Méthodes avancées de raisonnement en logique propositionnelle : application aux réseaux métaboliques

Martin Morterol

NNT : 2016SACLS548

THÈSE DE DOCTORAT de L'UNIVERSITÉ PARIS-SACLAY
préparée à L'UNIVERSITÉ PARIS-SUD
Laboratoire de Recherche en Informatique
ÉCOLE DOCTORALE STIC n°: 580
Sciences et Technologies de l'Information et de la Communication
Spécialité : INFORMATIQUE

Méthodes avancées de raisonnement en logique propositionnelle : application aux réseaux métaboliques

Martin Morterol

Thèse présentée et soutenue à Orsay, le 13/12/2016

Composition du Jury

Président du jury : Sylvain Conchon, Professeur, Université Paris-Saclay
Directeur : Philippe Dague, Professeur, Université Paris-Saclay
Co-encadrante : Sabine Peres, Maître de conférences, Université Paris-Saclay
Co-encadrant (invité) : Laurent Simon, Professeur, Université Bordeaux
Rapporteur : Anne Siegel, Directeur de Recherche, CNRS, Université Rennes 1
Rapporteur : Marie Beurton-Aimar, Maître de conférences, Université Bordeaux
Examinateur : Celine Rouveirol, Professeur, Université Paris-Nord

Remerciement

Merci à Camille qui m'a rappelé d'écrire ces remerciements.

Je remercie vivement Anne Siegel et Marie Beurton-Aimar d'avoir accepté d'être rapporteurs et pour leur relecture attentive de ce manuscrit. Merci également à Sylvain Conchon et Celine Rouveirol d'avoir accepté de faire partie de mon jury.

Je tiens à remercier Philippe Dague, Sabine Peres et Laurent Simon pour leur encadrement et leurs conseils tout au long de cette thèse et plus particulièrement Philippe pour sa patience durant les nombreuses relectures.

Mes remerciements vont également à Dalek, Cabou et Hassan qui ont participé à créer une excellente ambiance (de travail ?) dans notre bureau durant ces trois ans. Je remercie aussi Lénaïc pour les nombreux déjeuners passés à parler de Méta-prog et de C++.

Je remercie également l'ensemble de l'équipe Lahdak pour toujours avoir été là quand j'ai eu besoin d'aide ainsi que Stéphanie Druetta pour son aide face à ~~l'adversité~~ adum et sa patience.

Et enfin je remercie tous les élèves qui m'ont fait aimer l'enseignement, que ça soit pour leur attention ou leur personnalité, avec une pensée particulière pour Bastien, Ripper et Jonathan.

Table des matières

Liste des tableaux

Table des figures

Introduction

0.1 Motivation et Objectifs

La compréhension du fonctionnement cellulaire est une étape primordiale dans différents domaines clés comme la médecine, la nutrition, l'écologie... Une des étapes de cette compréhension passe par l'étude et la modélisation des cellules, et comment le génome s'exprime dans celles-ci.

Nous nous intéressons plus particulièrement à ce qu'on appelle un réseau métabolique, représentant l'ensemble des réactions biochimiques. Ce réseau modélise synthétiquement l'action des enzymes (protéines catalytiques produites pas le génomes) et des métabolites dans une cellule ou dans une sous-partie d'une cellule.

Lorsqu'on ne dispose pas de donnée expérimentales suffisantes (telles que les paramètres cinétiques) l'étude des réseaux métaboliques se base sur l'énumération et l'étude des « voies métaboliques » à l'état stationnaire minimales (pour l'inclusion en termes des réactions impliquées) qu'on appelle EFMs pour « elementary flux modes ».

Cependant, sur des réseaux de grande taille (à l'échelle du génome) ce nombre de voies dépasse largement les capacités de calcul actuelles. De plus le calcul se base uniquement sur la stœchiométrie qui, bien que ce soit un critère nécessaire à respecter, n'est pas suffisant, cela implique qu'un grand nombre de ces voies n'ont pas de sens biologique.

Afin de réduire l'espace de solution il faut pouvoir énumérer les solutions respectant d'autres contraintes comme les contraintes thermodynamiques ou de régulation.

Bien que certaines approches commencent à intégrer ces contraintes elles ne peuvent pas, pour l'instant, intégrer de façon efficace toutes les contraintes de régulation. Nous allons présenter une approche novatrice ne se basant pas sur des calculs matriciels, comme les approches existantes, mais sur une modélisation du réseau métabolique sous forme de contraintes.

Nous présenterons un algorithme permettant de répondre à cette problématique en calculant les EFMs respectant des contraintes de régulation grâce à l'utilisation de la logique propositionnelle et d'un solveur SAT (pour « satisfiability », il s'agit d'un solveur cherchant si une formule en logique propositionnelle admet une solution ou non) ainsi qu'une version plus perfectionnée se basant sur l'utilisation d'un solveur SMT (pour « Satisfiability modulo theories », de même que le solveur SAT, le solveur SMT va chercher si une formule admet une solution ou non, cependant cette formule n'est plus forcément en logique propositionnelle mais est exprimée dans une « théorie », par exemple l'algèbre linéaire) nous permettant d'ajouter des contraintes linéaires.

Nous pouvons ainsi ajouter simplement et efficacement des contraintes de régulation tout en étant compatible avec les contraintes thermodynamiques. Ainsi nous pouvons d'une part énumérer un ensemble d'EFMs plus pertinent (et également plus petit) mais également énumérer des EFMs intervenant dans une cellule dans des cas spécifiques, tels que l'action d'un médicament ou l'expression d'une mutation.

Nous enrichirons également l'étude des voies métaboliques en proposant un nouvel outil, les $MCFM$s, pour « minimal constrained flux modes » représentant les voies métaboliques respectant des contraintes booléennes de façon minimale, adapté à l'énumération des voies métaboliques lorsque les contraintes ne peuvent pas être satisfaites par une voie minimale.

0.2 Plan de la thèse

Chapitre 1 État de l'art : Nous présenterons ici la problématique biologique puis les solutions existantes pour le calcul des EFMs et enfin les notions de logique liées au SAT solveur nécessaires à la compréhension de cette thèse.

Chapitre 2 Approche Sat : Dans ce chapitre nous présenterons les différentes étapes de la réalisation de notre première solution permettant de lister les EFMs satisfaisant une requête. Nous commencerons par trouver les voies dans un cas précis où la stœchiométrie n'a pas besoin d'être considérée puis nous ajouterons la stœchiométrie et verrons les ajouts que nous avons apportés à notre premier outil puis les performances de celui-ci sur des cas concrets.

Chapitre 3 Approche SMT : Dans ce chapitre nous verrons comment améliorer notre première solution en utilisant un SMT. Dans un premier temps nous présenterons la modélisation d'un réseau métabolique sous forme de contraintes, puis nous nous intéresserons à la minimisation des voies métaboliques et enfin nous nous comparerons à l'état de l'art sur un problème concret.

Chapitre 4 Les $MCFM$s : Dans ce chapitre nous présenterons un nouveau concept, proche de celui de EFMs, utile pour trouver des voies métaboliques lorsque l'on interroge un réseau métabolique de façon très précise. Après avoir présenté le principe des $MCFM$s nous détaillerons le calcul de ceux-ci et nous analyserons ce concept à travers diverses expérimentations.

Conclusion : Finalement, nous résumerons les méthodes que nous avons proposées au cours de cette thèse et nous proposerons des extensions possibles à notre travail.

Chapitre 1

État de l'art

1.1 Introduction

Le métabolisme cellulaire est l'ensemble des réactions qui permettent de produire toutes les molécules nécessaires au bon fonctionnement de la cellule, comme assurer ses besoins énergétiques et sa croissance. L'étude du métabolisme est une étape nécessaire pour mieux comprendre le fonctionnement de la cellule.

Les progrès actuels de la biologie et de la bio-informatique permettent de reconstruire le métabolisme cellulaire à l'échelle du génome (Thiele et al., 2013). Parallèlement l'augmentation de la puissance de calcul et les progrès en matière de traitement des données ont permis d'augmenter considérablement la taille des réseaux métaboliques qui sont abordables par les algorithmes de recherche d'EFMs. Ces évolutions ont toutefois plusieurs limitations :

D'une part, les plus gros réseaux restent encore hors d'atteinte et actuellement le génome humain reste difficilement exploitable. Il a été estimé que le réseau reconstitué du génome humain contenait 3311 réactions et pouvait contenir jusqu'à 10^{29} EFMs (Yeung et al., 2007).

D'autre part, le calcul des voies métaboliques ne prend pas en compte toutes les connaissances acquises par les biologistes. Il se fonde principalement sur des règles simples de

structure des réseaux métaboliques et souvent n'intègre pas, par exemple, des données telles que les contraintes thermodynamiques ou de régulation génique (les premiers travaux en ce sens sont très récents (Jungreuthmayer et al., 2012; Jungreuthmayer et al., 2013b)). Ce manque implique qu'un grand pourcentage des voies calculées sont des voies ne pouvant pas être mises en jeu dans une cellule.

Enfin la résolution de problèmes de plus en plus complexes entraîne des résultats de plus en plus volumineux. Ce qui rend l'énumération brute des EFMs de moins en moins pertinente : une liste de 10^{29} solutions, même si l'on était capable de la produire, serait difficilement exploitable par le biologiste sans moyens d'interrogation.

Le but de cette thèse est de proposer une approche innovante permettant d'interroger un réseau métabolique sans passer par une phase d'énumération afin d'obtenir des résultats plus pertinents pour les biologistes tout en augmentant la taille des réseaux interrogeables.
Nous allons dans un premier temps présenter la problématique biologique ainsi que les méthodes actuellement utilisées pour y répondre puis nous détaillerons les technologies SAT (« Satisfiability ») et SMT (« Satisfiability modulo theories ») que nous utiliserons dans notre approche.

1.2 Formalisation des réseaux métaboliques

L'étude du comportement d'une cellule passe par la compréhension de son fonctionnement interne qui est lui même le résultat de l'expression du génome sous les contraintes de l'environnement. En effet bien que le génome soit encodé dans l'ADN, son expression n'est pas directe. L'ADN va dans un premier temps être retranscrit en ARN qui sert d'intermédiaire aux gènes pour la synthèse des protéines, celles-ci vont intervenir dans toutes les fonctions de la cellule comme la respiration, la production d'énergie, la croissance... (Gerstl et al., 2015b; Gerstl et al., 2015a)

Dans cette partie nous présenterons les informations que nous jugeons nécessaires à la compréhension de la problématique biologique.

1.2.1 Métabolisme

Les enzymes

Le métabolisme est l'ensemble des réactions chimiques, transformations moléculaires et transferts d'énergie, qui opèrent dans une cellule ou être vivant. Ces réactions chimiques, pour qu'elles puissent avoir lieu, dans une échelle de temps compatible avec la vie d'une cellule, doivent être en présence de « catalyseurs biologiques » : les enzymes. On parle de réactions enzymatiques.

Definition 1.1 (Réaction enzymatique) *Une réaction enzymatique est la fixation d'une enzyme sur une molécule appelée substrat, qui forme un complexe enzyme-substrat, suivie de la formation d'un produit et de la libération de l'enzyme.*

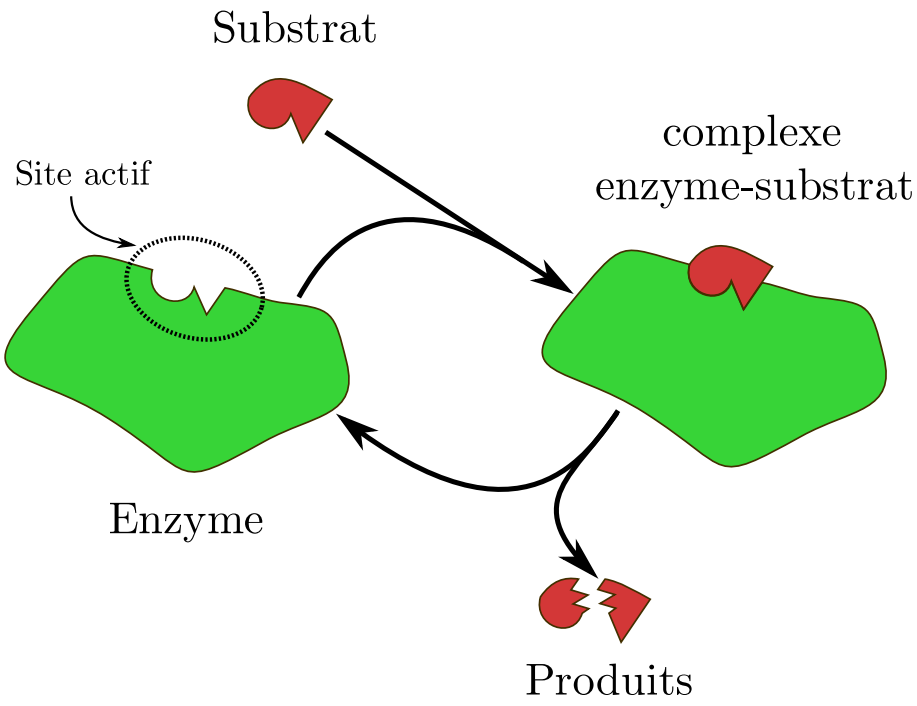


FIGURE 1.1: Décomposition d'un substrat grâce à une enzyme

Les enzymes agissent en faible concentration et se retrouvent intactes après la réaction. Le fonctionnement d'une enzyme est lié à son « site actif », un élément particulier de sa structure qui reconnaît une ou plusieurs molécules selon un principe de complémentarité de type clé-serrure

(Cornish-Bowden, 2012). Ces réactions chimiques peuvent donc être modélisées par la transformation, par des enzymes, de molécules nommées « métabolites » en d'autres métabolites. La figure 1.1 présente schématiquement ce fonctionnement.

La transformation effectuée par une enzyme peut, en fonction de contraintes thermodynamiques et de concentrations, agir de façon réversible. On parlera donc naturellement de réactions réversibles ou irréversibles.

Les transporteurs

La majorité des réactions enzymatiques se déroule à l'intérieur de la cellule ou dans un compartiment spécifique, comme par exemple la mitochondrie, mais il ne s'agit pas des seules réactions. Pour assurer son fonctionnement la cellule doit pouvoir interagir avec son environnement, cela se fait principalement par des échanges de molécules. Cette fonction est à la charge des « transporteurs ». Les transporteurs sont des protéines qui font passer des molécules d'un côté d'une membrane à l'autre. Ce transfert se fait naturellement dans un sens, par exemple de l'intérieur vers l'extérieur, et peut être inversé mais cela nécessite de l'énergie.

1.2.2 Les réseaux métaboliques

Un réseau métabolique est un ensemble de réactions enzymatiques et de procédures de transports, il représente aussi les relations et les contraintes entre ces ensembles. Nous allons présenter les principales modélisations des réseaux métaboliques.

L'hypergraphe orienté

Une des représentations d'un réseau métabolique les plus intuitives est celle de l'hypergraphe orienté.

Definition 1.2 (Hypergraphe orienté) *Un hypergraphe orienté d'ordre n, avec $n \in \mathbb{N}^*$, est un couple (V, E) où V est un ensemble de n sommets et E est l'ensemble des hyper-*

arcs, où chaque hyperarc est un couple de sous-ensembles non vides de sommets, de la forme $(\{x_{i_1}, \ldots, x_{i_p}\} \rightarrow \{y_{j_1}, \ldots, y_{j_q}\})$ *avec* $1 \leq p \leq n$, $1 \leq q \leq n$ *et* $x_{i_k}, y_{j_k} \in V$.

Propriété 1.1 (Matrice d'incidence d'un hypergraphe) *Un hypergraphe orienté sans boucle peut être représenté sous la forme d'une matrice à coefficients -1, 0 ou +1, dont chaque colonne a au moins un coefficient égal à -1 et un coefficient égal à +1 et chaque ligne au moins un coefficient non nul. En effet, un tel hypergraphe* $H = (V, E)$ *d'ordre* n *ayant* m *hyperarcs correspond de manière univoque à la matrice* $A(n, m)$ *telle que :*

$$A_{i,j} = \begin{cases} +1 & \textit{si } v_i \textit{ est un sommet origine de } E_j \\ -1 & \textit{si } v_i \textit{ est un sommet extrémité de } E_j \\ 0 & \textit{sinon} \end{cases}$$

Un réseau métabolique peut être représenté par un hypergraphe orienté. Pour cela, les métabolites doivent correspondre aux sommets du graphe et les enzymes et transporteurs à ses hyper-arcs. Certains auteurs (Planes and Beasley, 2009) utilisent des graphes bipartis pour modéliser les réseaux métaboliques. Les enzymes et les métabolites sont représentés comme sommets du graphe. Notons que nous n'utiliserons pas ce type de représentation dans la suite de nos travaux.

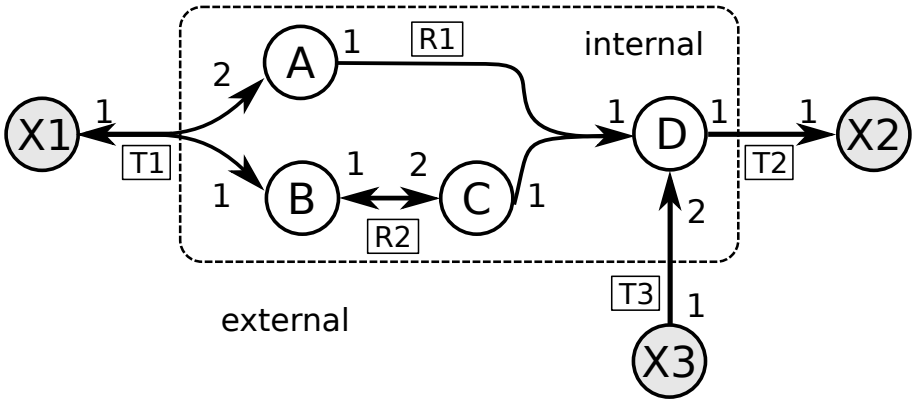


FIGURE 1.2: Représentation d'un réseau métabolique sous forme d'hypergraphe orienté et pondéré

La figure 1.2 est un exemple simple de représentation de réseau métabolique. X1, X2 et X3 sont des métabolites externes. A, B, C et D sont des métabolites internes. Les réactions T1, T2 et T3 sont des réactions d'échange et les réactions T1 et R2 sont réversibles.

Les pondérations indiquées aux extrémités d'un hyper-arc correspondent aux nombres de métabolites produits et consommés par la réaction représentée par cet hyper-arc c'est-à-dire aux coefficients stœchiométries.
Dans l'exemple :

— T1 consomme 1 X1 pour produire 2 A et 1 B ;

— T2 consomme 1 D pour produire 1 X2 ;

— T3 consomme 1 X3 pour produire 2 D ;

— R1 consomme 1 A et 1 C pour produire 1 D ;

— R2 consomme 1 B pour produire 2 C.

Les voies métaboliques

Une voie métabolique à l'état stationnaire (que nous appellerons par la suite simplement « voie métabolique ») traduit la transformation successive des métabolites par des enzymes de la cellule. Nous pouvons voir une voie métabolique, dans l'hypergraphe représentant le réseau métabolique, comme :

— un chemin pondéré tel que tous les nœuds initiaux et terminaux soient des métabolites externes ;

— un cycle ;

— la superposition de tels chemins.

Propriété 1.2 (Voie métabolique) *Une voie métabolique est une combinaison linéaire à coefficients réels (en fait rationnels) de réactions et se représente comme un vecteur de $\mathbb{R}^r$, où r est le nombre de réactions dans le réseau.*

Par exemple les voies $v_1 : 2\ T2 + T3$ et $v_2 : T1 + 2\ R1 + R2 + 2\ T2$ sont représentées par les vecteurs $(0, 0, 0, 2, 1)^t$ et $(1, 2, 1, 2, 0)^t$ dans la base {T1,R1,R2,T2,T3}.

Le support d'une voie métabolique est l'ensemble des enzymes participant à cette voie, c'est-à-dire ayant un coefficient non nul dans le vecteur représentant la voie. Le support d'une voie ne fait pas intervenir la stœchiométrie, il peut donc se représenter par un vecteur binaire.

Propriété 1.3 (Support d'une voie métabolique) *Le support d'une voie métabolique représente les réactions actives dans une voie métabolique. Il se modélise comme un vecteur de* $(\mathbb{Z}/2\mathbb{Z})^r$*, où* r *est le nombre de réactions dans le réseau.*

Par exemple, $Supp(v_1) = \{T2, T3\}$ et $Supp(v_2) = \{T1, R1, R2, T2\}$ sont, respectivement, représentés par les vecteurs $(0, 0, 0, 1, 1)^t$ et $(1, 1, 1, 1, 0)^t$.

La matrice de stœchiométrie

Il est possible de représenter un réseau métabolique grâce à sa « matrice de stœchiométrie ». Cette représentation est la plus communément admise car elle permet l'utilisation de l'algèbre linéaire, sur laquelle reposent les principaux outils de calcul des voies métaboliques. La matrice de stœchiométrie est un enrichissement de la matrice d'incidence précédente par les coefficients de pondération. Cette matrice se construit avec, en lignes, les métabolites internes et, en colonnes, les enzymes et transporteurs. Chaque case contient la production, si la valeur est positive, ou consommation, si la valeur est négative, d'un métabolite par l'enzyme. Ce qui nous donne la définition suivante :

Definition 1.3 (Matrice de stœchiométrie) *Soit un réseau métabolique contenant* m *métabolites internes et* r *réactions (enzymes ou transporteurs). La matrice de stœchiométrie* $N = (n_{i,j})$ *avec* $1 \leq i \leq m$, $1 \leq j \leq r$ *de ce réseau est définie par :*

- $n_{i,j} = a$ *si la réaction* j *produit le métabolite* i *avec la stœchiométrie* $a \in \mathbb{Q}_+^*$
- $n_{i,j} = -a$ *si la réaction* j *consomme le métabolite* i *avec la stœchiométrie* $a \in \mathbb{Q}_+^*$

– $n_{i,j} = 0$ *si le métabolite* i *n'intervient pas dans la réaction* j

La figure 1.2 devient donc :

métabolites/enzymes	T1	R1	R2	T2	T3
A	2	-1	0	0	0
B	1	0	-1	0	0
C	0	-1	2	0	0
D	0	1	0	-1	2

FIGURE 1.3: Représentation d'un réseau métabolique sous forme de matrice de stœchiométrie

Le principal problème de la représentation matricielle est que l'on perd l'information de la réversibilité d'une réaction. Pour pallier ce problème nous pouvons dédoubler les réactions réversibles en deux réactions irréversibles. Nous utiliserons la représentation avec les réactions réversibles dédoublées dans notre approche.

Une fois les réactions réversibles dédoublées la figure 1.2 devient :

$$\begin{array}{c} \text{métabolites/enzymes} \\ \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \end{array} \begin{array}{c} \begin{array}{ccccccc} \text{T1} & \text{T1}_{rev} & \text{R1} & \text{R2} & \text{R2}_{rev} & \text{T2} & \text{T3} \end{array} \\ \begin{pmatrix} 2 & -2 & -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 2 \end{pmatrix} \end{array} = N$$

FIGURE 1.4: Représentation d'un réseau métabolique sous forme de matrice de stœchiométrie une fois les réactions réversibles dédoublées

1.2.3 Modélisation des réseaux métaboliques et des EFMs

Modélisation dynamique

Lors de l'activité d'une cellule, celle-ci ne réagit pas toujours de la même façon. Les contraintes extérieures telles que la température, l'acidité du milieu, ou simplement la présence ou l'absence de métabolites externes influencent l'activité cellulaire.

La variation dynamique des concentrations des différents objets d'un réseau métabolique en fonction du temps peut se modéliser grâce à des équations différentielles. Bien que nous ne l'utilisions pas dans nos travaux, il s'agit d'un moyen courant de modélisation.

L'état d'un réseau métabolique est modélisé par les quantités de métabolites internes à un instant donné. Une variable $x_i(t)$ désigne la concentration d'un métabolite interne à un instant t. Un réseau métabolique est constitué de métabolites $x_1, ..., x_m$. Le système à un instant t est constitué d'un ensemble fini de variables $x_i(t)$ et l'état du système peut être représenté par le vecteur :

$$\vec{x}(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \\ ... \\ x_m(t) \end{pmatrix}$$

On considère que l'évolution du système différentiel ne dépend que son état à un moment donné. L'évolution des concentrations désignées par le vecteur $\vec{x}(t)$ est représentable par un système différentiel du premier ordre :

$$\forall t,\ \frac{d\vec{x}(t)}{dt} = F(\vec{x}(t))$$

où F est un champ de vecteurs sur $\mathbb{R}^m$. (Peres, 2005)

L'hypothèse d'état stationnaire

L'état stationnaire correspond à un laps de temps durant lequel on suppose que la variation des concentrations des métabolites est nulle. Le vecteur de concentrations de métabolites internes X_0 dans un système à l'état stationnaire est tel que le système ci-dessous a pour unique solution $X(t) = X_0$:

$$\begin{cases} \frac{dX(t)}{dt} = N.V(X(t)) \\ X(0) = X_0 \end{cases} \tag{1.1}$$

Ainsi, pour tout vecteur de concentration de métabolites $X(t)$ à l'état stationnaire, on a $\frac{dX(t)}{dt} = 0$, donc il vérifie $N.V(X_0) = 0$. Cela signifie que pour chaque métabolite x_i, sa vitesse de production est égale à sa vitesse de consommation.

Une voie à l'état stationnaire est un vecteur $V(X_0)$. Ainsi, toutes les voies à l'état stationnaire se situent dans le noyau de N. Nous résolvons analytiquement l'équation :

$$N.V = 0. \tag{1.2}$$

Tous les vecteurs des voies à l'état stationnaire d'un réseau, représenté par une matrice N, peuvent être déterminés par combinaison linéaire des vecteurs d'une base du noyau de N. Remarquons que certains vecteurs de la base peuvent utiliser une réaction irréversible dans le sens opposé. Ainsi le noyau de N est un sur-ensemble de l'espace de solution des voies.

Dans la représentation matricielle, une voie métabolique est représentée par un vecteur de réactions. Chaque indice de ce vecteur indique le nombre de fois que la réaction intervient dans la voie.

Exemple : On vérifie que les deux voies précédentes v_1 et v_2 sont dans le noyau de N et par conséquent sont des voies à l'état stationnaire.

$$N.v_1 = \begin{pmatrix} 2 & -2 & -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \\ 0 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$N.v_2 = \begin{pmatrix} 2 & -2 & -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Soit une voie utilisant toutes les réactions sauf la 5 et une voie composée uniquement des réactions 4 et 5. Dans la première voie, les réactions 1 et 3 produisent deux fois chaque métabolite A et C. Afin que les concentrations de ces métabolites restent constantes, et donc que le vecteur de la voie métabolique soit dans le noyau de N, les réactions 2 et 4 doivent avoir lieu deux fois. Toutes les voies métaboliques valides sont dans le noyau de N, mais tous les vecteurs du noyau ne sont pas des voies métaboliques valides. Cela est dû au fait que le calcul du noyau ne prend pas en compte l'irréversibilité de certaines réactions. Par exemple le vecteur $(0\ 0\ 0\ -2\ -1)^t$ est dans le noyau mais n'est pas une voie métabolique.

Dans le reste de ce chapitre nous sous-entendrons le fait qu'une voie est à l'état stationnaire et nous parlerons simplement de voie métabolique.

Les modes élémentaires de flux

L'ensemble des voies métaboliques peut être engendré par combinaisons linéaires à coefficients positifs à partir d'un ensemble de voies métaboliques non décomposables. (Schuster and Hilgetag, 1994) appelle les éléments de cet ensemble des « EFM » pour « elementary flux mode ». Ils forment un ensemble générateur des voies métaboliques.

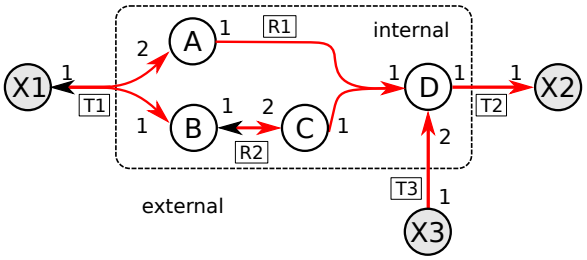
Definition 1.4 (EFM) *Un EFM est une voie (donc pouvant opérer à l'état stationnaire en tenant compte des réactions irréversibles) minimale, en terme d'inclusion des supports.*

Un vecteur $e = (e_1, ..., e_r)^t \in \mathbb{R}^r$ est un *mode élémentaire* (EFM) s'il vérifie les conditions suivantes :

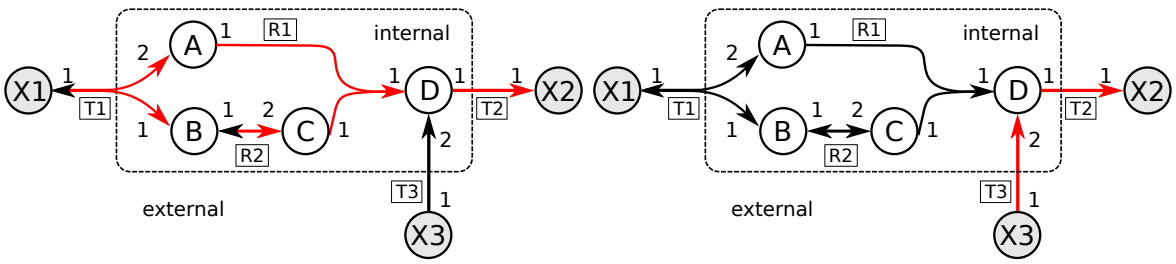
1. *état stationnaire* : $Ne = 0$.
2. *faisabilité* : Pour chaque indice j d'une réaction irréversible $e_j \geq 0$.
3. *minimalité* : Soit $supp(v) = \{j \in \mathbb{N} : v_j \neq 0\}$. Pour chaque voie e' vérifiant 1 et 2, $supp(e') \subseteq supp(e) \Rightarrow \exists\ \alpha \in \mathbb{R}$ tel que $e' = \alpha e$.

Le dernier point assure la minimalité (au sens de l'inclusion des supports) des EFMs, qui se traduit par le fait que le support d'un EFM ne peut pas être inclus dans le support d'un autre EFM. On parle d'indépendance génétique (Schuster et al., 2000). Dans le cas où toutes les réactions sont irréversibles (ce que nous supposerons dans notre approche, après dédoublement des réactions réversibles) la minimalité, au sens de l'inclusion des supports, équivaut à la non décomposabilité. Ainsi toute voie métabolique v peut s'exprimer sous la forme d'une somme d'EFMs $e_1, \ldots, e_n$.

Ainsi par exemple la voie métabolique v ={T1, R2, R3, 2 T3, 2 T2} :



Se décompose en les EFMs e_1 ={T1,R2,R3,T2} et e_2 ={T3,T2} :



1.2.4 Principales méthodes d'analyse des voies métaboliques à l'état stationnaire

L'analyse des voies métaboliques comporte deux branches principales, les méthodes exactes et les méthodes biaisées. Le schéma 1.5 résume les principales méthodes de modélisation à base de contraintes pouvant utiliser la matrice de stœchiométrie.

L'approche que l'on présentera dans cette thèse se situe dans la branche « Unbiased » puisqu'il s'agit d'une énumération d'EFMs.

Nous allons maintenant présenter rapidement une méthode de la branche « Biased », permettant de calculer une voie optimale : FBA.

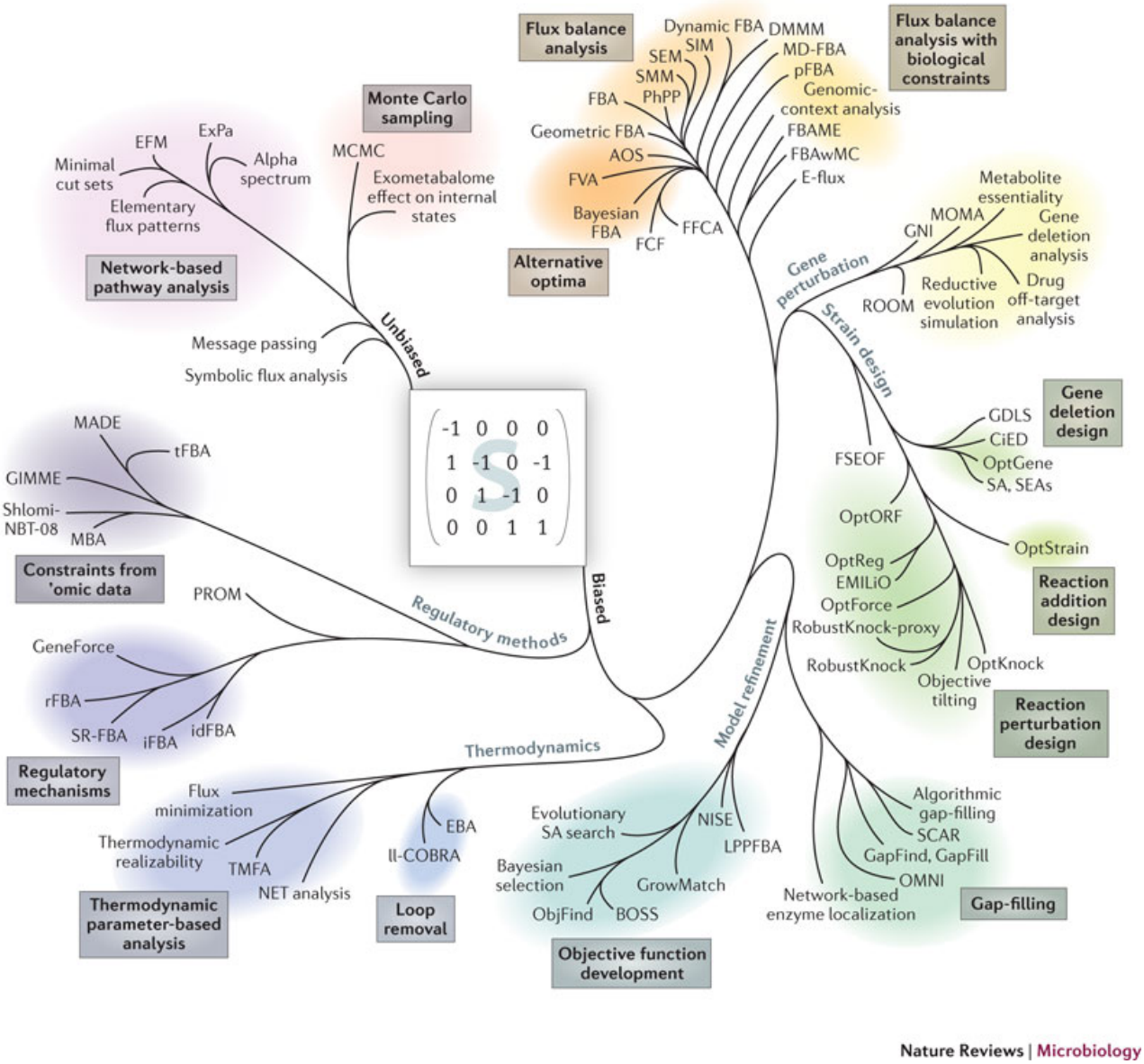


FIGURE 1.5: Phylogénie des méthodes de modélisation utilisant des contraintes, issue de l'article (Lewis et al., 2012)

Flux Balance Analysis : FBA

La méthode FBA cherche à optimiser une fonction objectif qui peut être la production/consommation d'un ou plusieurs métabolites. Il s'agit de la méthode la plus couramment utilisée pour les gros réseaux car elle permet de trouver un résultat là ou l'énumération des EFMs se heurte à l'explosion combinatoire. Le calcul se fait en plusieurs étapes. Premièrement, pour que les réactions soient à l'état stationnaire, on définit un vecteur V tel que $NV = 0$. Illustré par la figure 1.6.

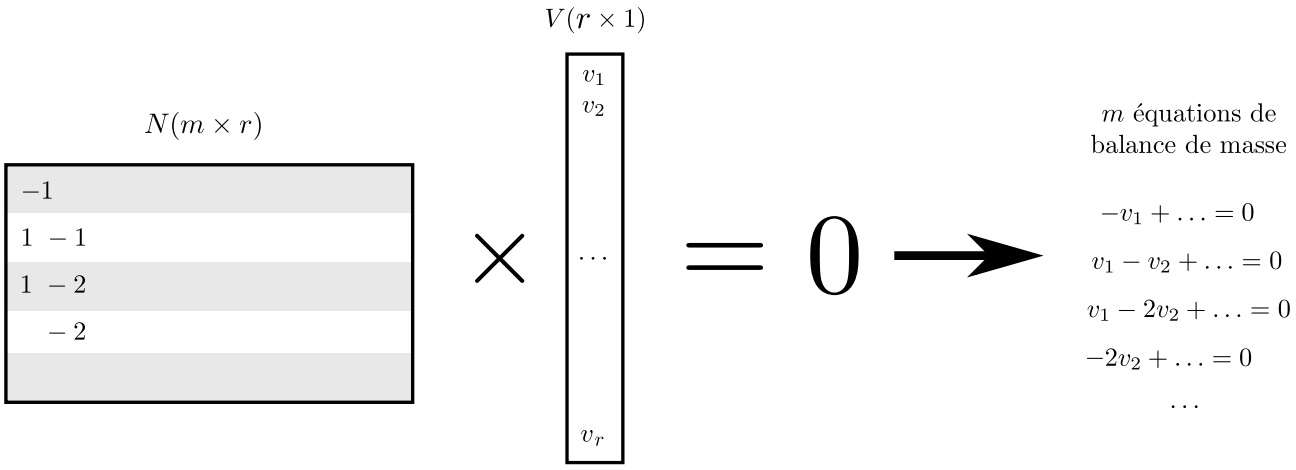


FIGURE 1.6: Application des contraintes de balance de masse, issue de l'article (Orth et al., 2010)

Ensuite on pondère les réactions que l'on souhaite maximiser/minimiser grâce à un vecteur C de taille r, ces réactions sont mises à 1 dans C, les autres sont mises à 0. Illustré par la figure 1.7.

La dernière étape consiste à utiliser des algorithmes d'optimisation linéaire, comme le simplexe, pour optimiser le résultat de C^tV. Illustré par la figure 1.8.

À l'opposé de l'approche FBA qui permet de trouver une voie métabolique unique optimisant une fonction objectif, nous allons maintenant présenter l'énumération des EFMs qui permet de caractériser l'espace de solution.

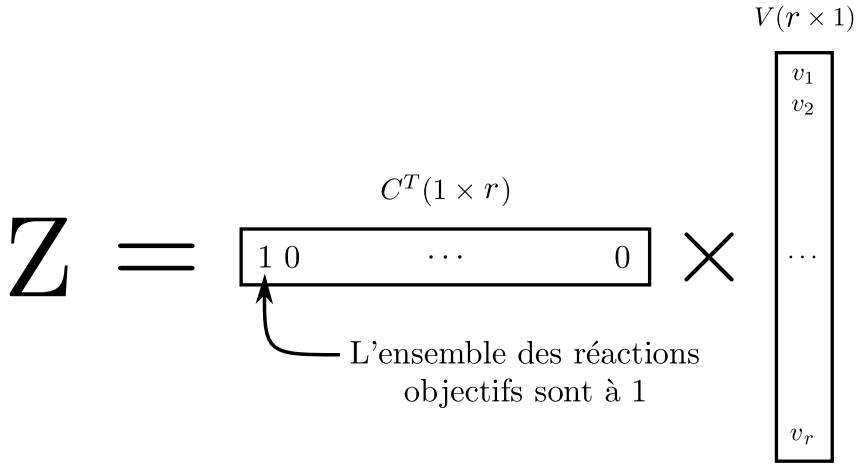


FIGURE 1.7: Définition de la fonction objectif, issue de l'article (Orth et al., 2010)

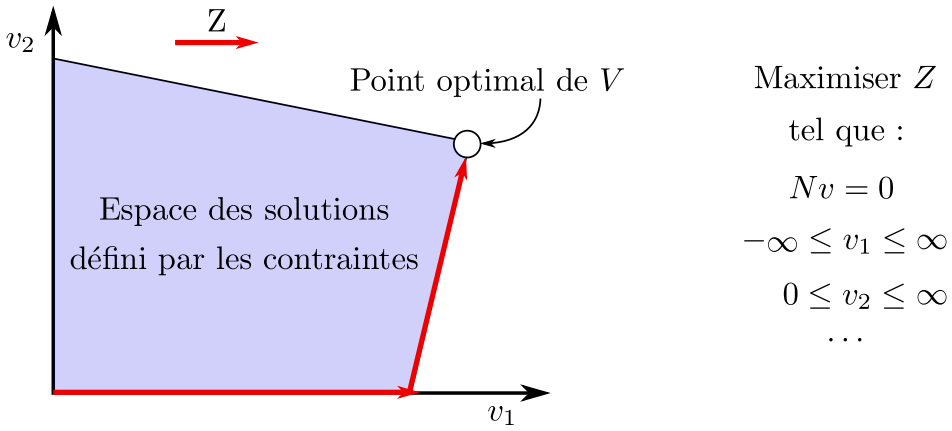


FIGURE 1.8: Optimisation de Z grâce à de l'optimisation linéaire, issue de l'article (Orth et al., 2010)

1.2.5 L'énumération des EFMs

Cette approche ne permet pas de trouver directement la voie maximisant la production d'un métabolite, elle permet en revanche d'étudier la structure du réseau et d'en déduire des informations comme sa robustesse (Stelling et al., 2002) ou sa fragilité (Klamt and Gilles, 2004). La production d'un métabolite est dite robuste si elle ne dépend pas que d'un faible nombre de voies métaboliques et si ces voies n'utilisent pas toutes une même réaction (si c'est le cas, l'inhibition de cette réaction conduit à l'arrêt de la production du métabolite).

Notre approche étant une méthode énumération d'EFMs permettant d'énumérer uniquement les EFMs satisfaisant certaines contraintes, nous allons détailler le fonctionnement des algorithmes classiques pour cette tâche dans la section suivante.

1.3 Calcul des EFMs

1.3.1 Les premiers algorithmes des EFMs

La modélisation matricielle des réseaux métaboliques permet d'utiliser des algorithmes d'algèbre linéaire pour calculer le noyau de la matrice de stœchiométrie auquel appartiennent les EFMs du réseau. Il faut ensuite intégrer les contraintes de positivité des flux associés aux réactions irréversibles et déterminer les éléments minimaux de cet espace de solutions. Cependant la réversibilité de certaines voies n'est pas prise en compte dans ce calcul. Une solution peut être de décomposer chaque réaction réversible en deux réactions irréversibles mais il faut s'assurer que ces deux réactions n'interviennent pas simultanément dans une voie (ou filtrer après coup ces cycles) (Schuster and Hilgetag, 1994) propose une solution plus élégante qui permet de calculer les EFMs en utilisant directement les réactions réversibles. Pour cela ils utilisent une version modifiée du pivot de Gauss qui se présente comme suit.
On sépare les réactions réversibles des réactions irréversibles. Puis il faut dérouler l'algorithme

de Gauss (Beezer, 2008) avec une restriction : on ne peut soustraire les réactions irréversibles (contrainte de positivité). Reprenons l'exemple de la Fig. 1.2 :

Matrice identité					A	B	C	D	réaction	
1	0	0	0	0	2	1	0	0	T1	rev
0	1	0	0	0	0	-1	2	0	R2	rev
0	0	1	0	0	-1	0	-1	1	R1	irr
0	0	0	1	0	0	0	0	-1	T2	irr
0	0	0	0	1	0	0	0	2	T3	irr

Nous avons mis les réactions réversibles en haut et les irréversibles en bas de la matrice (qui est la transposée de la matrice de stœchiométrie). La matrice identité sert à conserver les combinaisons de lignes effectuées.

Nous souhaitons mettre des 0 dans toutes les colonnes des métabolites. Nous commençons par la première. Pour cela nous cherchons toutes les combinaisons possibles entre les lignes ayant une valeur non nulle dans la colonne A. Bien que l'explosion combinatoire ne soit pas visible dans notre petit exemple, il s'agit de la raison pour laquelle cet algorithme ne permet pas de traiter de gros réseaux. Si deux lignes sont équivalentes à un facteur près nous pouvons en supprimer une.

					A	B	C	D	
0	1	0	0	0	0	-1	2	0	
1	0	2	0	0	0	1	-2	2	= l3*2+l1
0	0	0	1	0	0	0	0	-1	
0	0	0	0	1	0	0	0	2	

Nous faisons de même avec la colonne B.

					A	B	C	D	
1	1	2	0	0	0	0	0	2	= l2+l1
0	0	0	1	0	0	0	0	-1	
0	0	0	0	1	0	0	0	2	

Il ne nous reste que des enzymes irréversibles. Nous n'avons donc que deux combinaisons différentes.

					A	B	C	D	
1	1	2	2	0	0	0	0	0	= l2 * 2 + l1
0	0	0	2	1	0	0	0	0	= l2 * 2 + l3

Nous arrivons bien sur les deux EFMs trouvés dans la partie précédente. Puisqu'à chaque étape de l'algorithme, nous utilisons toutes les combinaisons possibles pour annuler une colonne, lors de la dernière étape, nous obtenons l'ensemble des EFMs.

1.3.2 Méthode de la double description

Une autre méthode qui a rapidement été appliquée au calcul des EFMs a été empruntée au domaine de la programmation linéaire. Il s'agit de la double description (Motzkin et al., 1953) qui s'appuie sur le théorème suivant :

Théorème 1.4 (Théorème de Minkowski pour les cônes polyédraux) *Soit P un cône polyédral dans $\mathbb{R}^d$ défini par une matrice réelle de représentation $A(m \times d)$: $P = P(A) = \{x \in \mathbb{R}^d | Ax \geq 0\}$. Il existe une matrice réelle $R(d \times n)$ qui engendre P : $P = \{x \in \mathbb{R}^d | x = Ry$ avec $y \geq 0\}$.*

On appelle la paire (A, R) une « paire de double description » ou « DD paire ». La réciproque de ce théorème est également vraie (théorème de Weyl).

La non trivialité de ce théorème vient du fait que la matrice R a un nombre fini n de colonnes. Cela implique que l'ensemble des vecteurs du cône s'exprime comme l'ensemble des combinaisons linéaires à coefficients non négatifs de ces n vecteurs colonnes.

Si on ajoute comme condition que l'on souhaite que la matrice R soit minimale (ne possède pas de sous-matrice propre engendrant P), ses vecteurs colonnes forment un ensemble minimal de vecteurs générateurs de P. Un tel ensemble est unique (à un facteur positif près pour

chaque vecteur) si le cône est pointé. Les vecteurs colonnes de R sont aussi les arêtes (« extreme rays ») du cône, c'est-à-dire ses rayons indécomposables. Ce concept, appliqué à une matrice de stœchiométrie (dans laquelle on suppose toutes les réactions mises sous forme irréversible), correspond à celui d'EFM. La minimalité de l'EFM en terme d'inclusion ensembliste de réactions équivaut en effet alors à sa non décomposabilité en termes d'autres EFMs.

Nous avons donc une matrice de stœchiométrie qui, avec les contraintes de positivité des flux dans les réactions supposées irréversibles, constitue la matrice A dans la double description et nous souhaitons obtenir la matrice R dont les colonnes nous fourniront les EFMs. Il n'existe pas d'algorithme général connu qui soit polynomial pour calculer R (Terzer, 2009) .

La méthode classique pour calculer R consiste à partir d'une DD paire (A_K, R) telle que A_K soit la sous-matrice de A composée des lignes indexées par un sous-ensemble K d'indices des lignes de A, à ajouter une à une à A_K les lignes restantes de A et à recalculer R à chaque itération, jusqu'à ce que toutes les lignes soient dans A_K.

Il nous faut maintenant deux choses :

- La paire initiale (A_K, R). Nous avons deux solutions, soit partir d'une paire avec $|K| = 1$ (donc d'une seule ligne), soit, mieux, partir d'une sous-matrice maximale A_K de A dont toutes les lignes sont linéairement indépendantes, ce qui peut se trouver via un pivot de Gauss et de R l'inverse de A_K
- Une méthode pour passer de (A_K, R) à (A_{K+i}, R'). Géométriquement, ajouter une nouvelle ligne A_i correspond à couper le cône $P(A_K)$ par un hyperplan (dont l'équation est donnée par cette ligne) et à en garder la partie positive. Trouver la matrice R' correspond à déterminer les nouvelles arêtes du cône obtenu. Ces nouvelles arêtes étant sur des anciennes faces du cône peuvent s'exprimer par la combinaison de deux anciennes arêtes (une positive et une négative vis-à-vis de l'hyperplan).

Trouver les nouvelles arêtes correspond donc à déterminer les combinaisons de deux anciennes arêtes telles que :

$$A_i(a\ ancienneAretePositive + b\ ancienneAreteNegative) = 0$$

soit (à un facteur près) :

$$a = -A_i ancienneAreteNegative \text{ et } b = A_i ancienneAretePositive$$

La découverte des nouvelles arêtes est illustrée par la la figure 1.9

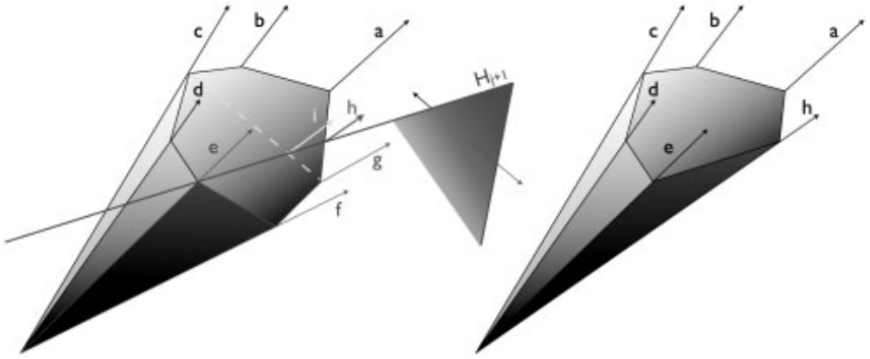


FIGURE 1.9: Représentation de l'ajout d'une contrainte. (Terzer and Stelling, 2008)

Le problème de cette méthode est que l'on doit tester toutes les combinaisons d'arêtes de part et d'autre de l'hyperplan. Cependant, une simple solution à ce problème consiste à ne tester que les paires adjacentes d'arêtes. En effet on sait que la nouvelle arête se trouve sur une face existante et que les bords de cette face sont séparés par l'hyperplan créé par la nouvelle ligne. La combinaison peut donc se limiter à toutes les paires adjacentes dont les éléments sont dans un demi-espace différent par rapport à l'hyperplan. EFMtool (Terzer and Stelling, 2008) propose une solution permettant d'optimiser la recherche en utilisant des arbres binaires pour représenter la liste courante des arêtes adjacentes et sa mise à jour lors de l'itération. De plus ils utilisent une variante de la méthode de double description introduite par (Wagner, 2004) qui a la particularité de partir du noyau de la matrice A et d'ajouter les contraintes de positivité au fur et à mesure. Cela permet de simplifier les calculs puisque les contraintes de positivité sont représentées par des vecteurs composés de zéros et d'un un. La combinaison de ces deux techniques est utilisée par les principaux outils de calcul d'EFMs tels que METATOOL (Von Kamp and Schuster, 2006), CellNetAnalyzer (Klamt et al., 2007) et EFMtool (Terzer and Stelling, 2009).

Exemple du fonctionnement de l'algorithme de base

Reprenons l'exemple de la Fig. 1.2. Les réactions 1 et 3 sont réversibles, nous les divisons en deux réactions irréversibles.

Une fois cette décomposition faite, la matrice A devient :

métabolites/enzymes	T1	$T1_{rev}$	R1	R2	$R2_{rev}$	T2	T3
A	2	-2	-1	0	0	0	0
B	1	-1	0	-1	1	0	0
C	0	0	-1	2	-2	0	0
D	0	0	1	0	0	-1	2

Les contraintes de stœchiométrie s'expriment par des égalités de la forme $A_i x = 0$, que l'on réécrit comme deux inégalités $A_i x \geq 0$ et $-A_i x \geq 0$ si l'on veut respecter la forme générale de la définition du cône $P(A) = \{x \in \mathbb{R}^d | Ax \geq 0\}$, ce qui revient à dupliquer les quatre lignes de la matrice par leurs opposées. De plus les contraintes de positivité des flux dans les réactions qui ont toutes été rendues irréversibles s'expriment en ajoutant une matrice identité. La matrice A devient donc :

1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1
2	-2	-1	0	0	0	0
1	-1	0	-1	1	0	0
0	0	-1	2	-2	0	0
0	0	1	0	0	-1	2
-2	2	1	0	0	0	0
-1	1	0	1	-1	0	0
0	0	1	-2	2	0	0
0	0	-1	0	0	1	-2

Comme configuration initiale de l'algorithme, on part d'une sous matrice maximale A_K de A dont les lignes sont linéairement indépendantes (donc $|K| = 7$ puisque A est de rang maximal) et d'une matrice R tel que $A_K R = I$, soit l'inverse de A_K. Un choix naturel est par exemple $A_{1..7}$, qui est la matrice identité, d'où $R = A_{1..7}$.

Nous appellerons la ligne que l'on ajoute à chaque étape *newLine*. Par mesure de simplicité, cette ligne sera simplement la prochaine ligne de A qui n'est pas encore dans $A_K = A_{1..k}$, soit la $(k+1)^{eme}$ ligne. Ici *newLine* est donc la 8^{eme} ligne. A_K devient donc :

$$A_{1..8} = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & -2 & -1 & 0 & 0 & 0 & 0 \end{matrix}$$

Nous devons recalculer R. Pour cela nous devons déterminer quels vecteurs colonnes de R sont au dessus et en dessous de l'hyperplan défini par $newLine$. Pour cela on effectue simplement

$$newLine\ R = (2\ \ -2\ \ -1\ 0\ 0\ 0\ 0).$$

On voit que le vecteur colonne 1 est au dessus de l'hyperplan, les vecteurs colonnes 2 et 3 au dessous et les quatre suivants sur l'hyperplan. Pour calculer les nouvelles arêtes, nous faisons les combinaisons linéaires entre deux arêtes positive et négative de façon à ce que la nouvelle arête soit dans l'hyperplan que l'on vient d'ajouter.

$$\text{newColonne1} = (1\ 0\ 0\ 0\ 0\ 0\ 0)^t + (0\ 1\ 0\ 0\ 0\ 0\ 0)^t = (1\ 1\ 0\ 0\ 0\ 0\ 0)^t$$
$$\text{newColonne2} = (1\ 0\ 0\ 0\ 0\ 0\ 0)^t + 2\ (0\ 0\ 1\ 0\ 0\ 0\ 0)^t = (1\ 0\ 2\ 0\ 0\ 0\ 0)^t$$

Une nouvelle colonne ne peut pas contenir simultanément une réaction et sa réaction opposée, ainsi newColonne1 est invalide car elle contient la réaction 1 dans les deux sens, elle sera donc ignorée.

Il faut maintenant mettre à jour la matrice R en supprimant ses colonnes 2 et 3, qui sont dans la partie négative de l'espace coupé par l'hyperplan, et en ajoutant newColonne2.

R devient donc :

$$R = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{matrix}$$

Les étapes suivantes étant les répétions de celle-ci jusqu'à ajout de toutes les lignes de A dans A_K nous les détaillerons moins.

$$newLine\ R = (1\ -1\ 1\ 0\ 0\ 1)$$
$$\text{newColonne1} = (\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ )^t$$
$$\text{newColonne2} = (\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ )^t$$
$$\text{newColonne3} = (\ 1\ 0\ 2\ 1\ 0\ 0\ 0\ )^t$$

newColonne2 est invalide ce qui nous donne R :

$$R = \begin{matrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{matrix}$$

Étape suivante :

$$newLine\ R = (0\ -2\ 0\ 0\ -2\ 2\ 0)$$
$$\text{newColonne1} = (\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ )^t$$
$$\text{newColonne2} = (\ 2\ 0\ 2\ 1\ 0\ 0\ 0\ )^t$$

$$R = \begin{matrix} 1 & 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{matrix}$$

Étape suivante :

$$newLine\ R = (0\ -1\ 2\ 0\ 2\ 2)$$
$$\text{newColonne1} = (\ 0\ 0\ 0\ 0\ 0\ 2\ 1\ )^t$$
$$\text{newColonne2} = (\ 1\ 0\ 2\ 1\ 0\ 2\ 0\ )^t$$
$$\text{newColonne3} = (\ 2\ 0\ 2\ 1\ 0\ 2\ 0\ )^t$$

$$R = \begin{matrix} 1 & 0 & 1 & 1 & 2 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 2 & 2 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{matrix}$$

Étape suivante :

$$newLine\ R = (-2\ 0\ -2\ 0\ -2\ 0\ 0\ -2)$$

Ici il n'y a pas de vecteurs positifs, on supprime simplement les vecteurs négatifs.

$$R = \begin{matrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 2 \\ 1 & 0 & 1 & 0 \end{matrix}$$

Étape suivante :

$$newLine\ R = (0\ 0\ 0\ 0)$$

Tous les vecteurs sont dans l'hyperplan, on n'a pas besoin de modifier R.

Étape suivante :

$$newLine\ R = (0\ 0\ 0\ 0)$$

Étape finale :

$$newLine\ R = (-2\ \ -2\ 0\ 0)$$

$$R = \begin{matrix} 0 & 1 \\ 0 & 0 \\ 0 & 2 \\ 0 & 1 \\ 0 & 0 \\ 2 & 2 \\ 1 & 0 \end{matrix}$$

On retrouve bien les deux EFMs que l'on avait trouvés précédemment. Pour des raisons de simplicité nous avons choisi de partir de la matrice identité, qui satisfait les contraintes de

positivité, puis nous avons ajouté les contraintes d'égalité stœchiométrique. Comme nous l'avons évoqué dans la sous-section 1.3.2 les outils actuels partent d'une initialisation par le noyau de N, vérifiant ainsi toutes les contraintes d'égalité stœchiométrique et également un certain nombre des contraintes de positivité. L'algorithme satisfait alors une à une incrémentalement les m contraintes de positivité restantes.

1.3.3 RegEFMTool

Quelle que soit la méthode de calcul des EFMs, l'énumération de l'ensemble des résultats pose des problèmes de débordement mémoire et/ou est coûteuse en temps dans le meilleur des cas. Elle peut être simplement irréaliste lorsque le réseau admet 10^{29} EFMs. RegEfmTool (Jungreuthmayer et al., 2012), un outil basé sur EFMtool, cherche à contourner cette limitation en ne listant que certains EFMs. L'idée derrière cette sélection est que la vaste majorité des EFMs classiquement engendrés ne sont pas valides biologiquement parlant. Cette sélection est faite en rajoutant des contraintes supplémentaires au calcul, dans leur cas des contraintes de « régulation de gènes » exprimant que l'activation ou l'inhibition de certains gènes est conditionnée par celles d'un ou plusieurs autres gènes, ce qui se traduit, au niveau métabolique, par des contraintes booléennes entre réactions enzymatiques.

Lors de l'étape de calcul des arêtes, chaque nouvelle arête est une combinaison linéaire de deux arêtes adjacentes (puisqu'une nouvelle arête se situe naturellement sur une ancienne face). Le support de la nouvelle arête est donc le résultat d'un « OU » entre les supports des deux précédentes arêtes. Les implémentations modernes de la méthode de double description codent la partie des vecteurs déjà traités en binaire et procèdent donc à des opérations optimisées sur des vecteurs bits (Gagneur and Klamt, 2004; Terzer and Stelling, 2008). Ainsi, pour les requêtes booléennes négatives (conjonctions de clauses négatives), si une arête ne respecte pas la requête alors aucune arête construite à partir de celle-ci ne la respectera. Ceci vient du fait que l'insatisfaisabilité d'une clause négative est une propriété monotone vis-à-vis de l'inclusion des supports : si cette insatisfaisabilité est vérifiée par une voie (c'est-à-dire $Supp(v)$ contient toutes les réactions correspondant aux symboles propositionnels dans la clause), elle le sera pour

toute voie dont le support contient $Supp(v)$ (Jungreuthmayer et al., 2013b). À l'inverse toute requête n'étant pas, dans le cas général, monotone ne pourra pas servir de facteur d'élimination dans la phase de calcul des arêtes et devra donc être prise en compte dans une phase de « post-traitement ».

La solution que nous allons proposer dans cette thèse permet d'interroger un réseau métabolique afin de trouver les solutions consistantes avec l'encodage du réseau et avec une formule logique encodant la requête.

L'utilisation de RegEfmTool étant très proche de celle de notre solution, nous nous comparerons à ses résultats afin de valider l'intérêt de notre approche.

1.3.4 Énumération des chemins basée sur un SAT solveur

L'utilisation d'un SAT solveur pour calculer des voies métaboliques est une méthode qui a déjà été utilisée, (Soh et al., 2012) se base sur un SAT solveur afin de calculer les voies minimales dans un réseau métabolique n'utilisant pas un ensemble de gènes. Ces travaux ont permis d'attirer l'attention sur l'utilisation des SAT solveurs dans le calcul de voies métaboliques en prédisant les voies minimales utilisées par une cellule lors d'un « gene knockout » (suppression de l'expression d'un gène).

Cette approche diffère de ce que nous présenterons par la suite sur plusieurs points :

— La représentation du réseau métabolique contient des informations supplémentaires modélisant les gènes à l'origine des enzymes. Ainsi une enzyme ne peut être utilisée que si au moins un des gènes qui la génère est présent.

— La modélisation intègre une notion de « temps ». Un métabolite n'est utilisable au temps t que si au moins une enzyme qui le produit est utilisable au temps $t-1$. Le principe est le même pour les enzymes qui seront utilisables au temps t si tous leurs métabolites sont disponibles au temps $t-1$. L'utilisation de t permet de limiter indirectement la taille des voies calculées.

— La modélisation ne prend pas en compte la stœchiométrie.

Nous souhaitons calculer des EFMs et pour ce faire nous avons besoin de prendre en compte la stœchiométrie, sans quoi nous ne pouvons calculer qu'un sous-ensemble des EFMs comme nous le verrons dans la section 2.1. La prise en compte de la stœchiométrie est un problème ne pouvant pas être modélisé simplement en logique propositionnelle, nous n'avons donc pas pu nous baser sur cette approche.

1.4 SAT

Notre approche se base sur la modélisation d'un réseau métabolique par une formule logique, grâce à l'utilisation d'un solveur « SMT » *ad hoc*, pour « Satisfiability Modulo Theories », afin de pouvoir caractériser les EFMs du réseau initial. Avant de définir plus formellement ce qu'est un SMT, nous allons présenter la problématique SAT ainsi que le fonctionnement des « SAT solveurs », dont certains mécanismes sont à la base du fonctionnement des SMT considérés dans cette partie.

1.4.1 SAT : satisfaisabilité d'une formule propositionnelle

Le problème SAT, pour satisfaisabilité, consiste à chercher si une formule Φ en logique propositionnelle admet une solution ou non. Une formule en logique propositionnelle est composée de variables a (nous utiliserons pour représenter les variables les premières lettres de l'alphabet $a\ b\ c\ldots$) et d'opérateurs logiques. Un littéral est une variable (a) ou la négation d'une variable ($\neg a$). Nous représenterons l'ensembles des variables par $Var(\Phi)$.

Une affectation ν d'une formule Φ est une fonction qui associe une valeur $\nu(p) \in \{0,1\}$ (0 correspondant à faux et 1 à vrai) aux variables $p \in Var(\Phi)$. Un modèle (ou impliquant) d'une formule Φ est une affectation ν qui satisfait la formule, noté $\nu \models \Phi$. Le problème SAT consiste à décider si une formule admet, ou non, un modèle.

Les opérateurs logiques sont :

$\vee$: La disjonction (ou « ou ») est un opérateur tel que l'évaluation de $\vee$ est 1 si au moins un opérande est affecté à 1. La table de vérité pour l'opérateur $\vee$ est la suivante :

a	b	$a \vee b$
1	1	1
1	0	1
0	1	1
0	0	0

$\wedge$: La conjonction (ou « et ») est un opérateur tel que l'évaluation de $\wedge$ est 1 si les deux opérandes sont affectés à 1. La table de vérité pour l'opérateur $\wedge$ est la suivante :

a	b	$a \wedge b$
1	1	1
1	0	0
0	1	0
0	0	0

$\neg$: La négation. La négation d'un littéral (l) est son littéral complémentaire ($\neg l$) tel que si $l = 1$, $\neg l = 0$ et inversement. De plus, d'après les lois de De Morgan, la négation d'une conjonction est équivalente à la disjonction des négations des littéraux (et vice-versa).

$$\neg(a_0 \wedge \ldots \wedge a_n) \equiv \neg a_0 \vee \ldots \vee \neg a_n$$
$$\neg(a_0 \vee \ldots \vee a_n) \equiv \neg a_0 \wedge \ldots \wedge \neg a_n$$

Definition 1.5 (Solution d'une formule) *Une solution d'une formule logique propositionnelle est une affectation (potentiellement partielle) de ses variables permettant son évaluation à vraie (elle représente donc un ensemble de modèles de la formule).*

Une formule logique peut s'exprimer sous plusieurs formes, dont les plus connues sont :

DNF, pour *Disjunctive normal form*. Le problème est alors écrit sous forme d'une disjonction de conjonctions de littéraux (appelés « cubes ») positifs ou négatifs.
L'exemple suivant est une DNF :

$$(a \wedge b \wedge c) \vee (b \wedge f) \vee (j) \vee (b \wedge c \wedge \neg b)$$

La résolution d'un problème SAT sous forme DNF est triviale, il suffit de rendre vrai un cube quelconque, c'est-à-dire tous ses littéraux. L'exemple précédent admet les trois solutions suivantes : $\{j\}$,$\{b, f\}$ et $\{a, b, c\}$. Notons que dire si une DNF est une tautologie (c'est-à-dire dont toute affectation est solution) est un problème coNP-difficile.

CNF, pour *Conjunctive normal form*. Le problème s'écrit sous la forme d'une conjonction de disjonctions de littéraux (appelés « clauses »). Montrer qu'une CNF est une tautologie est simple, il suffit que montrer que toutes ses clauses sont des tautologies. A l'inverse montrer qu'une CNF n'admet pas de solution est un problème coNP-difficile.
L'exemple suivant est une CNF :

$$(a \vee b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg b \vee a)$$

La résolution d'un problème sous forme CNF demande de trouver une affectation de variables de façon à ce que chaque disjonction, appelée clause, soit vraie, c'est-à-dire qu'au moins un de ses littéraux soit vrai. Dans l'exemple précédent $\{\neg a, \neg b, \neg c\}$ et $\{a, c\}$ sont des solutions.

Les algorithmes SAT s'intéressent aux CNF pour plusieurs raisons :

- De nombreux problèmes s'expriment naturellement sous forme de conjonctions de contraintes, il suffira de transformer chaque contrainte pour obtenir une CNF.
- La transformation de Tseitin (Tseitin, 1983) permet de transformer une formule logique propositionnelle en CNF tout en préservant sa satisfaisabilité. L'avantage de cette transformation est qu'elle se fait en temps linéaire pour toute formule grâce à l'ajout de nouvelles variables. Cet ajout va permettre de limiter l'explosion combinatoire qui aurait eu lieu lors d'une distribution classique, il n'y a donc pas d'équivalence logique entre la formule initiale et la formule finale, mais cette transformation garantit que les deux formules sont « équisatisfaisables ».

Une méthode de résolution naïve d'une formule CNF pourrait consister à énumérer toutes les combinaisons possibles, par exemple sous forme d'une table de Karnaugh, ou de transformer la formule en DNF afin d'obtenir simplement les solutions. Sur des problèmes triviaux cela reste envisageable, cependant leurs complexité, $O(2^n)$ pour la table de Karnaugh et $O(\prod_{i=1}^{n} |conj_i|)$, avec $|conj_i|$ la taille de la i^{eme} conjonction, pour l'énumération les rendent impraticables sur des problèmes non triviaux.

1.4.2 Les premiers algorithmes SAT

Avant d'introduire l'un des algorithmes, nous allons introduire la notion de résolution. Le principe de résolution est un principe très ancien de raisonnement (on trouve un raisonnement similaire dans le syllogisme introduit par Aristote). Si dans deux clauses d'une formule Φ un littéral apparaît avec des signes opposés, on peut ajouter à la formule une nouvelle clause c, qui est l'union des deux clauses moins ce littéral. La formule Φ', tel que $\Phi' = \Phi + c$, et Φ sont équivalentes, en effet la clause c est impliqué par Φ (on parle parfois de clause implicite). Ainsi la résolution des clauses $a \vee b$ et $\neg b \vee c$ produit $a \vee c$, ce qui s'écrit :

$$\frac{a \vee b \quad \neg b \vee c}{a \vee c}$$

Une façon simple de le voir est d'écrire $a \vee b$ comme $\neg a \Rightarrow b$ et $\neg b \vee c$ comme $b \Rightarrow c$, ce qui donne naturellement $\neg a \Rightarrow c$ soit $a \vee c$. On dit que $a \vee c$ est le résolvant des clauses $a \vee b$ et $\neg b \vee c$.

DP, 1960

La procédure Davis-Putnam (DP) (Davis and Putnam, 1960) est l'application systématique de toutes les résolutions possibles, une variable après l'autre, d'une formule CNF. La procédure se termine par l'obtention d'une clause vide (UNSAT) ou d'une formule vide (SAT). En effet, une fois que toutes les résolutions sur une variable ont été faites, toutes les clauses contenant cette variable peuvent être éliminées, tout en préservant la satisfaisabilité de la formule initiale,

on appelle cette opération l'élimination d'une variable. La procédure DP peut donc être vue comme l'élimination successive de toutes les variables. Nous présentons cette procédure dans l'algorithme 1.

Algorithm 1 ProcédureDP(Φ)
Input : Φ une formule en logique propositionnelle sous forme de CNF.
Output : La satisfaisabilité de Φ

```
1: while (Var(Φ) ≠ {}) do
2:    Choisir une variable a dans Var(Φ)
3:    Remplacer dans Φ toutes les clauses contenant le littéral a (ou ¬a) par celles pouvant
      être obtenues par résolution sur a.
4:    if la clause vide apparaît then
5:       return Insatisfaisable
6:    end if
7: end while
8: return Satisfaisable
```

Le déroulement de DP sur l'exemple précédent avec l'ordre de résolution b, c, a donnerait :

$$(a \vee b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg b \vee a)$$

$$(a \vee a \vee \neg c) \wedge (\neg a \vee c)$$

$$(a \vee \neg a) \Leftrightarrow \top$$

La résolution a donné une tautologie, la formule est satisfaisable.

Le problème de cet algorithme est qu'il demande à chaque étape de produire tous les résolvants pour la variable en cours. Le nombre de clauses à l'étape $n+1$ après la résolution sur le littéral l est donc $|$clauses avec l à l'étape $n| * |$clauses avec $-l$ à l'étape $n| + |$ clauses sans l à l'étape $n|$. On observe une véritable explosion combinatoire dans le nombre de clauses potentielles à maintenir.

Des implémentations de cette procédure (Dechter and Rish, 1994; Chatalic and Simon, 1999) sont toujours utilisées dans les SAT solveurs modernes lors de la phase de pré-traitement.

DLL, 1962

La procédure Davis-Logeman-Loveland (DLL ou DPLL pour ne pas oublier l'importance de Putnam dans ce travail) (Davis et al., 1962) se base sur un parcours d'arbre systématique. A chaque étape, l'algorithme choisit et affecte une variable. Si cette affectation ne produit pas de conflit avec les affectations ayant déjà été faites, c'est-à-dire qu'aucune clause n'est vide, l'algorithme affecte le littéral suivant jusqu'à l'apparition d'un conflit ou l'affectation de tous les littéraux. Si tous les littéraux sont affectés, alors la formule est satisfaisable et l'affectation en est une solution. Si, au contraire, un conflit apparaît, l'algorithme remonte dans l'arbre jusqu'à la dernière variable qu'il a affectée librement pour explorer le cas opposé.

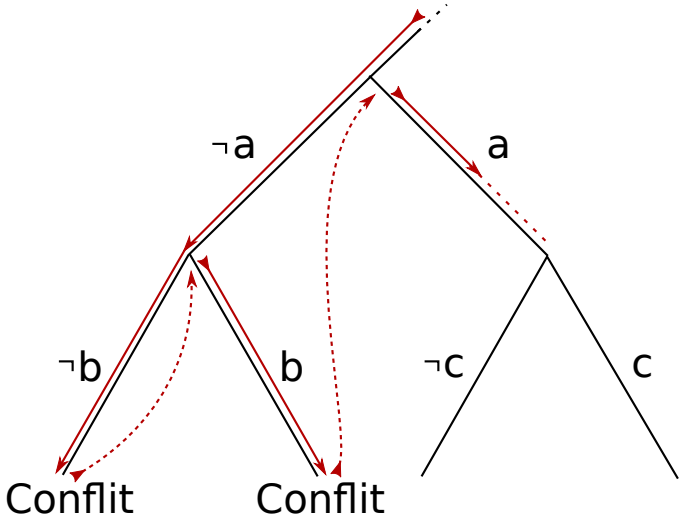


FIGURE 1.10: Illustration du fonctionnement d'un solveur DLL

Ce parcours assure que l'algorithme est « complet », c'est-à-dire que quelle que soit la formule logique l'algorithme est en mesure de trouver une solution s'il en existe une ou de déterminer que la formule est insatisfaisable.

L'algorithme 2 présente la procédure DPLL.

La méthode *simplifier* (ligne 1) se charge de la propagation unitaire ainsi que de s'assurer que tous les littéraux purs sont affectés.

Definition 1.6 (Propagation unitaire) *La propagation unitaire intervient lorsque l'affecta-*

Algorithm 2 ProcédureDP(Φ)
Input : Φ une formule en logique propositionnelle sous forme de CNF.
Output : La satisfaisabilité de Φ

```
1: Φ = simplifier(Φ)
2: if (Φ = ⊥) then
3:    return faux % Φ est trivialement insatisfaisable
4: end if
5: if (Φ = {}) then
6:    return vrai % Φ est trivialement satisfaisable
7: end if
8: a = choisirLiteral(Φ)
9: if ProcédureDP(Φ ∪ {a}) = vrai) then
10:   return vrai
11: else
12:   return ProcédureDP(Φ ∪ {¬a})
13: end if
```

tion courante des variables implique l'existence d'une clause de taille n dont $n-1$ littéraux sont du signe contraire à cette affectation et un littéral n'est pas actuellement affecté : ce littéral devient l'unique moyen de satisfaire la clause et doit donc être affecté en ce sens.

La propagation unitaire est l'un des points clés du fonctionnement des SAT solveurs modernes et doit donc être implémentée avec soin (Zhang and Stickel, 2000).

La méthode *choisirLiteral* (ligne 8) se charge de choisir l'ordre d'affectation des littéraux. Dans un parcours d'arbre l'ordre de branchement est primordial, considérons la formule suivante composée de cinq clauses :

1. $\neg a \vee b$
2. $b \vee c \vee a$
3. $c \vee \neg b$
4. $\neg c \vee a \vee \neg b$
5. a

Nous représentons figure 1.11 le parcours selon deux ordres différents, en indiquant lors des conflits quelle clause en est la cause.

L'ordre de parcours peut être critique pour résoudre efficacement certaines formules. Dans un premier temps des indices statiques ont été utilisés tel que la taille des clauses dans lesquelles

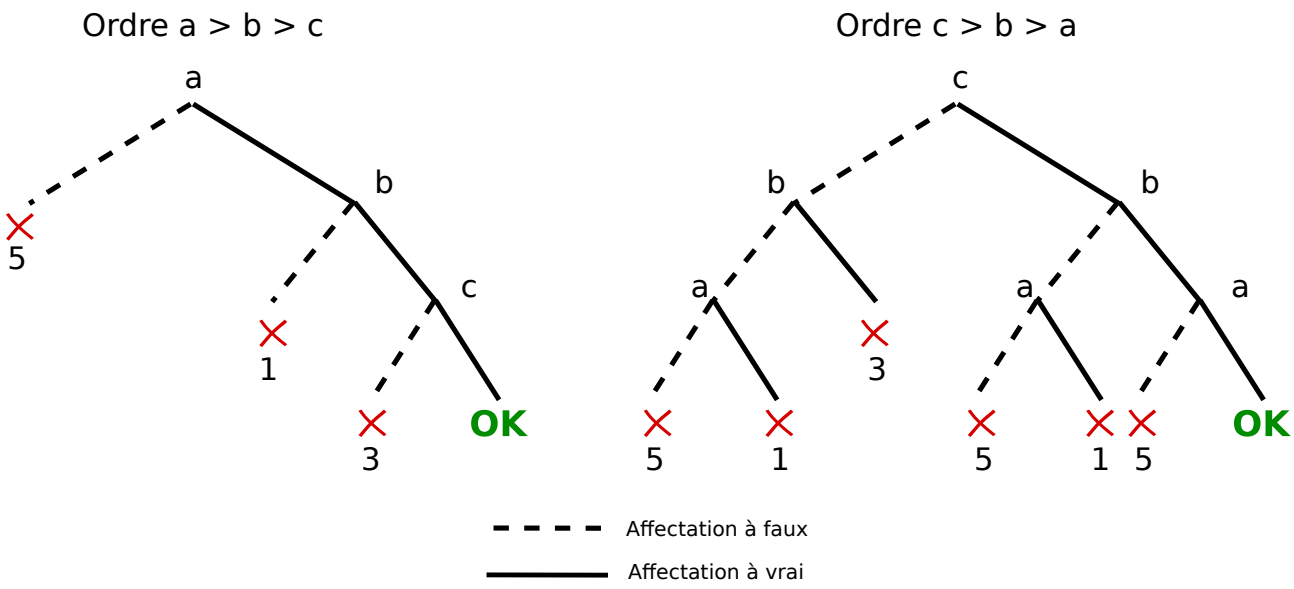


FIGURE 1.11: Impact de l'ordre d'exploration sur un petit exemple

apparaît chaque littéral, en considérant la formule simplifiée au noeud de l'arbre de recherche considéré. Intuitivement, vérifier les clauses unitaires en premier permet de couper l'arbre au plus tôt. De même les littéraux apparaissant dans des clauses n'ayant plus que deux littéraux non affectés permettent d'obtenir des coupes supplémentaires par propagation unitaire. En généralisant, on obtient une heuristique nommée « Maximum Occurrence in clauses of Minimum Size » (Jeroslow and Wang, 1990)

1.4.3 Les solveurs modernes et le CDCL

Le CDCL

Les solveurs modernes se fondent sur l'algorithme CDCL (Silva and Sakallah, 1997) (Conflict-Driven Clause Learning). Cet algorithme utilise les conflits qu'il engendre pour apprendre de nouvelles clauses. Si la formule est insatisfaisable, le solveur va finir par apprendre une clause unitaire a qui, lors de la propagation unitaire, va impliquer $\neg a$ et ainsi démontrer que la formule est insatisfaisable. Si la formule est satisfaisable un modèle sera trouvé au cours du déroulement, on ne sait pas si la formule est SAT avant d'avoir affecté toutes les variables.

Le fonctionnement du CDCL est présenté dans l'algorithme 3.

Algorithm 3 CDCL(Φ)
Input : Φ une formule en logique propositionnelle sous forme de CNF.
Output : La satisfaisabilité de Φ

```
1: niveauDecision = 0
2: while (toutesVariablesAffectée(Φ) = faux) do
3:    {conflit, clause} = propagationUnitaire(Φ)
4:    if (conflit ≠ CONFLIT) then
5:       a = choisirLiteral(Φ)
6:       Φ = Φ ∪ {a}
7:       niveauDecision+ = 1
8:    else
9:       if (niveauDecision = 0) then
10:         return faux
11:      else
12:         {niveauBackTrack, nouvelleClause} = analyseConflit(clause)
13:         backTrack(Φ, niveauBackTrack) % se charge également de mettre à jour le niveau de décision
14:         Φ = Φ ∪ {nouvelleClause}
15:      end if
16:   end if
17: end while
18: return vrai
```

Explication de la procédure CDCL : initialement, les littéraux ne sont pas affectés, l'heuristique « *choisirLiteral* » va sélectionner et affecter un littéral. Chaque décision de l'heuristique est appelée un niveau (« *niveauDecision* »), il s'agit simplement du comptage du nombre de décisions. Ensuite l'algorithme va propager les conséquences de cette affectation via la propagation unitaire.

Lors d'une propagation unitaire le solveur peut apprendre une clause vide, on parle alors de conflit. Cela se produit lorsque l'affectation des littéraux lors des décisions ne peut pas mener à une solution. C'est à ce processus que l'algorithme doit son nom de « Conflict-Driven Clause Learning ». Afin de ne pas reproduire l'erreur de décision, il va appeler la méthode « *analyseConflit* » afin de trouver le littéral qui a causé le conflit et la raison pour laquelle il crée un conflit. Pour faire cela le solveur va apprendre, par résolutions successives, une clause ne contenant qu'un seul littéral ayant un niveau égal à celui du conflit. Cette clause est appelée une « clause assertive ».

Le fonctionnement de la clause assertive peut être compris comme suit. On sait que l'affectation

de certains littéraux plus le dernier littéral à avoir été affecté, $l_{conflit}$, engendre un conflit. Si on ajoute une clause contenant la négation de tous ces littéraux et que l'on relance l'algorithme avec cette nouvelle clause en plus dans la formule en gardant le même ordre de décision, nous arriverons à un moment à devoir propager $\neg l_{conflit}$. $l_{conflit}$ n'est donc plus un littéral de décision relativement aux décisions faites au dessus. En ajoutant cette clause, l'algorithme a appris à éviter ce conflit. En réalité l'algorithme n'a pas besoin de recommencer depuis le début pour obtenir ce résultat, il doit réaliser un retour arrière ou « backtrack ».

Definition 1.7 (Backtrack) *Un bactrack jusqu'au niveau X est le fait de libérer tous les littéraux ayant un niveau supérieur à X.*

Le backtrack se fait jusqu'au niveau où la clause assertive a exactement un littéral non affecté. Ainsi puisque la clause assertive a été ajoutée à la base de clauses, lors de la propagation unitaire, ce littéral sera affecté à une valeur différente de sa dernière affectation.

L'exemple ci-dessous illustre le fonctionnement du CDCL. Soit la formule suivante :
L'ordre de propagation sera X1, ¬X4, X7 et ¬X9,

(1)	¬X1 ∨ X3	(4)	X9 ∨ ¬X5 ∨ X10	(4)	¬X11 ∨ ¬X10 ∨ ¬X14
(1)	¬X1 ∨ ¬X2	(4)	¬X10 ∨ X11	(4)	¬X12 ∨ ¬X3 ∨ X15
(2)	X4 ∨ X5	(4)	¬X10 ∨ X6 ∨ X12	(4)	¬X13 ∨ X14 ∨ ¬X16
(2)	X4 ∨ X2 ∨ ¬X6	(4)	¬X11 ∨ X13	(4)	¬X15 ∨ X14 ∨ X16
(3)	¬X7 ∨ X4 ∨ ¬X8				

Le numéro entre parenthèses correspond au niveau auquel la clause sera complètement affectée. Le schéma, figure 1.12, modélise le fonctionnement du CDCL.

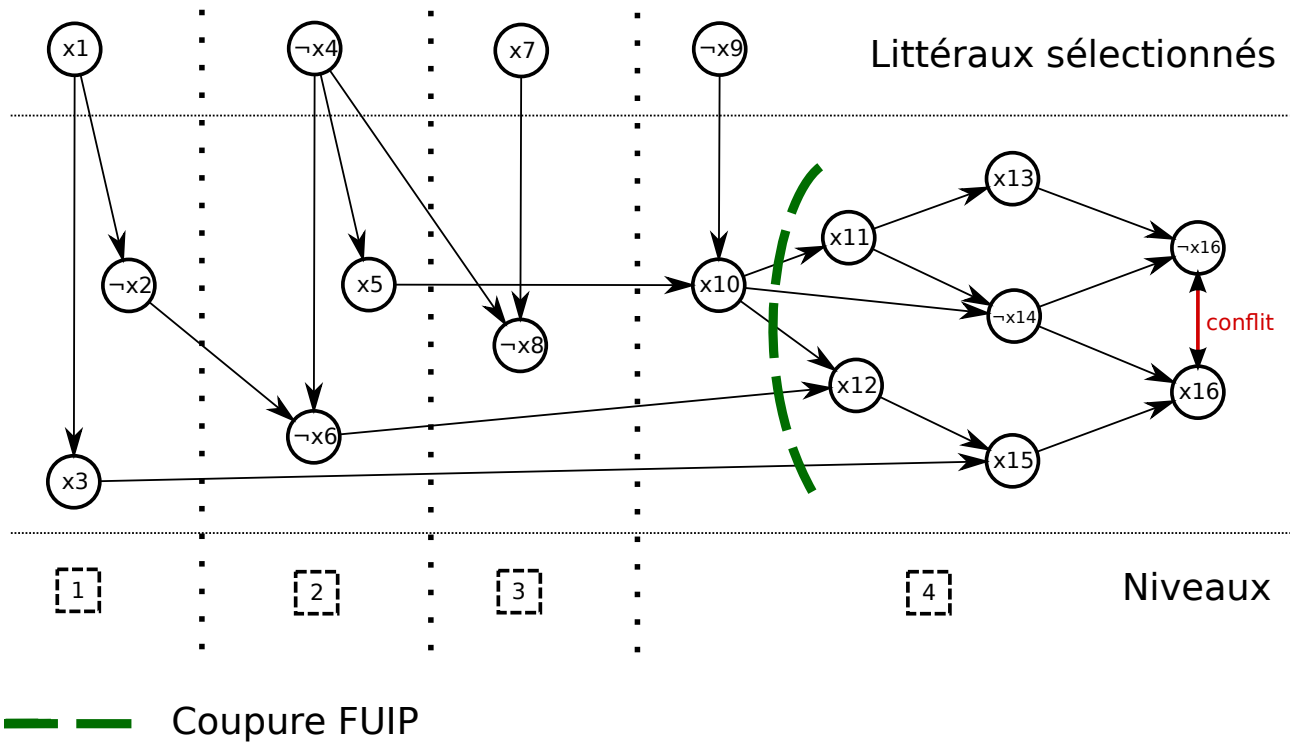


FIGURE 1.12: Fonctionnement général d'un CDCL

Nous voyons que ce modèle partiel conduit à un conflit. En effet, pour satisfaire ce modèle partiel, X16 devrait être simultanément assigné à vrai et à faux.

La phase d'analyse de conflit va sélectionner la clause assertive, ici $\neg X3 \vee X6 \vee \neg X10$ et l'ajouter à la base de clauses. Puis le solveur va faire un retour jusqu'au plus haut niveau des littéraux de la clause qui n'est pas celui du conflit, ici jusqu'au niveau 2. Et reprendre la propagation unitaire, ce qui nous donne la configuration présenté figure 1.13.

La clause apprise permet donc de ne pas reproduire une mauvaise affectation. Dans l'exemple précédent, le conflit était dû à l'affectation simultanée à Vrai de $X3$, $\neg X6$ et $X10$, configuration qui ne pourra dorénavant plus se produire. L'ajout de clauses permet donc d'assurer qu'un solveur CDCL se termine en temps fini. Cependant un solveur peut facilement générer un grand nombre de clauses et les conserver toutes ce qui, bien que cela assure la complétude du solveur, n'est pas viable. C'est pour cela que les solveurs modernes implémentent une stratégie de suppression de clauses. Cela permet de gagner de la mémoire et du temps de traitement en supprimant les clauses qui ne semblent pas intéressantes selon une certaine heuristique. Il a

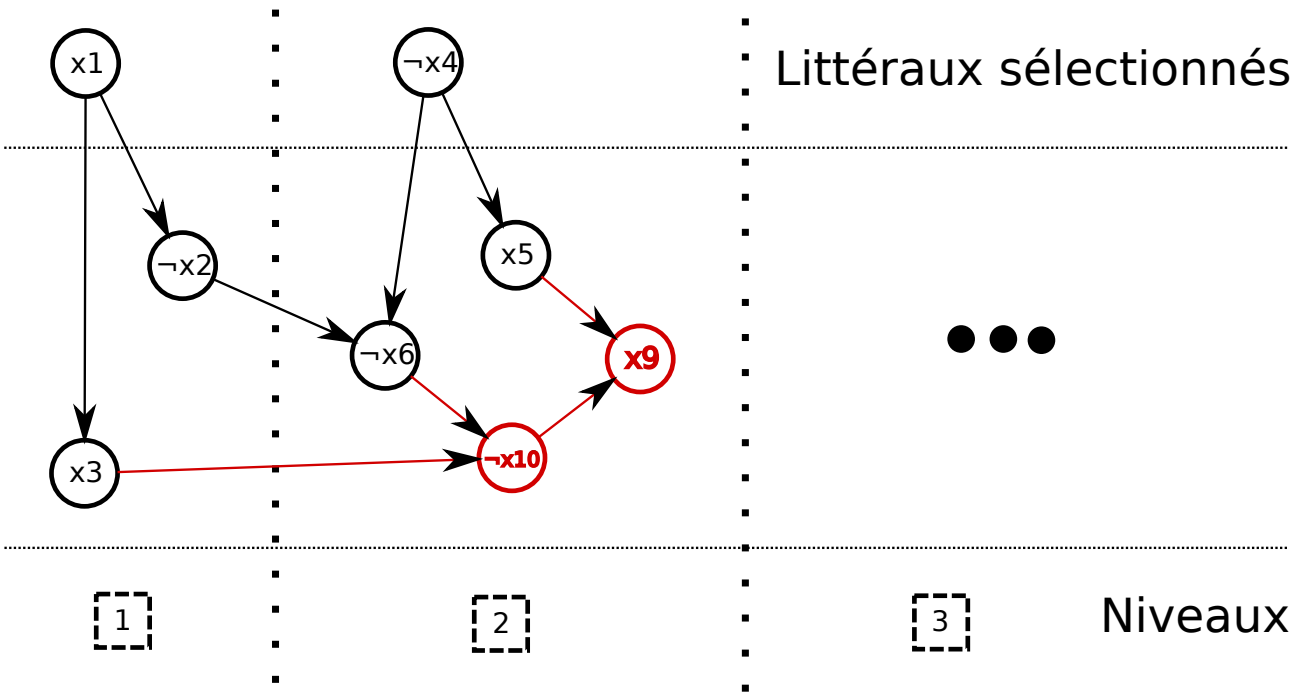


FIGURE 1.13: État du solveur après le processus d'apprentissage de clause

était montré que le choix de cette heuristique avait un impact significatif sur les performance du solveur (Long et al., 2013; Simon, 2014).

Comme pour le DPLL, l'ordre de décision joue un rôle primordial dans l'efficacité du solveur. L'heuristique de sélection qui s'est imposée parmi les solveurs modernes est VSIDS (Moskewicz et al., 2001) (Variable State Independent Decaying Sum). Elle repose sur les points suivants :

- Un score attribué à chaque variable (une variable représente un littéral sans la notion de signe, ainsi $X1$ et $\neg X1$ sont deux littéraux différents mais une seule et même variable $X1$). Ce score est initialisé à une valeur, qui peut par exemple être zéro ou un nombre aléatoire proche de zéro.
- Une valeur d'augmentation, qui sera ajoutée au score des variables.
- Un coefficient d'augmentation, qui va servir à faire croître la valeur d'augmentation.

Nous allons présenter l'implémentation de VSIDS utilisé par Minisat, bien qu'on puisse l'implémenter différemment, celle-ci est couramment utilisée. Chaque fois que le solveur apprend une clause lors d'un conflit, on ajoute la valeur d'augmentation au score des variables de la clause apprise. Puis la valeur d'augmentation est multipliée par le coefficient d'augmentation.

La valeur d'augmentation a donc une progression exponentielle. C'est grâce à cette progression exponentielle que le VSIDS acquiert sa réactivité. En effet, à titre d'exemple, dans les implémentations des solveurs modernes, si une variable X1 n'apparaît pas pendant une centaine de conflits, soit environ un dixième de seconde, elle aura un score inférieur à celui d'une variable X2 qui apparaît pour la première fois après ces cent conflits, et cela, même si X1 était dans les variables avec les meilleurs scores à sa dernière apparition. Cette réactivité permet au solveur de travailler sur les clauses qui sont actuellement au centre du problème et « d'oublier » les variables qui n'y sont plus. Cela n'empêche cependant pas le solveur de revenir sur une variable qu'il a « oubliée » si celle-ci réapparaît de manière récurrente dans les clauses apprises. Pour des raisons d'implémentation, notamment la limite de stockage physique du type double, on ne peut pas laisser les scores croître de manière exponentielle indéfiniment. Pour cette raison, le score de toutes les variables ainsi que la valeur d'augmentation sont divisés par une constante lorsqu'un score atteint une valeur trop importante (de l'ordre de 10^{100}) ce qui permet de simuler une croissance infinie.

Les raisons de l'efficacité de l'utilisation VSIDS est mal compris et reste un sujet d'étude (Liang et al., 2015).

Les optimisations classiques

La majorité des solveurs modernes utilise ces optimisations ou des versions très similaires. C'est le cas de MiniSat (Eén and Sörensson, 2003) qui depuis 2005 fournit une implémention de solveur CDCL comprenant toutes les optimisations décrites ci-dessous.

Le phase saving Le « phase saving » (Pipatsrisawat and Darwiche, 2007) consiste à garder en mémoire le signe de chaque variable lors de sa dernière affectation. Initialement le phase saving de chaque variable est affecté à faux (dans Minisat) puis sera modifié lors de l'affectation de la variable. Ainsi, après un retour à un niveau inférieur, le solveur va continuer sa progression dans le même espace de recherche. Une manière de voir le fonctionnement du phase saving est de considérer le solveur comme un algorithme de recherche dans un arbre : le fait de garder

la polarité des variables permet d'explorer de manière similaire une branche différente et ainsi d'augmenter la localité de la recherche. Il faut pour cela que l'ordre d'affectation des variables soit le même. La figure 1.14 présente un exemple de fonctionnement du « phase saving ».

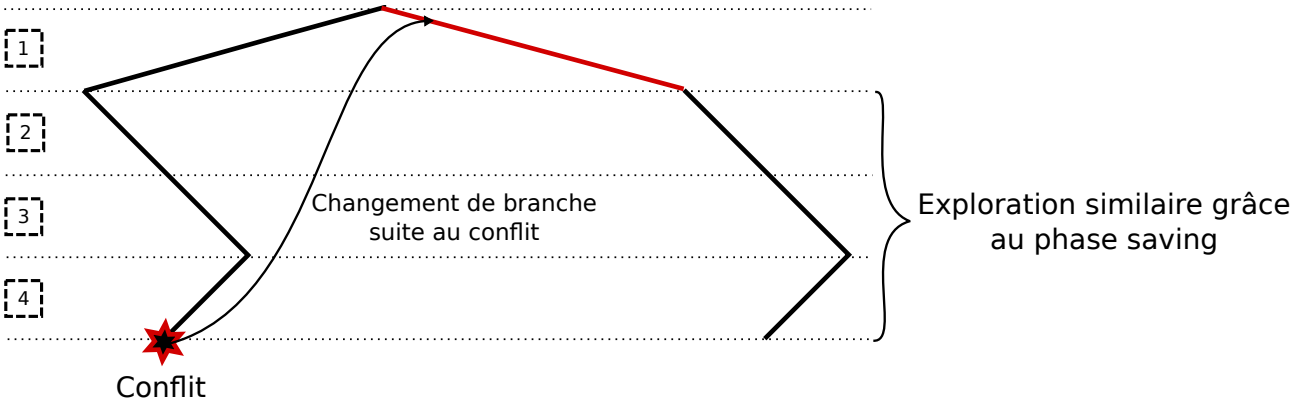


FIGURE 1.14: Exemple d'exploration d'arbre utilisant le phase saving

Les watched litterals Les « watched litterals » (Moskewicz et al., 2001) sont un moyen de réduire le coût de traitement de la propagation unitaire. Une implémentation naïve de la propagation unitaire consisterait, à chaque affectation d'une variable, à regarder toutes les clauses dans lesquelles apparaît cette variable puis à parcourir ces clauses jusqu'à trouver, soit deux littéraux non affectés, soit un littéral affecté à vrai. En effet ces deux cas nous assurent de ne pas avoir de propagation unitaire puisqu'on effectue une propagation unitaire lorsque tous les littéraux d'une clause sauf un sont affectés à faux et que le dernier n'est pas affecté. L'idée des watched litterals est de regarder uniquement deux littéraux par clause : si l'un des deux littéraux est vrai ou si les deux ne sont pas affectés, on peut simplement ignorer la clause. Lorsque l'un des watched litterals vient à être affecté à faux, on va chercher un autre littéral à surveiller ; si on n'en trouve pas, on est dans le cas où il faut faire une propagation unitaire du deuxième watched litteral. Les watched litterals permettent donc de passer d'une situation où on doit parcourir les clauses contenant la variable qui vient d'être affectée à une situation où l'on doit parcourir uniquement les clauses dans lesquelles cette variable est un watched litteral. Le coût de traitement d'une clause est donc, dans le cas général, indépendant du nombre de variables de celle-ci.

Les restarts Un « restart » (redémarrage) consiste à libérer toutes les variables. Les restarts ont dans un premier temps été vus comme un moyen de changer d'espace de recherche. Ils sont maintenant vus comme un moyen de réduire la taille de la preuve construite par le solveur en réordonnant l'ordre des variables (Gomes et al., 2000). En effet, puisque le score des variables n'est pas changé lors du restart, le solveur va commencer par réaffecter les variables ayant le score le plus haut donc en général dans un ordre qu'il n'avait pas sélectionné initialement (même si les affectations sont les mêmes, les dépendances entre littéraux assignés sont réordonnées). Les stratégies de restart sont très variables d'un solveur à l'autre : elles peuvent, par exemple, suivre une série géométrique ou une série de type Luby (Luby et al., 1993). Une série de type Luby permet d'avoir des restarts rapides tout en faisant rarement des restarts exponentiellement plus longs, la figure 1.15 présente un exemple de série Luby.

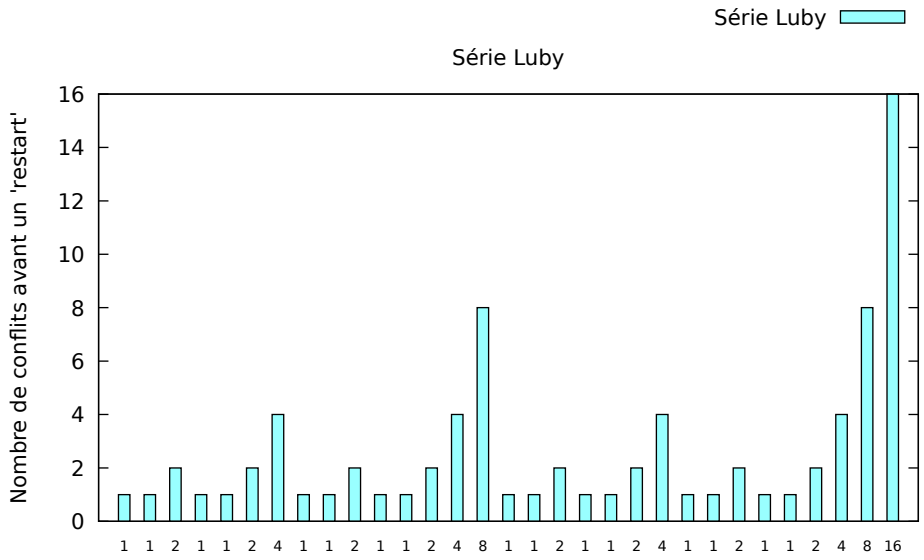


FIGURE 1.15: Exemple de série Luby

Bien qu'il soit dur de savoir à l'avance quelle stratégie sera efficace pour un type d'instance (Hamadi et al., 2008) les restarts améliorent grandement les performances des solveurs (Haim and Heule, 2014).

1.4.4 Les SMT

Le fonctionnement d'un SMT

La puissance des solveurs modernes a attiré l'attention d'autres domaines tels que la vérification de circuits électroniques ou la planification. L'idée des SMT est d'utiliser la capacité de raisonnement sur une formule logique des solveurs SAT sur un problème n'étant pas exprimable en formule logique propositionnelle. Cela permet, par exemple, de raisonner avec des nombres réels, de l'arithmétique linéaire ou diverses structures de données comme les listes, les tableaux ou les vecteurs de bits. Il existe de nombreux type de SMT mais nous nous limiterons à l'étude des CDCL(T) (aussi appelé DPLL(T)) (Nieuwenhuis et al., 2006).

Pour cela il faut deux choses :

- Un solveur SAT dont le fonctionnement sera légèrement modifié afin d'interagir avec le solveur de théorie ou SMT.
- Un SMT, qui permettra un raisonnement de plus haut niveau que le SAT solveur et qui sera à même de lui communiquer ses résultats.

La première étape de tout SMT est de transformer le problème exprimé dans une théorie en formules exploitables par un SAT solveur. Pour cela le SMT doit être capable de détecter les négations afin de pouvoir encoder le problème sous forme de littéraux. Par exemple $a < b \vee b = c$, $a \geq b$ sera transformé en $l_1 \vee l_2$, $\neg l_1$: puisque le solveur aura fait la correspondance entre le littéral « l_1 » et la proposition « $a < b$ » la négation de celle-ci, « $a \geq b$ » sera encodée par la négation du littéral.

Lorsqu'on utilise un SMT on va encoder le problème que l'on cherche à résoudre dans le formalisme du SMT. Celui va ensuite abstraire le problème en l'encodant en logique propositionnelle. Il devra ensuite, étant donnée une assignation totale au niveau de l'abstraction, de dire si le modèle est valide selon sa théorie. En effet, lors de la transformation en logique propositionnelle, le SMT a abstrait le problème. Cela implique que tout modèle qui n'est pas valide pour le SAT solveur ne peut pas être valide une fois étendu à la théorie considérée. Cependant, un modèle

valide pour le SAT solveur peut ne pas être valide pour le SMT ou bien mener à de multiple solutions pour le SMT.

Ainsi, si le SMT peut invalider un modèle, le SAT solveur pourra apprendre une clause empêchant de retrouver ce modèle, la solution la plus naïve étant une clause contenant la négation du modèle, ce qui assurera la correction du SMT. En effet, à partir d'un nombre fini de solutions valides pour le SAT solveur, le SMT finira par l'épuisement de toutes les solutions potentielles ou par la validation de l'une d'elles. Naturellement un SMT fonctionnant comme décrit ci-dessus ne tire que très peu profit du fonctionnement d'un SAT solveur.

La première amélioration que l'on peut envisager pour notre SMT de base est qu'il fournisse une raison lorsqu'il invalide un modèle. Ainsi le SAT solveur peut mettre en place les mécanismes d'apprentissage vus précédemment. Intuitivement, là ou avant on coupait une feuille dans l'arbre de recherche, en fournissant une explication on coupera une ou plusieurs branches.

L'amélioration suivante découle de celle-ci. Si on veut influencer la recherche autant que possible, il faut que le SMT puisse être interrogé non plus sur un modèle complet mais sur un modèle partiel afin de ne pas continuer la recherche si la solution actuelle est déjà invalide. Il faut toutefois faire attention car, en fonction du SMT, la validation d'un modèle partiel peut être coûteuse et, dans ce cas, une validation systématique à chaque affectation ne sera pas forcément une solution optimale. Une validation régulière mais non systématique peut être envisagée du moment que l'on valide les affectations complètes afin de garantir la correction de l'algorithme.

De plus la validation d'un modèle partiel permet également au SMT de propager les conséquences de l'affectation actuelle. De même que pour la validation partielle et pour la validation du modèle total, la propagation systématique côté SMT peut ne pas être optimale et, lors d'une propagation, le SMT devra en fournir une raison afin que l'apprentissage du CDCL puisse se dérouler de manière transparente.

Chapitre 2

Approche Sat

L'idée initiale de cette thèse est d'utiliser les progrès faits ces dernières années dans le domaine des solveurs SAT, plus particulièrement leur capacité à trouver un modèle et à démontrer l'insatisfaisabilité d'une formule, afin de proposer une nouvelle approche pour le calcul des EFMs dans un réseau métabolique. Bien que nous souhaitons pouvoir énumérer tous les EFMs, notre objectif principal est de lister efficacement les EFMs respectant des contraintes booléennes sur les réactions enzymatiques. Dans la suite, nous appellerons ces contraintes booléennes simplement « requêtes ».

Nous allons présenter ici les différentes étapes de cette approche. Le choix de l'utilisation d'un SAT solveur a été motivé par plusieurs raisons qui découlent du fait qu'à l'heure actuelle la principale méthode de calculs d'EFMs se base sur l'algorithme de « double description » : cet algorithme permet d'intégrer une partie des requêtes lors du calcul des EFMs, comme le fait RegEfmTool, mais certaines requêtes restent traitées en post-traitement. L'approche SAT permet d'intégrer toutes les requêtes dans le calcul des EFMs et ainsi de ne calculer que ce qui répond effectivement à la requête. Un autre avantage de l'approche SAT est que le calcul des EFMs se fait de façon incrémentale, ainsi sur un modèle trop gros pour en calculer exhaustivement toutes les solutions le SAT solveur restera en mesure de trouver des solutions là où un algorithme de double description n'en produira aucune car la mémoire nécessaire pour

la phase intermédiaire du calcul est trop importante. Un exemple particulièrement marquant peut être lorsqu'on veut vérifier qu'il n'existe pas d'EFM respectant une contrainte booléenne sur un gros réseau. L'approche SAT va pouvoir tirer parti de l'ensemble de la contrainte et ainsi fournir un résultat rapide là où la méthode de double description doit potentiellement lister tous les résultats avant de pouvoir affirmer qu'il n'y a pas de solution à cette requête.

L'utilisation classique des SAT solveurs est de trouver une solution d'une formule en logique propositionnelle ou de démontrer que celle-ci n'en admet pas. L'usage que l'on souhaite en faire pour le calcul des EFMs diffère donc en deux points de l'utilisation classique d'un SAT solveur :

1. Nous ne cherchons pas une solution mais l'ensemble des solutions ;
2. Le calcul d'EFM fait intervenir des coefficients de stœchiométrie ce qui est en dehors du domaine d'application des SAT solveurs puisque ceux-ci travaillent sur des domaines booléens.

Nous regarderons d'abord comment calculer le support d'une voie à l'état stationnaire lorsque l'on met de côté la stœchiométrie, puis nous développerons, dans un premier temps, le calcul d'un ensemble de supports de voies et, dans un second temps, le calcul de l'ensemble des supports d'EFMs. Enfin nous proposerons une solution pour le calcul des supports d'EFMs lorsqu'on prend en compte la stœchiométrie et nous développerons les problèmes que cela entraîne.

2.1 Modélisation d'une voie sans stœchiométrie à l'état stationnaire

Nous allons, dans cette section, présenter comment utiliser un SAT solveur afin qu'il puisse trouver les supports de voies à l'état stationnaire dans un réseau métabolique sans rencontrer de problèmes liés à la stœchiométrie. La première étape consiste à se placer dans une sous-partie du réseau dans laquelle toute voie sera une voie valide à l'état stationnaire. Pour ce faire il faut que ce sous-réseau respecte les conditions suivantes :

1. les réactions de ce sous-réseau ont des coefficients stœchiométriques égaux à un ;
2. chaque métabolite interne de ce sous-réseau est produit ou consommé par exactement une réaction.

La première condition découle naturellement du fait que l'on ne veut pas traiter de stœchiométrie. En effet, lorsque le réseau métabolique a des coefficients stœchiométriques de plus de un, une voie peut ne pas être à l'état stationnaire car elle peut créer/consommer plus de métabolites qu'elle en consomme/crée. Un exemple simple de cas problématique est représenté dans la figure 2.1.

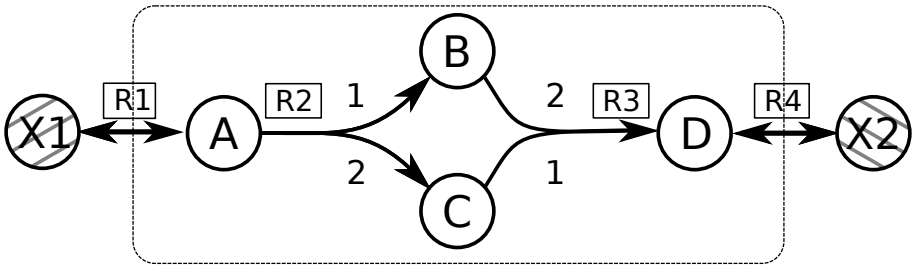


FIGURE 2.1: Exemple d'une voie n'étant pas un EFMcar ne pouvant pas être à l'état stationnaire.

Mais bloquer uniquement les réactions ayant une stœchiométrie supérieure à un ne suffit pas.

En effet si un métabolite est produit/consommé par plus d'une réaction les problématiques de stœchiométrie peuvent apparaître. La figure 2.2 présente un exemple de ce cas : la réaction $R6$ consomme un B et un F, or pour produire F il faut un D et un E ce qui implique deux C, et la réaction $R2$ produit autant de B que de C. Ainsi il faudra produire deux B pour produire un F ce qui entraînera une augmentation de la concentration de B.

Il nous faut donc ajouter la condition 2 afin d'assurer que ce cas ne se produira pas.

Sous ces conditions nous pouvons donc ignorer la stœchiométrie et il devient possible d'encoder une voie en logique propositionnelle, nous permettant ainsi d'utiliser un SAT solveur pour calculer une voie dans un réseau métabolique, s'il en existe une.

Dans ce chapitre nous modéliserons la présence (respectivement l'absence) dans une voie d'une enzyme par l'assignation à vrai (respectivement faux) d'une variable propositionnelle r. De

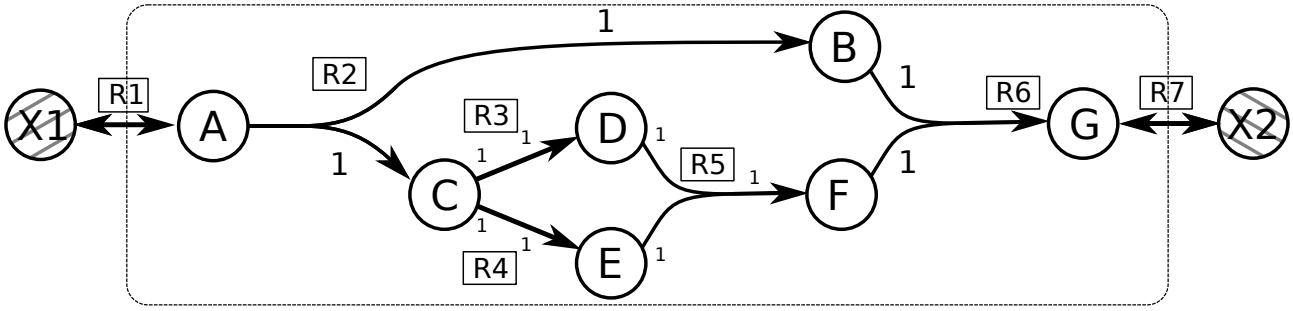


FIGURE 2.2: Exemple d'une voie ayant des stœchiométries à 1 et n'étant pas un EFMcar ne pouvant pas être à l'état stationnaire.

même l'utilisation (respectivement non utilisation) par une voie d'un métabolite sera modélisée par l'assignation à vrai (respectivement faux) d'une variable propositionnelle m.

Il y a une bijection entre la présence d'un enzyme/métabolite dans une voie et l'assignation à vrai de la variable encodant cet enzyme/métabolite dans l'assignation de variables modélisant cette voie. On parlera d'un enzyme/métabolite actif dans la voie si la variable qui l'encode est assignée à vrai.

Il est important de noter que $m = \top$ dans une assignation encodant une voie indique la présence du métabolite m dans cette voie sans pour autant garantir que la production et la consommation de celui-ci soient égales. Cette garantie est fournie par les conditions que nous avons fixées précédemment.

Dans la suite nous allons avoir besoin d'encoder les propriétés suivantes portant sur une variable r encodant une réaction donnée, parmi l'ensemble R de toutes les variables encodant les réactions :

— La réaction encodée par r produit un métabolite m ($r \in produit(m, R)$) ;

— La réaction encodée par r consomme un métabolite m ($r \in consomme(m, R)$) ;

— La réaction encodée par r ne fait pas intervenir de stœchiométrie supérieure à un ($r \notin maxSto(R > 1)$).

Pour des raisons de simplicité $produit(m, R)$, $consomme(m, R)$ et $maxSto(R > 1)$ sont exprimés comme des ensembles et pas en logique propositionnelle bien que cela soit possible. En effet, par exemple, pour encoder $produit(m, R)$ il faudrait ajouter un prédicat $produit(r, m, R)$ tel que

$produit(r, m, R) = \top$ représente la production de m par r (c'est-à-dire $r \in produit(m, R)$) et que $produit(r, m, R) = \bot$ représente la non production de m par r (c'est-à-dire $r \notin produit(m, R)$). La même logique serait applicable pour encoder $consomme(m, R)$ et $maxSto(R > 1)$ en logique propositionnelle.

2.1.1 Calcul d'une voie à l'état stationnaire sans stœchiométrie

Afin de simplifier l'expression des règles, nous divisons chaque réaction réversible r en deux réactions irréversibles r et r_{rev}. Ainsi le réseau 1.2 devient le réseau 2.3.

De plus nous considérons que la structure du réseau métabolique que nous encodons est valide, c'est-à-dire que chaque métabolite interne est au moins produit et consommé par une réaction et que chaque réaction produit et consomme au moins un métabolite. La structure du réseau est validée par une couche logicielle.

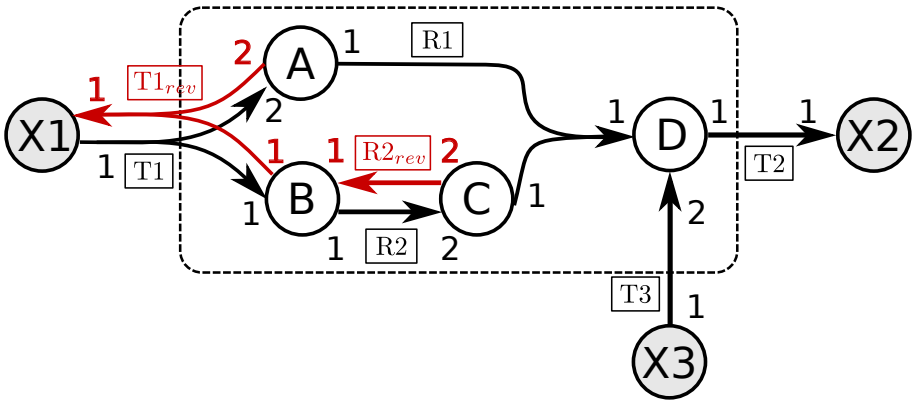


FIGURE 2.3: Hypergraphe dédoublé

Dans un premier temps nous ajoutons les contraintes assurant que nous pouvons ignorer la stœchiométrie :

1. La stœchiométrie ne doit pas être supérieure à un.

$$\forall r \in maxSto(R > 1), \neg r \tag{2.1}$$

Ce qui, une fois appliquée à notre exemple, donne :

$\neg T1$ $\neg R2_{rev}$

$\neg T1_{rev}$ $\neg T3$

$\neg R2$

2. Un métabolite interne n'est produit que par une seule réaction.

$$\forall m \in M, \bigwedge_{r,r' \in produit(m,R), r \neq r'} (\neg r \vee \neg r') \tag{2.2}$$

Ce qui, une fois appliquée à notre exemple, donne :

(B) : $\neg T1 \vee \neg R2_{rev}$ (D) : $\neg R1 \vee \neg T3$

3. Un métabolite interne n'est consommé que par une seule réaction.

$$\forall m \in M, \bigwedge_{r,r' \in consomme(m,R), r \neq r'} (\neg r \vee \neg r') \tag{2.3}$$

Ce qui, une fois appliquée à notre exemple, donne :

(A) : $\neg T1_{rev} \vee \neg R1$ (C) : $\neg R2_{rev} \vee \neg R1$

(B) : $\neg T1_{rev} \vee \neg R2$

Remarque : Pour la suite de cette section lorsque l'on parlera de voie métabolique nous supposerons que cette voie respecte les équations 2.1 à 2.3 et que l'on peut donc ignorer les problèmes liés à la stœchiométrie.

Les clauses suivantes modélisent la voie indépendamment de la stœchiométrie :

4. Une réaction réversible ne peut pas se produire en même temps que son inverse :

$$\forall r \in rev(R), r \Rightarrow \neg rev(r) \tag{2.4}$$

où $rev(R)$ est l'ensemble des réactions réversibles et $rev(r)$ une fonction qui donne la réaction inverse d'une réaction réversible. Ainsi $rev(R1) = R1_{rev}$ et $rev(R1_{rev}) = R1$

Ce qui, une fois appliquée à notre exemple, donne :

$$\neg T1 \vee \neg T1_{rev} \qquad \neg R2 \vee \neg R2_{rev}$$

5. Chaque métabolite interne actif doit avoir au moins une enzyme qui le produit activée. Ce qui nous donne la règle suivante :

$$\forall\ m \in M,\ m \Rightarrow \bigvee_{r_m \in produit(m,R)} r_m \tag{2.5}$$

Ce qui, une fois appliquée à notre exemple, donne :

$$\neg A \vee T1 \qquad \neg C \vee R2$$

$$\neg B \vee T1 \vee R2_{rev} \qquad \neg D \vee R1 \vee T3$$

6. Chaque métabolite interne actif doit avoir au moins une enzyme qui le consomme activée. Ce qui nous donne la règle suivante :

$$\forall\ m \in M,\ m \Rightarrow \bigvee_{r_m \in consomme(m,R)} r_m \tag{2.6}$$

Ce qui, une fois appliquée à notre exemple, donne :

$$\neg A \vee R1 \vee T1_{rev} \qquad \neg C \vee R2_{rev} \vee R1$$

$$\neg B \vee T1_{rev} \vee R2 \qquad \neg D \vee T2$$

7. Chaque enzyme présente dans la voie doit avoir tous ses substrats internes et tous ses produits internes actifs. Ce qui nous donne les règles suivantes :

$$\forall r \in R,\ \forall m \in \{m \in M | r \in produit(m,R)\},\ r \Rightarrow m \tag{2.7}$$

$$\forall r \in R,\ \forall m \in \{m \in M | r \in consomme(m,R)\},\ r \Rightarrow m \tag{2.8}$$

Ce qui, une fois appliquée à notre exemple, donne :

$\neg T1 \vee A$

$\neg T1 \vee B$

$\neg T1_{rev} \vee A$

$\neg T1_{rev} \vee B$

$\neg R1 \vee A$

$\neg R1 \vee C$

$\neg R1 \vee D$

$\neg R2 \vee B$

$\neg R2 \vee C$

$\neg R2_{rev} \vee C$

$\neg R2_{rev} \vee B$

$\neg T2 \vee D$

$\neg T3 \vee D$

8. Une voie doit contenir au moins une enzyme. Ce qui nous donne la règle suivante :

$$\bigvee_{r \in R} r \tag{2.9}$$

Ce qui, une fois appliquée à notre exemple, donne :

$T1 \vee T1_{rev} \vee R1 \vee R2 \vee R2_{rev} \vee T2 \vee T3$

Propriété 2.1 *Toute affectation ν satisfaisant les formules 2.1 à 2.9 modélise une voie de stœchiométrie 1 à l'état stationnaire.*

L'ensemble de ces clauses modélise une voie dans le réseau métabolique. Ainsi, s'il existe une voie, le SAT solveur la trouvera. En effet, si la voie contient une enzyme alors cela implique que tous les métabolites internes qui lui sont connectés sont également activés. De même pour tous ces métabolites internes il existe au moins une réaction qui les produit. L'activation des enzymes et des métabolites va donc se propager jusqu'à l'obtention d'une voie. On notera que la solution naïve consistant à ne prendre aucune enzyme dans la voie n'est pas valide grâce à la règle 2.9. On notera qu'une voie peut être un cycle interne.

***Preuve* : 1** *Soit ν une affectation des variables qui satisfait les formules 2.1 à 2.9. ν affecte à vrai au moins une variable encodant une réaction r (équation 2.9). De plus les variables encodant des métabolites $m_{r_{1\ldots n}}$ produits ou consommés par r seront également affectées à vrai (équations*

2.8 et 2.7). De même les variables encodant les réactions qui produisent ou consomment les métabolites $m_{r_{1...n}}$ sont également affectées à vrai (équations 2.5 et 2.6) et ainsi de suite.

Le respect de ces règles modélise une voie et puisque ν satisfait également les règles 2.1, 2.2 et 2.3) assurant que la voie est à l'état stationnaire, ν modélise bien une voie à l'état stationnaire, notre approche est donc correcte.

De façon triviale, on peut utiliser la propriété 2.1 pour montrer que si un réseau métabolique n'admet pas de voie à l'état stationnaire de stœchiométrie de 1 alors le SAT solveur trouvera UNSAT.

Propriété 2.2 *L'encodage d'une voie à l'état stationnaire de stœchiométrie de 1 dans un réseau métabolique respecte les équations 2.1 à 2.9.*

Puisque les équations 2.1 à 2.9 servent à définir une voie, trivialement une voie est valide vis-à-vis de ces équations.

2.1.2 Calcul d'un sur-ensemble des voies minimales

L'utilisation d'un SAT solveur nous permet, pour l'instant, de calculer une voie. Cependant, nous souhaitons connaître l'ensemble des voies minimales contenues dans le réseau métabolique. Pour cela nous allons simplement interroger le SAT solveur selon l'algorithme 4.

Pour que la formule soit satisfaite il faut qu'elle satisfasse toutes les clauses, que ce soit celles qui modélisent le réseau ou celles que l'algorithme ajoute. La ligne 7 nous assure que la solution actuelle ne respecte pas cette nouvelle clause, on remarque que toutes les solutions incluant la voie actuelle ne respectent également pas cette clause.

Propriété 2.3 *Soit ν_{v_i} une assignation satisfaisant une formule f, modélisant un réseau métabolique valide, encodant une voie v_i, et $getReaction(\nu_{v_i}, true)$l'ensemble des variables encodant des réactions affectées à vrai par ν_{v_i}. L'ajout d'une clause $c_{v_i} = \bigvee_{r \in getReaction(\nu_{v_i}, true)} \neg r$ à f*

Algorithm 4 CalculeSurEnsembleDesEFMsSansStœchiométrie
Output : un ensemble de voies sans stœchiométrie *sol* tel que toutes les voies minimales soient incluses dans *sol*

1: $formule = initialise()$ % Réalise l'encodage décrit dans les équations 2.1 à 2.9
2: $sol = \{\}$
3: **while** (SAT.check(formule) == true) **do**
4: % $getReaction(SAT.getSolution()), true)$ l'ensemble des variables encodant des réactions affectées à vrai par SAT dans la solution
5: $solution = getReaction(SAT.getSolution()), true)$
6: $sol = sol \cup \{\text{solution}\}$
7: $formule+ = \bigvee_{r \in solution} \neg r$ % On bloque la solution
8: **end while**
9: **return** *sol*

invalide ν_{v_i} pour v_i mais n'invalide aucune affectation ν_{efm_j} telle que $v_i \neq efm_j$ où ν_{efm_j} est une assignation encodant un EFM.

***Preuve* : 2** *Par construction l'affectation ν_{v_i} ne satisfait pas la clause c_{v_i}, ν_{v_i} n'est donc plus un modèle de f.*

Il faut maintenant montrer d'une part que

1. *si v_i n'est pas un EFM alors l'ajout de c_{v_i} à f ne bloque aucun EFM ;*
2. *si v_i est un EFM l'ajout de c_{v_i} à f ne bloque aucun autre EFM.*

Pour le point 2 puisqu'un EFM est minimal au sens de l'inclusion, pour toute paire d'EFMs efm_1, efm_2 telle que $efm_1 \neq efm_2$, il existe une réaction r tel que $r \in efm_1$, $r \notin efm_2$. Donc si f_i est un EFM la contrainte c_{f_i} est trivialement satisfaite, pour tout EFM différent de f_i, par au moins un r de cet EFM qui n'appartient pas à f_i.

Pour le point 1, puisque v_i n'est pas un EFM il est donc une combinaison d'EFM $e = \{efm_1, ...efm_n\}$ avec $n > 1$. On peut donc, comme précédemment, trouver pour tout $efm_i \in e$ une réaction de v_i qui ne soit pas dans EFM_i. L'ajout de c_{v_i} n'invalidera donc aucune ν_{efm_i}.

L'ajout de c_{v_i} ne bloque donc aucun EFM efm_j tel que $efm_j \neq v_i$.

Grâce à la propriété 2.3 et au fait que le nombre d'EFMs est fini, notre algorithme se terminera lorsqu'ils auront tous été trouvées.

Lorsque le SAT solveur ne trouvera plus de solution cela impliquera que tous les voies minimales auront été trouvées ainsi qu'un certain nombre de voies non minimales, en effet lorsqu'on trouve une solution non minimale rien n'empêche le SAT solveur de trouver une solution incluse dans celle-ci. Le filtrage de ces voies non minimales sera abordé dans la suite.

Propriété 2.4 *L'algorithme 4 retourne un ensemble fini de voies contenant l'ensemble des $EFMs$ sans stœchiométrie.*

***Preuve* : 3** *Un réseau métabolique valide contient un ensemble fini de voies $F = \{f_1, ..., f_n\}$ éventuellement vide. Or :*

- *La propriété 2.2 nous assure que le SAT solveur trouvera une solution si $n > 0$.*
- *La propriété 2.3 nous assure que bloquer un flux f_i ne bloque aucun EFM efm_j tel que $efm_j \neq f_i$.*

De plus puisque le SAT solveur retournera UNSAT si $n = 0$ et puisqu'à chaque itération nous bloquons au moins une solution le SAT solveur retournera UNSAT une fois toutes les solutions bloquées.

L'algorithme 4 retourne donc un ensemble fini de voies contenant l'ensemble des $EFMs$ sans stœchiométrie, ainsi que des voies non minimales.

Par souci de clarté, dans la suite, une solution sera représentée uniquement par les variables encodant les enzymes ayant une valeur strictement positive dans la solution. Ainsi la solution {R5, R4} désigne l'affectation des variables {A=$\bot$, B=$\bot$, C=$\bot$, D=$\top$, T1=$\bot$, T1$_{rev}$=$\bot$, R1=$\bot$, R2=$\bot$, R2$_{rev}$=$\bot$, T2=$\top$, T3=$\top$}.

Par exemple, dans notre cas, si le SAT solveur trouve comme solution la voie {T2, T3} la clause $\neg T2 \vee \neg T3$ sera ajoutée à la formule.

2.1.3 Calcul des EFMs sans prise en compte de la stœchiométrie

L'algorithme 4 considère une voie non minimale comme une solution. Notre but étant, in fine, de calculer des EFMs, nous devons inclure la notion de minimalité dans le calcul de nos voies.

Si nous avions un moyen de vérifier qu'une solution est minimale il serait simple de lister les voies en modifiant légèrement l'algorithme précédent.

Algorithm 5 CalculeEFMsSansStœchiométrie
Output : l'ensemble *sol* des voies minimales sans stœchiométrie

1: $formule = initialise()$
2: $sol = \{\}$
3: **while** (SAT.check(formule) == true) **do**
4: $solution = getReaction(SAT.getSolution()), true)$
5: **if** estMinimale($solution, formule$) **then**
6: $sol = sol \cup \{\text{solution}\}$
7: **end if**
8: $formule+ = \bigvee_{r \in solution} \neg r$
9: **end while**
10: **return** *sol*

Il s'agit simplement de ne pas considérer une voie non minimale comme une solution ainsi on supprimera toujours tous les sur-ensembles de la solution SAT qu'elle soit minimale ou non.

La méthode « estMinimale » se base sur un appel SAT permettant de prouver qu'il n'existe pas de solution incluse dans la solution actuelle, cette méthode a été initialement utilisée dans (Koshimura et al., 2009).

Algorithm 6 estMinimale(solution,formule)
Input : *solution* une affectation satisfaisant $formule$.
Ouput : un booléen indiquant si *solution* est minimale au sens de l'inclusion.

1: $active = getReactions(solution, true)$
2: $nonActive = getReactions(solution, false)$
3: $assertion = \bot$
4: $formule+ = \bigwedge_{r \in nonActive} assertion \vee \neg r$
5: $formule+ = assertion \vee \bigvee_{r \in active} \neg r$
6: $valeurRetour = SAT.check(formule)$
7: $assertion = \top$
8: **return** $valeurRetour$

Afin que les clauses ajoutées dans cet algorithme ne perturbent pas le reste de la recherche nous ajoutons aux clauses un littéral d'hypothèse, ce qui nous permet de désactiver toutes les clauses ayant ce littéral à la fin de l'algorithme.

Bien que cet algorithme soit parfaitement valide, il n'est pas optimal en terme de performance, nous allons voir pourquoi et comment l'optimiser.

2.1.4 Optimisation de la minimisation

Supposons une voie minimale $f = \{R1, R2\}$ et trois réactions, de R3 à R5, telles que toute combinaison entre f et ces réactions soit une voie valide. L'exécution du solveur peut être :

$\{R1, R2, R3, R4, R5\}$ => non minimale.

$\{R1, R2, R3, R4\}$ => non minimale.

$\{R1, R2, R3, R5\}$ => non minimale.

$\{R1, R2, R4, R5\}$ => non minimale.

$\{R1, R2, R3\}$ => non minimale.

$\{R1, R2, R5\}$ => non minimale.

$\{R1, R2, R4\}$ => non minimale.

$\{R1, R2\}$ => minimale.

Pour contrer ce phénomène, nous ajoutons la méthode « minimiseSolution » à notre algorithme afin de nous assurer que pour chaque solution que nous trouvons, nous bloquons bien une voie minimale. En faisant ainsi nous maximisons les sur-ensembles que nous bloquons en même temps.

L'idée de l'algorithme 7 (également issue de (Koshimura et al., 2009)) est de forcer le SAT solveur à trouver une solution qui contient au moins une réaction de moins. Pour cela nous bloquons les réactions qui n'étaient pas présentes dans la solution initiale, afin que le solveur ne trouve pas une solution complètement différente, puis nous ajoutons une contrainte assurant qu'au moins une des réactions qui était active ne le soit plus. Nous répétons ces étapes jusqu'à ce que le SAT solveur ne trouve plus de solution, ce qui indiquera que la solution en cours est minimale.

Grâce à l'algorithme 7 nous pouvons transformer l'algorithme 5 en l'algorithme 8.

Si nous appliquons l'algorithme 8 sur l'exemple précédent le pire des cas devient :

Algorithm 7 minimiseSolution(solution,formule)
Input : *solution* une affectation satisfaisant $formule$.
Ouput : un ensemble de variables (assignées à vrai dans *solution*) tel que si ces variables sont affectées à vrai elles forment une solution minimale de $formule$.

$active = getReactions(solution, true)$
$nonActive = getReactions(solution, false)$
$assertion = \bot$
$formuleMinimisation = formule$
$formuleMinimisation+ = \bigwedge_{r \in nonActive} assertion \vee \neg r$
$formuleMinimisation+ = assertion \vee \bigvee_{r \in active} \neg r$
$solutionMin = solution$
while (SAT.check(formuleMinimisation) = true) **do**
  $solutionMin = getReaction(SAT.getSolution()), true)$
  $active = getReactions(solutionMin, true)$
  $nonActive = getReactions(solutionMin, false)$
  $formuleMinimisation = formule$
  $formuleMinimisation+ = \bigwedge_{r \in nonActive} assertion \vee \neg r$
  $formuleMinimisation+ = assertion \vee \bigvee_{r \in active} \neg r$
end while
$assertion = \top$
return $solutionMin$

Algorithm 8 CalculeEFMsSansStœchiométrie
Output : l'ensemble *sol* des voies minimales sans stœchiométrie

$formule = initialise()$
$sol = \{\}$
while SAT.check(formule) == true) **do**
  $solution = getReaction(SAT.getSolution()), true)$
  $solution = minimiseSolution(solution, formule)$
  $sol = sol \cup \{\text{solution}\}$
  $formule+ = \bigvee_{r \in solution} \neg r$
end while
return sol

$\{R1, R2, R3, R4, R5\}$ => non minimale.

$\{R1, R2, R3, R4\}$ => non minimale.

$\{R1, R2, R3\}$ => non minimale.

$\{R1, R2\}$ => minimale.

Insatisfaisable.

Cette solution nous permet donc de passer d'un nombre d'appels au SAT solveur égal, dans le pire des cas, à l'exponentielle du nombre de réactions dans la solution initiale qui ne feront pas partie de l'EFM à un nombre d'appels égal à la taille de la solution initiale moins la taille de l'EFM plus un, le dernier appel servant à prouver que la solution finale est minimale.

2.1.5 Les requêtes

L'un des principaux avantages de notre modélisation est que l'on peut facilement intégrer des requêtes en logique propositionnelle pour filtrer les voies. En effet, puisque les réactions et les métabolites sont modélisés par des variables logiques, ajouter une requête booléenne équivaut à ajouter une ou plusieurs clauses dans la formule logique initiale. Ainsi toutes les requêtes exprimables en logique propositionnelle seront traitées directement par le solveur et l'ensemble des voies retournées par notre algorithme respecteront ces requêtes.
Il s'agit d'une caractéristique dont ne disposent pas les méthodes basées sur la double description, en effet celles-ci n'intègrent dans leurs calculs qu'un certain type de requête. Par exemple une requête de la forme « $R1 \vee \neg R2$ » sera examinée en « poste-traitement » par RegEfmTool ce qui implique qu'une partie du temps de calcul a été utilisée pour calculer des solutions qui seront éliminées par la suite.

La prise en compte de requêtes influe sur la définition de minimalité. Puisque la minimalité est actuellement assurée par le SAT solveur, si celui-ci contient des requêtes cela va changer la définition de minimalité. En effet l'algorithme 7 cherche une solution minimale respectant toutes les contraintes de la formule, requête y compris. Ainsi, la solution qu'elle retourne peut donc ne pas être un EFM. Ce point sera abordé plus en détail dans le chapitre 4.

La méthode « estUnEFM » doit donc assurer la minimalité sans tenir compte des requêtes qui ont potentiellement été ajoutées à la formule. L'idée va donc être de ne pas simplement ajouter une contrainte c mais d'ajouter $c \vee \neg assert_{requete}$. Ainsi lors de la recherche nous affecterons $assert_{requete}$ à vrai, forçant le solveur à satisfaire c comme nous le souhaitons. À l'inverse lors de la minimisation nous affecterons $assert_{requete}$ à faux satisfaisant ainsi toutes les clauses encodant les requêtes.

L'algorithme 7 devient donc 9.

Algorithm 9 minimiseSolution(solution,formule)
Input : *solution* une affectation satisfaisant $formule$.
Ouput : un ensemble de variables (assignées à vrai dans *solution*) tel que si ces variables sont affectées à vrai elles forme une solution de $formule$ modélisant un EFM.

1: $assert_{requete} = \bot$
2: $assertion = \bot$
3: $active = getReactions(solution, true)$
4: $nonActive = getReactions(solution, false)$
5: $formuleMinimisation = formule$
6: $formuleMinimisation+ = \bigwedge_{r \in nonActive} assertion \vee \neg r$
7: $formuleMinimisation+ = assertion \vee \bigvee_{r \in active} \neg r$
8: $solutionEFM = solution$
9: **while** (SAT.check(formuleMinimisation) = true) **do**
10: $solutionEFM = getReaction(SAT.getSolution()), true)$
11: $active = getReactions(solutionEFM, true)$
12: $nonActive = getReactions(solutionEFM, false)$
13: $formuleMinimisation = formule$
14: $formuleMinimisation+ = \bigwedge_{r \in nonActive} assertion \vee \neg r$
15: $formuleMinimisation+ = assertion \vee \bigvee_{r \in active} \neg r$
16: **end while**
17: $assertion = \top$
18: $assert_{requete} = \bot$
19: **return** $solutionEFM$

Pour des raisons de performance l'algorithme 9 n'est pas implémenté dans notre outil, en effet on lui préfère une méthode basée sur la matrice de stœchiométrie qui sera abordée dans la section 2.2.

Conclusion Nous avons présenté un algorithme permettant de calculer l'ensemble des EFMs n'utilisant pas de stœchiométrie. Cependant notre objectif est de calculer tous les EFMs pas un sous-ensemble.

Nous allons maintenant voir comment relâcher les contraintes que nous avons imposées ainsi que les modifications que cela implique sur notre algorithme.

2.2 Prise en compte de la stœchiométrie

La prise en compte de la stœchiométrie ne pouvant pas se faire au niveau de la formule SAT, nous allons garder l'algorithme qui nous permet de calculer des voies minimales et nous allons rajouter une sur-couche logicielle permettant d'assurer le respect de la stœchiométrie.

Une fois une solution minimale trouvée, nous calculons une base du noyau de la matrice de stœchiométrie réduite aux réactions actives dans cette solution. Cette méthode se base sur :

— les résultats de (Klamt et al., 2005) qui démontrent que la dimension du noyau de la matrice de stœchiométrie d'un EFM (c'est-à-dire la sous-matrice de stœchiométrie définie par les colonnes correspondant aux réactions constituant l'EFM) est de un ;

— le fait que nous n'avons que des réactions irréversibles (les réactions réversibles ayant été divisées en deux réactions irréversibles).

Un vecteur de base du noyau d'une voie minimale peut être strictement positif ou strictement négatif mais il ne peut ni contenir de zéro, sans quoi on pourrait supprimer au moins une réaction tout en restant à l'état stationnaire, ni contenir des valeurs positives et négatives car cela signifierait qu'une réaction irréversible est utilisée dans le « mauvais sens ».

Grâce à ce calcul, nous pouvons déduire des informations sur la solution proposée par le SAT solveur. L'algorithme 10 présente les différents cas, nous les détaillerons plus précisément dans la suite.

L'algorithme 10 a donc trois retours possibles :

1. **SOLUTION_VALIDE :** La dimension du noyau est de un et le vecteur de base ne contient que des valeurs strictement positives ou que des valeurs strictement négatives, la stœchiométrie est respectée ainsi que la minimalité : la solution est un EFM. On l'ajoute à l'ensemble des EFM et on la bloque comme précédemment.

Algorithm 10 informationSolution(solution,formule)
Input : *solution* une affectation satisfaisant $formule$.
Ouput : une énumération pouvant avoir comme valeur SOLUTION_VALIDE, SOLUTION_INCOMPLÈTE et SOLUTION_INVALIDE

```
reactions = getReaction(solution, true)
matrice = getMatriceStoechiometrique(reactions)
noyau = kernel(matrice)
if (dim(noyau) == 1) AND (min(noyau) × max(noyau) > 0) then
    return SOLUTION_VALIDE
else if (dim(noyau) == 0) then
    return SOLUTION_INCOMPLÈTE
else
    return SOLUTION_INVALIDE
end if
```

2. **SOLUTION_INCOMPLÈTE :** Le noyau est réduit à $\{0\}$ pour cette solution. C'est par exemple le cas du réseau métabolique présenté en figure 2.1 : la stœchiométrie ne permet pas d'être à l'équilibre.

   Ce cas est problématique. En effet, on ne peut pas bloquer la solution comme on la bloquait précédemment : rajouter une ou plusieurs réactions peut permettre de créer un EFM contenant cette voie. Nous allons donc bloquer uniquement cette solution en laissant la possibilité au SAT solveur de trouver une solution incluant celle-là. Pour faire cela on ajoute la clause suivante :

$$\bigvee_{r1 \in getReaction(solution,true)} \neg r1 \vee \bigvee_{r2 \in getReaction(solution,false)} r2 \tag{2.10}$$

   Ainsi toute solution, incluant strictement la solution, sera valide pour le SAT solveur.

   Remarque : l'ajout de cette formule a également un impact sur la minimisation puisque celle-ci se base sur la satisfaisabilité de la formule. Ainsi la minimalité d'une voie ne s'exprime plus comme la non inclusion d'une autre voie mais comme la non inclusion d'une autre voie valide.

3. **SOLUTION_INVALIDE :** Deux cas peuvent se présenter :

   Le premier est le cas où la dimension du noyau est supérieure à un, il s'agit du cas que nous avons évoqué lors de la sous-section 2.1.5, la minimalité telle que l'entend la définition des

EFMs n'est pas respectée et nous pouvons bloquer la solution. Dans ce cas, la solution trouvée par le SAT solveur est un ensemble d'EFMs, composé d'EFMs ne respectant pas la requête, tels que l'ensemble la satisfasse. Cette problématique sera abordée plus en détail dans le chapitre 4.

Le second cas est lorsqu'un vecteur de la base du noyau contient des valeurs strictement positives et négatives. Cela se produit lorsque la seule possibilité pour rendre une voie valide avec la stœchiométrie est d'utiliser une réaction à contre sens.

Un bon exemple est le cas suivant :

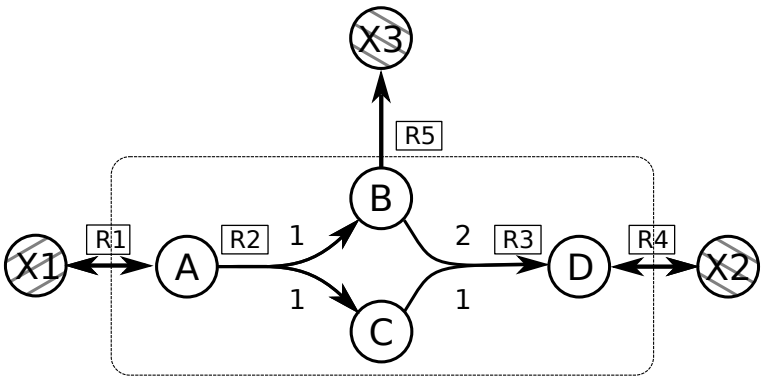


FIGURE 2.4: Réseau métabolique ayant un vecteur du noyau contenant des facteurs positifs et négatifs

Ici le manque de production de B est compensé dans le noyau par un facteur négatif sur la réaction R5.

Les solutions comprenant des réactions en « contre sens » et tous leurs sur-ensembles peuvent être bloqués de la manière vue précédemment.

La raison pour laquelle on peut bloquer également les sur-ensembles de ces solutions est due au fait que la solution que l'on trouve est minimale en terme de « solution valide ». Cela implique que chaque réaction dans la solution apporte quelque chose au réseau, soit une production soit une consommation de métabolite. Ainsi un sur-ensemble peut être une voie valide. Par exemple nous pouvons étendre l'exemple 2.4 en la figure 2.5.

Cependant, cette voie ne peut pas être minimale. En effet la réaction R5 n'apporte rien au réseau ainsi une fois le réseau complété afin de tenir compte de la stœchiométrie, il

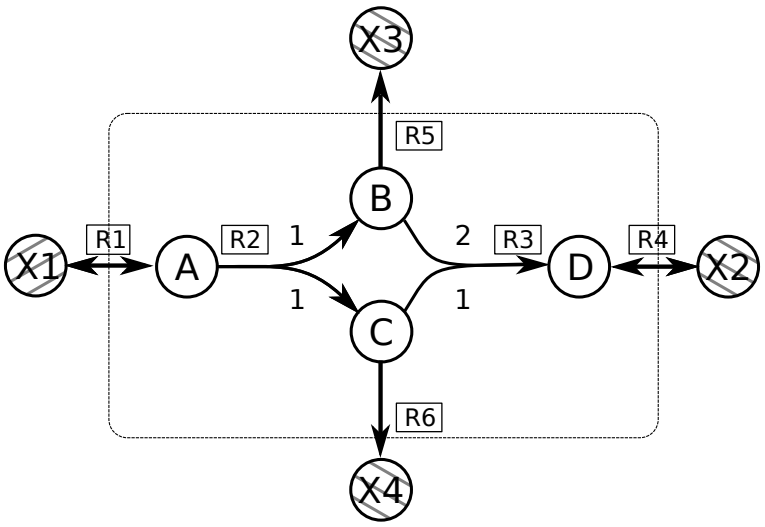


FIGURE 2.5: Sur-ensemble du réseau présenté figure 2.4

existera un sous-ensemble de ce réseau ne contenant pas R5 qui sera une voie valide.

Ainsi tout sur-ensemble d'une solution ayant un vecteur du noyau contenant des facteurs positifs et négatifs est soit invalide soit non minimal, on peut donc le bloquer sans affecter les EFMs du réseau.

Avec cette approche nous forçons le SAT solveur à respecter la stœchiométrie en bloquant les voies ne la respectant pas et nous pouvons ainsi énumérer tous les EFMs d'un réseau métabolique. L'algorithme 11 présente le calcul des EFMs en prenant en compte la stœchiométrie tel que nous venons de le voir.

Idée de preuve La démonstration rigoureuse de cet algorithme serait longue, nous allons donc présenter uniquement les grandes lignes de celle-ci.

Correction : On ajoute une solution à notre ensemble d'EFMs uniquement lorsque *informationSolution(solution,formule) == SOLUTION_VALIDE* ce qui revient à simplement appliquer les résultats de (Klamt et al., 2005) qui nous assurent que la solution est bien un EFM.

Complétude : L'idée est de montrer que les clauses que l'on ajoute à la formule ne supprimeront pas d'EFMs qui n'ont pas déjà été trouvés. Nous avons donc quatre cas :

1. On a trouvé une solution valide. On la bloque et la propriété 2.3 nous assure que nous ne

Algorithm 11 CalculeEFMs
Output : l'ensemble *sol* des voies minimales avec stœchiométrie

```
1: formule = initialise()
2: sol = {}
3: while (SAT.check(formule) == true ) do
4:    solution = getReaction(SAT.getSolution()), true)
5:    solution = minimiseSolution(solution, formule)
6:    if informationSolution(solution,formule) == SOLUTION_VALIDE then
7:       sol = sol ∪ {solution}
8:       formule+ = ⋁_{r∈solution} ¬r
9:    else if informationSolution(solution,formule) == SOLUTION_INCOMPLÈTE then
10:      nonActives = SAT.getEnzymesNonActives(formule)
11:      nonActives = getReaction(SAT.getSolution()), false)
12:      formule+ = ⋁_{r1∈solution} ¬r1 ⋁_{r2∈nonAcitves} r2
13:   else
14:      formule+ = ⋁_{r∈solution} ¬r
15:   end if
16: end while
17: return sol
```

bloquons pas d'autres EFMs.

2. On a trouvé une solution non complète. L'équation 2.10 ne bloque exactement que cette solution, et puisque celle-ci n'est pas l'encodage d'un EFM la bloquer ne gêne pas la complétude de l'algorithme.

3. La solution est minimale pour le SAT solveur mais pas pour le calcul matriciel. Le SAT solveur a trouvé une solution minimale pour l'inclusion de solution valide, mais cette solution est composée de plusieurs EFMs, on se retrouve dans le cas où l'on bloque un ensemble d'EFMs. la propriété 2.3 nous assure que nous ne bloquons pas d'EFMs.

4. La solution est une voie, mais pour que celle-ci soit à l'état stationnaire il faut qu'une de ses réactions soit utilisée « en contre sens ». Il ne s'agit donc pas d'un EFM et elle peut donc être bloquée. Il faut cependant montrer que aucun sur-ensemble de cette voie ne peut être un EFM. Supposons que la réaction r_{cs} soit utilisée à contre sens dans le calcul matriciel, cela veut dire que, utilisée dans le sens normal, elle ne participe pas à mettre le réseau à l'état stationnaire, au contraire, elle va produire des métabolites en trop, plus précisément elle va sur-produire des métabolites $m_{surplus}$ qui étaient déjà en sur-production. Pour arriver à l'état stationnaire, il va falloir ajouter une ou plusieurs réactions pour consommer $m_{surplus}$. Ainsi si nous arrivons à une voie v à l'état stationnaire,

celle-ci ne sera pas minimale car on pourra trouver une voie $v' = v - r_{cs}$. Nous pouvons donc bloquer cette solution sans bloquer d'EFMs.

L'une des principales faiblesses de cet algorithme réside dans le cas où nous trouvons une solution non complète. En effet du point de vue du SAT solveur, cette solution correspond parfaitement aux contraintes, cependant puisque le calcul matriciel informe que la stœchiométrie n'est pas respectée nous allons demander au solveur, via l'équation 2.10, d'ajouter au moins une réaction avant de revérifier la stœchiométrie. Malheureusement le solveur n'a pas d'indice pour guider la recherche et va donc juste ajouter une réaction au hasard jusqu'à épuisement des possibilités. En effet, même s'il trouve une solution il devra quand même vérifier toutes les autres possibilités afin de s'assurer de ne pas en oublier.

Nous allons maintenant présenter les améliorations que nous avons apportées à notre algorithme.

2.3 Améliorations de l'algorithme

Nous proposons différentes améliorations qui, bien que ne modifiant pas profondément le fonctionnement de l'algorithme, nous permettent de l'optimiser ou de rendre plus pertinents les résultats retournés lorsque l'exécution est interrompue avant la fin de l'énumération.

Phase sans SMT

Une première solution, pour limiter les problèmes de combinaisons, est de calculer les EFMs en deux étapes.

Dans un premier temps nous calculons les EFMs qui n'ont pas besoin de prendre la stœchiométrie en compte, comme nous l'avons fait dans l'algorithme 8. Puis nous calculons les EFMs restants. En faisant ainsi nous pouvons trouver rapidement les EFMs simples à calculer.

Afin de pouvoir faire la transition entre l'étape avec et sans stœchiométrie nous devons introduire une variable $assert_{SanStoechio}$ dans les équations 2.1, 2.2 et 2.3 afin de pouvoir les désactiver

lors du changement d'étape. Ces équations deviennent donc respectivement les équations 2.11, 2.12 et 2.13.

$$\forall r \in maxSto(R > 1), \neg r \vee \neg assert_{SanStoechio} \tag{2.11}$$

$$\forall m \in M, \bigwedge_{r,r' \in produit(m,R), r \neq r'} (\neg r \vee \neg r' \vee \neg assert_{SanStoechio}) \tag{2.12}$$

$$\forall m \in M, \bigwedge_{r,r' \in consomme(m,R), r \neq r'} (\neg r \vee \neg r' \vee \neg assert_{SanStoechio}) \tag{2.13}$$

Lors de l'étape sans stœchiométrie nous affectons $assert_{SanStoechio}$ à vrai, ce qui nous permet de retrouver les équations initiales. À l'inverse lorsque l'on souhaite prendre en compte la stœchiométrie on affecte $assert_{SanStoechio}$ à faux, ainsi les équations sont satisfaites grâce à la présence de $\neg assert_{SanStoechio}$ et les contraintes ne seront donc pas prises en compte dans la recherche de solution.

2.3.1 Recherche des solutions par taille

Le fait que notre solution se base sur un SAT solveur nous permet de modifier la fonction interne de celui ci, ce qui nous donne la possibilité d'ajouter des options telles que trouver les EFMs les plus courts en premier.

L'idée est de trouver toutes les solutions de taille deux, de démontrer qu'il n'en existe pas d'autre, puis d'incrémenter la taille et de recommencer. Puisque nous devons gérer un compteur ce problème ne s'exprime pas facilement en logique propositionnelle. Nous avons donc modifié le comportement du SAT solveur afin que celui-ci ne cherche pas de solution de taille supérieure à une taille maximale. Cependant, nous voulions faire cela de façon transparente pour le SAT solveur, c'est-à-dire sans perturber les mécanismes de propagation et de choix de littéraux.

Le fonctionnement est donc le suivant :
Lorsque le SAT solveur affecte une réaction à vrai, on incrémente le compteur de réactions actives. Tant qu'il est inférieur à la taille maximale, on ne change pas le comportement du

solveur, lorsqu'il atteint la taille maximale, nous savons qu'il ne peut pas y avoir d'autre enzyme active en plus, nous pouvons donc propager toutes les enzymes actuellement non définies à faux. Pour que cette propagation se fasse de façon transparente pour le SAT solveur il faut indiquer la raison de cette propagation. Nous allons donc créer une clause, que nous appellerons « ClauseVirtuelle » contenant toutes les enzymes actuellement actives, ainsi lors de la phase de « backtrack » le SAT solveur aura l'information qu'il n'a pas pu activer cette enzyme à cause des enzymes déjà activées.

On notera que nous n'avons besoin que d'une seule « ClauseVirtuelle », en effet puisque le fait d'avoir atteint la borne déclenche un conflit, le SAT solveur va faire un « retour arrière » (backtrack) et toutes les enzymes que nous avions affectées à faux à cause de la taille vont à nouveau être indéfinies. La « ClauseVirtuelle » n'aura donc plus d'usage et pourra être écrasée par la prochaine contrainte de taille.

Algorithm 12 CalculeEFMsTailleMax(tailleMax)
Input : La taille maximale des EFMs
Output : l'ensemble *sol* des voies minimales avec stœchiométrie de taille $tailleMax$ ou moins.

```
1: formule = initialise()
2: sol = {}
3: taille = 2
4: while (taille ≤ tailleMax) do
5:    while (SAT.checkWithTaille(formule,taille) == true ) do
6:       solution = getReaction(SAT.getSolution()), true)
7:       solution = minimiseSolution(solution, formule)
8:       if informationSolution(solution,formule) == SOLUTION_VALIDE then
9:          sol = sol ∪ {solution}
10:         formule+ = ⋁_{r∈solution} ¬r
11:      else if informationSolution(solution,formule) == SOLUTION_INCOMPLÈTE then
12:         nonAcitves = SAT.getEnzymesNonActives(formule)
13:         nonAcitves = getReaction(SAT.getSolution()), false)
14:         formule+ = ⋁_{r1∈solution} ¬r1 ⋁_{r2∈nonAcitves} r2
15:      else
16:         formule+ = ⋁_{r∈solution} ¬r
17:      end if
18:   end while
19:   taille+ = 1
20: end while
21: return sol
```

L'algorithme 12 présente le calcul des EFMs par taille croissante. La méthode « checkWithTaille » utilise la « ClauseVirtuelle » décrite ci-dessus afin d'assurer la limite de taille. Dans

l'implémentation de cet algorithme, nous pouvons détecter si toutes les solutions ont été trouvées ce qui évite de devoir démonter inutilement UNSAT pour les tailles suivantes. Pour faire cela nous regardons ce qui a causé UNSAT : si la « ClauseVirtuelle », donc la taille, n'est pas impliquée dans UNSAT, nous avons trouvé toutes les solutions et nous pouvons arrêter la recherche.

2.4 Résultats expérimentaux

La gestion de la stœchiométrie telle que nous l'avons décrite pose des problèmes de performance, en effet, lorsque nous trouvons une solution invalide nous empêchons de retrouver cette solution dans le futur mais nous n'orientons pas la recherche vers une solution valide. Encore pire, dans le cas d'une solution incomplète, nous bloquons uniquement la solution en cours, ce qui veut dire que dans le pire des cas le solveur peut tester l'ajout de toutes les combinaisons d'enzymes afin de trouver une solution valide, ce qui rend le passage à l'échelle clairement prohibitif.

Les expériences sont réalisées sur une version réduite du réseau métabolique de l'arabidopsis (Jungreuthmayer et al., 2013a) contenant environ 1 500 000 EFMs. Pour des raisons de cohérence avec les expériences des sections suivantes, le réseau est compressé grâce à RegEfmTool, la compression permet de représenter le réseau de façon plus compacte sans en modifier les EFMs, dans notre cas la compression a transformé un réseau ayant 82 réactions et 72 métabolites en un réseau ayant 58 réactions et 25 métabolites.

Les requêtes traitées se divisent en trois parties :

— Les requêtes négatives unitaire, il s'agit de requêtes de la forme $\neg a$ interdisant la présence d'une ou plusieurs réactions dans le réseau.

— Les requêtes négatives de longueur deux de la forme $\neg a \vee \neg b$, seront appelées « requêtes disjonctives ».

— Les requêtes positives unitaire, il s'agit de requêtes forçant la présence d'une ou plusieurs réactions dans le réseau.

Nous souhaitons avoir des requêtes satisfaites par un nombre croissant d'EFMs afin de voir comment notre solution passe à l'échelle. Pour faire cela, nous calculons l'ensemble des EFMs grâce à RegEfmTool. Puis nous sélectionnons aléatoirement une réaction que nous ajoutons à notre requête et nous évaluons la requête. Nous réitérons le processus jusqu'à ce que la requête soit satisfaite par moins de 5 000 EFMs. La limite supérieure a été fixée à 5 000 EFMs car nous estimons qu'au delà de 5 000 les résultats deviennent illisibles pour un humain et celui-ci aura tendance à spécifier d'avantage sa requête. En continuant à contraindre notre requête, nous engendrons ainsi des requêtes satisfaites par un nombre d'EFMs entre 0 et 5 000. De plus lorsque nous avons sélectionné les requêtes, nous avons fait en sorte qu'elles soient, autant que possible, bien réparties entre 0 et 5 000 afin que l'on puisse suivre l'évolution des performances.

Les exécutions ont était faites sur un serveur bi-pro Intel® Xeon® E5-2609v2 2.5GHz ,8 Core, 64 Go, ubuntu 64 bits.

Dans un premier temps nous allons regarder comment se répartit le temps entre la partie qui analyse si la solution est bonne, la partie qui gère la minimisation et la partie de recherche de solution.

Les figures 2.6, 2.7 et 2.8 présentent ces temps en secondes pour les requêtes positives, négatives et disjonctives. Les résultats présentés correspondent à des exécutions utilisant une étape sans stœchiométrie puis listant les résultats par taille croissante où timeout est fixé à une heure. Ces figures sont issues des tableaux de l'annexe A.

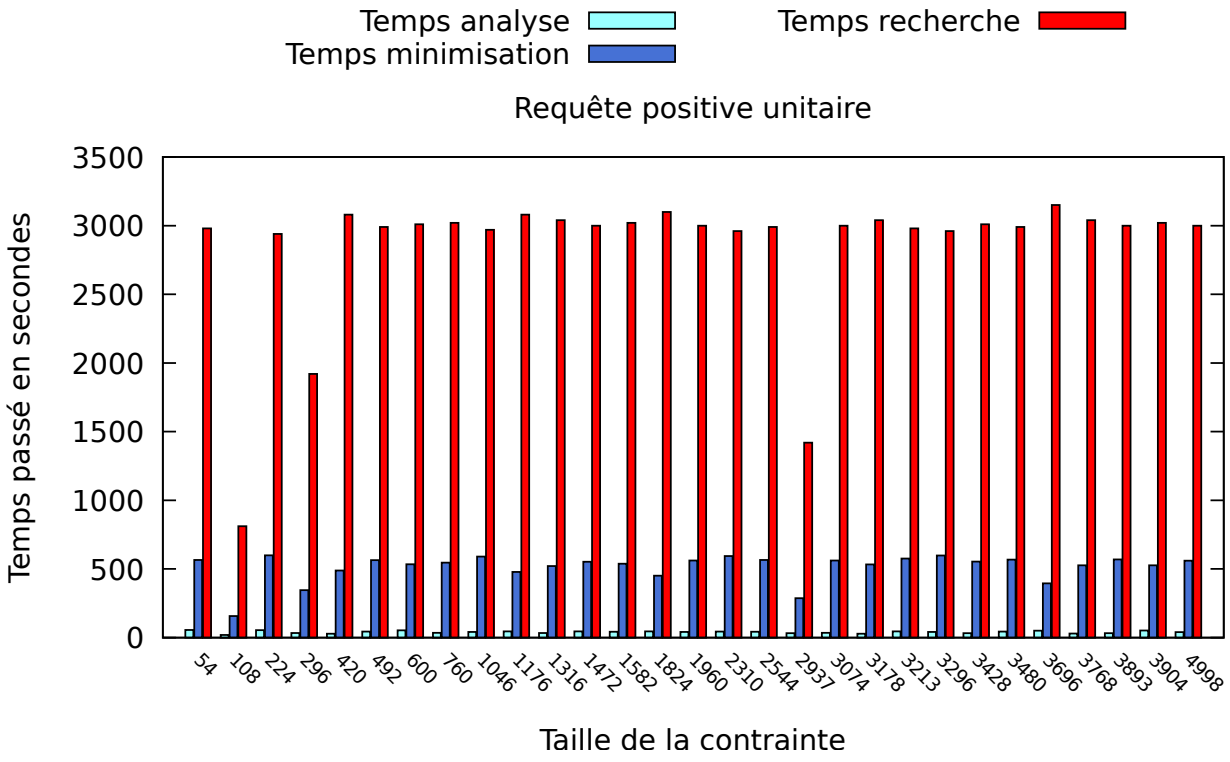


FIGURE 2.6: Temps en seconds pour les requêtes positives unitaires

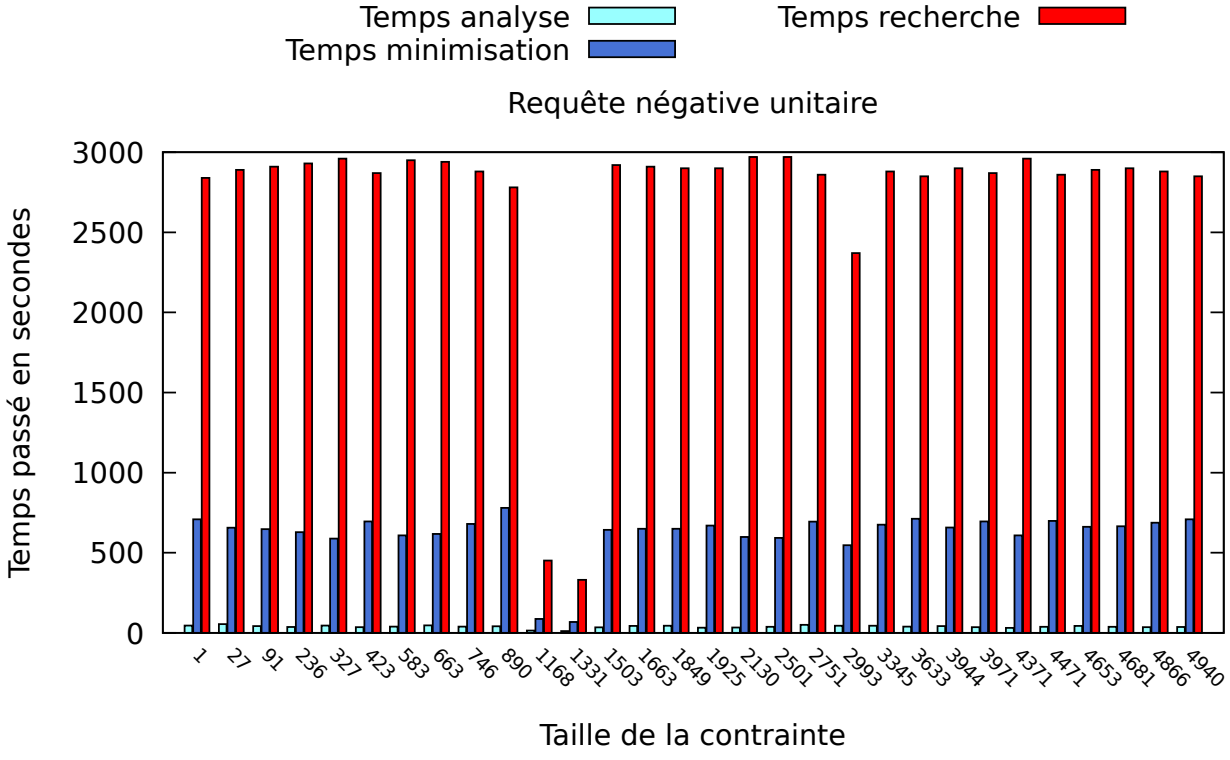


FIGURE 2.7: Temps en seconds pour les requêtes négatives unitaires

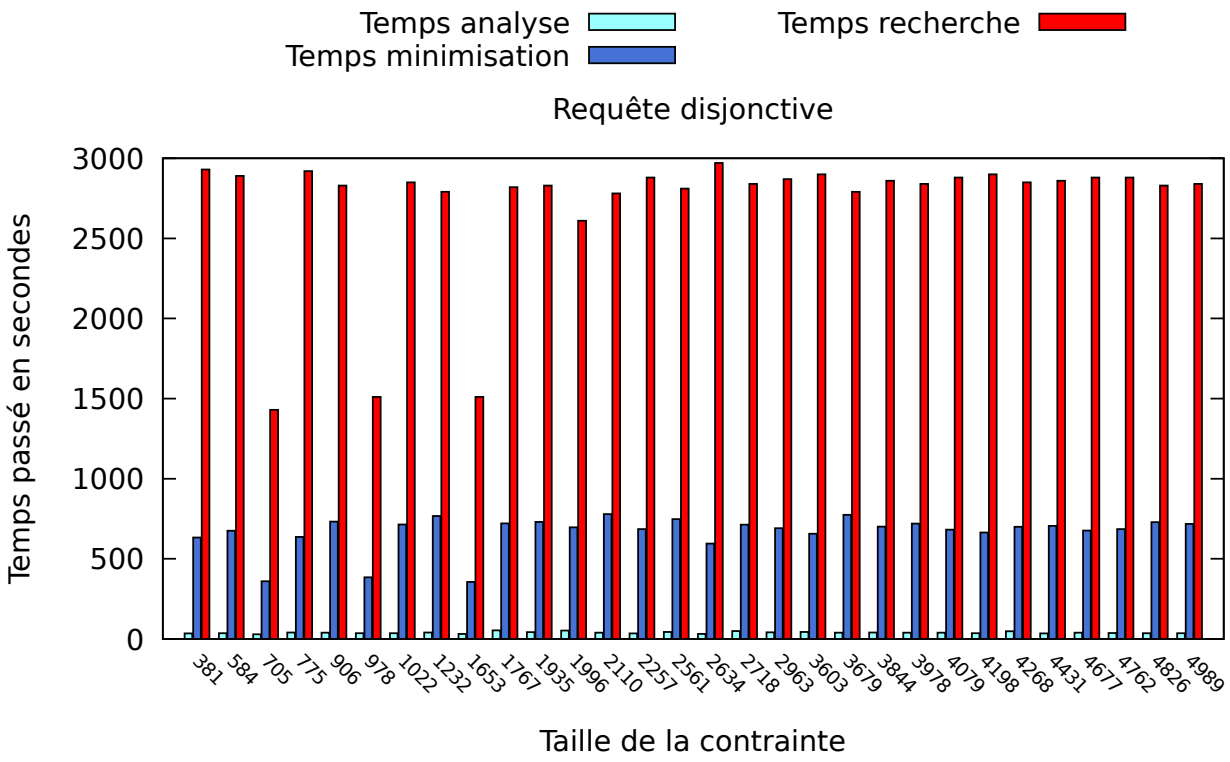


FIGURE 2.8: Temps en seconds pour les requêtes disjonctives

Les temps passés dans les différentes parties sont relativement constants pour les différents groupes de requêtes. En moyenne on passe 39 secondes dans l'analyse des solutions, un peu moins de 10 minutes dans la minimisation et légèrement plus de 45 minutes dans la recherche des solutions.

Regardons maintenant le pourcentage de solutions trouvées en une heure, ainsi que le nombre d'appels au SAT solveur dans ce même temps pour trouver un EFM, toujours sur les mêmes requêtes, dans les figures 2.9, 2.10 et 2.11. Ces graphes sont construits à partir des tableaux de l'annexe B.

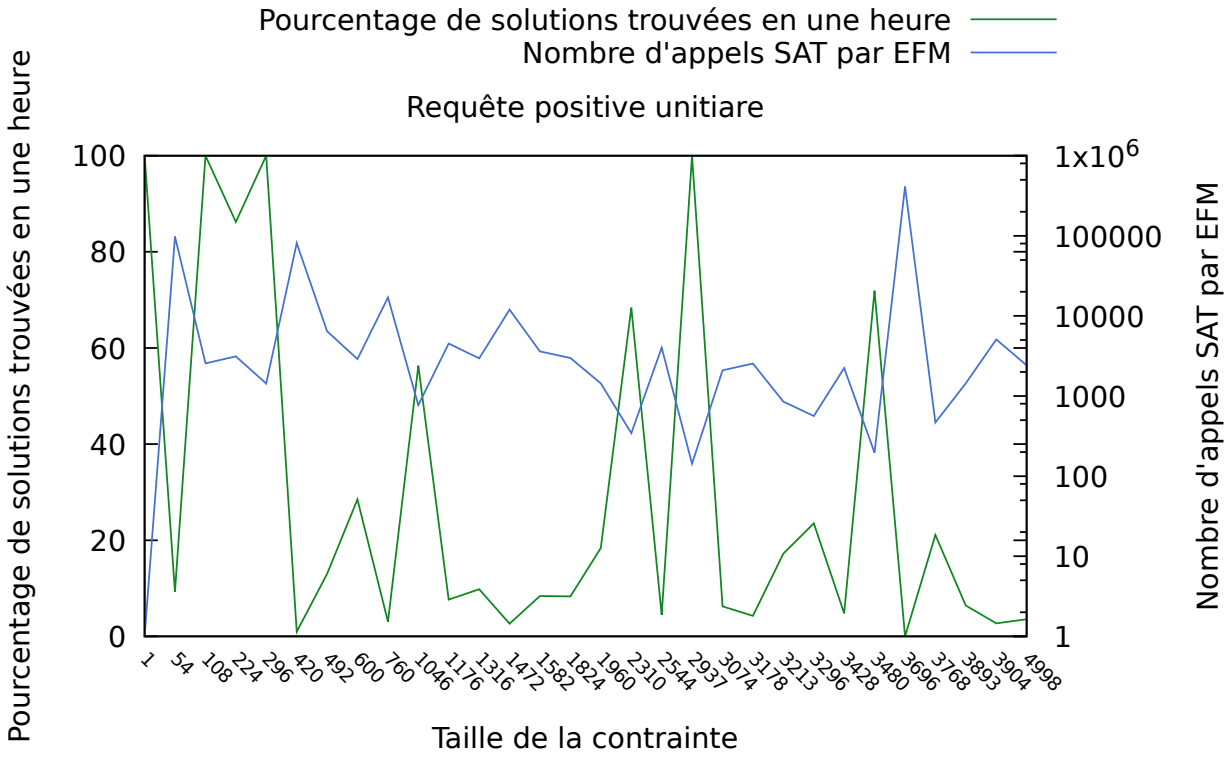


FIGURE 2.9: Pourcentage de solutions trouvées et nombre d'appels SAT par EFM pour les requêtes positives unitaires

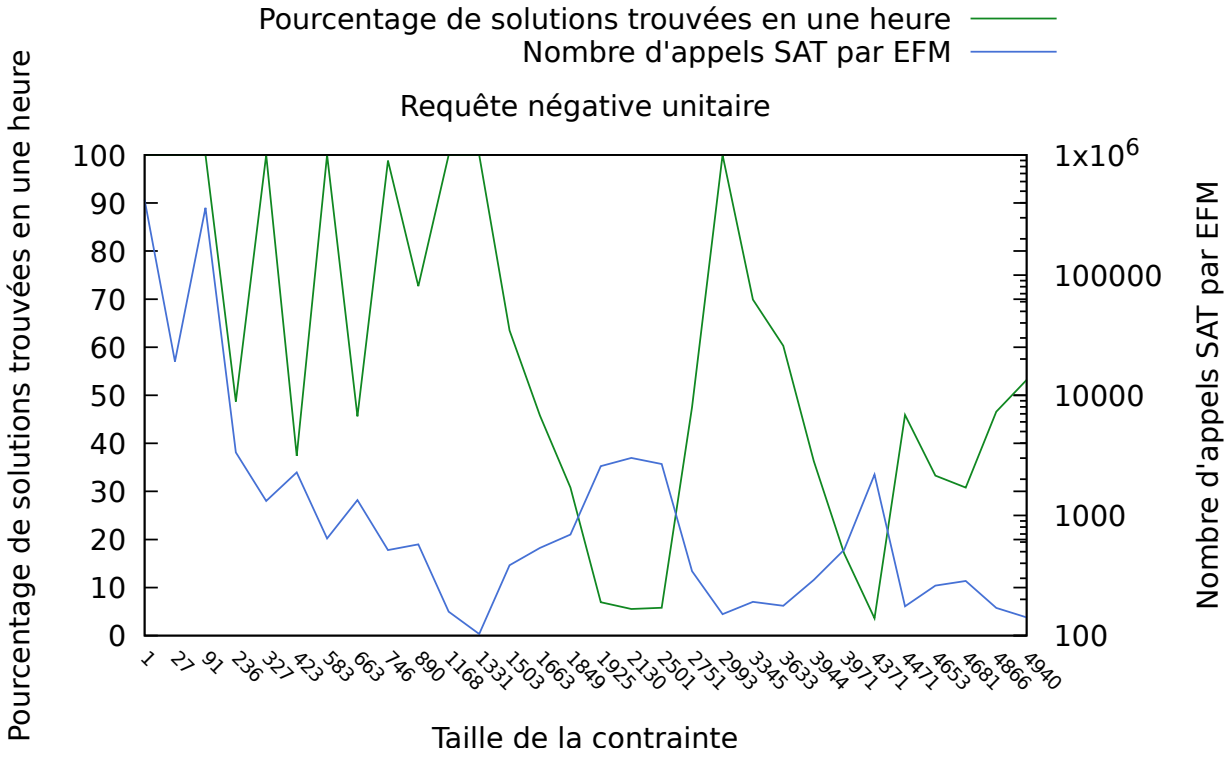


FIGURE 2.10: Pourcentage de solutions trouvées et nombre d'appels SAT par EFM pour les requêtes négatives unitaires

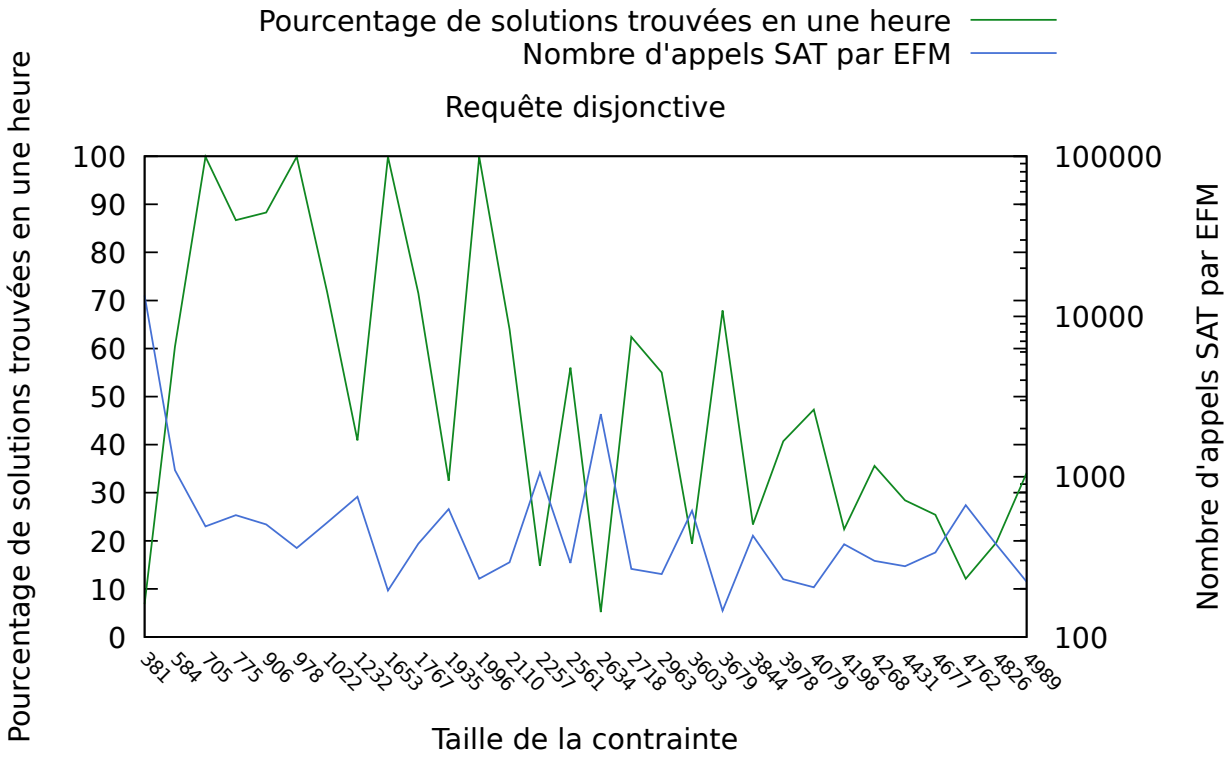


FIGURE 2.11: Pourcentage de solutions trouvées et nombre d'appels SAT par EFM pour les requêtes disjonctives

Dans la grande majorité des cas (87%) nous ne finissons pas le calcul avec une heure de timeout. Les résultats peuvent énormément varier d'un test à l'autre, cela est dû aux problèmes que rencontre le SAT solveur pour compléter une voie « incomplète » comme nous l'avons vu dans l'analyse de l'algorithme 11. Ces problèmes induisent un grand nombre d'appels au SAT solveur : on calcule en moyenne 8580 solutions côté SAT pour obtenir un EFM valide. Il s'agit donc d'un point crucial d'amélioration.

Cependant cette approche n'est pas complètement mauvaise : elle nous permet, puisque l'on calcule les solutions par taille croissante, de prouver que l'on a calculé tous les EFMs inférieurs à une taille. La taille pour laquelle on a trouvé tous les EFMs est reportée dans les figures 2.12, 2.13 et 2.14, on indique également le pourcentage de solution que cela représente.

On constate que l'on trouve pratiquement tous le temps tous les EFMs de taille 15. Un des usages de notre outil peut donc être de trouver la taille minimale min des EFMs puisqu'il devra, dans tous les cas démontrer que les tailles de 2 à $min - 1$ n'admettent pas de solution.

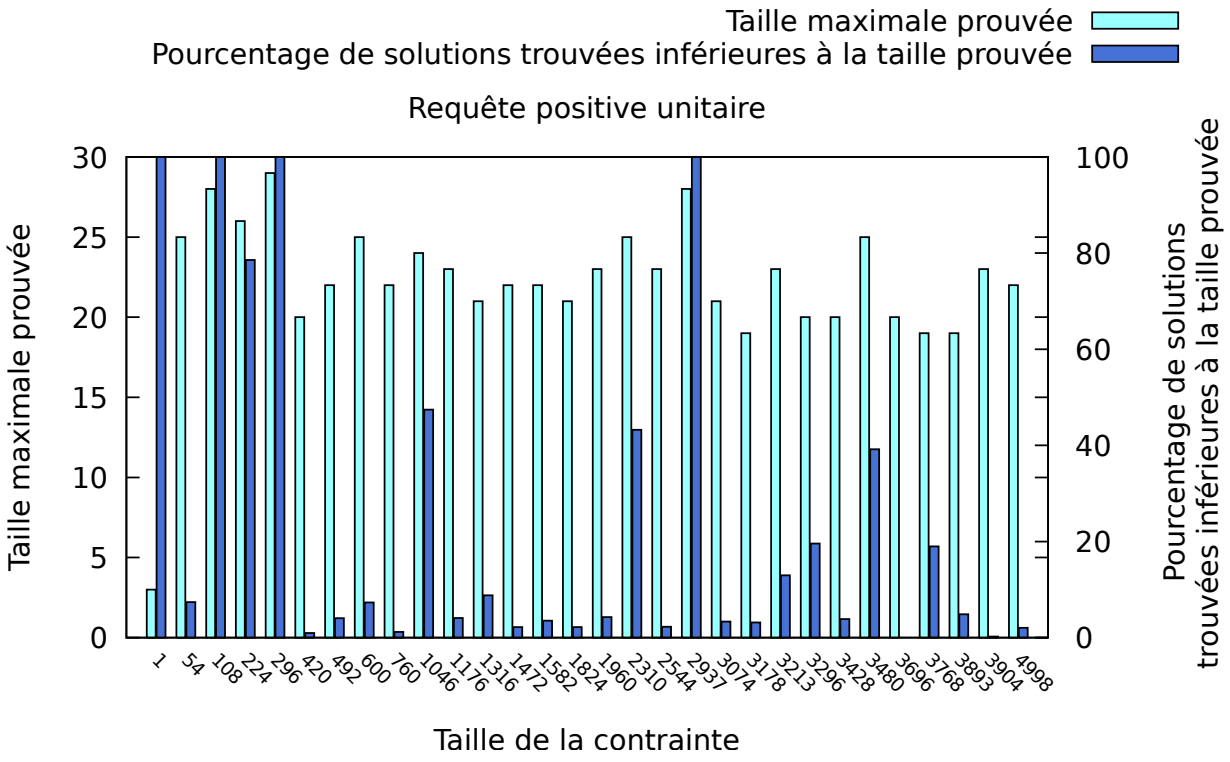


FIGURE 2.12: Taille maximale prouvée pour les requêtes positives unitaires

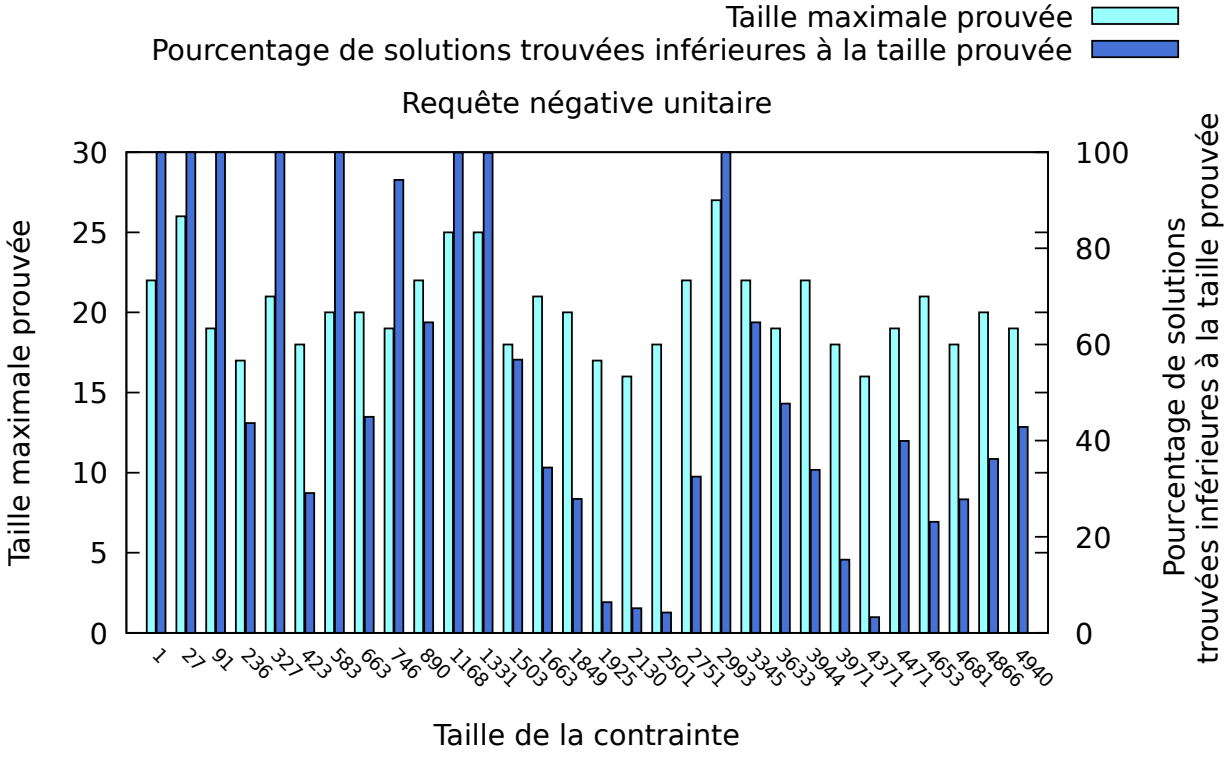


FIGURE 2.13: Taille maximale prouvée pour les requêtes négatives unitaires

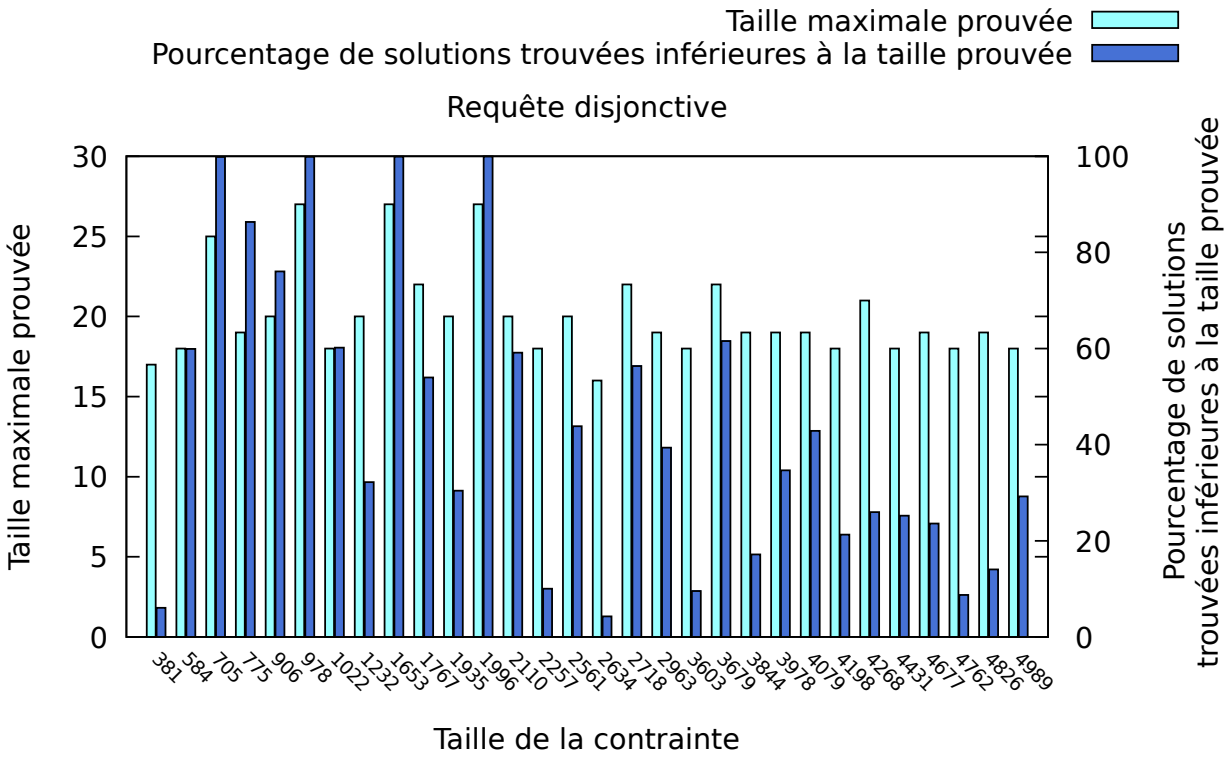


FIGURE 2.14: Taille maximale prouvée pour les requêtes disjonctives

Nous allons maintenant voir comment résoudre les problématiques de stœchiométrie de façon plus efficace en remplaçant l'utilisation d'un SAT solveur par un SMT.

Chapitre 3

Approche SMT

L'utilisation d'un SMT nous permet de prendre en compte naïvement les problématiques liées à la prise en compte de la stœchiométrie. En effet, utiliser un SMT ayant une théorie « Linear Real Arithmetic » (LRA) nous offre la possibilité d'exprimer des contraintes sur des valeurs rationnelles, nous permettant ainsi de calculer directement des EFMs (ou ensemble d'EFMs) et donc de ne plus avoir une architecture en deux parties, l'une trouvant des « candidats EFMs », la partie SAT, et l'autre validant la stœchiométrie de ces candidats, la partie matricielle, comme nous l'avons vu dans la section 2.2.

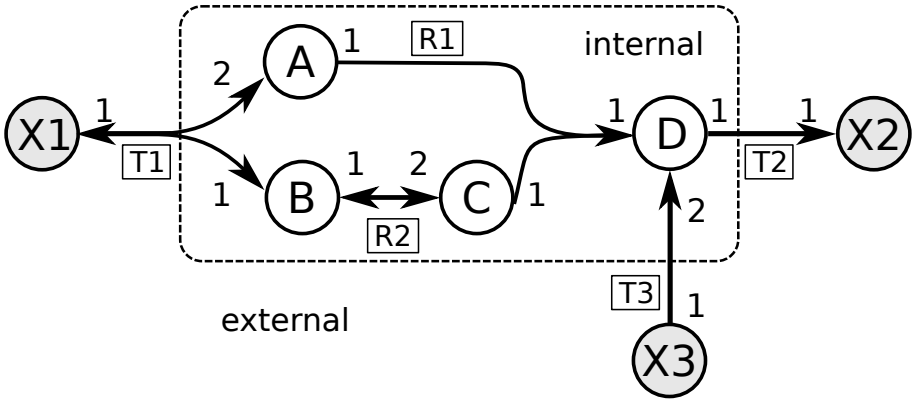
Nous allons dans un premier temps présenter comment modéliser un réseau métabolique dans une théorie LRA puis comment la minimisation est simplifiée par l'utilisation du SMT et enfin nous comparerons les performances de notre approche pour le calcul des EFMs respectant des contraintes booléennes avec RegEfmTool qui est l'état de l'art pour cette problématique.

3.1 Modélisation avec la stœchiométrie

3.1.1 Modélisation du réseau métabolique

La modélisation avec un SMT comme la modélisation avec un SAT solveur cherche à représenter un réseau métabolique, on retrouve dans la modélisation SMT les mêmes étapes qu'en 2.1.1.

Nous rappelons la figure 1.2.



Rappel de la figure 1.2

1. Puisque les réactions réversibles sont divisées en deux réactions irréversibles, toutes les réactions doivent avoir un coefficient positif :

$$\forall r \in R, r \geq 0 \tag{3.1}$$

Ce qui, une fois appliquée à notre exemple, donne :

$(T1 \geq 0)$ $(T2 \geq 0)$ $(R1 \geq 0)$ $(R2_{rev} \geq 0)$

$(T1_{rev} \geq 0)$ $(T3 \geq 0)$ $(R2 \geq 0)$

2. De même un métabolite interne ne peut pas être négatif :

$$\forall m \in M, m \geq 0 \tag{3.2}$$

Ce qui, une fois appliquée à notre exemple, donne :

$$(A \geq 0) \qquad (B \geq 0) \qquad (C \geq 0) \qquad (D \geq 0)$$

3. Un EFM doit avoir au moins une réaction active ; on force donc la somme des réactions à être strictement positive :

$$\sum_{r \in R} r > 0 \tag{3.3}$$

Ce qui, une fois appliquée à notre exemple, donne :

$$T1 + T1_{rev} + T2 + T3 + R1 + R2 + R2_{rev} > 0$$

4. Une réaction réversible ne peut pas se produire en même temps que son inverse :

$$\forall r \in rev(R), (r > 0) \Rightarrow (rev(r) = 0) \tag{3.4}$$

Ce qui, une fois appliquée à notre exemple, donne :

$$(R1 > 0) \Rightarrow (R1_{rev} = 0) \qquad (R1_{rev} > 0) \Rightarrow (R1 = 0)$$

$$(R3 > 0) \Rightarrow (R3_{rev} = 0) \qquad (R3_{rev} > 0) \Rightarrow (R3 = 0)$$

5. La stœchiométrie d'un métabolite interne utilisé dans un EFM doit être égale à la somme pondérée des enzymes le produisant. De plus puisque le système doit être à l'état stationnaire, la production doit être égale à la consommation du métabolite. Ce qui nous donne la règle suivante :

$$\begin{aligned} \forall\ m \in M, (m = & \sum_{r \in consomme(m,R)} stoechiométrie(m,r) \times r) \\ & \wedge (m = \sum_{r \in produit(m,R)} stoechiométrie(m,r) \times r) \end{aligned} \tag{3.5}$$

où $stoechiométrie(m, r)$ est le facteur de stœchiométrie du métabolite m dans la réaction r.

Ce qui, une fois appliquée à notre exemple, donne :

$$(A = 2 * T1) \wedge (A = R1 + 2 * T1_{rev})$$

$$(B = T1 + R2_{rev}) \wedge (B = R2 + T1_{rev})$$

$$(C = 2 * R2) \wedge (C = R1 + 2 * R2_{rev})$$

$$(D = 2 * T3 + R1) \wedge (D = T2)$$

Remarque : Puisque la stœchiométrie ainsi que la valeur des variables encodant les réactions sont positives, cette règle implique que si un métabolite n'est pas actif alors toutes les réactions qui le produisent et le consomment sont inactives.

L'ensemble de ces contraintes permet de modéliser un réseau métabolique et nous assurer que la solution retournée par un SMT avec ces contraintes sera un EFM, ou une combinaison d'EFMs.

Propriété 3.1 *Toute affectation ν satisfaisant les équations 3.1 à 3.5 modélise une voie à l'état stationnaire respectant la stœchiométrie.*

***Preuve* : 4** *Les équations 3.1 à 3.5 ayant la même structure que les équations 2.4 à 2.9, la preuve 1 reste applicable pour le fait que toute affectation ν satisfaisant les équations 3.1 à 3.5 modélise une voie.*

Il faut cependant montrer que cette voie est bien à l'état stationnaire. L'état stationnaire se définit par le fait que la voie ne fait pas varier la concentration des métabolites internes, cette propriété est assurée par l'équation 3.5 garantissant que consommation et la production d'un métabolite interne sont égales, donc que sa concentration est stable.
Toute affectation ν satisfaisant les équations 3.1 à 3.5 modélise donc une voie à l'état stationnaire.

Propriété 3.2 *L'encodage d'une voie à l'état stationnaire respectant la stœchiométrie dans un réseau métabolique respecte les équations 3.1 à 3.5.*

Puisque les équations 3.1 à 3.5 servent à définir une voie respectant la stœchiométrie, trivialement une voie est valide vis-à-vis de ces équations.

Nous allons maintenant présenter les contraintes que l'on peut ajouter à notre modélisation afin de pouvoir interroger le réseau.

3.1.2 Modélisation des contraintes

Le calcul des EFMs avec un SMT se fait en trois phases : l'encodage des contraintes, la recherche d'une solution et la minimisation. Les deux dernières phases vont se répéter jusqu'à l'énumération de tous les EFMs satisfaisant les contraintes. Les contraintes peuvent soit être ajoutées dans la première phase, donc avant les premiers résultats (on parlera alors de contraintes statiques), soit être ajoutées lors de la recherche des EFMs, cela permet de prendre en compte les EFMs trouvés pour la recherche des solutions suivantes. On parlera dans ce dernier cas de contraintes dynamiques.

Les contraintes statiques

L'utilisation d'un SMT nous permet d'exprimer simplement des contraintes. Bien entendu, les contraintes déjà facilement exprimables avec un SAT solveur restent simples à exprimer avec un SMT :

— Pour forcer une réaction ou un métabolite ($elem$) à être présent dans la solution nous ajoutons la contrainte $(elem > 0)$;

— Pour forcer une réaction ou un métabolite ($elem$) à être absent de la solution nous ajoutons la contrainte $(elem = 0)$;

— Toute formule en logique propositionnelle peut être exprimée. Cependant pour faciliter le traitement notre outil demande à ce que la formule soit exprimée en CNF. Bien que dans le pire des cas transformer une formule en logique propositionnelle en CNF soit exponentiellement dur, les requêtes étant courtes et souvent exprimées par des humains, la transformation reste calculable avec une approche naïve et est laissée aux soins de l'utilisateur.

L'expressivité du SMT nous permet d'exprimer uniquement sous forme de contraintes les requêtes sur la taille et nous n'avons plus besoin d'ajouter une sur-couche logicielle pour ce cas là. Afin d'imposer une contrainte sur la taille il nous faut un moyen de compter le nombre de

réactions présentes. Cela se fait assez simplement grâce à l'utilisation de l'opérateur « ITE » (« If then else »), qui est proposé par notre solveur LRA.

Opérateur « ITE » : Soit c une contrainte exprimable en LRA telle que $c = ITE(condition, a, b)$ où *condition* est un booléen ou une formule retournant un booléen, et a et b deux contraintes exprimables en LRA, alors :

$$c = \begin{cases} a \text{ si } condition = \top \\ b \text{ si } condition = \bot \end{cases}$$

Nous pouvons donc obtenir le nombre de réactions actives grâce à : $\sum_{r \in R} ITE((r > 0), 1, 0)$. Nous pouvons utiliser cette valeur dans une contrainte pour assurer qu'elle soit, par exemple, inférieure à une borne. De même, il est possible de s'intéresser uniquement à un sous-ensemble de réactions, ce qui permet d'exprimer des requêtes telles que « X enzymes dans un ensemble doivent être au moins/au plus activés simultanément ». Bien sûr la même approche est également applicable aux métabolites.

L'ensemble de ces requêtes s'applique directement sur la solution produite par le SMT ; la minimisation devra en tenir compte afin que les solutions finales soient valides vis-à-vis des contraintes.

Les contraintes dynamiques

Lors d'une recherche d'EFM sans contraintes dynamiques explicites nous ajoutons la contrainte suivante pour chaque EFM que nous trouvons : $\bigvee_{r \in efm}(r = 0)$ ce qui nous permet de nous assurer que les futures solutions ne contiendront pas un EFM déjà trouvé. De plus, cela assure également que la recherche se terminera lorsque tous les EFMs auront été trouvés.

Bien que les exemples suivants ne soient pas implémentés dans notre outil (que nous avons baptisé *MCFTool* en référence à sa capacité à calculer des $MCFM$s, notion que nous aborderons dans le chapitre 4) ils restent de bons exemples de ce que les contraintes dynamiques permettent de modéliser. Nous pouvons ainsi :

— Calculer des chemins indépendants. Il s'agit de trouver des chemins n'ayant pas d'enzyme en commun. Pour cela on ajoute simplement la clause suivante : $\bigwedge_{r \in efm}(r = 0)$. Il est important de noter que la contrainte assure uniquement que les EFMs trouvés ne partagent pas d'enzymes. De plus, il n'existe pas une solution unique à ce problème, et l'algorithme permettra de trouver une seule de ces solutions. Par exemple le réseau de la figure 3.1 contient trois EFMs : (1){ R1, R3}, (2){R2, R4} et (3){R2, R5, R3}. Si le

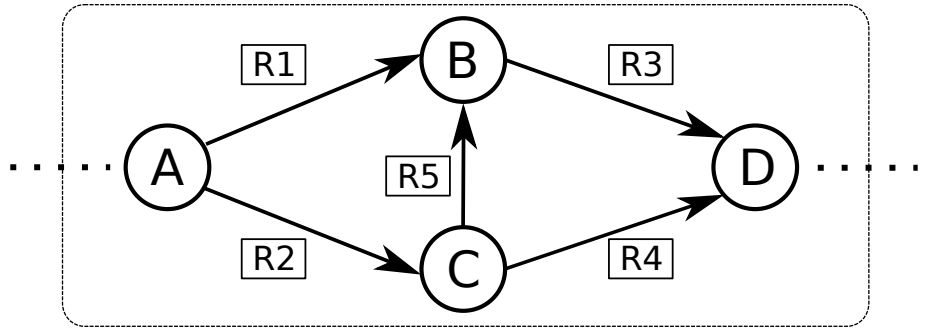


FIGURE 3.1: Réseau d'exemple pour le calcul de chemins indépendants

solveur trouve comme premier EFM la solution 1 ou 2 alors l'ensemble d'EFMs retourné sera {1, 2}, dans le cas contraire l'ensemble ne contiendra que le troisième EFM.

— Maximiser le rapport de production entre deux métabolites. Trouver l'EFM tel que le rapport entre deux métabolites A et B soit maximal revient à ajouter la contrainte suivante lorsqu'on trouve un EFM : $(A/B) > valeur$ où $valeur$ est la valeur du ratio A/B dans l'EFM que l'on vient de trouver. Le dernier EFM trouvé sera l'EFM maximisant ce ratio.

— Vérifier la robustesse du réseau. Il s'agit de s'assurer que tous les EFMs trouvés ont au moins X réactions de différence. Pour cela nous allons ajouter pour chaque EFM que nous trouvons la contrainte suivante : $(\sum_{r \in efm} ITE(r = 0, 1, 0)) \geq X$; ainsi tous les nouveaux EFMs devront avoir au moins X réactions inactives parmi les réactions de chaque EFM déjà trouvé. De même que dans le calcul de chemins indépendants, l'algorithme retournera un ensemble de solutions respectant les contraintes, d'autres pouvant exister.

Les contraintes que nous avons présentées nous permettent d'interroger un réseau métabolique afin de calculer uniquement des voies respectant une requête. Cependant si nous ne minimisons

pas les résultats fournis par le SMT la solution ne sera pas, dans le cas général, un EFM.

3.2 Minimisation

De même que l'utilisation du SMT a simplifié l'encodage des requêtes, elle va permettre plusieurs optimisations de la minimisation.

3.2.1 Minimisation naïve

Une première approche utilise la même structure d'algorithme que l'algorithme 7 (sous-section 2.1.4) et va faire appel au SMT afin de trouver une solution, respectant les contraintes mais contenant une réaction de moins.

Algorithm 13 MinimisationNaiveSMT(solution)
Input : *solution* une affectation satisfaisant $formule$ (la formule codant les voies du réseau métabolique).
Ouput : un ensemble de variables modélisant une voie incluse dans *solution*, minimale en terme d'inclusion de voie respectant les requêtes.

1: $SMT.checkPoint()$
2: $SMT.addConstraint(\bigvee_{r \in Supp(solution)} r = 0)$
3: $SMT.addConstraint(\bigwedge_{r \notin Supp(solution)} r = 0)$
4: **while** $SMT.isSAT()$ **do**
5: $\quad solution = SMT.getSolution()$
6: $\quad SMT.addConstraint(\bigvee_{r \in Supp(solution)} r = 0)$
7: $\quad SMT.addConstraint(\bigwedge_{r \notin Supp(solution)} r = 0)$
8: **end while**
9: $SMT.RestorecheckPoint()$
10: **return** *solution*

L'algorithme en entier donne donc :

Remarque : L'encodage nous garantit qu'une solution est bien une voie (propriété 3.1). De plus le mécanisme qui nous sert à bloquer une solution est le même que dans l'approche SAT et la propriété 2.3 nous assure que lorsque nous bloquons une voie que nous avons trouvée, nous ne perdons pas d'autres EFMs. Enfin puisque le nombre d'EFMs est fini et que tout EFM

Algorithm 14 CalculeEFMs
Ouput : Ensemble des EFMs respectant les contraintes.

```
1: formule = initialise()
2: sol = {}
3: while SMT.check(formule) == true do
4:    solMin = MinimisationNaiveSMT(SMT.getSolution())
5:    if dim(kernel(solMin)) == 1 then
6:       sol = sol ∪ {solMin}
7:    end if
8:    SMT.addConstraint(⋁_{r∈solMin} ¬r)
9: end while
10: return sol
```

est une solution (propriété 3.2) alors l'algorithme terminera lorsqu'il aura listé l'ensemble des EFMs satisfaisant les contraintes.

L'utilisation du SMT nous a permis de régler les problèmes dus à la stœchiométrie, mais lorsque l'on utilise une requête nous avons toujours le problème de minimalité que nous avons vu avec l'algorithme 9 et nous avons toujours besoin de la ligne 5 de l'algorithme 14 afin de nous assurer que la solution est bien un EFM (comme en section 2.2 nous utilisons les résultats de (Klamt et al., 2005)).

Nous n'avons donc plus les problématiques de « EFMs à compléter » qui apparaissaient, dans la version utilisant un SAT solveur, lorsque le noyau d'une voie trouvée par la partie SAT était réduit à $\{0\}$. Le fait que le SMT prenne directement en compte les contraintes de stœchiométrie va donc drastiquement améliorer les performances de l'algorithme. Le schéma 3.2 compare le nombre d'appels à SAT ou à SMT, pour la recherche de solution initiale (les appels dus à la minimisation ne sont donc pas représentés) en fonction du nombre de solutions trouvées, les tests ont été faits sur le réseau métabolique « central carbon metabolism » (Peres et al., 2011). Il s'agit d'un petit réseau comportant 61 EFMs .

Dans le cas de l'approche SMT le nombre d'appels au SMT est égal au nombre de solutions plus un. Le dernier appel au SMT démontrant qu'il n'existe plus de solution. Dans le cas de l'approche SAT il peut y avoir plus de 100 fois plus d'appels au SAT solveur dans les petits exemples. Sur des réseaux avec un plus grand nombre de réactions la différence peut être encore plus marquante. Nous avons vu dans la section 2.4 un cas où il fallait 9.8×10^4 appels au SAT

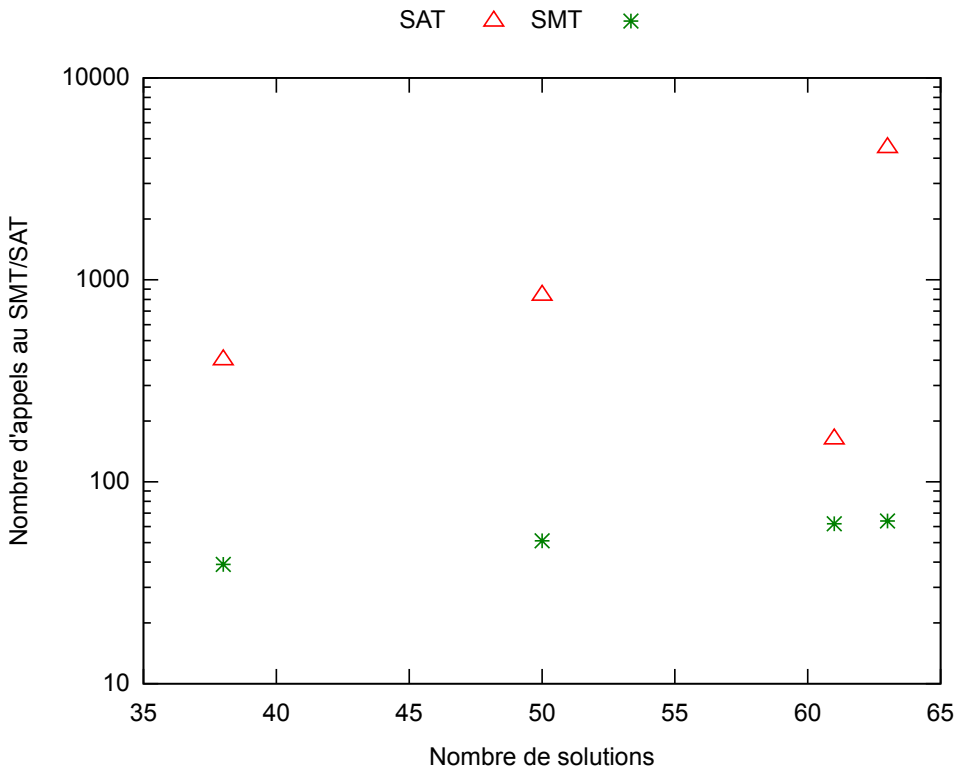


FIGURE 3.2: Comparaison du nombre d'appels, hors minimisation, au solveur entre les versions SAT et SMT sur des réseaux métaboliques issus du métabolisme de la levure (Peres et al., 2014)

solveur par solution.

Nous avons montré les avantages de l'utilisation de l'approche SMT mais celle-ci reste relativement naïve et chaque solution initiale trouvée par le SMT devra faire l'objet d'une minimisation ne produisant qu'un seul EFM. Pour pallier ce problème, nous allons présenter la « Minimisation multiple ».

3.2.2 Minimisation multiple

L'un des principaux défauts de la minimisation naïve est que pour une solution initiale du SMT on ne trouve qu'une seule solution minimale. Si une solution initiale est composée de plusieurs EFMs nous trouverons un premier EFM puis nous serons obligé de rechercher les autres EFMs sans tirer profit du fait que nous les avions déjà, en partie, trouvés auparavant.

Dans cette sous-partie nous allons présenter une méthode de minimisation multiple utilisant du calcul matriciel. Nous en verrons une autre utilisant le SMT en section 4.2.

Minimisation multiple via calcul matriciel

Lorsque que le SMT trouve une solution, la solution contient les réactions utilisées ainsi que la stœchiométrie pour chaque réaction. Or, les règles imposées au SMT en 3.1.1 exigent que la consommation et la production de métabolites internes s'annulent. De plus, les EFMs étant minimaux en terme d'inclusion il en découle que, d'une part, chaque réaction présente dans la solution initiale appartient à, au moins, un EFM et, d'autre part, il existe au moins une réaction qui n'appartient pas à tous les EFMs, sauf si bien sûr la solution initiale ne contient qu'un EFM. Nous proposons une solution utilisant ces propriétés pour décomposer une solution initiale en l'une de ses décompositions. Dans un premier temps nous présenterons le principe de base sans requête puis nous verrons comment intégrer les requêtes dans cette minimisation (auquel cas nous n'aurons pas, en général, une décomposition).

Cas sans requête : L'idée est de diviser récursivement la solution initiale S_0 en deux nouvelles solutions non nulles jusqu'à obtenir des EFMs. La première étape consiste à trouver une réaction qui ne soit pas utilisée par tous les EFMs de la solution. Pour cela nous utilisons le fait que toutes les solutions sont des EFMs ou une combinaison d'EFMs et que nous sommes à l'état stationnaire, avec uniquement des réactions irréversibles, ce qui implique que toutes les solutions définissent un cône convexe polyédral pointé C. Notre solution, en tant que combinaison d'EFMs, est une combinaison d'arêtes que nous ne connaissons pas. La première étape de notre minimisation consiste à trouver une solution qui utilise au moins un EFM de moins, donc une arête du cône C en moins. Pour cela nous allons chercher à annuler une composante du vecteur solution initial tout en restant dans le cône C. Pour ce faire, nous utiliserons un vecteur de base du noyau de la matrice de stœchiométrie qui n'est pas colinéaire avec la solution.

Algorithm 15 subtract(sol, S_1)
Input : $sol \in C$ et S_1 un vecteur du noyau N réduit aux réactions de $Supp(\{\text{sol}\})$
Output : $S_2 \in C$ tel que sol soit une combinaison linéaire positive de S_1 et S_2 et $supp(S_2) \subset supp(sol)$

1: $max = max\{i | R_i \in Supp(sol)\}(S_{1i}/sol_i)$
2: **return** $sol - (1/max) \times S_1$

Discussion sur l'algorithme 15 . Pouvons nous affirmer que le retour est bien une voie ? Nous savons que tout vecteur dans le cône est une voie, et que tout vecteur du noyau ayant uniquement des composantes positives est dans le cône. De plus si deux vecteurs v_1 et v_2 sont dans le noyau alors tout vecteur v_3 tel que $v_3 = v_1 - (1/\alpha) \times v_2$ est dans le noyau.
De plus, si v_1 est strictement positif (dans le cône), et v_2 n'est pas négatif (si celui ci est négatif on peut le remplacer par $-v_2$) on peut trouver un $\alpha > 0$ tel que pour tout $i, v_{3i} \geq 0$ et qu'il existe un $j, v_{3j} = 0$ ce qui peut également s'exprimer :

$$\begin{cases} \forall i, \alpha \geq v_{2i}/v_{1i} \\ \exists j, \alpha = v_{2j}/v_{1j} \end{cases}$$

Une solution à cette équation est de prendre $\alpha = max(v_{2i}/v_{1i})$.

Ainsi, dans l'algorithme 15, puisque le vecteur *sol* est dans le cône, donc dans le noyau, et que le vecteur S_1 est également dans le noyau, le résultat sera un vecteur du noyau dont toutes les

valeurs seront positives ou nulles avec au moins une nulle. Ce vecteur sera donc un vecteur de C représentant une voie incluse dans la voie initiale ou un vecteur nul.

Remarque : L'algorithme 15 ne retourne un vecteur nul que si *sol* et S_1 sont colinéaires.

Algorithm 16 extraire_sous_solution(sol)
Input : Un vecteur solution $sol \in C$.
Output : une sous-solution de sol $\in C$ ou {} si *sol* est indivisible.

1: **if** $dim(kernel(N^{Supp(sol)})) == 1$ **then**
2: **return** {}
3: **else**
4: $ker = getKernel(N, sol)$ % Retourne un vecteur de base du noyau non colinéaire à *sol*
5: **return** $subtract(sol, ker)$
6: **end if**

Discussion sur l'algorithme 16 Le cas où la dimension du noyau est de un nous indique que la solution est un EFM et que nous ne pouvons donc pas trouver de sous ensemble strict valide.

Dans le cas contraire, puisque *ker* et *sol* sont non-colinéaires, la fonction « subtract » retournera une voie valide étant une sous-solution de *sol*.

L'algorithme 16 permet de trouver le premier sous ensemble S_1. Nous voulons maintenant construire une solution S_2 telle que :

$$S_1 \subset S_0,\ S_2 \subset S_0 \tag{3.6}$$

$$S_1 \backslash S_2 \neq \emptyset \text{ et } S_2 \backslash S_1 \neq \emptyset \tag{3.7}$$

Pour cela nous utilisons encore l'algorithme 15 afin d'obtenir S_2 en annulant une ou plusieurs composantes de S_0. L'équation 3.6 est donc vérifiée.
De plus S_1 étant un sous ensemble de S_0 et S_0 étant une combinaison d'EFMs dans le cône, il est strictement positif. Puisque S_2 est le retour de l'algorithme 15, cela nous assure que :

— Les dimensions égales à zéro dans S_1 seront non nulles dans S_2 ;

— Au moins une des dimensions non nulles dans S_1 sera nulle dans S_2.

L'équation 3.7 est donc vérifiée.

Grâce aux algorithmes 16 et 15 nous pouvons diviser une voie non minimale S_0 en deux voies distinctes S_1 et S_2 strictement incluses dans S_0. L'algorithme 17 permet de trouver un ensemble de voies minimales incluses dans une voie S_0 en la divisant récursivement.

Algorithm 17 Minimise(sol)
Input : sol $\in C$.
Output : une décomposition de sol en $EFMs$

```
1: S1 = extraire_sous_solution(sol)
2: if S1 == {} then
3:     SMT.addConstraint(⋁_{R∈Supp(sol)} R = 0)
4:     return {sol}
5: end if
6: S2 = subtract(sol, S1)
7: return Minimise(S1) ∪ Minimise(S2)
```

Maintenant que nous avons une minimisation plus efficace nous pouvons simplement mettre à jour notre algorithme. Il est présenté dans l'algorithme 18. Le fait de ne plus trouver les solutions une à une ne change pas la validité de l'algorithme.

Algorithm 18 calculeEFM(réseau)
Input : Un réseau métabolique.
Output : La liste des EFMs contenus dans le réseau

```
1: encodeReseau(SMT, reseau)
2: listEFM = {}
3: while SMT.isSat() do
4:     sol = SMT.getSolution()
5:     listEFM = listEFM∪{minimise(sol)}   % La fonction minimise se charge de bloquer les EFMs
6: end while
7: return listEFM
```

L'intégration des requêtes se base sur le même principe, il faudra cependant modifier légèrement l'algorithme 17 pour en tenir compte.

Cas avec requête : La première question que l'on doit se poser lorsqu'on traite les requêtes est « A quel niveau doit-on prendre en compte la requête ? ». Nous savons que la solution

initiale S_0 respecte la requête, puisqu'elle a été trouvée par le SMT mais nous n'avons pas de garantie que tout sous-ensemble S_1 strictement inclus dans S_0 respecte la requête. De plus, dans le cas général, si S_1 ne respecte pas la requête, on ne peut pas déduire que S_2, tel que S_2 soit strictement inclus dans S_1, ne respecte également pas la contrainte. Nous ne pouvons donc, dans le cas général, prendre en compte la requête qu'une fois la minimisation terminée. Par exemple, si nous supposons la requête $T1 \vee \neg T3$ sur le réseau de la figure 3.3.

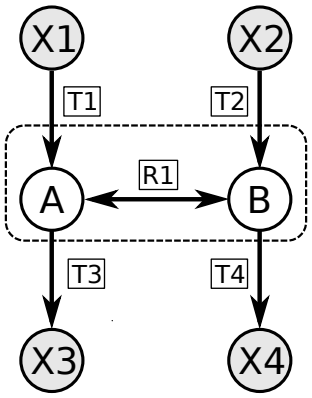


FIGURE 3.3: Réseau jouet pour les requêtes

Une solution initiale peut être : { T1, 2 T2, 2 T3, T4, $R1_{rev}$} qui peut être décomposée en (1) {T1, T3} et (2) {2 T2, T3 ,T4 ,$R1_{rev}$}. La solution (1) est minimale et respecte la requête, nous la gardons. La solution (2) ne respecte pas la requête mais nous ne pouvons pas nous arrêter à cette étape, en effet, elle peut être divisée en (3) {T2,$R1_{rev}$,T3} et (4){T2,T4} qui sont toutes deux minimales et où (3) ne respecte pas la requête contrairement à (4) qui la respecte.

Remarque : Pour certaines requêtes c, telles que trouver tous les EFMs ayant une taille supérieure à une valeur donnée ou les requêtes positives, une voie ne respectant pas c ne peut pas contenir d'EFMs la respectant. Il serait donc possible d'optimiser la minimisation pour ces cas là.

L'algorithme 19 présente une version de la minimisation qui prend en compte les requêtes.

Dans la partie suivante nous comparerons le temps de calcul des EFMs avec la minimisation naïve et multiple (table 3.1) et aussi avec le temps de calcul de RegEfmTool avec et sans requête.

Algorithm 19 MinimiseRequete(sol)
Input : sol $\in C$. Ne respecte pas forcement la requête.
Output : l'ensemble des EFMs respectant la requête inclus dans *sol*.

```
1: S1 = extraire_sous_solution(sol) % S1 ne respecte pas forcément la requête
2: if S1 == {} then
3:    SMT.addConstraint(⋁_{R∈Supp(sol)} R = 0) % Que la solution respecte ou non la requête,
                                               on ne veut plus la retrouver plus tard.
4:    if checkRequete(sol) then
5:       return {sol}
6:    else
7:       return {}
8:    end if
9: end if
10: S2 = subtract(sol, S1)
11: return MinimiseRequete(S1) ∪ MinimiseRequete(S2)
```

3.3 Résultats

Dans cette section nous allons comparer les performances des algorithmes présentés dans la section 3.2 avec l'état de l'art, RegEfmTool. Pour ce faire nous comparerons dans un premier temps les résultats lors de l'énumération brute des EFMs, puis lors de l'énumération d'EFMs respectant différentes requêtes.

3.3.1 Comparaison sans requête

Notre approche, bien qu'elle permette l'énumération de tous les EFMs, a été conçue pour une prise en compte efficace des requêtes. Nous souhaitons cependant connaître ses performances lorsqu'on n'utilise pas de requêtes. RegEfmTool est basé sur EfmTool, un outil utilisant la méthode de la double description qui est optimisée pour énumérer un grand nombre d'EFMs, de ce fait, celui-ci reste très efficace même lorsqu'on n'ajoute pas de contrainte au réseau.

Dans le tableau 3.1 nous comparons d'une part le nombre de solutions que notre approche a pu trouver dans le temps nécessaire à RegEfmTool pour calculer l'ensemble des EFMs pour quatre sous-réseaux issus de arabidopsis. (Jungreuthmayer et al., 2013a), et d'autre part le

résultat après 6h de calcul sur le réseau i AS253 (Smith and Robinson, 2011).

Problème	Temps	RegEfmTool	Approche naïve	approche matricielle
i AS253	Timeout (6h)	None	4 229	135 400
arabidopsis-1	1.30e+06	1 504 145	20 304	110 657
arabidopsis-2	1.29e+06	686 271	18 760	99 096
arabidopsis-3	1.36e+06	559 023	19 499	117 001
arabidopsis-4	144 199	277 262	2 803	16 597

TABLE 3.1: Comparaison de RegEfmTool et de $MCFTool$.

Notre approche n'est clairement pas compétitive avec RegEfmTool lorsqu'il s'agit de problèmes que celui-ci peut calculer, cependant, sur des problèmes qui ne sont pas accessibles pour RegEfmTool nous sommes en capacité de lister une fraction des EFMs. Nous pouvons également constater que l'approche de minimisation naïve est significativement moins efficace que la méthode de minimisation matricielle.

3.3.2 Comparaison avec requêtes

Les requêtes sont générées et exécutées dans les mêmes conditions que précédemment (présentées en section 2.4), nous reporterons la médiane de huit exécutions, en effet bien que l'algorithme soit déterministe l'utilisation de la médiane nous permet d'ignorer les cas de défaillance machine.

Sélections positives et négatives

Les requêtes de sélection positives et négatives unitaires ne profitent pas des optimisations principales de RegEfmTool. En effet la documentation de RegEfmTool nous indique « All rules that only require input reactions with the value 1 and an output reaction with the value 0 can be used during the iteration phase. » ce qui correspond à des requêtes qui sont des clauses négatives non unitaires. La sélection, positive ou négative, d'une réaction n'est pas modélisable directement dans RegEfmTool. Il faut écrire la requête de la façon suivante, où X est l'enzyme que l'on cherche à sélectionner et Y une réaction réversible quelconque (la logique serait la même avec une réaction irréversible mais la syntaxe serait légèrement différente) :

Sélection positive	Sélection négative
(1) $pX = (!0Y)$	(3) $pX = (0Y\|0Y)$
(2) $pX = (1bY\|1bY)$	(4) $pX = (!1bY)$

Ce qui se lit (nous précisons l'encodage en logique propositionnelle entre parenthèses) :

(1) Si Y n'est pas active alors X l'est. $(Y \vee X)$

(2) Si Y est active (quel que soit le sens de la réaction) alors X l'est. $((\neg Y \vee X) \wedge (\neg Y_{rev} \vee X))$

On a donc bien sélectionné X positivement.

(3) Si Y est inactive alors X l'est également. $(Y \vee \neg X\ )$

(4) Si Y est active (quel que soit le sens de la réaction) alors X est inactive. $(\ (\neg Y \vee \neg X) \wedge (\neg Y_{rev} \vee \neg X))$

Cela force donc X à être inactive. On remarquera que seule la contrainte (4) peut être prise en compte durant la phase d'itération de RegEfmTool, les autres seront vérifiées en post-traitement.

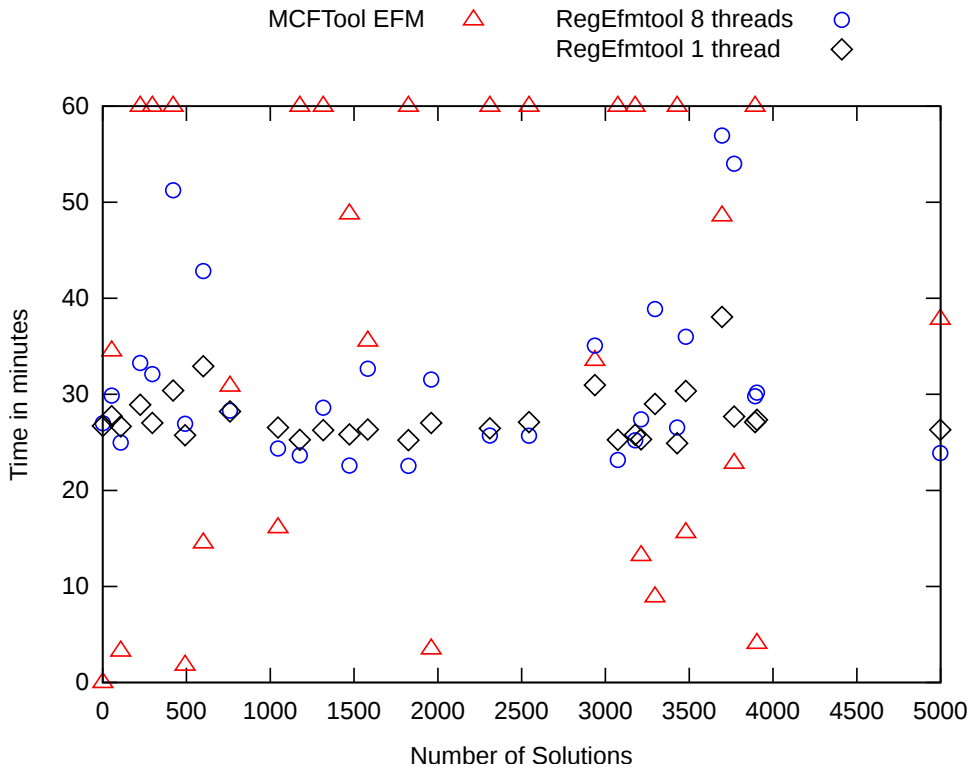


FIGURE 3.4: Temps d'exécutions des sélections positives unitaires

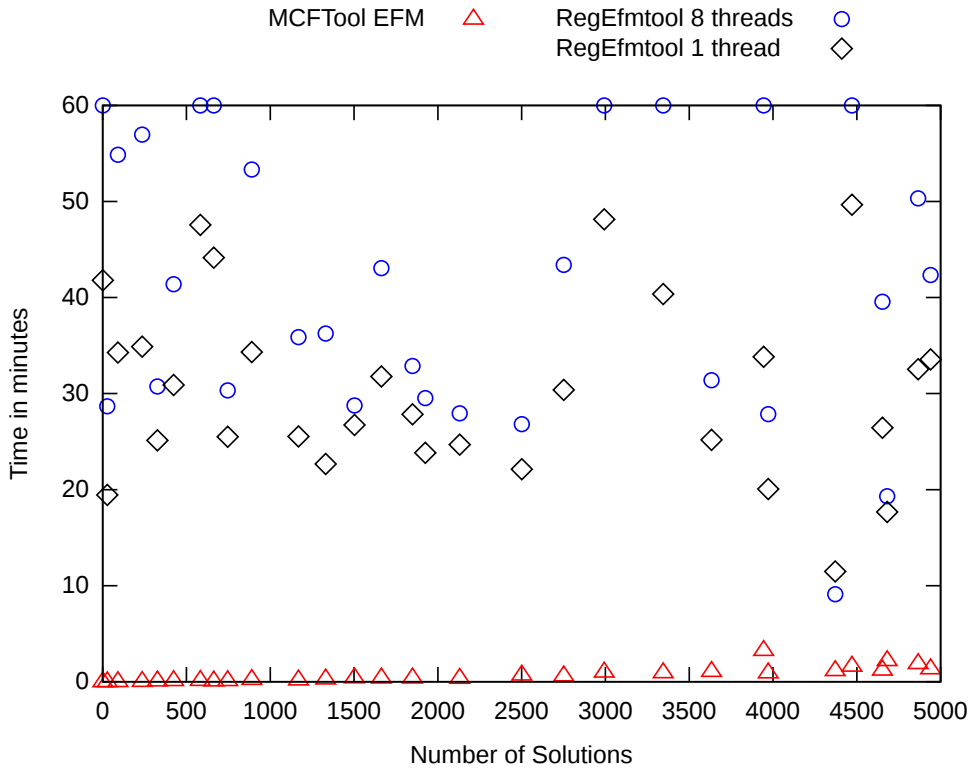


FIGURE 3.5: Temps d'exécution des sélections négatives unitaires

Les figures 3.4 et 3.5 mettent en évidence que :

— l'augmentation du nombre de threads pour RegEfmTool n'est pas systématiquement récompensé par une diminution du temps de calcul ;

— les performances de *MCFTool* sont particulièrement bonnes pour les sélections négatives alors qu'elle sont mitigées lors des sélections positives.

Cette différence de performance de *MCFTool* peut s'expliquer par le nombre moyen de contraintes dans ces deux cas de figures. En effet les requêtes positives ont une moyenne de 4.4 littéraux sélectionnés alors que les requêtes négatives en ont une moyenne de 13. Le réseau n'est pas assez contraint, les mécanismes d'apprentissages du SMT ne guident pas assez la recherche ainsi voir le problème comme un problème d'énumération comme le fait RegEfmTool est plus efficace.

Sélections de la forme $\neg x \vee \neg y$

Comme nous l'avons vu, les contraintes de cette forme seront prises en compte dans la phase d'itération de RegEfmTool et par conséquent les performances de celui-ci seront fortement affectées par l'ajout de ces contraintes comme on peut le voir dans la figure 3.6. Les requêtes ont une moyenne de 58.8 contraintes ce qui explique que *MCFTool* traite également correctement ces requêtes.
Le temps de calcul est sensiblement le même entre les deux approches mais les résultats de *MCFTool* se détériorent lorsque le nombre de solutions augmente.

Afin de pousser davantage les expérimentations sur ce type de requête, nous allons comparer les performances de RegEfmTool et de *MCFTool* sur des requêtes complètement aléatoires de ce type (il n'y a plus de sélection des requêtes afin que le nombre d'EFMs soit distribué aussi linéairement que possible entre 0 et 5 000).

Les graphes 3.7, 3.8 et 3.9 nous permettent de comparer les temps de calcul de RegEfmTool et *MCFTool* sur un ensemble de requêtes lorsqu'on augmente le nombre de réactions sélectionnées. Les points en dessous de la ligne « $x = y$ » sont les expérimentions où *MCFTool* a été plus rapide que RegEfmTool.

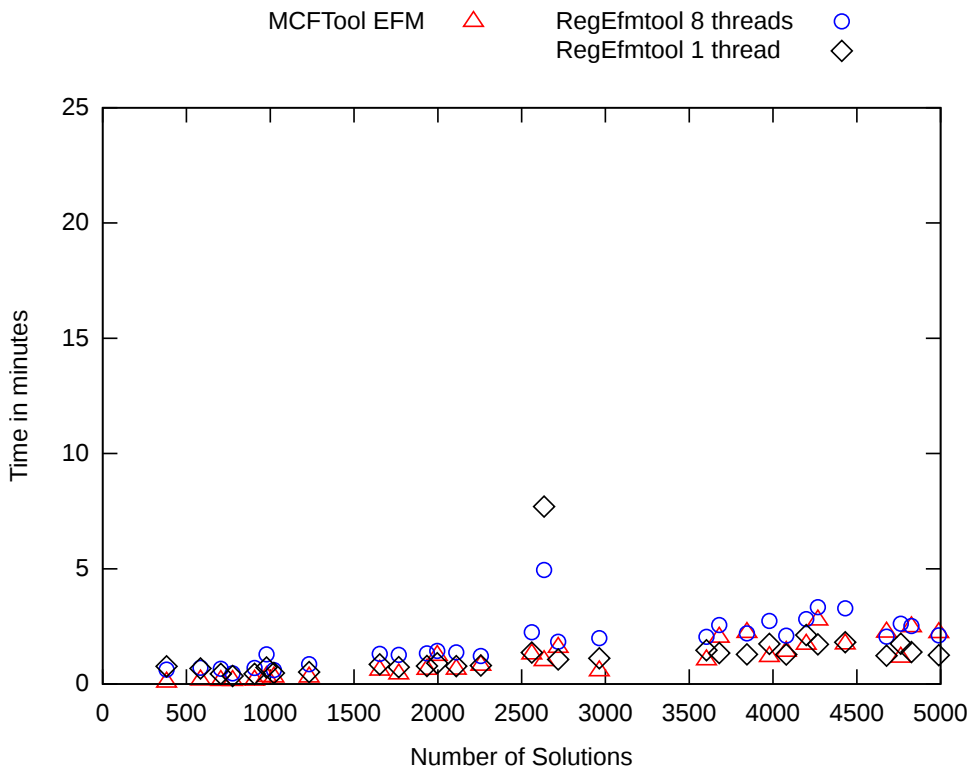


FIGURE 3.6: Temps d'exécutions des sélection de la forme $\neg a \vee \neg b$

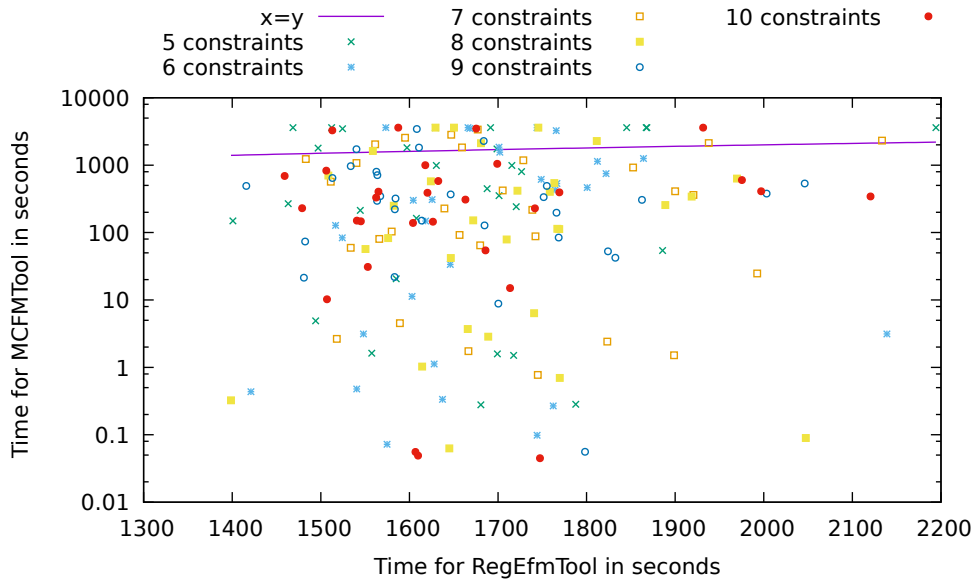


FIGURE 3.7: Temps d'exécution des sélections de la forme $\neg a \vee \neg b$ pour des requêtes composées de 5 à 10 clauses négatives

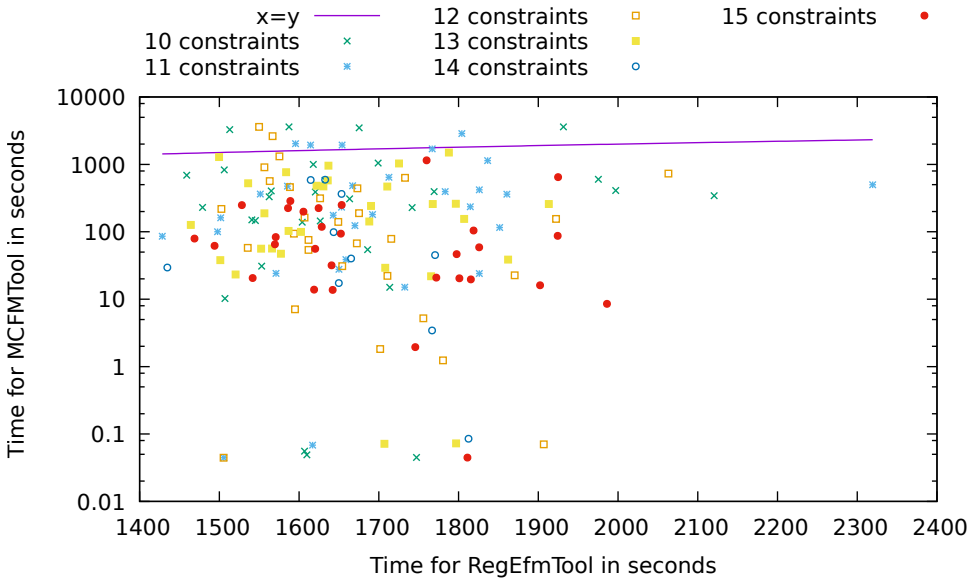


FIGURE 3.8: Temps d'exécution des sélections de la forme $\neg a \vee \neg b$ pour des requêtes composées de 10 à 15 clauses négatives

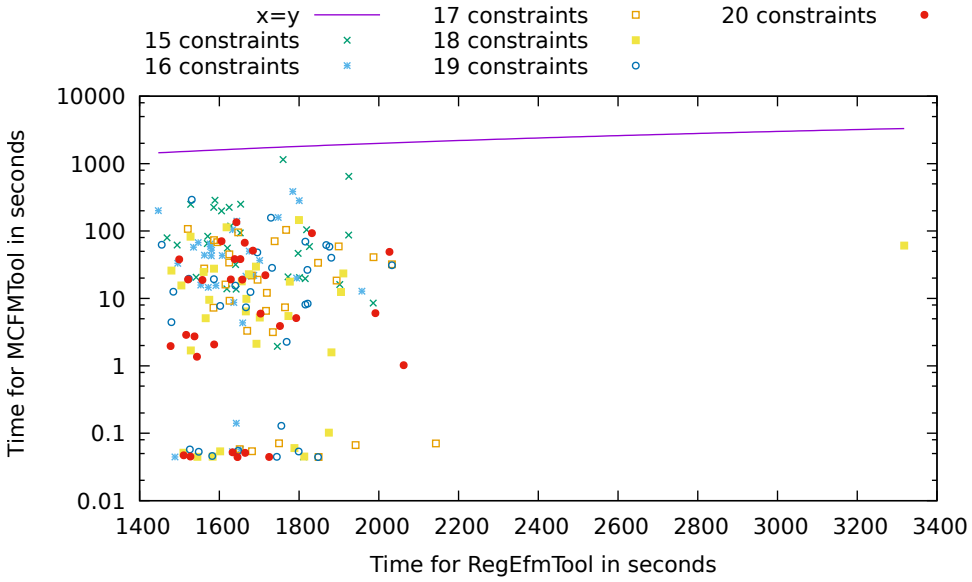


FIGURE 3.9: Temps d'exécution des sélections de la forme $\neg a \vee \neg b$ pour des requêtes composées de 15 à 20 clauses négatives

On voit clairement que lorsqu'on augmente le nombre de contraintes les performances de *MCFTool* s'améliorent, cela est dû à deux choses. Premièrement plus de contraintes permet au SMT de mieux filtrer les solutions et de ne pas juste faire de l'énumération, domaine dans lequel notre approche est clairement inférieure aux approches basées sur la double description. Deuxièmement le nombre de solutions est plus faible, en effet pour chaque taille de contraintes nous avons généré 30 requêtes aléatoirement sans nous assurer que le nombre de solutions soit réparti entre 1 et 5000.

3.4 Conclusion

MCFTool offre une solution alternative à RegEfmTool qui permet d'interroger, avec un langage de requête intelligible, des réseaux métaboliques de façon efficace pour toutes les requêtes. Il permet également d'interroger avec des requêtes des réseaux qui sont trop gros pour que l'ensemble des solutions puisse être calculé.

Le prix à payer pour permettre l'expressivité des requêtes et l'efficacité de *MCFTool* en présence de requêtes contraignant fortement le réseau se traduit par des performances très inférieures à RegEfmTool lorsqu'il n'y pas de requête ou que les requêtes ne contraignent pas suffisamment le réseau.

Chapitre 4

Les $MCFM$s

Dans les chapitres précédents nous nous sommes intéressé au calcul des EFMs et au calcul des EFMs respectant des contraintes.

Nous avons vu que toute voie peut se décomposer en EFMs et que les EFMs sont des voies minimales dans le cas général. Cependant l'ensemble des EFMs respectant les contraintes ne permet pas de reconstruire toutes les voies respectant les contraintes et toutes les voies minimales vis-à-vis des contraintes ne sont pas forcément des EFMs.

Dans ce chapitre nous allons introduire la notion de $MCFM$s pour « minimal constrained flux modes » (Morterol et al., 2016), il s'agit d'une notion, actuellement manquante dans l'étude des voies métaboliques, qui représente les voies minimales en présence de contraintes.

Il est également intéressant de noter que sans contraintes toutes les voies sont décomposables en voies minimales. Lorsqu'on utilise des contraintes ce n'est plus le cas, les propriétés de minimalité et d'indécomposabilité ne coïncident plus. Les $MCFM$s sont donc uniquement des voies minimales respectant les contraintes mais la composition de $MCFM$s ne permet pas de reconstruire toutes les voies respectant les contraintes. Il est possible de définir un ensemble de voies V, contenant tous les $MCFM$s, tel que toute voie soit une combinaison positive de $v_1 \dots v_i \in V$ mais cette notion ne sera pas traitée dans cette thèse et fera l'objet de futurs travaux.

De façon synthétique, les opérations de minimisation (notée « minimisation ») et de filtrage par les contraintes (notée « contraintes ») ne commutent pas :

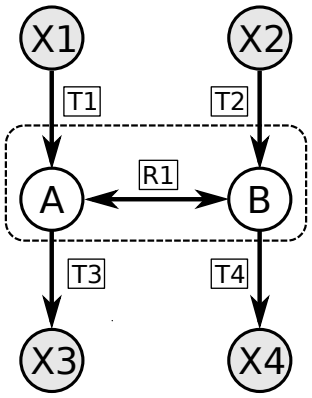
— EFM_c = contraintes(minimisation(Voies)), avec $EFM_c \subset EFM$

— $MCFM$ = minimisation(contraintes(Voies)), avec $EFM_c \subset MCFM$

Ce chapitre sera structuré comme ceci : nous présenterons dans un premier temps un exemple simple de $MCFM$ et nous modéliserons formellement la notion de $MCFM$ puis nous introduirons les algorithmes utilisés pour les calculer et enfin nous analyserons les temps de calcul des $MCFM$s à travers des résultats expérimentaux.

4.1 Exemple et formalisation

Pour notre exemple introductif, nous allons reprendre le réseau de la figure 3.3 :



Rappel de la figure 3.3

Ce réseau admet quatre EFMs :

(1) T1,T3,

(2) T2,T4,

(3) T1,R1,T4,

(4) T2,R1$_{rev}$,T3.

Si maintenant nous cherchons les voies contenant T1 et T2, aucun EFM ne les contient toutes les deux. Or il existe bien des flux respectant cette requête, par exemple {T1,T2,T3,T4}. Simplement filtrer les EFMs comme cela était fait auparavant ne permet donc pas de répondre à

cette requête. Cependant, si on combine des EFMs, on peut trouver des voies respectant cette contrainte :

(1)+(2) = T1,T3,T2,T4

(1)+(4) = T1,T2,R1$_{rev}$,T3

(3)+(2) = T1,R1,T2, T4

(3)+(4) = T1,R1,T4,T2,R1$_{rev}$,T3 = {T1,T4,T2,T3} = (1)+(2)

(1)+(2)+(3) = T1,T2,R1,T3,T4

(1)+(2)+(4) = T1,T2,R1$_{rev}$,T3,T4

On remarquera que les trois premières solutions sont minimales car on ne peut pas trouver de sous-ensemble strict à l'une d'elles qui soit à la fois une voie métabolique et qui respecte la contrainte, ce qui n'est pas le cas des deux dernières solutions. On appelle une solution minimale un $MCFM$. Ces $MCFM$s sont représentés figure 4.1.

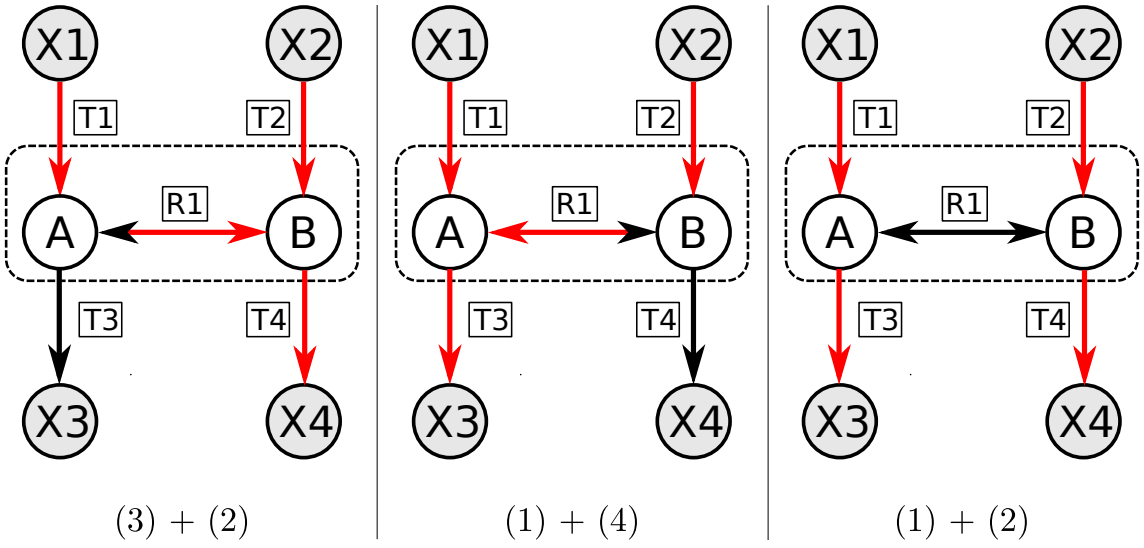


FIGURE 4.1: $MCFM$s satisfaisant la contrainte $T1 \wedge T2$ dans le réseau 3.3

Plus formellement on peut définir un $MCFM$ comme :

Definition 4.1 ($MCFM$) *Soit C^{rc} le sous-cône de C formé par l'ensemble des voies à l'état stationnaire respectant le sens des réactions irréversibles et les contraintes booléennes rc (pour « regulatory constraints ») tel que :*

$$C^{rc} = \{v \in C \mid rc(v)\} \tag{4.1}$$

Les $MCFMs$ sont les solutions ayant un support minimal :

$$M^{rc} = \{e \in C^{rc} \mid \nexists e' \in C^{rc}\ Supp(e') \subset Supp(e)\} \tag{4.2}$$

On remarquera que, d'une part, les $MCFMs$ étant des flux métaboliques, ils peuvent s'exprimer comme une combinaison d'$EFMs$, et d'autre part qu'un EFM qui satisfait la requête est donc un $MCFM$, puisqu'il est minimal par définition, cependant la réciproque est en toute généralité fausse. On notera aussi que C^{rc}, contrairement à C, n'est pas convexe en général et que, si un $MCFM$ est indécomposable dans C^{rc}, toute voie respectant la contrainte n'est pas forcment décomposable en plusieurs $MCFMs$ (contrairement au cas des $EFMs$).

Propriété 4.1 *Un $MCFM$ est une composition d'$EFMs$.*

Trivialement puisqu'un $MCFM$ est une voie, et que toute voie peut s'exprimer comme composition d'$EFMs$. On ne peut cependant rien dire des ces $EFMs$ vis-à-vis de la requête.

Propriété 4.2 *Un EFM qui satisfait les contraintes est un $MCFM$.*

Un $MCFM$ étant une voie minimale respectant les contraintes, si un EFM (qui par définition est minimal) respecte les contraintes alors celui-ci respecte la définition d'un $MCFM$.

Propriété 4.3 *Dans le cas général, un $MCFM$ n'est pas un EFM qui satisfait les contraintes.*

L'exemple introductif nous fournit un parfait exemple dans lequel un $MCFM$ est composé de plusieurs $EFMs$, nous montrons donc bien qu'un $MCFM$ peut ne pas être un EFM qui satisfait les contraintes.

Propriété 4.4 *Un $MCFM$ est indécomposable en plusieurs $MCFMs$.*

Par définition puisqu'un $MCFM$ est minimal le seul $MCFM$ pouvant être inclus dans un $MCFM$ est lui même.

Propriété 4.5 *Une voie respectant les contraintes peut ne pas être exprimable comme une combinaison de $MCFMs$.*

Nous pouvons trouver un contre exemple grâce à notre réseau figure 3.3. Supposons la requête « T1 », la voie {T1,T3,T2,T4} respecte cette requête, cependant cette voie ne contient qu'un seul $MCFM$: {T1,T3}. Elle est décomposable en un $MCFM$ plus un EFM ne respectant pas la contrainte.

Propriété 4.6 *Une voie respectant les contraintes contient forcément au moins un $MCFM$*

Soit une voie v respectant les contraintes, soit celle-ci est un $MCFM$ soit on peut trouver une voie v_i respectant les contraintes avec $Supp(v_i) \subset Supp(v)$, c'est-à-dire ayant au moins une réaction de moins. En répétant cette opération nous trouverons une voie v_j qui est minimale tout en respectant les contraintes, donc un $MCFM$.

Nous allons maintenant voir comment calculer les $MCFM$s.

4.2 Calcul des $MCFM$s

La principale différence en terme de calcul entre les EFMs et les $MCFM$s se trouve dans la fonction de minimisation, en effet celle-ci doit, dans le cas des $MCFM$s, prendre en compte la requête, ce qui n'est pas le cas pour les EFMs.

Nous allons présenter trois méthodes de minimisation permettant le calcul de $MCFM$s.

Algorithme de minimisation naïve

Dans l'algorithme 14, utilisant la minimisation naïve avec un SMT, la ligne 5 vérifie qu'il s'agit bien d'un EFM. Lors de cette étape, la solution a été minimisée en tenant compte des contraintes (il s'agit donc d'un $MCFM$) puis l'algorithme filtre après les $MCFM$s pour ne

garder que les EFMs. L'algorithme 20 issu de l'algorithme 14 est donc un premier algorithme naïf de calcul de $MCFM$s dans lequel nous avons simplement retiré le contrôle de minimalité en termes d'EFMs.

Algorithm 20 CalculeMCFM(réseau, requête)
Input : Un réseau métabolique valide et une requête sur ce réseau.
Output : L'ensemble des $MCFM$s contenus dans le réseau.

1: $formule = initialise(réseau, requête)$
2: $sol = \{\}$
3: **while** (SMT.check(formule) == true) **do**
4: % Algorithme 13, sous-section 3.2.1
5: $solMin = MinimisationNaiveSMT(SMT.getSolution())$
6: $sol = sol \cup \{solMin\}$
7: $SMT.addConstraint(\bigvee_{r \in solMin} \neg r)$
8: **end while**
9: **return** *sol*

Correction de l'algorithme 20 Puisque l'encodage nous assure que toute solution modélise une voie à l'état stationnaire (propriété 3.1) et que la méthode « MinimisationNaiveSMT » retourne une solution minimale respectant la requête alors la méthode « MinimisationNaiveSMT » retourne un $MCFM$.

Complétude de l'algorithme 20 Il existe un nombre fini de $MCFM$s dans un réseau métabolique, de plus puisqu'ils sont minimaux pour l'inclusion la propriété 2.3 qui assure que l'on peut bloquer un EFM e_1 sans bloquer d'EFM e_2 tel que $e_1 \neq e_2$ est transposable aux $MCFM$s. Par construction tous les $MCFM$s respectent la formule initiale, en les bloquant un à un nous assurons de les trouver tous et que la l'algorithme ne retourne plus de solution une fois tous les $MCFM$s trouvés.

On retrouve dans l'algorithme 20 (que nous appellerons algorithme naïf) le même problème qu'avec l'algorithme 14 c'est-à-dire que chaque solution initiale sera minimisée en un seul $MCFM$, ainsi si la solution initiale du SMT est une composition de nombreux $MCFM$s, il devra être appelé plusieurs fois, et donc demandera plus de recherche au SMT dans un cadre moins contraint que celui de la minimisation, ce qui affectera les performances. Par contre, l'algorithme naïf assure de n'avoir à démontrer qu'une fois UNSAT par solution, il est donc

particulièrement efficace lorsque le SMT trouve des solutions qui sont, soit minimales, soit non décomposables en plusieurs $MCFM$s.

La figure 4.2 schématise le fonctionnement de l'algorithme 20.

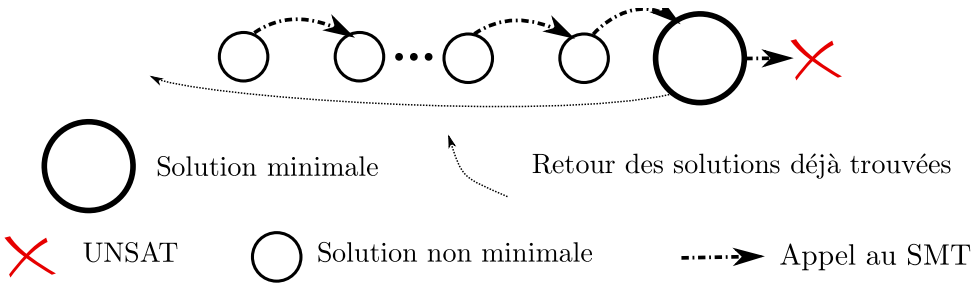


FIGURE 4.2: Déroulement schématique d'une minimisation naïve

Algorithme de minimisation partielle

Afin d'améliorer les performances de la minimisation, nous pouvons appliquer une approche similaire à celle que nous avons utilisée pour le calcul des EFMs dans l'algorithme 16 consistant à partir d'une première solution S_0 trouvée par le SMT, à chercher une solution S_1, satisfaisant les contraintes, telle que $S_1 \subset S_0$ puis trouver une solution S_2 telle que $S_2 \subset S_0, S_2 \not\subset S_1, S_1 \not\subset S_2$ en soustrayant S_1 à S_0. La différence avec l'approche de l'algorithme 16 est que l'on ne peut plus utiliser le calcul matriciel car celui-ci n'intègre pas les contraintes pour la minimisation. Nous allons donc utiliser le SMT pour réaliser la minimisation partielle. La version de l'algorithme 16 utilisant le SMT est présentée dans l'algorithme 21 (que nous appellerons minimisation partielle).

Discussion sur l'algorithme 21. Nous avons vu que toute affectation ν satisfaisant les contraintes 3.1 à 3.5 modélise une voie métabolique à l'état stationnaire et que, s'il n'existait pas de voie métabolique satisfaisant les contraintes, la formule était insatisfaisable. L'ajout de contraintes sur les réactions ne change pas le fait qu'une solution modélise une voie à l'état stationnaire, ainsi en forçant le SMT à trouver une solution S_1, telle que S_1 utilise uniquement

Algorithm 21 extraire_sous_solution(S_0)
Input : Un vecteur solution $S_0 \in C$
Output : une sous-solution de S_0 ou {} s'il n'en existe pas

1: $SMT.checkPoint()$
2: $SMT.addConstraint(\bigvee_{r \in Supp(S_0)} r = 0)$
3: $SMT.addConstraint(\bigwedge_{r \notin Supp(S_0)} r = 0)$
4: $retour = \{\}$
5: **if** $SMT.isSAT()$ **then**
6: $\quad retour = SMT.getSolution()$
7: **end if**
8: $SMT.RestorecheckPoint()$
9: **return** $retour$

des réactions de S_0 et au moins une réaction de moins que S_0, nous nous assurons de trouver une voie incluse dans S_0 s'il en existe une, et de trouver UNSAT si-non.

Cette version de « extraire_sous_solution(S_0) » peut être utilisée dans l'algorithme 22, dont la structure est proche de l'algorithme 17.

Algorithm 22 Minimise$MCFM(S_0)$
Input : $S_0 \in C$ (au premier appel $S_0 \in C^{cr}$, mais pas en général dans les appels récursifs.)
Output : Un ensemble de $MCFMs$ inclus dans S_0

1: $S_1 = extraire\_sous\_solution(S_0)$
2: **if** $S_1 == \{\}$ **then**
3: $\quad$ **if** checkRequete(S_0) **then**
4: $\quad\quad SMT.addConstraint(\bigvee_{R \in Supp(sol)} R = 0)$
5: $\quad\quad$ **return** $\{S_0\}$
6: $\quad$ **else**
7: $\quad\quad$ **return** {}
8: $\quad$ **end if**
9: **end if**
10: $S_1 = MinimiseMCFM(S_1)$
11: $S_2 = subtract(S_0, S_1)$
12: **return** $S_1 \cup MinimiseMCFM(S_2)$

Discussion sur l'algorithme 22.

Correction : L'algorithme 21 nous assure que $S_1 = \{\}$ uniquement si S_0 ne peut pas être minimisé. De plus nous retournons S_0 uniquement s'il respecte les contraintes. Nous sommes donc sûr que les retours sont des $MCFMs$.

Complétude : Ignorons toutes les solutions trouvées par soustraction, il reste une pile d'appels

au SMT supprimant une ou plusieurs réactions jusqu'a ce qu'on ne puisse plus réduire la solution. Si S_0 respecte les contraintes la minimisation grâce au SMT finira par trouver un $MCFM$ et le retourner. La sortie contiendra donc au moins un $MCFM$ inclus dans S_0 s'il en existe un.

La ligne 3 permet de faire la différence entre une solution minimale valide et une solution minimale invalide. Lors de la soustraction de voies nous n'avons pas la garantie que S_2 respecte les conditions et si celle-ci ne les respecte pas nous n'avons pas non plus la garantie que S_2 ne contienne aucune voie ne respectant les contraintes. La vérification doit donc se faire une fois que le SMT n'est plus capable de minimiser la solution.

Le comportement de cette minimisation est schématisé par la figure 4.3.

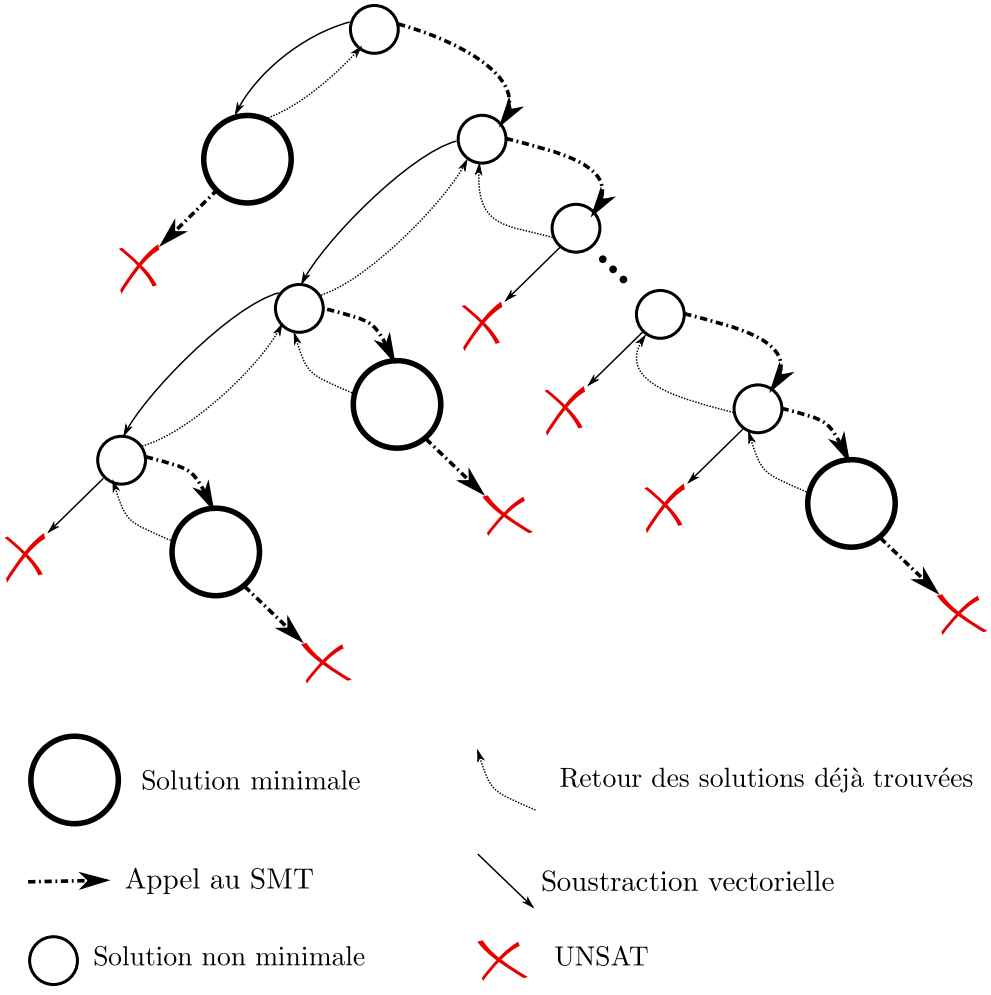


FIGURE 4.3: Déroulement schématique d'une minimisation « partielle »

La minimisation partielle doit essayer, après chaque soustraction, de minimiser le résultat.

Or la soustraction se fait sans tenir compte des contraintes et engendre souvent des solutions invalides ne pouvant pas être minimisées. Cela implique un grand nombre de démonstrations d'UNSAT, cependant ces démonstrations se font dans un cadre extrêmement contraint, puisque la recherche de sous-solution se fait en affectant à 0 toutes les réactions qui ne sont pas dans la solution initiale, et est peu coûteuse.

Cette méthode est efficace lorsqu'il y a plusieurs $MCFMs$ dans la solution initiale et que les coupures de branche sont triviales.

Algorithme de minimisation totale

L'utilisation du SMT pour la minimisation nous permet, contrairement à l'approche matricielle, de trouver directement une solution minimale directement comment nous l'avons fait dans la minimisation naïve.

Nous pouvons donc écrire un algorithme utilisant à la fois la minimisation complète de l'algorithme 20 tout en trouvant plusieurs $MCFMs$ comme dans la minimisation partielle.

Algorithm 23 extraire_sous_solution_total(S_0)
Input : Un vecteur solution $S_0 \in C$
Output : un $MCFM$ inclus dans S_0 ou {} s'il n'en existe pas

1: $retour = \{\}$
2: $SMT.checkPoint()$
3: **while** $SMT.isSAT()$ **do**
4: $\quad retour = SMT.getSolution()$
5: $\quad SMT.addConstraint(\bigvee_{r \in Supp(sol)} r = 0)$
6: $\quad SMT.addConstraint(\bigwedge_{r \notin Supp(sol)} r = 0)$
7: **end while**
8: $SMT.RestorecheckPoint()$
9: **return** $retour$

Discussion sur l'algorithme 23.

Cet algorithme peut se voir comme la répétition de l'algorithme 21, il a donc les mêmes propriétés de correction et complétude.

L'algorithme 23 est très proche de l'algorithme 13, la principale différence est qu'il retournera

un ensemble vide si l'ensemble S_0 ne contient pas de $MCFM$. Grâce à cela nous pouvons écrire la version suivante de « Minimise ».

Algorithm 24 Minimise_total(S_0)
Input : $S_0 \in C$
Output : un ensemble de $MCFM$s inclus dans S_0

```
1: S1 = extraire_sous_solution_total(S0)
2: retour = {}
3: if S1 ≠ {} then
4:    retour = S1
5:    S2 = subtract(S0, S1)
6:    if S2 ≠ {} then
7:       retour = retour ∪ Minimise_total(S2)
8:    end if
9: end if
10: return retour
```

Discussion sur l'algorithme 24.

Correction : Les solutions retournées par cet algorithme sont des solutions retournées par le SMT, respectant donc les équations 3.1 à 3.5 modélisant une voie et satisfaisant les contraintes, de plus ces solutions ont été minimisées, il s'agit donc de $MCFM$s.

Complétude : Ignorons toutes les solutions trouvées par soustraction, il reste un appel à l'algorithme 23 retournant un $MCFM$. La sortie contiendra donc au moins un $MCFM$ inclus dans S_0 s'il en existe un.

La minimisation totale assure de trouver N solutions avec $N + 1$ UNSAT ; de même que dans le cas de la minimisation partielle le dernier UNSAT peut être trivial. Cette méthode est plus efficace que la minimisation partielle lorsque les $MCFM$s sont sur « des branches opposées » car il évite de vérifier la soustraction à chaque minimisation. Cependant, il partage également ses faibles performances lorsque la solution initiale ne peut pas être décomposée en plusieurs $MCFM$s.

Nous pouvons donc écrire un algorithme utilisant soit la minimisation partielle soit la minimisation totale calculant l'ensemble des $MCFM$s.

Remarque : Dans l'algorithme 25 les fonctions de minimisation sont chargées de bloquer les

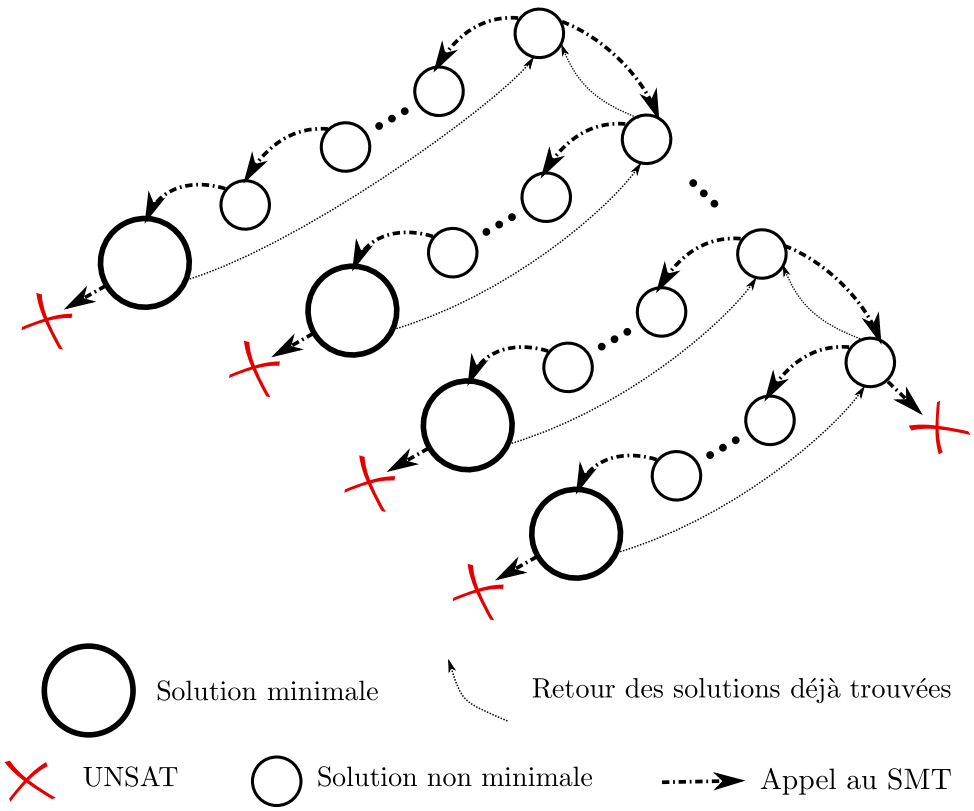


FIGURE 4.4: Déroulement schématique d'une minimisation « totale »

Algorithm 25 CalculeMCFM(réseau, requête, *minimisation*)
Input : Un réseau métabolique valide et une requête sur ce réseau. *minimisation* est une fonction de minimisation (totale ou partielle).
Output : L'ensemble des $MCFM$s contenue dans le réseau.

1: $formule = initialise(réseau, requête)$
2: $sol = \{\}$
3: **while** (SMT.check(formule) == true) **do**
4: $\quad listeMCFM = minimisation(SMT.getSolution())$
5: $\quad sol = sol \cup \{listeMCFM\}$
6: **end while**
7: **return** *sol*

$MCFM$s (et uniquement les $MCFM$s) trouvés lors de la minimisation. Ainsi la démonstration de la correction et de la complétude de cet algorithme est la même que celle de l'algorithme 20.

4.3 Expérimentations

Dans cette section nous allons analyser et comparer les performances des différentes implémentations de l'algorithme de calcul des $MCFM$s. Dans un premier temps, nous nous intéresserons aux requêtes positives unitaire, cas dans lequel les $MCFM$s et les EFMs ne sont pas égaux, puis nous étudierons le cas où les EFMs et les $MCFM$s sont confondus, plus particulièrement les requêtes négatives. Nous en profiterons pour comparer le calcul des EFMs et des $MCFM$s dans ce cas.

Toutes les expérimentations de cette section sont faites sur le même réseau métabolique que les expérimentations de la sous-section 3.3.2. Cette fois encore nous travaillons sur le réseau déjà compressé.

Lors des tests sur les EFMs nous avons comparé des requêtes positives unitaires, des requêtes négatives unitaires et des requêtes de la forme $\neg a \vee \neg b$, dans les deux derniers types de requêtes les $MCFM$s et les EFMs correspondent. En effet, pour que les EFMs et les $MCFM$s ne soient pas confondus, il faut pouvoir trouver un ensemble de chemins qui individuellement ne satisfont pas la requête mais dont l'union la respecte. Dans le cas d'une requête qui impose uniquement l'absence d'enzyme (ou une disjonction d'absences d'enzymes) pour que la combinaison de chemins respecte la requête il faut que chaque chemin la respecte également, ce qui en fait un EFM valide ; de ce fait, les $MCFM$s sont confondus avec les EFMs pour les requêtes négatives.

Nous traiterons le cas où les EFMs et les $MCFM$s sont confondus dans la sous-section 4.3.2 et nous nous concentrerons dans la sous-section 4.3.1 uniquement sur des requêtes positives unitaires.

4.3.1 Étude du calcul des $MCFM$s sur des requêtes positives unitaires

Pour la comparaison des versions de la minimisation des $MCFM$s nous allons utiliser des requêtes composées de conjonction de littéraux positifs, nous ferons varier la taille de ces requêtes afin d'avoir des requêtes de moins en moins contraintes et donc de plus en plus de résultats. Pour chacune des tailles de requêtes, les résultats indiqués sont la moyenne du temps d'exécution de 30 requêtes différentes dont les enzymes sélectionnées sont choisies aléatoirement. Toutes les exécutions ont un « timeout » d'une heure.

La figure 4.5 présente la production de $MCFM$s par seconde. On remarque que la minimisation partielle est, quelle que soit la taille de la contrainte, la moins efficace. La minimisation totale est la plus rapide lorsque le réseau est fortement contraint et la minimisation naïve devient plus performante une fois que la taille de la requête est inférieure à 22.

Pour le cas où la taille de la contrainte est inférieure à 22, nous pouvons expliquer le classement des algorithmes naïf > total > partiel par le fait que, comme nous sommes dans le cas de requêtes positives, lorsque l'on fait une soustraction de voies il y a de fortes chances que le résultat ne satisfasse plus les contraintes. Ainsi en moyenne toutes les minimisations retourneront un seul $MCFM$. La minimisation partielle cherchant le plus de minimisations possibles perd donc le plus de temps suivie par la minimisation totale dont la politique de minimisation est moins coûteuse et enfin la plus rapide est la minimisation naïve qui ne cherche pas à minimiser en plusieurs solutions.

Pour conforter cette analyse nous pouvons regarder le tableau 4.1 qui présente les résultats de requêtes sélectionnées parmi les requêtes ayant eu un nombre important de $MCFM$s. Ce tableau présente pour les algorithmes naïf et total, le nombre de $MCFM$s produit en une heure (colonne « nb $MCFM$s ») avec la moyenne de solutions par minimisation sur les 1 000 premières minimisations (colonne « moyenne ») ainsi que le maximum de solutions trouvées lors d'une

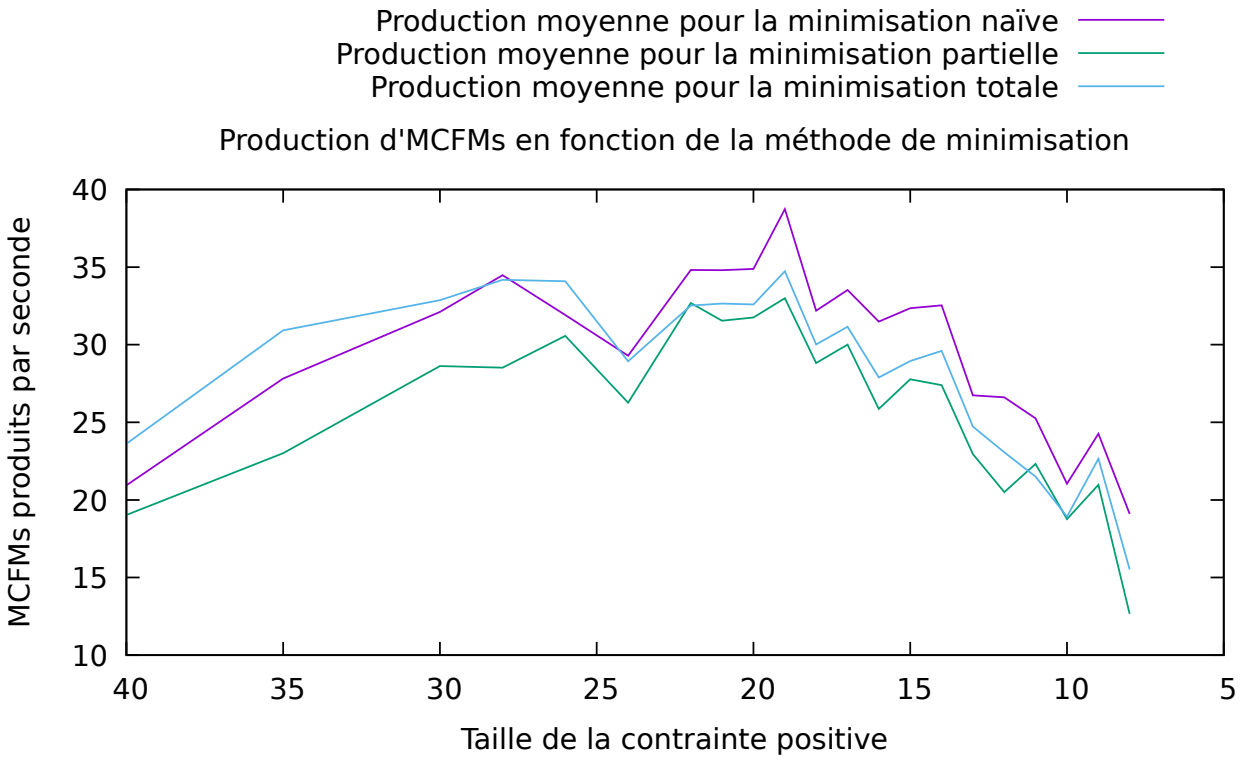


FIGURE 4.5: Production moyen de $MCFM$ par seconde pour les différentes approches en fonction de la taille de la contrainte

minimisation (colonne « max »). Nous indiquons également le nombre de solutions par minimisation pour les dix premières minimisations ainsi que la moyenne sur ces dix minimisations (notée « $\bar{x}$ »).

Algorithme	taille requête	nb $MCFM$s	moyenne	max
Total	12	11461	1.03796	2
10 premières minimisations : 2 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1.1)				
Partiel	12	11461	1.05894	4
10 premières minimisations : 1 1 1 1 1 1 1 2 1 1 ($\bar{x}$: 1.1)				
Total	12	21369	1.11089	4
10 premières minimisations : 2 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1.1)				
Partiel	12	21369	1.12488	4
10 premières minimisations : 4 2 1 1 1 1 1 1 1 1 ($\bar{x}$: 1.4)				
Total	12	5491	1.04496	3
10 premières minimisations : 1 2 1 1 1 1 1 1 2 1 ($\bar{x}$: 1.2)				
Partiel	12	5491	1.05794	3
10 premières minimisations : 1 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1)				
Total	13	43217	1.06394	3
10 premières minimisations : 1 1 2 1 1 1 1 1 1 1 ($\bar{x}$: 1.1)				
Partiel	13	43217	1.05994	6
10 premières minimisations : 6 1 2 1 1 1 1 1 1 1 ($\bar{x}$: 1.6)				
Total	14	8305	1.06593	3
10 premières minimisations : 1 1 3 2 1 2 1 1 1 1 ($\bar{x}$: 1.4)				
Partiel	14	8305	1.08891	4
10 premières minimisations : 2 1 1 1 1 1 2 1 1 1 ($\bar{x}$: 1.2)				
Total	15	3889	1.02198	3
10 premières minimisations : 1 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1)				
Partiel	15	3889	1.02498	5
10 premières minimisations : 5 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1.4)				
Total	16	2245	1.08092	5
10 premières minimisations : 3 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1.2)				
Partiel	16	2245	1.06194	7
10 premières minimisations : 7 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1.6)				
Total	18	3162	1.04496	3
10 premières minimisations : 1 1 1 1 1 1 1 1 1 1 ($\bar{x}$: 1)				
Partiel	18	3162	1.06194	4
10 premières minimisations : 1 2 1 1 2 2 2 1 1 1 ($\bar{x}$: 1.4)				

TABLE 4.1: Minimisation partielle et totale pour des requêtes positives

On remarque que la moyenne sur les 1 000 premières minimisations est proche de un, avec une moyenne très légèrement plus élevée pour la minimisation partielle, de plus le nombre maximal de résultats par minimisation est toujours plus grand (ou égal) lors de la minimisation partielle.

On remarquera également que dans la majorité des cas, la moyenne des dix premières minimisations est supérieure à la moyenne sur 1 000, l'efficacité de la minimisation est donc décroissante. Cela nous explique pourquoi la minimisation totale est plus efficace lorsque le réseau est plus fortement contraint car il y a moins de solutions, donc la proportion de minimisations réussies est plus grande.

Nous nous sommes intéressé à l'impact du choix de la minimisation sur les performances. Nous allons maintenant regarder sur quelle partie du calcul notre algorithme passe le plus de temps dans chacune des méthodes grâce aux figures 4.6, 4.7 et 4.8 qui présentent, respectivement, la répartition du temps de calcul pour les méthodes naïve, totale et partielle. Le champ « recherche » représente le temps passé dans le SMT pour trouver une solution S_0, les champs « SAT » et « UNSAT » représentent le temps passé dans le SMT pour minimiser S_0.

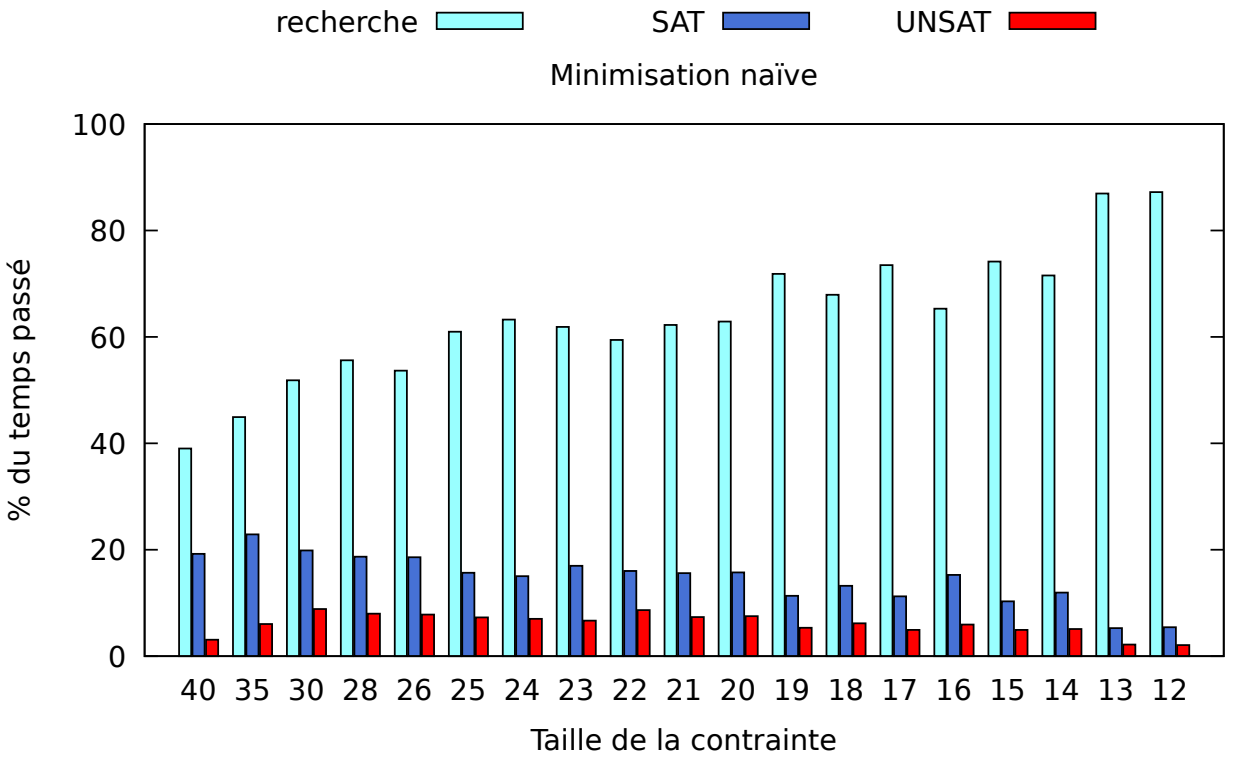


FIGURE 4.6: Répartition moyen du temps de calcul pour la minimisation naïve.

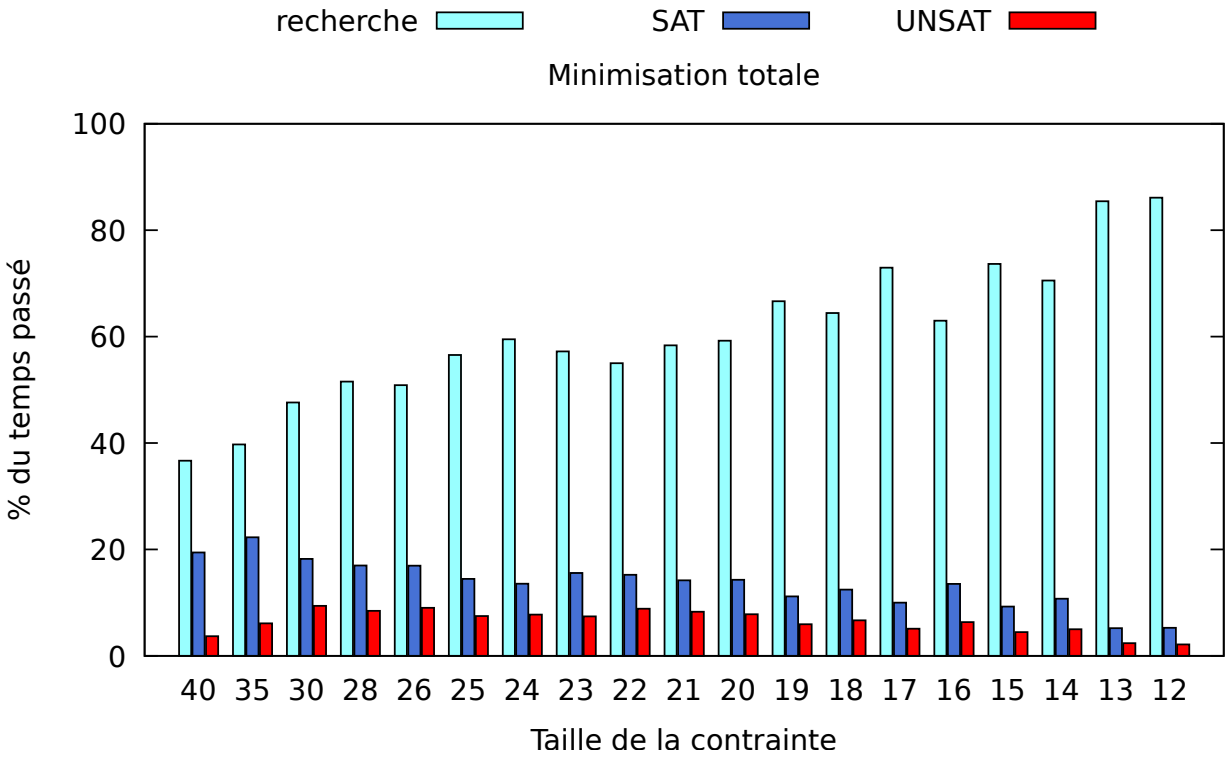


FIGURE 4.7: Répartition moyen du temps de calcul pour la minimisation totale.

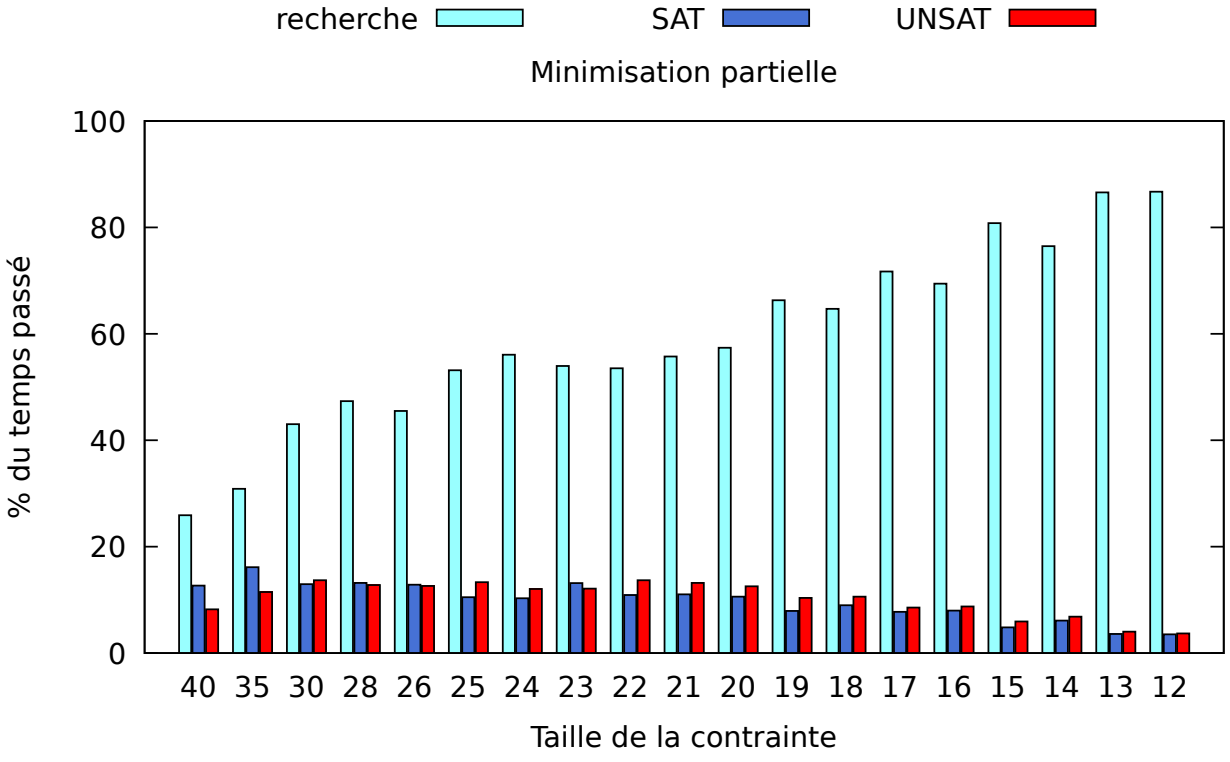


FIGURE 4.8: Répartition moyen du temps de calcul pour la minimisation partielle.

On remarquera que la plus grande partie du temps est passé dans la recherche de S_0 et que la répartition du temps de calcul entre la minimisation naïve et totale est sensiblement la même, ce qui est normal puisque dans notre cas la minimisation totale va être majoritairement une minimisation naïve plus une soustraction suivie d'un UNSAT.

Dans le cas de la minimisation partielle, figure 4.8, on peut clairement voir le surcoût dû aux démonstrations UNSAT répétitives.

Pour finir avec l'étude du comportement des $MCFM$s le graphe 4.9 présente le nombre de solutions trouvé en une heure ainsi que le nombre de timeouts (d'une heure) moyen, sur 30 requêtes faiblement contraintes (en moyenne 4 contraintes par requête), lorsqu'on fait varier la taille maximale des $MCFM$s. Nous avons limité la taille des $MCFM$s à 28 car à partir de cette taille le nombre de timeouts est fixe, le nombre de solution n'évoluera donc plus significativement.

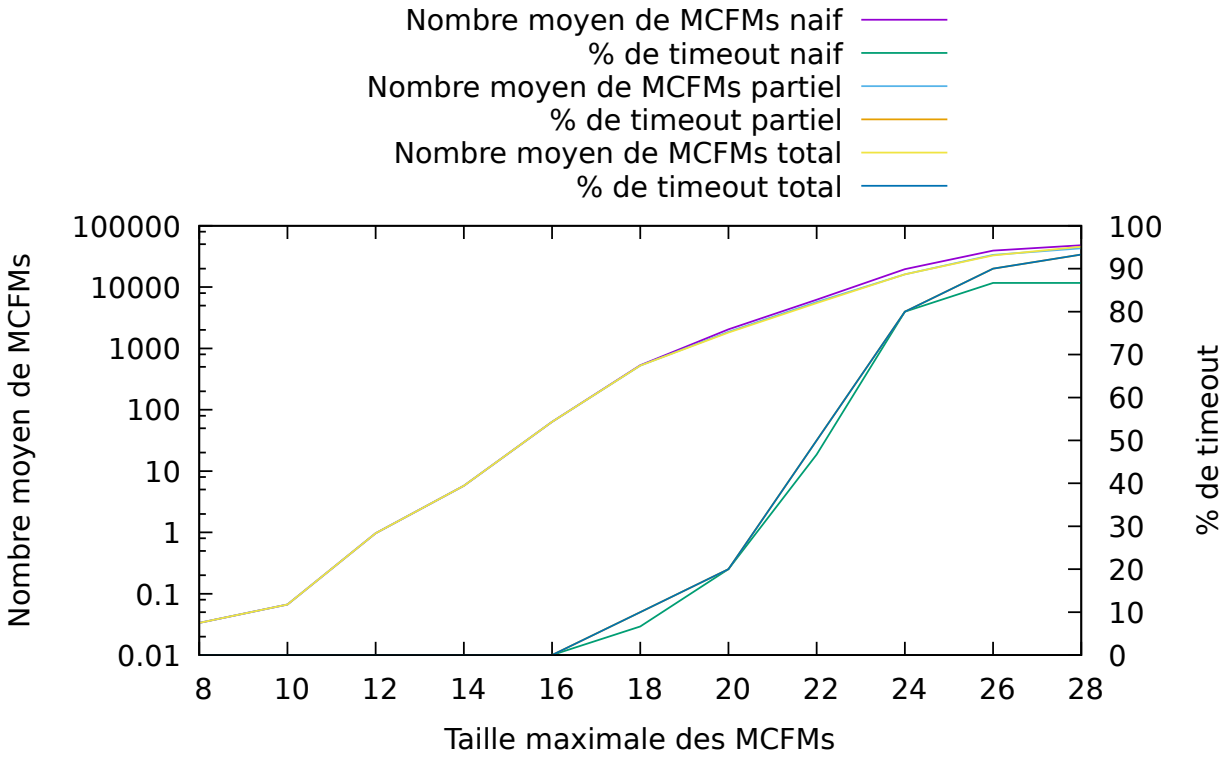


FIGURE 4.9: Nombre moyen de $MCFM$s et pourcentage de timeout en fonction de la taille maximale des $MCFM$s.

Le pourcentage de timeouts est mesuré à 93% et à 86% car deux des requêtes ont un nombre de

$MCFMs$ suffisamment petit pour pouvoir être calculé en moins d'une heure sans contraintes de taille et une troisième requête peut être calculée en moins d'une heure, mais uniquement avec l'algorithme naïf. Le principal apport de cette courbe est de montrer comment contraindre les $MCFMs$ afin d'obtenir un nombre de solutions traitables par un humain, en effet sur les requêtes positives faiblement contraintes le nombre de $MCFMs$ est largement plus grand que le nombre d'$EFMs$. Cette figure montre aussi les limites de performances de notre algorithme avec l'explosion des timeouts lorsque la taille maximale des $MCFMs$ dépasse la vingtaine. Nous remarquerons également la différence du nombre de résultats entre le calcul des $MCFMs$ et des $EFMs$ puisque, sans limites de taille, la requête ayant le plus d'$EFMs$ a moins de 5000 $EFMs$.

4.3.2 Étude des $MCFMs$ sur des requêtes négatives unitaires et de la forme $\neg a \vee \neg b$

Nous allons maintenant nous intéresser au cas où les $MCFMs$ et les $EFMs$ sont confondus. Cela nous permettra d'une part de nous situer par rapport au calcul des $EFMs$ et d'autre part de continuer la comparaison entre les différentes méthodes de minimisation.

On constate, dans la figure 4.10, que dans le cas des requêtes négatives unitaires le calcul des $MCFMs$ est plus rapide que RegEfmTool, quelle que soit la méthode de minimisation, cela s'explique simplement, encore une fois, par l'incapacité de RegEfmTool à tenir compte efficacement de ce type de requête. En effet, dès lors que l'on utilise des requêtes que RegEfmTool peut efficacement traiter, par exemple de la forme $\neg a \vee \neg b$ (figure 4.11) le calcul des $MCFMs$ est significativement plus lent que le calcul des $EFMs$ avec RegEfmTool.

Dans les deux cas le calcul des $MCFMs$ est toujours plus lent que le calcul des $EFMs$, ce qui est normal puisque le calcul des $MCFMs$ demande plus de vérifications lors de la minimisation.

L'information la plus intéressante de ces deux figures est le fait que les méthodes de minimisation pour les $MCFMs$ ont des performances inverses par rapport aux requêtes positives. En effet, la méthode naïve semble être systématiquement la plus lente suivie de la méthode totale et

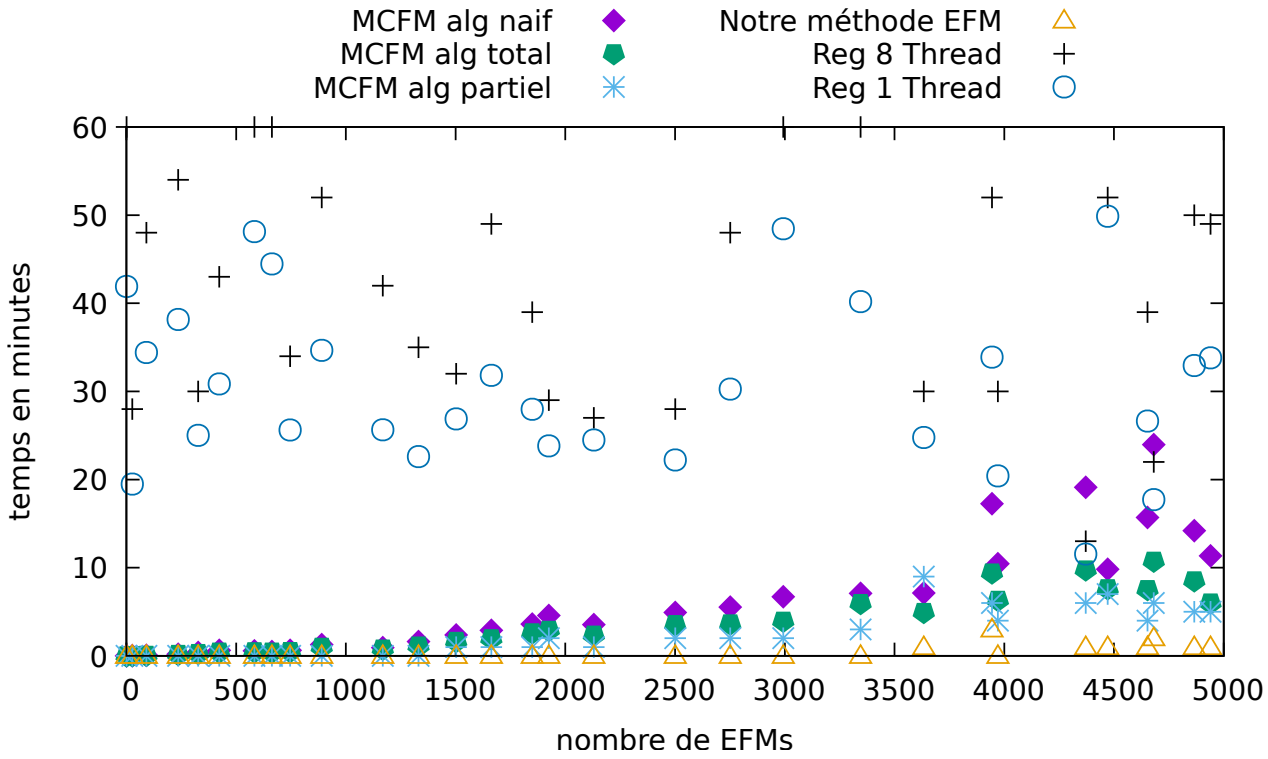


FIGURE 4.10: Temps en fonction du nombre de solutions pour les requêtes négatives unitaires

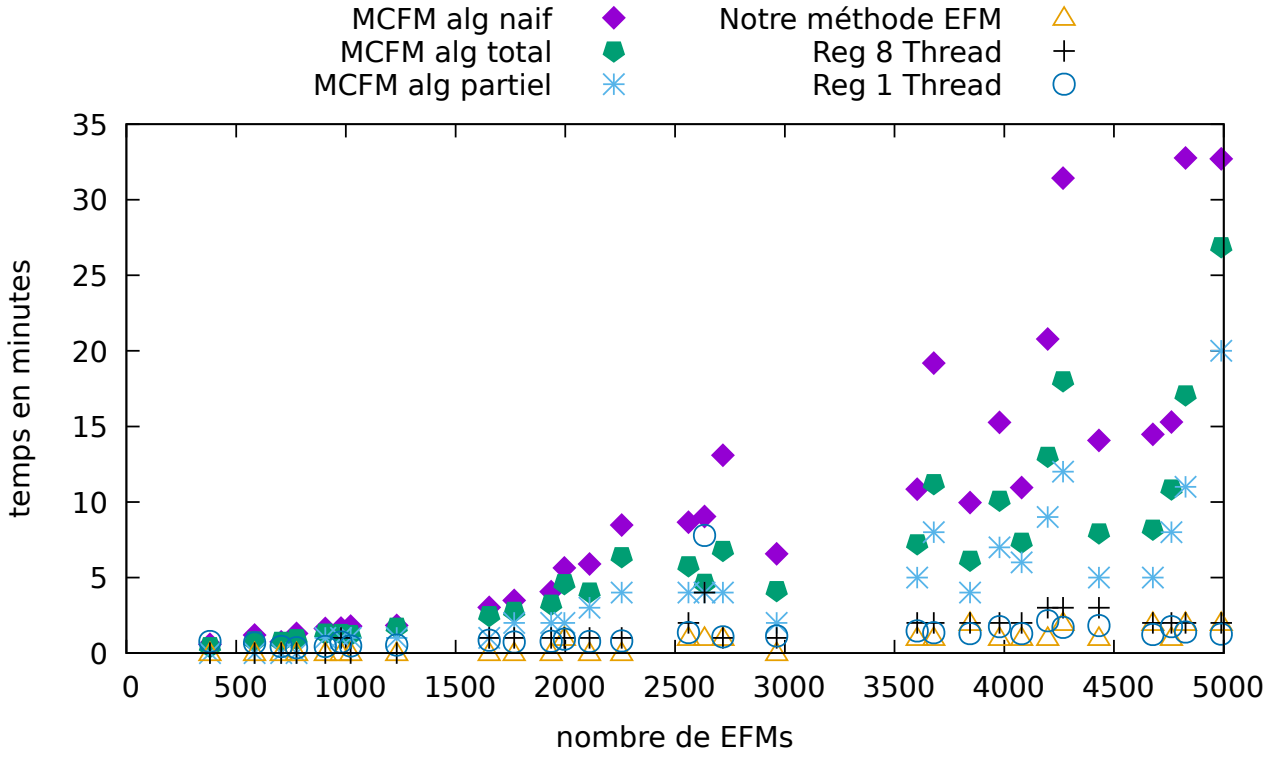


FIGURE 4.11: Temps en fonction du nombre de solution pour les requêtes forme $\neg a \vee \neg b$

la méthode partielle est toujours la plus rapide. Il convient donc de se poser les questions suivantes :

— **Pourquoi est ce que les méthodes de minimisation multiple sont plus efficaces dans ce cas ?** La minimisation naïve, retournant toujours un résultat, ne peut être moins efficace dans ce cas, ce sont donc les autres méthodes qui sont plus efficaces. Nous sommes dans le cas où les EFMs et les $MCFM$s sont confondus, ce qui implique que tout ensemble de solutions respectant la requête est composé d'EFMs respectant également la requête. De ce fait lors de la minimisation les branches issues d'une soustraction de voie permettront systématiquement de trouver des voies valides. Ce qui va naturellement augmenter le nombre de $MCFM$s trouvés par solution initiale et donc accélérer le processus.

— **Qu'est ce qui fait que la minimisation partielle soit sensiblement plus rapide ?** La minimisation partielle permet d'explorer plus de branches que la méthode totale. Puisque dans ce cas toutes les branches respecteront la requête la méthode partielle profite donc plus de la particularité de la requête. Le tableau 4.2 nous permet de voir le nombre d'EFMs moyen (pour les 1 000 premières minimisations) par solution initiale pour les requêtes négatives unitaires, ce qui confirme le fait que la méthode partielle permet une meilleure décomposition des solutions initiales, cet avantage est surtout marqué au début lorsque les solutions initiales sont des compositions d'un nombre important d'EFMs. Cependant l'avantage gagné en début de recherche est suffisant pour réduire significativement le temps de recherche des futures solutions initiales.

Algorithme	taille requête	nb $MCFM$s	moyenne	max
Total	14	3345	1.7992	20
10 premières minimisations : 20	1	2	1	1	1	2	4	2	2	($\bar{x}$: 3.6)
Partiel	14	3345	1.88212	80
10 premières minimisations : 80	3	2	1	1	2	1	2	2	2	($\bar{x}$: 9.6)
Total	15	4940	2.40959	18
10 premières minimisations : 18	4	1	3	3	7	4	4	2	2	($\bar{x}$: 4.8)
Partiel	15	4940	2.37662	65
10 premières minimisations : 65	1	1	4	1	4	3	2	2	4	($\bar{x}$: 8.7)
Total	1	4371	2.07692	22
10 premières minimisations : 22	2	1	4	1	1	1	1	3	2	($\bar{x}$: 3.8)
Partiel	1	4371	2.08092	61
10 premières minimisations : 61	1	1	5	1	1	2	1	1	1	($\bar{x}$: 7.5)
Total	4	2130	1.91808	14
10 premières minimisations : 14	3	1	1	3	3	2	1	1	1	($\bar{x}$: 3)
Partiel	4	2130	1.81518	29
10 premières minimisations : 29	3	1	1	2	1	3	2	1	1	($\bar{x}$: 4.4)

TABLE 4.2: Minimisation partielle et totale pour des requêtes négatives unitaire

Le tableau 4.2 est construit de la même manière que le tableau 4.1, on a cette fois choisi les requêtes négatives unitaires ayant le plus de $MCFM$s. On constate encore une fois un maximum de solutions par minimisation plus élevé pour la minimisation partielle. Cependant la moyenne sur 1 000 minimisations est proche de 2, ce qui explique les meilleurs performances des méthodes cherchant plusieurs $MCFM$s dans S_0. Cette fois encore la moyenne des 10 premières solutions est largement au dessus des 1 000 premières, avec une première minimisation qui est systématiquement maximale.

4.4 Conclusion

Le calcul des $MCFM$s diffère de celui des EFMs uniquement par la méthode de minimisation qui prend en compte la requête. Bien que la minimisation, quelle que soit la méthode utilisée, ne soit pas une phase très chronophage, moins de 10% du temps de calcul lorsque le nombre d'$MCFM$s est significatif, la façon de minimiser peut influencer significativement les performances en fonction du type de requête. Les requêtes négatives profitent particulièrement bien de la minimisation partielle à l'inverse des requêtes positives qui préfèrent une minimisation naïve car nos algorithmes sont en moyenne en mesure d'extraire des solutions initiales qu'un seul $MCFM$.

Dans le cas des requêtes positives (et de toutes requêtes pouvant être satisfaites par une voie sans être satisfaites par les EFMs qui la composent) les $MCFM$s sont un nouvel outil à la disposition des biologistes pour comprendre le fonctionnement des cellules, permettant l'énumération de voies lors de contraintes complexes.

Conclusion

5.1 Contributions

Cette thèse a permis d'étendre les travaux de (Soh et al., 2012) sur l'utilisation d'un SAT solveur pour le calcul des EFMs et de proposer une première approche de prise en compte de la stœchiométrie. Cependant cette approche présente un défaut majeur : lorsque le SAT solveur trouve une voie mais que cette voie n'est pas une voie respectant les contraintes de stœchiométrie le solveur doit essayer de compléter cette voie avec toutes les réactions qui ne sont pas actuellement dans la voie. Cela introduit une explosion combinatoire qui n'est pas contrainte, rendant cette approche non compétitive vis-à-vis des méthodes classiques (fondées sur la méthode de double description). Cette approche nous aura permis de prouver sa faisabilité et de nous servir de base pour la réalisation d'un outil se fondant sur un SMT.

L'utilisation d'un SMT nous a permis de résoudre les problèmes liés au manque de communication entre la partie de recherche d'une solution et de la vérification de la stœchiométrie, puisque le SMT est capable de générer directement des solutions respectant les contraintes de stœchiométrie, et ainsi de proposer un calcul d'EFMs compétitif lorsque le réseau est assez contraint ou que le réseau est trop gros pour être traité avec les méthodes actuelles. D'autre part l'approche SMT nous a également permis de prendre en compte une grande diversité de requêtes (sélection de gènes via des requêtes booléennes, limitations de taille...).

Lors de nos travaux sur les EFMs nous avons remarqué que la prise en compte des requêtes par les outils basé sur la méthode de double description ne permettait que de lister les EFMs satisfaisant les requêtes alors que l'ensembles des solutions, lorsqu'on ajoute une requête, ne peut

plus s'exprimer simplement par la combinaison de cesEFMs. Ainsi lorsqu'on cherche l'ensemble des voies minimales respectant une contrainte on ne peut plus se baser sur les EFMs. En effet, dans certains cas, il existe des voies respectant la contrainte bien qu'il n'existe pas d'EFMs la respectant. Pour pallier ce problème nous avons introduit la notion de $MCFM$ représentant les voies minimales respectant les contraintes. Les $MCFM$s permettent de trouver des voies respectant les contraintes même si aucun EFM, pris individuellement, ne les respecte. De plus nous avons proposé une implémentation du calcul des $MCFM$s se basant sur un SAT solveur et nous avons porté notre attention sur différentes politiques de minimisation et leurs impacts sur les performances en fonction du type de requêtes.

5.2 Perspectives

Bien que notre outil soit parfaitement fonctionnel pour le calcul des EFMs et des $MCFM$s et pour la prise en compte des requêtes booléennes, il reste améliorable, en effet il nous faudrait maintenant tester et optimiser la prise en compte des contraintes thermodynamiques (des résultats préliminaires ont validé la preuve de concept sur quelques exemples) et implémenter des requêtes spécifiques telles que vérifier la robustesse d'un réseau ou maximiser le rapport de production entre deux métabolites.

De plus la notion d'impliquant premier dans une formule propositionnelle modélisant des voies métaboliques et la notion de support d'EFM dans un réseau sans problématique de stœchiométrie sont confondues. Dans cette thèse nous n'avons pas exploré le calcul des impliquants premiers dans un SMT qui pourrait être une méthode permettant de calculer directement les EFMs sans avoir à passer par une phase de minimisation. Il serait donc intéressant d'implémenter un calcul d'impliquants premiers et de comparer ses performances avec nos méthodes de minimisation.

Notre approche, à base de SMT, peut également être utilisée dans le calcul des « cut-set ». Les cut sets peuvent se calculer par deux méthodes. Soit comme «hitting sets» minimaux de l'ensemble concerné des EFMs soit comme les EFMs d'un réseau dual. Dans le premier

cas, notre approche est directement applicable par calcul des EFMs du réseau primal vérifiant les contraintes biologiques supplémentaires. Dans le second cas, il faut étudier l'intégration directe de ces contraintes dans le réseau dual et appliquer à nouveau ensuite notre approche. La validation et la comparaison des deux approches seraient bénéfiques. On peut prédire des résultats compétitifs lorsque le réseau est très contraint ou trop gros pour être exploité par les méthodes basées sur la double description.

Une autre extension de notre travail peut être de prendre en compte des contraintes plus générales telles que les contraintes cinétiques (qui incluent les contraintes thermodynamiques). La prise en compte des contraintes cinétiques est un problème non convexe ce qui le rend extrêmement dur à résoudre. Cependant un résultat indique que lorsqu'on cherche à maximiser un flux, ou une combinaison à coefficients fixés de flux, le maximum est atteint en un EFM. Un second résultat (non encore publié) indique que l'optimisation pour un EFM donné devient un problème convexe, qu'on peut donc résoudre par les solveurs existants de programmation convexe. On peut donc lister l'ensemble des EFMs puis calculer pour chacun d'eux la valeur de l'optimum, l'optimum global est le plus grand d'entre eux. Notre outil étendu à la prise en compte de la cinétique serait donc adapté au calcul préliminaire des EFMs compatibles avec ces contraintes cinétiques. Enfin, il faudrait intégrer la prise en compte de ces contraintes cinétiques et de celles de régulation génique que nous avons traitées, en revisitant de la même façon le concept d'EFM.

Appendix A

Tableaux de répartition du temps de calcul pour l'approche SAT

Les trois tableaux suivants présentent la répartition du temps de calcul pour l'approche SAT pour les requêtes positives, négatives et disjonctives. Les résultats présentés correspondent à des exécutions utilisant une étape sans stœchiométrie puis listant les résultats par taille croissante où timeout est fixé à une heure.

Le chiffre à la fin du nom d'une requête représente le nombre d'EFMs dans le réseau pour cette requête.

Nom	analyse	minimisation	recherche
req_neg_1	45.3	709	2.84e+03
req_neg_1168	14.3	87.2	451
req_neg_1331	11.3	67.6	331
req_neg_1503	35	643	2.92e+03
req_neg_1663	44	650	2.91e+03
req_neg_1849	44.5	650	2.9e+03
req_neg_1925	32.6	670	2.9e+03
req_neg_2130	32.9	598	2.97e+03
req_neg_236	36.9	629	2.93e+03
req_neg_2501	38.3	593	2.97e+03
req_neg_27	54.3	656	2.89e+03
req_neg_2751	49.7	694	2.86e+03
req_neg_2993	45.1	547	2.37e+03
req_neg_327	45.7	588	2.96e+03
req_neg_3345	44.9	675	2.88e+03
req_neg_3633	39.2	712	2.85e+03
req_neg_3944	42.7	658	2.9e+03
req_neg_3971	35.6	695	2.87e+03
req_neg_423	35.3	695	2.87e+03
req_neg_4371	31.5	608	2.96e+03
req_neg_4471	37.6	699	2.86e+03
req_neg_4653	43.7	662	2.89e+03
req_neg_4681	38.4	665	2.9e+03
req_neg_4866	35.5	688	2.88e+03
req_neg_4940	37.1	709	2.85e+03
req_neg_583	38.7	609	2.95e+03
req_neg_663	46.8	617	2.94e+03
req_neg_746	38.7	680	2.88e+03
req_neg_890	41.7	780	2.78e+03
req_neg_91	41.9	648	2.91e+03
moyenne	38.64	619.39	2712.4

TABLE A.1: Expérience avec des requêtes négatives unitaires.

Nom	analyse	minimisation	recherche
req_disj_1022	36.1	714	2.85e+03
req_disj_1232	40.5	767	2.79e+03
req_disj_1653	31.6	356	1.51e+03
req_disj_1767	53.2	721	2.82e+03
req_disj_1935	42.2	730	2.83e+03
req_disj_1996	52.5	697	2.61e+03
req_disj_2110	39.5	779	2.78e+03
req_disj_2257	34.7	685	2.88e+03
req_disj_2561	43.2	748	2.81e+03
req_disj_2634	31.1	595	2.97e+03
req_disj_2718	48.8	713	2.84e+03
req_disj_2963	41.3	691	2.87e+03
req_disj_3603	43.1	656	2.9e+03
req_disj_3679	38.9	775	2.79e+03
req_disj_381	34	633	2.93e+03
req_disj_3844	39.7	701	2.86e+03
req_disj_3978	39	720	2.84e+03
req_disj_4079	38.5	682	2.88e+03
req_disj_4198	35.7	664	2.9e+03
req_disj_4268	47.8	700	2.85e+03
req_disj_4431	34.2	705	2.86e+03
req_disj_4677	39.2	676	2.88e+03
req_disj_4762	37.2	685	2.88e+03
req_disj_4826	36.1	729	2.83e+03
req_disj_4989	36.2	718	2.84e+03
req_disj_584	36	675	2.89e+03
req_disj_705	29.3	360	1.43e+03
req_disj_775	39.9	636	2.92e+03
req_disj_906	38.5	732	2.83e+03
req_disj_978	35.3	385	1.51e+03
moyenne	39.11	667.6	2712.66

TABLE A.2: Expérience avec des requêtes négatives de taille deux.

Nom	analyse	minimisation	recherche
req_pos_1	0	2e-05	0.00231
req_pos_1046	41.7	590	2.97e+03
req_pos_108	19.9	157	811
req_pos_1176	45.6	478	3.08e+03
req_pos_1316	33.7	522	3.04e+03
req_pos_1472	45.7	552	3e+03
req_pos_1582	42.4	538	3.02e+03
req_pos_1824	45.8	451	3.1e+03
req_pos_1960	41.5	562	3e+03
req_pos_224	54.3	600	2.94e+03
req_pos_2310	44.2	594	2.96e+03
req_pos_2544	42.9	565	2.99e+03
req_pos_2937	32	287	1.42e+03
req_pos_296	33.6	346	1.92e+03
req_pos_3074	35	562	3e+03
req_pos_3178	28.8	533	3.04e+03
req_pos_3213	45.1	576	2.98e+03
req_pos_3296	41.4	598	2.96e+03
req_pos_3428	32.1	554	3.01e+03
req_pos_3480	43.6	568	2.99e+03
req_pos_3696	50.2	395	3.15e+03
req_pos_3768	30.5	527	3.04e+03
req_pos_3893	32.8	569	3e+03
req_pos_3904	52.2	526	3.02e+03
req_pos_420	29.1	489	3.08e+03
req_pos_492	44.1	564	2.99e+03
req_pos_4998	40.1	561	3e+03
req_pos_54	55.9	566	2.98e+03
req_pos_600	53.1	534	3.01e+03
req_pos_760	35.7	546	3.02e+03
moyenne	39.1	497	2750.7

TABLE A.3: Expérience avec des requêtes positives unitaires.

Appendix B

Tableaux récapitulatifs pour le nombre de solutions trouvées pour l'approche SAT

Le chiffre à la fin du nom d'une requête représente le nombre d'EFMs dans le réseau pour cette requête, la colonne « 1h timeout » indique le nombre de EFMs trouvés en une heure, la colonne « % » indique le pourcentage du nombre total d'EFM que cela représente et la colonne « fini » indique si l'algorithme a terminé. Ainsi dans la première ligne, bien que 100% des solutions aient été trouvées l'algorithme cherche encore d'autres solutions.

Nom	1h timeout	%	nb call SAT	Call par EFM	fini
req_neg_1	1	100	424351	4.24e+05	**non**
req_neg_1168	1168	100	184201	158	oui
req_neg_1331	1331	100	136509	103	oui
req_neg_1503	955	63.5	367433	385	**non**
req_neg_1663	762	45.8	409259	537	**non**
req_neg_1849	570	30.8	394870	693	**non**
req_neg_1925	134	6.96	344457	2.57e+03	**non**
req_neg_2130	118	5.54	355173	3.01e+03	**non**
req_neg_236	115	48.7	385419	3.35e+03	**non**
req_neg_2501	145	5.8	388481	2.68e+03	**non**
req_neg_27	27	100	516383	1.91e+04	**non**
req_neg_2751	1307	47.5	448997	344	**non**
req_neg_2993	2993	100	453420	151	oui
req_neg_327	327	100	431728	1.32e+03	**non**
req_neg_3345	2337	69.9	446425	191	**non**
req_neg_3633	2191	60.3	388152	177	**non**
req_neg_3944	1433	36.3	416676	291	**non**
req_neg_3971	680	17.1	352011	518	**non**
req_neg_423	158	37.4	360290	2.28e+03	**non**
req_neg_4371	157	3.59	343793	2.19e+03	**non**
req_neg_4471	2053	45.9	359259	175	**non**
req_neg_4653	1550	33.3	405184	261	**non**
req_neg_4681	1442	30.8	411359	285	**non**
req_neg_4866	2266	46.6	385437	170	**non**
req_neg_4940	2626	53.2	372017	142	**non**
req_neg_583	583	100	375420	644	**non**
req_neg_663	302	45.6	406147	1.34e+03	**non**
req_neg_746	737	98.8	379413	515	**non**
req_neg_890	647	72.7	371381	574	**non**
req_neg_91	91	100	363415	3.99e+03	**non**

TABLE B.1: Informations pour les requêtes négatives unitaires

Nom	1h timeout	%	nb call SAT	Call par EFM	fini
req_disj_1022	734	71.8	379572	517	**non**
req_disj_1232	504	40.9	377670	749	**non**
req_disj_1653	1652	99.9	322276	195	oui
req_disj_1767	1264	71.5	483255	382	**non**
req_disj_1935	628	32.5	394151	628	**non**
req_disj_1996	1995	99.9	461627	231	oui
req_disj_2110	1350	64	394779	292	**non**
req_disj_2257	333	14.8	353825	1.06e+03	**non**
req_disj_2561	1433	56	414127	289	**non**
req_disj_2634	136	5.16	334626	2.46e+03	**non**
req_disj_2718	1695	62.4	450914	266	**non**
req_disj_2963	1630	55	402539	247	**non**
req_disj_3603	698	19.4	428481	614	**non**
req_disj_3679	2498	67.9	365484	146	**non**
req_disj_381	26	6.82	352091	1.35e+04	**non**
req_disj_3844	899	23.4	384639	428	**non**
req_disj_3978	1620	40.7	371083	229	**non**
req_disj_4079	1929	47.3	393749	204	**non**
req_disj_4198	941	22.4	356531	379	**non**
req_disj_4268	1518	35.6	452071	298	**non**
req_disj_4431	1258	28.4	347171	276	**non**
req_disj_4677	1190	25.4	401190	337	**non**
req_disj_4762	574	12.1	381981	665	**non**
req_disj_4826	944	19.6	356421	378	**non**
req_disj_4989	1700	34.1	374892	221	**non**
req_disj_584	353	60.4	388175	1.1e+03	**non**
req_disj_705	704	99.9	344877	490	oui
req_disj_775	672	86.7	386102	575	**non**
req_disj_906	800	88.3	403022	504	**non**
req_disj_978	977	99.9	350650	359	oui

TABLE B.2: Informations pour les requêtes disjonctives

Nom	1h timeout	%	nb call SAT	Call par EFM	fini
req_pos_1	1	100	1	1	oui
req_pos_1046	589	56.3	453671	770	**non**
req_pos_108	108	100	275974	2.56e+03	oui
req_pos_1176	90	7.65	406549	4.52e+03	**non**
req_pos_1316	129	9.8	380653	2.95e+03	**non**
req_pos_1472	39	2.65	469777	1.2e+04	**non**
req_pos_1582	133	8.41	480180	3.61e+03	**non**
req_pos_1824	152	8.33	454387	2.99e+03	**non**
req_pos_1960	360	18.4	517365	1.44e+03	**non**
req_pos_224	193	86.2	603987	3.13e+03	**non**
req_pos_2310	1579	68.4	541199	343	**non**
req_pos_2544	113	4.44	452862	4.01e+03	**non**
req_pos_2937	2947	100	421471	143	oui
req_pos_296	296	100	422253	1.43e+03	oui
req_pos_3074	192	6.25	401795	2.09e+03	**non**
req_pos_3178	136	4.28	347005	2.55e+03	**non**
req_pos_3213	552	17.2	471310	854	**non**
req_pos_3296	775	23.5	435932	562	**non**
req_pos_3428	165	4.81	369846	2.24e+03	**non**
req_pos_3480	2502	71.9	492394	197	**non**
req_pos_3696	0	0	410922	–	**non**
req_pos_3768	796	21.1	372816	468	**non**
req_pos_3893	250	6.42	358787	1.44e+03	**non**
req_pos_3904	106	2.72	539850	5.09e+03	**non**
req_pos_420	4	0.952	323550	8.09e+04	**non**
req_pos_492	64	13	412871	6.45e+03	**non**
req_pos_4998	179	3.58	432897	2.42e+03	**non**
req_pos_54	5	9.26	489821	9.8e+04	**non**
req_pos_600	171	28.5	496346	2.9e+03	**non**
req_pos_760	23	3.03	391692	1.7e+04	**non**

TABLE B.3: Informations pour les requêtes positives unitaires

Bibliographie

Beezer, R. A. (2008). *A first course in linear algebra*. Beezer.

Chatalic, P. and Simon, L. (1999). Davis et Putnam, 40 ans après : une première expérimentation. *Actes des Sixièmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets*, pages 9–16.

Cornish-Bowden, A. (2012). *Cinétique enzymatique*. EDP sciences.

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

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3) :201–215.

Dechter, R. and Rish, I. (1994). Directional resolution : The davis-putnam procedure, revisited. *KR*, 94 :134–145.

Eén, N. and Sörensson, N. (2003). An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer.

Gagneur, J. and Klamt, S. (2004). Computation of elementary modes : a unifying framework and the new binary approach. *BMC bioinformatics*, 5(1) :1.

Gerstl, M. P., Jungreuthmayer, C., and Zanghellini, J. (2015a). tefma : computing thermodynamically feasible elementary flux modes in metabolic networks. *Bioinformatics*, page btv111.

Gerstl, M. P., Ruckerbauer, D. E., Mattanovich, D., Jungreuthmayer, C., and Zanghellini, J. (2015b). Metabolomics integrated elementary flux mode analysis in large metabolic networks. *Scientific reports*, 5 :8930.

Gomes, C. P., Selman, B., Crato, N., and Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2) :67–100.

Haim, S. and Heule, M. (2014). Towards ultra rapid restarts. *arXiv preprint arXiv :1402.4413*.

Hamadi, Y., Jabbour, S., and Sais, L. (2008). Manysat : a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :245–262.

Jeroslow, R. G. and Wang, J. (1990). Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4) :167–187.

Jungreuthmayer, C., Beurton-Aimar, M., and Zanghellini, J. (2013a). Fast computation of minimal cut sets in metabolic networks with a berge algorithm that utilizes binary bit pattern trees. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 10(5) :1–1.

Jungreuthmayer, C., Ruckerbauer, D. E., and Zanghellini, J. (2012). Utilizing gene regulatory information to speed up the calculation of elementary flux modes. *arXiv preprint arXiv :1208.1853*.

Jungreuthmayer, C., Ruckerbauer, D. E., and Zanghellini, J. (2013b). regefmtool : Speeding up elementary flux mode calculation using transcriptional regulatory rules in the form of three-state logic. *Biosystems*, 113(1) :37–39.

Klamt, S., Gagneur, J., and von Kamp, A. (2005). Algorithmic approaches for computing elementary modes in large biochemical reaction networks. *IEE Proceedings-Systems Biology*, 152(4) :249–255.

Klamt, S. and Gilles, E. D. (2004). Minimal cut sets in biochemical reaction networks. *Bioinformatics*, 20(2) :226–234.

Klamt, S., Saez-Rodriguez, J., and Gilles, E. D. (2007). Structural and functional analysis of cellular networks with cellnetanalyzer. *BMC systems biology*, 1(1) :1.

Koshimura, M., Nabeshima, H., Fujita, H., and Hasegawa, R. (2009). Minimal model generation with respect to an atom set. *First-order Theorem Proving FTP 2009*, page 49.

Lewis, N. E., Nagarajan, H., and Palsson, B. O. (2012). Constraining the metabolic genotype–phenotype relationship using a phylogeny of in silico methods. *Nature Reviews Microbiology*, 10(4) :291–305.

Liang, J. H., Ganesh, V., Zulkoski, E., Zaman, A., and Czarnecki, K. (2015). Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Haifa Verification Conference*, pages 225–241. Springer.

Long, G., Jabbour, S., and LAKHDAR, S. (2013). Stratégies d'élimination des clauses apprises dans les solveurs sat modernes. *9ièmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 147–156.

Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. In *Proceedings of the 2nd Israel Symposium on the Theory and Computing Systems*, pages 128–133. IEEE.

Morterol, M., Dague, P., Peres, S., and Simon, L. (2016). Minimality of metabolic flux modes under boolean regulation constraints. 12th International Workshop on Constraint-Based Methods for Bioinformatics (WCB'16).

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

Motzkin, T. S., Raiffa, H., Thompson, G. L., and Thrall, R. M. (1953). The double description method. *Contributions to the Theory of Games II*, 8 :51–73.

Nieuwenhuis, R., Oliveras, A., and Tinelli, C. (2006). Solving sat and sat modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6) :937–977.

Orth, J. D., Thiele, I., and Palsson, B. Ø. (2010). What is flux balance analysis ? *Nature biotechnology*, 28(3) :245–248.

Peres, S. (2005). *Analyse de la structure des réseaux métaboliques : application au métabolisme énergétique mitochondrial*. PhD thesis, Université de Bordeaux 2, Bordeaux.

Peres, S., Morterol, M., and Simon, L. (2014). Sat-based metabolics pathways analysis without

compilation. In et al. (Eds.) : CMSB, P. M., editor, *Lecture Note in Bioinformatics*, volume 8859, pages 20–31. Springer International Publishing.

Peres, S., Vallée, F., Beurton-Aimar, M., and Mazat, J. (2011). Acom : a classification method for elementary flux modes based on motif finding. *Biosystems*, 103(3) :410–419.

Pipatsrisawat, K. and Darwiche, A. (2007). A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 294–299. Springer.

Planes, F. and Beasley, J. (2009). Path finding approaches and metabolic pathways. *Discrete Applied Mathematics*, 157(10) :2244 – 2256. Networks in Computational Biology.

Schuster, S., Dandekar, T., and Fell, D. (2000). Description of the algorithm for computing elementary flux modes. *URL : http ://bms-mudshark. brookes. ac. uk/algorithm. pdf.*

Schuster, S. and Hilgetag, C. (1994). On elementary flux modes in biochemical reaction systems at steady state. *Journal of Biological Systems*, 2(02) :165–182.

Silva, J. P. M. and Sakallah, K. A. (1997). Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society.

Simon, L. (2014). Oublier pour mieux régner : une courte étude expérimentale. *Dixièmes Journées Francophones de Programmation par Contraintes (JFPC 2014).*

Smith, A. C. and Robinson, A. J. (2011). A metabolic model of the mitochondrion and its use in modelling diseases of the tricarboxylic acid cycle. *BMC Systems Biology*, 5(1) :1–13.

Soh, T., Inoue, K., Baba, T., Takada, T., and Shiroishi, T. (2012). Predicting gene knockout effects by minimal pathway enumeration. In *The 4th International Conference on Bioinformatics, Biocomputational Systems and Biotechnologies (BIOTECHNO 2012)*, pages 11–19. Citeseer.

Stelling, J., Klamt, S., Bettenbrock, K., Schuster, S., and Gilles, E. D. (2002). Metabolic network structure determines key aspects of functionality and regulation. *Nature*, 420(6912) :190–193.

Terzer, M. (2009). *Large scale methods to enumerate extreme rays and elementary modes*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 18538, 2009.

Terzer, M. and Stelling, J. (2008). Large-scale computation of elementary flux modes with bit pattern trees. *Bioinformatics*, 24(19) :2229–2235.

Terzer, M. and Stelling, J. (2009). Parallel extreme ray and pathway computation. In *International Conference on Parallel Processing and Applied Mathematics*, pages 300–309. Springer.

Thiele, I., Swainston, N., Fleming, R. M., Hoppe, A., Sahoo, S., Aurich, M. K., Haraldsdottir, H., Mo, M. L., Rolfsson, O., Stobbe, M. D., et al. (2013). A community-driven global reconstruction of human metabolism. *Nature biotechnology*, 31(5) :419–425.

Tseitin, G. S. (1983). On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer.

Von Kamp, A. and Schuster, S. (2006). Metatool 5.0 : fast and flexible elementary modes analysis. *Bioinformatics*, 22(15) :1930–1931.

Wagner, C. (2004). Nullspace approach to determine the elementary modes of chemical reaction systems. *The Journal of Physical Chemistry B*, 108(7) :2425–2431.

Yeung, M., Thiele, I., and Palsson, B. Ø. (2007). Estimation of the number of extreme pathways for metabolic networks. *BMC bioinformatics*, 8(1) :1.

Zhang, H. and Stickel, M. (2000). Implementing the davis–putnam method. *Journal of Automated Reasoning*, 24(1-2) :277–296.