partial-*MCS*es of $\langle \Sigma_1, \Sigma_2 \rangle$.*

The latter property allows us to derive and justify the original recursive Algorithm 2, called `TC-MCS`, for *Transition-Clauses-Based Enumeration of* MCS*es*. This algorithm extends `ELS` by computing not just one but several MCSes in a recursive way by means of a model rotation method (lines 7-12, Alg. 2). Its input is a couple of CNF formulas $\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$; the output is a set of Partial-MCSes for this couple. Remember that U is the set of selectors corresponding to the clauses that have not been assigned so far to either an MCS or an MSS under construction. We assume that Σ_1 is satisfiable and thus, that $\Sigma_1 \cup \Sigma_2^S$ is satisfiable. Σ_2^S is actually built from Σ_2 using selectors as explained earlier. Notice that hard clauses do not need selectors as they must always be satisfied and thus be activated. The Partial-MCSes of $\langle \Sigma_1, \Sigma_2 \rangle$ are derived directly from the Partial-MCSes of $\langle \Sigma_1 \cup \Sigma_2^S, S \rangle$, where S is the set of unit clauses corresponding to all selectors.

The algorithm starts by checking if U is empty. In the positive case, the procedure returns $\emptyset$ since the set of Partial-MCSes of $\langle \Sigma_1 \cup \Sigma_2^S, U \rangle$ is empty as $\Sigma_1 \cup \Sigma_2^S$ is always satisfiable by construction. This is also the non-recursive step. Otherwise, the set of computed Partial-MCSes Θ and the set of selectors M^+ are initialized to the empty set. Remember that M^+ records the selectors corresponding to the clauses that can be activated when $\Sigma_1 \cup \Sigma_2^S \cup M^+$ is satisfiable. s_α is initialized to $\top$, i.e., the tautology.

Next, the loop (lines 3–6) incrementally augments M^+ with a sequence of s_α, which are removed from U. This process is iterated while U is not empty and, at the same time, while s_α is not a TC of $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$. s_α is a TC when $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ becomes unsatisfiable. Indeed, all the selectors inserted so far in M^+ have guaranteed that $\Sigma_1 \cup \Sigma_2^S \cup M^+$ remained satisfiable. Thus, when an incoming s_α makes the formula become unsatisfiable, this selector s_α is a TC. At the end of the loop, two cases can occur: (1) $s_\alpha \in M^+$. In this case, M^+ has captured all the selectors from U and the input formula was actually satisfiable. Accordingly, the empty set is returned; (2) $s_\alpha \notin M^+$ and thus s_α is a TC. In this case, we look for a core Γ of the current unsatisfiable formula $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ (line 8). It is easy to

see that s_α belongs to Γ and it is a TC of Γ. As already mentioned, modern SAT solvers return as a byproduct a core that is often smaller than the formula that is shown unsatisfiable. We choose to use this core instead of the latter formula since the core often contains more TCes. It is easy to prove by contradiction that every TC is present in the core. Once a core is extracted, a method is called in order to identify TCes belonging to the core. In the next section, we present an approach to achieve this process, but for the moment, we just assume that TCes are extracted. Notice that we have no guarantee that this approach is efficient and identifies s_α as a TC. For this reason, we directly add s_α into T, i.e., the set of transitions clauses, in order to ensure the termination of the algorithm. Then, for each identified TC $s_\beta \in (T \cap M^+)$ we recursively call the function with $\langle \Sigma_1 \cup \Sigma_2^S \cup (\Gamma \setminus \{\neg s_\beta\}), U \cup (M^+ \setminus \Gamma)\rangle$ as input. It is easy to show that these parameters match the conditions stated in Corollary 1. Accordingly, all the Partial-MCSes that we can compute from $\langle \Sigma_1 \cup \Sigma_2^S \cup (\Gamma \setminus \{\neg s_\beta\}), U \cup (M^+ \setminus \Gamma)\rangle$ can be augmented with s_β to yield a Partial-MCS of the input formula.

Algorithm 2: TC-MCS ($\langle \Sigma_1 \cup \Sigma_2^S, U\rangle$);

Input : $\langle \Sigma_1 \cup \Sigma_2^S, U\rangle$ a couple of CNF formulas
Output : Θ a set of Partial-MCSes of $\langle \Sigma_1 \cup \Sigma_2^S, U\rangle$

1 **if** $U = \emptyset$ **then return** $\emptyset$;
2 $\Theta \leftarrow \emptyset$; $M^+ \leftarrow \emptyset$; $s_\alpha \leftarrow \top$;
3 **while** $U \neq \emptyset$ *and* $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ *is satisfiable* **do**
4 $\quad M^+ \leftarrow M^+ \cup \{s_\alpha\}$;
5 $\quad s_\alpha \leftarrow$ **choose** $s_\alpha \in U$;
6 $\quad U \leftarrow U \setminus \{s_\alpha\}$;
7 **if** $s_\alpha \notin M^+$ **then**
8 $\quad \Gamma \leftarrow \texttt{Core}(\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\})$;
9 $\quad T \leftarrow \{s_\alpha\} \cup$ Find-TC(Γ);
10 $\quad$ **foreach** $s_\beta \in T \cap M^+$ **do**
11 $\quad\quad \Omega \leftarrow$ TC-MCS($\langle \Sigma_1 \cup \Sigma_2^S \cup (\Gamma \setminus \{\neg s_\beta\}), U \cup (M^+ \setminus \Gamma)\rangle$);
12 $\quad\quad \Theta \leftarrow \Theta \cup (\beta \times \Omega)$;
13 **return** Θ;

TC-MCS ($\langle \Sigma_1 \cup \Sigma_2^S, U\rangle$) can be inserted in Alg. 1 to yield a new enumeration algorithm for MCSes, depicted in Alg. 3.

Algorithm 3: Enum-ELS-RMR;

Input : an unsatisfiable CNF formula Σ
Output : all MCSes of Σ

1 $\Sigma^S \leftarrow \{\alpha \vee \neg s_\alpha | \alpha \in \Sigma\}$; // with s_α fresh variables
2 $S \leftarrow \{s_\alpha | \alpha \in \Sigma\}$; // a set of selectors
3 $\Delta \leftarrow \emptyset$;
4 **while** $\Sigma^S \cup \Delta$ *is satisfiable* **do**
5 $\quad \Theta \leftarrow$ TC-MCS ($\langle \Sigma^S \cup \Delta, S\rangle$);
6 $\quad$ **foreach** $M^- \in \Theta$ **do**
7 $\quad\quad output(M^-)$;
8 $\quad\quad \Delta \leftarrow \Delta \cup (\bigvee_{s_\alpha \in M^-} s_\alpha)$; // blocking clauses

6 Using Model Rotation

Let us now explain how the Find-TC procedure (line 9 of Alg. 2) computes additional TCes. A naive method would consist in computing all the MCSes of $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ that are singletons contained in M^+. As this complete direct method can be too time-consuming, we propose an incomplete process to compute additional TCes, based on the so-called recursive model rotation paradigm [Belov and Marques-Silva, 2011b] noted rmr. This latter paradigm has been initially defined in the context of computing one or several Minimal Unsatisfiable Subsets (MUSes) of an unsatisfiable CNF formula, where computing additional TCes in a fast way can also play a key role. As for [Bacchus and Katsirelos, 2015] where MUSes enumeration is targeted, we take advantage of the duality between MUSes and MCSes. However, we use the rmr paradigm for a very different task, which consists in enumerating MCSes. rmr is based on a model-theoretical perspective of TC: a TC of an unsatisfiable CNF formula Σ is any clause $\alpha \in \Sigma$ such that there exists a complete interpretation μ of Σ that satisfies $\Sigma \setminus \{\alpha\}$ (and falsifies α, otherwise the formula Σ would be satisfiable). Starting from a model μ of $\Sigma \setminus \alpha$, rmr consists in flipping variables from α and checking if this new interpretation μ' satisfies all the clauses of Σ except some α' of Σ. In the positive case, α' is marked as being a TC of Σ provided that it was not already marked as such, and the process is recursively repeated with μ' and α'.

rmr can thus compute several TCes when a first one is identified as such. In the search of one Partial-MCS, this situation occurs when we have shown that there exists a model μ of $\Sigma_1 \cup \Sigma_2^S \cup M^+$ and proven that $\Sigma_1 \cup \Sigma_2^S \cup M^+ \cup \{s_\alpha\}$ is unsatisfiable. μ then plays the role of the initial interpretation and we keep the detected TCes belonging to $\{\beta \in \Sigma_2 \mid s_\beta \in M^+\}$, only. Finally, the set of selectors associated with the discovered TCes can be returned. It is important to note that we make sure that the clause selectors are never flipped by the process.

Although the rmr process is a polynomial one, it turns out that it can be time consuming in practice; it can actually reduce the practical efficiency of the enumeration algorithm. Such a situation occurs when the number of clauses contained in the set Δ of blocking clauses becomes too large. To avoid such a drawback, we point out some sufficient conditions to deprive clauses of Δ of the rmr process.

First, let us show the possible critical role of Δ in the search for additional TCes. Consider $\langle \Sigma^S, S\rangle$ with $\Sigma^S = \{\neg s_1 \vee a \vee b, \neg s_2 \vee \neg a, \neg s_3 \vee \neg b\}$. Assume that one MCS, namely the singleton $\{s_1\}$, has been computed, already. Thus, at this step $\Delta = \{s_1\}$. Let us iterate the process and compute one more MCS and call TC-MCS on $\langle \Sigma^S \cup \Delta, S\rangle$. Let us suppose that we stop the main loop of TC-MCS when $M^+ = \{s_1, s_2\}$ and $s_\alpha = s_3$. The condition in line 7 is satisfied; one core that is the complete formula is computed. Then, we search more TCes of $\Sigma^S \cup S$ instead of $\Sigma^S \cup S \cup \Delta$. In this case, it is easy to show that s_1, s_2, s_3 are TCes and twice the same MCS will be computed. Obviously, a posteriori checking whether an MCS has already been computed by consulting Δ can be too time-consuming.

However, a useful feature is that Σ^S and Δ do not share

literals. Indeed, Δ is a set of positive clauses composed of selectors whereas these selectors occur negatively in Σ^S. Accordingly, the satisfiabilty of $\Sigma^S \cup \Delta$ can be split into two independent sub-problems following a partition $\{P, N\}$ of the set of selectors, where P and N denote the set of selectors that are positive and the set of negative ones, respectively. More precisely:

Property 3. *Let $\{P, N\}$ be a partition of S, Σ^S a* CNF *formula augmented with the set of selectors S and Δ a* CNF *formula built on selector variables and formed of positive clauses, only. $\Sigma^S \cup \Delta \cup \bigwedge_{s\in P} s \cup \bigwedge_{s\in N} \neg s$ is satisfiable if and only if $\Sigma^S \cup \bigwedge_{s\in P} s$ and $\Delta \cup \bigwedge_{s\in N} \neg s$ are satisfiable.*

When an MCS is under construction we are implicitly computing a bi-partition $\{M^+, M^-\}$ of S such that $\Sigma^S \cup \bigwedge_{s\in M^+} s$ and $\Delta \cup \bigwedge_{s\in M^-} \neg s$ are satisfiable. The aim is then to move as many as possible selectors from M^- to M^+ while keeping both $\Sigma^S \cup \bigwedge_{s\in M^+} s$ and $\Delta \cup \bigwedge_{s\in M^-} \neg s$ satisfiable. It is easy to prove that if the bi-partition $\{M^+, M^-\}$ of S is such that $\Sigma^S \cup \bigwedge_{s\in M^+} s$ and $\Delta \cup \bigwedge_{s\in M^-} \neg s$ are satisfiable, then if we move one element s' from M^- to M^+ such that $\Sigma^S \cup \bigwedge_{s\in M^+\cup\{s'\}} s$ is satisfiable then $\Delta \cup \bigwedge_{s\in M^-\setminus\{s'\}} \neg s$ is satisfiable (as all the clauses of Δ are positive, assigning one more positive selector cannot make the formula become unsatisfiable). Let us stress that symmetric results can be obtained when one element is moved from M^+ to M^-.

In order to avoid the need to consider Δ in the `rmr` procedure, we propose the following process. First, compute a partition $\{M^+, M^-\}$ of S that satisfies $\Sigma^S \cup \bigwedge_{s\in M^+} s$ and $\Delta \cup \bigwedge_{s\in M^-} \neg s$. M^+ is then used as starting point to be evolved into an MSS and all the clauses of M^- are marked as being candidates for being a TC. Then, each time we augment M^+ with a new selector, we have that $\Sigma^S \cup \bigwedge_{s\in M^+} s$ is satisfiable. Let us notice that, when we add an element to M^+, in some sense we "move" this element from M^- to M^+. Consequently, both $\Sigma^S \cup \bigwedge_{s\in M^+} s$ and $\Delta \cup \bigwedge_{s\in M^-} \neg s$ are satisfiable. In some sense, the marked selectors are responsible for the satisfiability of Δ. Thus, since M^+ is constructed such that $\Sigma^S \cup \bigwedge_{s\in M^+} s$ is satisfiable, it becomes useless to check the satisfiability of Δ during the `rmr` process when we forbid the marked clauses to be selected as TCes.

7 Experimental Study

We have implemented all our algorithms in C++ and used `Minisat http://minisat.se/` as backend SAT solver. We have selected the 866 benchmarks used in [Previti *et al.*, 2017; Marques-Silva *et al.*, 2013]: 269 instances are plain Max-SAT ones and the remaining 597 are Partial-Max-SAT ones. We have enriched this experimental setting by also considering a second series of plain Max-SAT benchmarks made of the instances from the MUS competition `http://www.satcompetition.org/2011`. Some of these instances were already present in the benchmarks proposed by [Previti *et al.*, 2017]. As we only kept the new ones, 1090 benchmarks were considered in total: 493 of them are plain Max-SAT instances and 597 are Partial-Max-SAT ones.

All experimentations have been conducted on Intel Xeon E52643 (3.30GHz) processors with 64Gb memory on Linux CentOS. Time-out was set to 1800 seconds for each run of an algorithm on an instance; memory-out was set to 8 Gb for each such run. All data, results and software used in the experimentations are available from `http://www.cril.fr/enumcs`.

First, we compared our own implementation of `Enum-ELS-RMR` (Alg. 4) with the same algorithm deprived of `rmr`. Our version of `ELS` included the exploitation of computed models [Grégoire *et al.*, 2014], as well as of backbone literals [Marques-Silva *et al.*, 2013]. The comparison was made in terms of the total number of computed MCSes for each benchmark instance. As shown by Fig. 1 the `rmr` paradigm allowed us to compute more (or the same number) of MCSes for every plain Max-SAT benchmark instance. The same result was obtained for most Partial-Max-SAT benchmarks, too.

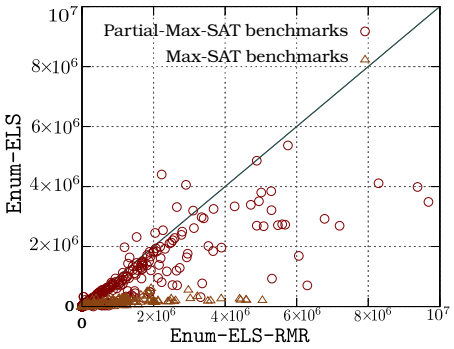


Figure 1: `Enum-ELS-RMR` vs. `Enum-ELS`

Then, we combined the caching technique with `Enum-ELS-RMR`, giving rise to `Enum-ELS-RMR-Cache`. Following the recommendation of [Previti *et al.*, 2017], we did not include the backbone feature in `ELS` since it might slow down the caching technique. `Enum-ELS-RMR-Cache` allowed a largest number of MCSes to be computed, most often (Fig. 2). Noticeably, it appeared that the instances for which `Enum-ELS-RMR-Cache` delivered a smaller number of MCSes where such that `Enum-ELS-Cache` already produced a smaller number of MCSes than `Enum-ELS-RMR`. Also, introducing the caching method leads to more memory-out conclusions. These two last points can be explained by the fact that, as already pointed out in [Previti *et al.*, 2017], the caching memory can become too large to handle.

Finally, we have compared the `mcscache-els` tool from [Previti *et al.*, 2017], which implements the caching technique on a state-of-the-art version of `Enum-ELS`, with `Enum-ELS-RMR-Cache`. Fig. 3 clearly shows that `Enum-ELS-RMR-Cache` outperforms `mcscache-els` for almost all benchmarks and the difference between both approaches is generally even more significant for the instances with the largest numbers of MCSes.

8 Conclusion and Perspectives

In this paper, we have enhanced the most efficient tool to enumerate all the minimal correction subsets (MCSes) of a Boolean CNF formula. Although the number of MCSes can

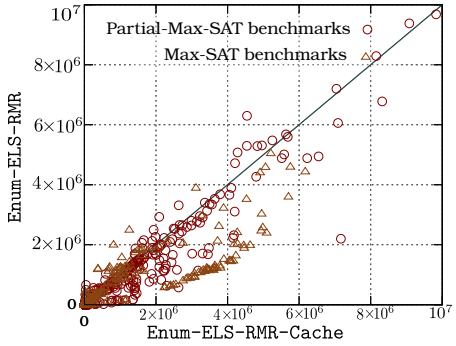


Figure 2: `Enum-ELS-RMR-Cache` vs. `Enum-ELS-RMR`

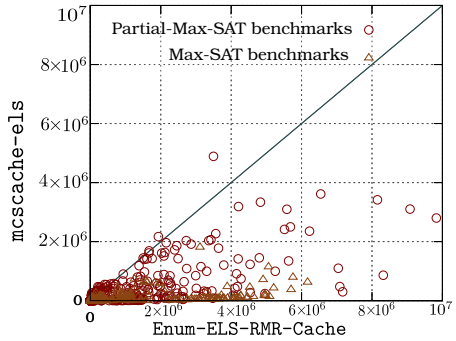


Figure 3: `Enum-ELS-RMR-Cache` vs. `mcscache-els`

be exponential in the worst case, it remains low in many real-life problems, especially in problems for which the number and the cardinality of the different minimal sources of unsatisfiability remain low. Indeed, there exists a hitting set correspondence between the set of MCSes and the set of MUSes, namely of minimal unsatisfiable subsets. Accordingly, computational progress made in enumerating all MCSes of CNF formulas when their number of MCSes is large opens new perspectives for enumerating all MUSes for the same formula. Indeed, the approches for listing all MUSes that first compute all MCSes before they compute MUSes from them can clearly benefit from these improvements obtained in the MCSes enumeration task. We believe that this study opens other various paths for further research, too. Specifically, Property 3 opens the way for some parallelization of the enumeration task. It could be interesting to extend `rmr` using forms of local search to detect additional TCes. Also, we plan to adapt this study to the enumeration of all preferred MCSes when the clauses of Σ obey some preference pre-orderings. Finally, as skeptical reasoning in the presence of conflicting information can amount to computing the intersection of all maximal satisfiable subsets (MSSes), new progress in enumerating all MSSes, and thus all MCSes, can prove valuable for implementing such forms of reasoning.

9 Proofs

Proof of Property 1. By contradiction. Let us assume that $\Gamma \cup \{\alpha\}$ is not an MCS of Σ. That means that either (1) $\Sigma \setminus (\Gamma \cup \{\alpha\})$ is unsatisfiable or (2) $\exists \beta \in \Gamma \cup \{\alpha\}$ such that $\Sigma \setminus ((\Gamma \cup \{\alpha\}) \setminus \{\beta\})$ is satisfiable.

Assume (1). As Γ is defined as a Partial-MCS of $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$, we have that Γ is an MCS of $(\Sigma \setminus \{\alpha\})$. Thus, $(\Sigma \setminus \{\alpha\}) \setminus \Gamma$ is satisfiable. This entails that $((\Sigma \setminus \{\alpha\}) \cup \{\alpha\}) \setminus (\Gamma \cup \{\alpha\})$ is satisfiable. This latter formula can be simplified into $\Sigma \setminus (\Gamma \cup \{\alpha\})$, which is thus also satisfiable. This contradicts (1). Accordingly, (1) never occurs.

Assume (2). Let us suppose that $\beta \in \Gamma$ (then $\beta \neq \alpha$). Since Γ is a Partial-MCS of $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$, we have that Γ is an MCS of $\Sigma \setminus \{\alpha\}$. Thus, $(\Sigma \setminus \{\alpha\}) \setminus (\Gamma \setminus \{\beta\})$ is unsatisfiable. This entails that $((\Sigma \setminus \{\alpha\}) \cup \{\alpha\}) \setminus ((\Gamma \setminus \{\beta\}) \cup \{\alpha\})$ is unsatisfiable. Because we supposed $\beta \neq \alpha$, we have $(\Gamma \setminus \{\beta\}) \cup \{\alpha\} = (\Gamma \cup \{\alpha\}) \setminus \{\beta\}$. Consequently, $\Sigma \setminus ((\Gamma \cup \{\alpha\}) \setminus \{\beta\})$ is unsatisfiable. This contradicts the assumption and thus β and α must be the same clause.

Because Γ is a Partial-MCS of $\langle \Sigma' \setminus \{\alpha\}, \Sigma \setminus \Sigma' \rangle$, we have $(\Sigma' \setminus \{\alpha\}) \cap \Gamma = \emptyset$ and $\Gamma \subseteq \Sigma \setminus \Sigma'$. By hypothesis $\alpha \in \Sigma'$, this means $\alpha \notin \Gamma$ and thus $((\Sigma' \setminus \{\alpha\}) \cup \{\alpha\}) \cap \Gamma = \emptyset$. Hence $\Sigma' \cap \Gamma = \emptyset$. If $\beta = \alpha$, then we have $\Sigma \setminus ((\Gamma \cup \{\alpha\}) \setminus \{\alpha\}) = \Sigma \setminus \Gamma$ is satisfiable according to assumption (2). Since $\Sigma' \subseteq \Sigma$, we also have $\Sigma' \setminus \Gamma$ is satisfiable. Since $\Sigma' \cap \Gamma = \emptyset$ we have that Σ' is satisfiable. This contradicts the hypothesis that asserts that Σ' is an unsatisfiable subset of Σ.

Proof of Corollary 1. Property 1 allows us to conclude that $\Gamma \cup \{\alpha\}$ is an MCS of $\Sigma_1 \cup (\Sigma_2' \setminus \{\alpha\}) \cup (\Sigma_2 \setminus \Sigma_2') \cup \{\alpha\}$, and thus of $\Sigma_1 \cup \Sigma_2$. Since $\Gamma \cup \{\alpha\} \subseteq \Sigma_2$, we directly conclude that $\Gamma \cup \{\alpha\}$ is a Partial-MCS of $\langle \Sigma_1, \Sigma_2 \rangle$.

Proof of Property 2. From Corollary 1 it is easy to show that $\Gamma_1 \cup \{\alpha_1\}$ and $\Gamma_2 \cup \{\alpha_2\}$ are both Partial-MCSes of $\langle \Sigma_1, \Sigma_2 \rangle$. Now, let us show that these Partial-MCSes are different. It is sufficient to show that $\alpha_1 \notin \Gamma_2$. By definition of a Partial-MCS, we have $\Gamma_2 \subseteq \Sigma_2 \setminus \Sigma_2'$. As $\alpha_1 \in \Sigma_2'$, it is straightforward that $\alpha_1 \notin \Gamma_2$ and then $\Gamma_1 \cup \{\alpha_1\} \neq \Gamma_2 \cup \{\alpha_2\}$.

Proof of Property 3. Let us show that if $\Sigma^S \cup \Delta \cup \bigwedge_{s \in P} s \cup \bigwedge_{s \in N} \neg s$ then $\Sigma^S \cup \bigwedge_{s \in P} s$ and $\Delta \cup \bigwedge_{s \in N} \neg s$ are satisfiable and *vice versa*. ($\Rightarrow$) Straightforward. ($\Leftarrow$) If $\Sigma^S \cup \bigwedge_{s \in P} s$ and $\Delta \cup \bigwedge_{s \in N} \neg s$ are both satisfiable, then there exists a model μ that satisfies $\Sigma^S \cup \bigwedge_{s \in P} s$. By definition of Σ^S, whatever the satisfiable interpretation considered is, it is always possible to flip the truth value of selectors from 1 to 0 and keep the resulting interpretation μ_P such that μ_P is a model of Σ^S (this is possible because assigning a selector to 0 deactivates clauses of Σ^S and then weakens this formula). Thus, since $P \cap N = \emptyset$, we can construct the interpretation μ_P^N that is equivalent to μ_P except on the truth value of the selectors belonging to N where we force them to be assigned to 0. It is clear that μ_P^N is a model of $\Sigma^S \cup \bigwedge_{s \in P} s$. Now, let us prove that μ_P^N is a model of $\Delta \cup \bigwedge_{s \in N} \neg s$ too. By construction, Δ only contains positive clauses composed of selectors. Then, whatever the model of $\Delta \cup \bigwedge_{s \in N} \neg s$ considered is, it is always possible to flip the truth value of some selectors from 0 to 1 and keep this interpretation as a model of Δ. Thus, since all models of $\Delta \cup \bigwedge_{s \in N} \neg s$ must satisfy $\bigwedge_{s \in N} \neg s$, we can construct the interpretation $\bigwedge_{s \in N} \neg s \wedge \bigwedge_{s \in P} s$ that satisfies $\Delta \cup \bigwedge_{s \in N} \neg s$. Thus, since Δ is only constructed on selector variables, it is easy to show that μ_P^N also satisfies $\Delta \cup \bigwedge_{s \in N} \neg s$. Consequently, μ_P^N satisfies $\Sigma^S \cup \Delta \cup \bigwedge_{s \in P} s \cup \bigwedge_{s \in N} \neg s$.

Acknowledgments

This work has been supported in part by the CPER DATA project funded by the *Hauts-de-France* Region. We thank the reviewers for their useful comments.

References

[Audemard *et al.*, 2013] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Proc. of SAT 2013*, pages 309–317, 2013.

[Bacchus and Katsirelos, 2015] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Proc. of CAV 2015*, pages 70–86, 2015.

[Bacchus and Katsirelos, 2016] Fahiem Bacchus and George Katsirelos. Finding a collection of muses incrementally. In *Proc. of CPAIOR 2016*, pages 35–44, 2016.

[Bacchus *et al.*, 2014] Fahiem Bacchus, Jessica Davies, Maria Tsimpoukelli, and George Katsirelos. Relaxation search: A simple way of managing optional clauses. In *Proc. of AAAI 2014*, pages 835–841, 2014.

[Belov and Marques-Silva, 2011a] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proc. of FMCAD 2011*, pages 37–40, 2011.

[Belov and Marques-Silva, 2011b] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proc. of FMCAD 2011*, pages 37–40, 2011.

[Bobrow, 1980] Daniel G. Bobrow. Special issue non non-monotonic logics. *Artificial Intelligence*, 13(1-2), 1980.

[Chen and Toda, 1995] Zhi-Zhong Chen and Seinosuke Toda. The complexity of selecting maximal solutions. *Information and Computation*, 119(2):231–239, 1995.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Selected Revised Papers of the 6th Int. Conf. of SAT 2003*, pages 502–518, 2003.

[Felfernig *et al.*, 2012] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012.

[Fermé and Hansson, 2011] Eduardo L. Fermé and Sven Ove Hansson. AGM 25 years - twenty-five years of research in belief change. *Journal of Philosophical Logic*, 40(2):295–331, 2011.

[Grégoire *et al.*, 2007] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *Proc. of IJCAI 2007*, pages 2300–2305, 2007.

[Grégoire *et al.*, 2014] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. An experimentally efficient method for (MSS, CoMSS) partitioning. In *Proc. of (AAAI 2014)*, pages 2666–2673, 2014.

[Hamscher *et al.*, 1992] Walter Hamscher, Luca Console, and Johan de Kleer. *Readings in model-based diagnosis*. Morgan Kaufmann, 1992.

[Lagniez and Biere, 2013] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In *Proc. of SAT 2013*, pages 276–292, 2013.

[Lagniez *et al.*, 2015] Jean-Marie Lagniez, Emmanuel Lonca, and Jean-Guy Mailly. Coquiaas: A constraint-based quick abstract argumentation solver. In *Proc. of ICTAI 2015*, pages 928–935, 2015.

[Liffiton and Sakallah, 2008] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[Liffiton *et al.*, 2016] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016.

[Marques-Silva *et al.*, 2013] João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *Proc. of IJCAI 2013*, pages 615–622, 2013.

[Marques-Silva *et al.*, 2015] João Marques-Silva, Mikolás Janota, Alexey Ignatiev, and António Morgado. Efficient model based diagnosis with maximum satisfiability. In *Proc. of IJCAI 2015*, pages 1966–1972, 2015.

[Mencía *et al.*, 2015] Carlos Mencía, Alessandro Previti, and João Marques-Silva. Literal-based MCS extraction. In *Proc. of IJCAI 2015*, pages 1973–1979, 2015.

[Mencía *et al.*, 2016] Carlos Mencía, Alexey Ignatiev, Alessandro Previti, and Joao Marques-Silva. MCS extraction with sublinear oracle queries. In *Proc. of SAT 2016*, pages 342–360, 2016.

[Meseguer *et al.*, 2003] Pedro Meseguer, Noureddine Bouhmala, Taoufik Bouzoubaa, Morten Irgens, and Martí Sánchez-Fibla. Current approaches for solving over-constrained problems. *Constraints*, 8(1):9–39, 2003.

[Nadel *et al.*, 2014] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *Journal of Satisfiability JSAT*, 9:27–51, 2014.

[Previti and Marques-Silva, 2013] Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *Proc. of AAAI 2013*, 2013.

[Previti *et al.*, 2017] Alessandro Previti, Carlos Mencía, Matti Järvisalo, and Joao Marques-Silva. Improving MCS enumeration via caching. In *Proc. of SAT 2017*, pages 184–194, 2017.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

Chapitre 6

Improving Model Counting by Leveraging Definability

Article paru dans les actes de « *the 25th International Joint Conference on Artificial Intelligence* » (IJCAI'16), pages 751 – 757, juillet 2016.

Co-écrit avec Emmanuel Lonca et Pierre Marquis.

Improving Model Counting by Leveraging Definability

Jean-Marie Lagniez and **Emmanuel Lonca** and **Pierre Marquis**
CRIL-CNRS and Université d'Artois, Lens, France
email:{lagniez, lonca, marquis}@cril.univ-artois.fr

Abstract

We present a new preprocessing technique for propositional model counting. This technique leverages definability, i.e., the ability to determine that some gates are implied by the input formula Σ. Such gates can be exploited to simplify Σ without modifying its number of models. Unlike previous techniques based on gate detection and replacement, gates do not need to be made explicit in our approach. Our preprocessing technique thus consists of two phases: computing a bipartition $\langle I, O\rangle$ of the variables of Σ where the variables from O are defined in Σ in terms of I, then eliminating some variables of O in Σ. Our experiments show the computational benefits which can be achieved by taking advantage of our preprocessing technique for model counting.

1 Introduction

Propositional model counting (alias the #SAT problem) is the task consisting in computing the number of models of a given propositional formula Σ. This problem and its direct generalization, weighted model counting, are central to many AI problems including probabilistic inference [Sang *et al.*, 2005; Chavira and Darwiche, 2008; Apsel and Brafman, 2012; Choi *et al.*, 2013] and forms of planning [Palacios *et al.*, 2005; Domshlak and Hoffmann, 2006]. They have also many applications outside AI, like in SAT-based automatic test pattern generation, for evaluating the vulnerability to malicious fault attacks in hardware circuits (see e.g., [Feiten *et al.*, 2012]). However, propositional model counting is computationally hard (a #P-complete problem), actually much harder in practice than the satisfiability issue (the SAT problem). Its significance explains why much effort has been spent for the last decade in developing new algorithms for model counting (either exact or approximate) which prove practical for larger and larger instances [Samer and Szeider, 2010; Bacchus *et al.*, 2003; Gomes *et al.*, 2009].

In this paper, we present a new preprocessing technique for improving exact model counting. Preprocessing techniques are nowadays acknowledged as computationally valuable for a number of automated reasoning tasks, especially SAT solving and QBF solving [Bacchus and Winter, 2004; Subbarayan and Pradhan, 2004; Lynce and Marques-Silva, 2003; Een and Biere, 2005; Piette *et al.*, 2008; Han and Somenzi, 2007; Heule *et al.*, 2010; Järvisalo *et al.*, 2012; Heule *et al.*, 2011]. As such, they are now embodied in some state-of-the-art SAT solvers, like `Glucose` [Audemard and Simon, 2009] which takes advantage of the `SatELite` preprocessor [Een and Biere, 2005], `Lingeling` [Biere, 2014] which has an internal preprocessor, and `Riss` [Manthey, 2012b] which takes advantage of the `Coprocessor` preprocessor [Manthey, 2012a].

Our approach elaborates on [Lagniez and Marquis, 2014], which describes a number of preprocessing techniques that can be exploited for improving the model counting task, computationally speaking. Among them is *gate detection and replacement*. Basically, every variable y of the input formula Σ which turns out to be defined in Σ in terms of other variables $X = \{x_1, \ldots, x_k\}$ can be replaced by its definition Φ_X, while preserving the number of models of Σ. Indeed, whenever a partial assignment over the variables of X is considered, either it is jointly inconsistent with Σ or every model of Σ which extends this partial assignment gives to y the same truth value. In [Lagniez and Marquis, 2014], literal equivalences, AND/OR gates and XOR gates are detected (either syntactically or using Boolean Constraint Propagation). The empirical results reported in [Lagniez and Marquis, 2014] about the preprocessor `pmc` equipped with the so-called *#eq* combination of preprocessings clearly show that huge computational benefits can be achieved through the detection and the replacement of gates. However, `pmc` remains limited due to the small number of families of gates which are targeted (literal, AND, XOR gates and their negations).

In order to fill the gap, our preprocessing technique to model counting aims at exploiting in a much more aggressive way the existence of gates within the input formula Σ. The key idea of our approach is that *one does not need to identify the gates themselves but it can be enough to determine that such gates exist*. To be more precise, it proves sufficient to detect that some definability relations between variables hold, without needing to identify the corresponding definitions. This distinction is of tremendous importance since on the one hand, the search space for the possible definitions Φ_X is very large (2^{2^k} elements up to logical equivalence, when X contains k variables), and on the other hand, in the general case, the size of any explicit definition Φ_X of y in Σ is

not polynomially bounded in $|\Sigma| + |X|$ unless NP $\cap$ coNP $\subseteq$ P/poly (which is considered unlikely in complexity theory) [Lang and Marquis, 2008].

Thus, in the following, we describe a new preprocessor B + E which associates with a given CNF formula[1] Σ a CNF formula Φ which has the same number of models as Σ, but is at least as simple as Σ w.r.t. the number of variables and the size. B + E consists of two parts: B which aims at determining a *Bipartition* $\langle I, O\rangle$ of the variables of Σ such that every variable of O is defined in Σ in terms of the remaining variables (in I), and E which aims at *Eliminating* in Σ some variables of O. Our contribution mainly consists of the presentation of the algorithms B and E, a property establishing the correctness of our preprocessor, and some empirical results showing the computational improvements for model counting offered by B + E compared to pmc. The benchmarks used, the implementation (runtime code) of B + E, some detailed empirical results, and a full-proof version of the paper are available on line from www.cril.fr/KC/.

The rest of the paper is organized as follows. Section 2 gives some background on propositional definability. In Section 3 we introduce our preprocessor B + E and prove that it is correct. Section 4 presents results from our large scale experiments, showing B + E as a challenging preprocessor for model counting, especially when compared with pmc. Finally, Section 5 concludes the paper and lists some perspectives for further research.

2 On Definability

Let $\mathcal{L}$ be the (classical) propositional language defined inductively from a countable set $\mathcal{P}$ of propositional variables, the usual connectives ($\neg$, $\vee$, $\wedge$, $\leftrightarrow$, etc.) and including the Boolean constants $\top$ and $\bot$. Formulae are interpreted in the classical way. $\models$ denotes logical entailment and $\equiv$ logical equivalence. For any formula Σ from $\mathcal{L}$, $Var(\Sigma)$ is the set of variables from $\mathcal{P}$ occurring in Σ, and $\|\Sigma\|$ is the number of models of Σ over $Var(\Sigma)$. A literal ℓ is a variable $\ell = x$ from $\mathcal{P}$ or a negated one $\ell = \neg x$. When ℓ is a literal, $var(\ell)$ denotes the variable upon which ℓ is built. A term is a conjunction of literals or $\top$, and a clause is a disjunction of literals or $\bot$. A CNF formula is a conjunction of clauses. Let X be any subset of $\mathcal{P}$. A canonical term γ_X over X is a consistent term into which every variable from X appears (either as a positive literal or as a negative one, i.e., as a negated variable). $\exists X.\Sigma$ denotes any formula from $\mathcal{L}$ equivalent to the forgetting of X in Σ, i.e., the strongest logical consequence of Σ which is independent of variables from X.

Let us now recall the two (equivalent) forms under which the concept of definability in propositional logic can be encountered:

Definition 1 (implicit definability) *Let $\Sigma \in \mathcal{L}$, $X \subseteq \mathcal{P}$ and $y \in \mathcal{P}$. Σ* implicitly defines *y in terms of X if and only if for every canonical term γ_X over X, we have $\gamma_X \wedge \Sigma \models y$ or $\gamma_X \wedge \Sigma \models \neg y$.*

Definition 2 (explicit definability) *Let $\Sigma \in \mathcal{L}$, $X \subseteq \mathcal{P}$ and $y \in \mathcal{P}$. Σ* explicitly defines *y in terms of X if and only if there exists a formula $\Phi_X \in PROP_X$ s.t. $\Sigma \models \Phi_X \leftrightarrow y$. In such a case, Φ_X is called a* definition (or gate) *of y on X in Σ, y is the output variable of the gate, and X are its input variables.*

Example 1 *Let Σ be the CNF formula consisting of the following clauses:*

$a \vee b$, $a \vee c \vee \neg e$, $a \vee \neg d$, $b \vee c \vee \neg d$, $\neg a \vee \neg b \vee d$, $\neg a \vee \neg c \vee d$, $\neg a \vee \neg b \vee c \vee \neg e$, $\neg a \vee b \vee \neg c \vee \neg e$, $a \vee e$, $b \vee c \vee e$, $\neg b \vee \neg c \vee e$.

d and e are implicitly defined in Σ in terms of $X = \{a, b, c\}$. For instance, the canonical term $\gamma_X = a \wedge b \wedge \neg c$ is such that $\gamma_X \wedge \Sigma \models d \wedge \neg e$. On the other hand, $\gamma'_X = \neg a \wedge \neg b \wedge \neg c$ is such that $\gamma'_X \wedge \Sigma$ is inconsistent. d *and* e *are also explicitly defined in Σ in terms of $X = \{a, b, c\}$ since Σ implies*

$$d \leftrightarrow (a \wedge (b \vee c)) \text{ and } e \leftrightarrow (\neg a \vee (b \leftrightarrow c)).$$

What happens in this example is not fortuitous due to the following theorem from [Beth, 1953]:

Theorem 1 *Let $\Sigma \in \mathcal{L}$, $X \subseteq \mathcal{P}$ and $y \in \mathcal{P}$. Σ implicitly defines y in terms of X if and only if Σ explicitly defines y in terms of X.*

Since implicit definability and explicit definability coincide, one can simply say that y is defined in terms of X in Σ. An interesting consequence of this theorem is that it is not mandatory to point out a definition Φ_X of y in terms of X in order to prove that such a definition exists. Indeed, it is enough to show that Σ implicitly defines y in terms of X to do the job, and this problem is "only" coNP-complete [Lang and Marquis, 2008]. To prove it, we can take advantage of the following result (Padoa's theorem [Padoa, 1903]), restricted to propositional logic and recalled in [Lang and Marquis, 2008]; this theorem gives an entailment-based characterization of (implicit) definability:

Theorem 2 *For any $\Sigma \in \mathcal{L}$ and any $X \subseteq \mathcal{P}$, let Σ'_X be the formula obtained by replacing in Σ in a uniform way every propositional symbol z from $Var(\Sigma) \setminus X$ by a new propositional symbol z'. Let $y \in \mathcal{P}$. If $y \notin X$, then Σ (implicitly) defines y in terms of X if and only if $\Sigma \wedge \Sigma'_X \wedge y \wedge \neg y'$ is inconsistent.*[2]

3 A New Preprocessor to Model Counting

Instead of detecting gates and replacing them in Σ in order to remove output variables, our preprocessing technique consists in detecting output variables, then in forgetting them in Σ. To be more precise, the objective is first to find (if possible) a definability bipartition $\langle I, O\rangle$ of Σ where I contains as few elements as possible.

[1] Requiring the input to be a CNF formula is not a major restriction since Tseitin transformation [Tseitin, 1968] can be used to turn any propositional circuit into a CNF formula which has the same number of models – indeed, the variables which are introduced are actually defined from the original ones and the transformation consists in adding gates to the input. Interestingly, the CNF format is the one considered by state-of-the-art model counters.

[2] Obviously enough, in the remaining case when $y \in X$, Σ defines y in terms of X.

Definition 3 (definability bipartition) *Let $\Sigma \in \mathcal{L}$. A* definability bipartition *of Σ is a pair $\langle I, O\rangle$ such that $I \cup O = Var(\Sigma)$, $I \cap O = \emptyset$, and Σ defines every variable $o \in O$ in terms of I.*

Then in a second step, variables from O are forgotten in Σ so as to simplify it. This leads to the preprocessing algorithm B + E (*B(ipartition), then E(liminate)*) given at Algorithm 1:

Algorithm 1: B + E

input : a CNF formula Σ
output: a CNF formula Φ such that $\|\Phi\| = \|\Sigma\|$
1 $O \leftarrow B(\Sigma)$;
2 $\Phi \leftarrow E(O, \Sigma)$;
3 **return** Φ

Interestingly, both steps in this algorithm can be tuned in order to keep the preprocessing phase light from a computational standpoint. On the one hand, it is not necessary to determine a definability bipartition $\langle I, O\rangle$ of Σ for which the cardinality of I is minimal (identifying a reasonable amount of output variables can prove sufficient). On the other hand, it is not necessary to forget (i.e., eliminate) in Σ every variable from O but focusing on a subset $E \subseteq O$ is enough. Formally, our approach is based on the following result, which establishes the correctness of B + E:

Proposition 1 *Let $\Sigma \in \mathcal{L}$. Let $\langle I, O\rangle$ be a definability bipartition of $Var(\Sigma)$. Let $E \subseteq O$. Then $\|\Sigma\| = \|\exists E.\Sigma\|$.*

The ability to identify only a subset O of output variables in the bipartition generation phase, and to consider only a subset E of O in the elimination phase is valuable. In fact, computing a shortest base (i.e., a subset I of minimal cardinality such that every variable not in I is definable in Σ in terms of I) [Lang and Marquis, 2008] would be prohibitive; indeed, computing such a base using a branch-and-bound algorithm would require, in the worst case, exponentially many definability tests in the number of variables occurring in Σ. Furthermore, while forgetting variables in Σ obviously leads to diminishing the number of variables occurring in it, it may also lead to an exponential increase of its size. Eliminating in Σ only a subset E of variables from those found in O renders it possible to focus on those variables for which the elimination step will not increase the size of Σ (à la NiVER [Subbarayan and Pradhan, 2004]), or only by a negligible factor. More generally, the elimination of an output variable can be committed only if the size of Σ after the elimination step remains small enough, once some additional preprocessing has been achieved. Among the equivalence-preserving preprocessings of interest are occurrence simplification [Lynce and Marques-Silva, 2003] and vivification [Piette *et al.*, 2008] (already considered in [Lagniez and Marquis, 2014]), which aim at shortening some clauses, and at removing some clauses (for vivification).

Example 2 (Example 1 cont'ed) *No literal equivalences, AND/OR gates or XOR gates are logical consequences of Σ. Nevertheless, since Σ implies*

$$d \leftrightarrow (a \wedge (b \vee c)) \text{ and } e \leftrightarrow (\neg a \vee (b \leftrightarrow c))$$

a definability bipartition of $Var(\Sigma)$ is $\langle\{a, b, c\}, \{d, e\}\rangle$. Now, forgetting d and e in Σ leads to the generation of two non-valid clauses $a \vee c$ and $a \vee b \vee c$ so that $\exists\{d, e\}.\Sigma$ can then be computed as the conjunction of:

$$a \vee b, \qquad a \vee c, \qquad a \vee b \vee c.$$

which can be simplified further into $(a \vee b) \wedge (a \vee c)$. This CNF formula has only 5 models, hence this is also the case of Σ.

Algorithm 2: B

input : a CNF formula Σ
output: a set O of output variables, i.e., variables defined in Σ in terms of $I = Var(\Sigma) \setminus O$
1 $\langle \Sigma, O\rangle \leftarrow$ backbone(Σ);
2 $V \leftarrow$ sort($Var(\Sigma)$);
3 $I \leftarrow \emptyset$;
4 **foreach** $x \in V$ **do**
5 **if** defined?$(x, \Sigma, I \cup$ succ$(x, V),$ max#C$)$ **then**
6 $O \leftarrow O \cup \{x\}$;
7 **else**
8 $I \leftarrow I \cup \{x\}$;
9 **return** O

Algorithm 2 shows how a bipartition $\langle I, O\rangle$ of $Var(\Sigma)$ is computed by B in a greedy fashion. At line 1, backbone(Σ) computes the backbone of Σ (i.e., the set of all literals implied by Σ), and initializes O with the corresponding variables (indeed, a literal ℓ belongs to the backbone of Σ precisely when $var(\ell)$ is defined in Σ in terms of $\emptyset$). Boolean Constraint Propagation is also done on Σ completed by its backbone (this typically leads to simplifying Σ). While the variables of the backbone can be simplified away in Σ by fixing their values, they are nevertheless kept in O in order to ensure that the set O of variables returned by B is such that $\{I, O\}$ is a bipartition of $Var(\Sigma)$. At line 2, the remaining variables occurring in Σ are sorted by considering their number of occurrences from less to more frequent. At line 4, defined? takes advantage of Padoa's method (Theorem 2) for determining whether x is defined in Σ in terms of $I \cup$ succ(x, V), where succ(x, V) is the set of all variables of V which appear after x in V. defined? takes advantage of an anytime SAT solver solve based on CDCL architecture for achieving the (un)satisfiability test required by Padoa's method. In our implementation, the input of solve is the CNF formula $\Sigma \wedge \Sigma'_\emptyset \wedge \bigwedge_{z \in Var(\Sigma)}((\neg s_z \vee \neg z \vee z') \wedge (\neg s_z \vee z \vee \neg z'))$, completed by *assumptions*: for every z belonging to $I \cup$ succ(x, V), the unit clause s_z associated with z is added as an assumption to the CNF formula (its effect is to make z equivalent to its copy z'); then, x and $\neg x'$ are also added as assumptions. Interestingly, clauses which are learnt at each call to solve are kept for the subsequent calls. defined? is parameterized by max#C which bounds the number of clauses which can be learnt. When no contradiction has been found before max#C is reached, defined? returns false (i.e., x is considered as not defined in Σ in terms of $I \cup$ succ(x, V), while this could

be questioned had a larger bound be considered). Clearly, the number of output variables found by B is not guaranteed to be maximal, but this is on purpose for the sake of efficiency (observe that the number of calls to solve does not exceed the number of variables occurring in Σ).

Algorithm 3: E

input : a CNF formula Σ and a set of output variables $O \subseteq Var(\Sigma)$
output: a CNF formula Φ such that $\Phi \equiv \exists E.\Sigma$ for some $E \subseteq O$

```
1  Φ←Σ;
2  iterate←true; P←O;
3  while iterate do
4      E←P; P←∅; iterate←false;
5      Φ←vivificationSimpl(Φ, E);
6      while E ≠ ∅ do
7          x←select(E, Φ);
8          E←E \ {x};
9          Φ←occurrenceSimpl(Φ, x);
10         if #(Φ_x) × #(Φ_¬x) > max#Res then
11             P←P ∪ {x}
12         else
13             R←removeSub(Res(x, Φ), Φ);
14             if #((Φ \ Φ_x,¬x) ∪ R) ≤ #(Φ) then
15                 Φ←(Φ \ Φ_x,¬x) ∪ R;
16                 iterate←true;
17             else
18                 P←P ∪ {x}
19 return Φ
```

Algorithm 3 shows how variables from O are eliminated in Σ by E. P contains variables x from O which are candidates for elimination. P is initialized with the full set O (line 2). The main loop at line 3 is repeated while the elimination of at least one variable is effective (line 16). At line 4, the set E of variables that will be tentatively eliminated during the iteration is initialized with P, and P is reset to $\emptyset$. At line 5, the clauses of Φ are successively vivified using a slight variant of the vivification algorithm vivificationSimpl reported in [Lagniez and Marquis, 2014]. Vivification [Piette *et al.*, 2008] is a preprocessing technique which aims at reducing the input CNF formula, i.e., to remove some clauses in it and some literals in the other clauses while preserving equivalence, using Boolean Constraint Propagation. The additional parameter E is used to sort the literals within the clauses of Σ so that the literals over E are put first (i.e., one tries to eliminate occurrences of literals over E in priority). At line 6, one enters into the inner loop that operates while there are remaining variables in E. At line 7, a variable x is selected in E for being possibly eliminated by counting the number $\#(\Phi_x)$ of clauses of Φ where x appears as a positive literal, and the number $\#(\Phi_{\neg x})$ of clauses of Φ where $\neg$x appears as a negative literal; x is retained if it minimizes $\#(\Phi_x) \times \#(\Phi_{\neg x})$, which is a upper bound of the number of resolvents that the elimination of x in Φ may generate. At line 8, x is removed from E. Then, at line 9, one tries first to eliminate in Φ some occurrences of variable x using occurrenceSimpl. occurrenceSimpl is a restriction of the algorithm for occurrence simplification reported in [Lagniez and Marquis, 2014], where instead of considering the whole set of literals occurring in Φ, we just focus on those in $\{x, \neg x\}$. At line 10, one recomputes $\#(\Phi_x) \times \#(\Phi_{\neg x})$ and checks whether it exceeds or not a preset bound max#Res. If this is the case, then we possibly postpone the elimination of x in Φ at the next iteration by adding it to P (line 11). Otherwise, we compute the set $\text{Res}(x, \Phi)$ of all non-valid resolvents of clauses from Φ on x and we remove from it using removeSub every clause which is properly subsumed by a clause of Φ or another clause from $\text{Res}(x, \Phi)$; the resulting set of clauses is R (line 13). At line 14, we test whether the elimination of x in Φ, obtained by removing from Φ its subset $\Phi_{x,\neg x}$ of the clauses into which variable x occurs (either as a positive literal or as a negative literal), and adding the resolvents from R, leads or not to increasing the number of clauses in Φ. If so, we possibly postpone the elimination of x in Φ at the next iteration by adding it to P (line 18). If not, the elimination of x in Φ is committed (line 15). Clearly, it can be the case that some variables of O are not eliminated by E, but again, this is on purpose for efficiency reasons.

4 Empirical Results

In our experiments, we have considered 703 CNF instances from the SAT LIBrary.[3] They are gathered into 8 data sets, as follows: BN (Bayesian networks) (192), BMC (Bounded Model Checking) (18), Circuit (41), Configuration (35), Handmade (58), Planning (248), Random (104), Qif (7) (Quantitative Information Flow analysis - security). Our experiments have been conducted on Intel Xeon E5-2643 (3.30 GHz) processors with 32 GiB RAM on Linux CentOS. A time-out of 1h and a memory-out of 7.6 GiB has been considered for each instance. We set max#Res to 500.

As a matter of comparison, we have considered the pmc preprocessor for model counting, described in [Lagniez and Marquis, 2014] and available from www.cril.fr/KC/. To be more precise, we considered pmc equipped with the *#eq* combination of preprocessings, which combines backbone simplification, occurrence elimination, vivification and gates detection and replacement. pmc equipped with *#eq* proved empirically as a very efficient preprocessor for model counting [Lagniez and Marquis, 2014].

We evaluated the impact of B + E (for several values of max#C) by coupling it with exact model counters. We considered the search-based model counters Cachet[4] [Sang *et al.*, 2004] and SharpSAT[5] [Thurley, 2006], run with their default settings. Though compilation-based approaches do much more than model counting (since they compute equivalent, compiled representations of the input CNF formula Σ and not only the number of models of Σ), some of them appear as competitive for the model counting purpose. Thus, we

[3] www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html

[4] www.cs.rochester.edu/~kautz/Cachet/

[5] sites.google.com/site/marcthurley/sharpsat

also took advantage of the C2D compiler [6] [Darwiche, 2001; 2004] for achieving the downstream model counting task. C2D generates a Decision-DNNF representation Σ^* of Σ. The size of Σ^* is exponential in the size of Σ in the worst case, but the number of models of Σ conditioned by any consistent term γ can be computed efficiently from Σ^* in every case. And when γ is $\top$, one gets the number of models of Σ. C2D has been invoked with the following options `-count -in_memory -smooth_all`, which are suited when C2D is used as a model counter.

By the way, it is worth noting that B + E cannot be considered upstream to compilation-based approaches to model counting, while preserving the possibility of counting efficiently the number of models of the input conditioned by *any* consistent term. Indeed, when B + E(Σ) is not equivalent to Σ, the Decision-DNNF representation (B + E(Σ))* computed by C2D is not equivalent to Σ^*. Therefore, the possibility of efficient model counting after any conditioning is lost, but general conditioning must be downsized to a restricted form of conditioning where terms γ built up from I are allowed, but no other terms. Interestingly, such a restricted form of conditioning can prove enough in some scenarios. Especially, when the set of variables of Σ can be partitioned into a set of controllable variables (those which may require to be conditioned) and a remaining set of uncontrollable variables, one may take advantage of a slight variant of B + E ensuring that every controllable variable is put into I in order to simplify the input CNF formula Σ before compiling it.

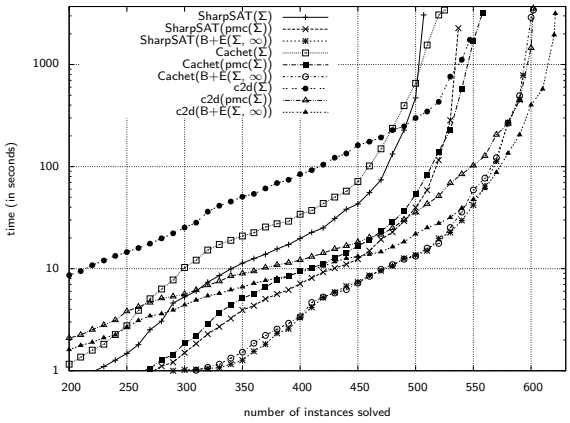
The next table makes precise the number of instances (over 703) solved within 1h by each of the model counters Cachet, SharpSAT, and C2D (first column), when no preprocessing has been applied (second column), pmc (equipped with *#eq*) has been applied first (third column), and finally B + E(Σ) for several values of max#C has been applied first (the remaining columns). The preprocessing time is taken into account in the computations (it is part of the 1h CPU time allocated per instance).

model counter	no preprocessing	pmc	10	100	1000	∞
Cachet	525	558	586	588	594	602
SharpSAT	507	537	575	581	586	593
C2D	547	602	605	613	616	621

The results reported in this table show the benefits which can be achieved by applying B + E before using a model counter. In particular, B + E leads to better performances than pmc. Since the best performances of B + E are achieved for max#C = ∞, we focus on this parameter assignment in the following.

The cactus plot given in the next figure illustrates the performances of Cachet, SharpSAT, and C2D, possibly empowered by pmc or by B + E. For each value t on the y-axis (a model counting time, in seconds) and each dot of a curve for which this value is reached on the y-axis, the corresponding value on the x-axis makes precise how many instances have been solved by the approach associated with the curve within a time limit of t (which includes the preprocessing time, when

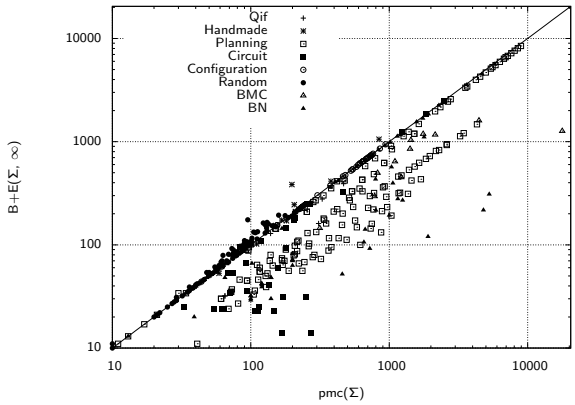
[6] reasoning.cs.ucla.edu/c2d/



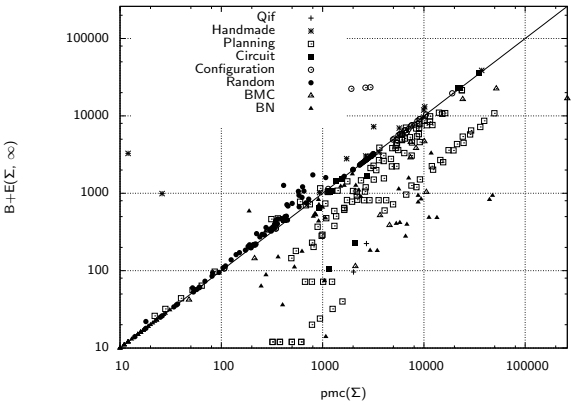
a preprocessing has been used). For the sake of readibility, only 10% of the dots have been printed. Again, the plot clearly shows B + E as a better preprocessor than pmc.

In order to determine how much applying B + E leads to reduction of the input CNF formula Σ compared to pmc, we considered two measures for assessing the reduction of Σ: #var(Σ), the number of variables of Σ, and #lit(Σ), the number of literals occurring in Σ (i.e., the size of Σ). Empirically, the results are presented on the two scatter plots (a) and (b) where each point corresponds to an instance Σ, its x-coordinate corresponds to the value of the measure (#var (a) or #lit (b)) on pmc(Σ) equipped with the *#eq* combination of preprocessings, while its y-coordinate corresponds to the value of the same measure on B + E(Σ) (with max#Conflicts = ∞). The scales used for both coordinates are logarithmic ones. Clearly enough, B + E often leads to much larger reductions than pmc for both measures. The benefits appear as very significant for instances from the Planning family.

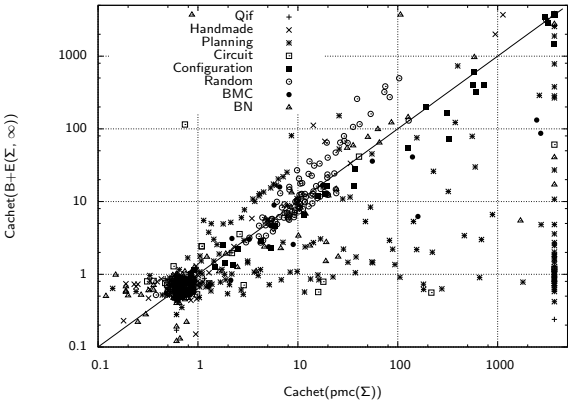
Finally, we have evaluated how much B + E leads to reduction of the overall model counting time compared to pmc. The results are presented on the two scatter plots (with logarithmic scales) (c) and (d), for the two model counters Cachet and C2D (which appeared as the best counters in our experiments) considered downstream. Each point corresponds to an instance Σ, its x-coordinate corresponds to the time (in seconds) required to compute $\|\Sigma\|$ by computing pmc(Σ) first, then calling the model counter on the resulting CNF formula, while its y-coordinate corresponds to the time required to compute $\|\Sigma\|$ by computing B + E(Σ) (with max#Conflicts= ∞) first, then calling the model counter on the resulting CNF formula. Again, whatever the downstream model counter, B + E appears often as a more efficient preprocessor than pmc. The rightmost parts of the two scatter plots cohere with the results reported in the previous table, showing a number of instances which can be solved by any of the model counters when B + E has been applied first, while they cannot be solved within the time limit of 1h when pmc is used instead. Finally, note that considering only the preprocessing times (and not the overall time needed to count the number of models of the input) for evaluating the preprocessor would be misleading: for some instances, the preprocess-



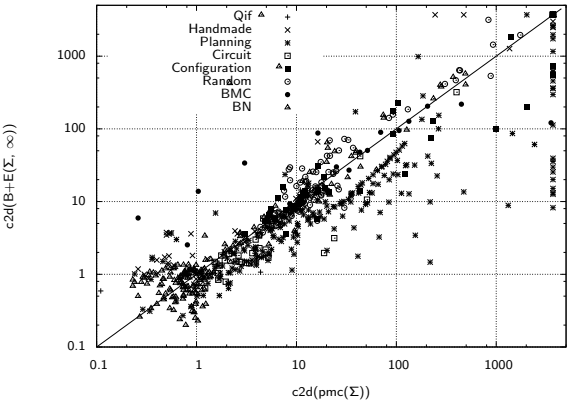
(a) #var



(b) #lit



(c) B + E+Cachet vs. pmc+Cachet



(d) B + E+C2D vs. pmc+C2D

ing times can be (relatively) long (details are available from www.cril.fr/KC/), just because the preprocessor does almost all the job (it may happen that the simplification of the instance is so important that the downstream model counter has almost nothing to do afterwards).

5 Conclusion

We have defined a new preprocessing technique B + E which associates with a given CNF formula Σ a CNF formula B + E(Σ) which has the same number of models as Σ, but is often simpler w.r.t. the number of variables and the size. B + E is based on standard theorems in classical logic by Beth and Padoa. Remarkably enough, while those results are quite old, they prove useful for defining a very effective preprocessing technique to model counting. Thus, experiments have shown that for many instances Σ, the overall computation time needed to calculate $\|$B + E(Σ)$\|$ using state-of-the art exact model counters is often much lower than the time needed to compute $\|\Sigma\|$ with the same counters.

This work opens a number of perspectives for further research. Considering other heuristics in B for determining a bipartition of the variables and determining how to tune the constants max#C and max#Res depending on the instance at hand will be studied in the future. Other perspectives concern the notion of projected model counting, as considered in [Aziz *et al.*, 2015]. The purpose is to compute $\|\exists E.\Sigma\|$ given a set E of variables and a formula Σ. Instead of taking advantage of B + E followed by any model counter to compute $\|\Sigma\|$, we could instead use B followed by any projected model counter (where the projection is onto I). The other way around, we could also exploit E on E and Σ as a preprocessor for projected model counters. It would be interesting to implement both approaches and to determine whether they are helpful in practice.

Acknowledgments

We would like to thank the anonymous reviewers for their attentive reading, useful comments and advices.

References

[Apsel and Brafman, 2012] U. Apsel and R. I. Brafman. Lifted MEU by weighted model counting. In *Proc. of AAAI'12*, 2012.

[Audemard and Simon, 2009] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solver. In *Proc. of IJCAI'09*, pages 399–404, 2009.

[Aziz *et al.*, 2015] R. A. Aziz, G. Chu, Ch. J. Muise, and P. J. Stuckey. #∃sat: Projected model counting. In *Proc. of SAT'15*, pages 121–137, 2015.

[Bacchus and Winter, 2004] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proc. of SAT'04*, pages 341–355, 2004.

[Bacchus *et al.*, 2003] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #sat and Bayesian inference. In *Proc. of FOCS'03*, pages 340–351, 2003.

[Beth, 1953] E.W. Beth. On Padoa's method in the theory of definition. *Indagationes mathematicae*, 15:330–339, 1953.

[Biere, 2014] A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In *Proc. of POS'14*, page 88, 2014.

[Chavira and Darwiche, 2008] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[Choi *et al.*, 2013] A. Choi, D. Kisa, and A. Darwiche. Compiling probabilistic graphical models using sentential decision diagrams. In *Proc. of ECSQARU'13*, pages 121–132, 2013.

[Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *Journal of the Association for Computing Machinery*, 48(4):608–647, 2001.

[Darwiche, 2004] A. Darwiche. New advances in compiling cnf into decomposable negation normal form. In *Proc. of ECAI'04*, pages 328–332, 2004.

[Domshlak and Hoffmann, 2006] C. Domshlak and J. Hoffmann. Fast probabilistic planning through weighted model counting. In *Proc. of ICAPS'06*, pages 243–252, 2006.

[Een and Biere, 2005] N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of SAT'05*, pages 61–75, 2005.

[Feiten *et al.*, 2012] L. Feiten, M. Sauer, T. Schubert, A. Czutro, E. Böhl, I. Polian, and B. Becker. #SAT-based vulnerability analysis of security components - A case study. In *Proc. of DFT'12*, pages 49–54, 2012.

[Gomes *et al.*, 2009] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, pages 633–654. 2009.

[Han and Somenzi, 2007] H. Han and F. Somenzi. Alembic: An efficient algorithm for cnf preprocessing. In *Proc. of DAC'07*, pages 582–587, 2007.

[Heule *et al.*, 2010] M. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for cnf formulas. In *Proc. of LPAR'10*, pages 357–371, 2010.

[Heule *et al.*, 2011] M. Heule, M. Järvisalo, and A. Biere. Efficient cnf simplification based on binary implication graphs. In *Proc. of SAT'11*, pages 201–215, 2011.

[Järvisalo *et al.*, 2012] M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on cnf. *Journal of Automated Reasoning*, 49(4):583–619, 2012.

[Lagniez and Marquis, 2014] J.-M. Lagniez and P. Marquis. Preprocessing for propositional model counting. In *Proc. of AAAI'14*, pages 2688–2694, 2014.

[Lang and Marquis, 2008] J. Lang and P. Marquis. On propositional definability. *Artificial Intelligence*, 172(8-9):991–1017, 2008.

[Lynce and Marques-Silva, 2003] I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proc. of ICTAI'03*, pages 105–110, 2003.

[Manthey, 2012a] N. Manthey. Coprocessor 2.0 - A flexible CNF simplifier - (tool presentation). In *Proc. of SAT'12*, pages 436–441, 2012.

[Manthey, 2012b] N. Manthey. Solver description of RISS 2.0 and PRISS 2.0. Technical report, TU Dresden, Knowledge Representation and Reasoning, 2012.

[Padoa, 1903] A. Padoa. Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une théorie déductive quelconque. In *Bibliothèque du Congrès International de Philosophie*, pages 309–365. Paris, 1903.

[Palacios *et al.*, 2005] H. Palacios, B. Bonet, A. Darwiche, and H. Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proc. of ICAPS'05*, pages 141–150, 2005.

[Piette *et al.*, 2008] C. Piette, Y. Hamadi, and L. Saïs. Vivifying propositional clausal formulae. In *Proc. of ECAI'08*, pages 525–529, 2008.

[Samer and Szeider, 2010] M. Samer and S. Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, pages 475–482, 2005.

[Subbarayan and Pradhan, 2004] S. Subbarayan and D.K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *Proc. of SAT'04*, pages 276–291, 2004.

[Thurley, 2006] M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. of SAT'06*, pages 424–429, 2006.

[Tseitin, 1968] G.S. Tseitin. *On the complexity of derivation in propositional calculus*, chapter Structures in Constructive Mathematics and Mathematical Logic, pages 115–125. Steklov Mathematical Institute, 1968.

Chapitre 7

An Improved `Decision-DNNF` Compiler

Article paru dans les actes de « *the 26th International Joint Conference on Artificial Intelligence* » (IJCAI'17), pages 667 – 673, août 2017.

Co-écrit avec Pierre Marquis.

An Improved Decision-DNNF Compiler

Jean-Marie Lagniez and **Pierre Marquis**
CRIL, U. Artois & CNRS, F-62300 Lens, France,
e-mail: {lagniez, marquis}@cril.univ-artois.fr

Abstract

We present and evaluate a new compiler, called D4, targeting the Decision-DNNF language. As the state-of-the-art compilers C2D and Dsharp targeting the same language, D4 is a top-down tree-search algorithm exploring the space of propositional interpretations. D4 is based on the same ingredients as those considered in C2D and Dsharp (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching). D4 takes advantage of a dynamic decomposition approach based on hypergraph partitioning, used sparingly. Some simplification rules are also used to minimize the time spent in the partitioning steps and to promote the quality of the decompositions. Experiments show that the compilation times and the sizes of the Decision-DNNF representations computed by D4 are in many cases significantly lower than the ones obtained by C2D and Dsharp.

1 Introduction

Knowledge compilation (KC) is acknowledged as a challenging approach for circumventing the intractability of many practical reasoning problems based on propositional representations (see [Cadoli and Donini, 1998; Darwiche, 2014; Marquis, 2015] for surveys). Since its very beginning, research on KC has been developed following a number of directions, including the identification of the compilability of some problems, the definition and the study of new target languages for KC, the design and the evaluation of compilers, and the use of the KC languages, compilers and reasoners in various applications.

As the previous works [Darwiche, 2004; Muise *et al.*, 2012; Oztok and Darwiche, 2014; 2015a], we are concerned with the third research direction in this paper: our objective is to describe a new compiler, called D4.[1] D4 is a top-down compiler associating with every input CNF formula an equivalent Decision-DNNF representation. Like the state-of-the-art compilers C2D and Dsharp targeting the same language, D4 is a tree-search algorithm exploring the whole space of propositional interpretations. Accordingly, D4 takes advantage of the techniques used in C2D and Dsharp for efficiency reasons (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching). However, the decomposition heuristics used for guiding the exploration of the search space in D4 differs from the ones considered in C2D and in Dsharp, while trying to keep the best of them. Thus, like Dsharp, D4 computes a *dynamic* decomposition of the input CNF formula under the current partial assignment (disjoint components are not computed prior to the search as with C2D). However, like C2D, D4 partitions the dual hypergraph associated with the input CNF formula under the current partial assignment to find a decomposition (while the branching heuristics of Dsharp is based on the VSADS score of the variables). What is brand new in D4 is that the decomposition is achieved in a parsimonious way (hypergraph partitioning is not used at each decision step) and that some simplification rules are used to minimize the time spent in the partitioning steps and to promote the quality of the resulting decompositions. In order to assess D4 and to compare it with C2D and Dsharp, we performed experiments on many benchmarks, coming from several families. The results obtained clearly show the benefits offered by D4.

The rest of the paper is organized as follows. After some formal preliminaries (Section 2), our compiler D4 is presented in Section 3. An empirical evaluation is provided in Section 4, before the concluding section (Section 5). The runtime code of D4, the benchmarks used, and some additional empirical results are available on line from https://www.cril.univ-artois.fr/KC/.

2 Formal Preliminaries

We consider a propositional language $PROP_{PS}$ defined from a finite set PS of propositional symbols and the standard connectives. $PROP_{PS}$ is interpreted in a classical way. For every formula Σ in $PROP_{PS}$, $Var(\Sigma)$ is the set of propositional variables occurring in Σ. A CNF formula Σ is a conjunction of clauses, where a clause is a disjunction of literals. Every CNF is viewed as a set of clauses, and every clause is viewed as a set of literals. For every literal ℓ, $var(\ell)$ denotes the variable x of ℓ ($var(x) = x$ and $var(\neg x) = x$), and $\sim\ell$ denotes the complementary literal of ℓ (i.e., for every variable x, $\sim x = \neg x$ and $\sim\neg x = x$). The conditioning of a CNF for-

[1] D4: a **D**ecision-**D**NNF compiler based on **D**ynamic **D**ecomposition.

mula Σ by a literal $\ell = x$ (resp. $\ell = \neg x$) is the CNF formula $\Sigma \mid \ell$ obtained by removing from Σ every clause containing x (resp. $\neg x$) and removing from the remaining clauses every occurrence of $\neg x$ (resp. x).

The *dual hypergraph* of a CNF formula Σ is $DH(\Sigma) = (N_d, H_d)$ where $N_d = \Sigma$ and $H_d = \{\{\delta \in \Sigma \mid x \in Var(\delta)\} \mid x \in Var(\Sigma)\}$. The nodes of N_d correspond to the clauses of Σ and the hyperedges of H_d are labelled by sets of variables of Σ (for each variable x, the corresponding hyperedge is the set of clauses $Cls_\Sigma(x)$ of Σ containing x).

Example 1 *Let $\Sigma = (a \vee b) \wedge (a \vee \neg c) \wedge (a \vee \neg d) \wedge (b \vee \neg c) \wedge (b \vee \neg d)$. We have $DH(\Sigma) = (N_d, H_d)$, with $N_d = \{n_1, n_2, n_3, n_4, n_5\}$ (n_1 (resp. n_2, n_3, n_4, n_5) corresponds to $a \vee b$ (resp. $a \vee \neg c$, $a \vee \neg d$, $b \vee \neg c$, $b \vee \neg d$) and $H_d = \{\{n_1, n_2, n_3\}, \{n_1, n_4, n_5\}, \{n_2, n_4\}, \{n_3, n_5\}\}$ (labelled respectively by $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$).*

BCP denotes a Boolean Constraint Propagator [Zhang and Stickel, 1996; Moskewicz *et al.*, 2001], which is a key component of many solvers. BCP(Σ) returns $\{\emptyset\}$ if there exists a unit refutation from the clauses of the CNF formula Σ, and it returns the set of literals (unit clauses) which are derived from Σ using unit propagation in the remaining case. As a side effect, BCP "virtually" simplifies Σ by conditioning it by the unit clauses which are generated. For efficiency reasons, such a simplification is not "physically" performed on the CNF (instead the set of literals derived using unit propagation is maintained).

d-DNNF is the language consisting of the Boolean circuits with a single output (its root), where each input is a literal or a Boolean constant, and each internal gate is either a decomposable $\wedge$ gate of the form $N = \wedge(N_1, \ldots, N_k)$ ("decomposable" means here that for each $i, j \in \{1, \ldots, k\}$ with $i \neq j$ the subcircuits of N rooted at N_i and N_j do not share any common variable) or deterministic $\vee$ gate of the form $N = \vee(N_1, \ldots, N_k)$ ("deterministic" means here that for each $i, j \in \{1, \ldots, k\}$ with $i \neq j$ the subcircuits of N rooted at N_i and N_j are jointly inconsistent).

Decision-DNNF is defined in the same way, except that deterministic $\vee$ gates are replaced by decision gates of the form $N = ite(x, N_1, N_2)$. x is the decision variable at gate N, it does not occur in the subcircuits N_1, N_2, and ite is a ternary connective whose semantics is given by $ite(X, Y, Z) = (\neg X \wedge Y) \vee (X \wedge Z)$ ("ite" means "if ... then ... else ...: if X then Z else Y).

Example 2 (Example 1 cont'ed) *The Decision-DNNF representations given on Figure 1 are equivalent to the CNF formula Σ considered in Example 1 (the nodes labelled by $\wedge$ are decomposable $\wedge$ nodes and the circled nodes are decision nodes).*

Decision-DNNF representations, also known as decomposable decision graphs [Fargier and Marquis, 2006], can be turned in linear time into specific d-DNNF representations. Indeed, when one replaces in a Decision-DNNF representation a decision node of the form $N = ite(x, N_1, N_2)$ by $N = (\neg x \wedge N_1) \vee (x \wedge N_2)$, the two $\wedge$ nodes which are conveyed by the replacement are decomposable ones (x appears neither in N_1 nor in N_2) and the $\vee$ node is deterministic ($(\neg x \wedge N_1) \wedge (x \wedge N_2)$ is inconsistent).

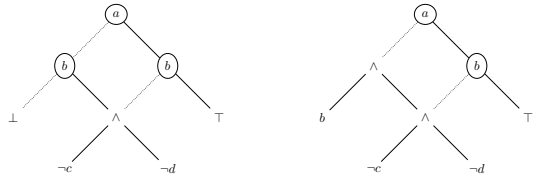


Figure 1: Two Decision-DNNF representations.

3 An Improved Decision-DNNF Compiler: D4

Our top-down compiler D4 associates with every input CNF formula Σ an equivalent Decision-DNNF representation. It elaborates over C2D and Dsharp.

In C2D (reasoning.cs.ucla.edu/c2d/) [Darwiche, 2001; 2004], the generation of the decision nodes in the resulting Decision-DNNF representation is guided by a decomposition tree (dtree) of the input CNF instance Σ, which is computed first. Both the time required to generate a Decision-DNNF representation of Σ and the size of the representation are linear in the size of Σ, yet exponential in the width of the dtree which is used.

In Dsharp (www.haz.ca/research/dsharp/) [Muise *et al.*, 2012], the generation of the decision nodes is not guided by a decomposition tree, but a dynamic decomposition approach is considered instead. During the search, the variable selected by the branching heuristics is one with a maximal Variable State Aware Decaying Sum (VSADS) score [Sang *et al.*, 2005], as in the (exact) model counters sharpSAT (sites.google.com/site/marcthurley/sharpsat) [Thurley, 2006] and Cachet (www.cs.rochester.edu/~kautz/Cachet/index.htm) [Sang *et al.*, 2004]. Implicit BCP is used at every decision point in order to find some failed literals, i.e., literals which are falsified in every model of the formula corresponding to the decision point. The literals which belong to clauses that became binary in the most recent call to BCP are selected as candidates for failed literals. A conflict clause is learned for each failed literal found. Dsharp also incorporates a restricted form of backbone detection, based on BCP.

As it is the case for the previous compilers C2D and Dsharp [Huang and Darwiche, 2007; Muise *et al.*, 2012], the Decision-DNNF representation of Σ which is computed by D4 corresponds to the trace of a solver. We used our own solver solve, which is based on the state-of-the-art SAT solver MiniSAT 2.2 [Eén and Sörensson, 2003]. solve is called at each decision point, and if a model is found, instead of stopping, the search backtracks up to the level of the last decision node which has been created. Assumptions are exploited in solve, so that the clauses which are learnt at each call to solve can be kept for the subsequent calls. D4 exploits the same techniques as the ones used in C2D and Dsharp for efficiency reasons (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching[2]). However, the decomposition heuristics used by

[2]In our cache implementation, following the approach already

D4 for guiding the exploration of the search space is different from the ones considered in C2D and Dsharp.

The pseudo-code of D4. Algorithm 1 provides the pseudo-code of the compiler D4. The compilation of a given CNF formula Σ is achieved by calling D4(Σ, $\emptyset$).

Algorithm 1: D4(Σ, LV)

```
   input  : a CNF formula Σ
   input  : a list of variables LV
   output : the root node N of a Decision-DNNF
            representation of Σ
1  S ← solve(Σ);
2  if S = {∅} then return leaf(⊥);
3  if Var(Σ) = ∅ then return aNode(S, [leaf(⊤)]);
4  if cache(Σ) ≠ nil then return aNode(S, [cache(Σ)]);
5  comps ← connectedComponents(Σ);
6  LN_d ← [];
7  foreach c ∈ comps do
8      LV_c ← restrict(LV, Var(c));
9      if LV_c = ∅ or
       #(Var(S) ∩ Var(c)) > (1/10)#(Var(c)) then
           LV_c ← sort(HGP(c));
10     v ← head(LV_c);
11     LV_c ← tail(LV_c);
12     N_d ← ite(v, D4(c|¬v, LV_c), D4(c|v, LV_c));
13     LN_d ← add(N_d, LN_d);
14 N_∧ ← aNode(S, LN_d);
15 cache(Σ) ← N_∧;
16 return N_∧;
```

At line 1, solve is called on Σ. S is equal to $\{\emptyset\}$ if Σ is inconsistent, and in this case a Decision-DNNF representation of it (leaf($\bot$)) is generated at line 2. Otherwise S is equal to BCP(Σ) (please keep in mind that BCP simplifies Σ). The second base case is addressed at line 3: if Σ (once simplified by BCP) contains no variable, then a $\wedge$ node is generated; aNode($\{\ell_1, \ldots, \ell_k\}, [N_1, \ldots, N_m]$) generates a $\wedge$ node with $k + m$ children: the k literals from the set $\{\ell_1, \ldots, \ell_k\}$ and the m nodes from the list $[N_1, \ldots, N_m]$. Hence aNode(S, [leaf($\top$)]) returns a $\wedge$ node gathering all the literals of S (completed by a $\top$ leaf for handling the case S is empty). At line 4, one first determines whether the current formula Σ has already been encountered or not during the search. One takes advantage of the cache function which associates CNF formulae with Decision-DNNF representations given by their root nodes. If Σ has already been found, then the algorithm simply returns the root node of the corresponding stored Decision-DNNF representation. At line 5 connectedComponents returns a set *comps* of CNF formulae, the connected components of Σ. The loop at line 7 considers every element c of *comps* successively. For a given c, the variables of LV are restricted to those in c. One then tests at line 9 whether the resulting list LV_c is empty or not, and if the number of literals obtained by applying BCP on Σ when projected onto the set of variables of the current component c is large enough (at least 10% of the number of variables of c). If at least one of the two conditions is satisfied, then one updates the current list LV_c of variables. HGP(c) computes a cutset of the dual hypergraph of c after applying to it some simplification rules. The variables of this cutset are sorted according to their VSADS score (the largest score first) to produce an updated list sort(HGP(c)). Then one selects the first variable v of LV_c (line 10) and remove it from LV_c (line 11). A decision node N_d with decision variable v is computed next, via two recursive calls to D4 corresponding to the two ways of conditioning v in c (line 12). N_d is then added to the list LN_d of nodes at line 13, which has been initialized to the empty list before the loop (line 6). Then a $\wedge$ node $N_\wedge$ gathering the literals of S and the decision nodes of LN_d is generated (line 14), and added to the cache (associated with the entry Σ) (line 15), and finally returned as the result of the main call (line 16).

Dynamic decomposition based on hypergraph partitioning. Like Dsharp, D4 computes a *dynamic decomposition* of the input formula under the current partial assignment (disjoint components are computed during the search and not prior to the search). However, like C2D -dt_method 1, D4 partitions the dual hypergraph associated with the current formula to find a decomposition. Basically, the HGP procedure takes advantage of the PaToH partitioner – Partitioning Tools for Hypergraph, v. 3.2 (http://bmi.osu.edu/~umit/software.html) [Catalyürek and Aykanat, 2011] to do the job. PaToH is similar to hMETIS (the partitioner exploited by C2D) but, unlike hMETIS, it runs on the 64 bit architecture used for our experiments. Given the dual hypergraph $DH(\Sigma) = (N_d, H_d)$ of Σ, PaToH roughly looks for a subset C of H_d containing as few elements as possible such that removing the cutset C from H_d leads to a hypergraph containing at least two disjoint components having sizes as close as possible (see [Catalyürek and Aykanat, 2011] for more details). When the variables corresponding to the elements of C are assigned (whatever the way they are assigned) it is guaranteed that the current formula conditioned by the corresponding assignment has at least two disjoint components, so that a decomposable $\wedge$ node can be generated in the compiled form.

When C2D -dt_method 1 is used, a dtree is computed entirely at start using hypergraph partitioning: a first cutset C of the dual hypergraph (N_d, H_d) of Σ is computed, and then one recursively looks for a decomposition of each of the two hypergraphs (the resulting connected components) obtained from the removal of C in (N_d, H_d), until a hypergraph with a single node is obtained. Contrastingly, the decomposition process followed by D4 is dynamic: for each assignment of the variables of corresponding to the hyperedges of C, the hypergraph associated with Σ conditioned by the assignment (and possibly simplified further thanks to BCP) is a candidate for a further decomposition. The variables of C are ordered according to their VSADS score (the ones with a maximal score being considered first as in Dsharp). That way, D4 tries to keep the best of the decomposition heuristics used by

used in our MDDG compiler [Koriche *et al.*, 2015], we do not store in the current bucket the clauses which have not been shortened.

the two compilers C2D and Dsharp.

Each of the static and dynamic decomposition approaches has some pros and some cons. On the one hand, computing a static decomposition as in C2D -dt_method 1 limits the number of calls to the hypergraph partitioner since there is no need to compute a cutset for each assignment. This property is not shared when a dynamic decomposition approach is considered, as in D4. However, limiting the number of calls to the hypergraph partitioner is important since hypergraph partitioning is time consuming. Especially, profiling the code of D4, we observed that in average for the instances considered in our experiments at least 50% of the computation time of D4 is consumed by PaToH (and for some instances the part of the time consumed by PaToH exceeds 90%). On the other hand, the static decomposition approach does not lead in general to the most efficient decompositions; indeed, for each cutset C, the chosen assignment of the variables corresponding to the hyperedges of C (and the additional propagations which can be made from it) can have a huge impact both on the size and the structure of the resulting formula once simplified, so that a different, and actually much better, decomposition can be reached for it, compared to the ones associated with the other assignments of the variables of C.

Example 3 (Example 1 cont'ed) *$C = \{\{n_1, n_2, n_3\}, \{n_1, n_4, n_5\}\}$ is a cutset of $DH(\Sigma)$: removing the elements of C from H_d leads to two disjoint hypergraphs since the remaining hyperedges $\{n_2, n_4\}$, $\{n_3, n_5\}$ do not share any node. $\{n_1, n_2, n_3\}$ (resp. $\{n_1, n_4, n_5\}$) are labelled by the sets of variables $\{a\}$ (resp. $\{b\}$), thus assigning those two variables a and b in Σ (whatever the assignment) leads to two disjoint components (one of them consists of the clause $b \vee \neg c$ simplified by the assignment and the other one of the clause $b \vee \neg d$ simplified by the assignment). This can be observed on Figure 1 (left). However, one can also observe on Figure 1 (right) that assigning* all *the variables corresponding to the hyperedges of C is not necessary to generate disjoint components: setting a to false is enough to get some disjoint components, independently of the way b will be assigned. Thus considering b (which corresponds to the remaining hyperedge of C) as the next branching variable when a has been set to false is not guaranteed to be the best choice. For each connected component resulting from the assignment of a to false, some new decompositions should be looked for.*

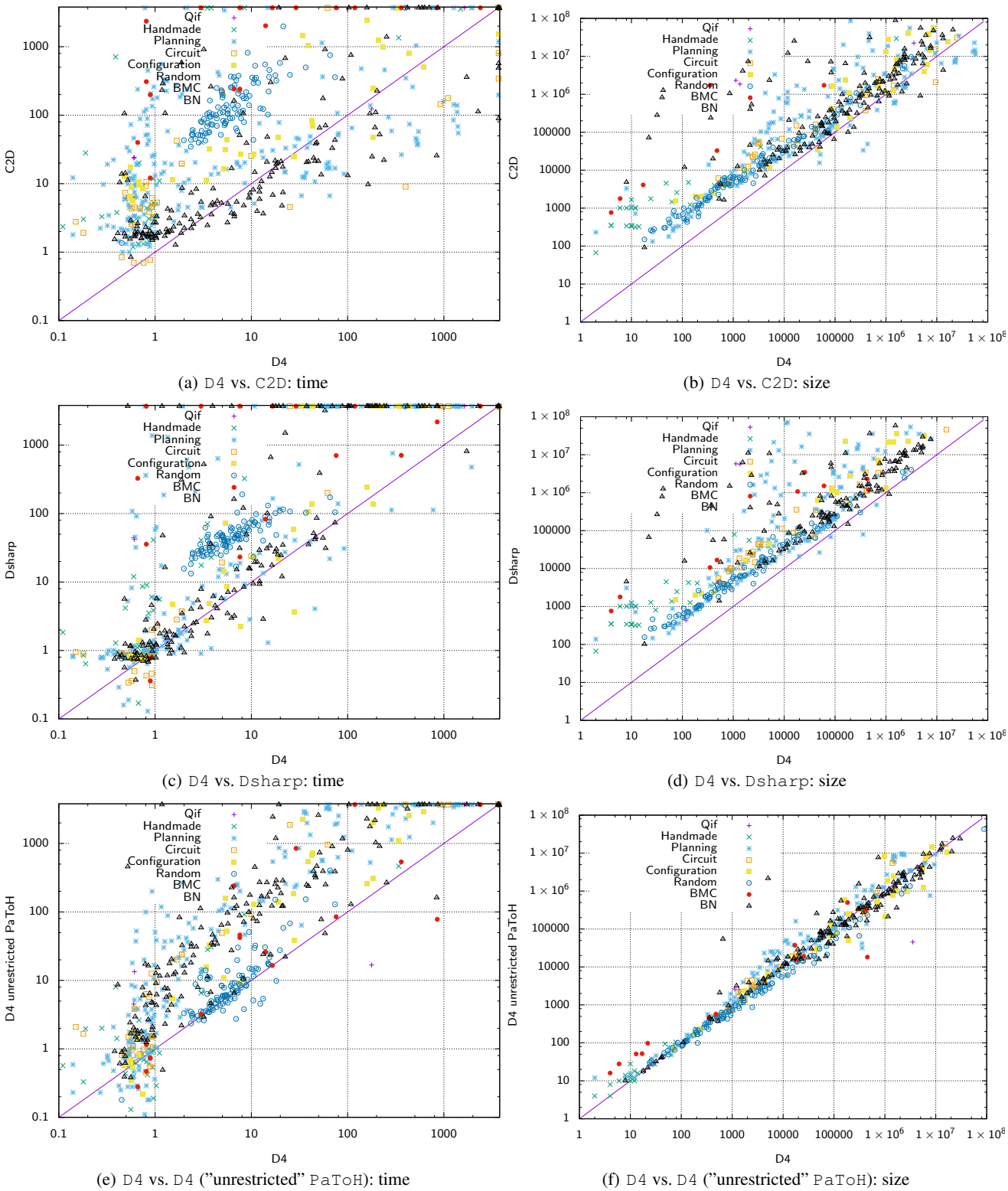
Improving the hypergraph partitioning steps. In order to circumvent the complexity of the hypergraph partitioning steps, the strategy used in D4 is twofold. On the one hand, we avoid calling HGP at each recursion step or each time a decision node must be generated. Thus, a new partition is computed only if the condition at line 9 is satisfied, i.e., when the current list of variables LV_c is empty, or when the number of propagations obtained via BCP (projected onto the set of variables of the current component c) is "large enough". On the other hand, we designed some specific rules which are used inside HGP and aim at simplifying the hypergraph (both in terms of the number of its nodes and in terms of the number of its hyperedges) associated with the current formula c before calling PaToH on it. The simplification achieved can also lead PaToH to find better decompositions. It guarantees that the CNF formula $\texttt{BCP}(c \mid \gamma_c)$ contains at least two disjoint components when γ_c is any total assignment of the variables from the list LV_c computed at line 9. One first exploits an algorithm for the detection of literal equivalences based on BCP. This algorithm, which is a by-product of the algorithm equivSimpl reported in [Lagniez and Marquis, 2014], is used for determining for each literal ℓ appearing in the current formula c a list of equivalent literals (its equivalence class). Those lists are generated by computing for each ℓ, $\texttt{BCP}(c \cup \{\ell\})$ and $\texttt{BCP}(c \cup \{\sim \ell\})$.[3] Once this has been done, for each $var(\ell)$, the variables $v_1^\ell, \ldots, v_k^\ell$ associated with the literals of the equivalence class of ℓ (but $var(\ell)$ itself) are suppressed from the labels of the hyperedges of $DH(c)$, the hyperedges without any label are then removed from this set, and the hyperedge $Cls_c(var(\ell))$ is replaced by $Cls_c(var(\ell)) \cup \bigcup_{i=1}^{k} Cls_c(v_i^\ell)$. When all $var(\ell)$ have been considered, two rules are applied on the resulting hypergraph in a systematic fashion in order to simplify it further. The simplification process stops when no rule can be applied or when the current hypergraph has a single hyperedge. The first rule concerns the nodes of this hypergraph. Suppose that the current set of nodes contains two distinct nodes n_i, n_j such that every hyperedge of the current set of hyperedges containing n_i also contains n_j. Then n_i can be safely removed from the current set of nodes, and from the hyperedges which contain it. The labels of the hyperedges containing n_i are marked and added to the labels of the hyperedges containing n_j. The second rule concerns the unmarked labels of hyperedges of size 1 which can obviously be removed.

Example 4 *Let $\Sigma = (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee a) \wedge (a \vee c \vee d)$. We have $DH(\Sigma) = (N_d, H_d)$, with $N_d = \{n_1, n_2, n_3, n_4\}$ (n_1 (resp. n_2, n_3, n_4) corresponds to $\neg a \vee b$ (resp. $\neg b \vee c$, $\neg c \vee a$, $a \vee c \vee d$) and $H_d = \{\{n_1, n_3, n_4\}, \{n_1, n_2\}, \{n_2, n_3, n_4\}, \{n_4\}\}$ (labelled respectively by $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$). The first step leads to identify the equivalence class of a as $\{a, b, c\}$. The resulting hypergraph is $(\{n_1, n_2, n_3, n_4\}, \{\{n_1, n_2, n_3, n_4\}, \{n_4\}\})$. The set of labels of $\{n_1, n_2, n_3, n_4\}$ is $\{a\}$ and the set of labels of $\{n_4\}$ is $\{d\}$. Then the nodes n_1, n_2, n_3 are removed, a is marked and added to the set of labels of $\{n_4\}$ which becomes $\{a, d\}$. Finally, label d is removed from this set. The resulting hypergraph is thus $(\{n_4\}, \{\{n_4\}\})$ where $\{n_4\}$ is labelled by $\{a\}$.*

4 Empirical Evaluation

We have considered 703 CNF instances from the SAT LIBrary www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html, gathered into 8 data sets, as follows: BN (Bayesian networks) (192), BMC (Bounded Model Checking) (18), Circuit (41), Configuration (35), Handmade (58), Planning (248), Random (104), Qif (7) (Quantitative Information Flow

[3] Running this algorithm is also useful for determining that some literals are fixed in c, while they are not necessarily detected as such when running BCP on c. Thus, if $\texttt{BCP}(c \cup \{\ell\}) = \{\emptyset\}$, then we have $c \models \neg\ell$; if $\ell' \in \texttt{BCP}(c \cup \{\ell\})$ and $\ell' \in \texttt{BCP}(c \cup \{\sim \ell\})$, then $c \models \ell'$. In D4, such literals are used to simplify c for the subsequent computations.



(a) D4 vs. C2D: time

(b) D4 vs. C2D: size

(c) D4 vs. Dsharp: time

(d) D4 vs. Dsharp: size

(e) D4 vs. D4 ("unrestricted" PaToH): time

(f) D4 vs. D4 ("unrestricted" PaToH): size

Figure 2: Comparing D4 with C2D, Dsharp, and a version of D4 where PaToH is applied at each decision node.

analysis - security). The experiments have been conducted on Intel Xeon E5-2643 (3.30 GHz) processors with 32 GiB RAM on Linux CentOS. A time-out of 1h and a memory-out of 7.6 GiB has been considered for each instance.

We have computed the compilation times and the sizes of the corresponding Decision-DNNF compiled forms obtained by using D4, C2D -dt_method 1, and Dsharp, on each instance. The results are reported on the scatter plots given in Figure 2. Each dot represents an instance; the time (in seconds) needed to solve it or the size (in number of edges) of the resulting compiled form, using the compiler corresponding to the x-axis (resp. y-axis), is given by its x-coordinate (resp. y-coordinate). Logarithmic scales are used for both coordinates. In part (a) and (b) of the figure, the x-axis corresponds to D4 while the y-axis corresponds to C2D. In part (a) compilation times are reported while in part (b), sizes of compiled forms are reported. Parts (c) and (d) present the corresponding results when Dsharp is used instead of C2D. In (d) the number of edges reported for Dsharp is the number of edges in the compressed representations.[4] The numbers of instances solved within the time and memory limits are 574 (over 703) for D4, 546 for C2D -dt_method 1, and 467 for Dsharp.

The experimental results obtained show that the compilation times of D4 are significantly lower than the ones obtained by C2D and Dsharp on many instances. More importantly, the sizes of the Decision-DNNF representations computed by D4 are substantially lower for the great majority of instances (sometimes by several orders of magnitude) than the ones obtained by C2D and Dsharp.

An important issue was to determine the very reasons of the efficiency of D4 compared to C2D and Dsharp. Indeed, beyond its specific decomposition heuristics, D4 is based on a more recent SAT solver than the ones considered in those two compilers. Does this explain the computational benefits achieved by D4? In order to answer this question, we developed a Dsharp-like compiler based on D4, and we evaluated this compiler on the same benchmarks set and considering the same resource bounds as above. It turns out that the Dsharp-like compiler has been able to solve 515 instances, which is much more than Dsharp, but far less than the 574 instances solved by D4. So it clearly appears that the gain offered by D4 is not solely due to the use of a solver based on MiniSAT (even if this helps). Assessing the impact of using PaToH (with default setting, as used in D4) instead of hMETIS was also an issue. Interestingly, a comparison of PaToH and hMETIS has already been done, see http://bmi.osu.edu/umit/PaToH/table1.html. It turns out PaToH and hMETIS have similar performances, especially as to the quality of the decompositions found. Similarly, in order to assess the improvements obtained by limiting the time consumed by PaToH by using it sparingly and after the application of simplification rules (as implemented in D4), we have also compared D4 with a version of it where PaToH is called in an unrestricted fashion (i.e., when the condition at line 9 is true) and no simplification rules are applied before calling PaToH. This version of D4 with "unrestricted" use of PaToH solved "only" 531 instances, and does not lead to size improvements. More detailed results are reported in parts (e) and (f) of Figure 2.

[4] Contrastingly, no reduction rules are applied to the Decision-DNNF representations computed by D4, so that the sizes which are reported correspond to uncompressed representations.

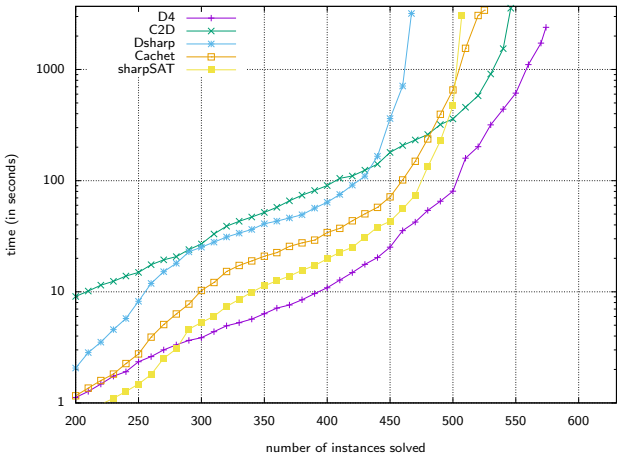


Figure 3: Number of instances solved by D4, C2D, Dsharp, Cachet, and sharpSAT in a given time.

Finally, we wanted to assess the performances of D4 as a model counter, and to compare it with miniC2D, Cachet and sharpSAT on the same benchmarks set and with the same resource bounds. miniC2D (http://reasoning.cs.ucla.edu/minic2d/) [Oztok and Darwiche, 2015b] is a compiler targeting the language Decision-SDD. While it is based on a more recent SAT solver than the ones used in C2D and Dsharp, it has been able to solve only 414 instances (thus, far less than C2D). Cachet (resp. sharpSAT) solved 525 (resp. 507) instances (over 703) within the time and memory limits. The cactus plots in Figure 3 give for D4, C2D, Dsharp, Cachet, and sharpSAT the number of instances solved in a given time. The results show that while D4 (as a compiler) requires some computational overhead for generating compiled forms, it often proves quite efficient as an exact model counter.

5 Conclusion

We have described D4, a new top-down, tree-search compiler targeting the Decision-DNNF language. In D4, a dynamic decomposition of the input CNF formula under the current partial assignment is computed during the search. This decomposition is based on hypergraph partitioning, used sparingly. Specific hypergraph simplification rules are also used in order to minimize the time spent in each partitioning step and to promote the quality of the decompositions. Experiments have shown that substantial computational savings can be obtained using D4, both in terms of compilation times and (even more) in terms of the sizes of the compiled forms which are generated. D4 also appears as a quite competitive model counter for many instances (despite the computational overhead due to the generation of the compiled form).

References

[Cadoli and Donini, 1998] M. Cadoli and F.M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3–4):137–150, 1998.

[Catalyürek and Aykanat, 2011] U.V. Catalyürek and C. Aykanat. *PaToH (Partitioning Tool for Hypergraphs)*, pages 1479–1487. Encyclopedia of Parallel Computing. 2011.

[Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *Journal of the Association for Computing Machinery*, 48(4):608–647, 2001.

[Darwiche, 2004] A. Darwiche. New advances in compiling cnf into decomposable negation normal form. In *Proc. of ECAI'04*, pages 328–332, 2004.

[Darwiche, 2014] A. Darwiche. *Tractable Knowledge Representation Formalisms*, pages 141–172. Tractability – Practical Approaches to Hard Problems. Cambridge University Press, 2014.

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. of SAT'03*, pages 502–518. 2003.

[Fargier and Marquis, 2006] H. Fargier and P. Marquis. On the use of partially ordered decision graphs in knowledge compilation and quantified Boolean formulae. In *Proc. of AAAI'06*, pages 42–47, 2006.

[Huang and Darwiche, 2007] J. Huang and A. Darwiche. The language of search. *Journal of Artificial Intelligence Research (JAIR)*, 29:191–219, 2007.

[Koriche *et al.*, 2015] F. Koriche, J.-M. Lagniez, P. Marquis, and S. Thomas. Compiling constraint networks into multivalued decomposable decision graphs. In *Proc. of IJCAI'15*, pages 332–338, 2015.

[Lagniez and Marquis, 2014] J.-M. Lagniez and P. Marquis. Preprocessing for propositional model counting. In *Proc. of AAAI'14*, pages 2688–2694, 2014.

[Marquis, 2015] P. Marquis. Compile! In *Proc. of AAAI'15*, pages 4112–4118, 2015.

[Moskewicz *et al.*, 2001] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.

[Muise *et al.*, 2012] Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

[Oztok and Darwiche, 2014] U. Oztok and A. Darwiche. On compiling CNF into Decision-DNNF. In *Proc. of CP'14*, pages 42–57, 2014.

[Oztok and Darwiche, 2015a] U. Oztok and A. Darwiche. A top-down compiler for sentential decision diagrams. In *Proc. of IJCAI'15*, pages 3141–3148, 2015.

[Oztok and Darwiche, 2015b] U. Oztok and A. Darwiche. A top-down compiler for sentential decision diagrams. In *Proc. of IJCAI'15*, pages 3141–3148, 2015.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, pages 475–482, 2005.

[Thurley, 2006] M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. of SAT'06*, pages 424–429, 2006.

[Zhang and Stickel, 1996] H. Zhang and M.E. Stickel. An efficient algorithm for unit propagation. In *Proc. of ISAIM'96*, pages 166–169, 1996.

Chapitre 8

A Recursive Algorithm for Projected Model Counting

Article paru dans les actes de « *the 33rd AAAI Conference on Artificial Intelligence* » (AAAI'19), pages 1536 – 1543, février 2019.

Co-écrit avec Pierre Marquis.

A Recursive Algorithm for Projected Model Counting

Jean-Marie Lagniez and **Pierre Marquis**[1]
CRIL, U. Artois & CNRS
Institut Universitaire de France[1]
F-62300 Lens, France
{lagniez, marquis}@cril.fr

Abstract

We present a recursive algorithm for projected model counting, i.e., the problem consisting in determining the number of models $\|\exists X.\Sigma\|$ of a propositional formula Σ after eliminating from it a given set X of variables. Based on a "standard" model counter, our algorithm projMC takes advantage of a disjunctive decomposition scheme of $\exists X.\Sigma$ for computing $\|\exists X.\Sigma\|$. It also looks for disjoint components in its input for improving the computation. Our experiments show that in many cases projMC is significantly more efficient than the previous algorithms for projected model counting from the literature.

Introduction

In this paper, we are concerned with the projected model counting problem. Given a propositional formula Σ and a set X of propositional variables to be forgotten, one wants to compute the number of interpretations of the variables occurring in Σ but not in X, which coincide on X with a model of Σ. Stated otherwise, the objective is to count the number of models of the quantified Boolean formula $\exists X.\Sigma$ over its variables (i.e., the variables occurring in Σ but not in X).

The projected model counting problem is a central issue to a number of AI problems (for instance, in planning, when the objective is to compute the robustness of a given plan given by the number of initial states from which the execution of the plan reaches a goal state (Aziz et al. 2015)), but also outside AI (especially it proves useful in some formal verification problems (Klebanov, Manthey, and Muise 2013)).

Since it generalizes the standard model counting problem (recovered when $X = \emptyset$), the projected model counting problem is at least as hard as the latter (#P-hard). The presence of variables X to be forgotten nevertheless may render the problem easier in some cases (thus, when every variable of Σ belongs to X, the problem boils down to deciding the satisfiability of Σ). That said, a naive approach which would consist first in forgetting the variables of X from Σ, then in counting the number of models of the resulting formula would be impractical in many cases, in particular when X is large. Indeed, forgetting the variables of X from Σ in a brute-force way often leads to a formula which is much larger than Σ (in the worst case, an exponential blow-up may occur).

Despite the importance of the problem, few algorithms for projected model counting have been pointed out so far. An FPT algorithm, where the parameter is the treewidth of the primal graph of the input instance, has been designed recently (Fichte et al. 2018), but this algorithm is practical only for instances having a small treewidth. Three other algorithms for the projected model counting task have been presented in (Aziz et al. 2015), namely dSharp$_{\mathsf{P}}$, #clasp, and d2c. Those algorithms are quite dissimilar in essence:

- dSharp$_{\mathsf{P}}$ is an adaptation of the model counter dSharp (Muise et al. 2012), which computes a Decision-DNNF representation of its input Σ to determine the number of models $\|\Sigma\|$. dSharp$_{\mathsf{P}}$ considers as an additional input a set P of protected variables (i.e., those variables of Σ which should not be forgotten). The search achieved by dSharp is constrained so that the decision variables are taken in priority from P. Whenever there is no variable of P in the current formula (i.e., this formula contains only variables from X), a sat solver is used to determine whether it is consistent. If so, its number of models is 1, otherwise it is equal to 0. This technique has also been considered in (Klebanov, Manthey, and Muise 2013). Note that the constraint imposed on the variable ordering limits the ability to find cutsets of small size, and this may have a strong (yet negative) impact on the quality of the conjunctive decompositions of Σ found by dSharp.
- #clasp is an extension of clasp, an algorithm for model enumeration on a projected set of variables (Gebser, Kaufmann, and Schaub 2009), which basically amounts to computing a deterministic DNF representation of $\exists X.\Sigma$. Each time an implicant of $\exists X.\Sigma$ is found, a blocking clause equivalent to the negation of this implicant is added to Σ. In #clasp, whenever an implicant of $\exists X.\Sigma$ is found, a prime implicant is first extracted from it in a greedy fashion (this is reminiscent to the approach considered in (Schrag 1996; Castell 1996)). Compared to clasp, this further extraction step often leads to the adding of far less blocking clauses to Σ, so that #clasp performs usually better than clasp.

- d2c consists in computing first a Decision-DNNF representation of Σ, then forgetting in it all the variables of X (this can be achieved in linear time, but the resulting representation is not deterministic any longer in the general case). The next step consists in turning this resulting representation into a CNF one. This can be done in linear time via the introduction of new variables while preserving the number of models of the input (Tseitin 1968). Finally, the number of models of the resulting CNF formula is evaluated using sharpSAT (Thurley 2006).

Unlike those algorithms, our algorithm for projected model counting, called projMC, is a recursive algorithm exploiting a disjunctive decomposition scheme of $\exists X.\Sigma$ for computing $\|\exists X.\Sigma\|$. More precisely, $\exists X.\Sigma$ is split into an equivalent (disjunctively interpreted) set of pairwise inconsistent formulae, so that $\|\exists X.\Sigma\|$ can be computed by summing up the corresponding projected model counts. projMC also looks for disjoint components in its input for improving the computation.

To evaluate the performance of our approach, we measured the time required by projMC for achieving the projected model counting task on a number of benchmarks (uniform random 3-CNF formulae, random Boolean circuits, planning instances). Our experiments show that in many cases projMC challenges the previous algorithms for projected model counting from the literature. For some benchmarks, when compared to dSharp$_{\mathsf{P}}$, #clasp, and d2c, the time savings achieved by projMC are of several orders of magnitude. In order to verify that the improvements obtained in practice with projMC are actually due to the underlying approach and not to the performance of the "standard" model counter used in it (which is D4 (Lagniez and Marquis 2017) in our implementation), we also developed D4$_{\mathsf{P}}$, which is the same algorithm as dSharp$_{\mathsf{P}}$, but using D4 as a model counter instead of dSharp. While D4$_{\mathsf{P}}$ performs typically better than dSharp$_{\mathsf{P}}$ (and #clasp and d2c), projMC turns out to be a better performer than D4$_{\mathsf{P}}$ for many instances.

The rest of the paper is organized as follows. In the next section, we present some formal preliminaries. Then we describe our new projected model counter. Afterwards we present the empirical protocol which has been considered in the experiments, as well as the corresponding experimental results. Finally, a last section concludes the paper and gives some perspectives for further research. The binary code of projMC, as well as the benchmarks used in our experiments and additional empirical results are available from www.cril.fr/KC/.

Formal Preliminaries

Let $\mathcal{L}_{\mathcal{P}}$ be a propositional language built up from a finite set of propositional variables $\mathcal{P}$ and the usual connectives. $\bot$ (resp. $\top$) is the Boolean constant always false (resp. true). An *interpretation* (or world) ω is a mapping from $\mathcal{P}$ to $\{0,1\}$.The set of all interpretations is denoted $\mathcal{W}$. An interpretation ω is a *model* of a formula $\varphi \in \mathcal{L}_{\mathcal{P}}$ if and only if it makes it true in the usual truth functional way. $Mod(\varphi)$ denotes the set of models of the formula φ, i.e., $Mod(\varphi) = \{\omega \in \mathcal{W} \mid \omega \models \varphi\}$. $\models$ denotes logical entailment and $\equiv$ logical equivalence, i.e., $\varphi \models \psi$ iff $Mod(\varphi) \subseteq Mod(\psi)$ and $\varphi \equiv \psi$ iff $Mod(\varphi) = Mod(\psi)$. When $\varphi \in \mathcal{L}_{\mathcal{P}}$ and $X \subseteq \mathcal{P}$, $\exists X.\varphi$ is a quantified Boolean formula denoting (up to logical equivalence) the most general consequence of φ which is independent from the variables of X (see e.g., (Lang, Liberatore, and Marquis 2003)). $Var(\varphi)$ denotes the set of variables occurring in $\varphi \in \mathcal{L}_{\mathcal{P}}$ and $Var(\{\varphi_1, \ldots, \varphi_k\}) = \bigcup_{i=1}^{k} Var(\varphi_i)$; when $X \subseteq \mathcal{P}$, we have $Var(\exists X.\varphi) = Var(\varphi) \setminus X$.

A *literal* ℓ is a propositional variable or a negated one. When ℓ is a literal over x, its *complementary literal* $\sim\ell$ is given by $\sim\ell = \neg x$ if $\ell = x$ and $\sim\ell = x$ if $\ell = \neg x$, and we note $var(\ell) = x$. A literal ℓ is *pure* in a CNF formula Σ when Σ contains no occurrence of $\sim\ell$. One also says that the corresponding variable $var(\ell)$ is pure in Σ. A *term* is a conjunction of literals. It is also viewed as the set of its literals when this is convenient. A *clause* is a disjunction of literals. When $\delta = \bigvee_{j=1}^{m} \ell_j$ is a clause, $\sim\delta$ denotes the term given by $\bigwedge_{j=1}^{m} \sim\ell_j$. A clause δ_1 is a *subclause* of a clause δ_2 when every literal of δ_1 is a literal of δ_2. A CNF formula φ is a conjunction of clauses. It is also viewed as the set of its clauses when this is convenient. $\|\varphi\|$ denotes the number of models of φ over $Var(\varphi)$.

Given a subset X of $\mathcal{P}$ and a world ω, $\omega[X]$ is the term $\bigwedge_{x \in X \mid \omega \models x} x \wedge \bigwedge_{x \in X \mid \omega \models \neg x} \neg x$. The *conditioning* of a CNF formula φ by a consistent term γ is the CNF formula $\varphi \mid \gamma$ obtained from φ by removing each clause containing a literal of γ and by shortening the remaining clauses, removing from them the complementary literals of those of γ.

BCP denotes a *Boolean Constraint Propagator* (Zhang and Stickel 1996; Moskewicz et al. 2001), which is a key component of many solvers. BCP(Σ) returns the CNF formula Σ once simplified using unit propagation.

The *primal graph* of a CNF formula Σ is the (undirected) graph where vertices correspond to the variables of Σ and an edge connecting two variables exists whenever one can find a clause of Σ where both variables occur. Every connected component of this graph (i.e., a maximal subset of vertices which are pairwise connected by a path) corresponds to a subset of clauses of Σ, referred to as a *connected component* of the formula Σ.

A Recursive Algorithm for Projected Model Counting

Our projected model counter projMC computes $\|\exists X.\Sigma\|$ where Σ is a CNF formula and X a set of propositional variables. Assuming Σ being in CNF is harmless since one can use Tseitin technique (Tseitin 1968) to associate in linear time with any propositional circuit into a CNF formula having the same number of models.

Unlike the previous algorithms for projected model counting sketched in the introductive section, projMC is a recursive algorithm guided by a *deterministic disjunctive form* for Σ w.r.t. X. Such a form is a (disjunctively interpreted) set of formulae $\{\varphi_1, \ldots, \varphi_{k+1}\}$ over $Var(\Sigma)$ satisfying the following conditions:

- $\forall i \in \{1,\ldots,k+1\}$, $Var(\varphi_i) \cap X = \emptyset$;
- $\forall i,j \in \{1,\ldots,k+1\}$, if $i \neq j$ then $\varphi_i \wedge \varphi_j \models \bot$;
- $\bigvee_{i=1}^{k+1} \varphi_i \equiv \top$.

Because $\bigvee_{i=1}^{k+1} \varphi_i$ is valid, we have $\exists X.\Sigma \equiv (\exists X.\Sigma) \wedge (\bigvee_{i=1}^{k+1} \varphi_i) \equiv \bigvee_{i=1}^{k+1} (\exists X.\Sigma) \wedge \varphi_i \equiv \bigvee_{i=1}^{k+1} \exists X.(\Sigma \wedge \varphi_i)$ since no element of a deterministic disjunctive form for Σ w.r.t. X contains a variable of X. Furthermore, since the elements of a deterministic disjunctive form for Σ w.r.t. X are pairwise conflicting, we get that $\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|(\exists X.\Sigma) \wedge \varphi_i\|$, hence

$$\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|\exists X.(\Sigma \wedge \varphi_i)\|.$$

The set $\{\Sigma \wedge \varphi_i \mid i \in \{1,\ldots,k+1\}\}$ is referred to as *the disjunctive decomposition* associated with $\{\varphi_1,\ldots,\varphi_{k+1}\}$.

Technically speaking, such a notion of disjunctive decomposition can be related to several concepts considered in some previous works about SAT or #SAT. On the one hand, it can be viewed as a specific case of the notion of set of scattered formulae from a formula Σ (see (Hyvärinen, Junttila, and Niemelä 2006) for details), obtained by focusing on clauses/terms not containing any variable from X. However, the objective pursued in (Hyvärinen, Junttila, and Niemelä 2006) was quite distinct from our own one (in this paper, the decomposition was used as a distribution method for SAT solving in grids). On the other hand, when it consists of consistent formulae, a deterministic disjunctive form for Σ w.r.t. X forms a partition, just as the primes of an X-partition of Σ, used for generating an SDD representation of Σ (see (Darwiche 2011) for details). Nevertheless, the connection to SDD does not extend further: whenever Σ contains variables from X and from $Var(\Sigma) \setminus X$, the disjunctive decomposition associated with a deterministic disjunctive form is not a $(X, Var(\Sigma) \setminus X)$-decomposition of Σ: disjunctive decompositions do not aim to create formulae that do not share variables.

Clearly enough, generating a disjunctive decomposition $\{\Sigma \wedge \varphi_i \mid i \in \{1,\ldots,k+1\}\}$ associated with a deterministic disjunctive form for Σ w.r.t. X is computationally useful for computing $\|\exists X.\Sigma\|$ only if the formulae $\Sigma \wedge \varphi_i$ are somewhat more simple than Σ. To ensure it and guarantee the termination of projMC, one does not compute any deterministic disjunctive form for Σ w.r.t. X but one *induced by a model* ω of Σ (if there is not such model, then $\exists X.\Sigma$ is inconsistent and $\|\exists X.\Sigma\| = 0$). One starts by considering the CNF formula $\Sigma \mid \omega[X] = \bigwedge_{i=1}^{k} \delta_i$, called the *core* φ_1 of the deterministic disjunctive form $\{\varphi_1,\ldots,\varphi_{k+1}\}$ for Σ w.r.t. X induced by ω. By construction, $\varphi_1 = \bigwedge_{i=1}^{k} \delta_i$ contains variables occurring in $Var(\Sigma)$ but not in X. Then we generate a (disjunctively interpreted) set of CNF formulae $\{\varphi_2,\ldots,\varphi_{k+1}\}$ which is equivalent to the negation of φ_1, by defining, for each $i \in \{1,\ldots,k\}$, $\varphi_{i+1} = (\bigwedge_{j=1}^{i-1} \delta_j) \wedge \sim\delta_i$. By construction, $\{\varphi_1,\ldots,\varphi_{k+1}\}$ satisfies the conditions of a deterministic disjunctive form for Σ w.r.t. X. Note also that $\exists X.(\Sigma \wedge \varphi_1)$ is equivalent to φ_1. Indeed, $\exists X.(\Sigma \wedge \varphi_1)$ is equivalent to $(\exists X.\Sigma) \wedge \varphi_1$ since φ_1 does not contain any variable of X. Furthermore, since $\omega[X] \wedge \Sigma \models \Sigma$, we have that $\exists X.(\omega[X] \wedge \Sigma) \models \exists X.\Sigma$. But $\exists X.(\omega[X] \wedge \Sigma)$ is equivalent to $\Sigma \mid \omega[X]$, hence to φ_1, so $\varphi_1 \models \exists X.\Sigma$, and as a consequence $(\exists X.\Sigma) \wedge \varphi_1$ is equivalent to φ_1. Accordingly, when computing the number of models of the disjunctive decomposition $\{\Sigma \wedge \varphi_i \mid i \in \{1,\ldots,k+1\}\}$ associated with $\{\varphi_1,\ldots,\varphi_{k+1}\}$, one replaces $\Sigma \wedge \varphi_1$ by φ_1. Since $Var(\varphi_1) \cap X = \emptyset$, one can take advantage of a "standard" model counter for computing $\|\varphi_1\|$ and no recursive call of projMC is needed (this is a base case for the recursion).

Our algorithm projMC also takes advantage of any conjunctive decomposition of its input Σ into disjoint connected components whenever such a decomposition exists. Indeed, whenever Σ can be split into two sets of clauses α and β not sharing any variable so that $\Sigma \equiv \alpha \wedge \beta$, then the computation of $\exists X.\Sigma$ can be achieved by computing $\exists X.\alpha$ and $\exists X.\beta$ since $\exists X.\Sigma \equiv \exists X.(\alpha \wedge \beta) \equiv (\exists X.\alpha) \wedge (\exists X.\beta)$. And since $\exists X.\alpha$ and $\exists X.\beta$ do not share any variable, one can compute $\|\exists X.\Sigma\| = \|\exists X.\alpha\| \times \|\exists X.\beta\|$. As it is the case for the model counting problem, taking advantage of conjunctive decompositions proves to be very useful for the efficiency purpose.

Algorithm 1: projMC(Σ, X)

input : a CNF formula Σ
input : a set of variables X to be forgotten
output: the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$

```
1  Σ ← BCP(Σ)
2  if Var(Σ) ∩ X = ∅ then return MC(Σ)
3  if cache(Σ) ≠ nil then return cache(Σ)
4  comps ← connectedComponents(Σ)
5  if #(comps) ≠ 1 then
6  |   cpt ← 1
7  |   foreach φ ∈ comps do
8  |   |   cpt ← cpt× projMC(φ, X)
9  else
10 |   ω ← sat(Σ)
11 |   cpt ← 0
12 |   if ω = ∅ then break
13 |   dd ← DD(Σ, ω, X)
14 |   foreach φ ∈ dd do
15 |   |   cpt ← cpt+
   |   |       projMC(φ, X) ×2^#(Var(dd)\(Var(φ)∪X))
16 cache(Σ) ← cpt
17 return cpt
```

More in detail, Algorithm 1 provides the pseudo-code of the projected model counter projMC. The projected model counting of a given CNF formula Σ w.r.t. a set of variables X to be forgotten is achieved by calling projMC(Σ, X). At line 1, Σ is simplified using Boolean constraint propagation. At line 2, we consider the case when Σ does not contain any variable of X. In this case, there is no variable to be forgot-

ten and it is enough to compute the number of models of Σ, using any model counter MC. This is a base case of our algorithm (no recursion takes place). At line 3 one looks into a cache to check whether or not the current formula Σ has already been encountered during the search. The cache, initially empty, gathers pairs consisting of a CNF formula and the corresponding projected model count w.r.t. X. Each time Σ has already been cached, instead of re-computing $\|\exists X.\Sigma\|$ from scratch, it is enough to return cache(Σ). In our implementation, the cache is residual and shared with the model counter MC. At line 4, one partitions Σ into a set of CNF formulae that are pairwise variable-independent (i.e., two distinct elements of *comps* are built up from two disjoint sets of variables). connectedComponents is a "standard" procedure used in previous model counters. It is based on the search for the connected components of the primal graph of Σ and it returns a set *comps* of CNF formulae composed of clauses from Σ, so that every pair of distinct formulae from *comps* do not share any common variable. At line 5, one tests how many connected components have been found in Σ. If there are more than one component, then at lines 6, 7, and 8, one recursively computes the projected model counts corresponding to each of them, and we store in cpt the product of the corresponding counts. For efficiency reasons, in our implementation, one uses a specific trick that it is not made explicit in the algorithm for the sake of readibility: systematically, one sets aside all the components forming a consistent term containing as many literals as possible. For instance, if the input Σ is equal to $x_1 \wedge \neg x_2 \wedge (x_3 \vee x_4)$, one sets aside the two components x_1 and $\neg x_2$ to keep only the component $x_3 \vee x_4$. Indeed, for a set of components forming a consistent term like $\{x_1, \neg x_2\}$, the corresponding projected model count is always equal to 1 (whatever the variables occurring in the term belongs to X or not). Hence projMC returns directly 1 in this case, without needing to consider the literals of the term independently in distinct recursive calls. At line 9, one considers the remaining case when only one component has been found (i.e., no decomposition exists). At line 10, one looks for a model ω of Σ over $Var(\Sigma)$. If no such model exists ($\omega = \emptyset$), then Σ is inconsistent, hence so is $\exists X.\Sigma$, and the algorithm exits from the conditional (line 12). The heuristic used by the solver sat tries to satisfy Σ by assigning in priority the variables from X. An interesting consequence of this heuristic is that it leads to a lazy handling of the clauses of Σ which contain a literal ℓ pure in Σ and such that $var(\ell) = x \in X$. Indeed, when ℓ is pure in Σ, ℓ will be set to 1 by ω so that no clauses of Σ containing ℓ will belong to $\Sigma \mid \omega[X]$. This does not invalidate the correctness of the approach since such clauses can be removed from Σ without changing its projected model count. Thus, while the "standard" pure literal rule (i.e., the one consisting in removing every clause of Σ containing a variable which is pure in Σ) cannot be applied safely (it does not preserve the number of models of its input in the general case), its restriction where only pure literals over X are considered is correct: let α (resp. β) be the subset of the clauses of Σ containing ℓ (resp. not containing ℓ). We have $\exists X.\Sigma \equiv \exists X.(\exists\{x\}.(\alpha \wedge \beta))$. Since ℓ is pure in Σ, x does not occur in β. Hence $\exists\{x\}.(\alpha \wedge \beta) \equiv (\exists\{x\}.\alpha) \wedge \beta$. Now, since every clause of α contains ℓ, $\exists\{x\}.\alpha$ is valid, therefore $\exists X.\Sigma \equiv \exists X.\beta$. At line 13, once computes the disjunctive decomposition dd associated with the deterministic disjunctive form for Σ w.r.t. X induced by ω. At lines 11, 14, and 15, one sums the projected model counts for the formulae belonging to dd (note that when the elements φ of dd do not contain the same set of variables, a preliminary normalization step – multiplying each count by 2 to the power of the number of variables occurring in dd but not in φ and not in X – before summing up, is necessary). Finally, at line 16, one adds Σ to the cache, associated with the corresponding projected model count cpt, and at line 17, one returns the value of cpt.

Algorithm 2: DD(Σ, ω, X)

input : a CNF formula Σ
input : a model ω of Σ over $Var(\Sigma)$
input : a set of variables X to be forgotten
output: a set dd of CNF formulae, the disjunctive decomposition associated with the deterministic disjunctive form for Σ w.r.t. X induced by ω

1 $\{\delta_1, \ldots, \delta_k\} \leftarrow \mathtt{BCP}(\Sigma \mid \omega[X])$
2 $dd \leftarrow \{\bigwedge_{i=1}^{k} \delta_i\}$
3 foreach $\delta_j \in \{\delta_1, \ldots, \delta_k\}$ do
4 $\quad dd \leftarrow dd \cup \{\Sigma \wedge (\bigwedge_{l=1}^{j-1} \delta_l) \wedge \sim\delta_j\}$
5 return dd

Algorithm 2 provides the pseudo-code of the program which generates the disjunctive decomposition associated with the deterministic disjunctive form of Σ w.r.t. X induced by the model ω of Σ used for guiding the search. At line 1, Σ is first conditioned by the consistent term $\omega[X]$. This means that every clause of Σ containing a literal belonging to $\omega[X]$ is removed from Σ, and in the remaining clauses, every literal ℓ over X such that $\sim\ell$ is a literal of $\omega[X]$ is removed. Finally, Boolean constraint propagation is applied to $\Sigma \mid \omega[X]$ in order to simplify it further (if possible). Clearly, no variable of X occurs in the resulting (conjunctively interpreted) set of clauses $\{\delta_1, \ldots, \delta_k\}$, which is the core of the deterministic disjunctive form for Σ w.r.t. X induced by ω that is computed. At line 2, we first initialize the disjunctive decomposition dd as the singleton containing the core. At lines 3 and 4, we add to dd k CNF formulae since the core contains k clauses. At line 4, the clauses of Σ subsumed by the clauses δ_l are removed (this is not detailed in the pseudo-code for the sake of readability). By construction, the formulae of dd are pairwise inconsistent. Furthermore, every δ_j of the core is a subclause of a clause of Σ since the core is obtained by conditioning Σ using a consistent term. Accordingly, in our implementation, when δ_j is equal to a clause of Σ, δ_j is not taken into consideration within the loop (indeed, in this case, $\Sigma \wedge \delta_j$ is equivalent to Σ and $\Sigma \wedge \sim\delta_j$ is inconsistent, so that its contribution to the total model count will be equal to 0). Finally, at line 5, one returns the disjunctive decomposition dd which has been computed.

An Example. Here is an example illustrating how projMC works. Let $\Sigma = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_5)$. Let $X = \{x_2, x_3\}$. The formula $\exists X.\Sigma$ is equivalent to $x_1 \vee x_4 \vee x_5$, so that $\|\exists X.\Sigma\| = 7$.

When run on Σ and X, the first instructions of projMC have no effect (applying BCP does not change Σ, and Σ has a unique connected component). Suppose that the model ω of Σ setting every variable to 1 has been found at line 10. Then the core $\Sigma \mid \omega[X]$ of the deterministic disjunctive form for Σ w.r.t. X induced by ω will be equal to x_5. As a consequence, the other formula belonging to the disjunctive decomposition associated with this deterministic disjunctive form will be $\Sigma \wedge \neg x_5$.

The recursive call of projMC on the core x_5 and X will lead to calling MC on x_5 as expected (since no variable of X occurs in the core). Then MC returns 1, and finally the recursive call of projMC on x_5 returns $1 \times 2^2 = 4$ since the two variables x_1, x_4 belong to the set of variables of dd but not to the set of variables of the associated core (this normalization step is achieved at line 15).

The recursive call of projMC on $\Sigma \wedge \neg x_5$ and X leads first to simplify $\Sigma \wedge \neg x_5$ using BCP, so that the formula $\neg x_5 \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_3$ is got. Since the two components $\neg x_5$ and $\neg x_3$ form together a consistent term, only one component actually needs to be considered, namely $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4)$.

So, at that step, projMC is called on $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4)$ and X. Suppose again that the model ω of Σ setting every variable to 1 has been found. Then the resulting core will be equal to x_4, and the resulting disjunctive decomposition will be $\{x_4, (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_4\}$. Since $\neg x_4$ does not contain a variable of X, at the next recursive call to projMC, MC is used to compute its model counts, equal to 1, and finally the recursive call of projMC on x_4 returns $1 \times 2^1 = 2$ since the variable x_1 belongs to the set of variables of the disjunctive decomposition but not to the set of variables of the associated core. The recursive call of projMC on $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_4$ and X leads first to simplify (using BCP) the input into $x_1 \wedge \neg x_2 \wedge \neg x_4$, which is a consistent term. Hence it has only 1 model. Thus one obtains that $\|\exists X.((x_1 \vee x_2) \wedge (\neg x_2 \vee x_4))\| = 2 + 1 = 3$. Finally the previously computed counts are summed up. One thus gets $4 + 3 = 7$ models, as expected.

Interestingly, it can be observed that projMC handles adequately the cases when $X = \emptyset$ or $X = Var(\Sigma)$, in the sense that it does not waste too much time in many recursive calls for any of those two "extreme" situations. Indeed, when $X = \emptyset$, projMC mainly boils down to calling a "standard" model counter MC (line 2), as expected. When $X = Var(\Sigma)$, if Σ is unsatisfiable, then it will be detected as such during the first call to projMC (line 12) and a count of 0 will be returned. In the remaining case, a model ω of Σ will be found. Because $\omega[X]$ coincides with ω when $X = Var(\Sigma)$, the core of the deterministic disjunctive form for Σ w.r.t. X induced by ω will be equal to the empty set of clauses ($\Sigma \mid \omega$ is valid when ω is a model of Σ), so that the disjunctive decomposition associated with this deterministic disjunctive form will consist only of this core. Since this core contains no variable, the next recursive call to projMC (i.e., the one with the core as an operand) will mainly consist in calling the model counter MC (line 2) on it, and MC will return 1 in this case.

Finally, one can prove that our algorithm projMC actually does the job for which it has been designed:

Proposition 1 *Algorithm 1 is correct and terminates.*

Proof: The correctness of projMC (i.e., the fact that the result provided is equal to $\|\exists X.\Sigma\|$ on inputs Σ and X) comes from the fact that each rule used in the algorithm is sound w.r.t. the projected model counting task, as explained previously. The key equalities are $\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|\exists X.(\Sigma \wedge \varphi_i)\|$ for any deterministic disjunctive form $\{\varphi_1, \ldots, \varphi_{k+1}\}$ for Σ w.r.t. X, and $\|\exists X.(\alpha \wedge \beta)\| = \|\exists X.\alpha\| \times \|\exists X.\beta\|$ when $Var(\alpha) \cap Var(\beta) = \emptyset$.

The termination of projMC comes from the fact that each time a deterministic disjunctive form for Σ w.r.t. X is computed, its core does not contain any variable of X, so that the recursive call to projMC concerning it will lead to the base case of the recursion (line 2). As to the other disjoints $\Sigma \wedge \varphi_i$ of the corresponding disjunctive decomposition, by construction, every φ_i contains the negation of a subclause δ_{i-1} of Σ. Hence, at the next call to projMC concerning this disjoint, the literals of the term $\sim\delta_{i-1}$ will be assigned and the corresponding variables will be removed from the input using Boolean constraint propagation (line 1). Thus the number of variables of the input strictly diminishes at each step and this ensures the termination of the algorithm. ■

Note that the detection of disjoint components (lines 4 to 9 in Algorithm 1) coud be frozen in projMC without questioning the correctness and the termination of this algorithm (however, it has a significant impact on the efficiency of the computation on some instances). Similarly, the use of a cache has no impact on the correctness or the termination of the algorithm, so that the instructions at lines 3 and 16 could be frozen as well (but again using a cache proves to be computationally useful in many cases).

Empirical Evaluation

In order to evaluate the benefits offered by projMC, we performed some experiments. The empirical protocol we followed is precisely the same as the one considered in (Aziz et al. 2015). Thus we have considered 260 instances coming from three data sets. The first data set consists of 100 instances, based on random 3-CNF formulae, where the number of variables is set to 100 and the number of clauses is varied. Clause-to-variable ratios of 1, 1.5, 2, 3, and 4 have been considered. We let also the number of variables of X to vary (the elements of X are chosen uniformly at random). The second data set consists of 60 instances, corresponding to random Boolean circuits based on 30 variables. Those circuits are generated as follows. One keeps a set containing at start the 30 variables, and as long as the set is not a singleton, we randomly select an operator o (AND, OR, NOT), pick up operands V for o in the set at random, create a new variable v, add the gate $o \leftrightarrow o(V)$ to the circuit, and put v back in the set. The process is repeated c times, with c equals

to 1, 5, or 10. Finally, the third data set consists of 100 instances generated from five classical planning problems (depots, driver, rovers, logistics, and storage) considered under varying planning horizons. For each problem and value of the horizon, two variants are considered, one with the goal state fixed and one where the goal is relaxed to be any viable goal. For the first variant, the projected model count to be computed represents the number of initial states the given plan can achieve the goal from. For the second variant, it gives the number of initial states plus all goal configurations that the given plan works for.

For each instance, we measured the time (in seconds) required by projMC to achieve the projected model counting job. In the experiments, the model counter MC used in projMC is the top-down compilation-based model counter D4 described in (Lagniez and Marquis 2017). For the sake of comparison, we have also run the previous projected model counters dSharp$_\mathsf{P}$, #clasp, and d2c on the same instances (those solvers are available from people.eng.unimelb.edu.au/pstuckey/countexists) and measured the corresponding computation times. In addition, we have also compared projMC with D4$_\mathsf{P}$, which is the same algorithm as dSharp$_\mathsf{P}$, but using D4 as the underlying model counter instead of dSharp. All the experiments have been conducted on a cluster of Intel Xeon E5-2643 (3.30 GHz) quad core processors with 32 GiB RAM. The kernel used was CentOS 7, Linux version 3.10.0-514.16.1.el7.x86_64. The compiler used was gcc version 5.3.1. Hyperthreading was disabled, and no memory share between cores was allowed. A timeout of 600s and a memory-out of 7.6 GiB has been considered for each instance.

The results are reported on the scatter plots given in Figure 1. Each dot represents an instance; the time (in seconds) needed to solve it using the projected model counter corresponding to the x-axis (resp. y-axis), is given by its x-coordinate (resp. y-coordinate). Logarithmic scales are used for both coordinates. In part (a) (resp. (b), (c), (d)) of the figure, the x-axis corresponds to dSharp$_\mathsf{P}$ (resp. #clasp, d2c, D4$_\mathsf{P}$). The y-axis corresponds to projMC in each part of the figure.

The four scatter plots in Figure 1 clearly show that for a great majority of instances the time needed by projMC to count the number of projected models is smaller (and often significantly smaller) than the corresponding computation times when the other projected model counters are used. This is especially the case when the "trivial" instances (i.e., those solved with a second or alike) are neglected. Notwithstanding those instances, it is interesting to observe that projMC appears at least as efficient as any of the other projected model counters for all the instances from the random and the planning data sets.

The cactus plot in Figure 2 gives for dSharp$_\mathsf{P}$, #clasp, d2c, D4$_\mathsf{P}$, and projMC the number of instances solved in a given amount of time. Clearly enough, projMC outperforms the previous projected model counters: when the "trivial" instances have been discarded, projMC typically solves more instances than any of them in any given amount of time. Especially, some significant benefits in terms of the number of instances solved have been obtained. Thus, Table 1 makes precise for each projected model counter under consideration the number of instances (over 260) which have been solved within the time limit of 600s. It can be observed that projMC has been able to solve many more instances than the other projected model counters.

projected model counter	# of instances solved
dSharp$_\mathsf{P}$	115
#clasp	94
d2c	71
D4$_\mathsf{P}$	140
projMC	192

Table 1: Number of instances solved within the time limit depending on the projected model counter used.

Table 2 reports for each projected model counter under consideration the number of instances which have been solved by it, and only by it.

projected model counter	# of instances uniquely solved
dSharp$_\mathsf{P}$	0
#clasp	1
d2c	0
D4$_\mathsf{P}$	1
projMC	44

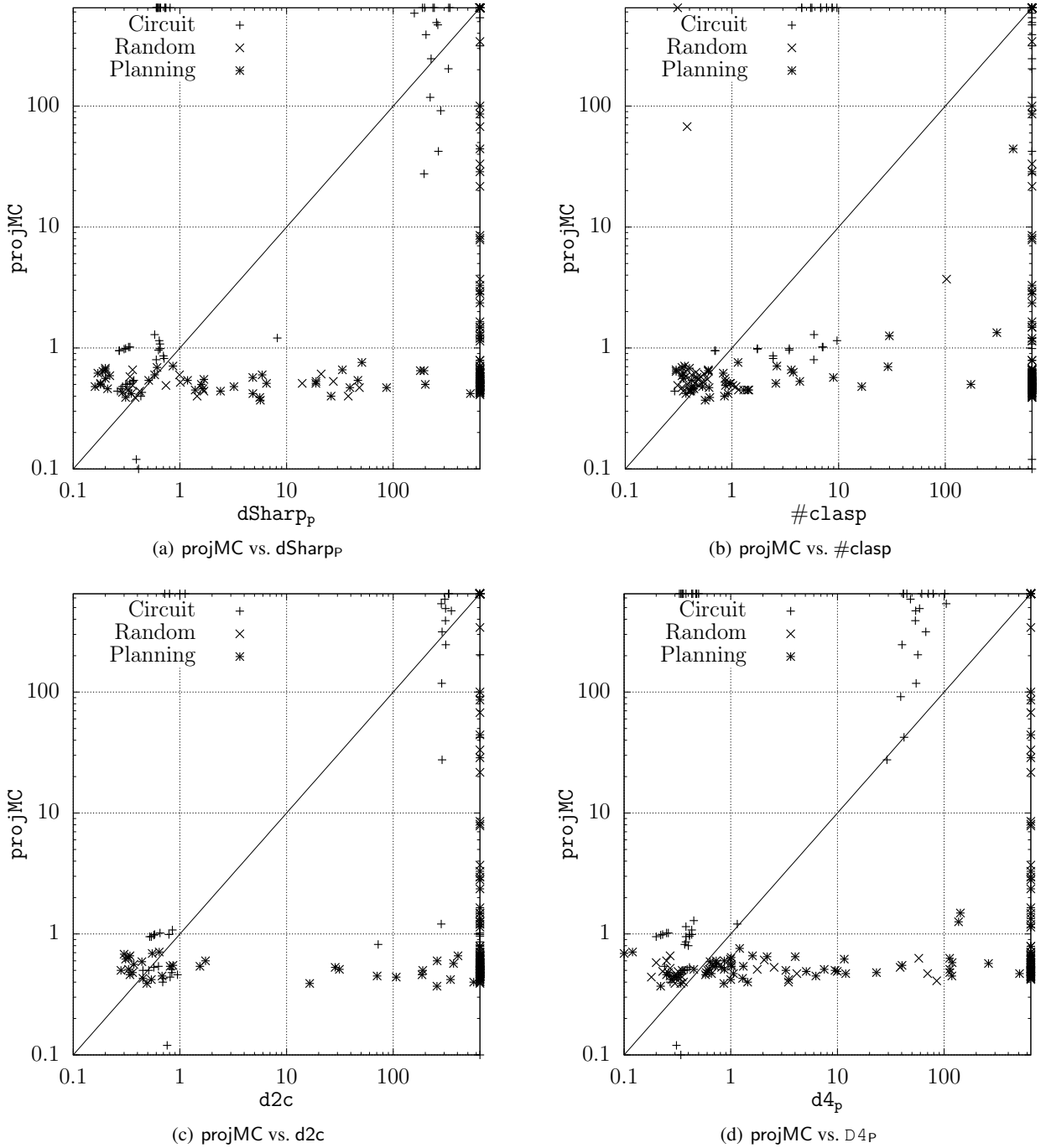
Table 2: Number of instances uniquely solved within the time limit depending on the projected model counter used.

Table 2 shows that projMC was able to solve a significant number of instances that were out of reach for the other projected model counters, given the time limit under consideration. That mentioned, the virtual best solver for our experiments would solve 211 instances, which is slightly above 192. This coheres with the results reported in Figure 1 (d), showing that for the circuit instances, D4$_\mathsf{P}$ typically challenges projMC.

Conclusion and Perspectives

We have presented a new algorithm, projMC, for computing the number of models $\|\exists X.\Sigma\|$ of a propositional formula Σ after eliminating from it a given set X of variables. Unlike previous algorithms, projMC takes advantage of a disjunctive decomposition scheme of $\exists X.\Sigma$ for computing $\|\exists X.\Sigma\|$. It also looks for disjoint components in its input for improving the computation. Our experiments have shown that projMC can be significantly more efficient than the existing algorithms dSharp$_\mathsf{P}$, #clasp, and d2c for projected model counting. Empirically, projMC also proved better than D4$_\mathsf{P}$ on many instances, showing that the improved performance of projMC is not solely due to the the fact that it is "powered" by D4.

A first perspective for further research consists in turning our projected model counter into a compiler generating a d-DNNF representation from a CNF formula containing



(a) projMC vs. dSharpP

(b) projMC vs. #clasp

(c) projMC vs. d2c

(d) projMC vs. D4P

Figure 1: Comparing projMC with dSharpP, #clasp, d2c, and D4P. The coordinates correspond to computation times in seconds. Logarithmic scales are used.

some existentially quantified variables. Provided that the underlying model counter MC is replaced by a d-DNNF compiler (like C2D (Darwiche 2002; 2004), dSharp (Muise et al. 2012) or D4 (Lagniez and Marquis 2017)), the changes to be done mainly consist in modifying the instructions at lines 14 and 15 of Algorithm 1 to generate a deterministic OR node instead of making a summation. In such a compiler, when a model of Σ is generated (i.e., at a step corresponding to line 10 of Algorithm 1), one could look for an assignment maximizing the number of clauses which are satisfied by

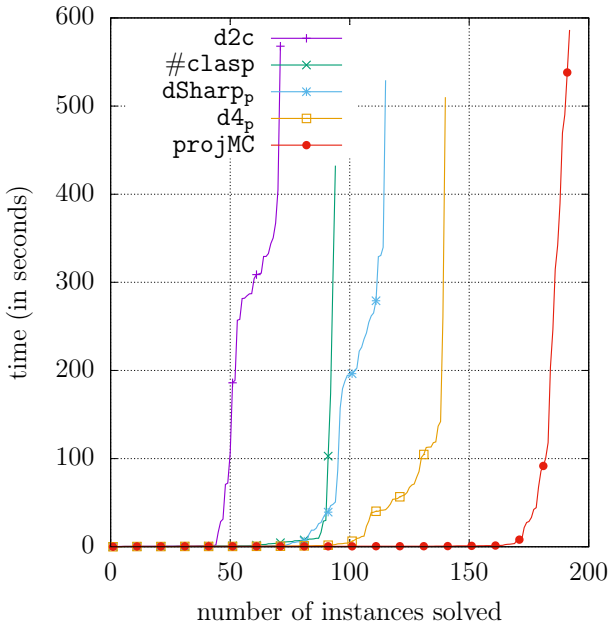


Figure 2: Number of instances solved by dSharp$_P$, #clasp, d2c, D4$_P$, and projMC in a given amount of time.

some existentially quantified literals of the model, so as to get a deterministic disjunctive form which contains as few elements as possible (hence a deterministic OR node with as few children as possible). In fact, one already tested this approach within projMC but the benefits achieved are not that significant in this case – the time spent in the maximisation processes can be quite large. However, things can be different when the objective is to generate a compiled representation since, in such a setting, one is typically ready to spend more off-line time in the compilation step, provided that the size of the associated compiled form is significantly smaller. We plan to make some experiments in this direction to determine whether this approach could prove useful for generating more succinct compiled representations.

A second perspective will consist in taking advantage of the two programs B and E for gate detection and replacement within CNF formulae, used as the key components of the preprocessor for model counting reported in (Lagniez, Lonca, and Marquis 2016). Indeed, B and E could be exploited as additional inprocessing filtering techniques in projMC. It would be interesting to determine whether this could be computationally useful, especially for solving circuit instances where, by construction, many gates can be found.

A last perspective will consist in evaluating projMC and the other projected model counters on other benchmarks, especially those reported in the repository available from `https://github.com/dfremont/counting-benchmarks/tree/master/benchmarks/projection`.[1]

[1]We would like to thank an anonymous reviewer for pointing out this dataset.

References

Aziz, R. A.; Chu, G.; Muise, C. J.; and Stuckey, P. J. 2015. #∃SAT: Projected model counting. In *Proc. of SAT'15*, 121–137.

Castell, T. 1996. Computation of prime implicates and prime implicants by a variant of the Davis and Putnam procedure. In *Proc. of ICTAI'96*, 428–429.

Darwiche, A. 2002. A compiler for deterministic decomposable negation normal form. In *AAAI'02*, 627–634.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In *Proc. of ECAI'04*, 328–332.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI'11*, 819–826.

Fichte, J. K.; Hecher, M.; Morak, M.; and Woltran, S. 2018. Exploiting treewidth for projected model counting and its limits. In *Proc. of SAT'18*, 165–184.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2009. Solution enumeration for projected Boolean search problems. In *Proc. of CPAIOR'09*, 71–86.

Hyvärinen, A. E. J.; Junttila, T. A.; and Niemelä, I. 2006. A distribution method for solving SAT in grids. In *Proc. of SAT'06*, 430–435.

Klebanov, V.; Manthey, N.; and Muise, C. J. 2013. SAT-based analysis and quantification of information flow in programs. In *Proc. of QUEST'13*, 177–192.

Lagniez, J.-M., and Marquis, P. 2017. An improved decision-DNNF compiler. In *Proc. of IJCAI'17*, 667–673.

Lagniez, J.-M.; Lonca, E.; and Marquis, P. 2016. Improving model counting by leveraging definability. In *Proc. of IJCAI'16*, 751–757.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence: Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research* 18:391–443.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, 530–535.

Muise, C.; McIlraith, S.; Beck, J.; and Hsu, E. 2012. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, 356–361.

Schrag, R. 1996. Compilation for critically constrained knowledge bases. In *Proc. of AAAI'96*, 510–515.

Thurley, M. 2006. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. of SAT'06*, 424–429.

Tseitin, G. 1968. *On the complexity of derivation in propositional calculus*. Steklov Mathematical Institute. chapter Structures in Constructive Mathematics and Mathematical Logic, 115–125.

Zhang, H., and Stickel, M. 1996. An efficient algorithm for unit propagation. In *Proc. of ISAIM'96*, 166–169.

Chapitre 9

Knowledge Compilation for Model Counting : Affine Decision Trees

Article paru dans les actes de « *the 25rd International Joint Conference on Artificial Intelligence* » (IJCAI'13), pages 947 – 953, août 2013.

Co-écrit avec Frédéric Koriche, Pierre Marquis et Samuel Thomas.

Knowledge Compilation for Model Counting: Affine Decision Trees*

Frédéric Koriche[1], **Jean-Marie Lagniez**[2], **Pierre Marquis**[1], **Samuel Thomas**[1]
[1]CRIL-CNRS, Université d'Artois, Lens, France
[2]FMV, Johannes Kepler University, Linz, Austria
{koriche,marquis,thomas}@cril.fr Jean-Marie.Lagniez@jku.at

Abstract

Counting the models of a propositional formula is a key issue for a number of AI problems, but few propositional languages offer the possibility to count models efficiently. In order to fill the gap, we introduce the language EADT of *(extended) affine decision trees*. An extended affine decision tree simply is a tree with affine decision nodes and some specific decomposable conjunction or disjunction nodes. Unlike standard decision trees, the decision nodes of an EADT formula are not labeled by variables but by affine clauses. We study EADT, and several subsets of it along the lines of the knowledge compilation map. We also describe a CNF-to-EADT compiler and present some experimental results. Those results show that the EADT compilation-based approach is competitive with (and in some cases is able to outperform) the model counter Cachet and the d−DNNF compilation-based approach to model counting.

1 Introduction

Model counting is a key issue in a number of AI problems, including inference in Bayesian networks and contingency planning [Littman *et al.*, 2001; Bacchus *et al.*, 2003; Sang *et al.*, 2005; Darwiche, 2009]. However, this problem is computationally hard (#P-complete) [Valiant, 1979]. Accordingly, few propositional languages offer the possibility to count models *exactly* in an efficient way [Roth, 1996].

The knowledge compilation (KC) map, introduced by Darwiche and Marquis [2002] and enriched by several authors (see among others [Wachter and Haenni, 2006; Subbarayan *et al.*, 2007; Mateescu *et al.*, 2008; Fargier and Marquis, 2008; Darwiche, 2011; Marquis, 2011; Bordeaux *et al.*, 2012]) is a multi criteria evaluation of languages, where languages are compared according to the queries and the transformations they support in polynomial time, as well as their relative succinctness (i.e., their ability to represent information using little space).

*This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE), and by the project BR4CP ANR-11-BS02-008 of the French National Agency for Research.

Among the languages which have been studied and classified according to the KC map, only the language d−DNNF of formulae in deterministic decomposable negation normal form [Darwiche, 2001], together with is subsets $\text{OBDD}_<$ [Bryant, 1986], FBDD [Gergov and Meinel, 1994], and SDD [Darwiche, 2011], satisfy the **CT** query (model counting). Yet, another interesting language which satisfies **CT** is AFF, the set of all affine formulae [Schaefer, 1978], defined as finite conjunctions of affine clauses (aka XOR-clauses). Unfortunately, AFF is not a *complete* language, because some propositional formulae (e.g. the clause $x \vee y$) cannot be represented into conjunctions of affine clauses.

By coupling ideas from affine formulae and decision trees, this paper introduces a new family of propositional languages that are complete and satisfy **CT**. The blueprint of our family is the class EADT of *extended affine decision trees*. In essence, an extended affine decision tree is a tree with decision nodes and some specific decomposable conjunction or disjunction nodes. Unlike usual decision trees, the decision nodes in an EADT are labeled by affine clauses instead of variables. Our family covers several subsets of EADT, including ADT (the set of affine decision trees where conjunction or disjunction nodes are prohibited), EDT (the set of extended decision trees where decomposable conjunction or disjunction nodes are allowed but decision nodes are mainly restricted to standard ones), and DT, the intersection of ADT and EDT.

Following the lines of the KC map, we prove that ADT and its subclass DT satisfy all queries and transformations offered by ordered binary decision diagrams ($\text{OBDD}_<$). Analogously, EADT and its subclass EDT satisfy all queries offered by d−DNNF and more transformations ($\neg$**C** is not satisfied by d−DNNF). Importantly, we also show that none of $\text{OBDD}_<$, CNF, and DNF is at least as succinct as any of ADT or EADT, and that EADT is strictly more succinct than ADT.

Finally, we describe a CNF-to-EADT compiler (which can be downsized to a compiler targeting ADT, EDT or DT). We used this program to compile a number of benchmarks from different domains. This empirical evaluation aimed at addressing two practical issues: (1) how challenging is an EADT compilation-based approach to model counting, compared to a direct, uncompiled method using a state-of-the-art model counter? (2) how does the EADT compilation-based approach perform compared to a d−DNNF compilation-based method? Our experimental results show that the EADT compilation-

based approach is competitive with both methods and, in some cases, it really outperforms them.

The rest of the paper is organized as follows. After introducing some background in Section 2, EADT and its subsets are defined and examined along the lines of the KC map in Section 3. In Section 4, our compiler is described and, in Section 5, some empirical results are presented and discussed. Finally, Section 6 concludes the paper. The runtime code of our compiler can be downloaded at `http://www.cril.fr/ADT/`

2 Preliminaries

We assume the reader familiar with propositional logic (including the notions of model, consistency, validity, entailment, and equivalence). All languages examined in this study are defined over a finite set PS of Boolean variables, and the constants $\top$ (true) and $\bot$ (false).

Affine Formulae. Let PS be a denumerable set of propositional variables. A literal (over PS) is an element $x \in PS$ (a positive literal) or a negated one $\neg x$ (a negative literal), or a Boolean constant $\top$ (true) or $\bot$ (false). An *affine clause* (aka XOR-clause) δ is a finite XOR-disjunction of literals (the XOR connective is denoted by $\oplus$). $var(\delta)$ is the set of variables occurring in δ. δ is *unary* when it contains precisely one literal. Obviously enough, each affine clause can be rewritten in linear time as a *simplified* affine clause, i.e., a finite XOR-disjunction of positive literals occurring once in the formula, plus possibly one occurrence of $\top$ (just take advantage of the fact that $\oplus$ is associative and commutative, and of the equivalences $\neg x \equiv x \oplus \top$, $x \oplus x \equiv \bot$, $x \oplus \bot \equiv x$, viewed as rewrite rules, left-to-right oriented). For instance, the affine clause $\neg x \oplus x \oplus \neg y \oplus \neg z$ can be turned in linear time into the equivalent simplified affine clause $y \oplus z \oplus \top$. An *affine formula* is a finite conjunction of affine clauses.

Knowledge Compilation. For space reasons, we assume the reader has a basic familiarity with the languages CNF, DNF, $\text{OBDD}_<$, SDD, BDD, FBDD, d-DNNF_T, d-DNNF, and DAG-NNF, which are considered in the following (see [Darwiche and Marquis, 2002; Pipatsrisawat and Darwiche, 2008; Darwiche, 2011] for formal definitions). The basic queries considered in the KC map include tests for consistency **CO**, validity **VA**, implicates (clausal entailment) **CE**, implicants **IM**, equivalence **EQ**, sentential entailment **SE**, model counting **CT**, and model enumeration **ME**. The basic transformations are conditioning (**CD**), (possibly bounded) closures under the connectives ($\wedge$**C**, $\wedge$**BC**, $\vee$**C**, $\vee$**BC**, $\neg$**C**), and forgetting (**FO**, **SFO**).

Finally, let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two propositional languages.

- $\mathcal{L}_1$ is *at least as succinct as* $\mathcal{L}_2$, denoted $\mathcal{L}_1 \leq_s \mathcal{L}_2$, iff there exists a polynomial p such that for every formula $\phi \in \mathcal{L}_2$, there exists an equivalent formula $\psi \in \mathcal{L}_1$ where $|\psi| \leq p(|\phi|)$.
- $\mathcal{L}_1$ is *polynomially translatable* into $\mathcal{L}_2$, noted $\mathcal{L}_1 \geq_p \mathcal{L}_2$, iff there exists a polynomial-time algorithm f such that for every $\phi \in \mathcal{L}_1$, $f(\phi) \in \mathcal{L}_2$ and $f(\phi) \equiv \phi$.

$<_s$ is the asymmetric part of $\leq_s$, i.e., $\mathcal{L}_1 <_s \mathcal{L}_2$ iff $\mathcal{L}_1 \leq_s \mathcal{L}_2$ and $\mathcal{L}_2 \not\leq_s \mathcal{L}_1$. When $\mathcal{L}_1 \geq_p \mathcal{L}_2$ holds, every query which is supported in polynomial time in $\mathcal{L}_2$ also is supported in polynomial time in $\mathcal{L}_1$; conversely, every query which is not supported in polynomial time in $\mathcal{L}_1$ unless P = NP is not supported in polynomial time in $\mathcal{L}_2$, unless P = NP.

3 The Affine Family

All propositional languages in our family are subsets of the very general language of *affine decision networks*:

Definition 1 ADN *is the set of all* affine decision networks*, defined as single-rooted finite DAGs where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are $\wedge$ nodes or $\vee$ nodes (with arbitrarily many children) or* affine decision nodes*, i.e., binary nodes of the form $N = \langle \delta, N_-, N_+ \rangle$ where δ is the affine clause labeling N and N_- (resp. N_+) is the left (resp. right) child of N.*

The size $|\Delta|$ of an ADN formula Δ is the sum of number of arcs in it, plus the cumulated size of the affine clauses used as labels in it. For every node N in an ADN formula Δ, $Var(N)$ is defined inductively as follows:

- if N is a leaf node, then $Var(N) = \varnothing$;
- if N is an affine decision node $N = \langle \delta, N_-, N_+ \rangle$, then $Var(N) = var(\delta) \cup Var(N_-) \cup Var(N_+)$;
- if N is a $\wedge$ node (resp. $\vee$ node) with children $N_1, \ldots, N_k$, then $Var(N) = \bigcup_{i=1}^{k} Var(N_i)$.

Clearly, $Var(\Delta) = Var(R_\Delta)$ (where R_Δ is the root of Δ) can be computed in time linear in the size of Δ. Every ADN formula Δ is interpreted as a propositional formula $I(\Delta)$ over $Var(\Delta)$, where $I(\Delta) = I(R_\Delta)$ is defined inductively as:

- if N is a leaf node labeled by $\top$ (resp. $\bot$), then $I(N) = \top$ (resp. $\bot$);
- if N is an affine decision node $N = \langle \delta, N_-, N_+ \rangle$, then $I(N) = ((\delta \oplus \top) \wedge I(N_-)) \vee (\delta \wedge I(N_+))$;
- if N is a $\wedge$ node (resp. $\vee$ node) with children $N_1, \ldots, N_k$, then $I(N) = \bigwedge_{i=1}^{k} I(N_i)$ (resp. $\bigvee_{i=1}^{k} I(N_i)$).

Finally, $\|\Delta\|$ represents the number of models of the ADN formula Δ over $Var(\Delta)$.

The DAG-NNF language considered in [Darwiche and Marquis, 2002] is polynomially translatable into a subset of ADN, where decision nodes have leaf nodes as children. Indeed, every leaf node labeled by a positive literal x (resp. negative literal $\neg x$) in a DAG-NNF formula is equivalent to the affine decision node N labeled by $\delta = x$ and such that $N_- = \bot$ (resp. $= \top$) and $N_+ = \top$ (resp. $= \bot$). Thus, ADN is a highly succinct yet intractable representation language; especially, it does not satisfy the **CT** query unless P = NP. To this point, we need to focus on tractable subsets of ADN.

Let us start with the EADT language, a class of tree-structured formulae defined in term of affine decomposability. Formally, a $\wedge$ (resp. $\vee$) node N with children $N_1, \ldots, N_k$ in an ADN Δ is said to be *affine decomposable* if and only if:

(1) for any $i, j \in 1, \ldots, k$, if $i \neq j$, then $Var(N_i) \cap Var(N_j) = \varnothing$, and

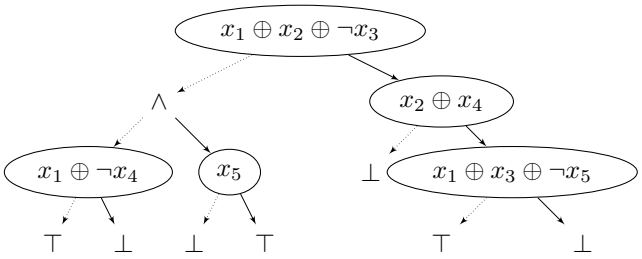


Figure 1: An EADT formula. Every dotted (resp. plain) arc links its source N to N_- (resp. N_+). The formula rooted at the node labeled by $x_2 \oplus x_4$ is an ADT formula.

(2) for every affine decision node N' of Δ which is a parent node of N and which is labelled by the affine clause $\delta_{N'}$, at most one child N_i of N is such that $var(\delta_{N'}) \cap Var(N_i) \neq \varnothing$.

If only the first condition holds, then the node N is said to be *(classically) decomposable*.

Definition 2 EADT *is the set of all* extended affine decision trees*, defined as finite trees where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are affine decision nodes, or affine decomposable $\wedge$ nodes, or affine decomposable $\vee$ nodes.*

An example of EADT formula is given at Figure 1. Some relevant subclasses of EADT are defined as follows:

Definition 3

- ADT *is the set of all* affine decision trees*, i.e., the subset of* EADT *consisting of finite trees where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are affine decision nodes.*
- EDT *is the set of all* extended decision trees*, i.e., the subset of* EADT *where affine decision nodes are labeled by unary affine clauses.*[1]
- DT*, the set of all* decision trees*, is the intersection of* ADT *and* EDT.

Based on this family, it is easy to check that the language TE of all terms and the language CL of all clauses are linearly translatable into DT, and hence, into each of ADT, EDT, and EADT. Furthermore, the language AFF is also polynomially translatable into ADT (hence into its superset EADT).

In contrast to TE, CL, and AFF, the class DT and its supersets ADT, EDT, and EADT are *complete* propositional languages. The completeness property also holds for $\texttt{ODT}_<$, which is the subset of DT consisting of formulae Δ in which every path from the root of Δ to a leaf respects the given, total, strict ordering $<$ (i.e., the variables labeling the decision nodes in the path are ordered in a way which is compatible with $<$). Clearly, $\texttt{ODT}_<$ also is a subset of $\texttt{OBDD}_<$ (to be more precise, $\texttt{ODT}_<$ is the intersection of DT and $\texttt{OBDD}_<$), and both CL and TE are polynomially translatable to it.

We are now in position to explain how any EADT formula Δ can be translated in linear time into a tree $T(\Delta)$ where internal nodes are decomposable $\wedge$ nodes or decomposable $\vee$ nodes or deterministic binary $\vee$ nodes, and the leaves are labeled with affine formulae. The translation T consists in rewriting Δ by parsing it in a top-down way, and collecting sets of affine clauses (those sets are the values of an inherited attribute a defined for each node N of Δ) along the paths of Δ during the translation. T proceeds recursively as follows starting with $N = R_\Delta$ and $a(R_\Delta) = \varnothing$:

- if $N = \langle \delta, N_-, N_+ \rangle$, then $T(N) = T(N_-) \vee T(N_+)$, $a(N_-) = a(N) \cup \{\delta \oplus \top\}$, and $a(N_+) = a(N) \cup \{\delta\}$;
- if $N = \bigwedge_{i=1}^k N_i$, then $T(N) = \bigwedge_{i=1}^k T(N_i)$, and for every $i \in 1, \ldots, k$, $a(N_i) = \{\delta \in a(N) \mid var(\delta) \cap Var(N_i) \neq \varnothing\}$;
- if $N = \bigvee_{i=1}^k N_i$, then $T(N) = \bigvee_{i=1}^k T(N_i)$, and for every $i \in 1, \ldots, k$, $a(N_i) = \{\delta \in a(N) \mid var(\delta) \cap Var(N_i) \neq \varnothing\}$;
- if $N = \top$, then $T(N) = \bigwedge_{\delta \in a(N)} \delta$;
- if $N = \bot$, then $T(N) = \bot$.

By construction, the translation T consists in replacing every affine decision node by a deterministic binary $\vee$ node, every affine decomposable $\wedge$ (resp. $\vee$) node by a (classically) decomposable $\wedge$ (resp. $\vee$) node. Thus, when Δ is an ADT formula, $T(\Delta)$ simply is a deterministic disjunction of affine formulae, and when Δ is a DT formula, $T(\Delta)$ simply is a deterministic DNF formula.

With this translation in hand, it is easy to show that EADT satisfies **CT**. The proof is by structural induction on $\phi = T(\Delta)$. First, $\|\phi\|$ can be computed in polynomial time when ϕ is an affine formula, since ϕ can be viewed as a finite system of linear equations modulo 2. Indeed, ϕ can be turned in polynomial time into its equivalent *reduced row echelon form* ϕ^r, and when $Var(\phi^r)$ contains n variables and ϕ^r contains k affine clauses, $\|\phi\|$ is equal to 2^{n-k}. This solves the base case. As to the inductive step, it is enough to check that:

- if $\phi = \bigwedge_{i=1}^k \phi_i$ (where $\bigwedge$ is a decomposable $\wedge$ node),
$$\|\phi\| = \prod_{i=1}^k \|\phi_i\|$$
- if $\phi = \bigvee_{i=1}^k \phi_i$ (where $\bigvee$ is a decomposable $\vee$ node),
$$\|\phi\| = 2^{|Var(\phi)|} - \prod_{i=1}^k (2^{|Var(\phi_i)|} - \|\phi_i\|)$$
- if $\phi = \phi_1 \vee \phi_2$ (where $\vee$ is a deterministic $\vee$ node), then
$$\|\phi\| = \|\phi_1\| \times 2^{|Var(\phi_2) \setminus Var(\phi_1)|} + \|\phi_2\| \times 2^{|Var(\phi_1) \setminus Var(\phi_2)|}$$

More generally, our results concerning queries and transformations of the KC map are summarized in Proposition 1. Languages d-DNNF and $\texttt{OBDD}_<$ (which are not subsets of EADT) are reported for the comparison matter.

[1] The affine decomposability condition can be given up for EDT formulae since every EDT formula can be translated in linear time into an equivalent EDT formula which is read-once (i.e., for every path from the root of the tree to a leaf, the list of all variables labeling the decision nodes of the path contains at most one occurrence of each variable).

$\mathcal{L}$	CO	VA	CE	IM	EQ	SE	CT	ME
EADT	√	√	√	√	?	∘	√	√
EDT	√	√	√	√	?	∘	√	√
ADT	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√
ODT$_<$	√	√	√	√	√	√	√	√
d-DNNF	√	√	√	√	?	∘	√	√
OBDD$_<$	√	√	√	√	√	√	√	√

Table 1: Queries. √ means "satisfies" and ∘ means "does not satisfy unless P = NP."

$\mathcal{L}$	CD	FO	SFO	$\wedge$C	$\wedge$BC	$\vee$C	$\vee$BC	$\neg$C
EADT	√	∘	∘	∘	∘	∘	∘	√
EDT	√	∘	∘	∘	∘	∘	∘	√
ADT	√	∘	√	∘	√	∘	√	√
DT	√	∘	√	∘	√	∘	√	√
ODT$_<$	√	∘	√	∘	√	∘	√	√
d-DNNF	√	∘	∘	∘	∘	∘	∘	?
OBDD$_<$	√	∘	√	∘	√	∘	√	√

Table 2: Transformations. √ means "satisfies," while ∘ means "does not satisfy unless P = NP".

Proposition 1 *The results given in Tables 1 and 2 hold.*

In a nutshell, ADT and its subclass DT are equivalent to OBDD$_<$ with respect to queries and transformations. Similarly, EADT and its subclass EDT are essentially equivalent to d-DNNF with respect to queries and transformations. In particular, EDT, EADT, and d-DNNF do not satisfy **SE** unless P = NP, and it is unknown whether they satisfy **EQ**. It is also unknown whether d-DNNF satisfies $\neg$**C**, but this transformation can be done in linear time for both EADT and EDT.

The inclusion graph of the different languages is given in Figure 2. In light of this inclusion graph and the fact that DT (resp. EDT) does not satisfy more queries or transformations than ADT (resp. EADT), it follows that DT (resp. EDT) cannot prove a better choice than ADT (resp. EADT) from the KC point of view. This is why we focus on ADT and EADT in the following. For these languages, we obtained the following succinctness results:

Proposition 2 CNF $\not\leq_s$ ADT, DNF $\not\leq_s$ ADT, OBDD$_<$ $\not\leq_s$ ADT, d-DNNF$_T$ $\not\leq_s$ ADT, *and* EADT $<_s$ ADT.

Based on these results, it turns out that OBDD$_<$ does not dominate ADT from the KC point of view (i.e., it does not offer any query/transformation not supported by ADT, and is not strictly more succinct than ADT). This, together with the fact that ADT $\subseteq$ EADT implies that OBDD$_<$ does not dominate EADT. Our succinctness results also reveal that none of the "flat" languages CNF and DNF is at least as succinct as any of ADT or EADT. Although we ignore how d-DNNF and EADT compare w.r.t. succinctness, we know that the subclass d-DNNF$_T$ does not dominate any of ADT or EADT.

4 A CNF-to-EADT Compiler

A natural approach for compiling an arbitrary propositional formula into an EADT formula is to exploit a generalized form of Shannon expansion. Given two formulae Δ and δ, and a variable x, we denote by $\Delta\mid_{x\leftarrow\delta}$ the formula obtained by replacing every occurrence of x in Δ by δ. With this notation in hand, the *generalized Shannon expansion* states that for every propositional formulae Δ and δ and every variable x, we have:

$$\Delta \equiv ((x \Leftrightarrow \neg\delta) \wedge \Delta\mid_{x\leftarrow\neg\delta}) \vee ((x \Leftrightarrow \delta) \wedge \Delta\mid_{x\leftarrow\delta})$$

Observe that the standard expansion introduced by Shannon [1949] is recovered by considering $\delta = \top$. The validity of the generalized expansion comes from the fact that Δ is equivalent to $((x \Leftrightarrow \neg\delta) \wedge \Delta) \vee ((x \Leftrightarrow \delta) \wedge \Delta)$, and the fact that, for every propositional formula δ (or its negation), the expression $(x \Leftrightarrow \delta) \wedge \Delta$ is equivalent to $(x \Leftrightarrow \delta) \wedge \Delta\mid_{x\leftarrow\delta}$.

In our setting, Δ is an ECNF formula and δ is an affine clause. ECNF, the language of extended CNF, is the set of all finite conjunctions of extended clauses, where an extended clause is a finite disjunction of affine clauses. Thus, for instance, $x_1 \vee (x_2 \oplus x_3 \oplus \top) \vee (x_1 \oplus x_3)$ is an extended clause. Clearly, CNF is linearly translatable into ECNF Now, since $x \Leftrightarrow \neg\delta$ is equivalent to $x \oplus \delta$, and $x \Leftrightarrow \delta$ is equivalent to $x \oplus \delta \oplus \top$, the generalized Shannon expansion can be restated as the following branching rule:

$$\Delta \equiv ((x \oplus \delta) \wedge \Delta\mid_{x\leftarrow\delta\oplus\top}) \vee ((x \oplus \delta \oplus \top) \wedge \Delta\mid_{x\leftarrow\delta})$$

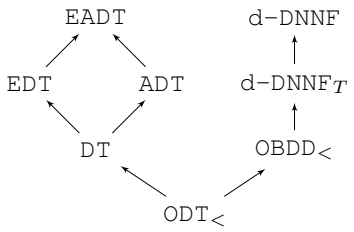


Figure 2: Inclusion graph. $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ indicates that $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

The Compilation Algorithm. Based on previous considerations, Algorithm 1 provides the pseudo-code for the compiler eadt, which takes as input an ECNF formula Δ, and returns as output an EADT formula equivalent to Δ. The first two lines deal with the specific cases where Δ is valid or unsatisfiable. In both cases, the corresponding leaf is returned.

We note in passing that the unsatisfiability problem (resp. the validity problem) for ECNF has the same complexity as for its subset CNF, i.e., it is coNP-complete (resp. it is in P). This is obvious for the unsatisfiability problem. For the validity problem, an ECNF formula is valid iff every extended clause in it is valid, and an extended clause $\delta_1 \vee \ldots \vee \delta_k$ (where each δ_i, $i \in 1, \ldots, k$ is an affine clause) is valid iff the affine formula $(\delta_1 \oplus \top) \wedge \ldots \wedge (\delta_k \oplus \top)$ is contradictory, which can be tested in polynomial time.

In the remaining case, Δ is split into a decomposable conjunction of components $\Delta_1, \cdots, \Delta_k$ (Line 3). These components are recursively compiled into EADT formulae and conjoined as a $\wedge$ node using the aNode function. Note that decomposition takes precedence over branching: only when Δ consists of a single component, the compiler chooses an affine clause $x \oplus \delta$ for which all variables occur in Δ (Line 5), and then branches on this clause using the generalized Shannon

Algorithm 1: $\mathsf{eadt}(\Delta)$

input : an ECNF formula Δ
output: an EADT formula equivalent to Δ

1 if $\Delta \equiv \top$ then return $\mathsf{leaf}(\top)$
2 if $\Delta \equiv \bot$ then return $\mathsf{leaf}(\bot)$
3 let $\Delta_1, \cdots, \Delta_k$ be the connected components of Δ
4 if $k > 1$ then return $\mathsf{aNode}(\mathsf{eadt}(\Delta_1), \cdots, \mathsf{eadt}(\Delta_k))$
5 choose a simplified affine clause $x \oplus \delta$ such that

$$var(x \oplus \delta) \subseteq Var(\Delta)$$

6 return $\mathsf{dNode}(x \oplus \delta, \mathsf{eadt}(\Delta\mid_{x \leftarrow \delta \oplus \top}), \mathsf{eadt}(\Delta\mid_{x \leftarrow \delta}))$

expansion (Line 6). Here, the dNode function returns a decision node labeled with the first argument, having the second argument as left child, and having the third argument as right child. When Line 3 is omitted, the CNF-to-EADT compiler boils down to a CNF-to-ADT compiler.

Algorithm 1 is guaranteed to terminate. Indeed, by definition of a simplified affine clause, δ in $x \oplus \delta$ is an affine clause which does not contain x. Since none of $\Delta\mid_{x \leftarrow \delta \oplus \top}$ and $\Delta\mid_{x \leftarrow \delta}$ contains x, the steps 5 and 6 can be applied only a finite number of times. Furthermore, the EADT formula returned by the algorithm is guaranteed to satisfy the affine decomposition rule. This property can be proved by induction on the structure of the resulting tree: the only non-trivial case is when the tree consists of a $\wedge$ node with parents N and children $N_1, \cdots, N_k$ each N_i formed by calling eadt on the connected component Δ_i of the formula Δ. Since each parent clause in N is of the form $x \oplus \delta$, where x is excluded from δ, and since the components do not share any variable, it follows that $x \oplus \delta$ overlaps with at most one component in $\Delta_1, \cdots, \Delta_k$. This, together with the fact that the generalized Shannon expansion is valid, establishes the correctness of the algorithm.

Implementation. Algorithm 1 was implemented on top of the state-of-the-art SAT solver MiniSAT [Eén and Sörensson, 2003]. We extended MiniSAT to deal with ECNF formulae. The heuristic used at Line 5 for choosing affine clauses of the form $x \oplus \delta$ is based on the concept of variable activity (VSIDS, Variable State Independent Decaying Sum) [Moskewicz *et al.*, 2001]. Specifically, for each extended clause C of Δ, the score of C is computed as the sum of the scores of each affine clause in it, where the score of an affine clause is the the sum of the VSIDS scores of its variables. Based on this metric, an extended clause C of Δ of maximal score is selected, and the variables of C are sorted by decreasing VSIDS score; the selected variable x is the first variable in the resulting list, and the affine clause δ is formed by the next $k-1$ variables in the list. Note that selecting all the variables of $x \oplus \delta$ from the same extended clause C of Δ prevents us from generating connections between variables which are not already connected in the constraint graph of Δ. We also took advantage of a simple filtering method, which consists in finding implied affine clauses, used only at the first top nodes of the search tree. In our experiments, we bounded the size of affine clauses to $k = 2$, and used the filtering method up to depth 5.

5 Experiments

Setup. The empirical protocol we followed is very close to the one conducted in [Schrag, 1996] (and other papers). We have considered a number of CNF benchmark instances from different domains provided by the SAT LIBrary (www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html). For each CNF instance Δ, we generated 1000 queries; each query is a 3-literal term γ the 3 variables of which are picked up at random from the set of variables of Δ, following a uniform distribution; the sign of each literal is also selected at random with probability $\frac{1}{2}$. The objective is to count the number of models of the conditioned formula $\Delta \mid \gamma$ for all queries γ. Our experiments have been conducted on a Quad-core Intel XEON X5550 with 32Gb of memory. A time-out of 3 hours has been considered for the off-line compilation phase, and a time-out of 3 hours per query has been established for addressing each of the 1000 queries during the on-line phase. Based on this setup, three approaches have been examined:

- A direct, uncompiled approach: we considered a state-of-the-art model counter, namely Cachet (www.cs.rochester.edu/~kautz/Cachet/index.htm) [Sang *et al.*, 2004]. Here, $\#F_{\mathsf{Cachet}}$ is the number of elements of F_{Cachet}, the set of "feasible" queries, i.e., the queries in the sample for which Cachet has been able to terminate before the time-out (or a segmentation fault). $\overline{Q}_{\mathsf{Cachet}}$ gives the mean time needed to address the feasible queries, i.e.,

$$\overline{Q}_{\mathsf{Cachet}} = \frac{1}{\#F_{\mathsf{Cachet}}} \sum_{\gamma \in F_{\mathsf{Cachet}}} Q_{\mathsf{Cachet}}(\Delta|\gamma)$$

  where $Q_{\mathsf{Cachet}}(\Delta|\gamma)$ is the runtime of Cachet for $\Delta|\gamma$.

- Two compilation-based approaches: d-DNNF and EADT have been targeted. We took advantage of the c2d compiler (reasoning.cs.ucla.edu/c2d/) to generate (smooth) d-DNNF compilations,[2] and our own compiler to compute EADT compiled forms. For each $\mathcal{L}$ among d-DNNF and EADT, Δ has been first turned into a compiled form $\Delta^* \in \mathcal{L}$ during an off-line phase. The compilation time $C_{\mathcal{L}}$ needed to compute Δ^* and the mean query-answering time $\overline{Q}_{\mathcal{L}}$ have been measured. We also computed for each approach two ratios:

$$\alpha_{\mathcal{L}} = \frac{\overline{Q}_{\mathcal{L}}}{\overline{Q}_{\mathsf{Cachet}}} \text{ and } \beta_{\mathcal{L}} = \left\lceil \frac{C_{\mathcal{L}}}{\overline{Q}_{\mathsf{Cachet}} - \overline{Q}_{\mathcal{L}}} \right\rceil$$

  Intuitively, $\alpha_{\mathcal{L}}$ indicates how much on-line time improvement is got from compilation: the lower the better. The quantity $\beta_{\mathcal{L}}$ captures the number of queries needed to amortize compilation time. Clearly, the compilation-based approach targeting $\mathcal{L}$ is useful only if $\alpha_{\mathcal{L}} < 1$. By convention, $\beta_{\mathcal{L}} = +\infty$ when $\alpha_{\mathcal{L}} \geq 1$.

[2] Primarily, we also planned to use the d-DNNF compiler Dsharp [Muise *et al.*, 2012] but unfortunately, we encountered the same problems as mentioned in [Voronov, 2013] to run it, which prevented us from doing it.

Instance			Cachet		c2d				eadt			
name	$\#var$	$\#cla$	$\#F$	$\overline{Q}$	C	$\overline{Q}$	α	β	C	$\overline{Q}$	α	β
ais6	61	581	1000	0.531	1.23	$4\,\mathrm{E}{-5}$	$8\,\mathrm{E}{-7}$	2	0.01	$< 1\,\mathrm{E}{-7}$	$< 2\,\mathrm{E}{-7}$	**1**
ais8	113	1520	866	0.540	3.04	$2\,\mathrm{E}{-4}$	$3\,\mathrm{E}{-4}$	5	0.24	$1\,\mathrm{E}{-5}$	$2\,\mathrm{E}{-5}$	**1**
ais10	181	3151	325	0.578	12.3	$1\,\mathrm{E}{-3}$	$2\,\mathrm{E}{-3}$	21	7.69	$1\,\mathrm{E}{-4}$	$2\,\mathrm{E}{-4}$	**13**
ais12	265	5666	80	0.573	-	-	-	-	410	$1\,\mathrm{E}{-3}$	$2\,\mathrm{E}{-3}$	**717**
bmc-ibm-2	2810	11683	1000	0.569	-	-	-	-	0.37	$2\,\mathrm{E}{-5}$	$4\,\mathrm{E}{-5}$	**1**
bmc-ibm-3	14930	72106	999	13.04	412	0.93	$7\,\mathrm{E}{-2}$	34	180	$6\,\mathrm{E}{-3}$	$5\,\mathrm{E}{-4}$	**13**
bmc-ibm-4	28161	139716	1000	5.412	1128	9.09	1.679	$+\infty$	-	-	-	-
bw_large.a	459	4675	1000	0.537	15.05	$2\,\mathrm{E}{-5}$	$4\,\mathrm{E}{-5}$	28	$< 1\,\mathrm{E}{-4}$	$< 1\,\mathrm{E}{-7}$	$< 2\,\mathrm{E}{-7}$	**1**
bw_large.b	1087	13772	1000	0.612	48.88	$5\,\mathrm{E}{-5}$	$8\,\mathrm{E}{-5}$	79	0.01	$< 1\,\mathrm{E}{-7}$	$< 2\,\mathrm{E}{-7}$	**1**
bw_large.c	3016	50457	996	2.452	283.3	$1\,\mathrm{E}{-4}$	$6\,\mathrm{E}{-5}$	115	0.16	$1\,\mathrm{E}{-5}$	$4\,\mathrm{E}{-6}$	**1**
bw_large.d	6325	131973	896	31.44	-	-	-	-	1.9	$2\,\mathrm{E}{-5}$	$6\,\mathrm{E}{-7}$	**1**
(bw) medium	116	953	1000	0.526	2.39	$2\,\mathrm{E}{-5}$	$4\,\mathrm{E}{-5}$	4	$< 1\,\mathrm{E}{-4}$	$< 1\,\mathrm{E}{-7}$	$< 2\,\mathrm{E}{-7}$	**1**
(bw) huge	459	7054	1000	0.543	15.11	$2\,\mathrm{E}{-5}$	$4\,\mathrm{E}{-5}$	27	$< 1\,\mathrm{E}{-4}$	$< 1\,\mathrm{E}{-7}$	$< 2\,\mathrm{E}{-7}$	**1**
hanoi4	718	4934	505	0.557	559.6	$3\,\mathrm{E}{-5}$	$5\,\mathrm{E}{-5}$	1004	0.13	$< 1\,\mathrm{E}{-7}$	$< 2\,\mathrm{E}{-7}$	**1**
hanoi5	1931	14468	440	0.619	2240	$8\,\mathrm{E}{-5}$	$1\,\mathrm{E}{-4}$	3621	1.1	$1\,\mathrm{E}{-5}$	$2\,\mathrm{E}{-5}$	**1**
logistics.a	828	6718	993	1.266	-	-	-	-	6757	2.12	1.676	$+\infty$
ssa7552-038	1501	3575	1000	0.634	20.99	0.042	0.065	**35**	-	-	-	-

Table 3: Some experimental results

Results. Table 3 presents the obtained results. Each line corresponds to a CNF instance Δ identified by the leftmost column. The first two columns give respectively the number $\#var$ of variables of Δ and the number $\#cla$ of clauses of Δ, and the remaining columns give the measured values. The reported computation times are in seconds.

We can observe that both compilation-based approaches typically prove valuable whenever the off-line compilation phase terminates. On the one hand, for each compilation-based approach $\mathcal{L}$, all the 1000 queries have been "feasible" when the compilation process terminated in due time. For this reason, we did not report in the table the number $\#F_{\mathcal{L}}$ of feasible queries. By contrast, the number of feasible queries for the direct, uncompiled approach is sometimes significantly lower than 1000, and the standard deviation of the on-line query-answering time (not given in the table for space reasons) for such queries is often significantly greater than the corresponding deviations measured from compilation-based approaches. On the other hand, the number of queries β to be considered for balancing the compilation time is finite for all, but two (one for d−DNNF and one for EADT), instances.

Furthermore, the experiments revealed that some instances of significant size are compilable. When the compilation succeeds, β is typically small and, accordingly, on-line time savings of several orders of magnitude can be achieved. Especially, the optimal value 1 for the "break-even" point β has been reached for many instances when the EADT language was targeted. This means that in many cases the off-line time spend to built the EADT compiled form is immediately balanced by the first counting query. Stated otherwise, the eadt compiler proves also competitive as a model counter.

Finally, our experiments show EADT compilation challenging with respect to d−DNNF compilation in many (but not all) cases. When compilation succeeds in both cases, the number of nodes in EADT and d−DNNF formulae are about the same order, but the EADT formulae are slightly faster for processing queries, due to their arborescent structure.

6 Conclusion

The propositional language EADT introduced in the paper appears as quite appealing for the representation purpose, when **CT** is a key query. Especially, EADT offers the same queries as d−DNNF, and more transformations (among those considered in the KC map). The subset ADT of EADT offers all the queries and the same transformations as those satisfied by the influential $\mathtt{OBDD}_<$ language. Furthermore, $\mathtt{OBDD}_<$ is not at least as succinct as ADT, which shows ADT as a possible challenger to $\mathtt{OBDD}_<$. In practice, the EADT compilation-based approach to model counting appears as competitive with the model counter Cachet and the d−DNNF compilation-based approach to model counting.

This work opens a number of perspectives for further research. From the theoretical side, a natural extension of ADT is the set of all single-rooted finite DAGs, where leaves are labeled by a Boolean constant ($\top$ or $\bot$), and internal nodes are affine decision nodes. However, this language is not appealing as a target language for knowledge compilation, because it contains the language BDD of binary decision diagrams (alias branching programs) [Bryant, 1986] as a subset and BDD does not offer *any* query from the KC map [Darwiche and Marquis, 2002], unless P = NP. Thus, the problem of finding interesting classes of affine decision graphs that are tractable for model counting looks stimulating.

From the practical side, there are many ways to improve our compiler. Notably, it would be interesting to take advantage of preprocessing techniques [Piette *et al.*, 2008; Järvisalo *et al.*, 2012] in order to simplify the input CNF formulae before compiling them. Furthermore, it could prove useful to exploit Gaussian elimination for handling more efficiently (see e.g. [Li, 2003; Chen, 2007; Soos *et al.*, 2009]) instances that contain subproblems corresponding to affine formulae, like those reported in [Crawford and Kearns, 1995; Cannière, 2006]. Finally, considering other heuristics for selecting the branching affine clauses (e.g. criteria based on the mutual information metric) could also prove valuable.

References

[Bacchus *et al.*, 2003] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Proc. of FOCS'03*, pages 340–351, 2003.

[Bordeaux *et al.*, 2012] L. Bordeaux, M. Janota, J. P. Marques Silva, and P. Marquis. On unit-refutation complete formulae with existentially quantified variables. In *Proc. of KR'12*, 2012.

[Bryant, 1986] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[Cannière, 2006] Ch. De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *Proc. of ISC'06*, pages 171–186, 2006.

[Chen, 2007] J. Chen. XORSAT: An efficient algorithm for the dimacs 32-bit parity problem. *CoRR*, abs/cs/0703006, 2007.

[Crawford and Kearns, 1995] J.M. Crawford and M.J. Kearns. The minimal disagreement parity problem as a hard satisfiability problem. Technical report, 1995.

[Darwiche and Marquis, 2002] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Darwiche, 2009] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[Darwiche, 2011] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI'11*, pages 819–826, 2011.

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. of SAT'03*, pages 502–518. 2003.

[Fargier and Marquis, 2008] H. Fargier and P. Marquis. Extending the knowledge compilation map: Krom, Horn, affine and beyond. In *Proc. of AAAI'08*, pages 442–447, 2008.

[Gergov and Meinel, 1994] J. Gergov and C. Meinel. Efficient analysis and manipulation of OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43(10):1197–1209, 1994.

[Järvisalo *et al.*, 2012] M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. In *Proc. of IJCAR'12*, pages 355–370, 2012.

[Li, 2003] C. Li. Equivalent literal propagation in the dll procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.

[Littman *et al.*, 2001] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.

[Marquis, 2011] P. Marquis. Existential closures for knowledge compilation. In *Proc. of IJCAI'11*, pages 996–1001, 2011.

[Mateescu *et al.*, 2008] R. Mateescu, R. Dechter, and R. Marinescu. AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. *Journal of Artificial Intelligence Research*, 33:465–519, 2008.

[Moskewicz *et al.*, 2001] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.

[Muise *et al.*, 2012] Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

[Piette *et al.*, 2008] C. Piette, Y. Hamadi, and L. Saïs. Vivifying propositional clausal formulae. In *Proc. of ECAI'08*, pages 525–529, 2008.

[Pipatsrisawat and Darwiche, 2008] K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In *Proc. of AAAI'08*, pages 517–522, 2008.

[Roth, 1996] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, pages 475–482, 2005.

[Schaefer, 1978] Th. J. Schaefer. The complexity of satisfiability problems. In *Proc. of STOC'78*, pages 216–226, 1978.

[Schrag, 1996] R. Schrag. Compilation for critically constrained knowledge bases. In *Proc. of AAAI'96*, pages 510–515, 1996.

[Shannon, 1949] C.E. Shannon. The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[Soos *et al.*, 2009] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT'09*, pages 244–257, 2009.

[Subbarayan *et al.*, 2007] S. Subbarayan, L. Bordeaux, and Y. Hamadi. Knowledge compilation properties of tree-of-BDDs. In *Proc. of AAAI'07*, pages 502–507, 2007.

[Valiant, 1979] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Voronov, 2013] A. Voronov. *On Formal Methods for Large-Scale Product Configuration*. Ph.D. thesis, Chalmers University, 2013.

[Wachter and Haenni, 2006] M. Wachter and R. Haenni. Propositional DAGs: A new graph-based language for representing Boolean functions. In *Proc. of KR'06*, pages 277–285, 2006.

Chapitre 10

A SAT-Based Approach for Solving the Modal Logic S5-Satisfiability Problem

Article paru dans les actes de « *the 31st AAAI Conference on Artificial Intelligence* » (AAAI'17), pages 3864 – 3870, février 2017.

Co-écrit avec Thomas Caridroit, Daniel Le Berre, Tiago de Lima et Valentin Montmirail.

Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)

A SAT-Based Approach for Solving the Modal Logic S5-Satisfiability Problem

Thomas Caridroit, Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima and **Valentin Montmirail**

CRIL, Univ. Artois and CNRS, F62300 Lens, France
{caridroit,lagniez,leberre,delima,montmirail}@cril.fr

Abstract

We present a SAT-based approach for solving the modal logic S5-satisfiability problem. That problem being NP-complete, the translation into SAT is not a surprise. Our contribution is to greatly reduce the number of propositional variables and clauses required to encode the problem. We first present a syntactic property called diamond degree. We show that the size of an S5-model satisfying a formula ϕ can be bounded by its diamond degree. Such measure can thus be used as an upper bound for generating a SAT encoding for the S5-satisfiability of that formula. We also propose a lightweight caching system which allows us to further reduce the size of the propositional formula. We implemented a generic SAT-based approach within the modal logic S5 solver S52SAT. It allowed us to compare experimentally our new upper-bound against previously known one, i.e. the number of modalities of ϕ and to evaluate the effect of our caching technique. We also compared our solver against existing modal logic S5 solvers. The proposed approach outperforms previous ones on the benchmarks used. These promising results open interesting research directions for the practical resolution of others modal logics (e.g. K, KT, S4)

Introduction

Over the last twenty years, modal logics have been used in various areas of artificial intelligence like formal verification (Fairtlough and Mendler 1994), game theory (Lorini and Schwarzentruber 2010), database theory (Fitting 2000) and distributed computing (VII, Crary, and Harper 2005) for example. More recently, the modal logic S5 was used for contingent planning (Niveau and Zanuttini 2016) and in knowledge compilation (Bienvenu, Fargier, and Marquis 2010). For this reason, automated reasoning in modal logics has been vastly studied (e.g. (Sebastiani and Villafiorita 1998; Masssacci 2000; Sebastiani and Vescovi 2009)).

Ladner (Ladner 1977; Fagin et al. 1995) showed that the satisfiability problem for several modal logics including K, KT and S4 is PSPACE-Complete while it is NP-Complete for S5 (see (Halpern and Rêgo 2007) for more details). Since SAT solvers became a quite efficient practical NP-oracle for many problems, we are interested in studying SAT encoding for the S5-SAT problem from a practical perspective. Using a SAT-oracle in the context of modal logic is not new: a comprehensive overview of previous works can be found for example in (Sebastiani and Tacchella 2009). However, most of them tackle the satisfiability of modal logic K. *SAT (Giunchiglia, Giunchiglia, and Tacchella 1999; Giunchiglia and Tacchella 2000; Giunchiglia et al. 2000; Giunchiglia, Tacchella, and Giunchiglia 2002) uses a SAT-oracle to decide the satisfiability of 8 different modal logics, including K, but not S5. In the same spirit as our work, a translation of modal logic K to SAT has been proposed in Km2SAT (Giunchiglia and Sebastiani 2000; Sebastiani and Vescovi 2009). More recently, the solver InKreSAT (Kaminski and Tebbi 2013) proposed an innovative SAT-based system where the SAT solver drives the development of a tableaux method. Theoretically, an approach based on Satisfiability Modulo Theory (Areces, Fontaine, and Merz 2015) has also been proposed. None of those methods are applicable to modal logic S5.

The number of variables required to reduce S5-SAT to SAT depends on an upper bound of the number of possible worlds to be considered in the S5-model. Minimizing that upper bound is thus crucial to obtain CNF formulas of reasonable size. We propose a new upper bound based on a syntactical property of the formula, that we call "diamond degree". We provide some experimental evidences that our approach improves significantly the state-of-the-art S5 solvers.

The remainder of the paper is organized as follows: we first present the modal logic S5 and the different bounds on the size of a S5-models; then we detail the reduction from S5-SAT to SAT, parameterized by the number of worlds to consider. We present two improvements to reduce the size of the SAT encoding: a better upper bound on the number of worlds to consider and structural caching. Finally, we compare experimentally the efficiency of our approach to state-of-the-art S5 solvers.

Preliminaries

Let $\mathbb{P}$ be a finite non-empty set of propositional variables. The language $\mathcal{L}$ of the modal logic S5 is the set of formulas ϕ defined by the following grammar in BNF:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \Box\phi \mid \Diamond\phi$$

where p ranges over $\mathbb{P}$. A formula of the form $\Box\phi$ (*box phi*) means ϕ is necessarily true. A formula of the form $\Diamond\phi$ (*di-*

amond phi) means ϕ is possibly true. Any formula $\phi \in \mathcal{L}$ may be translated to an equivalent formula in negation normal form (NNF), that is, negations appear only in front of propositional variables (see the definition in (Robinson and Voronkov 2001)), noted $\text{nnf}(\phi)$. This can be done in polynomial time. Formulas in $\mathcal{L}$ are interpreted using pointed S5-models. A S5-model is a pair (W, V), where W is a non-empty set of possible worlds and V is a valuation function with signature $W \to (\mathbb{P} \to \{0, 1\})$. A pointed S5-model is a triplet (W, V, w), where (W, V) is a S5-model and $w \in W$. The satisfaction relation $\models$ between formulas in $\mathcal{L}$ and pointed S5-models is recursively defined as follows:

$(W, V, w) \models \top$
$(W, V, w) \models p$ iff $V(w)(p) = 1$
$(W, V, w) \models \neg\phi$ iff $(W, V, w) \not\models \phi$
$(W, V, w) \models \phi_1 \wedge \phi_2$ iff $(W, V, w) \models \phi_1$ and $(W, V, w) \models \phi_2$
$(W, V, w) \models \phi_1 \vee \phi_2$ iff $(W, V, w) \models \phi_1$ or $(W, V, w) \models \phi_2$
$(W, V, w) \models \Box\phi$ iff for all $w' \in W$ we have $(W, V, w') \models \phi$
$(W, V, w) \models \Diamond\phi$ iff there is $w' \in W$ s.t. $(W, V, w') \models \phi$

Validity and satisfiability are defined as usual. A formula $\phi \in \mathcal{L}$ is valid, noted $\models \phi$, if and only if, for all pointed S5-models (W, V, w), we have $(W, V, w) \models \phi$. Moreover, ϕ is satisfiable if and only if $\not\models \neg\phi$.

From S5-SAT to SAT

It was shown in (Ladner 1977) that if an S5 formula ϕ with n modal connectives is satisfiable, then there is an S5-model satisfying ϕ with at most $n + 1$ worlds. As a consequence, we know that there exists an algorithm running in polynomial time able to transform the S5-SAT problem into the SAT problem. However, to the best of our knowledge, no one evaluated this approach in practice. It has not been compared against state-of-the-art solvers until now. Yet SAT-based approaches are known to be quite efficient in practice.

A SAT encoding is represented here by a translation function tr, which takes as input an S5-formula ϕ and a number of worlds n in the S5-model and produces a propositional formula. This is inspired by the standard translation to FOL (Patrick Blackburn and Wolter 2007). Note that the accessibility relation does not need to be represented in an S5-model, because it is an equivalence relation.

$$\text{tr}(\phi, n) = \text{tr}'(\text{nnf}(\phi), 1, n)$$
$$\text{tr}'(\top, i, n) = \top \qquad \text{tr}'(\neg\top, i, n) = \neg\top$$
$$\text{tr}'(p, i, n) = p_i \qquad \text{tr}'(\neg p, i, n) = \neg p_i$$
$$\text{tr}'((\phi \wedge \cdots \wedge \delta), i, n) = \text{tr}'(\phi, i, n) \wedge \cdots \wedge \text{tr}'(\delta, i, n)$$
$$\text{tr}'((\phi \vee \cdots \vee \delta), i, n) = \text{tr}'(\phi, i, n) \vee \cdots \vee \text{tr}'(\delta, i, n)$$
$$\text{tr}'(\Box\phi, i, n) = \bigwedge_{j=1}^{n} (\text{tr}'(\phi, j, n))$$
$$\text{tr}'(\Diamond\phi, i, n) = \bigvee_{j=1}^{n} (\text{tr}'(\phi, j, n))$$

The translation adds fresh Boolean variables p_i to the formula, denoting the truth value of p in the world w_i. In such function, the i parameter represents the index of the world. The function is defined over a Negative Normal Form (NNF) formula for sake of simplicity.

Example 1. *Let $\phi = \Diamond(a \wedge \Box b)$, Figure 1 shows the effect of applying* tr *to the formula ϕ with $n = 2$.*

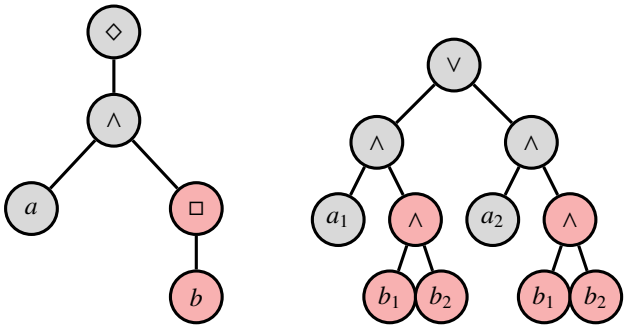


Figure 1: From S5 ϕ (left) to propositional logic with n=2 (right).

If the value of n is an upper-bound of the number of worlds in the model, then the translation is equi-satisfiable to the original S5 formula. In particular, it is the case for the upper-bound shown in (Ladner 1977):

Definition 1. *Let ϕ be in $\mathcal{L}$, $\mathtt{nm}(\phi)$ denotes the number of modal connectives in the formula ϕ.*

Theorem 1. *$\phi \in \mathcal{L}$ is satisfiable if and only if $tr(\phi, \mathtt{nm}(\phi)+1)$ is satisfiable.*

Proof. Theorem 1 is proved in the same way as the standard translation to FOL plus Lemma 6.1 in (Ladner 1977). □

Note that the result of the translation is not in CNF. As such, classical translation into CNF such as (Tseitin 1983) is needed to use a SAT oracle.

Improvements on the upper-bound

The size of the encoding depends on $\mathtt{nm}(\phi)$. In practice, such upper bound produces unreasonably large formulas. As such, a first step is to improve that upper bound. We propose to use a new metric called the diamond degree as a new upper bound.

Definition 2 (Diamond-Degree)**.** *The diamond degree of $\phi \in \mathcal{L}$, noted $\mathtt{dd}(\phi)$, is defined recursively, as follows:*

$$\mathtt{dd}(\phi) = \mathtt{dd}'(\text{nnf}(\phi))$$
$$\mathtt{dd}'(\top) = \mathtt{dd}'(\neg\top) = \mathtt{dd}'(p) = \mathtt{dd}'(\neg p) = 0$$
$$\mathtt{dd}'(\phi \wedge \psi) = \mathtt{dd}'(\phi) + \mathtt{dd}'(\psi)$$
$$\mathtt{dd}'(\phi \vee \psi) = \max(\mathtt{dd}'(\phi), \mathtt{dd}'(\psi))$$
$$\mathtt{dd}'(\Box\phi) = \mathtt{dd}'(\phi) \qquad \mathtt{dd}'(\Diamond\phi) = 1 + \mathtt{dd}'(\phi)$$

The computation of the diamond degree requires to convert ϕ in NNF. As mentioned in the previous section, this operation can always be performed in both polynomial time and space. Then, the diamond degree of any formula can be

computed in polynomial time. Without loss of generality, we will assume that $\phi \in$ NNF and consider dd' instead of dd.

Informally, the diamond degree represents an upper bound of the number of diamonds to be taken into account to satisfy the formula. To show that the diamond degree is a valid upper bound for our SAT encoding, we will use a tableau method. Therefore, we need some additional definitions. Let ϕ be a formula in NNF and let sub(ϕ) be the set of all sub-formulas of ϕ. A tableau for ϕ is a non-empty set $T = \{s_0, s_1, \ldots, s_n\}$ such that each $s_i \in T$ is a subset of sub(ϕ) and $\phi \in s_0$. In addition, each set $s_i \in T$ satisfies the following conditions:

1. $\neg\top \notin s$.
2. if $p \in s$ then $\neg p \notin s$.
3. if $\neg p \in s$ then $p \notin s$.
4. if $\psi_1 \wedge \psi_2 \in s$ then $\psi_1 \in s$ and $\psi_2 \in s$.
5. if $\psi_1 \vee \psi_2 \in s$ then $\psi_1 \in s$ or $\psi_2 \in s$.
6. if $\Box\psi_1 \in s$ then $\forall s' \in T$ we have $\psi_1 \in s'$.
7. if $\Diamond\psi_1 \in s$ then $\exists s' \in T$ s.t. $\psi_1 \in s$.

Lemma 1. *Let ϕ be in NNF. There is a tableau for ϕ if and only if ϕ is satisfiable.*

Proof. Direct from the definition of the NNF (which preserves the satisfiability) and from the definition of the tableau method (there will be a tableau for ϕ if and only if ϕ is satisfiable). □

Lemma 2. *Let ϕ be in NNF. The number of elements of the set T created by constructing its tableau is bounded by $\mathsf{dd}'(\phi) + 1$.*

Proof. Let $\psi \in$ sub(ϕ). Let $g(\psi)$ be the number of sets s added to T because of ψ. That is, $g(\psi)$ is the number of times the condition involving operator $\Diamond$ is triggered for sub-formulas of ψ. We show that, for all $\psi \in$ sub(ϕ) we have $g(\psi) \leq \mathsf{dd}'(\psi)$. We do so by induction on the structure of ψ.

Induction base. We consider four cases: (1) $\psi = \top$, (2) $\psi = \neg\top$, (3) $\psi = p$ and (4) $\psi = \neg p$. In all cases, the condition involving $\Diamond$ will never be triggered for formulas in sub(ψ). Then $g(\psi) = 0 \leq \mathsf{dd}'(\psi)$.

Induction step. We consider four cases:

1. $\psi = \psi_1 \wedge \psi_2$. Assume $\psi \in s$, for some $s \in T$. In this case, the algorithm adds ψ_1 and ψ_2 to s. Therefore, $g(\psi)$ is bounded by $g(\psi_1) + g(\psi_2)$. The latter is bounded by $\mathsf{dd}'(\psi_1) + \mathsf{dd}'(\psi_2)$ (by the induction hypothesis). Then $g(\psi) \leq \mathsf{dd}'(\psi)$.
2. $\psi = \psi_1 \vee \psi_2$. Assume $\psi \in s$, for some $s \in T$. In this case, the algorithm adds either ψ_1 or ψ_2 to s. Therefore, $g(\phi)$ is bounded by $\max(g(\psi_1), g(\psi_2))$. The latter is bounded by $\max(\mathsf{dd}'(\psi_1), \mathsf{dd}'(\psi_2))$ (by the induction hypothesis). Then $g(\psi) \leq \mathsf{dd}'(\psi)$.
3. $\psi = \Box\psi_1$. Assume $\psi \in s$, for some $s \in T$. In this case, the algorithm adds ψ_1 to all $s' \in T$. Therefore, $g(\psi)$ is bounded by $g(\psi_1)$. The latter is bounded by $\mathsf{dd}'(\psi_1)$ (by the induction hypothesis). Then $g(\psi) \leq \mathsf{dd}'(\psi)$.
4. $\psi = \Diamond\psi_1$. Assume $\psi \in s$, for some $s \in T$. In this case, if there is no s' containing ψ_1 then the algorithm adds a new s'' to T and adds ψ_1 to s''. Therefore, $g(\psi)$ is bounded by $1 + g(\psi_1)$. The latter is bounded by $1 + \mathsf{dd}'(\psi_1)$ (by the induction hypothesis). Then $g(\psi) \leq \mathsf{dd}'(\psi)$.

Therefore, we have $|T| = 1 + g(\phi) \leq 1 + \mathsf{dd}'(\phi)$. □

Thus, for any $\phi \in \mathcal{L}$, each s_i of the tableau T corresponds to a $w_i \in W$ in the S5-model, $|T| \leq \mathsf{dd}(\phi) + 1$ means that the number of worlds in the S5-model is bounded by $\mathsf{dd}(\phi) + 1$.

Theorem 2. *$\phi \in \mathcal{L}$ is satisfiable if and only if $tr(\phi, \mathsf{dd}(\phi)+1)$ is satisfiable.*

Structural Caching

Caching is a classical way to avoid redundant work. *SAT performs caching using a "bit matrix" (Giunchiglia and Tacchella 2001). Efficient implementation of BDD (Bryant 1986) packages also rely in caching, to build an explicit graph. These two examples require additional time and space to search and cache already performed works. Here, our technique is a "simple but efficient" trade-off. It does not memoize the work, so it may not cache all possible formulas, but it only requires a flag to detect redundant work.

In the example depicted in Figure 2.b, sub-formula $(b_1 \wedge b_2)$ appears twice. The translation of the first diamond creates two sub-formulas $a_1 \wedge \Diamond b$ and $a_2 \wedge \Diamond b$, where each $\Diamond b$ needs to be translated.

Because we are in S5 (all worlds are connected), the translations of $\Diamond b$ on different worlds are equivalent, so we can reuse the same sub-formula. It means that instead of using a tree, we can work with a DAG, which allows a more efficient translation to CNF.

Lemma 3. $\mathrm{tr}'(\circ\phi, i, n) = \mathrm{tr}'(\circ\phi, j, n)\ \forall i, j$ *and* $\circ \in \{\Diamond, \Box\}$

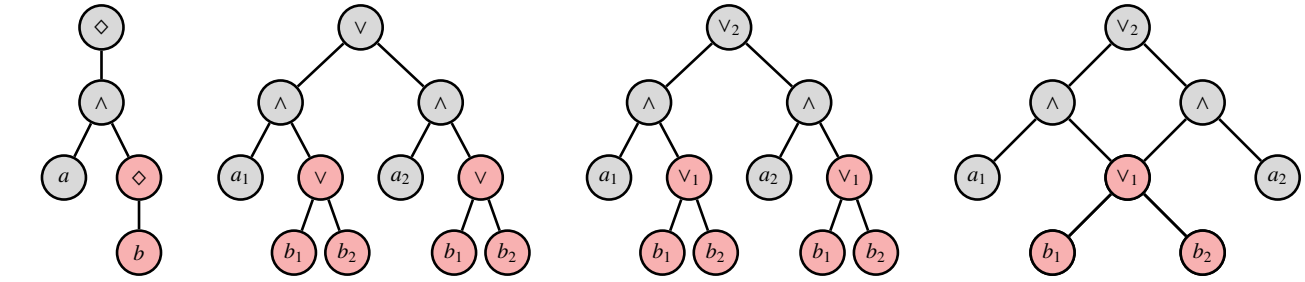
Proof of Lemma 3. By definition, there are only 2 cases:
- $(\circ = \Box)$ then, $\mathrm{tr}'(\Box\phi, i, n) = \bigwedge_{k=1}^{n}(\mathrm{tr}'(\phi, k, n))$
- $(\circ = \Diamond)$ then $\mathrm{tr}'(\Diamond\phi, i, n) = \bigvee_{k=1}^{n}(\mathrm{tr}'(\phi, k, n))$

In each case, the result is independent from i, so choosing j as an index gives the exact same result. □

Informally, lemma 3 shows the fact that no matter how embedded the modal sub-formula is, its translation will always give the same result (independent from the index i). Therefore, we can start by translating the most embedded sub-formula, tag the corresponding node and backtrack. The resulting formula may contain several nodes with the same tag. This means that these sub-formulas are syntactically identical (see Figure 2.c). Then, we maintain only one occurrence of the sub-formula, transforming the tree in a DAG (see Figure 2.d). Structural caching is thus performed on the fly before translating to CNF. The translation function using this technique is noted tr^{+}.

Experiments

We considered LCKS5TabProver (Abate, Goré, and Widmann 2007) and SPASS 3.7 (Weidenbach et al. 2009), which are, to the best of our knowledge, the state-of-the-art in



(a) $\phi = \Diamond(a \wedge \Diamond b)$ (b) $\mathrm{tr}(\phi, \mathrm{dd}(\phi))$ (c) $\mathrm{tr}^+(\phi, \mathrm{dd}(\phi))$ (d) $\mathrm{tr}^+(\phi, \mathrm{dd}(\phi))$ with simplification

Figure 2: Translation of $\Diamond(a \wedge \Diamond b)$ (left), initial translation (middle-left), tagged formula (middle-right), final translation (right)

S5-satisfiability solving. We compared them to 4 different configurations of S52SAT: `nm`, `nm+` (with caching), `dd`, `dd+` (with caching) (http://www.cril.univ-artois.fr/ montmirail/s52SAT). In order to evaluate rigorously the solvers, we used the same input for each solver, in InToHyLo format (Hoffmann 2010). For this purpose, we modified the code of LCKS5TabProver to make it able to read that format. The modifications are simple enough to guarantee that the performance of the solver is not affected.

We used SPASS 3.7 and not the latest available version 3.9 because the former reads `dfg` format which can be produced from InToHyLo benchmarks using the tool `ftt` (Fast Transformation Tool) embedded in Spartacus (Götzmann, Kaminski, and Smolka 2010). The difference between 3.7 and 3.9 is minimal according to the solver's web site. The translation time is negligible in our experiments. We use Glucose 4.0 (Audemard and Simon 2009) as backend SAT solver. We also considered MetTel2 (Tishkovsky, Schmidt, and Khodadadi 2012) and LoTREC (Gasquet et al. 2005) but unfortunately they are not designed to efficiently solve modal logic S5 problems. We evaluated these solvers on well established modal logic benchmarks: $3CNF_K$ (Patel-Schneider and Sebastiani 2003), $MQBF_K$ (Massacci 1999), $TANCS\,2000_K$ (Massacci and Donini 2000) and $LWB_{K,KT,S4}$ (Balsiger, Heuerding, and Schwendimann 2000). Note that they are designed for modal logic K, KT and S4. As a consequence, some of them are trivial in the S5 setting. However, we believe that the results on those benchmarks are still significant. S5-SAT entails K, KT and S4-SAT, so we could envision S5-SAT as a preprocessing step for those modal logics.

We set the memory limit to 8GB and the runtime limit to 900 seconds. We rarely reached the timeout (804 times out of 12444 runs). Most unsolved benchmarks are due to lack of memory.

In the following tables, we provide the number of benchmarks solved, in bold the best results of a given row/column (according to the orientation of the Table); and between parenthesis, we provide the number of benchmarks which cannot be solved because of lack of memory, if any.

3CNF_K : Caching does not help

The results are displayed in the Table 1. `dd`, `dd+` and SPASS are quite close to each other. The formulas consist of big conjunctions where each conjunct has modal depth of at most 2,4 or 6. They are constructed in such a way that our caching algorithm could not find redundancies on such formulas. This is why the caching does not provide any benefit on those benchmarks.

Solver	d=2	d=4	d=6	Total
LckS5TabProver	0 (17)	0 (29)	0 (40)	0
S52SAT `nm`	21 (24)	0 (45)	0 (45)	21
S52SAT `nm+`	21 (24)	0 (45)	0 (45)	21
S52SAT `dd`	**45 (0)**	**8 (27)**	0 (45)	**53**
S52SAT `dd+`	**45 (0)**	**8 (27)**	0 (45)	**53**
SPASS 3.7	**45 (0)**	5 (40)	0 (45)	50

Table 1: #instances solved in $3CNF_K$

modKSSS and modKLadn : Caching helps

n,*a*	#	LckS5...	`nm`	`nm+`	`dd`	`dd+`	SPASS 3.7
4,4	40	0 (12)	32	**40**	**40**	**40**	**40**
4,6	40	0 (23)	32	**40**	**40**	**40**	32 (8)
8,4	40	0 (16)	32	**40**	39 (1)	**40**	16 (24)
8,6	40	0 (17)	24	**40**	**40**	**40**	10 (30)
16,4	40	0 (10)	22	**40**	**40**	**40**	8 (32)
16,6	40	0 (16)	19	**40**	39 (1)	**40**	2 (38)
total	240	0	161	**240**	238	**240**	108
4,4	40	**40**	0 (40)	**40**	0 (40)	**40**	0 (40)
4,6	40	14 (0)	0 (40)	32 (8)	0 (40)	**40**	0 (40)
8,4	40	2 (0)	0 (40)	8 (32)	0 (40)	**40**	0 (40)
8,6	40	0 (0)	0 (40)	0 (40)	0 (40)	**8** (32)	0 (40)
16,4	40	0 (0)	0 (40)	0 (40)	0 (40)	0 (40)	0 (40)
16,6	40	0 (0)	0 (40)	0 (40)	0 (40)	0 (40)	0 (40)
total	240	56	0	80	0	**128**	0

Table 2: Upper: modKSSS — Lower: modKLadn

n represents the number of variables and *a* represents the number of alternations in the original QBF prefix, see (Massacci 1999) for more details, and # corresponds to the number of benchmarks available for a given pair (*n*,*a*).

In this category, we can see that `dd` and `dd+` are much more efficient than SPASS. The formulas in these benchmarks contain a lot of redundancies: it can be seen that enabling caching allows us to solve all modKSSS benchmarks for instance.

TANCS-2000 : Memory Demanding

n,*a*	#	LckS5...	nm	nm+	dd	dd+	SPASS 3.7
4,4	40	0 (38)	**40**	**40**	**40**	**40**	**40**
4,6	40	0 (33)	**40**	**40**	**40**	**40**	**40**
8,4	40	0 (38)	**40**	**40**	**40**	**40**	**40**
8,6	40	0 (39)	**40**	**40**	**40**	**40**	**40**
16,4	40	0 (40)	**40**	**40**	**40**	**40**	**40**
16,6	40	0 (40)	**40**	**40**	**40**	**40**	35 (5)
total	240	0	**240**	**240**	**240**	**240**	235
4,4	40	23 (0)	3 (37)	**40**	**40**	**40**	**40**
4,6	40	3 (0)	0 (40)	**40**	**40**	**40**	31 (9)
total	80	26	3	**80**	**80**	**80**	71

Table 3: TANCS-2000. Upper: qbfMS — Lower: qbfML

Here, we can see that these problems can be solved by almost all the solvers. The instances not solved by SPASS 3.7 are due to lack of memory. As pointed out in the following section, by increasing the memory limit to 32GB, these instances are solved by SPASS.

LWB K, KT, S4 : Both dd and caching help

Solver	Logic K	Logic KT	Logic S4	Total
LckS5TabProver	227 (73)	206 (102)	194 (111)	627
S52SAT nm	307 (66)	344 (33)	292 (86)	943
S52SAT nm+	351 (21)	363 (12)	349 (28)	1063
S52SAT dd	333 (40)	355 (23)	337 (40)	1025
S52SAT dd+	**357 (12)**	**364 (12)**	**363 (12)**	**1084**
SPASS 3.7	343 (16)	363 (15)	360 (17)	1066

Table 4: #instances solved in $LWB_{K,KT,S4}$

The benchmarks are originally separated on SAT/UNSAT formulas for the logics K, KT and S4. Obviously, the satisfiability in S5 may differ so we removed the SAT/UNSAT separation. The results are displayed in the Table 4. Here again, dd+ performs slightly better than SPASS. The benchmarks which cannot be solved are a specific modal logic encoding of the pigeon hole principle (Haken 1985) in the corresponding logic.

Overall results on all the benchmarks

Solver	# solved	# SAT	MO	TO
LckS5TabProver	709	143	710	655
S52SAT nm	1377	411	667	30
S52SAT nm+	1733	452	292	49
S52SAT dd	1645	433	412	17
S52SAT dd+	**1834**	**460**	**203**	37
SPASS 3.7	1530	451	528	**16**

Table 5: Overall results on all the benchmarks

SPASS is outperformed in part because of its poor results on the modKSSS and modKLadn benchmarks. While the default SAT encoding often exhausts the available memory, each of the two proposed improvements significantly

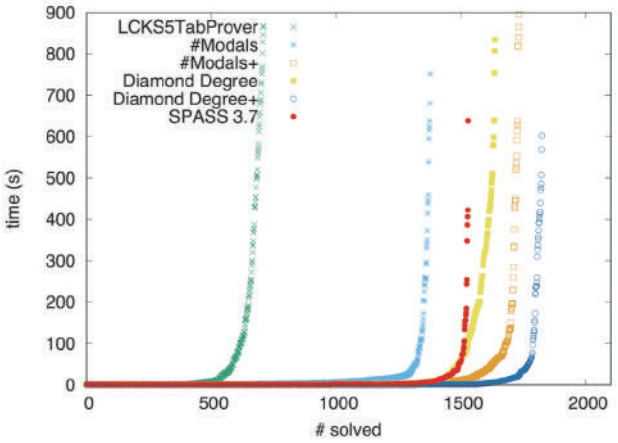


Figure 3: runtime distribution

reduces the number of memory-out, the best results being obtained when both are enabled.

It seems pretty clear that the structural caching is key in our approach to efficiently solve those benchmarks. This is witnessed by the scatter plot in Figure 4. The x-axis corresponds to the time used by dd and the y-axis corresponds to the time used by dd+ to solve the problem.

Solver	avg	median	max
nm	6 881 821	1 100 054	58 653 264
nm+	1 619 923	118 040	29 492 779
dd	2 385 515	169 324	55 813 557
dd+	269 891	27 090	22 914 442

Table 6: Number of clauses in the generated CNF formulas

The main reason of the improvement is the reduction of the size of the CNF encoding, as shown by Table 6. We had a median value of 1,100,054 clauses for nm and this value drops down to 27,090 for dd+ on the exact same problems.

Importance of the memory

We repeated the experiments with 32GB of RAM. Note that for the SAT based approach, the lack of memory happens during the translation phase, while for the others, the memory is exhausted in the solving phase.

One may wonder what would be the performance with more memory. Table 7 summarizes the number of problems solved with 8GB and 32GB by each solver. Providing 32GB to SPASS does not change the results significantly: it solved 30 additional instances. Our SAT approach also benefits from that increased amount of memory, up to 98 additional benchmarks can be solved by nm without caching. 32 GB is sufficient for LckS5TabProver (no memory-out), however only one additional benchmark can be solved.

Figure 6 compares the memory consumption of SPASS 3.7 and S52SAT dd+ in MB. SPASS usually requires more memory than dd+. The runtime and the memory consumption are roughly correlated (the Pearson product-moment

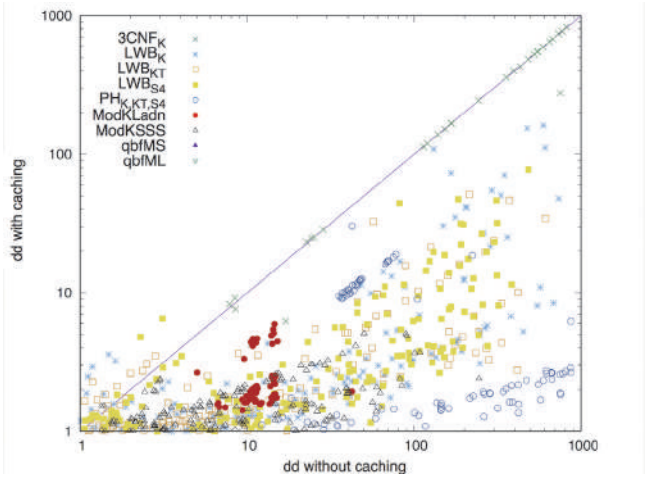


Figure 4: Runtime of dd with and without caching

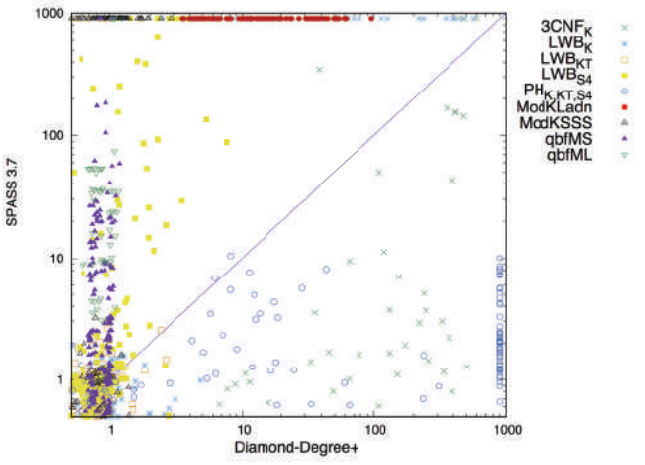


Figure 5: Runtime of SPASS 3.7 vs dd+

Solver	8GB	32GB	MO with 32GB
LckS5TabProver	709	710	0
SPASS 3.7	1530	1560	498
S52SAT nm	1377	1475	382
S52SAT nm+	1733	1800	166
S52SAT dd	1645	1672	317
S52SAT dd+	1834	1888	96

Table 7: #Solved with 8GB and 32GB

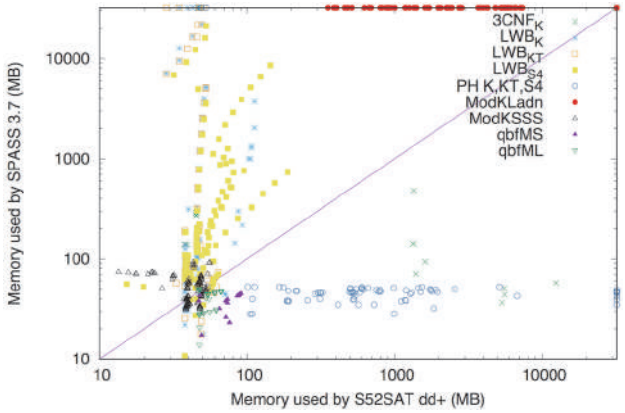


Figure 6: Memory comparison (32 MB) SPASS 3.7 vs dd+

correlation coefficient for dd+ is equal to 0.58 and for SPASS it is 0.57). In the category ModKLadn, SPASS runs out of memory even with 32 GB.

Comparison against the state-of-the-art

Figure 5 shows a detailed runtime comparison between dd+ and SPASS on all benchmarks. In most cases, our approach outperforms SPASS. The two cases where it is less efficient consists in the hard combinatorial UNSAT Pigeon-Hole problems and the so-called "3CNF" benchmarks on which neither the diamond degree nor the caching helps.

Conclusion

We presented a new SAT encoding to solve the S5-SAT problem using a SAT solver. It is based on a reduction from S5-SAT to SAT with two improvements: a better upper bound on the number of required worlds and a structural caching. We compared our approach against solvers representing, to the best of our knowledge, the state-of-the-art for practical S5-SAT solving, on a wide range of classical modal logic benchmarks. The SAT based approach with all improvements enabled outperformed those solvers.

Even if the benchmarks may not be representative of practical S5 benchmarks because they come from other modal logics (K, KT, S4), those results open interesting perspectives. It is indeed the case that proving the satisfiability of a modal logic formula in S5 entails that the formula is also satisfiable on less restrictive systems (i.e. in K, KT, S4). Since our S5-solver provides a S5-model in just a few seconds (2.06s median time), we could perfectly use it as a preprocessing step to solve benchmarks in other modal logics.

Preliminary results on that direction are quite encouraging: on 276 satisfiable benchmarks in logic S5 (thus in K), S52SAT outperforms the state of the art Spartacus (Götzmann, Kaminski, and Smolka 2010) on 158 of them. Another perspective, would be to adapt S52SAT in order to solve KD45-SAT problems given the fact that KD45 is also NP-Complete (Fagin et al. 1995).

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We thank Renate Schmidt for providing help on finding S5 solvers. Part of this work was supported by the French Ministry for Higher Education and Research and the Nord-Pas de Calais Regional Council through the "Contrat de Plan État Région (CPER)" and by an EC FEDER grant.

References

Abate, P.; Goré, R.; and Widmann, F. 2007. Cut-free single-pass tableaux for the logic of common knowledge. In *Workshop on Agents and Deduction at TABLEAUX'07*.

Areces, C.; Fontaine, P.; and Merz, S. 2015. Modal satisfiability via SMT solving. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing*, 30–45.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proc. of IJCAI'09*, 399–404.

Balsiger, P.; Heuerding, A.; and Schwendimann, S. 2000. A Benchmark Method for the Propositional Modal Logics K, KT, S4. *J. Autom. Reasoning* 24(3):297–317.

Bienvenu, M.; Fargier, H.; and Marquis, P. 2010. Knowledge compilation in the modal logic S5. In *Proc. of AAAI'10*.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* 35(8):677–691.

Fagin, R.; Halpern, J. Y.; Moses, Y.; and Vardi, M. Y. 1995. *Reasoning About Knowledge*. The MIT Press.

Fairtlough, M., and Mendler, M. 1994. An Intuitionistic Modal Logic with Applications to the Formal Verification of Hardware. In *Proc. of CSL'94*, 354–368.

Fitting, M. 2000. Modality and databases. In *Proc. of TABLEAUX'00*, 19–39.

Gasquet, O.; Herzig, A.; Longin, D.; and Sahade, M. 2005. LoTREC: Logical Tableaux Research Engineering Companion. In *Proc. of TABLEAUX'05*, 318–322.

Giunchiglia, F., and Sebastiani, R. 2000. Building Decision Procedures for Modal Logics from Propositional Decision Procedures: The Case Study of Modal K(m). *Inf. Comput.* 162(1-2):158–178.

Giunchiglia, E., and Tacchella, A. 2000. System description: *SAT: A platform for the development of modal decision procedures. In *Proc. of CADE'00*, 291–296. Springer.

Giunchiglia, E., and Tacchella, A. 2001. A subset-matching size-bounded cache for testing satisfiability in modal logics. *Ann. Math. Artif. Intell.* 33(1):39–67.

Giunchiglia, E.; Sebastiani, R.; Giunchiglia, F.; and Tacchella, A. 2000. SAT vs. Translation based decision procedures for modal logics: a comparative evaluation. *Journal of Applied Non-Classical Logics* 10(2):145–172.

Giunchiglia, E.; Giunchiglia, F.; and Tacchella, A. 1999. The sat-based approach for classical modal logics. In *Proc. of AI*IA'99*, 95–106.

Giunchiglia, E.; Tacchella, A.; and Giunchiglia, F. 2002. SAT-Based Decision Procedures for Classical Modal Logics. *J. Autom. Reasoning* 28(2):143–171.

Götzmann, D.; Kaminski, M.; and Smolka, G. 2010. Spartacus: A tableau prover for hybrid logic. *Electr. Notes Theor. Comput. Sci.* 262:127–139.

Haken, A. 1985. The intractability of resolution. *Theoretical Computer Science* 39(0):297 – 308.

Halpern, J. Y., and Rêgo, L. C. 2007. Characterizing the NP-PSPACE Gap in the Satisfiability Problem for Modal Logic. In *Proc. of IJCAI'07*, 2306–2311.

Hoffmann, G. 2010. *Tâches de raisonnement en logiques hybrides*. Ph.D. Dissertation, Henri Poincaré University, Nancy.

Kaminski, M., and Tebbi, T. 2013. InKreSAT: Modal Reasoning via Incremental Reduction to SAT. In *Proc. of CADE'13*, 436–442.

Ladner, R. E. 1977. The Computational Complexity of Provability in Systems of Modal Propositional Logic. *SIAM J. Comput.* 6(3):467–480.

Lorini, E., and Schwarzentruber, F. 2010. A modal logic of epistemic games. *Games* 1(4):478–526.

Massacci, F., and Donini, F. M. 2000. Design and results of TANCS-2000 non-classical (modal) systems comparison. In *Proc. of Tableaux'00*, 52–56.

Massacci, F. 1999. Design and results of the tableaux-99 non-classical (modal) systems comparison. In *Proc. of Tableaux'99*, 14–18.

Masssacci, F. 2000. Single step tableaux for modal logics: Methodology, computations, algorithms. *Journal of Autom.* 24(3):319–364.

Niveau, A., and Zanuttini, B. 2016. Efficient Representations for the Modal Logic S5. In *Proc. of IJCAI'16*, 1223–1229.

Patel-Schneider, P. F., and Sebastiani, R. 2003. A new general method to generate random modal formulae for testing decision procedures. *J. Artif. Intell. Res.* 18:351–389.

Patrick Blackburn, J. v. B., and Wolter, F. 2007. *Handbook of Modal Logic*. Elsevier.

Robinson, J. A., and Voronkov, A., eds. 2001. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press.

Sebastiani, R., and Tacchella, A. 2009. SAT Techniques for Modal and Description Logics. *Handbook of Satisfiability* 185:781–824.

Sebastiani, R., and Vescovi, M. 2009. Automated Reasoning in Modal and Description Logics via SAT Encoding: the Case Study of K(m)/ALC-Satisfiability. *JAIR* 35(1):343.

Sebastiani, R., and Villafiorita, A. 1998. Sat-based decision procedures for normal modal logics: A theoretical framework. In *Proc. of AIMSA'98*, 377–388.

Tishkovsky, D.; Schmidt, R. A.; and Khodadadi, M. 2012. The Tableau Prover Generator MetTeL2. In *Proc. of JELIA'12*, 492–495.

Tseitin, G. S. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer. 466–483.

VII, T. M.; Crary, K.; and Harper, R. 2005. Distributed control flow with classical modal logic. In *Proc. of CSL'05*, 51–69.

Weidenbach, C.; Dimova, D.; Fietzke, A.; Kumar, R.; Suda, M.; and Wischnewski, P. 2009. SPASS version 3.5. In *Proc. of CADE'09*, 140–145.

Chapitre 11

A Recursive Shortcut for CEGAR : Application To The Modal Logic K Satisfiability Problem

Article paru dans les actes de « *the 26th International Joint Conference on Artificial Intelligence* » (IJCAI'17), pages 674 – 680, août 2017.

Co-écrit avec Daniel Le Berre, Tiago de Lima et Valentin Montmirail.

Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)

A Recursive Shortcut for CEGAR: Application to the Modal Logic K Satisfiability Problem

Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima and **Valentin Montmirail**

CRIL, Univ. Artois and CNRS, F62300 Lens, France
{lagniez,leberre,delima,montmirail}@cril.fr

Abstract

Counter-Example-Guided Abstraction Refinement (CEGAR) has been very successful in model checking. Since then, it has been applied to many different problems. It is especially proved to be a highly successful practical approach for solving the PSPACE complete QBF problem. In this paper, we propose a new CEGAR-like approach for tackling PSPACE complete problems that we call RECAR (Recursive Explore and Check Abstraction Refinement). We show that this generic approach is sound and complete. Then we propose a specific implementation of the RECAR approach to solve the modal logic K satisfiability problem. We implemented both CEGAR and RECAR approaches for the modal logic K satisfiability problem within the solver MoSaiC. We compared experimentally those approaches to the state-of-the-art solvers for that problem. The RECAR approach outperforms the CEGAR one for that problem and also compares favorably against the state-of-the-art on the benchmarks considered.

1 Introduction

SAT technology has proven to be a very successful practical approach to solve some NP-complete problems [Biere *et al.*, 2009]. One of the main issues is to find the "right" encoding for the problem, i.e. to find a polynomial reduction from the original problem into a propositional formula in Conjunctive Normal Form (CNF, a set of clauses) which can be efficiently solved by a SAT solver [Prestwich, 2009]. The SAT solver is a generic problem solving engine, whose input is a satisfiability equivalent CNF representing the original problem. It often happens that either the SAT solver can solve efficiently the CNF or not at all (see e.g. the results of the SAT competitions [Järvisalo *et al.*, 2012]). A particular case is when the resulting CNF is very large: the time for generating and reading the input is greater than the time to solve it. This is due to the limited available main memory allocated to the approach and not the SAT solver itself. We address one of such particular cases in this paper.

For huge CNF encodings, specific approaches have been designed in the past, where a SAT solver is used as an oracle in a more complex procedure. One such procedure is called Counter-Example-Guided Abstraction Refinement (CEGAR) [Clarke *et al.*, 2003]. The SAT solver is fed with an abstraction of the original problem allowing more models (which we will call under-approximation). If the abstraction is unsatisfiable, then the original problem is also unsatisfiable (UNSAT shortcut). Else the procedure is able to verify if the model found for that abstraction is a correct solution for the original problem. In this case, we have an additional SAT shortcut to decide the satisfiability of the formula. If it is not the case, new constraints are added to prevent the solver from finding such spurious examples (refinement step) and the process repeats. Eventually, a complete satisfiability equivalent propositional formula is provided, and the SAT solver can decide the problem. One of the reasons for using CEGAR is that the complete formula is in practice too large to even be generated, so the only hope to solve the original problem is to "get lucky" (satisfiable shortcut) or to be able to take into account a specific structure of the problem (unsatisfiable shortcut).

This framework is elegant and has been applied to many areas: Satisfiability Modulo Theory [Brummayer and Biere, 2009], Planning [Seipp and Helmert, 2013] and more recently QBF [Janota *et al.*, 2016]. The latter is especially inspiring, because it appears to be the best practical solution overall to solve QBF formulas according to the latest QBF competition [Pulina, 2016]. The aim in this work is to follow these steps on another PSPACE complete problem, which is the satisfiability of modal logic K formulas. Several previous SAT-based approaches have already been proposed in the field of Modal Logic [Sebastiani and Tacchella, 2009], one could even argue that *SAT [Giunchiglia *et al.*, 2002] is already a CEGAR approach for the modal logic K.

In this paper, we introduce an extension of the CEGAR approach which includes a recursive step to introduce a new shortcut in the original CEGAR procedure. In our context, the main CEGAR loop contains a SAT shortcut ($\lightning$sat), while the recursive step allows the procedure to provide an UNSAT shortcut ($\lightning$unsat). We call this extension "Recursive Explore and Check Abstraction Refinement" (RECAR). The idea of mixing SAT and UNSAT shortcuts in a CEGAR procedure is not new: it has been already used for SMT [Brummayer and Biere, 2009] and for bug detection [Wang *et al.*, 2007]. Here the novelty is that we use an abstraction of the original problem in the loop, made possible by a recursive call to the main pro-

cedure. The RECAR procedure is generic, i.e. it is not bounded to a specific domain. In this paper, we present the conditions required on the abstractions used, and the correctness of the approach. Then, we instantiate our framework for the satisfiability of modal logic K, by providing abstraction functions for this problem and experimental results of the approach against the state-of-the-art provers. We believe that the good practical results we obtained by using RECAR on that particular domain are promising for other areas. This paper presents, in that respect, a generic procedure with a successful use case. The remainder of the paper is organized as follows: we first present the CEGAR approach; then we propose our new framework called RECAR and show its soundness and completeness; we provide an implementation of the RECAR approach for modal logic K; we finally compare the efficiency of the RECAR approach against the state-of-the-art modal logic K solvers.

2 CEGAR Preliminaries

Counter-Example-Guided Abstraction Refinement, CEGAR, is an incremental way to decide the satisfiability of formulas in classical propositional logic (CPL). It has been originally designed for model checking [Clarke *et al.*, 2003], i.e. to answer questions such as "Does $S \models P$ hold?" or, equivalently, "Is $S \wedge \neg P$ unsatisfiable?", where S describes a system and P a property. In such highly structured problems, it is often the case that only a small part of the formula is needed to answer the question. The idea behind CEGAR is to replace $\phi = S \wedge \neg P$ by an approximation ϕ', where ϕ' is easier to solve in practice than ϕ. There are two kinds of approximations: (1) an over-approximation of ϕ is a formula $\hat{\phi}$ such that $\hat{\phi} \models \phi$ holds: $\hat{\phi}$ has at most as many models as ϕ; (2) an under-approximation of ϕ is a formula $\check{\phi}$ such that $\phi \models \check{\phi}$ holds: $\check{\phi}$ has at least as many models as ϕ. Usually, ϕ is in CNF. Then, a classical way to under-approximate ϕ is to "forget" some clauses, i.e. $\check{\phi}$ is a subset of the clauses in ϕ. A model of $\check{\phi}$ also may by chance satisfy ϕ. Moreover, if $\check{\phi}$ is found to be unsatisfiable, then so is ϕ. This double possibility to conclude earlier makes under-approximation based CEGAR very popular. A classical way to over-approximate is to bound the generation of the formula ϕ to a given n smaller than the one needed to reach equi-satisfiability to the original problem (as in bounded model checking [Clarke *et al.*, 2003] or planning [Seipp and Helmert, 2013]). As such a model of $\hat{\phi}$ can be extended to a model of ϕ but the unsatisfiability of ϕ means that the bound n has to be increased and the process is repeated.

An example of a CEGAR using over-approximations is given on Fig. 1. It receives a formula ϕ as input and computes an over-approximation ψ. Then it uses a SAT solver to check whether ψ is satisfiable. If so it concludes that ϕ is satisfiable. Otherwise, ψ is refined, i.e. it gets closer to ϕ, until it is satisfiable, or until the refined over-approximation is detected to be equi-satisfiable to ϕ, denoted $\psi \equiv_{\text{sat}} \phi$, (i.e. $\exists M, M \models_1 \psi$ iff $\exists M', M' \models_2 \phi$)[1], where it concludes that ϕ is unsatisfiable. In the following, $\phi \equiv^{?}_{\text{sat}} \psi$ means an incomplete efficient equi-satisfiability test which returns yes or unknown. Recent

[1] $\models_1$ and $\models_2$ denote possibly different consequence relations (for propositional logic and modal logic K for instance).

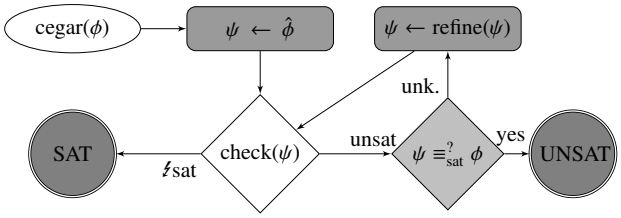


Figure 1: The CEGAR framework with over-approximation

SAT solvers are able to check satisfiability "under assumption" [Eén and Sörensson, 2003], i.e. given the satisfiability of a set of literals called assumptions, and to provide in case of unsatisfiability a "reason" in terms of those literals for the unsatisfiability of a formula.

Definition 1 (Unsatisfiable core with assumptions)**.** *Let ϕ a CNF and A a consistent set of literals from ϕ. Let ϕ be satisfiable and $(\phi \wedge \bigwedge_{a \in A} a)$ be unsatisfiable. $L \subseteq A$ is an unsatisfiable core of ϕ if and only if $(\phi \wedge \bigwedge_{l \in L} l)$ is unsatisfiable.*

Therefore, a SAT oracle for ϕ, given A, can be seen as a procedure providing a pair (d, ψ) with $d \in \{\text{SAT}, \text{UNSAT}\}$ and ψ is a model of ϕ if $d = \text{SAT}$ or an unsatisfiable core of ϕ if $d = \text{UNSAT}$. Modern SAT-based procedures are able to take into account ψ in both cases. Unsatisfiable cores have been used for instance in a CEGAR approach for deciding the satisfiability of the propositional fragment of first-order logic [Khasidashvili *et al.*, 2015].

3 Recursive Explore and Check Abstraction Refinement

A classic CEGAR approach with over-approximation and a SAT shortcut performs well when the input is satisfiable. But generally, it does not perform well in problems which are unsatisfiable. The reason is that it may have to keep refining until it reaches equi-satisfiability with the original problem. One way to address this issue is to mix SAT and UNSAT shortcuts, as in [Brummayer and Biere, 2009] and [Wang *et al.*, 2007]. In these approaches, the methods alternate between over and under approximations.

The RECAR approach, depicted in Fig. 2 and Fun. 1, interleaves both kinds of approximations: each abstraction is performed with the information retrieved from solving the previous one. The UNSAT shortcut is implemented using a recursive call to the main procedure when a strict under-approximation $\check{\phi}$ can be built. One should also note that the proposed approach permits abstractions on two different levels: one is used to simplify the problem at the domain level (recursive call), while the other one is used to approximate the problem at the oracle level. In order to apply RECAR, the under-approximation $\check{\phi}$ and the over-approximation $\hat{\phi}$ must satisfy some properties. In the following, isSAT(ϕ) means that ϕ is satisfiable ($\not\models_1 \neg\phi$) and isUNSAT(ϕ) means ($\models_2 \neg\phi$), but on possibly different consequence relations. $RC(\phi, \check{\phi})$ denotes a Boolean function deciding if a Recursive Call should occur.

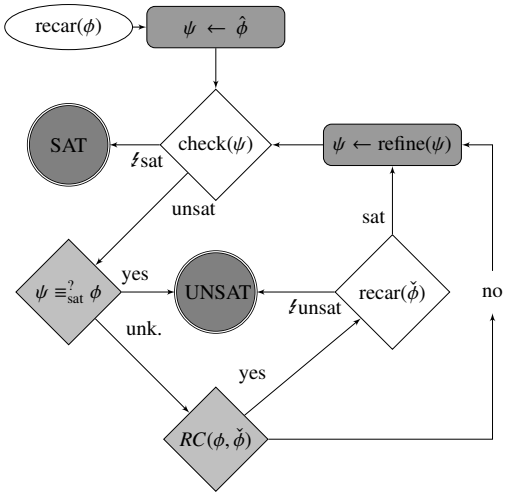


Figure 2: The RECAR framework

RECAR assumptions:

1. Function 'check' is a sound and complete implementation of 'isSAT' which terminates.
2. isSAT($\hat{\phi}$) implies isSAT(refine($\hat{\phi}$)).
3. There exists $n \in \mathbb{N}$ such that refine$^n(\hat{\phi}) \equiv^?_{\text{sat}} \phi$.
4. isUNSAT($\check{\phi}$) implies isUNSAT(ϕ).
5. Let under(ϕ) $= \check{\phi}$. There exists $n \in \mathbb{N}$ such that RC(under$^n(\phi)$, under$^{n+1}(\phi)$) evaluates to false.

Note that we have isSAT($\hat{\phi}$) implies ϕ is satisfiable by Assumptions 2 and 3 together. In the following, we show that, under these assumptions, RECAR is sound, complete and terminates. To do so, we present the algorithm recar(ϕ) in Fun. 1.

Function 1: recar(ϕ)

```
1 ψ ← ϕ̂
2 while ψ ≡?sat ϕ returns "unknown" do
3     if check(ψ) = SAT then return SAT ;
4     if RC(ϕ, ϕ̌) then
5         if recar(ϕ̌) = UNSAT then return UNSAT ;
6     ψ ← refine(ψ) ;
7 return check(ψ)
```

Theorem 1 (Soundness)**.** *If* recar(ϕ) *returns* SAT *then* ϕ *is satisfiable.*

Proof. Assume that recar(ϕ) returns SAT. This happens only if check(ψ) returns SAT, either from line 3 or from line 7. Thus, we know that isSAT(ψ) holds (by Assump. 1). But ψ equals to $\hat{\phi}$ or equals to refine$^n(\phi)$ for some $n \in \mathbb{N}$. Then ϕ is satisfiable (by Assump. 2 and 3). □

The intuition behind the proof of Th. 2 is that there are two ways to conclude that ϕ is UNSAT. In the first case, $\hat{\phi}$ is refined a finite number of times until it is detected equi-satisfiable to ϕ and check returns UNSAT. Then ϕ is unsatisfiable. In the second case, one of the under-approximations is shown UNSAT, then ϕ is UNSAT (by Assump. 4).

Theorem 2 (Completeness)**.** *If* recar(ϕ) *returns* UNSAT *then* isUNSAT(ϕ).

Proof. By induction on the number k of recursive calls to recar (Line 5). Assume recar(ϕ) returns UNSAT after k recursive calls. In the induction base $k = 0$ (no recursive call). Then we must have exited the loop ($\psi \equiv^?_{\text{sat}} \phi$) and check($\psi$) returns UNSAT. This means that ψ is unsatisfiable (by Assump. 1) and therefore isUNSAT(ϕ) holds (because of equi-satisfiability). The induction hypothesis is: for all $k \leq n$, if recar(ϕ) returns UNSAT after k recursive calls then isUNSAT(ϕ). In the induction step $k = n + 1$. Then the conditions of lines 4 and 5 of the algorithm are true. This means that recar($\check{\phi}$) returns UNSAT after k recursive calls to recar. Then isUNSAT($\check{\phi}$) (by I.H.). Then isUNSAT(ϕ) (by Assump. 4). □

The intuition behind the proof of Th. 3 is that the function performs a finite number of recursive calls (Assump. 5). Moreover, each of these calls will have a finite number of refinements before terminating (Assump. 3).

Theorem 3 (Termination)**.** RECAR *terminates for any input* ϕ*.*

Proof. We have that (1) For all ϕ, there exists $n \in \mathbb{N}$ such that RC(under$^n(\phi)$, under$^{n+1}(\phi)$) evaluates to false (by Assump. 5) and (2) For each $i \leq n$ there is $m_i \in \mathbb{N}$ such that refine$^{m_i}(\hat{\phi}) \equiv^?_{\text{sat}} \phi$ (by Assump. 3). Then, for any input ϕ, the recursive call of line 5 of the algorithm will be executed at most n times before the condition of line 4 becomes false. For each one of these recursive calls, the while-loop of the algorithm will be executed at most m_i times before the condition of line 2 becomes false. Therefore, for any input, recar halts after at most $n + \Sigma_{i=0}^{n}(m_i)$ recursive calls. □

4 Solving K-SAT with RECAR

In this section, we show how RECAR is applied to solve the satisfiability problem for modal logic K, which is a PSPACE complete problem [Ladner, 1977; Halpern and Moses, 1992].

4.1 Modal Logic K

Before showing how we applied the RECAR approach, let us define formally what the modal logic K is.

Let a finite non-empty set of propositional variables $\mathbb{P} = \{p_1, p_2, \dots\}$ and a set of m unary modal operators $\mathbb{M} = \{\Box_1, \dots, \Box_m\}$ be given. The language of K (noted $\mathcal{L}$) is the set of formulas containing $\mathbb{P}$, closed under the set of propositional connectives $\{\neg, \wedge\}$ and the set of modal operators in $\mathbb{M}$. We also use the standard abbreviations for $\top$, $\bot$, $\vee$ and $\Diamond$. For instance $\Diamond_a\phi = \neg\Box_a\neg\phi$.

Definition 2 (Model)**.** *A Kripke model is a triplet* $M = \langle W, \{R_a \mid \Box_a \in \mathbb{M}\}, V\rangle$*, where: W is a non-empty set of possible worlds; Each* $R_a \subseteq W \times W$ *is a binary accessibility relation on W;* $V : \mathbb{P} \to 2^W$ *is a valuation function which associates, to each* $p \in \mathbb{P}$*, the set of possible worlds from W where p is*

true. A pointed Kripke model is a pair $\langle M, w\rangle$, where M is a Kripke model and w is a possible world in W. Thereafter, whenever we use the term 'model' we refer to 'pointed Kripke model'.

In the following, the size of a model $\langle M, w\rangle$, which is the number of elements in W, is denoted by $|M|$.

Definition 3 (Satisfaction relation)**.** *The relation $\models$ between models and formulas is recursively defined as follows:*

$$\begin{aligned}
&\langle M, w\rangle \models p && \textit{iff} \quad w \in V(p)\\
&\langle M, w\rangle \models \neg\phi && \textit{iff} \quad \langle M, w\rangle \not\models \phi\\
&\langle M, w\rangle \models \phi_1 \wedge \phi_2 && \textit{iff} \quad \langle M, w\rangle \models \phi_1 \textit{ and } \langle M, w\rangle \models \phi_2\\
&\langle M, w\rangle \models \Box_a\phi && \textit{iff} \quad (w, w') \in R_a \textit{ implies } \langle M, w'\rangle \models \phi
\end{aligned}$$

Definition 4 (Validity)**.** *As usual, a formula $\phi \in \mathcal{L}$ is valid (noted $\models \phi$) if and only if it is satisfied by all models $\langle M, w\rangle$. A formula $\phi \in \mathcal{L}$ is satisfiable in K (noted* isKSAT(ϕ)*) if and only if $\not\models \neg\phi$. We also use* isKUNSAT(ϕ) *to mean $\models \neg\phi$.*

In the sequel, we define a translation from modal logic K to classical propositional logic.

Definition 5 (Translation)**.**

$$\begin{aligned}
\mathrm{tr}(\phi, n) &= \mathrm{tr}'(\mathrm{nnf}(\phi), 0, n)\\
\mathrm{tr}'(p, i, n) &= p_i\\
\mathrm{tr}'(\neg p, i, n) &= \neg p_i\\
\mathrm{tr}'(\phi \wedge \psi, i, n) &= \mathrm{tr}'(\phi, i, n) \wedge \mathrm{tr}'(\psi, i, n)\\
\mathrm{tr}'(\phi \vee \psi), i, n) &= \mathrm{tr}'(\phi, i, n) \vee \mathrm{tr}'(\psi, i, n)\\
\mathrm{tr}'(\Box_a\phi, i, n) &= \bigwedge_{j=0}^{n} (r^a_{i,j} \rightarrow \mathrm{tr}'(\phi, j, n))\\
\mathrm{tr}'(\Diamond_a\phi, i, n) &= \bigvee_{j=0}^{n} (r^a_{i,j} \wedge \mathrm{tr}'(\phi, j, n))
\end{aligned}$$

The translation adds fresh variables p_i and $r^a_{i,j}$ to the formula: p_i denotes that variable p is true in the world w_i whereas $r^a_{i,j}$ corresponds to w_j being accessible from w_i by the relation a. We have the following as an immediate result (where $nm(\phi)$ is the number of modal operators in ϕ) based on §25.2.2 [Sebastiani and Tacchella, 2009].

Theorem 4. isKSAT(ϕ) *if and only if* isSAT(tr($\phi, nm(\phi)+1$)).

Therefore, in order to decide the satisfiability of a formula $\phi \in \mathcal{L}$, one can simply feed a SAT solver with tr($\phi, nm(\phi)+1$). In fact, this is the approach proposed in Km2SAT [Sebastiani and Vescovi, 2009]. The main issue is that the translation may generate an exponentially larger formula. In this paper, we try to circumvent this problem by using RECAR. For simplicity, from now on we use tr(ϕ) instead of tr($\phi, nm(\phi) + 1$).

In [Caridroit *et al.*, 2017], the authors also use this translation from modal logic S5 to classical propositional logic, but replacing $nm(\phi)$ by the diamond degree dd(ϕ), which is generally smaller. Unfortunately, this cannot be used for K. The counter-example below shows why.

Counter-Example 1. *Let $\phi = (p_1 \wedge p_2 \wedge p_3) \wedge (\Diamond_a(p_1 \wedge p_2 \wedge \neg p_3 \wedge \Box_a(p_1 \wedge \neg p_2 \wedge p_3))) \wedge (\Diamond_a(p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge \Box_a(\neg p_1 \wedge \neg p_2 \wedge p_3))) \wedge (\Box_a\Diamond_a p_3)$, with* dd($\phi$) = 3*. This formula is satisfied by the model in Fig. 3. However, it is easy to see that there is no model satisfying ϕ with less than 5 possible worlds.*

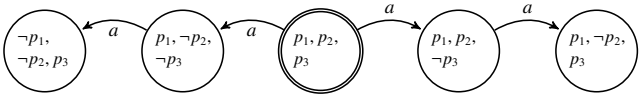


Figure 3: $M \models \phi$

4.2 Over-approximation

Now, in order to apply the RECAR approach, we first need to find an over-approximation which respects the assumptions presented in Sec. 3.

Definition 6 (Over-approximation)**.** *Let $\phi \in \mathcal{L}$. The over-approximation of ϕ, denoted $\hat{\phi}$, is the formula* tr(ϕ, 1).

Definition 7 (Refinement)**.** *Let $1 \leq n \leq nm(\phi)+1$. The refinement of* tr(ϕ, n)*, noted* refine(tr(ϕ, n)) *is the formula* tr($\phi, n+1$).

Theorem 5. *If* isSAT(tr(ϕ, n)) *then* isSAT(tr($\phi, n+1$))*, for all $1 < n \leq nm(\phi) + 1$. (*RECAR *Assump. 2)*

Proof Sketch. The idea is that if ϕ is satisfied by a model M with n worlds, then we can find a model M' with $n+1$ worlds satisfying ϕ. The additional world is just not accessible from the ones already in M. □

The latter result allows us to use this over-approximation and refinement in the RECAR approach. It is easy to see that RECAR assumptions 2 and 3 are satisfied.

4.3 Under-approximation

To understand the intuition behind the under-approximation we use an example. Let $\phi = (\Diamond p \wedge \Box\neg p \wedge \chi)$ for some $\chi \in \mathcal{L}$, where $nm(\chi)$ is huge. This is clearly unsatisfiable because $(\Diamond p \wedge \Box\neg p)$ is unsatisfiable. One can see that right away without even knowing what χ looks like. However, a CEGAR approach using the over-approximation and refinement defined earlier will take a long time before finally conclude it. The reason is that each refinement tr($\phi, n + 1$) of the original formula will be shown unsatisfiable and it will not stop until the huge number $nm(\phi) + 1$ is reached.

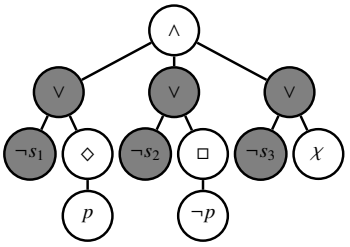


Figure 4: How selectors are applied to $\phi = (\Diamond p \wedge \Box\neg p \wedge \chi)$

To avoid these pathological cases, the RECAR approach also performs under-approximations. To see how it works, let us take that formula ϕ again. First, we add to each conjunct in ϕ a fresh variable s_i (a selector) that will be assumed to be true by the SAT solver, as done in Fig. 4. Then, we make the first over-approximation tr(ϕ, 1) and give it to a modern SAT solver. The solver will return UNSAT with an unsatisfiable core. From this core, we extract a set of selectors *core*. Let

us assume, in our example, that $core = \{s_1, s_2\}$. This means that the formula $\check{\phi} = (\Diamond p \wedge \Box\neg p)$, which is the one labelled by the selectors, is enough to prove the unsatisfiability of ϕ with only 1 possible world. Proving the unsatisfiability of $\check{\phi}$ will imply that ϕ is unsatisfiable. Note that, in this specific case, $nm(\check{\phi})$ is much smaller than $nm(\phi)$. Thus the CEGAR approach applied to $\check{\phi}$ will succeed much earlier, while it may have failed for the entire formula ϕ. Formally, we have the following.

Definition 8 (Under-Approximation)**.**

$$\text{under}(p, core) = p$$
$$\text{under}(\neg p, core) = \neg p$$
$$\text{under}(\Box_a\phi, core) = \Box_a(\text{under}(\phi, core))$$
$$\text{under}(\Diamond_a\phi, core) = \Diamond_a(\text{under}(\phi, core))$$
$$\text{under}((\phi \wedge \psi), core) = \text{under}(\phi, core) \wedge \text{under}(\psi, core)$$
$$\text{under}((\psi \vee \chi), core) = \begin{cases} \text{under}(\chi, core) & \text{if } \psi = \neg s_i, s_i \in core \\ \top & \text{if } \psi = \neg s_i, s_i \notin core \\ (\text{under}(\psi, core) & \\ \vee \text{under}(\chi, core)) & \text{otherwise} \end{cases}$$

Theorem 6. isKUNSAT(under($\phi, core$)) *implies* isKUNSAT(ϕ).

The intuition of the proof is that each selector s_i enables an operand in a conjunction of the formula. Each time function 'under' is called with a non-empty *core*, operands not enabled with a selector from the *core* will be removed from the formula.

Proof. Let ϕ be in NNF. We show that isKSAT(ϕ) implies isKSAT(under($\phi, core$)) by induction on the structure of ϕ. Assume isKSAT(ϕ). Then $\exists\, M, w$ s.t. $\langle M, w\rangle \models \phi$. There are two cases in the induction base: (1) $\phi = p$ and (2) $\phi = \neg p$. In both of them under($\phi, core$) $= \phi$. There are four cases in the induction step:
(1) $\phi = \Diamond_a(\psi)$. $\exists\, M, w$ s.t. $\langle M, w\rangle \models \Diamond_a(\psi)$. Then $\exists\, M, w'$ s.t. $(w, w') \in R$, $\langle M, w'\rangle \models \psi$. Then $\langle M, w'\rangle \models$ under($\psi, core$) by induction hypothesis. Thus $\langle M, w\rangle \models$ under($\phi, core$);
(2) $\phi = \Box_a(\psi)$. This case is analogous to (1).
(3) $\phi = (\psi \wedge \chi)$. $\exists\, \langle M, w\rangle \models (\psi \wedge \chi)$. Then $\langle M, w\rangle \models \psi$ and $\langle M, w\rangle \models \chi$. Then $\langle M, w\rangle \models$ under($\psi, core$) and $\langle M, w\rangle \models$ under($\chi, core$) by induction hypothesis. Thus $\langle M, w\rangle \models$ under($\phi, core$);
(4) $\phi = (\psi \vee \chi)$. We consider the three cases:
(4.a) $\psi = \neg s_i$ and $s_i \in core$. Then $\exists\, \langle M, w\rangle \models (\neg s_i \vee \chi)$ but $s_i \in V(w)$, then $\langle M, w\rangle \models \chi$. Then $\langle M, w\rangle \models$ under($\chi, core$) by induction hypothesis. Thus $\langle M, w\rangle \models$ under($\phi, core$);
(4.b) $\psi = \neg s_i$ and $s_i \notin core$. $\exists\, \langle M, w\rangle \models (\neg s_i \vee \chi)$. but we always have $\langle M, w\rangle \models \top$. Thus $\langle M, w\rangle \models$ under($\phi, core$).
(4.c) This case is analogous to (3). □

Theo. 6 shows that function 'under' satisfies RECAR Assump. 4. To see that it also satisfies Assump. 5, note that the length of under$^{n+1}(\phi, core)$ is smaller or equal to that of under$^n(\phi, core')$ (even though the sets *core* and *core*′ usually differ).

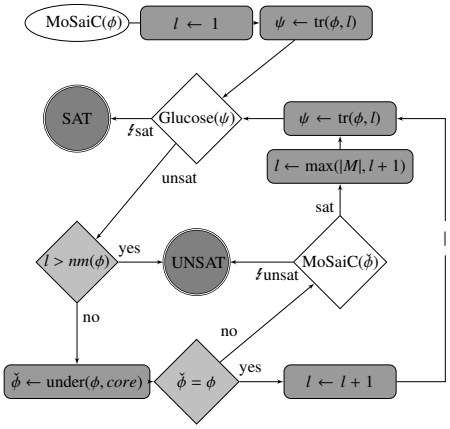


Figure 5: MoSaiC : RECAR for modal logic K

5 MoSaiC : RECAR for K-SAT

We implemented the RECAR approach for modal logic K satisfiability problem within the solver MoSaiC, using the over and under approximations defined in the previous section. MoSaiC combines several features found in state-of-the-art solvers. As in Km2SAT [Sebastiani and Vescovi, 2009], it optimises the input by performing the rules: Box Lifting, Flattening, and Truth Propagation through modal and Boolean operators (see [Sebastiani and Vescovi, 2009] for more details). MoSaiC also uses the SAT solver Glucose in incremental mode [Eén and Sörensson, 2003; Audemard *et al.*, 2013] to decide the satisfiability of each ψ.

Note that some implementation details differ a bit from Fig. 5. For instance, we do not call Glucose on ψ but on an updated ψ' with selectors on conjuncts under the assumption that these selectors are satisfied; we do not need to generate the under approximation $\check{\phi}$ to test the condition $\check{\phi} = \phi$: we just need to know the number of selectors involved in the unsatisfiability of the formula. We also return a Kripke model in the main procedure, not just SAT/UNSAT. We take advantage of such information to provide a new bound for l. And finally, note that in our case $\max(|M|, l+1)$ always returns $|M|$ because it is not possible to find a model smaller than M by construction of $\check{\phi}$. The CEGAR solver presented in Sec. 6 uses a similar schema, but without the recursive RECAR step.

6 Experiments

We compared our approach against existing solvers considered state-of-the-art in [Nalon *et al.*, 2016], namely $K_S P$ 0.1 [Nalon *et al.*, 2016], BDDTab 1.0 [Goré *et al.*, 2014], FaCT++ 1.6.4 [Tsarkov and Horrocks, 2006], InKreSAT 1.0 [Kaminski and Tebbi, 2013], *SAT 1.3 [Giunchiglia *et al.*, 2002], Km2SAT 1.0 [Sebastiani and Vescovi, 2009] combined with the same Glucose SAT solver we use in MoSaiC, Spartacus 1.0 [Götzmann *et al.*, 2010] and a combination of the optimized functional translation [Horrocks *et al.*, 2007] with Vampire 4.0 [Kovács and Voronkov, 2013]. MoSaiC was configured as a standard CEGAR approach or with the proposed RECAR approach. We chose to execute these solvers on the following benchmarks: the complete set of TANCS-2000

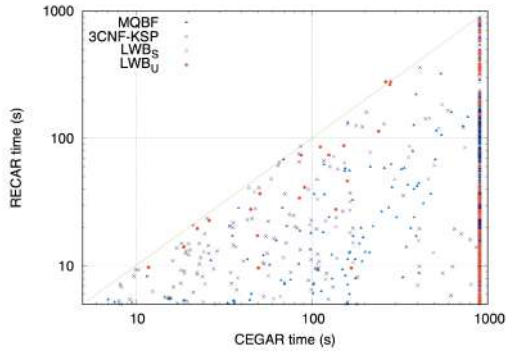


Figure 6: Scatter-plot CEGAR vs RECAR

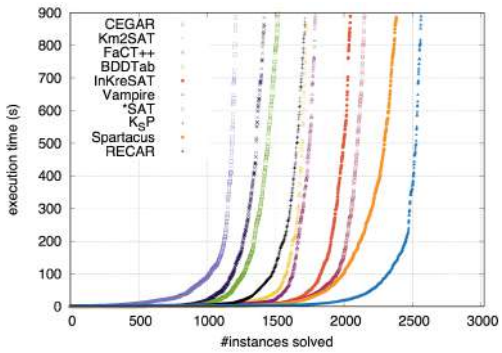


Figure 7: Runtime distribution on all the benchmarks

modalised random QBF (MQBF) formulae [Massacci and Donini, 2000] complemented by the additional MQBF formulae provided by [Kaminski and Tebbi, 2013], 1016 formulae, 617 satisfiable, 399 unsatisfiable; LWB basic modal logic benchmark formulae [Balsiger *et al.*, 2000], with 56 formulae chosen from each of the 18 parametrized classes, generated from the script given by the authors of [Nalon *et al.*, 2016], 1008 formulae, 504 satisfiable, 504 unsatisfiable; randomly generated $3CNF_{KSP}$ formulae [Patel-Schneider and Sebastiani, 2011] of depth 1 or 2, 1000 formulae, 457 satisfiable, 464 unsatisfiable. Few of the considered solvers could deal with multiple modalities like MoSaiC, however to the best of our knowledge, there is no standard benchmark for modal logic K containing multiple modalities. We used a memory limit of 32GB and a runtime limit of 900 seconds per solver per benchmark. Note that due to lack of space, the results presented here are global[2]. The behavior of the solvers vary a lot with the benchmark families. We believe however that they provide an interesting insight of the capabilities of the proposed approach.

RECAR vs CEGAR We can see on Fig. 6 that for most benchmarks, the RECAR approach outperforms the CEGAR one. The under approximation often provides a formula with a much smaller nm value, which produces a CNF of reasonable size to be handed in to the SAT solver. Note that in this plot, memory out for the CEGAR approach is denoted by a timeout, i.e. a point at 900s. This is mainly due to improvements in solving unsatisfiable benchmarks (1118/1367) for RECAR vs (155/1367) for CEGAR. For satisfiable benchmarks, the bound update resulting from the recursive call helps to reach faster a satisfiable formula. 1446 for RECAR vs 1053 for CEGAR.

Comparison with state-of-the-art We can see on Fig. 7 that our over-approximation CEGAR approach is the worst solver whereas our RECAR approach outperforms the other solvers. *Km2SAT* performs specific reasoning to detect earlier some UNSAT benchmarks without generating the CNF, which explains why it performs much better than our CEGAR approach. *SAT interleaves SAT reasoning and domain reasoning, and can be considered as an under-approximation CEGAR approach. It shows good results, despite being tied with the old SAT solver SATO[3]. Our best competitor, Spartacus, is based on a tableaux method, not on SAT: SAT based-techniques were not the best way to tackle such problems up to now. Spartacus reaches the timeout on unsolved benchmarks while we exhaust the available memory: the solvers behave quite differently and have different limits.

7 Conclusion

We proposed here a new approach to solve decision problems using a recursive abstraction refinement approach. We showed it is sound and complete and we instantiated it for the modal logic K satisfiability problem. We compared our approach against solvers representing, to the best of our knowledge, the state-of-the-art for practical K-SAT solving, on a wide range of classical modal logic benchmarks. A basic over-approximation based CEGAR approach is not competitive at all, because many of the available benchmarks are UNSAT. Our RECAR approach, mixing SAT and UNSAT shortcuts, outperformed the other solvers on the benchmarks considered. Those promising results are a first step toward more efficient modal logic solvers: MoSaiC can be extended to other modal logics such as KT, S4, S5 and KD45, by adapting the current translation into CNF. We also believe that RECAR is an appealing framework for tackling decision problems above NP in the polynomial hierarchy. It is not clear yet for us if existing CEGAR related approaches for QBF such as [Janota *et al.*, 2016] could fit in such framework. This is an exciting future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and for detecting a flaw which is now corrected. Part of this work was supported by the ANR project SATAS (ANR-15-CE40-0017), the French Ministry for Higher Education and Research and the Nord-Pas de Calais Regional Council through the "Contrat de Plan État Région (CPER)" and by an EC FEDER grant.

[2] Additional results can be found on `http://www.cril.univ-artois.fr/~montmirail/mosaic`

[3] *SAT is deeply integrated with SATO, which makes very difficult an update to a more recent SAT solver.

References

[Audemard *et al.*, 2013] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Proc. of SAT'13*, pages 309–317, 2013.

[Balsiger *et al.*, 2000] Peter Balsiger, Alain Heuerding, and Stefan Schwendimann. A Benchmark Method for the Propositional Modal Logics K, KT, S4. *J. Autom. Reasoning*, 24(3):297–317, 2000.

[Biere *et al.*, 2009] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Brummayer and Biere, 2009] Robert Brummayer and Armin Biere. Effective bit-width and under-approximation. In *Proc. of EUROCAST'09*, pages 304–311, 2009.

[Caridroit *et al.*, 2017] Thomas Caridroit, Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima, and Valentin Montmirail. A SAT-based Approach For Solving The Modal Logic S5-Satisfiability Problem. In *Proc. of AAAI'17*, 2017.

[Clarke *et al.*, 2003] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[Eén and Sörensson, 2003] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proc. of SAT'03*, pages 502–518, 2003.

[Giunchiglia *et al.*, 2002] Enrico Giunchiglia, Armando Tacchella, and Fausto Giunchiglia. SAT-Based Decision Procedures for Classical Modal Logics. *J. Autom. Reasoning*, 28(2):143–171, 2002.

[Goré *et al.*, 2014] Rajeev Goré, Kerry Olesen, and Jimmy Thomson. Implementing Tableau Calculi Using BDDs: BDDTab System Description. In *Proc. of IJCAR'14*, pages 337–343, 2014.

[Götzmann *et al.*, 2010] Daniel Götzmann, Mark Kaminski, and Gert Smolka. Spartacus: A Tableau Prover for Hybrid Logic. *Electr. Notes Theor. Comput. Sci.*, 262:127–139, 2010.

[Halpern and Moses, 1992] Joseph Y. Halpern and Yoram Moses. A Guide to Completeness and Complexity for Modal Logics of Knowledge and Belief. *Artif. Intell.*, 54(2):319–379, 1992.

[Horrocks *et al.*, 2007] Ian Horrocks, Ullrich Hustadt, Ulrike Sattler, and Renate A. Schmidt. 4 Computational modal logic. *Studies in Logic and Practical Reasoning*, 3:181–245, 2007.

[Janota *et al.*, 2016] Mikolás Janota, William Klieber, Joao Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.

[Järvisalo *et al.*, 2012] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012.

[Kaminski and Tebbi, 2013] Mark Kaminski and Tobias Tebbi. InKreSAT: Modal Reasoning via Incremental Reduction to SAT. In *Proc. of CADE'13*, pages 436–442, 2013.

[Khasidashvili *et al.*, 2015] Zurab Khasidashvili, Konstantin Korovin, and Dmitry Tsarkov. EPR-based k-induction with Counterexample Guided Abstraction Refinement. In *Proc. of GCAI'15*, volume 36, pages 137–150, 2015.

[Kovács and Voronkov, 2013] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV'13*, pages 1–35, 2013.

[Ladner, 1977] Richard E. Ladner. The Computational Complexity of Provability in Systems of Modal Propositional Logic. *SIAM J. Comput.*, 6(3):467–480, 1977.

[Massacci and Donini, 2000] Fabio Massacci and Francesco M. Donini. Design and results of TANCS-2000 non-classical (modal) systems comparison. In *Proc. of TABLEAUX'00*, pages 52–56, 2000.

[Nalon *et al.*, 2016] Cláudia Nalon, Ullrich Hustadt, and Clare Dixon. $K_S P$: A Resolution-Based Prover for Multimodal K. In *Proc. of IJCAR'16*, pages 406–415, 2016.

[Patel-Schneider and Sebastiani, 2011] Peter F. Patel-Schneider and Roberto Sebastiani. A new general method to generate random modal formulae for testing decision procedures. *CoRR*, abs/1106.5261, 2011.

[Prestwich, 2009] Steven David Prestwich. CNF encodings. In *Handbook of Satisfiability*, pages 75–97. IOS Press, 2009.

[Pulina, 2016] Luca Pulina. The ninth QBF solvers evaluation - preliminary report. In *Proc. of the 4th International Workshop (QBF 2016) co-located with (SAT 2016)*, pages 1–13, 2016.

[Sebastiani and Tacchella, 2009] Roberto Sebastiani and Armando Tacchella. SAT techniques for modal and description logics. In *Handbook of Satisfiability*, pages 781–824. IOS Press, 2009.

[Sebastiani and Vescovi, 2009] Roberto Sebastiani and Michele Vescovi. Automated Reasoning in Modal and Description Logics via SAT Encoding: the Case Study of K(m)/ALC-Satisfiability. *J. Artif. Intell. Res. (JAIR)*, 35:343–389, 2009.

[Seipp and Helmert, 2013] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement. In *Proc. of ICAPS'13*, 2013.

[Tsarkov and Horrocks, 2006] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proc. of IJCAR'06*, pages 292–297, 2006.

[Wang *et al.*, 2007] Chao Wang, Aarti Gupta, and Franjo Ivancic. Induction in CEGAR for detecting counterexamples. In *Proc. of FMCAD'07*, pages 77–84, 2007.

Bibliographie

[proc-ijcai07 2007] : *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*. Hyderabad, India, Januar 6-16 2007

[Abate et al. 2007] Abate, Pietro ; Goré, Rajeev ; Widmann, Florian : Cut-Free Single-Pass Tableaux For The Logic of Common Knowledge. In : *Workshop on Agents and Deduction (WAD@TABLEAUX'07)*, Springer, July 2007, p. 1–20. – URL http://users.cecs.anu.edu.au/~rpg/Submissions/lck.pdf

[Amarilli et al. 2018] Amarilli, Antoine ; Capelli, Florent ; Monet, Mikaël ; Senellart, Pierre : Connecting Knowledge Compilation Classes and Width Parameters. In : *CoRR* abs/1811.02944 (2018). – URL http://arxiv.org/abs/1811.02944

[Apsel et Brafman 2012] Apsel, U. ; Brafman, R.I. : Lifted MEU by Weighted Model Counting. In : *Proc. of AAAI'12*, 2012

[Areces et al. 2015] Areces, Carlos ; Fontaine, Pascal ; Merz, Stephan : *Modal Satisfiability via SMT Solving*. In : De Nicola, Rocco (Editor) ; Hennicker, Rolf (Editor) : *Software, Services, and Systems : Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, 2015

[Argelich et al. 2010] Argelich, Josep ; Le Berre, Daniel ; Lynce, Inês ; Marques Silva, João P. ; Rapicault, Pascal : Solving Linux Upgradeability Problems Using Boolean Optimization. In : Lynce, Inês (Editor) ; Treinen, Ralf (Editor) : *Proceedings First International Workshop on Logics for Component Configuration* Volume 29, 2010, p. 11–22

[Astesana et al. 2010] Astesana, Jean-Marc ; Cosserat, Laurent ; Fargier, Hélène : Constraint-based Vehicle Configuration : A Case Study. In : *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, 2010, p. 68–75

[Audemard et al. 2014a] Audemard, Gilles ; Biere, Armin ; Lagniez, Jean-Marie ; Simon, Laurent : Améliorer SAT dans le cadre incrémental. In : *Revue d'Intelligence Artificielle* 28 (2014), Nr. 5, p. 593–614

[Audemard et al. 2008] Audemard, Gilles ; Bordeaux, Lucas ; Hamadi, Youssef ; Jabbour, Saïd ; Saïs, Lakhdar : A generalized framework for conflict analysis. In : *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*. Berlin, Heidelberg : Springer-Verlag, 2008 (SAT'08), p. 21–27

[Audemard et al. 2012] Audemard, Gilles ; Hoessen, Benoît ; Jabbour, Saïd ; Lagniez, Jean-Marie ; Piette, Cédric : Revisiting Clause Exchange in Parallel SAT Solving. In : *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, p. 200–213

[Audemard et al. 2014b] Audemard, Gilles ; Hoessen, Benoît ; Jabbour, Saïd ; Piette, Cédric : Dolius : A Distributed Parallel SAT Solving Framework. In : *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, 2014, p. 1–11

[Audemard et al. 2010a] AUDEMARD, Gilles ; KATSIRELOS, George ; SIMON, Laurent : A Restriction of Extended Resolution for Clause Learning SAT Solvers. In : *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010

[Audemard et al. 2009a] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; MAZURE, Bertrand ; SAIS, Lakhdar : Integrating Conflict Driven Clause Learning to Local Search. In : *Proceedings 6th International Workshop on Local Search Techniques in Constraint Satisfaction, LSCS 2009, Lisbon, Portugal, 20 September 2009.*, 2009, p. 55–68

[Audemard et al. 2009b] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; MAZURE, Bertrand ; SAIS, Lakhdar : Learning in Local Search. In : *ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, 2-4 November 2009*, 2009, p. 417–424

[Audemard et al. 2010b] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; MAZURE, Bertrand ; SAIS, Lakhdar : Boosting Local Search Thanks to CDCL. In : *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, 2010, p. 474–488

[Audemard et al. 2011a] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; MAZURE, Bertrand ; SAÏS, Lakhdar : On freezeing and reactivating learnt clauses. In : *Fourteenth International Conference on Theory and Applications of Satisfiability Testing(SAT'11)*, jun 2011

[Audemard et al. 2011b] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; MAZURE, Bertrand ; SAIS, Lakhdar : On Freezing and Reactivating Learnt Clauses. In : *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, 2011, p. 188–200

[Audemard et al. 2013a] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; SIMON, Laurent : Improving Glucose for Incremental SAT Solving with Assumptions : Application to MUS Extraction. In : *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 309–317

[Audemard et al. 2013b] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; SIMON, Laurent : Improving Glucose for Incremental SAT Solving with Assumptions : Application to MUS Extraction. In : *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 309–317

[Audemard et al. 2013c] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; SIMON, Laurent : Just-In-Time Compilation of Knowledge Bases. In : *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 447–453

[Audemard et al. 2016] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; SZCZEPANSKI, Nicolas ; TABARY, Sébastien : An Adaptive Parallel SAT Solver. In : *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, 2016, p. 30–48

[Audemard et al. 2017] AUDEMARD, Gilles ; LAGNIEZ, Jean-Marie ; SZCZEPANSKI, Nicolas ; TABARY, Sébastien : A Distributed Version of Syrup. In : *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 215–232

[Audemard et Simon 2009a] Audemard, Gilles ; Simon, Laurent : Glucose : a solver that predicts learnt clauses quality. 2009. – Research Report. SAT 2009 Competition Event Booklet, http//www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf

[Audemard et Simon 2009b] Audemard, Gilles ; Simon, Laurent : Predicting Learnt Clauses Quality in Modern SAT Solver. In : *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, jul 2009, p. 399–404

[Audemard et Simon 2012] Audemard, Gilles ; Simon, Laurent : Refining Restarts Strategies for SAT and UNSAT. In : *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, 2012, p. 118–126

[Audemard et Simon 2014] Audemard, Gilles ; Simon, Laurent : Lazy Clause Exchange Policy for Parallel SAT Solvers. In : *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014, p. 197–205

[Aziz et al. 2015] Aziz, Rehan A. ; Chu, Geoffrey ; Muise, Christian J. ; Stuckey, Peter J. : #∃SAT : Projected Model Counting. In : *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, 2015, p. 121–137

[Baader et Hollunder 1991] Baader, Franz ; Hollunder, Bernhard : A Terminological Knowledge Representation System with Complete Inference Algorithms. In : *Processing Declarative Knowledge, International Workshop PDK'91, Kaiserslautern, Germany, July 1-3, 1991, Proceedings*, 1991, p. 67–86

[Babka et al. 2013] Babka, Martin ; Balyo, Tomás ; Cepek, Ondrej ; Gurský, Stefan ; Kucera, Petr ; Vlcek, Václav : Complexity issues related to propagation completeness. In : *Artif. Intell.* 203 (2013), p. 19–34

[Bacchus et al. 2014] Bacchus, Fahiem ; Davies, Jessica ; Tsimpoukelli, Maria ; Katsirelos, George : Relaxation Search : A Simple Way of Managing Optional Clauses. In : *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 835–841

[Bacchus et Katsirelos 2015] Bacchus, Fahiem ; Katsirelos, George : Using Minimal Correction Sets to More Efficiently Compute Minimal Unsatisfiable Sets. In : *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, 2015, p. 70–86

[Bacchus et Katsirelos 2016] Bacchus, Fahiem ; Katsirelos, George : Finding a Collection of MUSes Incrementally. In : *Proc. of CPAIOR 2016*, 2016, p. 35–44

[Bacchus et Winter 2003] Bacchus, Fahiem ; Winter, Jonathan : Effective Preprocessing with Hyper-Resolution and Equality Reduction. In : *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, 2003, p. 341–355

[Balafrej et al. 2015] Balafrej, Amine ; Bessière, Christian ; Paparrizou, Anastasia : Multi-Armed Bandits for Adaptive Constraint Propagation. In : *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, p. 290–296

[Balsiger et al. 1998] Balsiger, Peter ; Heuerding, Alain ; Schwendimann, Stefan : Logics Workbench 1.0. In : *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '98, Oisterwijk, The Netherlands, May 5-8, 1998, Proceedings*, 1998, p. 35–37

[Balsiger et al. 2000] Balsiger, Peter ; Heuerding, Alain ; Schwendimann, Stefan : A Benchmark Method for the Propositional Modal Logics K, KT, S4. In : *J. Autom. Reasoning* 24 (2000), Nr. 3, p. 297–317

[Balyo et al. 2015] Balyo, Tomás ; Sanders, Peter ; Sinz, Carsten : HordeSat : A Massively Parallel Portfolio SAT Solver. In : *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, 2015, p. 156–172

[Barrett et al. 2009] Barrett, Clark W. ; Sebastiani, Roberto ; Seshia, Sanjit A. ; Tinelli, Cesare : Satisfiability Modulo Theories. In : *Handbook of Satisfiability*. 2009, p. 825–885

[Bart et al. 2014] Bart, Anicet ; Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre : Symmetry-Driven Decision Diagrams for Knowledge Compilation. In : *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, 2014, p. 51–56

[Bart et al. 2016] Bart, Anicet ; Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre : An Improved CNF Encoding Scheme for Probabilistic Inference. In : *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, 2016, p. 613–621

[Bayardo Jr. et Schrag 1997] Bayardo Jr., Roberto J. ; Schrag, Robert C. : Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In : *Proceedings of the Fourteenth American National Conference on Artificial Intelligence (AAAI'97)*. Providence (Rhode Island, USA) : AAAI Press / The MIT Press, Juli 1997, p. 203–208

[Beame et al. 2004] Beame, Paul ; Kautz, Henry A. ; Sabharwal, Ashish : Towards Understanding and Harnessing the Potential of Clause Learning. In : *Journal of Artificial Intelligence Research* 22 (2004), p. 319–351

[Beckers et al. 2011] Beckers, Sander ; De Samblanx, Gorik ; De Smedt, Floris ; Goedemé, Toon ; Struyf, Lars ; Vennekens, Joost : Parallel SAT-solving with OpenCL. In : *Proceedings of the IADIS International Conference on Applied Computing 2011*, 2011, p. 435–441

[Belahcène et al. 2018] Belahcène, Khaled ; Chevaleyre, Yann ; Labreuche, Christophe ; Maudet, Nicolas ; Mousseau, Vincent ; Ouerdane, Wassila : Accountable Approval Sorting. In : *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, p. 70–76

[Belov et Marques-Silva 2011] Belov, Anton ; Marques-Silva, João : Accelerating MUS extraction with recursive model rotation. In : *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, 2011, p. 37–40

[Benferhat et al. 2005] Benferhat, Salem ; Ben-Naim, Jonathan ; Jeansoulin, Robert ; Khelfallah, Mahat ; Lagrue, Sylvain ; Papini, Odile ; Wilson, Nic ; Würbel, Éric : Belief Revision of GIS Systems : the Results of REVIGIS. In : (ed.), Lluis G. (Editor) : *8th European Conferences on Symbolic and Quantitative Approaches to Reasoning with Uncertainty(ECSQARU'05)*, Springer, jul 2005, p. 452–464. – Lecture Notes in Computer Science, Vol. 3571

[Besnard et al. 2015] Besnard, Philippe ; Grégoire, Éric ; Lagniez, Jean-Marie : On Computing Maximal Subsets of Clauses that Must Be Satisfiable with Possibly Mutually-Contradictory Assumptive Contexts. In : *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 2015, p. 3710–3716

[Besnard et al. 2010] Besnard, Philippe ; Grégoire, Éric ; Piette, Cédric ; Raddaoui, Badran : MUS-Based Generation of Arguments and Counter-arguments. In : *11th IEEE International Conference on Information Reuse and Integration(IRI'10)*. Las Vegas (USA), aug 2010, p. 239–244

[Besnard et Hunter 2008] Besnard, Philippe ; Hunter, Anthony : *Elements of Argumentation*. MIT Press, 2008

[Beth 1953] Beth, E.W. : On Padoa's method in the theory of definition. In : *Indagationes mathematicae* 15 (1953), p. 330–339

[Biere 2008a] Biere, Armin : Adaptive Restart Strategies for Conflict Driven SAT Solvers. In : Kleine Büning, Hans (Editor) ; Zhao, Xishun (Editor) : *Theory and Applications of Satisfiability Testing - SAT 2008* Volume 4996, Springer Berlin / Heidelberg, 2008, p. 28–33

[Biere 2008b] Biere, Armin : PicoSAT essentials. In : *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), Nr. 75-97, p. 45

[Biere 2009] Biere, Armin : *Precosat system description*. SAT Competition, solver description. 2009

[Biere 2017] Biere, Armin : CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In : Balyo, Tomáš (Editor) ; Heule, Marijn (Editor) ; Järvisalo, Matti (Editor) : *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions* Volume B-2017-1, University of Helsinki, 2017, p. 14–15

[Biere et al. 1999a] Biere, Armin ; Cimatti, Alessandro ; Clarke, Edmund M. ; Fujita, Masahiro ; Zhu, Yunshan : Symbolic model checking using SAT procedures instead of BDDs. In : *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA : ACM, 1999 (DAC '99), p. 317–320

[Biere et al. 1999b] Biere, Armin ; Cimatti, Alessandro ; Clarke, Edmund M. J. ; Fujita, Masahiro ; Zhu, Yunshan : Symbolic Model Checking using SAT Procedures Instead of BDDs. In : *Design Automation Conference* 0 (1999), p. 317–320

[Biere et Fröhlich 2015] Biere, Armin ; Fröhlich, Andreas : Evaluating CDCL Variable Scoring Schemes. In : *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, 2015, p. 405–422

[Biere et al. 2009] Biere, Armin (Editor) ; Heule, Marijn J. H. (Editor) ; Maaren, Hans van (Editor) ; Walsh, Toby (Editor) : *Frontiers in Artificial Intelligence and Applications*. Volume 185 : *Handbook of Satisfiability*. IOS Press, Februar 2009. – 980 p. – ISBN 978-1-58603-929-5

[Birnbaum et Lozinskii 2003] Birnbaum, Elazar ; Lozinskii, Eliezer L. : Consistent subsets of inconsistent systems : structure and behaviour. In : *J. Exp. Theor. Artif. Intell.* 15 (2003), Nr. 1, p. 25–46

[Blackburn et al. 2006] Blackburn, Patrick ; Benthem, Johan van ; Wolter, Frank : *Handbook of Modal Logic*. Volume 3. Elsevier, 2006. – URL https://hal.inria.fr/inria-00120237. – ISBN 978-0444516909

[Blochinger et al. 2003] Blochinger, Wolfgang ; Sinz, Carsten ; Küchlin, Wolfgang : Parallel propositional satisfiability checking with distributed dynamic learning. In : *Parallel Comput.* 29 (2003), July, p. 969–994. – ISSN 0167-8191

[Bolander et Andersen 2011] Bolander, Thomas ; Andersen, Mikkel B. : Epistemic planning for single and multi-agent systems. In : *Journal of Applied Non-Classical Logics* 21 (2011), Nr. 1, p. 9–34

[Boussemart et al. 2004] Boussemart, Frédéric ; Hemery, Frédéric ; Lecoutre, Christophe ; Sais, Lakhdar : Boosting systematic search by weighting constraints. In : *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI'99)*. Valence, Spain, August 2004, p. 482–486

[Bradley 2012] Bradley, Aaron R. : IC3 and beyond : Incremental, Inductive Verification. In : *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, 2012, p. 4

[Brand et al. 2003] Brand, Sebastian ; Gennari, Rosella ; Rijke, Maarten de : Constraint Methods for Modal Satisfiability. In : *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2003, Budapest, Hungary, June 30 - July 2, 2003, Selected Papers*, 2003, p. 66–86

[Brandes et Erlebach 2005] Brandes, Ulrik (Editor) ; Erlebach, Thomas (Editor) : *Network Analysis : Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004]*. Volume *3418*. Springer, 2005. (Lecture Notes in Computer Science)

[Brandt et al. 2016] Brandt, Felix (Editor) ; Conitzer, Vincent (Editor) ; Endriss, Ulle (Editor) ; Lang, Jérôme (Editor) ; Procaccia, Ariel D. (Editor) : *Handbook of Computational Social Choice*. Cambridge University Press, 2016

[Brisoux et al. 1999] Brisoux, Laure ; Grégoire, Éric ; Saïs, Lakhdar : Improving Backtrack Search for SAT by Means of Redundancy. In : *ISMIS '99 : Proceedings of the 11th International Symposium on Foundations of Intelligent Systems*. London, UK : Springer-Verlag, 1999, p. 301–309. – ISBN 3-540-65965-X

[Brummayer et Biere 2009] Brummayer, Robert ; Biere, Armin : Effective Bit-Width and Under-Approximation. In : *Computer Aided Systems Theory - EUROCAST 2009, 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers*, 2009, p. 304–311

[Bryant 1986] Bryant, Randal E. : Graph-Based Algorithms for Boolean Function Manipulation. In : *IEEE Trans. Computers* 35 (1986), Nr. 8, p. 677–691

[Bylander 1994] Bylander, Tom : The Computational Complexity of Propositional STRIPS Planning. In : *Artif. Intell.* 69 (1994), Nr. 1-2, p. 165–204

[Cadoli et Donini 1997] Cadoli, Marco ; Donini, Francesco M. : A Survey on Knowledge Compilation. In : *AI Commun.* 10 (1997), Nr. 3-4, p. 137–150

[Cadoli et al. 2002] Cadoli, Marco ; Donini, Francesco M. ; Liberatore, Paolo ; Schaerf, Marco : Preprocessing of Intractable Problems. In : *Inf. Comput.* 176 (2002), Nr. 2, p. 89–120

[Cannière 2006] Cannière, Christophe D. : Trivium : A Stream Cipher Construction Inspired by Block Cipher Design Principles. In : *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, 2006, p. 171–186

[Capelli et al. 2014] Capelli, Florent ; Durand, Arnaud ; Mengel, Stefan : Hypergraph Acyclicity and Propositional Model Counting. In : *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014, p. 399–414

[Caridroit et al. 2017] Caridroit, Thomas ; Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin : A SAT-Based Approach for Solving the Modal Logic S5-Satisfiability Problem. In : *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, p. 3864–3870

[Castell 1996] Castell, Thierry : Computation of Prime Implicates and Prime Implicants by a variant of the Davis and Putnam procedure. In : *IEEE International Conference on Tools with Artificial Intelligence TAI'96, Toulouse France, 16/11/96-19/11/96*. Los Alamitos, California : IEEE Computer Society Press, novembre 1996, p. 428–429

[Çatalyürek et Aykanat 2011] Çatalyürek, Ümit V. ; Aykanat, Cevdet : PaToH (Partitioning Tool for Hypergraphs). In : *Encyclopedia of Parallel Computing*. 2011, p. 1479–1487

[Chellas 1980] Chellas, Brian F. : *Modal Logic : An Introduction*. Cambridge University Press, 1980. – URL https://books.google.fr/books?id=v4YIAQAAIAAJ. – ISBN 978-0521295154

[Chen et Marques-Silva 2012] Chen, Huan ; Marques-Silva, João : New & improved models for SAT-based bi-decomposition. In : *Great Lakes Symposium on VLSI 2012, GLSVLSI'12, Salt Lake City, UT, USA, May 3-4, 2012*, 2012, p. 141–146

[Chen et al. 2001] Chen, Hubie ; Gomes, Carla ; Selman, Bart : Formal Models of Heavy-Tailed Behavior in Combinatorial Search. In : Walsh, Toby (Editor) : *Principles and Practice of Constraint Programming - CP 2001* Volume 2239, Springer Berlin / Heidelberg, 2001, p. 408–421

[Chen 2007] Chen, Jing-Chao : XORSAT : An Efficient Algorithm for the DIMACS 32-bit Parity Problem. In : *CoRR* abs/cs/0703006 (2007)

[Chen et Toda 1995] Chen, Zhi-Zhong ; Toda, Seinosuke : The Complexity of Selecting Maximal Solutions. In : *Inf. Comput.* 119 (1995), Nr. 2, p. 231–239

[Choi et al. 2013] Choi, A. ; Kisa, D. ; Darwiche, A. : Compiling Probabilistic Graphical Models Using Sentential Decision Diagrams. In : *Proc. of ECSQARU'13*, 2013, p. 121–132

[Chrabakh et Wolski 2003] Chrabakh, Wahid ; Wolski, Rich : GridSAT : A Chaff-based Distributed SAT Solver for the Grid. In : *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. New York, NY, USA : ACM, 2003 (SC '03), p. 37–. – ISBN 1-58113-695-1

[Chu et Stuckey 2008] Chu, Geoffrey ; Stuckey, Peter J. : Pminisat : A parallelization of minisat 2.0. 2008. – Solver Description, SAT-race 2008

[Clarke et al. 2003] Clarke, Edmund M. ; Grumberg, Orna ; Jha, Somesh ; Lu, Yuan ; Veith, Helmut : Counterexample-guided abstraction refinement for symbolic model checking. In : *J. ACM* 50 (2003), Nr. 5, p. 752–794

[Cook 1971] Cook, Stephen A. : The complexity of theorem-proving procedures. In : *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1971, p. 151–158

[Cook 1976] Cook, Stephen A. : A short proof of the pigeon hole principle using extended resolution. In : *SIGACT News* 8 (1976), Nr. 4, p. 28–32

[Crawford et Kearns 1995] Crawford, J.M. ; Kearns, M.J. : The Minimal Disagreement Parity Problem as a Hard Satisfiability Problem. 1995. – Research Report

[Creignou et al. 2001] Creignou, Nadia ; Khanna, Sanjeev ; Sudan, Madhu : *Complexity classifications of boolean constraint satisfaction problems*. Volume 7. Society for Industrial Mathematics, 2001

[Dalal 1992] Dalal, Mukesh : Efficient Propositional Constraint Propagation. In : *Proceedings of the Tenth American National Conference on Artificial Intelligence (AAAI'92)*. San-Jose, California (USA), 1992, p. 409–414

[Darras et al. 2005] Darras, Sylvain ; Dequen, Gilles ; Devendeville, Laure ; Mazure, Bertrand ; Ostrowski, Richard ; Saïs, Lakhdar : Using Boolean Constraint Propagation for Sub-clause Deduction. In : *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)* Volume 3709. Sitges (Barcelona), Spain : Springer, Oktober 2005, p. 757–761

[Darwiche 2004] Darwiche, Adnan : New Advances in Compiling CNF into Decomposable Negation Normal Form. In : *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 2004, p. 328–332

[Darwiche et Hopkins 2001] Darwiche, Adnan ; Hopkins, Mark : Using Recursive Decomposition to Construct Elimination Orders, Jointrees, and Dtrees. In : *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 6th European Conference, ECSQARU 2001, Toulouse, France, September 19-21, 2001, Proceedings*, 2001, p. 180–191

[Darwiche et Marquis 2002] Darwiche, Adnan ; Marquis, Pierre : A Knowledge Compilation Map. In : *J. Artif. Intell. Res.* 17 (2002), p. 229–264

[Davis et al. 1962] Davis, Martin ; Logemann, George ; Loveland, Donald : A Machine Program for Theorem Proving. In : *Journal of the Association for Computing Machinery* 5 (1962), p. 394–397

[Dechter 1990] Dechter, Rina : Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition. In : *Artif. Intell.* 41 (1990), January, p. 273–312

[Dechter 1998] Dechter, Rina : Bucket Elimination : A Unifying Framework for Probabilistic Inference. In : *Learning in Graphical Models*. 1998, p. 75–104

[Delmas et al. 2018] Delmas, Rémi ; Garion, Christophe ; Giet, Josselin : MOLOSS, un solveur pour la satisfiabilité en logique modale. In : *Journées de l'Intelligence Artificielle Fondamentale (JIAF-2018)*, 2018, p. 100–110

[Dequen et Dubois 2004] Dequen, Gilles ; Dubois, Olivier : kcnfs : An Efficient Solver for Random k-SAT Formulae. In : (Giunchiglia et Tacchella 2004), p. 486–501

[Domshlak et Hoffmann 2006] Domshlak, C. ; Hoffmann, J. : Fast Probabilistic Planning through Weighted Model Counting. In : *Proc. of ICAPS'06*, 2006, p. 243–252

[Drummond et Bresina 1990] Drummond, Mark ; Bresina, John L. : Anytime Synthetic Projection : Maximizing the Probability of Goal Satisfaction. In : *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 - August 3, 1990, 2 Volumes.*, 1990, p. 138–144

[Eén et Biere 2005] Eén, Niklas ; Biere, Armin : Effective Preprocessing in SAT Through Variable and Clause Elimination. In : *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)* Volume 3569. St. Andrews, Scottland : Springer, Juni 2005, p. 61–75

[Eén et Sörenson 2004] Eén, Niklas ; Sörenson, Niklas : An Extensible SAT-solver. In : (Giunchiglia et Tacchella 2004), p. 333–336

[Eén et Sörensson 2003] Eén, Niklas ; Sörensson, Niklas : Temporal induction by incremental SAT solving. In : *Electr. Notes Theor. Comput. Sci.* 89 (2003), Nr. 4, p. 543–560

[Ehlers et al. 2014] Ehlers, Thorsten ; Nowotka, Dirk ; Sieweck, Philipp : Communication in Massively-Parallel SAT Solving. In : *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, 2014, p. 709–716

[Eiter et Gottlob 2002] Eiter, Thomas ; Gottlob, Georg : Hypergraph Transversal Computation and Related Problems in Logic and AI. In : *JELIA '02 : Proceedings of the European Conference on Logics in Artificial Intelligence* Volume 2424. London, UK : Springer-Verlag, 2002, p. 549–564. – ISBN 3-540-44190-5

[Fargier et Marquis 2008a] Fargier, Hélène ; Marquis, Pierre : Extending the Knowledge Compilation Map : Closure Principles. In : *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, 2008, p. 50–54

[Fargier et Marquis 2008b] Fargier, Hélène ; Marquis, Pierre : Extending the Knowledge Compilation Map : Krom, Horn, Affine and Beyond. In : *Proceedings of the Twenty-Third*

AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008, 2008, p. 442–447

[Fazekas et al. 2019] Fazekas, Katalin ; Biere, Armin ; Scholl, Christoph : Incremental Inprocessing in SAT Solving. In : *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, 2019, p. 136–154

[Feiten et al. 2012] Feiten, L. ; Sauer, M. ; Schubert, T. ; Czutro, A. ; Böhl, E. ; Polian, I. ; Becker, B. : #SAT-based vulnerability analysis of security components - A case study. In : *Proc. of DFT'12*, 2012, p. 49–54

[Felfernig et al. 2012] Felfernig, Alexander ; Schubert, Monika ; Zehentner, Christoph : An efficient diagnosis algorithm for inconsistent constraint sets. In : *AI EDAM* 26 (2012), Nr. 1, p. 53–62

[Fermé et Hansson 2011] Fermé, Eduardo L. ; Hansson, Sven O. : AGM 25 Years - Twenty-Five Years of Research in Belief Change. In : *J. Philosophical Logic* 40 (2011), Nr. 2, p. 295–331

[Freeman 1995] Freeman, Jon W. : *Improvements to Propositional Satisfiability Search Algorithms*. Department of Computer and Information Science, University of Pennsylvania, Ph.D. Thesis, 1995

[Frioux et al. 2017] Frioux, Ludovic L. ; Baarir, Souheib ; Sopena, Julien ; Kordon, Fabrice : PaInleSS : A Framework for Parallel SAT Solving. In : *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 233–250

[Frioux et al. 2019] Frioux, Ludovic L. ; Baarir, Souheib ; Sopena, Julien ; Kordon, Fabrice : Modular and Efficient Divide-and-Conquer SAT Solver on Top of the Painless Framework. In : *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, 2019, p. 135–151

[Fu et Malik 2006] Fu, Zhaohui ; Malik, Sharad : On Solving the Partial MAX-SAT Problem. In : *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, 2006, p. 252–265

[Gasquet et al. 2005] Gasquet, Olivier ; Herzig, Andreas ; Longin, Dominique ; Sahade, Mohamad : LoTREC : Logical Tableaux Research Engineering Companion. In : *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005, Proceedings*, 2005, p. 318–322

[Gebser et al. 2009] Gebser, Martin ; Kaufmann, Benjamin ; Schaub, Torsten : Solution Enumeration for Projected Boolean Search Problems. In : *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings*, 2009, p. 71–86

[Gelder 2005] Gelder, Allen V. : Toward leaner binary-clause reasoning in a satisfiability solver. In : *Ann. Math. Artif. Intell.* 43 (2005), Nr. 1, p. 239–253

[Genesereth et Kao 2016] Genesereth, Michael R. ; Kao, Eric : *Introduction to Logic, Third Edition*. Morgan & Claypool Publishers, 2016 (Synthesis Lectures on Computer Science). – URL https://doi.org/10.2200/S00734ED2V01Y201609CSL008

[Génisson et Siegel 1994] Génisson, Richard ; Siegel, Pierre : A polynomial method for sub-clauses production. In : *Proceedings of Sixth International Conference on Artificila Intelligence : Methodology, Systems, Applications (AIMSA'94)*. North Holland, 1994, p. 25–34

[Gent et Walsh 1994] Gent, Ian P. ; Walsh, Tobby : Easy problems are sometimes hard. In : *Artificial Intelligence* 70 (1994), p. 335–345

[Ginsberg 1987] Ginsberg, Matthew L. (Editor) : *Readings in Nonmonotonic Reasoning*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1987. – ISBN 0-934613-45-1

[Ginsberg 1993] Ginsberg, Matthew L. : Dynamic Backtracking. In : *Journal of Artificial Intelligence Research* 1 (1993), p. 25–46

[Giunchiglia et Tacchella 2001] Giunchiglia, Enrico ; Tacchella, Armando : A Subset-Matching Size-Bounded Cache for Testing Satisfiability in Modal Logics. In : *Ann. Math. Artif. Intell.* 33 (2001), Nr. 1, p. 39–67

[Giunchiglia et Tacchella 2004] Giunchiglia, Enrico (Editor) ; Tacchella, Armando (Editor) : *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*. Volume *2929*. Santa Margherita Ligure, Italy : Springer, Mai 2003. Published in 2004. (Lecture Notes in Computer Science)

[Giunchiglia et al. 2002a] Giunchiglia, Enrico ; Tacchella, Armando ; Giunchiglia, Fausto : SAT-Based Decision Procedures for Classical Modal Logics. In : *Journal of Automated Reasoning* 28 (2002), Feb, Nr. 2, p. 143–171

[Giunchiglia et al. 2002b] Giunchiglia, Enrico ; Tacchella, Armando ; Giunchiglia, Fausto : SAT-Based Decision Procedures for Classical Modal Logics. In : *J. Autom. Reasoning* 28 (2002), Nr. 2, p. 143–171

[Glorian et al. 2017] Glorian, Gael ; Boussemart, Frédéric ; Lagniez, Jean-Marie ; Lecoutre, Christophe ; Mazure, Bertrand : Combining Nogoods in Restart-Based Search. In : *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 129–138

[Glorian et al. 2018] Glorian, Gael ; Lagniez, Jean-Marie ; Montmirail, Valentin ; Sioutis, Michael : An Incremental SAT-Based Approach to Reason Efficiently on Qualitative Constraint Networks. In : *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, 2018, p. 160–178

[Glorian et al. 2019] Glorian, Gaël ; Lagniez, Jean-Marie ; Montmirail, Valentin ; Szczepanski, Nicolas : An Incremental SAT-Based Approach for Graph Colouring Problem. In : *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, Connecticut, U.S., September 30–4 October, Proceedings*, 2019

[Gogic et al. 1995] Gogic, Goran ; Kautz, Henry A. ; Papadimitriou, Christos H. ; Selman, Bart : The Comparative Linguistics of Knowledge Representation. In : *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, 1995, p. 862–869

[Goldberg 1979] Goldberg, Allen : On the Complexity of the Satisfiability Problem / New York University. 1979. – Research Report

[Goldberg et Novikov 2002] Goldberg, Eugene P. ; Novikov, Yakov : BerkMin : a Fast and Robust SAT-Solver. In : *In Proceedings of Design Automation and Test in Europe (DATE'02)*. Paris, 2002, p. 142–149

[Gomes et al. 2007] Gomes, Carla P. ; Hoffmann, Jörg ; Sabharwal, Ashish ; Selman, Bart : From Sampling to Model Counting. In : *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007, p. 2293–2299

[Gomes et al. 2009] Gomes, Carla P. ; Sabharwal, Ashish ; Selman, Bart : *Frontiers in Artificial Intelligence and Applications*. Volume 185 : *Model Counting*. Chap. 20, p. 633–654. see (Biere et al. 2009). – ISBN 978-1-58603-929-5

[Gomes et al. 2000] Gomes, Carla P. ; Selman, Bart ; Crato, Nuno ; Kautz, Henry : Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. In : *J. Autom. Reason.* 24 (2000), February, p. 67–100

[Goré 1999] Goré, Rajeev : *Tableau Methods for Modal and Temporal Logics*. p. 297–396. In : D'Agostino, Marcello (Editor) ; Gabbay, Dov M. (Editor) ; Hähnle, Reiner (Editor) ; Posegga, Joachim (Editor) : *Handbook of Tableau Methods*, Springer, 1999

[Goré et al. 2014] Goré, Rajeev ; Olesen, Kerry ; Thomson, Jimmy : Implementing Tableau Calculi Using BDDs : BDDTab System Description. In : *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, 2014, p. 337–343

[Götzmann et al. 2010] Götzmann, Daniel ; Kaminski, Mark ; Smolka, Gert : Spartacus : A Tableau Prover for Hybrid Logic. In : *Electr. Notes Theor. Comput. Sci.* 262 (2010), p. 127–139

[Grégoire et al. 2016a] Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie : On the Extraction of One Maximal Information Subset That Does Not Conflict with Multiple Contexts. In : *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, p. 3404–3410

[Grégoire et al. 2017] Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie : On Computing One Max_Subset Inclusion Consensus. In : *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, 2017, p. 838–845

[Grégoire et al. 2018a] Grégoire, Éric ; Izza, Yacine ; Lagniez, Jean-Marie : Boosting MCSes Enumeration. In : *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, p. 1309–1315

[Grégoire et Konieczny 2006] Grégoire, Éric ; Konieczny, Sébastien : Logic-based approaches to information fusion. In : *Information Fusion* 7 (2006), Nr. 1, p. 4–18

[Grégoire et al. 2016b] Grégoire, Éric ; Konieczny, Sébastien ; Lagniez, Jean-Marie : On Consensus Extraction. In : *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, p. 1095–1101

[Grégoire et Lagniez 2015] Grégoire, Éric ; Lagniez, Jean-Marie : On Anti-subsumptive Knowledge Enforcement. In : *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, 2015, p. 48–62

[Grégoire et Lagniez 2016a] Grégoire, Éric ; Lagniez, Jean-Marie : A Computational Approach to Consensus-Finding. In : *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands*, 2016, p. 795–801

[Grégoire et Lagniez 2016b] Grégoire, Éric ; Lagniez, Jean-Marie : A Computational Method for Enforcing Knowledge that Cannot be Subsumed. In : *International Journal on Artificial Intelligence Tools* 25 (2016), Nr. 4, p. 1–19

[Grégoire et Lagniez 2016c] Grégoire, Éric ; Lagniez, Jean-Marie : RCL : An A. I. Tool for Computing Maximal Consensuses. In : *International Journal on Artificial Intelligence Tools* 25 (2016), Nr. 4, p. 1–10

[Grégoire et al. 2011] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : A CSP Solver Focusing on FAC Variables. In : *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, 2011, p. 493–507

[Grégoire et al. 2012] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Relax ! In : *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, 2012, p. 146–153

[Grégoire et al. 2013a] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Improving MUC extraction thanks to local search. In : *CoRR* abs/1307.3585 (2013)

[Grégoire et al. 2013b] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Preserving Partial Solutions While Relaxing Constraint Networks. In : *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 552–558

[Grégoire et al. 2013c] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Questioning the Importance of WCORE-Like Minimization Steps in MUC-Finding Algorithms. In : *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, 2013, p. 923–930

[Grégoire et al. 2014a] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : An Experimentally Efficient Method for (MSS, CoMSS) Partitioning. In : *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 2666–2673

[Grégoire et al. 2014b] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Boosting MUC extraction in unsatisfiable constraint networks. In : *Appl. Intell.* 41 (2014), Nr. 4, p. 1012–1023

[Grégoire et al. 2014c] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Enforcing Solutions in Constraint Networks. In : *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic*, 2014, p. 1017–1018

[Grégoire et al. 2014d] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : An Experimentally Efficient Method for (MSS, CoMSS) Partitioning. In : *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 2666–2673

[Grégoire et al. 2014e] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : A General Artificial Intelligence Approach for Skeptical Reasoning. In : *Artificial General Intelligence - 7th International Conference, AGI 2014, Quebec City, QC, Canada, August 1-4, 2014. Proceedings*, 2014, p. 33–42

[Grégoire et al. 2014f] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : Multiple Contraction through Partial-Max-SAT. In : *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, 2014, p. 321–327

[Grégoire et al. 2015] Grégoire, Éric ; Lagniez, Jean-Marie ; Mazure, Bertrand : On getting rid of the preprocessing minimization step in MUC-finding algorithms. In : *Constraints* 20 (2015), Nr. 4, p. 414–432

[Grégoire et al. 2016c] Grégoire, Éric ; Lagniez, Jean-Marie ; Zhang, Du : Logical consensuses for case-based reasoning and for mathematical engineering of AI. In : *15th IEEE International Conference on Cognitive Informatics & Cognitive Computing ,ICCI*CC 2016, Palo Alto, CA, USA, August 22-23, 2016*, 2016, p. 29–33

[Grégoire et al. 2016d] Grégoire, Éric ; Lagniez, Jean-Marie ; Zhang, Du : On Computing Non-hypocritical Consensuses in Standard Logic. In : *27th International Workshop on Database and Expert Systems Applications, DEXA 2016 Workshops, Porto, Portugal, September 5-8, 2016*, 2016, p. 97–101

[Grégoire et al. 2018b] Grégoire, Éric ; Lagniez, Jean-Marie ; Zhang, Du : Consensus-finding that preserves mutually conflicting hypothetical information from a same agent. In : *AI Commun.* 31 (2018), Nr. 3, p. 303–317

[Grégoire et al. 2005] Grégoire, Éric ; Mazure, Bertrand ; Ostrowski, Richard ; Saïs, Lakhdar : Automatic extraction of functional dependencies. In : *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)* Volume 3542. Vancouver (BC) Canada : Sringer, Mai 10-13 2004. Revised selected papers published in 2005, p. 122–132

[Grégoire et al. 2007] Grégoire, Éric ; Mazure, Bertrand ; Piette, Cédric : Boosting a Complete Technique to Find MSS and MUS thanks to a Local Search Oracle. In : *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*. (proc-ijcai07 2007), p. 2300–2305

[Guo et al. 2010] Guo, Long ; Hamadi, Youssef ; Jabbour, Said ; Sais, Lakhdar : Diversification and Intensification in Parallel SAT Solving. In : Cohen, David (Editor) : *Principles and Practice of Constraint Programming – CP 2010* Volume 6308. Springer Berlin / Heidelberg, 2010, p. 252–265

[Guo et Lagniez 2011a] Guo, Long ; Lagniez, Jean-Marie : Ajustement dynamique de l'heuristique de polarité dans le cadre d'un solveur SAT parallèle. In : *Septièmes Journées Francophones de Programmation par Contraintes (JFPC11)*, Juni 2011 (JFPC11), p. 155–161

[Guo et Lagniez 2011b] Guo, Long ; Lagniez, Jean-Marie : Dynamic Polarity Adjustment in a Parallel SAT Solver. In : *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, 2011, p. 67–73

[Haim et Walsh 2009] Haim, Shai ; Walsh, Toby : Restart Strategy Selection Using Machine Learning Techniques. In : *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, 2009, p. 312–325

[Halpern et Rêgo 2007] Halpern, Joseph Y. ; Rêgo, Leandro C. : Characterizing the NP-PSPACE Gap in the Satisfiability Problem for Modal Logic. In : *J. Log. Comput.* 17 (2007), Nr. 4, p. 795–806. – URL https://doi.org/10.1093/logcom/exm029

[Hamadi et al. 2011] Hamadi, Youssef ; Jabbour, Saïd ; Piette, Cédric ; Sais, Lakhdar : Deterministic Parallel DPLL. In : *JSAT* 7 (2011), Nr. 4, p. 127–132. – URL https://satassociation.org/jsat/index.php/jsat/article/view/88

[Hamadi et al. 2009a] Hamadi, Youssef ; Jabbour, Saïd ; Saïs, Lakhdar : Learning for Dynamic Subsumption. In : *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence*. Washington, DC, USA : IEEE Computer Society, 2009 (ICTAI '09), p. 328–335

[Hamadi et al. 2009b] Hamadi, Youssef ; Jabbour, Said ; Sais, Lakhdar : ManySAT : a Parallel SAT Solver. In : *Journal on Satisfiability, Boolean Modeling and Computation* 6 (2009), p. 245–262

[Hamadi et al. 2010] Hamadi, Youssef ; Jabbour, Saïd ; Saïs, Lakhdar : *LySAT : Solver description*. SATRACE, solver description. 2010

[Hamadi et Sais 2018] Hamadi, Youssef (Editor) ; Sais, Lakhdar (Editor) : *Handbook of Parallel Constraint Reasoning*. Springer, 2018. – URL https://doi.org/10.1007/978-3-319-63516-3. – ISBN 978-3-319-63515-6

[Hamscher et al. 1992] Hamscher, Walter (Editor) ; Console, Luca (Editor) ; Kleer, Johan de (Editor) : *Readings in Model-based Diagnosis*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1992. – ISBN 1-55860-249-6

[Han et Somenzi 2007] Han, HyoJung ; Somenzi, Fabio : Alembic : An Efficient Algorithm for CNF Preprocessing. In : *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, 2007, p. 582–587

[Han et Somenzi 2009] Han, Hyojung ; Somenzi, Fabio : On-the-Fly Clause Improvement. In : *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Berlin, Heidelberg : Springer-Verlag, 2009 (SAT '09), p. 209–222

[Heinz et Sachenbacher 2009] Heinz, S. ; Sachenbacher, M. : Using Model Counting to Find Optimal Distinguishing Tests. In : *Proc. of CPAIOR'09*, 2009, p. 117–131

[Heule et al. 2010] Heule, Marijn ; Järvisalo, Matti ; Biere, Armin : Clause elimination procedures for CNF formulas. In : *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*. Berlin, Heidelberg : Springer-Verlag, 2010 (LPAR'10), p. 357–371

[Heule et al. 2011a] Heule, Marijn ; Järvisalo, Matti ; Biere, Armin : Efficient CNF Simplification Based on Binary Implication Graphs. In : *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, 2011, p. 201–215

[Heule et al. 2015] Heule, Marijn ; Järvisalo, Matti ; Lonsing, Florian ; Seidl, Martina ; Biere, Armin : Clause Elimination for SAT and QSAT. In : *J. Artif. Intell. Res.* 53 (2015), p. 127–168

[Heule et al. 2011b] Heule, Marijn ; Kullmann, Oliver ; Wieringa, Siert ; Biere, Armin : Cube and Conquer : Guiding CDCL SAT Solvers by Lookaheads. In : *Hardware and Software : Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, 2011, p. 50–65

[Heule et Kullmann 2006] Heule, Marijn J. H. ; Kullmann, Oliver : Decomposing clause-sets : Integrating DLL algorithms, tree decompositions and hypergraph cuts for variable- and clause-based graph representations of CNF's / Swansea University Prifysgol Abertawe. 2006. – Research Report

[Heule et Kullmann 2017] Heule, Marijn J. H. ; Kullmann, Oliver : The science of brute force. In : *Commun. ACM* 60 (2017), Nr. 8, p. 70–79

[der Hoek et Pauly 2007] Hoek, Wiebevan der ; Pauly, Marc : 20 Modal logic for games and information. In : *Handbook of Modal Logic*, 2007, p. 1077–1148

[Horrocks et al. 2006] Horrocks, Ian ; Hustadt, Ullrich ; Sattler, Ulrike ; Schmidt, Renate : Computational Modal Logic. In : *Handbook of Modal Logic*, 2006, p. 181–245

[Huang 2007] Huang, Jinbo : The Effect of Restarts on the Efficiency of Clause Learning. In : *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*. (proc-ijcai07 2007), p. 2318–2323

[Huang 2010] Huang, Jinbo : Extended clause learning. In : *Artif. Intell.* 174 (2010), Nr. 15, p. 1277–1284

[Huang et Darwiche 2004] Huang, Jinbo ; Darwiche, Adnan : Toward Good Elimination Orders for Symbolic SAT Solving. In : *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, 2004, p. 566–573

[Huang et al. 2019] Huang, Pei ; Liu, Minghao ; Wang, Ping ; Zhang, Wenhui ; Ma, Feifei ; Zhang, Jian : Solving the Satisfiability Problem of Modal Logic S5 Guided by Graph Coloring. In : *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, p. 1093–1100

[Hunter et Konieczny 2008] Hunter, Anthony ; Konieczny, Sébastien : Measuring inconsistency through minimal inconsistent sets. In : *Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, 2008, p. 358–366

[Hustadt et al. 1999] Hustadt, Ullrich ; Schmidt, Renate A. ; Weidenbach, Christoph : MSPASS : Subsumption Testing with SPASS. In : *Proceedings of the 1999 International Workshop on Description Logics (DL'99), Linköping, Sweden, July 30 - August 1, 1999*, 1999, p. 1–2

[Hutter et al. 2017] Hutter, Frank ; Lindauer, Marius ; Balint, Adrian ; Bayless, Sam ; Hoos, Holger H. ; Leyton-Brown, Kevin : The Configurable SAT Solver Challenge (CSSC). In : *Artif. Intell.* 243 (2017), p. 1–25

[Hyvärinen et Manthey 2012] Hyvärinen, Antti Eero J. ; Manthey, Norbert : Designing Scalable Parallel SAT Solvers. In : *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, 2012, p. 214–227

[Ignatiev et al. 2019] Ignatiev, Alexey ; Narodytska, Nina ; Marques-Silva, João : Abduction-Based Explanations for Machine Learning Models. In : *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 1511–1519

[Jabbour et al. 2018] Jabbour, Saïd ; Lonlac, Jerry ; Saïs, Lakhdar ; Salhi, Yakoub : Revisiting the Learned Clauses Database Reduction Strategies. In : *International Journal on Artificial Intelligence Tools* 27 (2018), Nr. 8, p. 1850033. – URL https://doi.org/10.1142/S0218213018500331

[Jabbour et al. 2014] Jabbour, Saïd ; Marques-Silva, João ; Sais, Lakhdar ; Salhi, Yakoub : Enumerating Prime Implicants of Propositional Formulae in Conjunctive Normal Form. In : *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, 2014, p. 152–165

[Janota et al. 2016] Janota, Mikolás ; Klieber, William ; Marques-Silva, João ; Clarke, Edmund M. : Solving QBF with counterexample guided refinement. In : *Artif. Intell.* 234 (2016), p. 1–25

[Janota et Marques-Silva 2016] Janota, Mikolás ; Marques-Silva, João : On the query complexity of selecting minimal sets for monotone predicates. In : *Artif. Intell.* 233 (2016), p. 73–83

[Järvisalo et al. 2012a] Järvisalo, Matti ; Biere, Armin ; Heule, Marijn : Simulating Circuit-Level Simplifications on CNF. In : *J. Autom. Reasoning* 49 (2012), Nr. 4, p. 583–619

[Järvisalo et al. 2012b] Järvisalo, Matti ; Heule, Marijn ; Biere, Armin : Inprocessing Rules. In : *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, 2012, p. 355–370

[Jégou et Terrioux 2017] Jégou, Philippe ; Terrioux, Cyril : Combining restarts, nogoods and bag-connected decompositions for solving CSPs. In : *Constraints* 22 (2017), Nr. 2, p. 191–229

[Jeroslow et Wang 1990] Jeroslow, Robert G. ; Wang, Jinchang : Solving Propositional Satisfiability Problems. In : *Annals of Mathematics and Artificial Intelligence* 1 (1990), p. 167–187

[Järvisalo et al. 2010] Järvisalo, Matti ; Biere, Armin ; Heule, Marijn : Blocked Clause Elimination. In : Esparza, Javier (Editor) ; Majumdar, Rupak (Editor) : *Tools and Algorithms for the Construction and Analysis of Systems* Volume 6015. Springer Berlin / Heidelberg, 2010, p. 129–144

[Kaminski et Tebbi 2013] Kaminski, Mark ; Tebbi, Tobias : InKreSAT : Modal Reasoning via Incremental Reduction to SAT. In : *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, 2013, p. 436–442

[Kautz et Selman 1996] Kautz, Henry ; Selman, Bart : Pushing the envelope : planning, propositional logic, and stochastic search. In : *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*, AAAI Press, 1996 (AAAI'96), p. 1194–1201

[Kilby et al. 2005] Kilby, Philip ; Slaney, John K. ; Thiébaux, Sylvie ; Walsh, Toby : Backbones and Backdoors in Satisfiability. In : *Proceedings, The 20th National Conference on Artificial Intelligence (AAAI'05)*, AAAI Press / The MIT Press, 2005, p. 1368–1373

[Klebanov et al. 2013] Klebanov, V. ; Manthey, N. ; Muise, Ch. J. : SAT-Based Analysis and Quantification of Information Flow in Programs. In : *Proc. of QUEST'13*, 2013, p. 177–192

[Konieczny et al. 2017a] Konieczny, Sébastien ; Lagniez, Jean-Marie ; Marquis, Pierre : Boosting Distance-Based Revision Using SAT Encodings. In : *Logic, Rationality, and Interaction - 6th International Workshop, LORI 2017, Sapporo, Japan, September 11-14, 2017, Proceedings*, 2017, p. 480–496

[Konieczny et al. 2017b] Konieczny, Sébastien ; Lagniez, Jean-Marie ; Marquis, Pierre : SAT Encodings for Distance-Based Belief Merging Operators. In : *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, p. 1163–1169

[Koriche et al. 2016] Koriche, Frédéric ; Berre, Daniel L. ; Lonca, Emmanuel ; Marquis, Pierre : Fixed-Parameter Tractable Optimization Under DNNF Constraints. In : *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, 2016, p. 1194–1202

[Koriche et al. 2013] Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre ; Thomas, Samuel : Knowledge Compilation for Model Counting : Affine Decision Trees. In : *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 947–953

[Koriche et al. 2015] Koriche, Frédéric ; Lagniez, Jean-Marie ; Marquis, Pierre ; Thomas, Samuel : Compiling Constraint Networks into Multivalued Decomposable Decision Graphs. In : *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, p. 332–338

[Kripke 1959] Kripke, Saul : A Completeness Theorem in Modal Logic. In : *J. Symb. Log.* 24 (1959), Nr. 1, p. 1–14. – URL https://doi.org/10.2307/2964568

[Kullmann 1999] Kullmann, Olivier : On a generalization of extended resolution. In : *Discrete Applied Mathematics* 96-97 (1999), p. 149 – 176

[Ladner 1977] Ladner, Richard E. : The Computational Complexity of Provability in Systems of Modal Propositional Logic. In : *SIAM J. Comput.* 6 (1977), Nr. 3, p. 467–480. – URL https://doi.org/10.1137/0206033

[Lagniez et al. 2016a] Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin : On Checking Kripke Models for Modal Logic K. In : *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016.*, 2016, p. 69–81

[Lagniez et al. 2017a] Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin : A Recursive Shortcut for CEGAR : Application To The Modal Logic K Satisfiability Problem. In : *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, p. 674–680

[Lagniez et al. 2018a] Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin : A SAT-Based Approach For PSPACE Modal Logics. In : *Principles of Knowledge Representation and Reasoning : Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018.*, 2018, p. 651–652

[Lagniez et al. 2018b] Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin : An Assumption-Based Approach for Solving the Minimal S5-Satisfiability Problem. In : *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, 2018, p. 1–18

[Lagniez et al. 2018c] Lagniez, Jean-Marie ; Berre, Daniel L. ; Lima, Tiago de ; Montmirail, Valentin : A SAT-Based Approach For PSPACE Modal Logics. In : *Principles of Knowledge Representation and Reasoning : Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018.*, 2018, p. 651–652

[Lagniez et Biere 2013] Lagniez, Jean-Marie ; Biere, Armin : Factoring Out Assumptions to Speed Up MUS Extraction. In : *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 276–292

[Lagniez et al. 2015] Lagniez, Jean-Marie ; Lonca, Emmanuel ; Mailly, Jean-Guy : CoQuiAAS : A Constraint-Based Quick Abstract Argumentation Solver. In : *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*, 2015, p. 928–935

[Lagniez et al. 2016b] Lagniez, Jean-Marie ; Lonca, Emmanuel ; Marquis, Pierre : Improving Model Counting by Leveraging Definability. In : *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, p. 751–757

[Lagniez et Marquis 2014] Lagniez, Jean-Marie ; Marquis, Pierre : Preprocessing for Propositional Model Counting. In : *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, p. 2688–2694

[Lagniez et Marquis 2017a] Lagniez, Jean-Marie ; Marquis, Pierre : An Improved Decision-DNNF Compiler. In : *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, p. 667–673

[Lagniez et Marquis 2017b] Lagniez, Jean-Marie ; Marquis, Pierre : On Preprocessing Techniques and Their Impact on Propositional Model Counting. In : *J. Autom. Reasoning* 58 (2017), Nr. 4, p. 413–481

[Lagniez et Marquis 2019a] Lagniez, Jean-Marie ; Marquis, Pierre : A Recursive Algorithm for Projected Model Counting. In : *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 1536–1543

[Lagniez et Marquis 2019b] Lagniez, Jean-Marie ; Marquis, Pierre : A Recursive Algorithm for Projected Model Counting. In : *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 1536–1543

[Lagniez et al. 2017b] Lagniez, Jean-Marie ; Marquis, Pierre ; Paparrizou, Anastasia : Defining and Evaluating Heuristics for the Compilation of Constraint Networks. In : *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 172–188

[Lagniez et al. 2018d] Lagniez, Jean-Marie ; Marquis, Pierre ; Szczepanski, Nicolas : DMC : A Distributed Model Counter. In : *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, p. 1331–1338

[Lang et al. 2002] Lang, Jérôme ; Liberatore, Paolo ; Marquis, Pierre : Conditional independence in propositional logic. In : *Artif. Intell.* 141 (2002), Nr. 1/2, p. 79–121

[Lang et Marquis 2008] Lang, Jérôme ; Marquis, Pierre : On propositional definability. In : *Artif. Intell.* 172 (2008), Nr. 8-9, p. 991–1017

[Lazaar et al. 2012] Lazaar, Nadjib ; Hamadi, Youssef ; Jabbour, Said ; Sebag, Michèle : BESS : Bandit Ensemble for parallel SAT Solving. URL https://hal.inria.fr/hal-00733282, September 2012 (RR-8070). – Research Report. – 18 p

[Le Berre 2001] Le Berre, Daniel : Exploiting the real power of Unit Propagation Lookahead. In : *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)* Volume 9. Boston University, Massachusetts, USA, June 14th-15th 2001, p. 59–80. – to appear

[Le Berre et Rapicault 2009] Le Berre, Daniel ; Rapicault, Pascal : Dependency management for the eclipse ecosystem : eclipse p2, metadata and resolution. In : *Proceedings of the 1st international workshop on Open component ecosystems*. New York, NY, USA : ACM, 2009, p. 21–30

[Lecoutre et al. 2007] Lecoutre, Christophe ; Sais, Lakhdar ; Tabary, Sébastien ; Vidal, Vincent : Nogood Recording from Restarts. In : *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007, p. 131–136

[Lewis 1978] Lewis, Harry R. : Renaming a Set of Clauses as Horn Set. In : *Journal of the Association for Computing Machinery* 25 (1978), p. 134–135

[Li 2003] Li, Chu M. : Equivalent literal propagation in the DLL procedure. In : *Discrete Applied Mathematics* 130 (2003), Nr. 2, p. 251–276. – URL https://doi.org/10.1016/S0166-218X(02)00407-9

[Li et Anbulagan 1997] Li, Chu M. ; Anbulagan, Anbulagan : Heuristics based on unit propagation for satisfiability problems. In : *Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997, p. 366–371

[Liang et al. 2017] Liang, Jia H. ; K., Hari Govind V. ; Poupart, Pascal ; Czarnecki, Krzysztof ; Ganesh, Vijay : An Empirical Study of Branching Heuristics Through the Lens of Global Learning Rate. In : *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 119–135

[Liang et al. 2018] Liang, Jia H. ; Oh, Chanseok ; Mathew, Minu ; Thomas, Ciza ; Li, Chunxiao ; Ganesh, Vijay : Machine Learning-Based Restart Policy for CDCL SAT Solvers. In : *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, 2018, p. 94–110

[Liberatore 2000] Liberatore, Paolo : On the complexity of choosing the branching literal in DPLL. In : *Artificial Intelligence* 116 (2000), January, p. 315–326. – ISSN 0004-3702

[Liffiton et al. 2016] Liffiton, Mark H. ; Previti, Alessandro ; Malik, Ammar ; Marques-Silva, Joao : Fast, flexible MUS enumeration. In : *Constraints* 21 (2016), Nr. 2, p. 223–250

[Liffiton et Sakallah 2008] Liffiton, Mark H. ; Sakallah, Karem A. : Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. In : *Journal of Automated Reasoning* 40 (2008), Januar, Nr. 1, p. 1–33

[Littman et al. 1998] Littman, Michael L. ; Goldsmith, Judy ; Mundhenk, Martin : The Computational Complexity of Probabilistic Planning. In : *J. Artif. Intell. Res.* 9 (1998), p. 1–36

[Littman et al. 2001] Littman, Michael L. ; Majercik, Stephen M. ; Pitassi, Toniann : Stochastic Boolean Satisfiability. In : *J. Autom. Reasoning* 27 (2001), Nr. 3, p. 251–296

[Luby et al. 1993] Luby, Michael ; Sinclair, Alistair ; Zuckerman, David : Optimal speedup of Las Vegas algorithms. In : *Information Processing Letters* 47 (1993), Nr. 4, p. 173–180. – ISSN 0020-0190

[Luo et al. 2017] Luo, Mao ; Li, Chu-Min ; Xiao, Fan ; Manyà, Felip ; Lü, Zhipeng : An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers. In : *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, p. 703–711

[Lynce et Marques-Silva 2003] Lynce, Inês ; Marques-Silva, João : Probing-Based Preprocessing Techniques for Propositional Satisfiability. In : *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*. Washington, DC, USA : IEEE Computer Society, 2003 (ICTAI '03), p. 105–. – ISBN 0-7695-2038-3

[Mackworth 1977] Mackworth, Alan K. : Consistency in networks of relations. In : *Artificial Intelligence* 8 (1977), Nr. 1, p. 99 – 118

[Manthey 2012] Manthey, N. : Solver Description of RISS 2.0 and PRISS 2.0 / TU Dresden, Knowledge Representation and Reasoning. 2012. – Research Report

[Marques-Silva et al. 2013] Marques-Silva, João ; Heras, Federico ; Janota, Mikolás ; Previti, Alessandro ; Belov, Anton : On Computing Minimal Correction Subsets. In : *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, p. 615–622

[Marques-Silva et al. 2015] Marques-Silva, João ; Janota, Mikolás ; Ignatiev, Alexey ; Morgado, António : Efficient Model Based Diagnosis with Maximum Satisfiability. In : *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, p. 1966–1972

[Marques-Silva et Sakallah 1996a] Marques-Silva, João P. ; Sakallah, Karem A. : GRASP—a new search algorithm for satisfiability. In : *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA : IEEE Computer Society, 1996, p. 220–227. – ISBN 0-8186-7597-7

[Marques-Silva et Sakallah 1996b] Marques-Silva, João P. ; Sakallah, Karem A. : GRASP—a new search algorithm for satisfiability. In : *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design.* (Marques-Silva et Sakallah 1996a), p. 220–227. – ISBN 0-8186-7597-7

[Marquis 2015] Marquis, Pierre : Compile ! In : *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, 2015, p. 4112–4118

[Martins et al. 2010] Martins, Ruben ; Manquinho, Vasco M. ; Lynce, Inês : Improving Search Space Splitting for Parallel SAT Solving. In : *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, 2010, p. 336–343

[Massacci 2000] Massacci, Fabio : Single Step Tableaux for Modal Logics. In : *J. Autom. Reasoning* 24 (2000), Nr. 3, p. 319–364

[Massacci et Donini 2000] Massacci, Fabio ; Donini, Francesco M. : Design and Results of TANCS-2000 Non-classical (Modal) Systems Comparison. In : *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2000, St Andrews, Scotland, UK, July 3-7, 2000, Proceedings*, 2000, p. 52–56

[McAllester 1980] McAllester, David A. : An Outlook on Truth Maintenance / MIT. 1980. – Research Report

[McCarthy 1980] McCarthy, John : Circumscription - A Form of Non-Monotonic Reasoning. In : *Artif. Intell.* 13 (1980), Nr. 1-2, p. 27–39

[Meel et al. 2016] Meel, Kuldeep S. ; Vardi, Moshe Y. ; Chakraborty, Supratik ; Fremont, Daniel J. ; Seshia, Sanjit A. ; Fried, Dror ; Ivrii, Alexander ; Malik, Sharad : Constrained Sampling and Counting : Universal Hashing Meets SAT Solving. In : *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016.*, 2016

[Mencía et al. 2015] Mencía, Carlos ; Previti, Alessandro ; Marques-Silva, João : Literal-Based MCS Extraction. In : *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, p. 1973–1979

[Minoux 1988] Minoux, Michel : LTUR : A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. In : *Information Processing Letters* 29 (1988), 15 September, Nr. 1, p. 1–12

[Möhle et Biere 2019] Möhle, Sibylle ; Biere, Armin : Backing Backtracking. In : *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, 2019, p. 250–266

[Monasson et al. 1999a] Monasson, Rémi ; Zecchina, Riccardo ; Kirkpatrick, Scott ; Selman, Bart ; Troyansky, Lidror : Determining computational complexity from characteristic 'phase transitions'. In : *Nature* 400 (1999), p. 133

[Monasson et al. 1999b] Monasson, Remi ; Zecchina, Riccardo ; Kirkpatrick, Scott ; Selman, Bart ; Troyansky, Lidror : Determining computational complexity from characteristic 'phase transitions'. In : *Nature* 400 (1999), Nr. 6740, p. 133–137

[Moore 1998] Moore, Gordon E. : Cramming More Components Onto Integrated Circuits. In : *Proceedings of the IEEE* 86 (1998), Nr. 1, p. 82–85

[Moskewicz et al. 2001a] Moskewicz, Matthew W. ; Madigan, Conor F. ; Zhao, Ying ; Zhang, Lintao ; Malik, Sharad : Chaff : engineering an efficient SAT solver. In : *DAC '01 : Proceedings of the 38th annual Design Automation Conference.* (Moskewicz et al. 2001b), p. 530–535. – ISBN 1-58113-297-2

[Moskewicz et al. 2001b] Moskewicz, Matthew W. ; Madigan, Conor F. ; Zhao, Ying ; Zhang, Lintao ; Malik, Sharad : Chaff : engineering an efficient SAT solver. In : *DAC '01 : Proceedings of the 38th annual Design Automation Conference.* New York, NY, USA : ACM, Juni 2001, p. 530–535. – ISBN 1-58113-297-2

[Muise et al. 2012] Muise, Christian J. ; McIlraith, Sheila A. ; Beck, J. C. ; Hsu, Eric I. : Dsharp : Fast d-DNNF Compilation with sharpSAT. In : *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, 2012, p. 356–361

[Nadel 2010] Nadel, Alexander : Boosting minimal unsatisfiable core extraction. In : *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, 2010, p. 221–229

[Nadel et Ryvchin 2010] Nadel, Alexander ; Ryvchin, Vadim : Assignment Stack Shrinking. In : Strichman, Ofer (Editor) ; Szeider, Stefan (Editor) : *Theory and Applications of Satisfiability Testing - SAT 2010* Volume 6175, Springer Berlin / Heidelberg, 2010, p. 375–381

[Nadel et Ryvchin 2018] NADEL, Alexander ; RYVCHIN, Vadim : Chronological Backtracking. In : *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, 2018, p. 111–121

[Nadel et al. 2014] NADEL, Alexander ; RYVCHIN, Vadim ; STRICHMAN, Ofer : Accelerated Deletion-based Extraction of Minimal Unsatisfiable Cores. In : *JSAT* 9 (2014), p. 27–51

[Nalon et al. 2016a] NALON, Cláudia ; HUSTADT, Ullrich ; DIXON, Clare : : A Resolution-Based Prover for Multimodal K. In : *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, 2016, p. 406–415

[Nalon et al. 2016b] NALON, Cláudia ; HUSTADT, Ullrich ; DIXON, Clare : K_SP : A Resolution-Based Prover for Multimodal K. In : *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, 2016, p. 406–415

[Narodytska et al. 2018] NARODYTSKA, Nina ; BJØRNER, Nikolaj ; MARINESCU, Maria-Cristina ; SAGIV, Mooly : Core-Guided Minimal Correction Set and Core Enumeration. In : *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, International Joint Conferences on Artificial Intelligence Organization, 7 2018, p. 1353–1361

[Narodytska et al. 2019] NARODYTSKA, Nina ; SHROTRI, Aditya A. ; MEEL, Kuldeep S. ; IGNATIEV, Alexey ; MARQUES-SILVA, João : Assessing Heuristic Machine Learning Explanations with Model Counting. In : *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, 2019, p. 267–278

[Nejati et al. 2017a] NEJATI, Saeed ; LIANG, Jia H. ; GEBOTYS, Catherine H. ; CZARNECKI, Krzysztof ; GANESH, Vijay : Adaptive Restart and CEGAR-Based Solver for Inverting Cryptographic Hash Functions. In : *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers*, 2017, p. 120–131

[Nejati et al. 2017b] NEJATI, Saeed ; NEWSHAM, Zack ; SCOTT, Joseph ; LIANG, Jia H. ; GEBOTYS, Catherine H. ; POUPART, Pascal ; GANESH, Vijay : A Propagation Rate Based Splitting Heuristic for Divide-and-Conquer Solvers. In : *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 251–260

[Nguyen 1999] NGUYEN, Linh A. : A New Space Bound for the Modal Logics K4, KD4 and S4. In : *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, 1999, p. 321–331

[Nöhrer et al. 2012] NÖHRER, Alexander ; BIERE, Armin ; EGYED, Alexander : Managing SAT inconsistencies with HUMUS. In : *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, 2012, p. 83–91

[Novikov 2003] NOVIKOV, Yakov : Local Search for Boolean Relations on the Basis of Unit Propagation. In : *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1.* Washington, DC, USA : IEEE Computer Society, 2003 (DATE '03), p. 10810–

[Ohmura et Ueda 2009] Ohmura, Kei ; Ueda, Kazunori : c-sat : A Parallel SAT Solver for Clusters. In : *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, 2009, p. 524–537

[Ostrowski 2004] Ostrowski, Richard : *Reconnaissance et exploitation de propriétés structurelles pour la résolution du probléme SAT*. Faculté des Sciences Jean Perrin, Lens, France, Université d'Artois, Thèse Doctorat, Dezember 2004

[Ostrowski et al. 2002] Ostrowski, Richard ; Grégoire, Éric ; Mazure, Bertrand ; Saïs, Lakhdar : Recovering and exploiting structural knowledge from CNF formulas. In : *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'02)* Volume 2470, Springer, September 2002, p. 185–199

[Oztok et Darwiche 2014] Oztok, Umut ; Darwiche, Adnan : On Compiling CNF into Decision-DNNF. In : *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, 2014, p. 42–57

[Padoa 1903] Padoa, A. : Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une théorie déductive quelconque. In : *Bibliothèque du Congrès International de Philosophie*. Paris, 1903, p. 309–365

[Palacios et al. 2005] Palacios, H. ; Bonet, B. ; Darwiche, A. ; Geffner, H. : Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In : *Proc. of ICAPS'05*, 2005, p. 141–150

[Palù et al. 2015] Palù, Alessandro D. ; Dovier, Agostino ; Formisano, Andrea ; Pontelli, Enrico : CUD@SAT : SAT solving on GPUs. In : *J. Exp. Theor. Artif. Intell.* 27 (2015), Nr. 3, p. 293–316. – URL https://doi.org/10.1080/0952813X.2014.954274

[Pan et Vardi 2004] Pan, Guoqiang ; Vardi, Moshe Y. : Symbolic Decision Procedures for QBF. In : *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, 2004, p. 453–467

[Papadimitriou 1994] Papadimitriou, Christos H. : *Computational Complexity*. Addison Wesley Pub. Co., 1994

[Papadimitriou et Wolfe 1988] Papadimitriou, Christos H. ; Wolfe, David : The Complexity of Facets Resolved. In : *J. Comput. Syst. Sci.* 37 (1988), Nr. 1, p. 2–13. – URL https://doi.org/10.1016/0022-0000(88)90042-6

[Papadimitriou et Yannakakis 1984] Papadimitriou, Christos H. ; Yannakakis, Mihalis : The Complexity of Facets (and Some Facets of Complexity). In : *J. Comput. Syst. Sci.* 28 (1984), Nr. 2, p. 244–259. – URL https://doi.org/10.1016/0022-0000(84)90068-0

[Park 2002] Park, James D. : MAP Complexity Results and Approximation Methods. In : *UAI '02, Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence, University of Alberta, Edmonton, Alberta, Canada, August 1-4, 2002*, 2002, p. 388–396

[Park et Darwiche 2004] Park, James D. ; Darwiche, Adnan : Complexity Results and Approximation Strategies for MAP Explanations. In : *J. Artif. Intell. Res.* 21 (2004), p. 101–133

[Patel-Schneider et Sebastiani 2003] Patel-Schneider, Peter F. ; Sebastiani, Roberto : A New General Method to Generate Random Modal Formulae for Testing Decision Procedures. In : *J. Artif. Intell. Res.* 18 (2003), p. 351–389

[Paulusma et al. 2016] Paulusma, Daniël ; Slivovsky, Friedrich ; Szeider, Stefan : Model Counting for CNF Formulas of Bounded Modular Treewidth. In : *Algorithmica* 76 (2016), Nr. 1, p. 168–194

[Pereira et al. 1993] Pereira, Luís Moniz ; Damásio, Carlos V. ; Alferes, José Júlio : Debugging by Diagnosing Assumptions. In : *AADEBUG* Volume 749, Springer, 1993, p. 58–74

[Perifel 2014] Perifel, S. : *Complexité algorithmique*. Ellipses, 2014 (Références sciences). – URL https://books.google.fr/books?id=87PUoAEACAAJ. – ISBN 9782729886929

[Piette et al. 2008] Piette, Cédric ; Hamadi, Youssef ; Lakhdar, Saïs : Vivifying propositional clausal formulae. In : *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'03)*. Patras (Greece), Juli 2008, p. 525–529

[Pipatsrisawat et Darwiche 2007] Pipatsrisawat, Knot ; Darwiche, Adnan : A Lightweight Component Caching Scheme for Satisfiability Solvers. In : *SAT*, 2007, p. 294–299

[Pipatsrisawat et Darwiche 2008] Pipatsrisawat, Knot ; Darwiche, Adnan : New Compilation Languages Based on Structured Decomposability. In : *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008, p. 517–522

[Pipatsrisawat et Darwiche 2009] Pipatsrisawat, Knot ; Darwiche, Adnan : Width-Based Restart Policies for Clause-Learning Satisfiability Solvers. In : *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Berlin, Heidelberg : Springer-Verlag, 2009 (SAT '09), p. 341–355. – ISBN 978-3-642-02776-5

[Pipatsrisawat et Darwiche 2011] Pipatsrisawat, Knot ; Darwiche, Adnan : On the power of clause-learning SAT solvers as resolution engines. In : *Artif. Intell.* 175 (2011), Nr. 2, p. 512–525

[Plaisted et Greenbaum 1986] Plaisted, David A. ; Greenbaum, Steven : A Structure-Preserving Clause Form Translation. In : *J. Symb. Comput.* 2 (1986), Nr. 3, p. 293–304

[Prates et al. 2019] Prates, Marcelo O. R. ; Avelar, Pedro H. C. ; Lemos, Henrique ; Lamb, Luís C. ; Vardi, Moshe Y. : Learning to Solve NP-Complete Problems : A Graph Neural Network for Decision TSP. In : *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 4731–4738

[Prcovic 2016] Prcovic, Nicolas : La XOR-résolution. In : *Proceedings of the Douzièmes Journees Francophones de Programmation par Contraintes, JFPC 2016, Montpellier, France, June 15-17, 2016*, 2016, p. 37–44

[Pretolani 1993] Pretolani, Daniele : *Satisfiability and Hypergraphs*. Genova, Italia, dipartimento di Informatica : Universita di Pisa, Ph.D. thesis, März 1993. – TD-12/93

[Previti et Marques-Silva 2013] Previti, Alessandro ; Marques-Silva, João : Partial MUS Enumeration. In : *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.*, 2013

[Previti et al. 2017] Previti, Alessandro ; Mencía, Carlos ; Järvisalo, Matti ; Marques-Silva, João : Improving MCS Enumeration via Caching. In : *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 2017, p. 184–194

[Previti et al. 2018] Previti, Alessandro ; Mencía, Carlos ; Järvisalo, Matti ; Marques-Silva, João : Premise Set Caching for Enumerating Minimal Correction Subsets. In : *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 2018, p. 6633–6640

[Prosser 1993] Prosser, Patrick : Hybrid algorithms for the constraint satisfaction problems. In : *Computational Intelligence* 9 (1993), Nr. 3, p. 268–299

[Rajasekaran et Reif 2007] Rajasekaran, Sanguthevar (Editor) ; Reif, John H. (Editor) : *Handbook of Parallel Computing - Models, Algorithms and Applications.* Chapman and Hall/-CRC, 2007

[Rauzy 1995] Rauzy, Antoine : Polynomial restrictions of SAT : What can be done with an efficient implementation of the Davis and Putnam's procedure. In : Montanari, U. (Editor) ; Rossi, F. (Editor) : *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'95)* Volume 976. Cassis, France : Springer, September 1995, p. 515–532

[Reiter 1987] Reiter, Raymond : A Theory of Diagnosis from First Principles. In : *Artif. Intell.* 32 (1987), Nr. 1, p. 57–95

[Rossi et al. 2006] Rossi, Francesca (Editor) ; Beek, Peter van (Editor) ; Walsh, Toby (Editor) : *Foundations of Artificial Intelligence.* Volume 2 : *Handbook of Constraint Programming.* Elsevier, 2006

[Roychoudhury et Chandra 2016] Roychoudhury, Abhik ; Chandra, Satish : Formula-based software debugging. In : *Commun. ACM* 59 (2016), Nr. 7, p. 68–77

[Ryan 2004] Ryan, Lawrence : *Efficient algorithms for clause-learning SAT solvers*, Simon Fraser University, Ph.D. thesis, 2004

[Sæther et al. 2015] Sæther, Sigve H. ; Telle, Jan A. ; Vatshelle, Martin : Solving #SAT and MAXSAT by Dynamic Programming. In : *J. Artif. Intell. Res.* 54 (2015), p. 59–82

[Salhi et al. 2012] Salhi, Yakoub ; Jabbour, Saïd ; Sais, Lakhdar : Graded Modal Logic GS5 and Itemset Support Satisfiability. In : *Information Search, Integration and Personalization - International Workshop, ISIP 2012, Sapporo, Japan, October 11-13, 2012. Revised Selected Papers*, 2012, p. 131–140

[Salhi et Sioutis 2015] Salhi, Yakoub ; Sioutis, Michael : A Resolution Method for Modal Logic S5. In : *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, 2015, p. 252–262

[Samer et Szeider 2009] Samer, Marko ; Szeider, Stefan : Fixed-Parameter Tractability. In : *Handbook of Satisfiability*. 2009, p. 425–454

[Samer et Szeider 2010] Samer, Marko ; Szeider, Stefan : Algorithms for propositional model counting. In : *J. Discrete Algorithms* 8 (2010), Nr. 1, p. 50–64

[Sang et al. 2005] Sang, T. ; Beame, P. ; Kautz, H. A. : Performing Bayesian Inference by Weighted Model Counting. In : *Proc. of AAAI'05*, 2005, p. 475–482

[Sang et al. 2004] Sang, Tian ; Bacchus, Fahiem ; Beame, Paul ; Kautz, Henry A. ; Pitassi, Toniann : Combining Component Caching and Clause Learning for Effective Model Counting. In : *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004

[Schiex et Verfaillie 1993a] Schiex, Thomas ; Verfaillie, Gérard : Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In : *Fifth International Conference on Tools with Artificial Intelligence, ICTAI '93, Boston, Massachusetts, USA, November 8-11, 1993*, 1993, p. 48–55

[Schiex et Verfaillie 1993b] Schiex, Thomas ; Verfaillie, Gérard : Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In : *International Journal of Artificial Intelligence Tools* 3 (1993), p. 48–55

[Schmidt et Tishkovsky 2008] Schmidt, Renate A. ; Tishkovsky, Dmitry : A General Tableau Method for Deciding Description Logics, Modal Logics and Related First-Order Fragments. In : *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, 2008, p. 194–209

[Schrag 1996] Schrag, Robert C. : Compilation for critically constrained knowledge bases. In : *Proceedings of the Thirteenth American National Conference on Artificial Intelligence (AAAI'96)*, Juli 1996, p. 510–515

[Schulz et Blochinger 2010] Schulz, Sven ; Blochinger, Wolfgang : Parallel SAT Solving on Peer-to-Peer Desktop Grids. In : *J. Grid Comput.* 8 (2010), Nr. 3, p. 443–471

[Schwind et al. 2019] Schwind, Nicolas ; Inoue, Katsumi ; Konieczny, Sébastien ; Lagniez, Jean-Marie ; Marquis, Pierre : What Has Been Said ? Identifying the Change Formula in a Belief Revision Scenario. In : *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, p. 1865–1871

[Schwind et al. 2018] Schwind, Nicolas ; Okimoto, Tenda ; Inoue, Katsumi ; Hirayama, Katsutoshi ; Lagniez, Jean-Marie ; Marquis, Pierre : Probabilistic Coalition Structure Generation. In : *Principles of Knowledge Representation and Reasoning : Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018.*, 2018, p. 663–664

[Sebastiani et McAllester 1997] Sebastiani, Roberto ; McAllester, David : New Upper Bounds for Satisfiability in Modal Logics the Case-study of Modal K / IRST, Trento, Italy. Citeseerx, October 1997. – Technical Report 9710-15. – URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.8332.

[Sebastiani et Vescovi 2009a] Sebastiani, Roberto ; Vescovi, Michele : Automated Reasoning in Modal and Description Logics via SAT Encoding : the Case Study of K(m)/ALC-Satisfiability. In : *J. Artif. Intell. Res.* 35 (2009), p. 343–389

[Sebastiani et Vescovi 2009b] Sebastiani, Roberto ; Vescovi, Michele : Automated Reasoning in Modal and Description Logics via SAT Encoding : the Case Study of K(m)/ALC-Satisfiability. In : *J. Artif. Intell. Res.* 35 (2009), p. 343–389

[Seipp et Helmert 2018] Seipp, Jendrik ; Helmert, Malte : Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. In : *J. Artif. Intell. Res.* 62 (2018), p. 535–577

[Selsam et al. 2019] Selsam, Daniel ; Lamm, Matthew ; Bünz, Benedikt ; Liang, Percy ; Moura, Leonardo de ; Dill, David L. : Learning a SAT Solver from Single-Bit Supervision. In : *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019

[Sharma et al. 2019] Sharma, Shubham ; Roy, Subhajit ; Soos, Mate ; Meel, Kuldeep S. : GANAK : A Scalable Probabilistic Exact Model Counter. In : *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, p. 1169–1176

[Shih et al. 2019] Shih, Andy ; Darwiche, Adnan ; Choi, Arthur : Verifying Binarized Neural Networks by Angluin-Style Learning. In : *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, 2019, p. 354–370

[Silver et al. 2016a] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent ; Driessche, George van den ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Panneershelvam, Veda ; Lanctot, Marc ; Dieleman, Sander ; Grewe, Dominik ; Nham, John ; Kalchbrenner, Nal ; Sutskever, Ilya ; Lillicrap, Timothy ; Leach, Madeleine ; Kavukcuoglu, Koray ; Graepel, Thore ; Hassabis, Demis : Mastering the Game of Go with Deep Neural Networks and Tree Search. In : *Nature* 529 (2016), p. 484–489

[Silver et al. 2016b] Silver, David ; Huang, Aja ; Maddison, Chris J. ; Guez, Arthur ; Sifre, Laurent ; Driessche, George van den ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Panneershelvam, Vedavyas ; Lanctot, Marc ; Dieleman, Sander ; Grewe, Dominik ; Nham, John ; Kalchbrenner, Nal ; Sutskever, Ilya ; Lillicrap, Timothy P. ; Leach, Madeleine ; Kavukcuoglu, Koray ; Graepel, Thore ; Hassabis, Demis : Mastering the game of Go with deep neural networks and tree search. In : *Nature* 529 (2016), Nr. 7587, p. 484–489. – URL https://doi.org/10.1038/nature16961

[Sinz et al. 2001] Sinz, Carsten ; Blochinger, Wolfgang ; Küchlin, Wolfgang : PaSAT – Parallel SAT-Checking with Lemma Exchange : Implementation and Applications. In : *Electronic Notes in Discrete Mathematics* 9 (2001), p. 205 – 216. – LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001). – ISSN 1571-0653

[Slivovsky et Szeider 2013] Slivovsky, Friedrich ; Szeider, Stefan : Model Counting for Formulas of Bounded Clique-Width. In : *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings*, 2013, p. 677–687

[Smullyan 1966] Smullyan, Raymond M. : Trees and Nest Structures. In : *J. Symb. Log.* 31 (1966), Nr. 3, p. 303–321

[Soh et al. 2014] Soh, Takehide ; Berre, Daniel L. ; Roussel, Stéphanie ; Banbara, Mutsunori ; Tamura, Naoyuki : Incremental SAT-Based Method with Native Boolean Cardinality Handling for the Hamiltonian Cycle Problem. In : *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, 2014, p. 684–693

[Soos et Meel 2019] Soos, Mate ; Meel, Kuldeep S. : BIRD : Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting. In : *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, 2019, p. 1592–1599

[Soos et al. 2009] Soos, Mate ; Nohl, Karsten ; Castelluccia, Claude : Extending SAT Solvers to Cryptographic Problems. In : *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, 2009, p. 244–257

[Sörensson et Biere 2009] Sörensson, Niklas ; Biere, Armin : Minimizing Learned Clauses. In : *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Berlin, Heidelberg : Springer-Verlag, 2009 (SAT '09), p. 237–243

[Stallman et Sussman 1977a] Stallman, Richard M. ; Sussman, Gerald J. : Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. In : *Artificial Intelligence* 9 (1977), Nr. 2, p. 135 – 196. – ISSN 0004-3702

[Stallman et Sussman 1977b] Stallman, Richard M. ; Sussman, Gerald J. : Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. In : *Artificial Intelligence* 9 (1977), Nr. 2, p. 135–196

[Stockmeyer 1976] Stockmeyer, Larry J. : The polynomial-time hierarchy. In : *Theoretical Computer Science* 3 (1976), Nr. 1, p. 1 – 22

[Stockmeyer 1983] Stockmeyer, Larry J. : The Complexity of Approximate Counting (Preliminary Version). In : *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, 1983, p. 118–126

[Subbarayan et Pradhan 2005] Subbarayan, Sathiamoorthy ; Pradhan, Dhiraj : NiVER : Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances. In : Hoos, Holger (Editor) ; Mitchell, David (Editor) : *Theory and Applications of Satisfiability Testing* Volume 3542. Springer Berlin / Heidelberg, 2005, p. 899–899

[Thurley 2006] Thurley, Marc : sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In : *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, 2006, p. 424–429

[Toda 1989] Toda, Seinosuke : On the Computational Power of PP and +P. In : *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, 1989, p. 514–519

[Tsarkov et Horrocks 2006] Tsarkov, Dmitry ; Horrocks, Ian : FaCT++ Description Logic Reasoner : System Description. In : *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, 2006, p. 292–297

[Tseitin 1968] Tseitin, G.S. : On the complexity of derivations in the propositional calculus. In : Slesenko, H.A.O. (Editor) : *Structures in Constructives Mathematics and Mathematical Logic, Part II*, 1968, p. 115–125

[Urquhart 1981] Urquhart, Alasdair : Decidability and the finite model property. In : *J. Philosophical Logic* 10 (1981), Nr. 3, p. 367–370. – URL https://doi.org/10.1007/BF00293428

[Valiant 1979] Valiant, Leslie G. : The Complexity of Enumeration and Reliability Problems. In : *SIAM J. Comput.* 8 (1979), Nr. 3, p. 410–421. – URL https://doi.org/10.1137/0208032

[Vander-Swalmen et al. 2009] Vander-Swalmen, Pascal ; Dequen, Gilles ; Krajecki, Michaël : A Collaborative Approach for Multi-Threaded SAT Solving. In : *International Journal of Parallel Programming* 37 (2009), Nr. 3, p. 324–342

[Velev et Bryant 2003] Velev, Miroslav N. ; Bryant, Randal E. : Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In : *J. Symb. Comput.* 35 (2003), February, p. 73–106

[Voronkov 1999] Voronkov, Andrei : KK : A Theorem Prover For K. In : *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, 1999, p. 383–387

[Walsh 1999] Walsh, Toby : Search in a small world. In : *Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999, p. 1172–1177

[Wang et al. 2007] Wang, Chao ; Gupta, Aarti ; Ivancic, Franjo : Induction in CEGAR for Detecting Counterexamples. In : *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, 2007, p. 77–84

[Wei et al. 2004] Wei, Wei ; Erenrich, Jordan ; Selman, Bart : Towards Efficient Sampling : Exploiting Random Walk Strategies. In : *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, 2004, p. 670–676

[Wei et Selman 2005] Wei, Wei ; Selman, Bart : A New Approach to Model Counting. In : *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, 2005, p. 324–339

[Weidenbach et al. 2009] Weidenbach, Christoph ; Dimova, Dilyana ; Fietzke, Arnaud ; Kumar, Rohit ; Suda, Martin ; Wischnewski, Patrick : SPASS Version 3.5. In : *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, 2009, p. 140–145

[Wetzler et al. 2014] Wetzler, Nathan ; Heule, Marijn ; Jr., Warren A. H. : DRAT-trim : Efficient Checking and Trimming Using Expressive Clausal Proofs. In : *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014, p. 422–429

[Wieringa et Heljanko 2013] Wieringa, Siert ; Heljanko, Keijo : Concurrent Clause Strengthening. In : *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, p. 116–132

[Zaatiti 2018] Zaatiti, Hadi : *Modeling and qualitative simulation of hybrid systems. (Modélisation et simulation qualitative de systèmes hybrides)*, University of Paris-Saclay, France, Ph.D. thesis, 2018

[Zaatiti et al. 2018] Zaatiti, Hadi ; Gallois, Jean-Pierre ; Ye, Lina ; Dague, Philippe : Automating Abstraction Computation of Hybrid Systems. In : *Joint Proceedings of the CME-EI, FMM, CAAT, FVPS, M3SRD, OpenMath Workshops, Doctoral Program and Work in Progress at the Conference on Intelligent Computer Mathematics 2018 co-located with the 11th Conference on Intelligent Computer Mathematics (CICM 2018), Hagenberg, Austria, August 13-17, 2018.*, 2018

[Zaatiti et al. 2017] Zaatiti, Hadi ; Ye, Lina ; Dague, Philippe ; Gallois, Jean-Pierre : Counterexample-Guided Abstraction-Refinement for Hybrid Systems Diagnosability Analysis. In : *28th International Workshop on Principles of Diagnosis (DX'17), Brescia, Italy, September 26-29, 2017*, 2017, p. 124–143

[Zanuttini 2003] Zanuttini, Bruno : *Acquisition de connaissances et raisonnement en logique propositionnelle*, Université de Caen, Ph.D. thesis, Juillet 2003. – Chapitre 3

[Zhang et Bonacina 1994] Zhang, Hantao ; Bonacina, Maria P. : Cumulating Search in a Distributed Computing Environment : A Case Study in Parallel Satisfiability. In : *Proceedings of the First Intelligence Symposium on Parallel Symbolic Computation*, Publishing Company, 1994, p. 422–431

[Zhang et al. 1996] Zhang, Hantao ; Bonacina, Maria P. ; Hsiang, Jieh : PSATO : a distributed propositional prover and its application to quasigroup problems. In : *J. Symb. Comput.* 21 (1996), June, p. 543–560

[Zhang et al. 2001] Zhang, Lintao ; Madigan, Conor F. ; Moskewicz, Matthew H. ; Malik, Sharad : Efficient conflict driven learning in a boolean satisfiability solver. In : *ICCAD '01 : Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA : IEEE Press, November 2001, p. 279–285. – ISBN 0-7803-7249-2

[Zhang et Malik 2002] Zhang, Lintao ; Malik, Sharad : The Quest for Efficient Boolean Satisfiability Solvers. In : *CADE-18 : Proceedings of the 18th International Conference on Automated Deduction*. London, UK : Springer-Verlag, 2002, p. 295–313. – ISBN 3-540-43931-5