Une approche basée sur la décomposition arborescente pour la résolution d'instances SAT structurées

Djamal Habet, Lionel Paris, Cyril Terrioux

Une approche basée sur la décomposition arborescente pour la résolution d'instances SAT structurées

Djamal Habet **Lionel Paris** **Cyril Terrioux**

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{djamal.habet, lionel.paris, cyril.terrioux}@univ-cezanne.fr

Résumé

Le principal objectif de ce papier est de proposer une nouvelle approche pour la résolution d'instances SAT structurées. La structure d'une instance SAT peut être capturée par un hypergraphe dont les sommets correspondent à ses variables et les hyperarêtes à ses clauses. La méthode proposée est basée sur une décomposition arborescente de cet hypergraphe qui servira principalement à guider le processus d'énumération d'un algorithme de type DPLL. Durant la recherche, cette méthode détecte certaines informations et les enregistre sous forme de *goods* et *nogoods* structurels. Par l'exploitation de telles informations, cette méthode évite de visiter inutilement plusieurs fois une zone de l'espace de recherche préalablement explorée. Elle garantit ainsi une borne de complexité théorique en temps qui dépend de la décomposition arborescente utilisée. Dans la partie expérimentale, cette approche sera évaluée sur des instances SAT structurées et comparée à d'autres approches.

1 Introduction

Le problème de satisfiabilité (SAT) consiste à tester si une formule booléenne écrite sous la forme normale conjonctive est satisfiable. SAT est l'un des problèmes NP-complets les plus étudiés à cause de son importance aussi bien sur le plan théorique que pratique. Encouragées par les progrès impressionnants dans la résolution pratique du problème SAT, diverses applications allant de la vérification formelle à la planification sont encodées et résolues dans le formalisme SAT. Par ailleurs, la majorité des solveurs actuels et efficaces pour SAT sont basés sur des algorithmes de recherche de type *backtrack* construits autour de la procédure Davis-Putnam-Logemann-Loveland (DPLL) [5]. Ces algorithmes sont améliorés par plusieurs techniques permettant notamment d'élaguer l'espace de recherche. Parmi ces techniques, on peut citer l'apprentissage, l'utilisation de méthodes liées à la propagation des contraintes étendues, à l'exploitation des symétries, ... L'impact de ces différentes techniques dépend particulièrement de la nature et du type des instances considérées. Par exemple, l'apprentissage peut s'avérer très utile et pertinent lors de la résolution d'instances SAT structurées et peu efficace dans la résolution d'instances générées aléatoirement. Cependant, dans le pire des cas, la complexité en temps de la résolution de SAT par un algorithme basé sur DPLL est donnée par $O(m2^n)$, où n et m sont respectivement le nombre de variables et le nombre de clauses de l'instance traitée.

Améliorer l'efficacité des solveurs par l'exploitation de la structure du problème considéré a été largement étudiée en CSP (Constraint Satisfaction Problem) (par exemple dans [10]) et d'une manière bien moins prononcée en SAT. La structure d'un problème est alors représentée par un (hyper)graphe. Par exploitation de la structure d'un problème, nous entendons le fait de tirer profit des propriétés structurelles qui peuvent être représentées et capturées par les propriétés de cet (hyper)graphe. Concernant SAT, la structure de l'instance modélisant un problème (une application) du monde réel est le résultat de la conception modulaire de celui-ci, tel que ces modules possèdent entre eux une inter-

connexion minimale exprimée par un nombre minimal de variables [12]. Par conséquent, la décomposition du problème initial en un ensemble de sous-problèmes pourrait faciliter sa résolution.

Partant de ce constat, le but de notre travail est de fournir une nouvelle approche basée sur la décomposition arborescente de l'hypergraphe qui représente une instance SAT. Cette décomposition offre un ordre sur les variables du problème qui sera exploité par une méthode de résolution de type DPLL. De plus, durant la recherche, plusieurs informations qui correspondent soit à des échecs, soit à des succès lors des tentatives de résolution de(s) (sous-)problème(s) sont apprises sous forme de *goods* et *nogoods*. L'avantage attendu d'un tel apprentissage est de pouvoir élaguer l'espace de recherche et de ne pas résoudre plusieurs fois le même sous-problème. Une telle démarche nous permet également de fournir des bornes de complexité en temps et en espace pour la méthode proposée. Nous appuyons notre travail par la présentation de résultats expérimentaux et une comparaison avec certaines méthodes existantes de la littérature.

La papier est organisé comme suit. La section 2 introduit les notions de base et les notations nécessaires au reste du papier. Elle inclut notamment la définition formelle du problème SAT, de certains concepts liés à la théorie des graphe et un court rappel de l'algorithme DPLL. La section 3 présente notre approche basée sur la notion de décomposition arborescente, que nous appelons DPLL-TD, en posant d'abord les bases théoriques qui sont obligatoires pour ensuite décrire DPLL-TD et justifier sa validité. Nous fournirons également, dans cette même section, les complexités en temps et en espace de notre algorithme. La section 4 présente quelques détails sur l'implémentation de DPLL-TD et expose les résultats obtenus sur un certain nombre d'instances SAT structurées issues des précédentes compétitions SAT. La section 5 recense les travaux existants autour de la notion de décomposition arborescente dans le cadre principalement du problème SAT. Enfin, la section 6 discute et conclut ce travail.

2 Notations

Cette section est dédiée à la définition du problème SAT et des notations qui lui correspondent, à un rappel de l'algorithme DPLL ainsi qu'à l'introduction d'un certain nombre de concepts de la théorie des graphes nécessaires à la compréhension du papier.

2.1 Le problème de satisfiabilité

Une instance $\mathcal{F}$ du problème de satisfiabilité (SAT) est définie par le couple $\mathcal{F} = (\mathcal{X}, \mathcal{C})$, tel que $\mathcal{X}$ est un ensemble de variables booléennes (leurs valeurs appartiennent à l'ensemble $\{vrai,\ faux\}$) et $\mathcal{C}$ est un ensemble de clauses. Une clause est une disjonction finie de littéraux et un littéral est soit une variable, soit sa négation.

Pour un littéral donné l, $var(l) = \{v | l = v\ ou\ l = \neg v\}$ correspond à l'ensemble singleton de la variable sur laquelle porte l. Par ailleurs, un littéral peut-être considéré comme une clause contenant uniquement ce littéral, ce qui correspond à la définition de la clause unitaire. De plus, pour une clause donnée c, $var(c) = \cup_{l\ dans\ c} var(l)$ est l'ensemble des variables apparaissant dans c (positivement ou négativement). Par exemple, si $c = x_1 \vee \neg x_2 \vee \neg x_3$ alors nous avons $var(\neg x_2) = \{x_2\}$ et $var(c) = \{x_1,\ x_2,\ x_3\}$.

Une interprétation I des variables de $\mathcal{F}$ est définie par un ensemble de littéraux, tel que $\forall (l_1,\ l_2) \in I^2,\ l_1 \neq l_2,\ var(l_1) \neq var(l_2)$. Également, une variable qui apparaît positivement (respectivement négativement) dans I signifie qu'elle est fixée à $vrai$ (respectivement à $faux$). Aussi, l'interprétation I est dite partielle si $|I| < |\mathcal{X}|$, complète sinon (toutes les variables sont fixées). De plus, pour une interprétation donnée I d'un ensemble de variables $Y \subseteq \mathcal{X}$ et pour un sous-ensemble $Z \subseteq Y$, $I[Z]$ est la projection de I réduite aux variables de Z. Enfin, un modèle de $\mathcal{F}$ est une interprétation complète qui satisfait toutes les clauses de $\mathcal{F}$ et $Sol(\mathcal{F})$ désigne l'ensemble de tous les modèles de $\mathcal{F}$.

Par conséquence, le problème de satisfiabilité (SAT) consiste à tester si $\mathcal{F}$ possède un modèle ($Sol(\mathcal{F}) \neq \emptyset$). Si c'est le cas alors $\mathcal{F}$ est dite satisfiable, sinon $\mathcal{F}$ est insatisfiable.

2.2 L'algorithme DPLL

Introduite en 1960 et en dépit de sa simplicité, la procédure de Davis-Putnam-Logemann-Loveland (DPLL) [5] subsiste comme l'une des meilleures méthodes complètes pour SAT. La procédure DPLL est un algorithme de type *backtrack*. Elle construit un arbre de recherche binaire dans lequel chaque nœud est associé à un appel récursif. Toutes les feuilles, exceptées éventuellement une pour les instances satisfiables, représentent l'aboutissement de la procédure à une contradiction (correspondant à une clause vide notée par $\square$).

Plus précisément, à chaque itération (appel récursif), DPLL choisit une variable non interprétée (libre), lui affecte la valeur $vrai$ et simplifie $\mathcal{C}$. Si l'instance résultant de l'appel courant est satisfiable alors $\mathcal{F}$ est également satisfiable. Dans le cas contraire, DPLL affecte la valeur $faux$ à cette variable et le même processus est répété. L'opération de simplification consiste essentiellement à supprimer les clauses qui deviennent satisfaites sous l'interprétation courante I et à réduire les clauses qui contiennent des littéraux falsifiés (évalués à $faux$ sous I également).

Le choix de la variable de branchement est un facteur primordial de la procédure DPLL dont les performances dépendent directement. La littérature regorge de plusieurs

Algorithme 1 : DPLL(in : $\mathcal{C}$, I)

```
1  Propagation-unitaire (C, I)
2  si □ ∈ C alors retourner faux
3  sinon
4      si C = ∅ alors
5          retourner vrai
6      sinon
7          Choisir une variable libre v
8          si DP (C ∪ {v}, I ∪ {v}) alors retourner vrai
9          sinon
10             retourner DP (C ∪ {¬v}, I ∪ {¬v})
```

Algorithme 2 : Propagation-Unitaire(in/out : $\mathcal{C}$, I)

```
1  tant que Il n'existe pas de clauses vides et qu'il existe une clause unitaire l
   dans F faire
2      I ← I ∪ l (satisfaire l)
3      simplifier C
```

heuristiques de branchement qui sont généralement basées sur la propagation unitaire et le nombre d'occurrences des variables (sous leurs formes positives ou négatives) dans les clauses.

2.3 Théorie des graphes

Comme nous l'avons annoncé dans l'introduction, notre but est d'exploiter les propriétés structurelles d'une instance SAT exprimées par sa représentation sous forme d'un graphe, qui sera donc décomposé en conséquence. Nous allons formuler et rappeler ici quelques définitions liées à ces points.

Définition 1 *Soit l'instance SAT $\mathcal{F} = (\mathcal{X}, \mathcal{C})$. L'hypergraphe $H = (\mathcal{V}, \mathcal{E})$ qui caractérise l'instance $\mathcal{F}$ est défini tel que toute variable de $\mathcal{X}$ est représentée par un sommet de $\mathcal{V}$ et chaque clause $c \in \mathcal{C}$ est représentée par une hyper-arête $e \in \mathcal{E}$ (c'est-à-dire un sous-ensemble de $\mathcal{V}$) telle que $var(c) = e$.*

Par exemple, considérons l'instance SAT $\mathcal{F} = (\{x_1, x_2, x_3\}, \{x_1 \vee \neg x_2 \vee x_3,\ x_1 \vee \neg x_3,\ x_2 \vee x_3\})$. Selon la définition précédente, l'hypergraphe lui correspondant est $H = (\{x_1, x_2, x_3\}, \{\{x_1, x_2, x_3\},\ \{x_1, x_3\},\ \{x_2, x_3\}\})$.

Nous introduisons, à présent, la notion de graphe primal qui à un hypergraphe permet d'associer un graphe. Cette notion est nécessaire pour pouvoir exploiter, par la suite, la notion de décomposition arborescente qui est définie sur les graphes, et non sur les hypergraphes.

Définition 2 *Le graphe primal d'un hypergraphe $H = (\mathcal{V}, \mathcal{E})$ est le graphe $G_H = (\mathcal{V}, \mathcal{E}_H)$ tel que $\mathcal{E}_H = \{\{x, y\} | x, y \in V$ et $\exists e \in \mathcal{E}, \{x, y\} \subseteq e\}$.*

Reprenons la formule SAT $\mathcal{F}$ ci-dessous et sa représentation sous forme de l'hypergraphe H. Le graphe primal associé à H est donc $G_H = (\{x_1, x_2, x_3\}, \{\{x_1, x_2\},\ \{x_1, x_3\},\ \{x_2, x_3\}\})$.

On note par $G_{(\mathcal{X},\mathcal{C})}$ le graphe primal associé à une instance $(\mathcal{X}, \mathcal{C})$. C'est ce graphe qui sera finalement utilisé pour représenter la structure d'une instance SAT et pouvoir exploiter ses propriétés structurelles notamment au travers de la notion de décomposition arborescente définie ci-dessous [14].

Définition 3 *Soit $G_{(\mathcal{X},\mathcal{C})} = (\mathcal{V}, \mathcal{E}_H)$ le graphe primal associé à une instance $(\mathcal{X}, \mathcal{C})$. Une décomposition arborescente de $G_{(\mathcal{X},\mathcal{C})}$ est définie par le couple $(E, \mathcal{T})$, où d'une part $\mathcal{T} = (J, F)$ est un arbre dont l'ensemble des nœuds est représenté par J et l'ensemble des arêtes par F et d'autre part, $E = \{E_i : i \in J\}$ est une famille de sous-ensembles de $\mathcal{X}$, tel que chaque sous-ensemble (appelé cluster) E_i est un nœud de $\mathcal{T}$ et qui vérifie :*

(i) $\cup_{i \in J} E_i = \mathcal{V}$,

(ii) pour toute arête $\{x, y\} \in \mathcal{E}_H$, il existe $i \in J$ tel que $\{x, y\} \subseteq E_i$, et

(iii) pour tout $i, j, k \in J$, si k est sur une chaîne entre i et j dans $\mathcal{T}$, alors $E_i \cap E_j \subseteq E_k$.

La largeur de la décomposition arborescente $(E, \mathcal{T})$ est égale à $max_{i \in J}|E_i| - 1$. La largeur d'arbre (ou tree-width) w de $G_{(\mathcal{X},\mathcal{C})}$ est la plus petite largeur sur toutes les décompositions arborescente de $G_{(\mathcal{X},\mathcal{C})}$.

On note par $Desc(E_j)$ l'ensemble des variables appartenant à E_j ou à un descendant E_k de E_j. Aussi, soient E_i un cluster et E_j l'un de ses clusters fils, on désigne par $E_{par(j)}$ le cluster parent E_i de E_j et on suppose que $E_{par(1)} = \emptyset$ (E_1 étant le cluster racine). De plus, $\mathcal{C}[E_i]$ est l'ensemble des clauses appartenant exclusivement au cluster E_i. En d'autres mots, nous avons $\mathcal{C}[E_i] = \{c \in \mathcal{C} | var(c) \subseteq E_i\ et\ var(c) \not\subseteq E_{par(i)}\}$.

Nous illustrons ces différentes définitions et notations à l'aide de l'exemple ci-dessous où la figure 1 correspond à une décomposition arborescente d'une une instance SAT donnée (non présentée ici). $E = \{E_i | i = 1, \ldots, 7\}$ est l'ensemble des clusters et E_1 est le cluster racine de cette décomposition arborescente. E_1 possède trois clusters fils E_2, E_3 et E_4, $E_2 = \{x_1, x_2, x_6, x_7\}$, $E_1 \cap E_2 = \{x_1,\ x_2\}$ (cette intersection est appelée *le séparateur* entre le cluster E_2 et son parent E_1), $E_{par(3)} = E_1$ et $Desc(E_3) = \{x_3, x_4, x_8, x_{15}, x_{18}, x_{20}\}$.

Avant de discuter les détails théoriques et algorithmiques de notre approche, nous allons d'abord donner ici l'idée de base qui est derrière. La décomposition arborescente de l'instance initiale divise celle-ci en parties (plus petites) et indépendantes, qui correspondent au clusters, mais qui restent toutefois interconnectées par les séparateurs. Résoudre

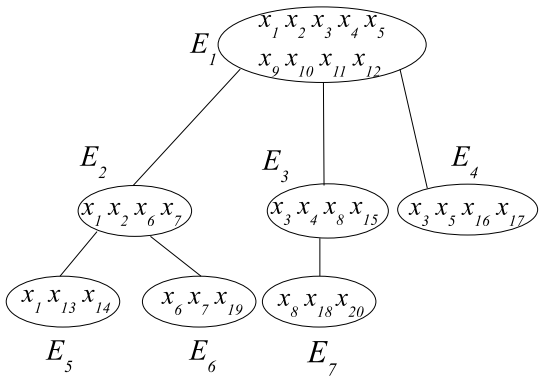

FIGURE 1 – Un exemple de décomposition arborescente.

l'instance initiale revient donc à résoudre ses différentes parties. Partant de la racine et par un parcours de type en profondeur d'abord, chaque cluster E_i est attaqué séparément. Si toutes les clauses restreintes à ce cluster sont satisfaites alors nous allons attaquer l'un de ses fils E_j (s'il existe) en tentant d'étendre l'interprétation des variables de E_i sur celles de E_j. Pour être plus précis, il est nécessaire de satisfaire les clauses de E_j en respectant les contraintes exprimées par l'interprétation des variables du séparateur entre E_i et E_j. En effet, ces variables sont partagées entre les deux clusters et par conséquence doivent être interprétées d'une manière identique. Si ce processus d'extension ne peut se faire alors on peut déduire qu'il ne sera jamais possible d'étendre l'interprétation des variables du séparateurs à un modèle. Plus intéressant encore, sauvegarder cette information peut-être utile pour éviter la répétition d'un même traitement dans le cas d'un retour-arrière au cluster E_i par exemple et l'occurrence de la même interprétation sur les variables du séparateur. De la même manière et pour les mêmes raisons, si les clauses du cluster E_i et celles de tous ses descendants sont satisfaites alors il est aussi utile de sauvegarder l'interprétation des variables du séparateur entre E_i et et son parent comme étant extensible à un modèle pour cette partie de l'instance. Pour résumé, nous appliquons une recherche de type *backtrack* sur la décomposition arborescente de l'instance SAT, tout en enregistrant toute information utile qui nous permettra d'élaguer l'espace de recherche.

3 Décomposition arborescente pour résoudre SAT

Cette section est la contribution majeure de notre travail, Dans un premier temps, elle souligne et explique les aspects théoriques liés à l'usage de la décomposition arborescente dans le cadre de SAT. Dans un second temps, elle introduit et détaille DPLL-TD qui est la traduction algorithmique de ces aspects théoriques. Enfin, nous exhibons des preuves sur les caractéristiques de DPLL-TD, à savoir qu'il est complet, correct et termine ainsi que sa complexité en temps et en espace.

3.1 Fondements théoriques

Dans la suite, nous considérons une instance SAT $\mathcal{F} = (\mathcal{X}, \mathcal{C})$ et une décomposition arborescente $(E, \mathcal{T})$ associée au graphe $G_{(\mathcal{X},\mathcal{C})}$.

Le but de ce premier théorème est de montrer que la décomposition arborescente de $G_{(\mathcal{X},\mathcal{C})}$ ne change en rien l'instance de départ (aucune information n'est perdue) et que chaque clause appartient à un et un seul cluster.

Théorème 1 *Les ensembles $(\mathcal{C}[E_i])_i$ forment une partition de $\mathcal{C}$.*

Preuve : De toute évidence, nous avons $\bigcup_i \mathcal{C}[E_i] \subseteq \mathcal{C}$. Démontrons maintenant que $\mathcal{C} \subseteq \bigcup_i \mathcal{C}[E_i]$.

Considérons une clause c. Par définition du graphe $G_{(\mathcal{X},\mathcal{C})}$, les sommets correspondant aux variables de $var(c)$ forment une clique de $G_{(\mathcal{X},\mathcal{C})}$. Ainsi, par définition de la décomposition arborescente, il existe au moins un cluster E_i tel que $var(c) \subseteq E_i$. En particulier, nous avons nécessairement un cluster E_k tel que $var(c) \subseteq E_k$ et $var(c) \not\subseteq E_{par(k)}$. Ainsi $c \in \mathcal{C}[E_k]$. D'où, $\bigcup_i \mathcal{C}[E_i] = \mathcal{C}$.

Maintenant, nous allons démontrer que pour des clusters quelconques E_i et E_j ($i \neq j$), $\mathcal{C}[E_i] \cap \mathcal{C}[E_j] = \emptyset$. Supposons que l'intersection n'est pas vide. Donc, il existe une clause c telle que $c \in \mathcal{C}[E_i]$ et $c \in \mathcal{C}[E_j]$. Il s'ensuit alors que $var(c) \subseteq E_i \cap E_j$. Par ailleurs, par définition de la décomposition arborescente, il existe une chaîne reliant E_i et E_j telle que chaque cluster rencontré sur cette chaîne contient $E_i \cap E_j$ et par conséquent $var(c)$. Sur une telle chaîne, on traversera nécessairement le père de E_i ou celui de E_j. Ainsi, nous avons $var(c) \subseteq E_{par(i)}$ ou $var(c) \subseteq E_{par(j)}$. Par conséquent, nous avons une contradiction puisque $c \in \mathcal{C}[E_i]$ et $c \in \mathcal{C}[E_j]$. Il en résulte que $\mathcal{C}[E_i] \cap \mathcal{C}[E_j] = \emptyset$.

En conclusion, les ensembles $(\mathcal{C}[E_i])_i$ forment effectivement une partition de $\mathcal{C}$. □

A partir des ensembles $\mathcal{C}[E_i]$, nous pouvons définir la notion de sous-problème induit :

Définition 4 *Soit E_i un cluster. Le sous-problème $\mathcal{F}_{E_i,I}$ enraciné en E_i et induit par l'interprétation I sur un sous-ensemble de $E_i \cap E_{par(i)}$ est défini par $(Desc(E_i), \bigcup_{E_k \subseteq Desc(E_i)} \mathcal{C}[E_k] \cup \{l | l \in I \text{ et } var(l) \subseteq E_i \cap E_{par(i)}\})$.*

En d'autres termes, $\mathcal{F}_{E_i,I}$ est constitué par toutes les clauses et variables du cluster E_i et celles des clusters de sa

descendance, avec des contraintes additionnelles exprimant l'interprétation courante des variable dans le séparateur entre E_i et son cluster parent. Ces contraintes sont formulées simplement par $\{l | l \in I \; et \; var(l) \subseteq E_i \cap E_{par(i)}\})$. Cette définition signifie aussi que résoudre le problème correspondant au sous-arbre enraciné à E_i doit respecter l'interprétation des variables qu'il partage avec son parent.

Propriété 1 *Considérons deux interprétations I et I' telles que $I \subseteq I'$ et un cluster E_i, nous avons alors $Sol(\mathcal{F}_{E_i,I'}) \subseteq Sol(\mathcal{F}_{E_i,I})$.*

Preuve : La seule différence entre les sous-problèmes $\mathcal{F}_{E_i,I'}$ et $\mathcal{F}_{E_i,I}$ est que le premier possède un certain nombre de clauses supplémentaires, notamment les clauses unitaires v tel que $v \in I' - I$ et $var(v) \subseteq E_i \cap E_{par(i)}$. Ainsi, nous avons nécessairement $Sol(\mathcal{F}_{E_i,I'}) \subseteq Sol(\mathcal{F}_{E_i,I})$. □

Cette propriété permet donc de caractériser la relation existant entre les ensembles de modèles de deux sous-problèmes enracinés en un même cluster E_i mais induit par deux interprétations différentes dont l'une est une extension de l'autre.

La définition suivante présente maintenant la notion de variable d'indépendance :

Définition 5 *Une variable v est une variable d'indépendance s'il existe un cluster E_i et deux de ses fils E_j et E_k tels que $v \in E_i \cap E_j \cap E_k$. On désigne par $\mathcal{X}_{ind}$ l'ensemble des variables d'indépendance de $\mathcal{F}$ par rapport à la décomposition arborescente considérée.*

Les variables d'indépendance jouent clairement un rôle particulier dans la décomposition arborescente. Plus précisément, pour pouvoir décomposer de façon valide un problème SAT en différents sous-problèmes indépendants, nous devons garantir que ces variables sont interprétées avec la même valeur dans les différents sous-problèmes dans lesquels elles interviennent. Par exemple, la décomposition arborescente de la figure 1 possède une seule variable d'indépendance x_3 localisée dans l'intersection de E_1 avec deux de ses fils E_3 et E_4. Donc, $\mathcal{X}_{ind} = \{x_3\}$. Si x_3 n'est pas interprétée lors de traitement de E_1 (rappelons qu'il est possible de satisfaire un ensemble de clauses sans nécessairement fixer toutes les variables apparaissant dans celles-ci) alors x_3 doit-être interprétée avec la même valeur lors du traitement des problèmes enracinés en E_3 et E_4. Le théorème suivant formalise l'indépendance entre sous-problèmes en termes de variables d'indépendance.

Théorème 2 *Soient un cluster E_i, E_j et E_k deux fils de E_i et une interprétation I d'un sous-ensemble Y de E_i. Si $\mathcal{X}_{ind} \cap E_i \subseteq Y$, alors les sous-problèmes $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ sont indépendants.*

Preuve : A partir de la définition de la décomposition arborescente, nous avons $Desc(E_j) \cap Desc(E_k) = E_j \cap E_k \subseteq E_i$. Clairement, nous avons $E_j \cap E_k \subseteq \mathcal{X}_{ind}$ et par conséquent $E_j \cap E_k \subseteq \mathcal{X}_{ind} \cap E_i$. Puisque les variables de $\mathcal{X}_{ind} \cap E_i$ sont interprétées dans I, cela assure que ces mêmes variables ont les mêmes interprétations sur n'importe quelle interprétation qui satisfait $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$, au regard des clauses unitaires v telles que $v \in I$ et $var(v) \subseteq E_i \cap E_j \cap E_k$. De plus, ces clauses unitaires sont les seules à appartenir à la fois à $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et à $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ (par application du théorème 1). Ainsi, les problèmes $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ sont bien indépendants. □

Le corollaire suivant établit que la satisfiabilité de sous-problèmes frères conduit à la satisfiabilité du sous-problème enarciné en leur cluster parent.

Corollaire 1 *Soient un cluster E_i et une interprétation I sur un sous-ensemble Y de E_i, si $\mathcal{X}_{ind} \cap E_i \subseteq Y$, I satisfait les clauses de $\mathcal{C}[E_i]$, et pour chaque fils E_j de E_i, le sous-problème $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ est satisfiable, alors I peut-être étendue à un modèle de $\mathcal{F}_{E_i,I[E_i \cap E_{par(i)}]}$.*

Preuve : Soit M_{E_j} un modèle pour le sous-problème $\mathcal{F}_{E_j,I[E_i \cap E_j]}$. Par définition de $\mathcal{F}_{E_j,I[E_i \cap E_j]}$, il n'existe pas de variables v tel que $I \cup M_{E_j}$ contienne deux littéraux opposés à v. Selon le théorème 2, pour tout fils E_j et E_k de E_i, $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ sont indépendants. Ainsi, l'interprétation $I \cup \bigcup_{E_j \in fils(E_i)} M_{E_j}$ satisfait à la fois les clauses de $\mathcal{C}[E_i]$ et celles de $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ pour chaque fils E_j de E_i. De cette manière, I peut-être étendue à un modèle de $\mathcal{F}_{E_i,I[E_i \cap E_{par(i)}]}$.□

Comme nous l'avons expliqué juste avant le début de cette section, nous voulons également garder trace de toute information utile qui nous permettra d'éviter plusieurs fois un même traitement, principalement la résolution récurrente de sous-problèmes. Cela revient à effectuer des coupes sur les branches de l'arbre explorant l'espace de recherche en présence d'informations indiquant si la satisfiabilité d'un sous-problème donné est déjà connue ou pas. De telles informations correspondent aux *goods* et *nogoods* que nous définissons comme suit :

Définition 6 *Etant donné un cluster E_i, une interprétation I sur un sous-ensemble de $E_i \cap E_{par(i)}$ est un* good *(respectivement un* nogood*) structurel de E_i si toute extension de I sur $E_i \cap E_{par(i)}$ peut-être étendue à un modèle de $\mathcal{F}_{E_i,I}$ (respectivement si $Sol(\mathcal{F}_{E_i,I}) = \emptyset$).*

En d'autres termes, un *good* (respectivement un *nogood*) structurel est une interprétation I d'un sous-ensemble de $E_i \cap E_{par(i)}$ qui peut (respectivement ne peut pas) être étendue à un modèle pour $\mathcal{F}_{I,E_i}$.

Propriété 2 *Soient un cluster E_i et un sous-ensemble $Y \subseteq X$ tel que $Desc(E_i) \cap Y \subseteq E_i \cap E_{par(i)}$. Pour tout* good *$g$ de E_i, toute interprétation I sur Y peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$ s'il existe une extension e_g de g sur $E_i \cap E_{par(i)}$ telle que $I[E_i \cap E_{par(i)}] \subseteq e_g$.*

Preuve : Par définition d'un *good*, e_g et par conséquent $I[E_i \cap E_{par(i)}]$ (puisque $I[E_i \cap E_{par(i)}] \subseteq e_g$) peuvent être étendues à un modèle M de $\mathcal{F}_{E_i,g}$. Il s'ensuit que M satisfait les clauses de $\bigcup_{E_k \subseteq Desc(E_i)} \mathcal{C}[E_k]$. De plus, il satisfait également les clauses unitaires de $\{v | v \in I \ et \ var(v) \subseteq E_i \cap E_{par(i)}\}$ puisque $I[E_i \cap E_{par(i)}] \subseteq e_g \subseteq M$. Ainsi, M est un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$ et $I[E_i \cap E_{par(i)}]$ peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$. □

La propriété précédente nous donne les conditions qui nous autorisent à effectuer une coupe selon les *goods* enregistrés. De la même manière, la propriété suivante exprime la condition de coupe par les *nogoods*.

Propriété 3 *Soient un cluster E_i et un sous-ensemble $Y \subseteq X$ tel que $Desc(E_i) \cap Y \subseteq E_i \cap E_{par(i)}$. Pour tout* nogood *$ng$ de E_i, aucune interprétation I sur Y tel que $ng \subseteq I$ ne peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$.*

Preuve : Nous avons $Sol(\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}) \subseteq Sol(\mathcal{F}_{E_i, ng})$ (propriété 1). Par ailleurs, par définition d'un *nogood*, $Sol(\mathcal{F}_{E_i, ng}) = \emptyset$. Donc $Sol(\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}) = \emptyset$ et I ne peut-être donc étendue à un modèle sur $Desc(E_i)$. □

Maintenant que nous avons apporté et démontré tous les éléments théoriques liés à notre approche, nous décrivons dans la section suivante leur exploitation algorithmique.

3.2 L'algorithme DPLL-TD

Considérons une instance SAT définie par $\mathcal{F} = (\mathcal{X}, \mathcal{C})$ et la décomposition arborescente $(E, \mathcal{T})$ du graphe primal obtenu à partir de la représentation en hypergraphe de $\mathcal{F}$. L'algorithme 3 décrit la méthode DPLL-TD (pour *DPLL with Tree Decomposition*) pour résoudre le problème de satisfiabilité. Durant la recherche, DPLL-TD tente d'étendre l'interprétation courante à un modèle, s'il existe. Pour cela, et en plus de l'ordre dicté par l'heuristique de choix de variables de branchement d'une procédure de type DPLL, la méthode DPLL-TD se sert de la décomposition arborescente $(E, \mathcal{T})$ afin d'affiner ce choix comme cela sera expliqué dans la suite. De plus, DPLL-TD exploite des informations préalablement apprises sous formes de *goods* (enregistrés dans $\mathcal{G}$) et de *nogoods* (enregistrés dans $\mathcal{N}$) dans le but d'élaguer l'espace de recherche.

Ainsi décrit, l'appel DPLL-TD($\mathcal{C}, I, E_i, \mathcal{G}, \mathcal{N}$) renvoie $vrai$ (respectivement $faux$) si le sous-problème $\mathcal{F}_{E_i, I}$ enraciné en E_i est satisfiable (respectivement insatisfiable).

Avant de détailler l'algorithme DPLL-TD, quelques notations et précisions supplémentaires sont nécessaires en plus de celles préalablement introduites. Les *goods* correspondant à un cluster donné E_i sont représentés et enregistrés dans l'ensemble $\mathcal{G}_{E_i}$ et $\mathcal{G}$ est un ensemble défini par $\mathcal{G} = \{\mathcal{G}_{E_i} | E_i \in E\}$. Par ailleurs, $\mathcal{N}$ désigne l'ensemble des *nogoods*.

Algorithme 3 : DPLL-TD(in : $\mathcal{C}$, I, E_i, in/out : $\mathcal{G}, \mathcal{N}$)

```
1  Propagation-unitaire (C ∪ N, I)
2  si □ ∈ C ∪ N alors retourner faux
3  sinon
4      si C ∪ N = ∅ alors retourner vrai
5      sinon
6          si (C ∪ N)[E_i] = ∅ et X_ind ∩ E_i = ∅ alors
7              sat ← vrai
8              S ← Fils(E_i)
9              tant que sat et S ≠ ∅ faire
10                 Choisir un cluster E_j dans S
11                 S ← S − {E_j}
12                 si ∃g ∈ G_{E_j}, ∃e_g extension de g sur
                   E_i ∩ E_j, I[E_i ∩ E_j] ⊆ e_g alors
13                     I ← I ∪ g
14                 sinon
15                     sat ← DPLL-TD (C, I, E_j, G, N)
16                     si sat alors
17                         Soit g ∈ G_{E_j} le dernier good enregistré
18                         I ← I ∪ g
                           sinon N ← N ∪ { ⋁_{l_k ∈ I[E_i ∩ E_j]} ¬l_k }
19
20             si sat alors G_{E_i} ← G_{E_i} ∪ {I[E_i ∩ E_par(i)]}
21             retourner sat
22         sinon
23             Choisir une variable non interprétée v dans E_i
24             si DPLL-TD (C ∪ {v}, I ∪ {v}, E_i, G, N) alors
25                 retourner True
26             sinon retourner DPLL-TD
               (C ∪ {¬v}, I ∪ {¬v}, E_i, G, N)
```

Le premier appel à DPLL-TD est fait avec les paramètres $(\mathcal{C}, \emptyset, E_1, \mathcal{G}, \mathcal{N})$, où E_1 est le cluster racine de la décomposition arborescente. En effet, au départ, aucune variable n'est interprétée. De même, aucun *(no)good* n'est connu (pour chaque cluster E_i, $\mathcal{G}_{E_i} = \emptyset$ et $\mathcal{N} = \emptyset$).

La première étape de cet algorithme (ligne 1) consiste à propager les clauses unitaire appartenant à $\mathcal{C} \cup \mathcal{N}$. En conséquence, si une clause vide est déduite alors l'algorithme retourne $faux$ signifiant que $\mathcal{F}$ est insatisfiable. Sinon, si $\mathcal{C} \cup \mathcal{N}$ est vidée alors $\mathcal{F}$ est satisfiable et l'algorithme retourne $true$ (ligne 4).

Maintenant, considérons l'ensemble des clauses $(\mathcal{C} \cup \mathcal{N})[E_i]$ et l'ensemble des variables d'indépendance restreintes au cluster courant E_i, $\mathcal{X}_{ind} \cap E_i$. Si l'un de ces deux ensembles n'est pas vide alors l'algorithme 3 procède à une énumération DPLL classique sur E_i dans le but d'interpréter ces variables (lignes 23 à 26). Par exemple, dans la figure 1, $x_3 \in \mathcal{X} \cap E_1$. Si x_3 n'est pas fixée alors une énumération sur x_3 (et éventuellement sur les variables non interprétées de $(\mathcal{C} \cup \mathcal{N})[E_1]$) est effectuée afin d'assurer la même interprétation de x_3 dans les clusters E_1, E_2 et E_3.

Ainsi, on garantit l'indépendance des sous-problèmes enracinés en E_3 et E_4 (corollaire 1).

Dans le cas contraire ($(\mathcal{C} \cup \mathcal{N})[E_i] = \emptyset$ et $\mathcal{X}_{ind} \cap E_i = \emptyset$), DPLL-TD traite les sous-problèmes enracinés en chaque fils E_j de E_i (pour rappel, si tous les sous-problèmes $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ sont satisfaits alors $\mathcal{F}_{E_i, I}$ est également satisfait).

Lors du traitement des fils de E_i, les *goods* et *nogoods* déjà enregistrés peuvent être utiles. En effet, DPLL-TD vérifie s'il existe un *good* g dans $\mathcal{G}_{E_j}$ entre E_j (fils de E_i) et E_i qui respecte la condition de coupe donnée par la propriété 2 (ligne 12). Si un tel *good* g existe alors $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ est satisfiable et il n'est pas nécessaire d'étudier à nouveau sa satisfiabilité. Inversement, si la satisfiabilité de $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ est inconnue, elle sera déterminée par l'appel récursif DPLL-TD($\mathcal{C}, I, E_j, \mathcal{G}, \mathcal{N}$). Dans ce dernier cas, si $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ est insatisfiable (le dernier appel renvoie $faux$), un *nogood* est détectée entre E_j et E_i. Celui-ci est représenté et enregistré comme une nouvelle clause définie par $\vee_{l_k \in I[E_i \cap E_j]} \neg l_k$ qui est ensuite rajoutée à l'ensemble des *nogoods* $\mathcal{N}$ (ligne 19). Revenons à la figure 1 et considérons le cluster E_1, son fils E_2 et une interprétation sur $E_1 \cap E_2 = \{x_1, x_2\}$ qui est $I[\{x_1, x_2\}] = \{\neg x_1, x_2\}$. Si $F_{2, \{\neg x_1, x_2\}}$ est insatisfiable alors un *nogood* est détecté et sauvegardé par la nouvelle clause $x_1 \vee \neg x_2$ dans $\mathcal{N}$.

Lors de l'occurrence de telles contradictions, DPLL-TD effectue un retour-arrière sur les variables de E_i et teste l'existence d'une nouvelle interprétation qui satisfait $(\mathcal{C} \cup \mathcal{N})[E_i]$. A noter que même si les clauses apprises (correspondant aux *nogoods*) sont sauvegardées dans un ensemble distinct $\mathcal{N}$, elles sont en pratique considérées comme une clause quelconque de $\mathcal{C}$. Nous effectuons cette distinction seulement dans le but de mettre en avant les *nogoods* appris tout le long de la recherche. Par ailleurs, la représentation sous forme clausale des *nogoods* nous dispense de l'application d'un traitement particulier lors de leur exploitation (incluant le test sur la possibilité de coupe dans l'arbre de recherche grâce aux *nogoods* appris). Ainsi, l'usage des *nogoods* est entièrement transparent dans DPLL-TD. Par ailleurs, ces clauses apprises permettent d'améliorer l'effet filtrant des propagations unitaires.

Finalement, les lignes 13 et 18 correspondent à l'extension de l'interprétation courante par le *good* utilisé par la coupe (par la propriété 2). Ainsi, I est étendue par l'ajout des littéraux du *good* g à ceux de I assurant un enregistrement d'un *good* valide sur E_i si $\mathcal{F}_{E_i, I}$ est satisfiable (ligne 20), où un *good* correspond à l'interprétation obtenue sur les variables de $E_i \cap E_{par(i)}$. A noter qu'aucune contradiction ne peut-être trouvée lors de cette extension, car les variables d'indépendance sont préalablement fixées comme expliqué précédemment. Par exemple, dans la figure 1, considérons le cluster E_2, son fils E_6 et une interprétation sur $E_2 \cap E_6 = \{x_6, x_7\}$ qui est $I[\{x_6, x_7\}] = \{x_6\}$. De plus, supposons qu'il existe un *good* $g = \{x_6, \neg x_7\} \in \mathcal{G}_{E_6}$. Par l'application de la propriété 2, $F_{E_6, \{x_6\}}$ est satisfiable et l'interprétation courante est étendue par $\{\neg x_7\}$.

3.3 Propriétés de DPLL-TD

Après avoir détailler le déroulement de l'algorithme DPLL-TD, nous allons nous intéresser à sa complexité en temps et en espace, sa complétude, sa validité et sa terminaison.

Théorème 3 *DPLL-TD est correct, complet et termine.*

Preuve : DPLL-TD diffère de DPLL par l'enregistrement des *goods* et des *nogoods* structurels utilisés afin d'éviter les redondances dans la recherche. Puisque DPLL est correct, complet et termine, il reste à prouver que les opérations supplémentaires effectuées par DPLL-TD ne remettent pas en cause ces propriétés.

Supposons que DPLL-TD($\mathcal{C}, I, E_i, \mathcal{G}, \mathcal{N}$) est l'appel courant. On veut vérifier si le sous-problème $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$ est satisfiable. Pour cela, considérons le cas où I satisfait toutes les clauses de $(\mathcal{C} \cup \mathcal{N})[E_i]$. DPLL-TD essaiera d'étendre I sur chacun des fils de E_i. Soit E_j l'un de ces fils. Si la condition de la ligne 12 est vérifiée alors (et grâce à la propriété 2) $\mathcal{F}_{E_j, I[E_j \cap E_i]}$ est satisfiable. Dans le cas contraire, la satisfiabilité de $\mathcal{F}_{E_j, I[E_j \cap E_i]}$ est inconnue et pour la déterminer, il faut faire un nouvel appel à DPLL-TD par DPLL-TD($\mathcal{C}, I, E_j, \mathcal{G}, \mathcal{N}$). Si ce dernier renvoie $faux$, cela signifie que $\mathcal{F}_{E_j, I[E_j \cap E_i]}$ ne possède pas de modèle. De ce fait, l'opération d'enregistrement d'un nouveau *nogood* sous la forme d'une clause $\bigvee_{l_k \in I[E_i \cap E_j]} \neg l_k$ est valide. Etant donné qu'un *nogood* structurel n'est qu'un cas particulier d'un *nogood* classique, son usage comme n'importe quelle clause de C est aussi valide.

Par ailleurs, à la fin de la boucle **tant que**, si I peut-être étendue à un modèle sur tous ses fils, alors (et par l'application du corollaire 1), $I[E_i \cap E_{par(i)}]$ peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$. De plus, puisque I était étendue par les valeurs des *goods* de ses clusters fils, alors l'enregistrement du *good* $I[E_i \cap E_{par(i)}]$ pour le cluster E_i est aussi une opération valide.

Pour conclure, du fait que l'enregistrement et l'utilisation des *goods* et *nogoods* sont valides, DPLL-TD est correct, complet et termine. □

Théorème 4 *DPLL-TD a une complexité en $O((|E| + m).2^{w+s})$ en temps et en $O(|E|.s.2^s)$ en espace pour une instance SAT possédant n variables et m clauses et une décomposition arborescente $(E, \mathcal{T})$ de largeur w et dont la taille de la plus grande intersection entre deux clusters est s.*

Preuve : Considérons un cluster E_j et une interprétation partielle I sur $E_j \cap E_{par(j)}$ et essayons d'étendre I sur E_j en supposant (dans un soucis de simplicité) que l'ordre des variables est statique. Si $E_j = E_1$ (le cluster racine de la décomposition arborescente), $I = \emptyset$ et DPLL-TD calculera au plus $2^{|E_1|+1}$ interprétations partielles pour étendre I. Sinon, il existe au plus $2^{|E_j|-|I|+1}$ interprétations partielles pour étendre I. Puisque, il y a au maximum $2^{|I|}$ interprétations partielles possibles de taille $|I|$ et que le sous-problème $\mathcal{F}_{E_j,I}$ n'est résolu qu'une fois grâce aux *goods* et *nogoods* enregistrés, le nombre maximum total d'interprétations partielles étudiées par DPLL-TD pour un cluster E_j est $\sum_{|I|=0}^{s} 2^{|E_j|-|I|+1}.2^{|I|} = s.2^{|E_j|+1}$. De plus, DPLL-TD va évaluer au maximum $O(2^{w+2})$ interprétations car le cluster E_j est indépendant de son cluster père du fait que les variables d'indépendance dans $\mathcal{X}_{ind} \cap E_j \cap E_{par(j)}$ sont déjà interprétées. Par ailleurs, pour chacune des interprétations partielles, DPLL-TD applique la propagation des clauses unitaires en $O(m + |\mathcal{N}|)$. Pour un séparateur donné de taille k (un séparateur est défini par l'intersection d'un cluster avec son père), le nombre d'interprétations partielles est borné par 2^{k+1}. Donc, le nombre total de *(no)goods* est donné par $|E|.2^{s+1}$. Finalement, l'enregistrement d'un *(no)good* peut se faire en $O(s)$ et le test de la condition de la ligne 12 en $O(s.2^{s+1})$. Ainsi, la complexité en temps de DPLL-TD est $O((m + |E|.2^{s+1}).2^{w+1} + s.2^{s+1}.2^{w+1})$, autrement dit $O((m + |E|).2^{w+s})$.

Concernant la complexité en espace, celle-ci dépend uniquement des *goods* et des *nogoods* enregistrés. Comme l'espace nécessaire pour la sauvegarde d'un *(no)good* est $O(s)$, la complexité en espace de DPLL-TD est $O(|E|.s.2^s)$. □

En d'autres termes, la complexité de DPLL-TD dépend des caractéristiques de la décomposition arborescente, à savoir sa largeur et la taille du plus grand cluster. Comparé à un algorithme DPLL classique, la complexité de ce dernier dépend du nombre de variables de l'instance traitée.

4 Résultats préliminaires

Cette section présente quelques détails sur l'implémentation de DPLL-TD et montre des résultats préliminaires obtenus sur des instances SAT issues des précédentes compétitions SAT, de 2002 à 2007 (`www.satcompetition.org`). Nous comparons DPLL-TD à 4 solveurs performants : Minisat[6], Rsat[13], Zchaff [16] et Satz [11] et toutes les instances ont été prétraitées avec SatElite [7]. L'algorithme DPLL-TD est codé sur la base de Satz . Ce choix est fait par notre bonne connaissance de son code source que nous avons toutefois remanié afin de prendre en compte les spécificités de notre approche, comme par exemple l'enregistrement de *nogoods* sous forme de clauses. Satz est donc utilisé comme algorithme d'énumération sur les clusters et les variables d'indépendance. Concernant le calcul de la décomposition arborescente, il repose sur une triangulation heuristique du graphe primal. Cette triangulation est effectuée grâce à la méthode Min-fill [15] qui s'avère être une des méthodes de triangulation heuristiques les plus robustes. Les expérimentations sont faites sous Linux sur un PC avec un processeur Pentium 4 à 3,2 GHz et une mémoire vive de 1 Go. Chaque instance est exécutée durant 600 secondes au maximum pour chacun des solveurs. Nous avons testé près de 1800 instances et nous présentons dans la table 1 une sélection des résultats les plus pertinents. La première colonne est le nom du benchmark et la deuxième le nombre d'instances que nous avons testé pour ce benchmark. Le reste de la table donne, pour chaque solveur, le nombre d'instances (#r) qu'il a pu résoudre ainsi que le somme des temps (t) en secondes nécessaires à les résoudre.

Au regard de la dernière colonne, qui donne le classement de DPLL-TD par rapport aux autres solveurs en termes de nombre d'instances résolues, nous pouvons annoncer que notre solveur est compétitif et améliore même les performances de deux solveurs majeurs, Minisat et Rsat, sur quelques instances. Egalement et comparativement à Satz, DPLL-TD améliore significativement les résultats de ce dernier. Ce qui peut nous laisser croire que l'implémentation de notre méthode sur la base d'un solveur plus récent et plus performant nous permettra encore d'obtenir des résultats de meilleure qualité.

Nous avons observé que les instances les plus facilement résolues par DPLL-TD sont majoritairement insatisfiables. Aussi, une première explication possible pour ce phénomène est que DPLL-TD, grâce à la décomposition arborescente qu'il exploite, est capable d'aborder la recherche par un cluster ayant peu ou pas de modèles. Si cette explication est certainement la bonne pour des instances de petite taille, nous avons pu observer que, dans la plupart des cas, DPLL-TD produit, mémorise et exploite un nombre conséquent de *goods* et de *nogoods* structurels. Aussi, les bons résultats observés s'explique également par l'apport des *goods* et des *nogoods* structurels, et plus précisément par les redondances dans l'espace de recherche qu'ils permettent d'éviter.

A la vue de ces premières expérimentations, il est possible d'annoncer que les résultats sont très encourageants et de suggérer que le comportement de DPLL-TD peut-être encore perfectionné par la prise en compte de différents paramètres, comme un meilleur choix de décomposition arborescente, un choix plus éclairé de cluster racine, ...

5 Travaux connexes

La structure des problèmes a été utilisée pour définir des heuristiques d'ordre sur les variables dans des méthodes de type backtrack depuis que Freuder [8] a présenté une fa-

		DPLL-TD		Minisat		Rsat		Zchaff		Satz		Classement de
Benchs.	# inst.	#r	t	#r	t	#r	t	#r	t	#r	t	DPLL-TD
linvrinv	8	3	1	3	1	3	2	3	9	3	1	1
mod2-3cage-unsat	23	6	2370	**18**	2579	1	540	0	-	0	-	2
mod2-rand3bip-sat	33	1	351	**17**	2534	6	1112	7	1646	8	1923	5
mod2-rand3bip-unsat	15	**11**	793	9	1062	6	872	6	1124	0	-	1
mod2c-3cage-unsat	6	**4**	115	3	563	0	-	0	-	0	-	1
mod2c-rand3bip-sat	33	0	-	17	2559	11	1410	**18**	3156	0	-	5
mod2c-rand3bip-unsat	15	**15**	676	10	1810	8	1337	6	1125	0	-	1
clqcolor	16	7	181	6	171	**9**	36	**9**	23	0	-	3
fclqcolor	16	9	136	9	169	9	22	9	11	0	-	1
fphp	42	16	617	**17**	165	16	49	16	28	**17**	555	3
php	42	16	123	6	258	16	134	12	231	**18**	721	2
sorge05	104	36	1752	**69**	5571	59	4347	54	2748	33	1316	4
driverLog	36	32	1	**36**	< 1	**36**	< 1	**36**	< 1	**36**	2	5
Ferry	36	20	496	**36**	6	**36**	4	**36**	22	17	299	4
rovers	22	22	3	22	< 1	22	< 1	22	< 1	22	1	1
satellite	20	20	22	20	< 1	20	< 1	20	< 1	20	259	1
difficult-contest05-jarvisalo	10	**3**	323	2	659	0	-	0	-	0	-	1
medium-contest05-jarvisalo	10	2	782	**9**	1955	1	470	3	850	3	497	4
spence-medium	10	3	220	**9**	1031	4	294	4	111	8	877	5
grieu	10	**5**	1222	1	71	**5**	347	3	525	**5**	378	1
industrial-jarvisalo	7	1	16	3	112	3	400	2	99	**4**	448	5

TABLE 1 – Les résultats de DPLL-TD et comparaisons

çon d'élaborer un tel ordre en calculant les composantes biconnexes du graphe de contraintes modélisant le problème traité. Par ailleurs, des travaux plus récents ont montré que l'usage de la décomposition arborescente peut guider avec succès l'heuristique de branchement utilisée par DPLL et nous présentons ici quelques-uns de ces travaux. L'heuristique de choix de variable présentée dans [9] utilise la méthode Dtree [4], qui fournit une décomposition sous forme d'un arbre binaire et statique, pour ordonner les variables par groupes. L'arbre de décomposition est construit avant la résolution par DPLL. De ce fait, l'ordre des groupes de variables ne change jamais au cours de l'exécution. Toutefois, l'ordre global établi n'est ni statique, ni dynamique, car si l'ordre des groupes est statique, l'ordre des variables à l'intérieur même de ces groupes reste dynamique. Dans [3], les auteurs présentent une heuristique pour créer dynamiquement un ordre sur des groupes de variables. Le bémol majeur de cette approche est qu'aucun résultat expérimental n'est présenté pour montrer si leur méthode est plus efficace qu'un ordre statique, comme c'est le cas dans le travail précédent. En outre, il n'existe aucune conclusion sur la façon d'organiser les sous-problèmes induits par la décomposition arborescente. Dans [2], les auteurs proposent une heuristique qui permet de résoudre d'abord les problèmes les plus contraints. Mais là encore, aucune évaluation expérimentale ne vient appuyer leur approche. Une heuristique d'ordre des variables basée sur une méthode récursive de type *min-cut bisection* de l'hypergraphe représentant l'instance a été proposée dans [1]. Cette approche ne nécessite pas la modification du solveur SAT utilisé. Cependant, la quasi-totalité des solveurs SAT modernes utilise un ordre de variables dynamique. L'ordre statique fourni par cette dernière approche sert notamment à remplacer, dans certaines situations, l'ordre dynamique établi par le solveur SAT. Dans [12], les auteurs décrivent une méthode de décomposition dynamique (sous forme d'un arbre) basée sur les séparateurs de l'hypergraphe représentant l'instance SAT traitée. L'usage de ces séparateurs dans l'établissement de l'ordre de choix de variables de branchement permet, sur quelques instances, d'accélérer les temps de réponse de certains solveurs SAT modernes. Comparée à Dtree, cette méthode ne nécessite pas de temps pour la construction complète de l'arbre de décomposition. Cette construction peut exiger, dans certains cas, plus de temps que la résolution effective de l'instance elle-même. Par ailleurs, dans [12], il est présenté un schéma de partitionnement contraint qui peut être exploité pour dériver un ordre sur les variables pour accélérer les solveurs SAT. Cette approche, qui est à l'opposé des schémas de décomposition arborescente qui cherchent à réduire la largeur de la décomposition arborescence, élabore un partitionnement sous contraintes de l'hypergraphe représentant l'instance SAT par l'analyse du nombre d'occurrences des variables dans les clauses de l'instance SAT et la connectivité existante entre ces clauses. Cette connectivité est exprimée par les variables communes à ces clauses.

Enfin, la méthode DPLL-TD a un lien de parenté évident avec la méthode BTD [10] proposée dans le cadre des problèmes de satisfaction de contraintes. Si ces deux méthodes reposent sur la notion de décomposition arborescente et son exploitation pour guider la recherche tout en mémorisant des *(no)goods* structurels, elles n'en demeurent pas moins différentes. D'une part, si les problèmes SAT et CSP sont voisins, leurs méthodes de résolution ne mettent pas en œuvre exactement les mêmes techniques. Par exemple, si toutes les variables doivent être instanciées pour produire une solution d'un CSP, il n'est pas nécessaire de toutes les instancier pour trouver un modèle d'une instance SAT. Cette simple différence rend nécessaire la définition d'un cadre formel spécifique au problème SAT pour l'al-

gorithme DPLL-TD (en particulier, avec l'introduction de la notion de variable d'indépendance). D'autre part, la notion de *good* diffère significativement entre DPLL-TD et BTD. De plus, les *goods* et *nogoods* de DPLL-TD peuvent être de taille variable en ne portant pas sur l'ensemble des variables du séparateur. Il en résulte ainsi des coupes potentiellement plus puissantes pour DPLL-TD que celles utilisées dans BTD.

6 Conclusion et discussions

Dans ce papier, nous avons proposé une nouvelle approche basée sur la décomposition arborescente pour résoudre le problème de satisfiabilité, notamment les instances structurées issues des problèmes du monde réel.

Le but de cette approche est de capturer la structure de l'instance SAT traitée. Cette structure est décrite sous la forme d'un hypergraphe qui est donc décomposé. La décomposition arborescente ainsi exploitée fournit un ordre sur les variables qui est utilisé pour guider une méthode énumérative de type DPLL. De plus et durant la recherche, des *goods* et *nogoods* structurels sont appris et utilisés pour élaguer l'espace de recherche. Cet apprentissage constitue l'une des originalités de notre approche, notamment en comparaison avec celles décrites dans la section précédente. Un autre apport de DPLL-TD est de fournir des bornes de complexité théorique, en temps et en espace, qui dépendent de la décomposition arborescente.

Contrairement au cadre des problèmes de satisfaction de contraintes (CSP), les approches basées sur la notion de décomposition arborescente sont peu exploitées dans le cadre SAT. Comme en témoigne la section précédente à nouveau, un nombre non négligeable de travaux s'inscrivant dans cette approche ne fournissent pas de résultats expérimentaux, ce qui rend difficilement mesurable leurs efficacités et contributions pratiques. Par conséquent, on pourrait dire que la littérature existante manque de maturité et de recul sur l'usage de la décomposition arborescente pour SAT. En effet, plusieurs aspects devraient être étudiés pour renforcer le travail que nous présentons ici. Parmi ces aspects, on peut citer la méthode de triangulation utilisée qui est nécessaire pour le calcul de la décomposition arborescente, le choix du cluster racine, l'élaboration de plusieurs heuristiques de choix du prochain cluster à traiter, ... Comme premières pistes, nous travaillerons sur l'usage de l'analyse des conflits afin d'apprendre de nouveaux nogoods à l'instar de ce qui fait dans les solveurs basés sur CDCL, l'usage des restarts tout en modifiant soit le cluster racine ou même la décomposition elle-même, à la base des informations apprises lors des précédentes exécutions. Par exemple, construire une nouvelle décomposition arborescente sur la base des variables les plus conflictuelles (contraintes). Cette information peut-être également exploitée à guider l'ordre de parcours des clusters en s'intéressant aux plus contraints d'abord.

Références

[1] F.A. Aloul, I.L. Markov, and K.A. Sakallah. MINCE : A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation. *Journal of Universal Computer Science (JUCS)*, 10(12) :1562–1596, 2004.

[2] E. Amir and S. Mcilraith. Solving satisfiability using decomposition and the most constrained subproblem. In *LICS workshop on Theory and Applications of Satisfiability Testing*, 2001.

[3] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, and Y. Zhu. Guiding sat diagnosis with tree decompositions. In *Proceedings of SAT 2004*, pages 315–329, 2004.

[4] A. Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of AAAI 2002*, pages 627–634, 2002.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of SAT 2003*, pages 402–518, 2003.

[7] N. Eén and N. Sörensson. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of SAT 2005*, pages 61–75, 2005.

[8] E.C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4) :755–761, 1985.

[9] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for sat. In *Proceedings of IJCAI 2003*, pages 1167–1172, 2003.

[10] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[11] C.M. Li and Anbulagan. Heuristic Based on Unit Propagation for Satisfiability. In *Proceedings of CP'97*, pages 342–356, Austria, 1997.

[12] W. Li and P. van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. In *Proceedings of ICTAI 2004*, pages 542–548, Washington, DC, USA, 2004. IEEE Computer Society.

[13] K. Pipatsrisawat and A. Darwiche. Rsat 2.0 : Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[14] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[15] U. Kjærulff. Triangulation of graphs : Algorithms giving small total state space. Technical report, University of Aalborg, 1990.

[16] L. Zhang and C.F. Madigan. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD 2001*, pages 279–285, 2001.