Inférence et certificats pour le problème de satisfiabilité maximum

Matthieu Py

NNT/NL : 2021AIXM0631/029ED184

THÈSE DE DOCTORAT

Soutenance effectuée le jeudi 9 décembre 2021

Matthieu PY

Inférence et certificats pour le problème de satisfiabilité maximum

Discipline
Informatique

École doctorale
ED 184 Mathématiques et Informatique

Laboratoire/Partenaires de recherche
Laboratoire d'Informatique & Systèmes

Composition du jury

Gilles Audemard Professeur des universités CRIL, Université d'Artois	Rapporteur
Jin-Kao Hao Professeur des universités LERIA, Université d'Angers	Rapporteur
Christine Solnon Professeure des universités CITI lab, INSA de Lyon	Examinatrice
Felip Manyà Senior Researcher Artificial Intelligence Research Institute, Spain	Examinateur
Djamal Habet Professeur des universités LIS, Université d'Aix-Marseille	Directeur de thèse

Je soussigné, Matthieu Py, déclare par la présente que le travail présenté dans ce manuscrit est mon propre travail, réalisé sous la direction scientifique de Djamal Habet, dans le respect des principes d'honnêteté, d'intégrité et de responsabilité inhérents à la mission de recherche. Les travaux de recherche et la rédaction de ce manuscrit ont été réalisés dans le respect à la fois de la charte nationale de déontologie des métiers de la recherche et de la charte d'Aix-Marseille Université relative à la lutte contre le plagiat. Ce travail n'a pas été précédemment soumis en France ou à l'étranger dans une version identique ou similaire à un organisme examinateur.

Fait à Marseille le 1er octobre 2021.

Liste des publications

Publications dans des conférences internationales

1. **Towards Bridging the Gap Between SAT and Max-SAT Refutations**
   Matthieu Py, Mohamed Sami Cherif and Djamal Habet
   32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pages 137-144, 2020

2. **A Proof Builder for Max-SAT**
   Matthieu Py, Mohamed Sami Cherif and Djamal Habet
   24th International Conference on Theory and Applications of Satisfiability Testing (SAT), pages 488–498, 2021

3. **Computing Max-SAT Refutations using SAT Oracles**
   Matthieu Py, Mohamed Sami Cherif and Djamal Habet
   33rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 8 pages (two columns per page), pending publishing, 2021

4. **Inferring Clauses and Formulas in Max-SAT**
   Matthieu Py, Mohamed Sami Cherif and Djamal Habet
   33rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 8 pages (two columns per page), pending publishing, 2021

Publications dans des conférences nationales

1. **Des réfutations SAT aux réfutations Max-SAT**
   Matthieu Py, Mohamed Sami Cherif et Djamal Habet
   Journées Francophones de Programmation par Contraintes (JFPC), 2 pages, 2021

Résumé

Dans cette thèse, on s'intéresse au problème de satisfiabilité maximum, ou problème Max-SAT, qui consiste, étant donnée une formule propositionnelle sous forme normale conjonctive, à trouver une interprétation des variables de la formule permettant de satisfaire le plus de clauses possible, ou, de manière équivalente, de falsifier le moins de clauses possible. Le système de preuve le plus utilisé dans Max-SAT est basé sur la règle d'inférence par max-résolution qui est l'adaptation pour Max-SAT de la règle de résolution utilisée pour le problème SAT. La règle de résolution déduit une nouvelle clause à partir de deux clauses qui s'opposent, ce qui permet notamment de certifier qu'une formule est insatisfiable en déduisant progressivement de nouvelles clauses jusqu'à en déduire une contradiction, représentée sous la forme d'une clause vide (on parle alors de *réfutation par résolution*). L'adaptation des réfutations par résolution pour Max-SAT sans en augmenter considérablement la taille est un problème ouvert depuis l'introduction de la max-résolution.

On propose dans cette thèse deux méthodes pour adapter n'importe quelle réfutation par résolution en réfutation valide pour Max-SAT, à laquelle on se réfère sous le terme *max-réfutation*. Une autre contribution de cette thèse est la construction de certificats qui permettent de démontrer l'optimalité d'une solution pour le problème Max-SAT. Pour générer de tels certificats, on utilise les max-réfutations que l'on est désormais capables de générer à partir des réfutations par résolution. Comme une max-réfutation permet de certifier qu'au moins une clause d'une formule doit être falsifiée, il est alors possible de certifier, en utilisant k max-réfutations, qu'au moins k clauses de cette même formule doivent être falsifiées. Cette séquence de max-réfutations, à laquelle on ajoute une interprétation des variables permettant de falsifier exactement k clauses de la formule, forme un certificat pour le problème Max-SAT. Enfin, on s'intéresse au problème qui consiste, étant donnée une formule initiale et une information donnée (clause ou formule), à inférer cette information par des règles d'inférence qui préservent l'équivalence Max-SAT. Comme la max-résolution est incomplète pour l'inférence dans Max-SAT, on propose un nouveau système de preuve ainsi qu'un algorithme permettant de construire n'importe quelle inférence de clauses ou de formules, ou de certifier qu'une telle inférence ne peut exister.

Mots clés : Intelligence Artificielle, Programmation par Contraintes, Problème Max-SAT, Problème SAT, Inférence.

Abstract

In this thesis, we are interested in the maximum satisfiability problem, or Max-SAT problem, which consists, given a Boolean formula in conjunctive normal form, in finding an assignment of the variables of the formula which allows to satisfy as many clauses as possible, or, equivalently, to falsify as few clauses as possible. The most widely used Max-SAT proof system is based on the Max-SAT resolution inference rule which is the adaptation for Max-SAT of the resolution rule used for the SAT problem. The resolution rule deduces a new clause from two opposing clauses, enabling to certify that a formula is unsatisfiable by gradually deducing new clauses until deducing a contradiction, represented in the form of an empty clause. The adaptation of such proofs, referred to as *resolution refutations*, for Max-SAT without considerably increasing its size is an open problem since the introduction of Max-SAT resolution.

We propose in this thesis two methods to adapt any resolution resfutation into a valid refutation for Max-SAT, referred to as *max-refutation*. Another contribution is the construction of certificates to demonstrate the optimality of a solution for the Max-SAT problem. To generate such certificates, we use the max-refutations that we are now able to generate from the resolution refutations. Indeed, as a max-refutation certifies that we must falsify at least one clause of a formula, it is then possible to certify, using k max-refutations that we must falsify at least k clauses of the same formula. This sequence of max-refutations, coupled with an interpretation of the variables allowing to falsify exactly k clauses of the formula, is a certificate for the Max-SAT problem. Finally, we are interested in the problem which consists, given an initial formula and a given information (clause or formula), of inferring this information by Max-SAT-equivalence-preserving inference rules. As the max-resolution is incomplete for the inference in Max-SAT, we propose a new proof system as well as an algorithm allowing to infer, if possible, a given clause or formula.

Keywords: Artificial Intelligence, Constraint Programming, Max-SAT Problem, SAT Problem, Inference.

Remerciements

Mes premiers remerciements vont à mon directeur de thèse, Djamal Habet, pour m'avoir accompagné pendant ces trois années de thèse. Je tiens également à remercier toutes les personnes ayant contribué à l'avancée de mes travaux de thèse ou de ce manuscrit, que ce soit Gilles Audemard, Felip Manyà, Christine Solnon et Jin-Kao Hao qui sont membres de mon jury de thèse, Philippe Jegou et Chu Min Li qui forment mon comité de suivi de thèse, mes collègues de l'équipe COALA avec qui j'ai travaillé depuis trois ans ainsi que l'ensemble des membres de l'Université d'Aix-Marseille avec lesquels j'ai pu être en contact.

J'adresse également mes remerciements aux personnes qui m'ont permis de me lancer dans cette thèse. Tout d'abord, le financement de cette thèse n'aurait pu être obtenu sans l'aide de l'École Doctorale en Mathématiques et Informatique de Marseille dirigée par Nadia Creignou ainsi que celle du président de l'Université d'Aix-Marseille Yvon Berland. Je fais également part de mes remerciements aux personnes qui m'ont suivies dans mon parcours qui a précédé la thèse, en particulier les membres de l'Établissement Infra Circulation des Pays de la Loire de Nantes, les enseignants-chercheurs du parcours Recherche Opérationnelle et Aide à la Décision de l'Université de Bordeaux ainsi que les enseignants-chercheurs de l'Université d'Aix-Marseille.

Enfin, je remercie ma famille, mes amis et les gens que j'ai rencontrés tout au long de ma vie, notamment ma compagne qui partage ma vie depuis ces neuf dernières années.

Cette thèse est dédiée à mon grand-père Jean et à mon oncle Gérald,
décédés du coronavirus pendant la réalisation de cette thèse.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Introduction

Introduction

Pour résoudre les problèmes de plus en plus complexes auxquels fait face notre société, plusieurs approches existent. L'une de ces approches est simplement de résoudre le problème *à la main*. Par exemple, si une entreprise doit créer le planning d'une équipe de dix personnes, une méthode courante sera de demander à l'un de ses collaborateurs d'essayer de faire des plannings *à la main*, jusqu'à arriver à trouver un planning jugé satisfaisant, qui respecte les contraintes réglementaires de l'entreprise et essaie d'atteindre certains critères de qualité. Supposons maintenant que l'équipe, plutôt que de contenir dix membres, contienne cent ou même mille personnes. Comment l'entreprise peut-elle réaliser les plannings de ses collaborateurs ? Là encore, la solution de faire ces plannings *à la main* est envisageable. Il suffira simplement d'allouer à cette tâche plusieurs collaborateurs sur une période de temps plus importante et d'accepter une imperfection des solutions proposées par rapport au planning idéal qui aurait pu être trouvé. Grâce à l'essor de l'informatique, les entreprises font de plus en plus appel à une autre solution qui consiste à résoudre par des méthodes informatiques ledit problème : c'est l'essor de l'algorithmique. En utilisant l'algorithmique, on peut, soit développer une méthode spécialisée dans la résolution d'un seul problème, celui rencontré par l'entreprise, soit modéliser le problème dans un formalisme plus global dans lequel celui-ci, vidé de toute sémantique superflue, est représenté sous sa forme la plus épurée possible. Ainsi, plutôt que de chercher une méthode de résolution pour chaque problème du monde réel, on cherche plutôt des méthodes de résolution pour des problèmes formels qui permettent de modéliser le plus de situations réelles possible, et, pour résoudre les problèmes du monde réel, on le modélise comme étant une instance d'un problème connu de la littérature.

Parmi les problèmes de la littérature les plus étudiés, on s'intéresse dans cette thèse au problème de satisfiabilité propositionnelle et plus particulièrement à sa version d'optimisation : le problème de satisfiabilité maximum. Le problème de satisfiabilité propositionnelle, ou problème SAT, consiste, étant donné un ensemble de choix devant respecter un certain nombre de contraintes, à déterminer quels choix effectuer de manière à respecter toutes les contraintes. Plus formellement, il consiste, étant donnée une formule propositionnelle sous forme normale conjonctive, à trouver une interprétation des variables de la formule permettant de satisfaire toutes les clauses de la formule, si une telle interprétation existe. Le problème de satisfiabilité maximum, ou problème Max-SAT, est l'extension du problème SAT en tant que problème d'optimisation. Il s'agit ici, non pas de chercher à satisfaire toutes les clauses de la formule, mais de chercher à en satisfaire le plus possible. Le problème Max-SAT permet par conséquent de modéliser plus de problèmes réels que le problème SAT, et c'est d'au-

tant plus vrai grâce aux versions plus élaborées du problème Max-SAT, permettant notamment de différencier l'importance des clauses les unes par rapport aux autres ou de poser certaines clauses comme devant être forcément satisfaites par toute solution proposée.

Au cours du début du vingt-et-unième siècle, il y a eu des progrès remarquables dans la résolution efficace du problème Max-SAT, que ce soit grâce aux algorithmes de type séparation et évaluation ou aux algorithmes utilisant des appels itératifs à des méthodes permettant de résoudre le problème de satisfiabilité ou le problème de programmation linéaire en nombres entiers. Cependant, si les programmes qui implémentent les algorithmes Max-SAT (on parle de *solveurs* Max-SAT) sont de plus en plus efficaces pour résoudre le problème Max-SAT, aucun d'entre eux n'est capable de générer des certificats pour la solution proposée, certificats qui permettent de démontrer que cette solution est bien correcte.

Dans les travaux qui forment les contributions de cette thèse de doctorat, on s'intéresse à la construction de certificats pour le problème Max-SAT. Tout d'abord, afin de générer de tels certificats, on s'attaque à un problème théorique sur l'adaptation des réfutations par résolution, utilisées dans le cadre du problème SAT pour démontrer qu'une formule ne peut pas être satisfaite, afin de les rendre utilisables pour le problème Max-SAT. En effet, comme une réfutation par résolution permet de garantir qu'il est nécessaire de falsifier une clause de la formule, on souhaite pouvoir utiliser cette réfutation dans Max-SAT de manière à pouvoir utiliser, par exemple, 42 réfutations pour garantir qu'il est nécessaire de falsifier 42 clauses de la formule. Cette réfutation par résolution, adaptée pour le problème Max-SAT et nommée *max-réfutation*, formerait ainsi un maillon important des certificats pour le problème Max-SAT. Utilisant les max-réfutations ainsi calculées, on propose ensuite un outil pour générer des certificats pour le problème Max-SAT, outil qui a été utilisé avec succès sur de nombreuses instances de la littérature. Enfin, on s'intéresse à des certificats plus génériques que ceux qui concernent le problème Max-SAT. On se pose la question de savoir, étant donné une formule initiale sous forme normale conjonctive et une information (clause ou formule) à déduire, comment transformer la formule initiale pour en déduire l'information voulue en utilisant des règles d'inférence valides pour Max-SAT. Comment la max-résolution est incomplète pour l'inférence de clauses (et donc de formules) dans Max-SAT, on propose alors un nouveau système de preuve complet pour l'inférence dans Max-SAT et un algorithme pour construire de telles inférences.

Ce document est organisé en deux parties. La première partie permet d'introduire au lecteur les notions utiles pour comprendre les contributions de cette thèse. Elle est composée des chapitres 1, 2 et 3. Le chapitre 1 introduit les notions préliminaires à la compréhension des problèmes SAT et Max-SAT et en particulier la notion de problèmes de décision et d'optimisation, l'algorithmique, la théorie de la complexité et le vocabulaire des formules propositionnelles sous forme normale conjonctive. Les

chapitres 2 et 3 présentent les problèmes SAT et Max-SAT, dont en particulier leurs définitions, les principales méthodes de résolution et les certificats permettant de démontrer l'optimalité d'une solution. La deuxième partie, composée des chapitres 4, 5, 6 et 7, a pour but de présenter les contributions de cette thèse. Les chapitres 4 et 5 présentent deux méthodes pour adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation. Le chapitre 6 présente un outil pour générer des certificats pour le problème Max-SAT, outil implémenté et testé sur les instances de la littérature et mis à disposition de la communauté scientifique. Ensuite, on s'intéresse dans le chapitre 7 à comment transformer une formule initiale sous forme normale conjonctive afin d'en déduire une information (clause ou formule) en utilisant des règles d'inférence valides pour Max-SAT. Enfin, le manuscrit se termine par une conclusion et des discussions sur les perspectives de ces travaux.

Les contributions de cette thèse ont fait l'objet de quatre publications dans des conférences internationales ainsi que d'une publication dans une conférence nationale :

- Les travaux du chapitre 4 ont été publiés à la conférence ICTAI 2021 dans l'article *Computing Max-SAT Refutations using SAT Oracles* [PCH21b].
- Les travaux du chapitre 5 ont été publiés à la conférence ICTAI 2020 dans l'article *Towards Bridging the Gap Between SAT and Max-SAT Refutations* [PCH20]. Un résumé a également été publié à la conférence JFPC 2021 dans l'article *Des réfutations SAT aux réfutations Max-SAT* [PCH21c].
- Les travaux du chapitre 6 ont été publiés à la conférence SAT 2021 dans l'article *A Proof Builder for Max-SAT* [PCH21a].
- Les travaux du chapitre 7 ont été publiés à la conférence ICTAI 2021 dans l'article *Inferring Clauses and Formulas in Max-SAT* [PCH21d].

Première partie

État de l’art

1. Préambule

Sommaire

1.1. Introduction

Dans ce chapitre, on présente les principales notions utiles à la compréhension de cette thèse : les problèmes de décision et d'optimisation dans la section 1.2, les algorithmes dans la section 1.3, la théorie de la complexité dans la section 1.4 et enfin les formules propositionnelles dans la section 1.5.

1.2. Problèmes de décision et d'optimisation

Les problèmes étudiés en informatique théorique permettent de résoudre de nombreux problèmes réels de notre société, problèmes qui sont épurés de toute information superflue et modélisés dans ces formalismes, afin de les résoudre. Il est courant de répartir les problèmes étudiés en plusieurs catégories et on va ici s'intéresser à deux catégories : les problèmes de décision et les problèmes d'optimisation.

Tout d'abord, un problème de décision est un énoncé dont la réponse est soit "oui", soit "non". Dans la littérature, les problèmes de décision qui intéressent la communauté scientifique ne sont pas triviaux (on ne peut pas déduire la réponse "oui"

ou "non" de manière évidente) et comportent un aspect générique. Par exemple, on préférera étudier l'énoncé "*Étant donné un nombre entier n, est-ce que n est un nombre premier ?*" plutôt que l'énoncé "*Est-ce que le nombre* 745 *est un nombre premier ?*" [1].

Définition 1.1 (Problème de décision)**.** *Un problème de décision est constituée d'un ensemble d'instances I, chaque instance $i \in I$ appartenant soit aux instances positives $I_Y \subseteq I$, soit aux instances négatives $I_N \subseteq I$. Résoudre le problème décision consiste à déterminer, étant donnée une instance $i \in I$, si i est une instance positive ou négative.* [2]

Voici quelques exemples de problèmes de décision bien connus.

Exemple 1.1 (primalité d'un nombre)**.** *Soit $n \in \mathbb{N}$ un nombre entier, n est-il un nombre premier ?*

Exemple 1.2 (connexité d'un graphe)**.** *Soit $G = (V, E)$ un graphe, G est-il connexe ?* [3]

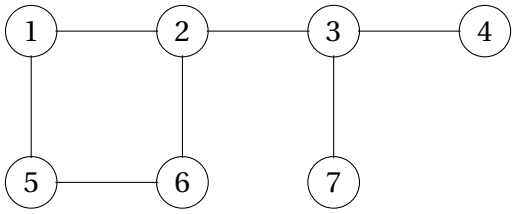


FIGURE 1.1. – Un graphe G connexe

Maintenance que l'on a présenté brièvement les problèmes de décisions, intéressons-nous aux problèmes d'optimisation. Un problème d'optimisation est une question dont la réponse est le plus souvent une valeur numérique. Dans un problème d'optimisation, il s'agit de chercher la meilleure réponse possible suivant un ensemble de critères bien définis.

Définition 1.2 (Problème d'optimisation)**.** *Un problème d'optimisation est un quadruplé (I, S, M, O) tel que :*

- *I est l'ensemble des instances du problème*
- *S est une fonction qui associe, pour toute instance du problème $i \in I$, un ensemble de solutions réalisables* [4] *de i*
- *M est une fonction qui associe, pour toute instance du problème $i \in I$ et toute solution réalisable $s \in S$ de l'instance, une valeur $v = M(i, s)$*
- *O est l'objectif min (minimiser) ou max (maximiser) du problème d'optimisation*

1. Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs : 1 et lui-même
2. définition extraite de [Wat17]
3. Un graphe, comme celui présenté à la figure 1.1, est connexe si l'on peut aller de n'importe quel sommet à n'importe quel autre en parcourant ses arêtes.
4. Sémantiquement, une solution réalisable est un ensemble de choix qui respectent toutes les contraintes du problème

Résoudre le problème d'optimisation (I, S, M, O) *consiste à déterminer, étant donnée une instance* $i \in I$*, la valeur* $M(i, s)$ *atteinte par toute solution optimale* $s \in S(i)$ *selon l'objectif* O*.*

Remarquons qu'il est fréquent, que ce soit pour les problèmes de décision ou d'optimisation, d'exiger davantage que la simple résolution du problème. Il est par exemple courant, lorsque l'on cherche à résoudre un problème d'optimisation, de ne pas simplement chercher la valeur optimale du problème mais également un moyen d'atteindre cette valeur. Par exemple, dans le problème du plus court chemin présenté plus bas, on essaie en général, en même temps que l'on donne la longueur du plus court chemin du graphe, de donner un exemple de chemin permettant d'atteindre cette longueur.

Exemple 1.3 (plus court chemin)**.** *Soit* $G = (V, E)$ *un graphe et* $v_1, v_2 \in V$ *deux sommets de ce graphe. Quelle est la longueur du plus court chemin entre* v_1 *et* v_2 *?*

Exemple 1.4 (sac à dos)**.** *Étant donné un sac à dos de capacité* W *et un ensemble de* n *objets, chaque objet numéro* i *ayant un profit* p_i *et un poids* w_i*, quel profit maximum est-il possible de réaliser en choisissant des objets avec un poids total inférieur à la capacité du sac* W *?*

Dans cette thèse, on va particulièrement s'intéresser au problème de décision qui est le problème de satisfiabilité propositionnelle, connu sous le nom de SAT, et surtout à son extension en tant que problème d'optimisation, le problème de satisfiabilité maximum, connu sous le nom de Max-SAT.

1.3. Algorithmes

Pour répondre aux problèmes de décision et d'optimisation comme ceux présentés précédemment, on essaie de plus en plus de formaliser des méthodes précises qui sont capables de fournir la bonne réponse au problème étudié quelque soit l'instance du problème en entrée. On appelle ces méthodes : des algorithmes.

Définition 1.3 (Algorithme)**.** *Un algorithme est une suite finie d'instructions non ambiguë permettant de résoudre un problème donné.*

On a vu précédemment quelques exemples de problèmes. Pour chaque problème présenté, il existe de nombreux algorithmes permettant de les résoudre. Par exemple, l'algorithme 1.1 permet de tester si un nombre n est premier ou non. Il consiste à tester, pour chaque nombre i entre 2 et $(n-1)$, si notre nombre n est un multiple de i.

Lorsque l'on s'intéresse à des algorithmes, on s'évertue souvent à démontrer leur correction et leur complexité [5]. Étudier la correction d'un algorithme, c'est dire si celui-ci retourne bien la *bonne* réponse. Par exemple, on est sûr que l'algorithme ci-dessus

5. On s'intéresse aussi souvent à leur terminaison (démontrer que l'exécution de l'algorithme termine forcément après un temps fini), mais cela est le plus souvent inclus dans l'analyse de leur complexité temporelle

Algorithme 1.1 Algorithme de primalité

Entrée : Nombre naturel n
Sortie : n est premier ou n n'est pas premier
1: **pour tout** $i \in \{2, ..., n-1\}$ **faire**
2: **si** n est un multiple de i **alors**
3: **retourner** n n'est pas premier
4: **retourner** n est premier

est correct car il teste tous les nombres qui pourraient être des diviseurs du nombre donné en entrée. La complexité, c'est la quantité de ressources (le plus souvent le temps, mais aussi l'espace mémoire) dont un algorithme a besoin pour fonctionner avant d'apporter la réponse au problème posé. Dans la sous-section suivante, on va détailler quelques éléments de complexité et notamment l'existence de différentes classes de complexité qui permettent de regrouper certains problèmes de difficulté proche d'un point de vue algorithmique.

1.4. Théorie de la complexité

Étant donné un problème et des algorithmes pour le résoudre, une question légitime que l'on se pose est de mesurer à quel point les algorithmes proposés sont efficaces pour résoudre le problème donné et à quel point le problème est difficile à résoudre. On parle alors de complexité des algorithmes et de complexité des problèmes. En général, ces complexités sont étudiées sous la forme de fonctions et permettent de quantifier les ressources dont on a besoin en fonction de la taille de l'instance du problème en entrée. Le plus souvent, on va surtout mesurer la complexité temporelle, c'est à dire le nombre d'unités de temps nécessaires pour appliquer tel ou tel algorithme ou pour résoudre tel ou tel problème, ou la complexité spatiale, c'est à dire l'espace mémoire dont on a besoin. Ces études de complexité sont souvent menées dans le pire des cas, mais aussi dans le meilleur des cas ou en moyenne.

1.4.1. Complexité des algorithmes

Définition 1.4 (Complexité algorithmique dans le pire des cas)**.** *Soit A un algorithme et $f : \mathbb{N} \to \mathbb{N}$ une fonction. On dit que la complexité dans le pire des cas de A pour une ressource R est en $O(f(n))$ s'il existe des constantes c et n_0 telles que la consommation de l'algorithme A pour toute instance de taille au plus n, notée $R(A,n)$, est majorée par $c \times f(n)$ à partir d'un certain $n > n_0$:*

$$\forall n > n_0, R(A,n) < c \times f(n) \tag{1.1}$$

Les fonctions utilisées pour analyser la complexité des algorithmes (et des problèmes) sont souvent restreintes à quelques fonctions usuelles représentées dans le

tableau 1.1 [6].

Complexité	Type de complexité	Temps d'exécution (en secondes)		
		$t = 10$	$t = 100$	$t = 1000$
1	constante	1	1	1
$log(t)$	logarithmique	1	2	3
t	linéaire (polynomiale)	10	100	1000
t^2	quadratique (polynomiale)	100	10^4	10^6
t^3	cubique (polynomiale)	100	10^6	10^9
2^t	exponentielle	1000	10^{30}	10^{300}
$t!$	factorielle	10^6	10^{157}	10^{2567}

Tableau 1.1. – Ordre de grandeur du temps d'exécution d'un algorithme selon sa complexité et la taille t de l'instance en entrée.

Exemple 1.5. *On s'intéresse à la complexité de l'algorithme de primalité présenté dans l'algorithme 1.1. D'une part, étant donné un nombre n écrit en base* 2*, on a besoin de* $t = \lceil log_2(n) + 1 \rceil$ *chiffres pour écrire n. L'algorithme se compose d'une boucle (au plus n itérations) et d'un test que l'on peut, pour simplifier, considérer comme étant une opération atomique. La complexité temporelle de l'algorithme est donc en* $O(n) = O(2^t)$ *opérations atomiques. Nous avons donc un algorithme exponentiel* [7].

1.4.2. Complexité des problèmes

En théorie de la complexité, on ne s'intéresse pas uniquement à la complexité des algorithmes mais aussi et surtout à la complexité des problèmes. La complexité d'un problème se résume souvent par la complexité du meilleur algorithme pour ce problème. Les problèmes sont ensuite regroupés en différentes classes de complexité, dont voici les principales classes pour la complexité temporelle [8].

Définition 1.5 (Classe P)**.** *On dit qu'un problème appartient à la classe P (pour polynomial time) s'il existe un algorithme polynomial pour ce problème.*

Définition 1.6 (Classe NP)**.** *On dit qu'un problème appartient à la classe NP (pour non-deterministic polynomial time) si, pour chaque instance positive i du problème, il existe un certificat de taille polynomiale par rapport à la taille de l'instance et un algorithme vérificateur de complexité polynomiale par rapport à la taille du certificat et de l'instance permettant de vérifier que l'instance i est bien une instance positive du problème.*

6. tableau partiellement repris depuis [Abr15]

7. On parle souvent dans ce cas-là d'algorithme pseudo-polynomial. Un algorithme pseudo-polynomial est un algorithme de complexité temporelle exponentielle par rapport à la taille de l'instance en entrée mais polynomiale par rapport à sa valeur, ce qui est le cas ici.

8. On utilisera des définitions qui ne font pas mention des machines de Turing.

Définition 1.7 (Classe EXP)**.** *On dit qu'un problème appartient à la classe EXP (pour exponential time) s'il existe un algorithme exponentiel pour ce problème.*

Exemple 1.6. *Le problème de primalité appartient à la classe P. En effet, bien que l'algorithme 1.1 ne soit pas un algorithme polynomial, il existe bien un algorithme polynomial pour le résoudre [AKS02].*

L'étude des problèmes n'a pas seulement permis de les regrouper dans des classes de complexité. Elle a aussi permis d'identifier, au sein de chaque classe de complexité, des problèmes dits *complets*, qui sont les plus difficiles de la classe en question et qui permettent théoriquement de résoudre tous les autres. Afin de détecter qu'un problème est complet pour une classe de complexité, il suffit de dire qu'il est dans cette classe et qu'il existe une réduction polynomiale de n'importe quel problème de cette classe vers ce problème-là.

Définition 1.8 (Réduction polynomiale)**.** *Soit P_1 et P_2 deux problèmes de décision. Une réduction polynomiale du problème P_1 vers le problème P_2 est un algorithme (au pire) polynomial permettant de transformer n'importe quelle instance p_1 du problème P_1 en une instance p_2 du problème P_2 de taille (au pire) polynomiale par rapport à la taille de p_1, et telle que p_1 est positive dans P_1 si et seulement si p_2 est positive dans P_2.*

Définition 1.9 (C-difficile)**.** *Soit P un problème de décision et C une classe de complexité. On dit que P est C-difficile si, pour tout problème de décision P' appartenant à la classe C, il existe une réduction polynomiale de P' vers P.*

Définition 1.10 (C-complet)**.** *Soit P un problème de décision et C une classe de complexité. On dit que P est C-complet si P appartient à la classe de complexité C et si P est C-difficile.*

La littérature regorge de problèmes complets, en particulier pour la classe NP. Le problème SAT a été le premier problème à avoir été démontré NP-complet [Coo71] et depuis de nombreux problèmes de la classe NP ont été démontrés NP-complets, souvent par réduction polynomiale depuis le problème SAT. En effet, dès lors que l'on connaît un problème NP-difficile, il suffit de démontrer qu'il existe une réduction polynomiale de ce problème vers un autre problème pour démontrer que celui-ci est NP-difficile.

Théorème 1.1. *Soit P_1 et P_2 deux problèmes de décision et C une classe de complexité. Si P_1 est C-difficile et s'il existe une réduction polynomiale de P_1 vers P_2, alors P_2 est C-difficile.*

Corollaire 1.1. *Soit P_1 et P_2 deux problèmes de décision et C une classe de complexité. Si P_1 est C-complet, si P_2 appartient à la classe de complexité C et s'il existe une réduction polynomiale de P_1 vers P_2, alors P_2 est C-complet.*

La théorie de la complexité possède cependant de nombreuses inconnues à propos des relations entre les diverses classes de complexité. En effet, si on connaît

la relation d'appartenance entre les diverses classes de problèmes depuis les plus simples jusqu'aux plus difficiles (par exemple, $P \subseteq NP$), on ignore s'il y a ou non une relation d'appartenance depuis les classes de problèmes les plus difficiles jusqu'aux plus simples. En particulier, le problème $P \stackrel{?}{=} NP$ est notamment l'une des questions ouvertes les plus importantes en informatique théorique et, si la majorité des chercheurs pensent que $P \neq NP$, aucune preuve de ce résultat n'a été diffusée à ce jour. En fonction de ce résultat, la relation d'inclusion entre les classes peut évoluer selon le diagramme présenté dans la figure 1.2.

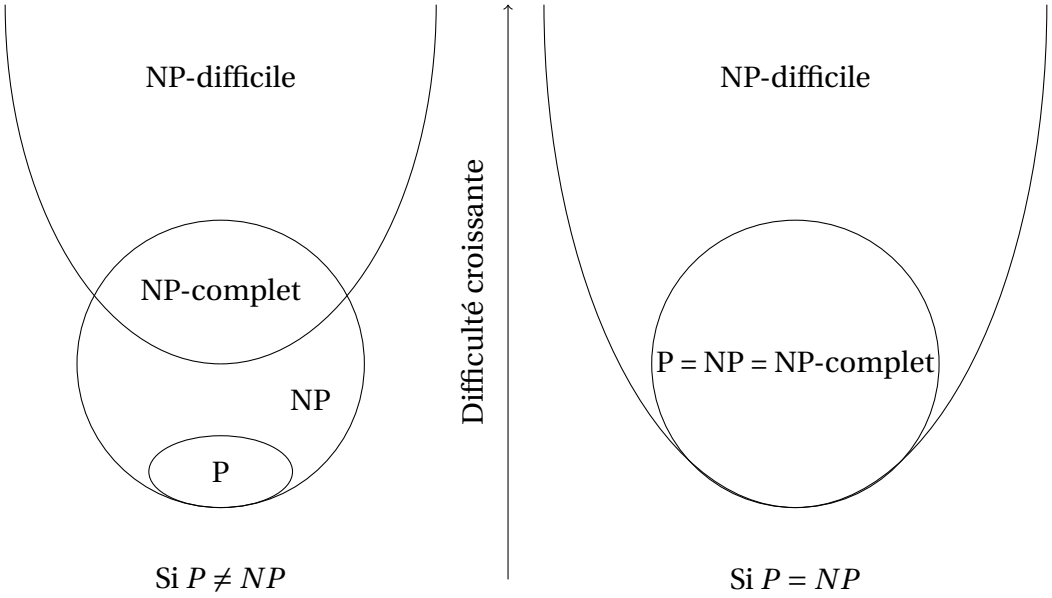


FIGURE 1.2. – Diagramme d'Euler des classes P, NP, NP-complet et NP-difficile

1.5. Logique booléenne et formules propositionnelles

Pour comprendre les problèmes SAT et Max-SAT, on introduit dans cette section les formules propositionnelles. Les formules propositionnelles sont composées de variables booléennes (que l'on peut affecter à $vrai$ ou $faux$) et de quelques opérateurs (la négation, la conjonction et la disjonction principalement). En particulier, on s'intéressera uniquement aux formules sous forme normale conjonctive que l'on va définir plus loin.

1.5.1. Formules propositionnelles

Les symboles de la logique propositionnelles sont les suivants :
— Les constantes *vrai* et *faux*, que l'on notera respectivement 1 et 0

— Les variables booléennes (appelées aussi variables propositionnelles) pouvant prendre une valeur qui est soit 1 soit 0
— L'opérateur unaire de négation $\neg$ ou $\overline{\ }$
— L'opérateur binaire de conjonction $\wedge$
— L'opérateur binaire de disjonction $\vee$
— L'opérateur binaire d'implication logique $\Rightarrow$
— L'opérateur binaire d'équivalence logique $\Leftrightarrow$
— Les symboles de priorité (et)

En utilisant ces symboles, on peut désormais construire des formules propositionnelles :

Définition 1.11 (Formule propositionnelle)**.** *Soit $X = \{x_1, ..., x_n\}$ un ensemble de variables booléennes. Une formule propositionnelle est construite récursivement suivant les règles suivantes :*

— *Les constantes* 1, 0 *et les variables de X sont des formules*
— *Si ϕ est une formule, alors (ϕ) est une formule*
— *Si ϕ est une formule, alors $\overline{\phi}$ est une formule*
— *Si ϕ et ϕ' sont des formules, alors $\phi \wedge \phi'$ est une formule*
— *Si ϕ et ϕ' sont des formules, alors $\phi \vee \phi'$ est une formule*
— *Si ϕ et ϕ' sont des formules, alors $\phi \Rightarrow \phi'$ est une formule*
— *Si ϕ et ϕ' sont des formules, alors $\phi \Leftrightarrow \phi'$ est une formule*

Maintenant que l'on sait définir des formules propositionnelles, on peut s'intéresser à affecter à chaque variable de la formule une valeur booléenne, puis à déduire la valeur de la formule ainsi obtenue.

Définition 1.12 (Interprétation d'une variable)**.** *Soit x une variable booléenne. Interpréter (ou affecter) x consiste à lui associer une valeur qui est soit* 1*, soit* 0*.*

Définition 1.13 (Interprétation d'un ensemble de variables)**.** *Soit X un ensemble de variables booléennes. Une interprétation (ou affectation) complète des variables $I : X \rightarrow \{vrai, faux\}$ associe chaque variable à une valeur booléenne (soit* 1*, soit* 0*). Une interprétation partielle des variables $I : p(X) \rightarrow \{vrai, faux\}$ (avec $p(X) \subseteq X$) associe certaines variables à une valeur booléenne (soit* 1*, soit* 0*).*

Par simplification, on parlera d'interprétation (sans précision de la complétude) pour parler d'une interprétation complète. En interprétant les variables d'une formule propositionnelle, on interprète par conséquent la formule elle-même. En effet, étant donné l'interprétation de deux formules ϕ et ϕ', l'interprétation des variables d'une formule se répercute elle-même suivant les règles décrites dans le tableau 1.2.

$I(\phi)$	$I(\phi')$	$I(\top)$	$I(\bot)$	$I(\overline{\phi})$	$I(\phi \vee \phi')$	$I(\phi \wedge \phi')$	$I(\phi \Rightarrow \phi')$	$I(\phi \Leftrightarrow \phi')$
0	0	1	0	1	0	0	1	1
0	1	1	0	1	1	0	1	0
1	0	1	0	0	1	0	0	0
1	1	1	0	0	1	1	1	1

Tableau 1.2. – Table de vérité des opérateurs de la logique booléenne

1.5.2. Forme Normale Conjonctive

Dans le cadre des problèmes SAT et Max-SAT, on considère un cas particulier des formules propositionnelles : les formules propositionnelles sous forme normale conjonctive (en anglais CNF pour *Conjunctive Normal Form*). On va donc introduire ici l'ensemble du vocabulaire et des notations utile dans le cadre des formules CNF. On considère dans cette section un ensemble de variables $X = \{x_1, ..., x_n\}$.

Tout d'abord, un littéral est soit une variable x, soit sa négation. Un littéral $l = x$ est satisfait par une interprétation (a pour valeur 1) si et seulement si la variable x est elle-même satisfaite par cette interprétation et falsifié (a pour valeur 0) sinon. Un littéral $l = \overline{x}$ est satisfait par une interprétation (a pour valeur 1) si et seulement si la variable x est elle-même falsifiée par cette interprétation et falsifié (a pour valeur 0) sinon. Remarquons que l'on représente souvent une interprétation d'un ensemble de variables comme un ensemble de littéraux, en considérant que si, par exemple, une variable x est affectée négativement, alors le littéral $\overline{x}$ appartient à l'interprétation.

Définition 1.14 (Littéral)**.** *Un littéral l est une variable $x \in X$ ou sa négation $\overline{x}$.*

Une clause est une disjonction de littéraux. Une clause est satisfaite par une interprétation I si au moins de ses littéraux est satisfait par I, sinon elle est falsifiée par l'interprétation I.

Définition 1.15 (Clause)**.** *Une clause x est une disjonction de littéraux $(l_1 \vee l_2 \vee ... \vee l_k)$ et peut-être aussi représentée comme un ensemble de littéraux $\{l_1, l_2, ..., l_k\}$. On note $var(c)$ l'ensemble des variables qui appartiennent à c.*

Parmi les clauses particulières, notons l'existence des clauses forcément satisfaites (les clauses tautologiques) et celles forcément falsifiées (les clauses vides). Enfin, les clauses unitaires ont également un intérêt pour la résolution du problème SAT notamment.

Définition 1.16 (Clause tautologique)**.** *Une clause tautologique est une clause qui contient un littéral l et sa négation $\overline{l}$.*

Définition 1.17 (Clause vide)**.** *La clause vide (notée $\square$) ne contient aucun littéral et est toujours falsifiée.*

Définition 1.18 (Clause unitaire)**.** *La clause unitaire est une clause contenant un seul littéral.*

Ensuite, on s'intéresse parfois aux relations des clauses les unes par rapport aux autres.

Définition 1.19 (Clause sous-sommée)**.** *Une clause c sous-somme une clause c' si chaque littéral de c est un littéral de c', c'est à dire si $\forall l \in c$, $l \in c'$. On dit aussi que la clause c' est sous-sommée par la clause c.*

Définition 1.20 (Clauses en opposition)**.** *Une clause c s'oppose à une clause c' si c contient un littéral dont la négation est dans c', c'est à dire si $\exists l \in c$, $\overline{l} \in c'$.*

Une formule propositionnelle sous forme normale conjonctive est une conjonction de clauses. Dans la suite de ce manuscrit, on considérera toujours qu'une formule est sous forme normale conjonctive. Une formule propositionnelle sous forme normale conjonctive est satisfaite par une interprétation I si I satisfait chacune des clauses de la formule. Elle est falsifiée par une interprétation I si I falsifie au moins une des clauses de la formule.

Définition 1.21 (Formule propositionnelle sous forme normale conjonctive)**.** *Une formule ϕ sous forme normale conjonctive (CNF de l'anglais Conjunctive Normal Form) est une conjonction de clauses $\phi = c_1 \wedge c_2 \wedge ... \wedge c_m$ et peut-être aussi représentée comme un multi-ensemble de clauses. On note $var(\phi)$ les variables qui sont dans la formule ϕ.*

Définition 1.22 (Modèle)**.** *Si une formule CNF ϕ est satisfaite par une interprétation I, alors on dit que I est un modèle de ϕ.*

Définition 1.23 (Formule satisfiable ou insatisfiable)**.** *Une formule ϕ est satisfiable s'il existe un modèle de ϕ, sinon elle est insatisfiable ou inconsistante.*

Définition 1.24 (Cœur inconsistant)**.** *Étant donnée une formule ϕ insatisfiable, on appelle cœur inconsistant un sous-ensemble de clauses de ϕ qui est lui-même insatisfiable.*

Pour finir, on définit une formule obtenue après propagation d'un littéral. Cette définition se généralise naturellement pour des formules après propagation d'un ensemble de littéraux : il suffit alors de propager successivement chacun des littéraux.

Définition 1.25 (Formule après propagation)**.** *Étant donnée une formule ϕ, on note $\phi_{|l}$ obtenue après la propagation du littéral l sur la formule. Pour obtenir $\phi_{|l}$ à partir de ϕ, il suffit de supprimer toutes les clauses de ϕ contenant le littéral l (ces clauses sont satisfaites) et de supprimer toutes les occurrences de $\overline{l}$ dans les clauses dans lesquels il apparaît.*

1.6. Conclusion

Dans ce chapitre, on a défini les notions utiles à la compréhension des problèmes SAT et Max-SAT. On a définit les problèmes de décision et d'optimisation, les algorithmes et les principales classes de complexité. Enfin, on a présenté les formules propositionnelles sous forme normale conjonctive, ce qui permettra dans les chapitres 2 et 3 de présenter les problèmes SAT et Max-SAT.

2. Le problème SAT

Sommaire

2.1. Introduction

Dans ce chapitre, on présente le problème de satisfiabilité propositionnelle (SAT). Étant donnée une formule propositionnelle sous forme normale conjonctive, le problème SAT consiste à déterminer si cette formule est satisfiable. Tout d'abord, on définit le problème SAT (section 2.2) et on en présente les principales règles d'inférence qui permettent de transformer la formule et ainsi de simplifier la résolution du problème SAT (section 2.3). Ensuite, on présente les principales de méthodes de résolutions pour SAT (section 2.4). Enfin, on s'intéresse aux certificats pour le problème SAT, c'est à dire aux preuves de satisfiabilité et d'insatisfiabilité d'une formule (section 2.5).

2.2. Définition

Définition 2.1 (Problème SAT)**.** *Étant donnée une formule propositionnelle ϕ sous forme normale conjonctive, le problème de satisfiabilité booléenne, ou problème SAT, consiste à déterminer si ϕ est satisfiable ou non.*

Notons que, par simplification, on considère souvent dans la littérature que résoudre le problème SAT consiste, étant donnée une formule propositionnelle sous forme normale conjonctive, à en exhiber un modèle, si cela est possible, et à dire (voire à démontrer) que cela est impossible sinon. Deux raisons peuvent justifier cette simplification. Tout d'abord, la plupart des méthodes pour résoudre le problème SAT fonctionnent de telle sorte qu'à l'instant où elles arrivent à déterminer qu'une formule est satisfiable, elles ont trouvé au même instant un modèle pour cette formule. Enfin, si le problème SAT est résolu dans un but applicatif (pour résoudre un problème industriel par exemple), il apparaît incohérent de ne faire que résoudre le problème SAT car exhiber un modèle, voire donner la raison de l'insatisfiabilité de la formule, peut permettre de répondre plus efficacement au besoin applicatif sous-jacent.

Le problème SAT est un problème central en informatique théorique. Tout d'abord, du point de vue de la théorie de la complexité, SAT est le premier problème à avoir été démontré NP-complet [Coo71]. Les autres problèmes à avoir été démontrés NP-complets ou NP-difficiles l'ont été par réduction polynomiale depuis un autre problème lui-même NP-difficile, et l'ensemble de ces démonstrations a pour point d'appui le fait que SAT ait été démontré NP-complet.

Théorème 2.1. *Le problème SAT est NP-complet. [Coo71]*

Ensuite, le problème SAT permet de modéliser un nombre important de problèmes du monde réel ou académique, comme la vérification de modèles [Cla+01 ; Bie+99a ; Bie+99b] ou de programmes [CKL04], la planification [KS99], ou encore les preuves mathématiques assistées par informatique [KL14 ; KL15 ; HKM16 ; HK17]. Pour plus de détails sur le problème SAT, on invite le lecteur à lire le *Handbook of satisfiability* [Bie+09], édité en 2009 et qui vient d'être mis à jour et réédité en 2021.

2.3. Règle d'inférences pour SAT

Les règles d'inférence pour SAT permettent de déduire de nouvelles informations ou de simplifier la formule initiale, l'objectif de ces transformations étant bien-sûr de faciliter la résolution du problème SAT. On peut distinguer deux types de règles d'inférence, les règles syntaxiques qui permettent de déduire de nouvelles clauses et les règles sémantiques qui permettent de fixer la valeur de certaines variables de l'instance. Ces règles d'inférence respectent généralement l'équivalence SAT définie ci-dessous qui dit que toute interprétation qui satisfait (respectivement falsifie) la formule avant l'application de la règle d'inférence satisfait (respectivement falsifie) la formule après application de la règle d'inférence. On peut aussi se contenter de

contraintes plus faibles, par exemple en s'assurant que la formule avant transformation est satisfiable si et seulement si la formule après l'est aussi.

Définition 2.2 (Équivalence SAT)**.** *On dit que deux formules ϕ et ϕ' sont équivalentes (pour le problème SAT) si pour toute interprétation I de leur variable, I est un modèle de ϕ si et seulement si I est un modèle de ϕ'.*

2.3.1. Règles syntaxiques

La liste des règles syntaxiques ci-dessous déduisent de nouvelles clauses à partir de clauses initiales. On note ces règles sous la forme suivante, où C_1 est l'ensemble des clauses initiales et C_2 l'ensemble des clauses déduites :

$$\frac{C_1}{C_2}$$

Suppression des clauses tautologiques Une clause tautologique est une clause de la forme $c = (x \vee \overline{x} \vee A)$ et elle est toujours satisfaite quelque soit l'interprétation des variables. Elle peut donc être supprimée de la formule sans en impacter sa satisfiabilité.

$$\frac{c = (x \vee \overline{x} \vee A)}{}$$

Suppression des clauses identiques Si une formule possède deux clauses identiques c et c', alors on peut supprimer l'une des deux de la formule sans en impacter sa satisfiabilité.

$$\frac{c = (A) \quad c' = (A)}{c}$$

Suppression des clauses sous-sommées Si une formule possède deux clauses c et c' telles que c sous-somme c', alors on peut supprimer la clause c' de la formule sans en impacter sa satisfiabilité. Cette règle d'inférence généralise la suppression des clauses identiques.

$$\frac{c = (A) \quad c' = (A \vee B)}{c}$$

Fusion des littéraux identiques Si une clause contient plusieurs fois le même littéral, on peut les supprimer jusqu'à n'en avoir qu'une seule occurrence dans la clause.

$$\frac{c = (l \vee l \vee A)}{c = (l \vee A)}$$

Règle de résolution L'une des règles les plus utilisées dans les algorithmes de résolution du problème SAT est la règle de résolution, introduite par John Alan Robinson en 1965 [Rob65]. Cette règle permet, étant données deux clauses $(x \vee A)$ et $(\overline{x} \vee B)$ qui s'opposent sur une variable x (que l'on appellera variable pivot de la résolution), de déduire, en plus des deux premières clauses, une troisième clause contenant l'union de leurs variables à l'exception de cette variable d'opposition.

$$\frac{c_1 = (x \vee A) \qquad c_2 = (\overline{x} \vee B)}{c_3 = (A \vee B)}$$

Affaiblissement d'une clause Cette règle est la règle réciproque de la règle des clauses sous-sommées. Elle dit simplement que si une formule contient une clause c, le fait d'ajouter une clause sous-sommée par c n'impacte pas la satisfiabilité. Cette règle est très peu utilisée dans le cadre du problème SAT mais on verra que la règle équivalente pour le problème Max-SAT est extrêmement utile.

$$\frac{c = (A)}{c' = (A \vee B)}$$

2.3.2. Règles sémantiques

Propagation Unitaire Si une formule possède une clause unitaire $c = l$, alors le littéral l ne pourra être falsifié par toute interprétation satisfaisant la formule. Par conséquent, il est équivalent de chercher à satisfaire la formule obtenue après la propagation du littéral l. Ainsi, toutes les clauses contenant le littéral l seront désormais satisfaites alors que celles contenant le littéral $\overline{l}$ auront un littéral en moins.

Règle des littéraux purs Si une variable apparaît uniquement positivement (respectivement négativement) dans la formule, alors elle peut être fixée à vrai (respectivement à faux).

2.4. Résolution du problème SAT

Dans cette section, on s'intéresse aux méthodes de résolution complètes pour le problème SAT. En particulier, on s'intéresse à un ensemble de méthodes qui sont basées sur l'algorithme DPLL proposé en 1962 [DLL62]. Ces méthodes explorent l'espace de recherche en faisant successivement des choix sur la valeur des variables et en appliquant certaines règles d'inférence, en particulier la propagation unitaire. La pertinence pratique de ces méthodes est argumentée par les résultats de la compétition SAT [MF], qui a lieu chaque année depuis 2002 et participe à améliorer l'efficacité pratique des solveurs SAT actuels en invitant les chercheurs du domaines à soumettre leurs solveurs, ce qui permet une comparaison pertinente de leurs performances et une visualisation plus aisé de l'état de l'art dans le domaine.

2.4.1. Algorithme DPLL

L'algorithme de David, Putnam, Loveland et Logemann (DPLL [DLL62]) est un algorithme de type recherche arborescente. C'est une amélioration de l'algorithme DP [DP60] ainsi que de l'algorithme de Quine [Qui50]. L'algorithme DPLL applique deux règles d'inférence : la propagation unitaire et la règle des littéraux purs. Après application de ces deux règles, une variable est sélectionnée et deux cas sont alors étudiés : le cas où la valeur de la variable est fixée à 1 et le cas où elle est fixée à 0. Dans ces deux cas, on répète la même procédure jusqu'à pouvoir conclure sur la satisfiabilité ou l'insatisfiabilité du cas étudié. Si l'étude d'un cas mène à conclure que la formule est insatisfiable suivant les hypothèses faites sur la valeur des variables, alors un retour en arrière est effectué pour explorer une autre hypothèse. La formule est satisfiable si l'une des séquences d'hypothèses effectuées permet de satisfaire toutes les clauses et insatisfiable si tous les cas étudiés ont menés à une formule insatisfiable. Une version récursive de l'algorithme DPLL est proposé dans l'algorithme 2.1 où on note $\phi_{|l}$ la formule obtenue après propagation du littéral l dans la formule ϕ. De plus, $DPLL(\phi_{|x})$ ou $DPLL(\phi_{|\overline{x}})$ vaut satisfiable si l'un des deux appels retourne satisfiable.

Algorithme 2.1 Algorithme de David, Putnam, Loveland et Logemann (DPLL)

Entrée : Formule CNF ϕ
Sortie : SATISFIABLE or INSATISFIABLE
1: **si** ϕ est vide **alors**
2: **retourner** SATISFIABLE
3: **si** $\square \in \phi$ **alors**
4: **retourner** INSATISFIABLE
5: **si** ϕ contient une clause unitaire (l) **alors**
6: **retourner** $DPLL(\phi_{|l})$
7: **si** ϕ contient un littéral pur l **alors**
8: **retourner** $DPLL(\phi_{|l})$
9: Choisir une variable $x \in \phi$
10: **retourner** $DPLL(\phi_{|x})$ ou $DPLL(\phi_{|\overline{x}})$

Exemple 2.1. *Considérons la formule* $\phi = c_1 \wedge c_2 \wedge c_3 \wedge c_4$ *avec* $c_1 = (\overline{x_1} \vee \overline{x_3} \vee x_4)$, $c_2 = (x_2 \vee x_3)$, $c_3 = (\overline{x_2} \vee x_3)$ *et* $c_4 = (\overline{x_3} \vee x_4)$. *L'application de l'algorithme DPLL sur la formule* ϕ, *en sélectionnant les variables dans l'ordre lexicographique et en négligeant l'utilisation de la règle des littéraux purs (par mesure de simplification) est représentée par la figure 2.1.*

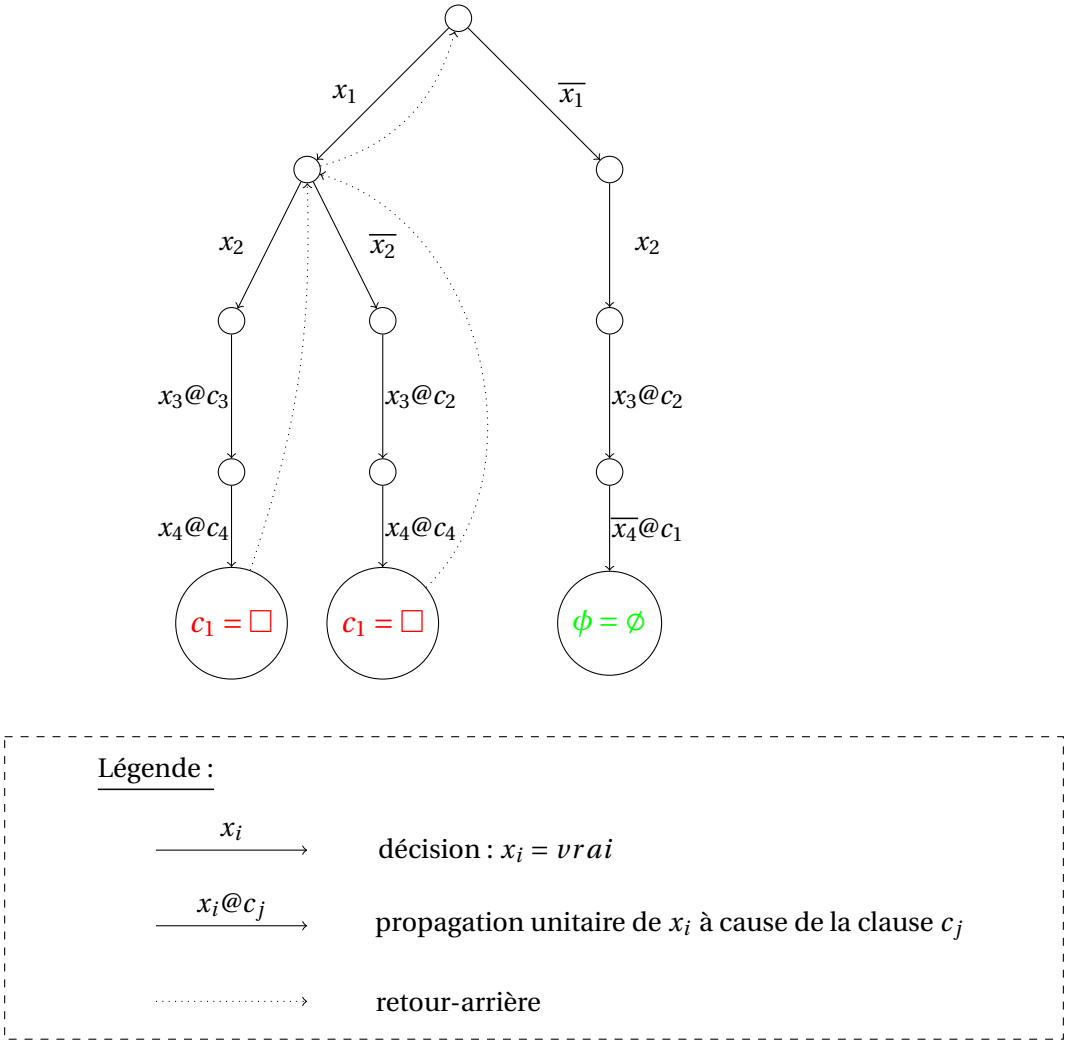


FIGURE 2.1. – Exemple d'application de l'algorithme DPLL [Abr15]

Notons l'importance des règles d'inférence dans l'élagage de l'arbre de recherche exploré par l'algorithme DPLL. En plus de la stricte résolution du problème SAT, l'algorithme DPLL est capable de retourner le modèle lorsque la formule est satisfiable. En effet, il suffit de se souvenir des propagations qui ont été faites jusqu'à ce que la formule soit déclarée satisfiable pour ainsi déduire l'interprétation (complète ou partielle) qui a mené à cette conclusion.

2.4.2. Les solveurs CDCL

L'algorithme DPLL est l'une des méthodes de base pour résoudre le problème SAT encore aujourd'hui. Celle-ci est en effet largement utilisée dans les solveurs actuels, avec toute une série d'améliorations pratiques qui sont présentées dans cette section. De très nombreux solveurs SAT modernes sont basés sur l'algorithme DPLL et ses différentes améliorations parmi lesquels GRASP [SS99], SATO [Zha97], Chaff

[Mos+01], Minisat [ES03], Glucose [AS09; AS18], Cadical[Bie+20] ou encore Kissat [Bie+20]. Les solveurs modernes dont ceux listés ci-dessus sont souvent regroupés sous la terminologie des solveurs *CDCL* (Conflict Driven Clause Learning). Deux éléments importants proviennent directement de l'algorithme DPLL : la propagation unitaire et l'exploration de l'arbre de recherche en affectant des valeurs aux variables.

2.4.2.1. Analyse de conflit

Lorsque l'exploration de l'arbre de recherche conduit à une interprétation partielle qui falsifie la formule (on parle alors de conflit), il peut être intéressant de s'arrêter et de déterminer les causes de ce conflit avant de continuer la recherche. L'analyse de conflit permet d'ajouter une nouvelle clause correspondant au conflit rencontré [SS96]. Cette nouvelle clause permettra, lorsque l'exploration de l'arbre de recherche mène à une situation similaire, d'éviter plus rapidement le conflit en question.

En particulier, le fait d'analyser un conflit peut permettre de faire des retours en arrière non chronologiques [SS99; Mos+01; MB19; NR]. Dans l'algorithme DPLL, les retours en arrière se font de manière chronologique, c'est à dire que si on est au niveau k de l'arbre de recherche (on a choisi la valeur de k littéraux), un conflit conduit à retourner au niveau $(k-1)$. Cependant, l'analyse de conflits peut permettre de déduire que le conflit actuel est dû à une décision qui a été prise à un niveau bien antérieur au niveau $(k-1)$. Dans cette situation, il peut être intéressant de remonter dans l'arbre de recherche jusqu'au niveau en question et recommencer l'exploration de l'arbre de recherche avec cette nouvelle information.

2.4.2.2. Heuristiques de branchement

Dans l'algorithme DPLL, le choix de la variable sur laquelle brancher a un impact significatif sur les performances pratiques d'un solveur car elle influence fortement la taille de l'arbre de recherche. L'heuristique de branchement choisie doit trouver un juste équilibre entre sa complexité (temporelle et/ou spatiale), afin de ne pas consommer trop de ressources dans la sélection des variables de branchement, et son efficacité, afin de brancher rapidement sur des variables qui permettront une simplification importante de la formule. Le choix de la variable de branchement n'est cependant pas facile et il est connu qu'il est NP-difficile de choisir le littéral optimal sur lequel brancher [Lib00].

Exemple 2.2. *Si on considère l'exemple 2.1, brancher dans l'ordre lexicographique n'était pas pertinent. En effet, brancher sur n'importe laquelle des variables x_2, x_3 ou x_4 aurait permis de simplifier immédiatement la formule par propagation unitaire.*

De nombreux travaux autour des heuristiques de branchement existent depuis les années 1990. Parmi les heuristiques de branchement de l'état de l'art, on peut citer les heuristiques qui cherchent à favoriser les variables qui apparaissent dans des clauses

de petite taille (heuristiques syntaxiques) [JW90 ; BB92], les heuristiques qui cherchent à simuler l'impact du choix de la variable sur la suite de la recherche (heuristiques *look-ahead*) [LA97 ; DD03] et enfin les heuristiques qui cherchent à apprendre au fur et à mesure de la recherche les variables sur lesquelles brancher (heuristiques *look-back*) [Mos+01 ; Lia+16 ; BGS99]. Ces dernières heuristiques, et notamment l'heuristique VSIDS (Variable State Independent Decaying Sum) [Mos+01] et CHB [Lia+16] (ainsi que leurs variantes) figurent parmi les heuristiques les plus couramment utilisés par les solveurs modernes. Elles conservent un score pour chaque littéral qu'elles mettent à jour régulièrement pendant l'exploration de l'arbre de recherche. Remarquons enfin qu'ajouter une part d'aléatoire dans le choix du prochain littéral peut parfois avoir un intérêt pratique [Sil99].

2.4.2.3. Structures paresseuses

Pour appliquer la propagation unitaire, il faut être en mesure de la détecter et de l'appliquer efficacement. Pour cela, une méthode naïve consiste à stocker, pour chaque clause, le nombre de littéraux encore non assignés et, pour chaque littéral, l'ensemble des clauses auxquelles il appartient. L'inconvénient de cette méthode naïve est qu'elle entraîne, du point de vue de la détection et de l'application de la propagation unitaire, des opérations inutiles. En effet, s'il est utile de savoir qu'une clause passe de deux littéraux non assignés à un seul littéral non assigné, les autres valeurs des compteurs sont inutiles et on peut perdre du temps à décrémenter et incrémenter des compteurs pour des clauses contenant strictement plus de deux littéraux non encore assignés. Ainsi, plutôt que de stocker pour chaque clause un compteur sur le nombre de littéraux non assignés, les solveurs modernes stockent deux littéraux encore non assignés par clause. Lorsque l'un de ses deux littéraux est assigné, on cherche un autre littéral pour prendre sa place et, si ce n'est pas possible, on détecte alors qu'il faut lancer la propagation unitaire sur le littéral restant. Ce type de stockage est appelé *structures paresseuses*.

Les structures paresseuses sont implémentées différemment suivant les solveurs. Le solveur SATO [Zha97] utilise deux pointeurs pour pointer respectivement le littéral non affecté en tête et en queue d'une clause [ZS00]. Le solveur Chaff [Mos+01] utilise la structure dite des littéraux observés. Il s'agit de deux pointeurs vers deux littéraux non affectés de chaque clause. Ce système améliore le précédent à cause du fait qu'un retour en arrière peut entraîner des modifications de la tête et de la queue d'une clause alors que cela n'entraîne aucun traitement conséquent en utilisant la structure des littéraux observés.

2.4.2.4. Redémarrages

Lors de l'exploration de l'arbre de recherche, il peut arriver que l'algorithme se perde dans une portion de l'arbre de recherche qui nécessitera un temps exponentiel pour être exploré alors que, en explorant différemment l'arbre de recherche et/ou en recommençant la recherche en utilisant les clauses apprises, le problème tout

entier pourrait être résolu plus vite [GSK98 ; GSC97 ; Gom+00]. Pour ces raisons, les solveurs modernes intègrent un système de redémarrage qui permet de recommencer l'exécution de l'algorithme en conservant les informations apprises telles que les nouvelles clauses ou l'influence de la recherche sur l'heuristique de branchement [Mos+01 ; ES03 ; Bie08].

2.5. Certificats pour le problème SAT

Lors de la résolution d'un problème algorithmique, on se pose souvent la question d'un certificat, c'est à dire, lorsque la réponse au problème algorithmique est "Oui", d'une preuve de la réponse donnée, de taille raisonnable par rapport à la taille de l'instance, et dont la vérification de ce certificat est facile par rapport à la résolution du problème. Dans le cadre du problème SAT, la question du certificat se pose à la fois lorsque la formule est satisfiable et lorsqu'elle est insatisfiable.

2.5.1. Certificat de satisfiabilité

Lorsque qu'une formule est satisfiable, il y a un certificat intuitif et bien connu : un modèle pour la formule, c'est à dire une interprétation des variables permettant de satisfaire la formule. Le modèle est un certificat de taille raisonnable, ici linéaire par rapport au nombre de variables (ce qui est raisonnable par rapport à la taille de la formule si on ne prend pas en compte certaines formules dont des variables seraient absentes des clauses). L'opération qui permet de vérifier qu'un modèle satisfait bien la formule est quant à elle linéaire par rapport à la taille de la formule (et du modèle).

Exemple 2.3. *Soit* $\phi = (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x2})$. *La formule* ϕ *est satisfiable et l'interprétation* $x_1 = 1$, $x_2 = 0$ *(que l'on peut aussi noter* $(x_1, \overline{x_2})$*) est un certificat de la satisfiabilité de* ϕ.

2.5.2. Certificat d'insatisfiabilité

Lorsqu'une formule est insatisfiable, il s'agit cette fois de démontrer qu'il n'existe aucun modèle pour cette formule. Un exemple de certificat bien connu dans ce cas-là est la réfutation par résolution. En effet, lorsqu'une formule est insatisfiable, il existe forcément une séquence de résolutions, partant des clauses de la formule et déduisant successivement de nouvelles clauses jusqu'à en déduire la clause vide [Rob65].

Malheureusement, les réfutations par résolution ne sont pas des certificats de taille au pire polynomiale par rapport à la taille de la formule. En effet, il existe des groupes de formules insatisfiables pour lesquelles on sait que toute réfutation par résolution est de taille exponentielle par rapport à la taille de la formule. Par exemple, il est connu que toute réfutation par résolution du problème des pigeons [1] est dans le cas général exponentielle [Hak85].

1. Le problème des pigeons est un problème insatisfiable qui consiste à essayer d'affecter $(k+1)$ pigeons dans k trous de manière à ce que chaque trou ne contienne au plus qu'un seul pigeon.

Définition 2.3 (Réfutation par résolution)**.** *Soit ϕ une formule CNF. On appelle réfutation par résolution une séquence de résolutions, partant des clauses de la formule et déduisant de nouvelles clauses jusqu'à déduire ultimement $\square$.*

Exemple 2.4. *On considère la formule $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$. Une réfutation par résolution de ϕ est représentée sous la forme d'un graphe (orienté acyclique) dans la figure 2.2*[2]*.*

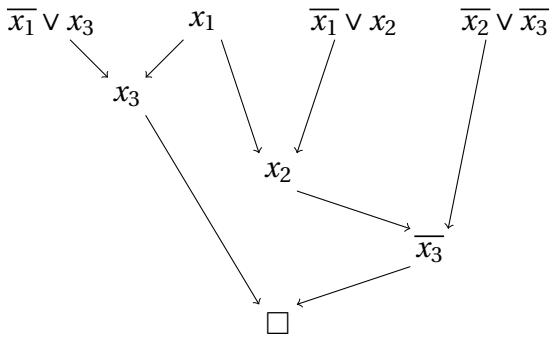


FIGURE 2.2. – Réfutation par résolution

Les réfutations par résolution forment les certificats d'insatisfiabilité les plus connus. Notons toutefois que les certificats d'insatisfiabilité exportés par les solveurs SAT modernes prennent de moins en moins la forme de réfutations par résolution explicites, qui sont certes faciles à vérifier mais prennent beaucoup de place en mémoire et demandent un effort de calcul supplémentaire pour être générés à cause des raffinements utilisés dans les solveurs. Ceux-ci préfèrent utiliser des systèmes de preuve plus efficaces comme la résolution étendue [HJW13b], ou utiliser des format de certificats plus implicites, qui ont certes le désavantage de demander un plus gros effort de calcul pour être vérifiés mais qui apportent des gains en espace mémoire pour être stockés, comme les formats RUP [GN03; Gel08], DRUP [HJW13a], RAT [HJW13b] ou encore DRAT [HJW15; WHJ14]. En pratique, on verra dans les contributions du chapitre 6 qu'on utilisera des réfutations par résolution pour générer des certificats pour le problème Max-SAT [Bie10; Bie06].

De nombreuses classes de réfutations par résolution ont été étudiées dans la littérature, on présente ici certaines classes qui seront utiles pour comprendre certaines contributions de cette thèse : les réfutations *read-once*, les réfutations *tree-like* et les réfutations *regular*. Remarquons qu'une réfutation peut bien entendu être dans plusieurs classes à la fois, c'est le cas des réfutations *tree-like regular* qui seront abordées dans le chapitre 5.

2. Cet exemple provient de [BLM06]

2.5.2.1. Read-once

La première classe de réfutation utile pour comprendre ce document est la classe des réfutations *read-once*. Une réfutation *read-once* est une réfutation particulière telle que chaque clause ne peut être utilisée que pour déduire au plus une nouvelle clause.

Définition 2.4 (Réfutation par résolution *read-once*)**.** *Une réfutation par résolution de ϕ est dite read-once si chaque clause de la réfutation est utilisée au plus une fois en tant que prémisse d'une étape de résolution.*

Exemple 2.5. *La réfutation par résolution présentée figure 2.2 n'est pas read-once parce que la clause (x_1) est utilisée deux fois en tant que prémisse d'une étape de résolution.*

Il est connu qu'il existe des formules CNF insatisfiables qui n'admettent pas de réfutations par résolution *read-once*. On verra plus tard que ce type de réfutation a un intérêt dans le cadre du problème Max-SAT, car, jusqu'aux contributions qui font l'objet des chapitres 4 et 5, on ne savait utiliser les réfutations par résolution dans le cadre du problème Max-SAT que si celles-ci étaient *read-once*.

Exemple 2.6. *La formule $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_3)$ est insatisfiable mais n'admet pas de réfutation par résolution read-once [IM95]*[3]*.*

2.5.2.2. Tree-like

La deuxième classe de réfutation est une généralisation de la première classe : la classe des réfutations *tree-like*. Une réfutation *tree-like* est une réfutation particulière telle que seules les clauses de la formule ont le droit d'être utilisées pour déduire strictement plus d'une nouvelle clause. Naturellement donc, toute réfutation *read-once* est également *tree-like*.

Définition 2.5 (Réfutation par résolution *tree-like*)**.** *Une réfutation par résolution de ϕ est dite tree-like si chaque clause de la réfutation qui ne provient pas de la formule d'origine est utilisée au plus une fois en tant que prémisse d'une étape de résolution.*

Exemple 2.7. *La réfutation par résolution présentée figure 2.2 est tree-like parce que toutes les clauses déduites par la réfutation sont utilisées soit* 1 *fois (les trois clauses (x_3), (x_2) et $(\overline{x_3})$), soit* 0 *fois (la clause vide).*

Proposition 2.1. *Toute réfutation par résolution read-once est également tree-like.*

Il est connu que, pour une formule insatisfiable, la taille de la plus petite réfutation par résolution *tree-like* peut être exponentiellement plus grande que la taille de la plus petite réfutation par résolution [BIW04]. Notons d'ailleurs que, pour toute formule insatisfiable, il existe toujours une réfutation par résolution *tree-like* pour cette formule

3. L'idée de la preuve est de démontrer que pour déduire n'importe quelle clause unitaire, il est nécessaire d'utiliser trois clauses. Ainsi, il faudrait au minimum six clauses pour déduire une clause unitaire et la clause unitaire opposée alors que la formule ne possède que cinq clauses.

puisque, dans la mesure où il existe toujours une réfutation par résolution pour cette formule, il suffit de répéter des morceaux de la réfutation jusqu'à la rendre *tree-like* [BIW04].

2.5.2.3. Regular

La troisième classe de réfutation par résolution est la classe des réfutations dites *regular*. Une réfutation *regular* est telle que, dès lors que l'on a enlevé une variable (grâce à une étape de résolution sur cette variable) dans une branche de la réfutation, cette variable ne réapparaît plus jamais.

Définition 2.6 (Réfutation par résolution *regular*)**.** *Une réfutation par résolution de ϕ est dite regular si chaque branche depuis une clause de la formule jusqu'à la clause vide contient au plus une résolution par variable.*

Définition 2.7 (Irrégularité)**.** *Etant donnée une réfutation par résolution de ϕ non regular, une irrégularité sur une variable x est une branche de la réfutation, depuis une clause de la formule jusqu'à la clause vide, qui contient au moins deux résolutions sur la variable x.*

Exemple 2.8. *La réfutation par résolution présentée figure 2.2 est regular parce que les* 5 *branches du graphe depuis les clauses de la formule jusqu'à la clause vide rencontrent chacune au plus une résolution par variable. Par exemple, la branche de gauche qui part depuis la clause $(\overline{x_1} \vee x_3)$ rencontre une résolution sur la variable x_1 et une résolution sur la variable x_3.*

Exemple 2.9. *On considère la réfutation par résolution de la figure 2.3. Cette réfutation n'est pas regular parce que, si on considère la branche qui part de la clause $(x_1 \vee \overline{x_2})$ et qui va jusqu'à la clause vide, cette branche rencontre deux résolutions sur la variable x_1 (la première et la dernière).*

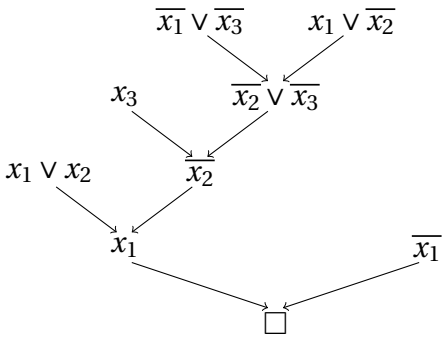


FIGURE 2.3. – Réfutation par résolution non *regular*

Similairement au cas *tree-like*, il est connu que toute formule insatisfiable admet une réfutation par résolution *regular* [Bie+09]. Il semble intuitif de penser que chercher à obtenir une réfutation *regular* est une bonne chose pour la simplicité de la réfutation,

tant on pourrait croire que pour une réfutation, ne pas être *regular* serait un signe que la réfutation comporte des redondances et des portions superflues, et que, d'une certaine manière, en la rendant *regular*, on pourrait également la simplifier et la raccourcir. Pourtant, cette intuition n'est valable que dans le cas des réfutations *tree-like*, où on est toujours sûr qu'en cherchant à simplifier et à réduire la taille de la réfutation, on tombera toujours sur une réfutation à la fois *tree-like* et *regular* [Urq01]. Cependant, cette intuition est fausse dans le cas général, et on sait qu'il existe des formules insatisfiables telle que la taille de la plus petite réfutation par résolution de cette formule est exponentiellement plus petite que la taille de la plus petite réfutation par résolution *regular* de cette formule [Ale+07 ; Ale+02 ; Urq11 ; Urq08].

2.6. Conclusion

Dans ce chapitre, on a présenté le problème SAT, les principales méthodes de résolution pour SAT ainsi que les certificats de satisfiabilité et d'insatisfiabilité. On va présenter dans le chapitre 3 une extension de SAT pour en faire un problème d'optimisation : Max-SAT.

3. Le problème Max-SAT

Sommaire

3.1. Introduction

Le problème Max-SAT est une extension du problème SAT. Étant donnée une formule propositionnelle, constatons qu'il n'est pas toujours suffisant de dire qu'elle est inconsistante. Si, par exemple, cette formule est la représentation formelle d'un problème réel qui a été modélisé, il peut être pertinent d'être en capacité de proposer une solution, qui certes ne permet pas de satisfaire toute la formule, mais qui permet de satisfaire le plus de clauses possibles. Ensuite, remarquons que la plupart des problèmes réels sont davantage des problèmes d'optimisation que des problèmes de décision. Pour ces deux raisons évoquées, il est intéressant d'étendre le problème SAT pour en avoir une version d'optimisation.

3.2. Définition et variantes

Le problème Max-SAT considère en entrée une formule propositionnelle, et, plutôt que de se demander si la formule est satisfiable et comment interpréter les variables pour la satisfaire, on va maintenant se demander quel est le nombre maximum de clauses pouvant être satisfaites par une interprétation des variables. Faisons ici la remarque que, formellement, on considère souvent que l'on cherche à minimiser le nombre de clauses falsifiées au lieu de considérer que l'on cherche à maximiser le nombre de clauses satisfaites. On dit aussi que résoudre le problème Max-SAT consiste à trouver le coût minimum de toutes les interprétations (que l'on appellera *optimum Max-SAT*), le coût d'une interprétation étant le nombre de clauses falsifiées par cette interprétation.

Définition 3.1 (Problème Max-SAT)**.** *Étant donnée une formule propositionnelle ϕ sous forme normale conjonctive, le problème Max-SAT consiste à déterminer le nombre maximum de clauses qu'il est possible de satisfaire par une interprétation des variables. De manière équivalente, le problème Max-SAT consiste à déterminer le nombre minimum de clauses que toute interprétation des variables doit nécessairement falsifier.*

Comme pour le problème SAT, on considère souvent, par simplification, que résoudre le problème Max-SAT (dont la réponse est un nombre) consiste à déterminer une interprétation des variables qui permette de falsifier le nombre minimum de clauses. Encore une fois, cela est dû au fait que la plupart des algorithmes pour le problème Max-SAT trouvent le nombre minimum de clauses à falsifier en même temps qu'une interprétation des variables permettant de falsifier exactement ce nombre de clauses ainsi qu'au fait qu'il est plus pertinent d'être capable de fournir une telle interprétation si la résolution du point de vue applicatif.

Exemple 3.1. *Soit $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$. L'optimum du problème Max-SAT de la formule ϕ est* 1*. En effet, l'interprétation $x_1 = x_2 = x_3 = 1$ falsifie une seule clause et il n'existe aucune interprétation des variables permettant de satisfaire toutes les clauses.*

Remarquons que le problème Max-SAT est NP-difficile [1], par réduction polynomiale depuis le problème SAT. En effet, résoudre le Max-SAT sur une formule permet aussi de résoudre SAT, car l'optimum Max-SAT d'une formule est égal à 0 si et seulement si cette formule est satisfiable.

Théorème 3.1. *Le problème Max-SAT est NP-difficile.*

Le problème Max-SAT énoncé ci-dessus est la version de base mais peut sembler limité en terme d'applications. En effet, le problème Max-SAT basique ne permet pas de hiérarchiser les clauses entre elles pour représenter le fait que certaines clauses sont plus importantes à satisfaire que d'autres. Par conséquent il existe plusieurs

1. Le problème Max-2-SAT, où chaque clause contient exactement deux littéraux, est également NP-difficile [RRR98], alors que le problème 2-SAT est polynomial (et donc dans la classe P)

variantes plus évoluées basées sur l'ajout d'un poids à chaque clause. On parle alors de *formule pondérée.*

Définition 3.2 (Clause pondérée)**.** *Une clause pondérée est l'association d'une clause c à son poids $w > 0$ et on la note (c, w).*

Définition 3.3 (Clause dure)**.** *Une clause dure est une clause pondérée (c, w) de poids w infini.*

Définition 3.4 (Clause souple)**.** *Une clause souple est une clause pondérée (c, w) de poids w fini.*

Définition 3.5 (Formule pondérée)**.** *Une formule pondérée est une conjonction de clauses pondérées $\phi = (c_1, w_1) \wedge (c_2, w_2) \wedge ... \wedge (c_m, w_m)$ et peut-être aussi représentée comme un multi-ensemble de clauses pondérées.*

Définition 3.6 (Coût d'une formule)**.** *Soit ϕ une formule et I une interprétation des variables de ϕ. Le coût de l'interprétation I pour la formule ϕ, noté $cout_I(\phi)$, est la somme des poids des variables falsifiées :*

$$cout_I(\phi) = \sum_{(c,w)\in\phi : c \text{ falsifiée par } I} w \tag{3.1}$$

Parmi les variantes du problème Max-SAT, le problème Max-SAT partiel permet de garantir la satisfaction de certaines des clauses grâce à un partitionnement des clauses en deux sous-ensembles : les clauses dures et les clauses souples. Le problème Max-SAT pondéré permet de hiérarchiser les clauses entre elles et chaque clause est associé à un poids fini correspondant à la pénalité payée en cas de falsification de cette clause. Enfin, Le problème Max-SAT pondéré partiel cumule les deux précédentes variantes. Les clauses sont partitionnées en deux sous-ensembles (clauses dures et clauses souples) mais en plus les clauses souples ont toutes un poids fini. On dit aussi que les clauses dures ont un poids infini. Résoudre le problème Max-SAT pondéré partiel consiste donc à minimiser la somme des poids des clauses souples falsifiées, tout en satisfaisant les clauses dures.

Remarquons qu'il y a une relation d'inclusion entre certaines variantes :

— Le problème Max-SAT simple est le cas particulier du problème Max-SAT partiel où il n'y a pas de clauses dures.
— Le problème Max-SAT simple est le cas particulier du problème Max-SAT pondéré où toutes les clauses ont un poids de 1.
— Le problème Max-SAT simple est le cas particulier du problème Max-SAT pondéré partiel où il n'y a pas de clauses dures et où toutes les clauses souples ont un poids de 1.
— Le problème Max-SAT pondéré est le cas particulier du problème Max-SAT pondéré partiel où il n'y a pas de clauses dures.
— Le problème Max-SAT partiel est le cas particulier du problème Max-SAT pondéré partiel où toutes les clauses souples ont un poids de 1.

3.3. Règles d'inférences pour Max-SAT

Pour résoudre le problème Max-SAT, certaines techniques utilisent des transformation progressives d'une formule Max-SAT vers une autre formule Max-SAT. En particulier, certaines transformations sont intéressantes parce qu'elles préservent certaines propriétés de la formule Max-SAT transformée. En particulier, on dit qu'une transformation préserve l'équivalence Max-SAT si toute interprétation des variables falsifie le même nombre de clauses avant la transformation et après la transformation.

Définition 3.7 (Équivalence Max-SAT)**.** *Soit ϕ et ϕ' deux formules CNF, on dit que ϕ est équivalent (dans le sens de Max-SAT) à ϕ' et on note $\phi \equiv \phi'$ si pour toute interprétation des variables $I : var(\phi) \cup var(\phi') \rightarrow \{1,0\}$, on a $cout_I(\phi) = cout_I(\phi')$*

On présente ici quelques règles d'inférence utiles pour le problème Max-SAT. Comme l'équivalence Max-SAT préserve le nombre de clauses falsifiées et non le nombre de clauses satisfaites, on s'intéresse donc plutôt à connaître l'optimum Max-SAT en nombre de clauses falsifiées. Certaines règles d'inférence remplacent un ensemble de clauses initiales (ou prémisses) C_1 par un ensemble de nouvelles clauses C_2 et on les note sous la forme suivante. On fera bien attention au fait que les clauses prémisses sont supprimées après application de la règle, contrairement aux règles d'inférence pour le problème SAT.

$$\frac{C_1}{C_2}$$

Règle des littéraux purs La règle des littéraux purs utilisée dans le cadre du problème SAT, consistant à affecter positivement (respectivement négativement) toute variable n'apparaissant que positivement (respectivement négativement) peut être utilisée pour Max-SAT [BR99]. En effet, bien que cette transformation ne préserve pas l'équivalence Max-SAT, elle préserve la valeur de l'optimum Max-SAT et toute interprétation optimale après application de la règle des littéraux purs est également optimale avant cette même application.

Suppression des clauses tautologiques Les clauses tautologiques sont, par définition, toujours satisfaites quelque soit l'interprétation des variables. Elles sont donc généralement supprimées. Cela n'affecte ni l'optimum Max-SAT du point de vue du nombre de clauses falsifiées, ni la liste des interprétation des variables permettant d'atteindre l'optimum Max-SAT.

Clauses unitaires dominantes La règle des clauses unitaires dominantes consiste à affecter à vrai les littéraux pour lesquels la somme des poids des clauses unitaires qui contient ce littéral est supérieur à la somme des poids des clauses dans lesquelles son complémentaire apparaît [NR00].

Clauses unitaires complémentaires Si deux clauses unitaires opposées appartiennent à la formule, alors on peut remplacer ces deux clauses par une clause vide [NR00].

$$\frac{c_1 = x \quad c_2 = \overline{x}}{c_3 = \square}$$

Clauses presque identiques ou coupe symétrique (*symmetric cut*) Si deux clauses sont identiques à l'exception d'un littéral complémentaire, alors on peut remplacer ces deux clauses par une seule clause contenant la partie qui leur est commune [BR99].

$$\frac{c_1 = (x \vee A) \quad c_2 = (\overline{x} \vee A)}{c_3 = (A)}$$

Cette règle généralise la règle des clauses unitaires complémentaires. Dans la suite de ce document, on fera référence à cette règle en utilisant la terminologie anglophone : *symmetric cut* [BL20].

Max-résolution La règle de résolution [Rob65] utilisée dans le cadre du problème SAT étant invalide pour Max-SAT (car ne conservant pas l'équivalence Max-SAT), il a été proposé une adaptation de la règle de résolution pour le problème Max-SAT, appelé max-résolution [BLM06] [BLM07] [LH05] et décrite ci-dessous (avec $A = a_1 \vee ... \vee a_s$ et $B = b_1 \vee ... \vee b_t$).

$$\frac{c_1 = x \vee A \qquad\qquad c_2 = \overline{x} \vee B}{\begin{array}{c} c_3 = A \vee B \\ cc_1 = x \vee A \vee \overline{b_1} \\ cc_2 = x \vee A \vee b_1 \vee \overline{b_2} \\ \vdots \\ cc_t = x \vee A \vee b_1 \vee ... \vee b_{t-1} \vee \overline{b_t} \\ cc_{t+1} = \overline{x} \vee B \vee \overline{a_1} \\ cc_{t+2} = \overline{x} \vee B \vee a_1 \vee \overline{a_2} \\ \vdots \\ cc_{t+s} = \overline{x} \vee B \vee a_1 \vee ... \vee a_{s-1} \vee \overline{a_s} \end{array}}$$

On dit que c_3 est la clause résolvante (ou résultante), que $cc_1, ..., cc_{t+s}$ sont les clauses de compensation et que la variable x est la variable pivot de la max-résolution. Les clauses de compensation sont essentielles car elles permettent de conserver l'équivalence Max-SAT. Cette règle généralise la règle des clauses presque identiques.

Exemple 3.2. *Soient $c_1 = (x_1 \vee x_2)$ et $c_2 = (\overline{x_1} \vee \overline{x_3})$. L'application de la max-résolution sur c_1 et c_2 permet de remplacer ces deux clauses par les trois clauses $c_3 = (x_2 \vee \overline{x_3})$, $cc_1 = (x_1 \vee x_2 \vee x_3)$ et $cc_2 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.*

Scission d'une clause (*split*) Il est possible de transformer n'importe quelle clause pour lui ajouter une variable. On remplace alors la clause initiale par deux clauses distinctes auxquelles on a ajouté un littéral.

$$\frac{c_3 = (A)}{c_1 = (x \vee A) \quad c_2 = (\overline{x} \vee A)}$$

Cette règle est la réciproque de la règle des clauses presque identiques. Dans la suite de ce document, on fera référence à cette règle en utilisant la terminologie anglophone utilisée dans la littérature : *split* [LR20b ; BL20].

3.4. Résolution du problème Max-SAT

Dans cette section, on s'intéresse aux principales méthodes utilisées pour résoudre en pratique le problème Max-SAT. La recherche concernant la résolution efficace du problème Max-SAT a connu de nombreuses améliorations ces quinze dernières années, grâce notamment à l'organisation des évaluations Max-SAT [Arg+08].

3.4.1. Résolution par appels itératifs à un oracle SAT

3.4.1.1. Transformation en problème de décision

Résoudre le problème Max-SAT revient à trouver un entier k tel qu'il est impossible de trouver une solution qui falsifie strictement moins de k clauses mais qu'il existe une solution falsifiant exactement k clauses. Dans sa version pondérée, on cherche à trouver un entier k tel qu'il soit impossible de falsifier strictement moins qu'un poids total de k mais qu'il existe une solution falsifiant un poids total de k. Ainsi, on peut résoudre le problème Max-SAT en résolvant itérativement le problème de décision suivant (en faisant progressivement varier la valeur de k) : "Est-il possible de trouver une solution qui falsifie au plus un poids total de k?"[FM06][BP10][Kos+12]. La question ici posée peut être codée sous forme normale conjonctive par une contrainte de cardinalité [2], ce qui permet de rendre possible de répondre à la question grâce à l'appel à un oracle SAT.

Les algorithmes utilisant ce principe diffèrent ensuite sur la manière de faire varier la valeur k, par exemple :

— Parcours linéaire [FM06 ; BP10 ; Kos+12] : on fait varier k de $\sum_{c\in\phi} w_i$ à 0, ou on fait au contraire varier k de 0 à $\sum_{c\in\phi} w_i$, en s'arrêtant quand on a trouvé l'entier k souhaité. Le pas peut être fixé à 1 ou fixé itérativement en fonction du résultat de l'appel à l'oracle SAT.

— Parcours binaire [FM06 ; Kos+12] : on fixe les bornes de l'intervalle possible pour k à $[0, \sum_{c\in\phi} w_i]$, puis on fixe itérativement k à la médiane de l'intervalle et on met à jour l'une des bornes suivant que l'on soit tombé sur un problème de

2. Une contrainte de cardinalité est une contrainte (ici un ensemble de clauses) permettant de modéliser le fait qu'au plus k littéraux d'un ensemble peuvent être satisfaits

décision satisfiable ou non. On s'arrête lorsque l'intervalle contient une unique valeur qui est la solution optimale.
— Parcours hybride [Kos+12] : On alterne entre les parcours linéaires et le parcours binaire.

Parmi les solveurs utilisant cette méthode, on peut citer le solveur QMaxSAT [Kos+12] ou le solveur Pacose [PRB18].

Exemple 3.3 (parcours linéaire décroissant)**.** *On considère la formule suivante*[3] *:*

$$\begin{aligned}\phi = &(x_1 \vee \overline{x_2}) \wedge (\overline{x_1}) \wedge (x_2 \vee x_4) \wedge (x_2 \vee x_6) \wedge (x_2 \vee \overline{x_6}) \wedge (x_3 \vee \overline{x_5}) \wedge \\ &(\overline{x_3}) \wedge (\overline{x_4} \vee x_5) \wedge (x_5 \vee x_7) \wedge (x_5 \vee \overline{x_7}) \wedge (x_6 \vee \overline{x_8}) \wedge (\overline{x_6} \vee x_8)\end{aligned} \tag{3.2}$$

On ajoute à la formule ci-dessous un ensemble de clauses permettant de modéliser le fait d'être autorisé à falsifier au plus 12 *clauses de la formule. Pour cela, on rajoute à chaque clause une variable auxiliaire* r_i *et on ajoute ensuite un ensemble de clauses pour encoder la contrainte (appelée contrainte de cardinalité) sur le nombre de clauses pouvant être falsifiées.*

$$\begin{aligned}\phi = &(x_1 \vee \overline{x_2} \vee r_1) \wedge (\overline{x_1} \vee r_2) \wedge (x_2 \vee x_4 \vee r_3) \wedge (x_2 \vee x_6 \vee r_4) \wedge \\ &(x_2 \vee \overline{x_6} \vee r_5) \wedge (x_3 \vee \overline{x_5} \vee r_6) \wedge (\overline{x_3} \vee r_7) \wedge (\overline{x_4} \vee x_5 \vee r_8) \wedge \\ &(x_5 \vee x_7 \vee r_9) \wedge (x_5 \vee \overline{x_7} \vee r_{10}) \wedge (x_6 \vee \overline{x_8} \vee r_{11}) \wedge (\overline{x_6} \vee x_8 \vee r_{12}) \wedge \\ &CNF(\sum_{i=1}^{12} r_i \leq 12)\end{aligned} \tag{3.3}$$

1. *Première itération*
   a) *On lance l'oracle SAT et il retourne que la formule est satisfiable. Dans le modèle proposé, trois variables auxiliaires* r_1, r_7 *et* r_9 *sont affectées à vrai, ce qui signifie qu'au plus trois clauses de la formule initiale ont été falsifiées.*
   b) *On modifie la contrainte de cardinalité afin d'autoriser la falsification d'au plus* 2 *clauses, ce qui revient à la remplacer par* $CNF(\sum_{i=1}^{12} r_i \leq 2)$.
2. *Deuxième itération*
   a) *On lance l'oracle SAT et il retourne que la formule est satisfiable. Dans le modèle proposé, deux variables auxiliaires sont affectées à vrai, ce qui signifie qu'au plus deux clauses de la formule initiale ont été falsifiées.*
   b) *On modifie la contrainte de cardinalité afin d'autoriser la falsification d'au plus* 1 *clause, ce qui revient à la remplacer par* $CNF(\sum_{i=1}^{12} r_i \leq 1)$.
3. *Troisième itération*
   a) *On lance l'oracle SAT et il retourne que la formule est insatisfiable. Il n'est par conséquent pas possible de trouver une interprétation des variables permettant de falsifier au plus* 1 *clause.*

3. tirée de [MDM14a]

4. *L'algorithme est terminé, l'optimum Max-SAT de la formule est donc de* 2 *clauses à falsifier. L'interprétation des variables lors du dernier retour positif de l'oracle SAT donne une solution optimale pour le problème Max-SAT.*

3.4.1.2. Guidage par les cœurs inconsistants

Une approche pour résoudre le problème Max-SAT, proche de la méthode précédente, consiste à encoder les contraintes de cardinalités uniquement sur les cœurs inconsistants. En effet, les solveurs SAT modernes sont désormais capables de fournir un cœur inconsistant lorsque la formule est insatisfiable, c'est à dire un sous-ensemble de clauses de la formule qui suffit à rendre celle-ci insatisfiable. Étant donnée une formule Max-SAT, une méthode consiste alors à itérativement appeler un oracle SAT pour obtenir un cœur inconsistant et à modifier la formule courante pour permettre à l'oracle SAT de falsifier une clause de ce cœur inconsistant. Cette modification de la formule consiste en l'ajout d'une contrainte de cardinalité, le plus souvent en utilisant des variables de relaxation. Cette méthode a d'abord été proposée par Fu and Malik dans l'algorithme msu1 [FM06] pour le problème Max-SAT Partiel, et nous en résumons l'idée via l'algorithme ci-dessous dans le cas simple du problème Max-SAT (non partiel et non pondéré).

Algorithme 3.1 Algorithme Fu and Malik

Entrée : Formule CNF ϕ
Sortie : L'optimum Max-SAT de ϕ

1: **tant que** $SAT(\phi)$ == INSATISFIABLE **faire**
2: $K \leftarrow calculer\_coeur\_inconsistant(\phi)$
3: **pour tout** clause $c_i \in K$ **faire**
4: Créer une nouvelle variable b_i
5: $\phi \leftarrow (\phi - c_i) \wedge (c_i \vee b_i)$
6: $\phi \leftarrow \phi \wedge CNF(\sum_i b_i \leq 1)$
7: $I \leftarrow calculer\_modele(\phi)$
8: **retourner** $cout_I(\phi)$

Exemple 3.4. *On considère la formule suivante*[4] *:*

$$\begin{aligned}\phi = &(x_1 \vee \overline{x_2}) \wedge (\overline{x_1}) \wedge (x_2 \vee x_4) \wedge (x_2 \vee x_6) \wedge (x_2 \vee \overline{x_6}) \wedge (x_3 \vee \overline{x_5}) \wedge \\ &(\overline{x_3}) \wedge (\overline{x_4} \vee x_5) \wedge (x_5 \vee x_7) \wedge (x_5 \vee \overline{x_7}) \wedge (x_6 \vee \overline{x_8}) \wedge (\overline{x_6} \vee x_8)\end{aligned} \tag{3.4}$$

Pour les besoins de l'exemple, on représente la formule sous la forme suivante :

4. tirée de [MDM14a]

$(x_2 \vee x_6)$	$(x_2 \vee \overline{x_6})$	$(x_1 \vee \overline{x_2})$	$(\overline{x_1})$
$(x_6 \vee \overline{x_8})$	$(\overline{x_6} \vee x_8)$	$(x_2 \vee x_4)$	$(\overline{x_4} \vee x_5)$
$(x_5 \vee x_7)$	$(x_5 \vee \overline{x_7})$	$(x_3 \vee \overline{x_5})$	$(\overline{x_3})$

Première itération : on lance l'oracle SAT et il retourne que la formule est insatisfiable avec le cœur inconsistant $K_1 = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1}) \wedge (x_2 \vee x_6) \wedge (x_2 \vee \overline{x_6})$ *. On modifie donc ces clauses en ajoutant quatre variables au modèle ainsi qu'une contrainte de cardinalité.*

$(x_2 \vee x_6 \vee b_1)$	$(x_2 \vee \overline{x_6} \vee b_2)$	$(x_1 \vee \overline{x_2} \vee b_3)$	$(\overline{x_1} \vee b_4)$
$(x_6 \vee \overline{x_8})$	$(\overline{x_6} \vee x_8)$	$(x_2 \vee x_4)$	$(\overline{x_4} \vee x_5)$
$(x_5 \vee x_7)$	$(x_5 \vee \overline{x_7})$	$(x_3 \vee \overline{x_5})$	$(\overline{x_3})$
$CNF(\sum_{i=1}^{4} b_i \leq 1)$			

Deuxième itération : on lance l'oracle SAT, il retourne que la formule est insatisfiable avec le cœur inconsistant $K_2 = (\overline{x_3}) \wedge (x_3 \vee \overline{x_5}) \wedge (x_5 \vee x_7) \wedge (x_5 \vee \overline{x_7})$ *. On modifie donc ces clauses en ajoutant quatre variables au modèle ainsi qu'une contrainte de cardinalité.*

$(x_2 \vee x_6 \vee b_1)$	$(x_2 \vee \overline{x_6} \vee b_2)$	$(x_1 \vee \overline{x_2} \vee b_3)$	$(\overline{x_1} \vee b_4)$
$(x_6 \vee \overline{x_8})$	$(\overline{x_6} \vee x_8)$	$(x_2 \vee x_4)$	$(\overline{x_4} \vee x_5)$
$(x_5 \vee x_7 \vee b_5)$	$(x_5 \vee \overline{x_7} \vee b_6)$	$(x_3 \vee \overline{x_5} \vee b_7)$	$(\overline{x_3} \vee b_8)$
$CNF(\sum_{i=1}^{4} b_i \leq 1)$		$CNF(\sum_{i=5}^{8} b_i \leq 1)$	

Troisième itération : on lance l'oracle SAT qui retourne que la formule est satisfiable avec l'interprétation $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 1, x_7 = 0, x_8 = 1, b_1 = 0, b_2 =$

$0, b_3 = 0, b_4 = 1, b_5 = 1, b_6 = 0, b_7 = 0, b_8 = 0$.

L'algorithme est terminé, l'optimum Max-SAT de la formule est donc 2 *et une solution est présentée dans le modèle ci-dessus qui falsifie les clauses* $(\overline{x_1})$ *et* $(x_5 \vee x_7)$.

Ensuite, des améliorations et extensions ont été proposées parmi lesquelles msu2 (aussi appelé msu1.1) [MP07], msu3 [MP07], msu4 [MP08], msu1.2 [MM08], PM2 [ABL09b][ABL09a], OLL [MDM14b], wmsu1 [MMP09], WPM1 [ABL09b], WPM2 [ABL10], BCD [HMM11], BCD2 [HMM12], etc.

Certaines techniques de partitionnement de la formule ont notamment été utilisées pour améliorer les performances en pratique des algorithmes basés sur les cœurs inconsistants sur les instances pondérées. Pour cela, on partitionne les clauses selon leur poids et on considère d'abord les clauses de poids important pour progressivement y insérer les clauses de poids inférieur [MML12] [Ans+12] [ABL13].

Parmi les solveurs basés sur des appels itératifs à un oracle SAT pour détecter et transformer des cœurs inconsistants, on peut citer RC2 [IMM19], Open-WBO [MML14] ou EVA [NB14]. Notons que le solveur EVA utilise la transformation par max-résolution lors du traitement des cœurs inconsistants. Plus de détails sur les algorithmes de résolution par appels itératifs à un oracle SAT peuvent être trouvés dans [Mor+13].

3.4.2. Résolution par appels itératifs à un oracle ILP

3.4.2.1. Transformation en problème ILP

Une méthode de résolution consiste à modéliser le problème Max-SAT comme un problème de programmation linéaire en nombres entiers (PLNE ou en anglais ILP pour *Integer Linear Programming*). Parmi les modélisations possibles pour le problème Max-SAT, on peut citer notamment une modélisation présentée dans [MMP09] et implémentée dans [AG13] :

1. Les variables de décisions sont :
   — Pour toute variable x_j de la formule ϕ, on a une variable x_j dans le programme ILP qui vaut 1 si et seulement si on satisfait la variable x_j
   — Pour toute clause souple c_i de la formule ϕ, on a une variable f_i dans le programme ILP qui vaut 1 si et seulement si on falsifie la clause c_i
2. Les contraintes sont :
   — Au moins un littéral de chaque clause dure doit être affecté à vrai
   — Au moins un littéral de chaque clause falsifiée doit être affecté à vrai ou la clause souple doit être déclarée falsifiée
3. L'objectif du programme ILP est de minimiser la somme pondérée des clauses souples falsifiées

On obtient le programme linéaire en nombre entiers suivant, où $CD(\phi)$ et $CS(\phi)$ sont respectivement l'ensemble des clauses dures et des clauses souples de ϕ :

$$
\begin{array}{lrcll}
\text{Minimiser} & \sum_{c_i \in CS(\phi)} w_i \times f_i & & & \\
\text{s.c.} & \sum_{l_j \in c_i} l_i & >= & 1 & \forall c_i \in CD(\phi) \\
& f_i + \sum_{x_j \in c_i} x_j + \sum_{\overline{x_j} \in c_i} (1 - x_j) & >= & 1 & \forall c_i \in CS(\phi) \\
& f_i & \in & \{0,1\} & \forall c_i \in \phi \\
& x_j & \in & \{0,1\} & \forall x_j \in var(\phi)
\end{array}
$$

3.4.2.2. Résolution itérative du problème d'ensemble couvrant minimal

Le problème Max-SAT est à l'intersection de deux domaines de l'informatique, qui sont la programmation par contraintes, qui traite notamment du problème SAT, et la recherche opérationnelle, qui étudie les problèmes d'optimisation. L'idée est alors apparue de séparer la résolution du problème Max-SAT en deux composantes correspondant à ces deux domaines. Cette idée a été exploitée dans le solveur MaxHS [DB11 ; Dav13].

Plus concrètement, l'algorithme est construit sur deux composantes. La première composante, la partie optimisation, se charge, étant donné un ensemble de cœurs inconsistants, de choisir quelles clauses falsifier de manière à falsifier le minimum de clauses et à couvrir l'ensemble des cœurs inconsistants (il s'agit de de résoudre le problème du *Maximum Hitting Set* [DB11]). La seconde composante, la partie décisionnelle, se charge, étant donné une couverture optimale de l'ensemble des cœurs inconsistants connus, de vérifier si la solution est bien réalisable, en vérifiant par exemple qu'il n'existe pas de cœur inconsistant non encore connu qui empêche la réalisabilité de la couverture proposée. Dans plusieurs versions du solveur MaxHS, la composante "optimisation" est basée sur l'appel à l'oracle ILP CPLEX [IBM09] alors que la composante "décision" est basée sur l'appel à l'oracle SAT Minisat [ES03]. On présente ci-dessous une version basique de l'algorithme utilisé par MaxHS ainsi qu'un exemple de son fonctionnement.

Algorithme 3.2 Algorithme MaxHS basique

Entrée : Formule CNF ϕ
Sortie : Une solution optimale s au problème Max-SAT

```
1: K ← ∅
2: tant que 1 faire
3:    hs ← calculer_couverture_coeurs(K)
4:    (sat, s, k) ← verifier_realisabilite_couverture(φ, K)
5:    si sat == SATISFIABLE alors
6:       retourner s
7:    sinon
8:       K ← K ∪ {k}
```

Exemple 3.5. *On considère la formule suivante*[5] *:*

$$\begin{aligned}\phi = &(x_1 \vee \overline{x_2}) \wedge (\overline{x_1}) \wedge (x_2 \vee x_4) \wedge (x_2 \vee x_6) \wedge (x_2 \vee \overline{x_6}) \wedge (x_3 \vee \overline{x_5}) \wedge \\ &(\overline{x_3}) \wedge (\overline{x_4} \vee x_5) \wedge (x_5 \vee x_7) \wedge (x_5 \vee \overline{x_7}) \wedge (x_6 \vee \overline{x_8}) \wedge (\overline{x_6} \vee x_8)\end{aligned} \tag{3.5}$$

Pour les besoins de l'exemple, on va représenter la formule sous la forme suivante :

$(x_2 \vee x_6)$	$(x_2 \vee \overline{x_6})$	$(x_1 \vee \overline{x_2})$	$(\overline{x_1})$
$(x_6 \vee \overline{x_8})$	$(\overline{x_6} \vee x_8)$	$(x_2 \vee x_4)$	$(\overline{x_4} \vee x_5)$
$(x_5 \vee x_7)$	$(x_5 \vee \overline{x_7})$	$(x_3 \vee \overline{x_5})$	$(\overline{x_3})$

Première itération ($K = \emptyset$) :

a) On lance l'oracle ILP en ne lui fournissant aucun cœur inconsistant. Il en conclut qu'il est possible de couvrir tous les cœurs inconsistants en ne falsifiant aucune clause.

b) On lance l'oracle SAT en laissant toutes les clauses et il retourne que la formule est insatisfiable en nous retournant le cœur inconsistant $K_1 = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1}) \wedge (x_2 \vee x_6) \wedge (x_2 \vee \overline{x_6})$

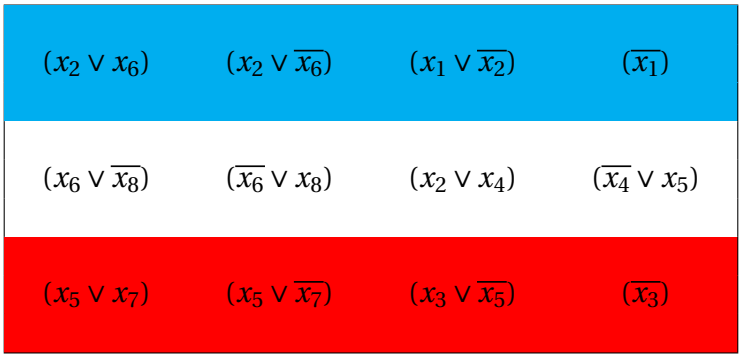
$(x_2 \vee x_6)$	$(x_2 \vee \overline{x_6})$	$(x_1 \vee \overline{x_2})$	$(\overline{x_1})$
$(x_6 \vee \overline{x_8})$	$(\overline{x_6} \vee x_8)$	$(x_2 \vee x_4)$	$(\overline{x_4} \vee x_5)$
$(x_5 \vee x_7)$	$(x_5 \vee \overline{x_7})$	$(x_3 \vee \overline{x_5})$	$(\overline{x_3})$

Deuxième itération ($K = \{K_1\}$) :

a) On lance l'oracle ILP en lui fournissant le cœur inconsistant K_1*. Il en conclut qu'il est possible de couvrir tous les cœurs inconsistants en falsifiant une seule clause, par exemple la clause* $(x_2 \vee x_6)$

b) On lance l'oracle SAT en ayant enlevé la clause $(x_2 \vee x_6)$ *de la formule, il retourne que la formule est insatisfiable en nous retournant le cœur inconsistant* $K_2 = (\overline{x_3}) \wedge (x_3 \vee \overline{x_5}) \wedge (x_5 \vee x_7) \wedge (x_5 \vee \overline{x_7})$

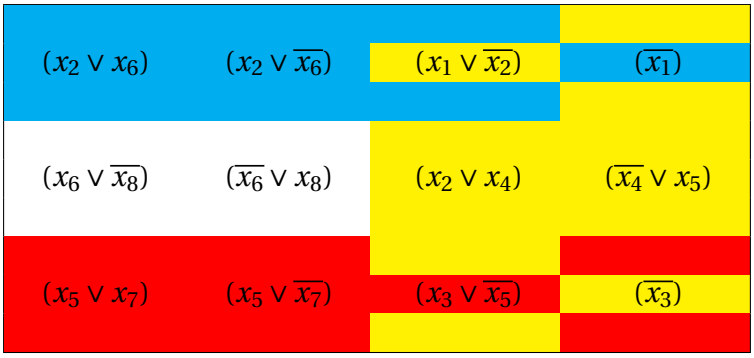
5. tirée de [MDM14a]



Troisième itération ($K = \{K_1, K_2\}$) :

a) On lance l'oracle ILP en lui fournissant les cœurs inconsistants K_1 *et* K_2*. Il en conclut qu'il est possible de couvrir tous les cœurs inconsistants en falsifiant deux clauses, par exemple les clauses* $(x_2 \vee x_6)$ *et* $(x_5 \vee x_7)$

b) On lance l'oracle SAT en ayant enlevé les clauses $(x_2 \vee x_6)$ *et* $(x_5 \vee x_7)$ *de la formule, il retourne que la formule est insatisfiable en nous retournant le cœur inconsistant* $K_3 = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1}) \wedge (\overline{x_3}) \wedge (x_3 \vee \overline{x_5})$



Quatrième itération ($K = \{K_1, K_2, K_3\}$) :

a) On lance l'oracle ILP en lui fournissant les cœurs inconsistants K_1, K_2 *et* K_3*. Il en conclut qu'il est possible de couvrir tous les cœurs inconsistants en falsifiant deux clauses, par exemple les clauses* $(\overline{x_1})$ *et* $(x_5 \vee x_7)$

b) On lance l'oracle SAT en ayant enlevé les clauses $(\overline{x_1})$ *et* $(x_5 \vee x_7)$ *de la formule, il retourne que la formule est satisfiable avec l'interprétation* $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 1, x_7 = 0, x_8 = 1$

L'algorithme est terminé, l'optimum Max-SAT de la formule est donc de 2 *clauses à falsifier et une solution est présentée dans le modèle ci-dessus qui falsifie les clauses* $(\overline{x_1})$ *et* $(x_5 \vee x_7)$.

L'algorithme MaxHS a ensuite été progressivement amélioré, surtout sur sa composante d'optimisation, qui est la plus coûteuse en temps de calcul. Tout d'abord, la modélisation du problème d'optimisation a été améliorée pour tenir compte de certaines contraintes de réalisabilité (par exemple, on ne peut pas choisir deux clauses

qui s'opposent) [DB11]. Ensuite, le problème d'optimisation a été modélisé plus finement utilisant des variables auxiliaires similaires à celles utilisées dans les solveurs itératifs et une phase disjointe précédant l'exécution de l'algorithme a été ajoutée afin de débuter l'algorithme avec un problème d'optimisation non-vide [DB13a]. Afin d'éviter de trop utiliser le solveur ILP, une phase de résolution non optimale à base d'algorithmes gloutons à été intégré lors de la détection de nouveaux cœurs inconsistants [DB13b]. Des notions venant de recherche opérationnelle comme les coûts réduits ont aussi été utilisés pour améliorer les performances du solveur ILP [Bac+17]. Enfin, lorsque le solveur ILP retourne une solution irréalisable, MaxHS a été amélioré afin d'en déduire plusieurs cœurs inconsistants [BBP20].

Concernant la partie décisionnelle, notons l'existence de travaux d'améliorations sur l'implémentation du solveur SAT afin d'améliorer les résultats de MaxHS [HB19]. Les travaux sur MaxHS ont aussi donné naissance au solveur LMHS [SBJ16; Bac+17; BBP20].

3.4.3. Résolution par algorithme de type séparation et d'évaluation

Un algorithme générique pour résoudre des problèmes d'optimisation est l'algorithme de type séparation et d'évaluation (en anglais *Branch and Bound*). Pour résoudre Max-SAT, cet algorithme explore l'arbre de recherche associé à la formule, arbre de recherche identique à celui considéré pour l'algorithme DPLL utilisé pour résoudre SAT. À chaque nœud de l'arbre de recherche, l'algorithme compare la meilleure solution trouvée (la borne supérieure) avec une sous-estimation du coût de la meilleure solution atteignable dans la branche actuelle de l'arbre de recherche (la borne inférieure). Si on se retrouve dans un nœud de l'arbre de recherche avec la borne inférieure qui est plus grande que la borne supérieure, alors il sera impossible d'améliorer la solution courante dans cette portion de l'arbre de recherche et on peut en stopper l'exploration et parcourir une autre portion de l'arbre de recherche. Lorsque l'intégralité de l'arbre de recherche a été explorée, la solution courante obtenue est une solution optimale pour le problème Max-SAT. Parmi les principaux solveurs Max-SAT fonctionnant sur ce principe, on peut notamment citer akmaxsat [Küg10], MaxSatz [LMP07; LMP05; LMP06; Dar+07; Li+08; LSL08; Li+09; Li+10; Liu+16], ahmaxsat [AH15a; AH14d; AH14a; AH14b; AH16; AH14c; Abr15; AH15b], ou encore MiniMaxSAT [HLO07; HLO08]. On peut représenter de manière générique les algorithmes de type séparation et évaluation pour Max-SAT par l'algorithme 3.3.

Algorithme 3.3 Algorithme de type séparation et évaluation pour Max-SAT

Entrée : Une formule CNF ϕ contenant n variables et une borne supérieure UB sur l'optimum Max-SAT de la formule
Sortie : L'optimum Max-SAT de ϕ

```
1: φ ← simplifier_formule(φ)
2: si φ ne contient que des clauses vides alors
3:    retourner nombre_clauses_vides(φ)
4: LB ← calculer_borne_inferieure(φ)
5: si LB ≥ UB alors
6:    retourner UB
7: Choisir une variable x ∈ φ
8: UB ← min(UB, MaxSAT(φ_|x, UB))
9: retourner min(UB, MaxSAT(φ_|x̄, UB))
```

Exemple 3.6. *Considérons la formule* $\phi = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \wedge c_7 \wedge c_8 \wedge c_9 \wedge c_{10}$ *avec* $c_1 = (x_1)$, $c_2 = (\overline{x_1})$, $c_3 = (\overline{x_2})$, $c_4 = (\overline{x_1} \vee x_2)$, $c_5 = (\overline{x_1} \vee \overline{x_3})$, $c_6 = (\overline{x_2} \vee \overline{x_3})$, $c_7 = (\overline{x_4} \vee \overline{x_5})$, $c_8 = (x_4 \vee x_5)$, $c_9 = (x_4 \vee \overline{x_5})$, $c_{10} = (\overline{x_1} \vee x_3 \vee \overline{x_5})$. *L'application de l'algorithme séparation et évaluation sur la formule* ϕ*, en sélectionnant les variables dans l'ordre lexicographique et en appliquant la propagation unitaire lorsque* $LB = UB - 1$ *est représentée par la figure 3.1.*

Au début de l'algorithme, les solveurs calculent une borne supérieure sur le nombre de clauses que la solution optimale devra falsifier. Cette valeur peut être initialisée avec une valeur inatteignable (comme le nombre de clauses plus un) ou en cherchant une solution initiale, par exemple à l'aide d'un algorithme incomplet [6].

Le calcul de la borne inférieure est une étape clé des solveurs de type séparation et évaluation et il faut trouver un compromis entre un calcul long mais qui donnera une borne inférieure plus fine ou un calcul plus court avec une borne inférieure moins précise. La méthode la plus rapide consiste simplement à compter le nombre de clauses actuellement falsifiées [BF98]. On peut améliorer ce calcul en comptant le nombre de clauses unitaires complémentaires [WF93], en détectant des motifs comme l'étoile (*star rule*) [AMP04], en simulant la propagation unitaire pour détecter d'autres cœurs inconsistants [LMP05 ; LMP06 ; Dar+07 ; LSL08], etc.

Certaines règle d'inférence sont appliquées, soit pour simplifier la formule pendant l'exploration de la branche courante, soit pour modifier définitivement la formule pendant le reste de l'exploration. Ces modifications peuvent être la transformation de motifs comme les *Unit Clause Subsets* [AH14d ; CHA20] ou l'application de la max-résolution sur les cœurs inconsistants ne correspondant à aucun motifs mais détectés par la propagation unitaire simulée [AH14b].

6. Un algorithme incomplet essaie de trouver la meilleure solution possible en un temps limité.

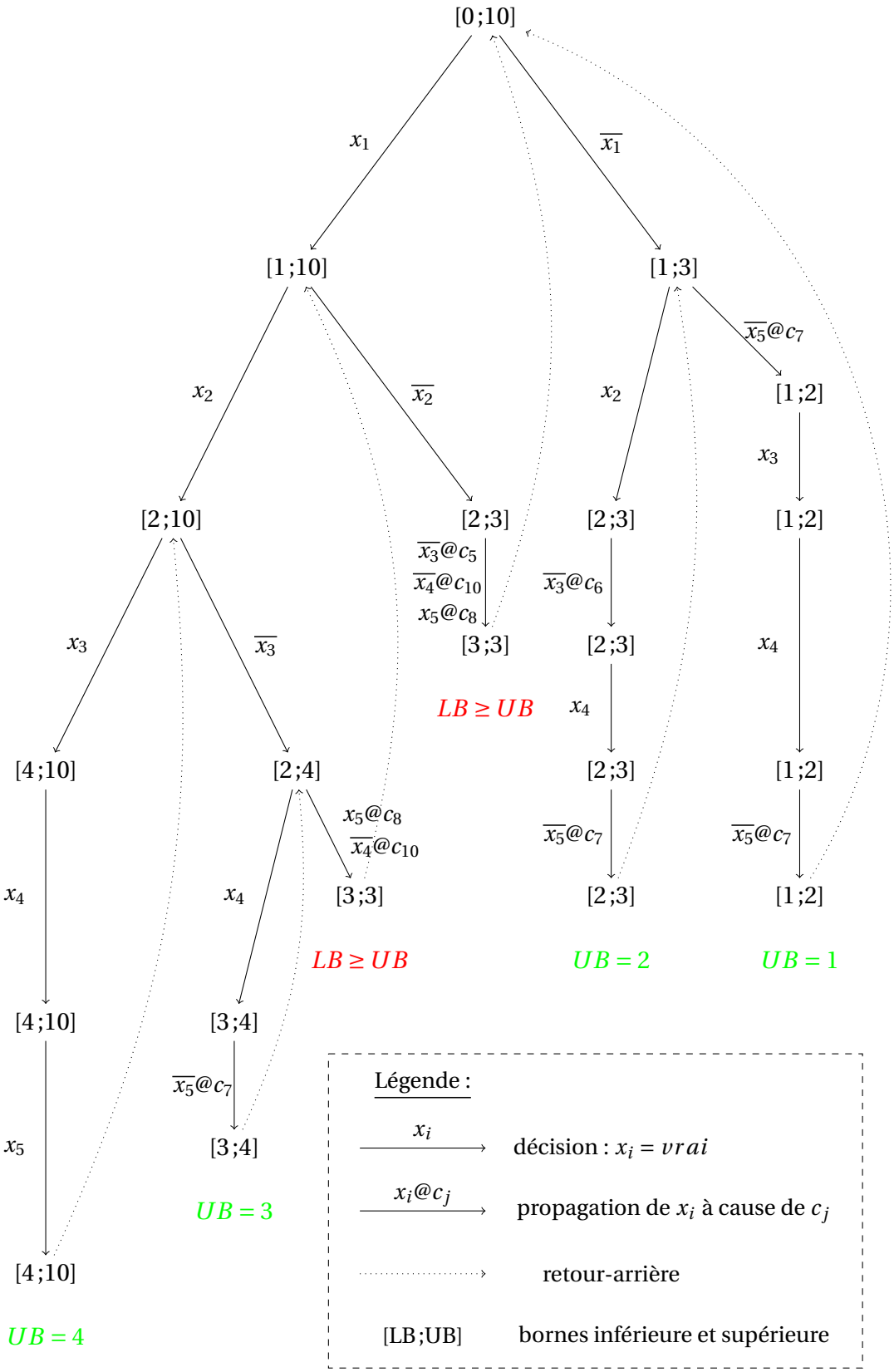


FIGURE 3.1. – Application d'un algorithme de type séparation et évaluation [Abr15]

3.5. Systèmes de preuve pour Max-SAT

De manière similaire au problème SAT, on peut se poser la question, pour Max-SAT, de ce qui pourrait être un certificat d'optimalité, c'est à dire une preuve que la valeur donnée en réponse au problème Max-SAT est bien la bonne. Cependant, contrairement au problème SAT, la question d'un certificat pour le problème Max-SAT a été très peu étudiée dans la littérature. Ainsi, plutôt que de s'intéresser aux certificats pour Max-SAT, que l'on étudiera dans les contributions du chapitre 6, on va s'intéresser aux éléments théoriques étudiés qui sont proches de cette notion de certificats : les systèmes de preuve. On s'intéressera en particulier aux systèmes de preuve en tant qu'ensembles de règles d'inférence pour Max-SAT et en particulier ceux utilisant la max-résolution, bien que d'autres travaux existent comme le *tableau calculus* [LMS16], le *SubCubeSums* [Fil+20], le système de preuves circulaires [AL19] ou encore le *Dual-Rail* [Bon+18 ; IMM17].

Définition 3.8 (Système de preuve)**.** *Un système de preuve pour Max-SAT est un ensemble de règles d'inférence qui respectent l'équivalence Max-SAT.*

3.5.1. La max-résolution

Lorsque la max-résolution a été introduite, il a été démontré qu'elle formait un système de preuve complet pour le problème Max-SAT, c'est à dire qu'il était possible, en utilisant uniquement la transformation par max-résolution, de transformer une formule ϕ en une formule équivalente $\phi_2 = \underbrace{\square \wedge \cdots \wedge \square}_{opt(\phi)} \wedge \phi'$ avec ϕ' satisfiable [BLM06]. Cela est possible en utilisant l'algorithme de saturation. Celui-ci consiste à saturer successivement chaque variable x, c'est à dire appliquer la max-résolution sur chaque paire de clauses opposées sur une variable x jusqu'à ce que toutes les clauses contenant x soient opposées deux à deux sur une autre variable. En appliquant l'algorithme de saturation, il est garanti de pouvoir démontrer l'optimum du problème Max-SAT en un nombre d'étapes d'inférence borné par $n \times m \times 2^n$ où n et m sont respectivement le nombre de variables et le nombre de clauses de la formule. L'algorithme de saturation est principalement intéressant pour son résultat théorique, même s'il est marginalement utilisé dans le solveur MaxSATz [ALM08].

Exemple 3.7. *Soit $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$, l'application de l'algorithme de saturation dans l'ordre lexicographique permet de démontrer que $\phi \equiv \square \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$ et la preuve ainsi produite est représentée dans la Figure 3.2.*

3.5.2. Complétude de la max-résolution pour l'adaptation des réfutations par résolution *read-once* pour Max-SAT

Parmi les certificats d'inconsistance d'une formule propositionnelle, la réfutation par résolution est intéressante du point de vue du problème Max-SAT. En effet, dans la mesure où une réfutation par résolution permet de déduire, en utilisant des règles

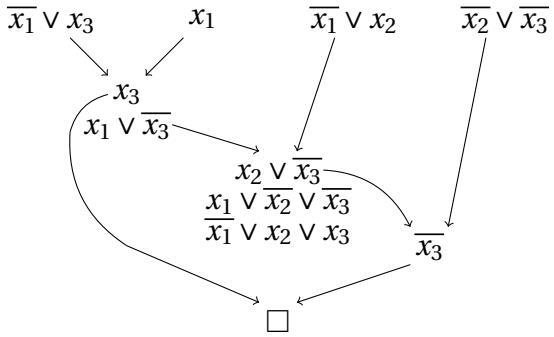


FIGURE 3.2. – Transformation d'une formule par saturation

d'inférence qui préservent l'équivalence SAT, une clause vide, il est intéressant de se demander si on pourrait utiliser les réfutations par résolution dans le cadre du problème Max-SAT, pour produire des certificats partiels pour le problème SAT permettant de transformer une formule en un équivalent contenant une clause vide supplémentaire. Ainsi, un ensemble de k réfutations préservant l'équivalence Max-SAT permettrait de certifier que l'optimum Max-SAT d'une formule est au moins de k, et il suffirait alors d'exhiber en plus une interprétation permettant de falsifier k clauses pour certifier que l'optimum Max-SAT d'une formule est bien k. On définit ci-dessous une réfutation utilisant uniquement des règles valides pour Max-SAT, sous le terme *max-réfutation*.

Définition 3.9 (Max-réfutation)**.** *Soit ϕ une formule. On appelle max-réfutation une séquence de transformations qui préservent l'équivalence Max-SAT, partant des clauses de la formule et déduisant de nouvelles clauses jusqu'à déduire ultimement $\square$.*

L'idée d'utiliser les réfutations par résolution et de les adapter en max-réfutations a déjà été utilisée, pour la résolution pratique du problème Max-SAT, dans le cas particulier des réfutations par résolution *read-once*. En effet, lorsque la réfutation par résolution est *read-once*, il est possible d'en déduire une max-réfutation en remplaçant chaque résolution par une max-résolution, ce qui permet ainsi de transformer le cœur inconsistant pour en extraire une clause vide [HM11].

Exemple 3.8. *La preuve par résolution présente en figure 3.3 est read-once, il est possible de l'adapter en une transformation par max-résolution du cœur inconsistant comme en figure 3.4.*

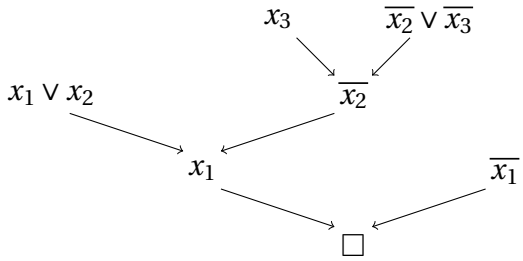


FIGURE 3.3. – Réfutation par résolution read-once

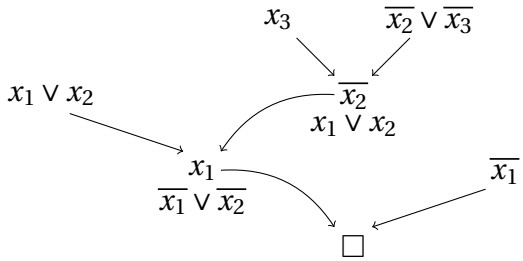


FIGURE 3.4. – Adaptation en max-réfutation

Actuellement, cette méthode est incomplète, dans le sens où elle n'est applicable que lorsque la réfutation par résolution est *read-once*. Dans le cas contraire, on ignore comment adapter une réfutation par résolution lorsqu'elle n'est pas *read-once* pour déduire une max-réfutation. L'une des contributions de cette thèse concerne justement l'adaptation des réfutations par résolution en max-réfutations, même dans les cas non-*read-once*, et deux méthodes sont présentées dans les chapitres 4 et 5. En répondant à cette question théorique, on verra ensuite dans le chapitre 6 qu'il est désormais possible de générer des certificats pour le problème Max-SAT en utilisant les réfutations par résolution : une séquence de k max-réfutations plus une interprétation falsifiant exactement k clauses formeront un certificat pour le problème Max-SAT permettant de démontrer que l'optimum Max-SAT de la formule est k.

3.5.3. Incomplétude de la max-résolution pour l'inférence Max-SAT

Si la max-résolution a été démontrée complète pour le problème Max-SAT [BLM06], elle n'est pas inférentiellement complète, c'est à dire que, étant donnée une clause c telle que $\phi \equiv c \wedge \phi'$ pour une certaine formule ϕ', la construction de la transformation permettant de passer de ϕ à son équivalent $c \wedge \phi'$ n'est pas toujours possible en utilisant uniquement la max-résolution (un exemple simple sert de preuve à la proposition 3.1). Cette limite de la max-résolution ainsi que des solutions pour les

dépasser ont été très récemment mis en évidence dans [LR20b] ainsi que dans les contributions de cette thèse qui font l'objet du chapitre 7.

Proposition 3.1. *La max-résolution n'est pas inférentiellement complète. [LR20b]*

Démonstration. Soit $\phi = (x,1) \wedge (y,1)$. Bien que $\phi \equiv (x \vee y,1) \wedge (x \vee \overline{y},1) \wedge (y,1)$, la clause $(x \vee y,1)$ ne peut pas être déduite par max-résolution. □

Pour combler les limites de la max-résolution, une idée récente est de lui ajouter de nouvelles règles comme le *split* [LR20b; BL20; Fil+20], l'extension (définie dans [LR20a]) ou encore la règle du *virtual* (définie dans [LR20b]). En particulier, le système **ResS** composé de la max-résolution et du *split* est inférentiellement complet [LR20b]. De plus, les relations entre différents systèmes de preuve ont été étudiées dans [BL20], avec notamment l'équivalence entre **ResS** et le système **Symmetric Cut + Split**, ce qui permet d'en déduire la complétude pour l'inférence dans Max-SAT.

Définition 3.10 (ResS)**.** *Le système **ResS** est composé de deux règles d'inférence : la max-résolution et le split.*

Proposition 3.2. ***ResS** est inférentiellement complet. [LR20b]*

L'étude des systèmes inférentiellement complets pour Max-SAT a très récemment été abordée dans des travaux de la littérature [LR20b; BL20] et on s'y intéressera dans les contributions du chapitre 7 qui portent sur comment déduire, à partir d'une formule initiale, n'importe quelle information (clause ou formule) en utilisant des transformations qui préservent l'équivalence Max-SAT. En particulier, on définira un nouveau système de preuve, **ExC**, que l'on démontrera comme étant inférentiellement complet pour Max-SAT. On proposera également un algorithme pour construire l'inférence de clauses et de formules dans **ExC** avec des bornes supérieures sur la taille des inférences calculées.

3.6. Conclusion

Dans ce chapitre, on a présenté le problème Max-SAT, les principales règles d'inférence, méthodes de résolution et systèmes de preuve. Les principales méthodes de résolution sont réparties entre les algorithmes de type *séparation et évaluation*, qui sont efficaces sur des instances aléatoires ainsi que certaines instances académiques, et les algorithmes qui font appels à un solveur SAT ou ILP qui sont efficaces sur des instances industrielles ainsi que certaines instances académiques. Ces méthodes ont en commun le fait de détecter et de traiter des cœurs inconsistants.

On a aussi observé que, contrairement à SAT, peu de travaux de la littérature porte sur les certificats pour Max-SAT, notamment à cause du fait que l'on ignore comment adapter n'importe quelle réfutation par résolution en max-réfutation sans en augmenter considérablement sa taille, ce qui est une question ouverte dans le domaine [BLM06]. Dans les chapitres 4 et 5, on proposera deux méthodes afin d'adapter

n'importe quelle réfutation par résolution pour en déduire une max-résolution. Cela permettra, dans le chapitre 6, de créer un outil permettant de créer des certificats pour le problème Max-SAT, outil testé sur des instances de la littérature [Bac+20] et mis à la disposition de la communauté scientifique [Py21].

Enfin, des travaux récents ont mis en évidence les limites de la max-résolution et son incomplétude pour l'inférence de clauses et de formules dans Max-SAT, ainsi que la complétude d'autres systèmes de preuve comme **ResS**. On propose dans le chapitre 7 d'étudier l'inférence de clauses et de formules dans Max-SAT, en proposant un nouveau système de preuve inférentiellement complet, **ExC**, ainsi qu'un algorithme pour déduire clauses et formules en utilisant ce système de preuve.

Deuxième partie

Contributions

4. Applications des réfutations par résolution grâce à l'appel à un oracle SAT

Sommaire

4.1. Introduction

Dans ce chapitre, ainsi que plus tard dans le chapitre 5, on s'intéresse à comment adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation. Rappelons qu'avant les travaux de cette thèse, l'adaptation des réfutations par résolution n'était possible que dans le cas *read-once*, où il suffit de remplacer chaque étape de résolution par une étape de max-résolution, comme dans l'exemple 3.8.

Dans ce chapitre, on propose un premier moyen d'adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation. Pour cela, on applique la même méthode que dans le cas *read-once*, en remplaçant successivement chaque résolution par une max-résolution et on applique progressivement la transformation ainsi générée. Cependant, comme la max-résolution consomme les clauses sur lesquelles elle est appliquée, il peut arriver que la max-résolution porte sur deux clauses dont au moins une n'existe pas dans la formule courante. Dans ce cas-là, on propage la falsification de la clause manquante et toute réfutation par résolution de la formule obtenue pourra être déduite en une transformation permettant de générer une copie de la clause manquante.

Ce chapitre est découpé en plusieurs parties. Tout d'abord, on présente dans la section 4.2 comment adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation. Cette adaptation est ensuite résumée sous la forme d'un algorithme dans la section 4.3. puis, on illustre par un exemple le fonctionnement de l'algorithme de génération de remplaçants dans la section 4.4. On conclut enfin dans la section 4.5.

Les travaux présentés dans ce chapitre ont été publiés à la conférence *ICTAI 2021* dans l'article *Computing Max-SAT Refutations using SAT Oracles* [PCH21b].

4.2. Principe

Dans cette section, on présente comment adapter n'importe quelle réfutation par résolution pour obtenir une max-réfutation. L'idée est de suivre les étapes de résolution de la réfutation, et de regarder s'il est possible d'appliquer la max-résolution sur les mêmes prémisses que la résolution. Si cela est possible, on le fait. Sinon, cela signifie qu'au moins une des deux clauses prémisses n'appartient pas dans la formule courante et on lance alors une sous-procédure pour en générer une copie.

On démontre dans le théorème 4.1 comment construire, étant donnée une formule ϕ et une réfutation par résolution $P = (r_1, r_2, ..., r_s)$ de ϕ, une max-réfutation de ϕ à partir de la réfutation P. Pour cela, on définit ci-dessous la projection Max-SAT d'une réfutation par résolution, qui correspond au remplacement de chaque résolution de la réfutation par une max-résolution sur les mêmes prémisses.

Définition 4.1 (Projection Max-SAT d'une réfutation par résolution)**.**
Soit $P = (r_1, r_2, ..., r_s)$ une réfutation par résolution contenant s étapes de résolution. On appelle projection Max-SAT de P, que l'on note $MS(P)$, la séquence de max-résolutions $MS(P) = (mr_1, mr_2, ..., mr_s)$, où mr_i est l'application de la max-résolution sur les prémisses de la résolution r_i.

Remarque 4.1. *$MS(P)$ est une max-réfutation valide de ϕ si P est read-once.*

Maintenant, on montre dans le théorème 4.1 qu'il est toujours possible de construire une max-réfutation de ϕ qui contienne toutes les étapes d'inférence de $MS(P)$ dans le même ordre.

Théorème 4.1. *Soit ϕ une formule inconsistante, P une réfutation par résolution de ϕ. Il existe une max-réfutation ϕ qui contient chaque étape de $MS(P)$ dans le même ordre.*

Démonstration. Soit ϕ une formule inconsistante, $P = (r_1, r_2, ..., r_s)$ une réfutation par résolution de ϕ et $MS(P) = (mr_1, mr_2, ..., mr_s)$ la projection Max-SAT de P. Pour chaque étape de résolution r_i ($i \in \{1, .., s\}$) sur les clauses prémisses c_1 et c_2, on applique l'étape de max-résolution mr_i si les deux clauses sont présentes dans la formule. Quand une clause c n'est pas dans la formule, on génère un remplaçant pour cette

clause en utilisant la méthode suivante. Soit $\phi_{|\overline{c}}$ la formule obtenue à partir de ϕ après la propagation de chaque littéral dans $\{\overline{l} \mid l \in c\}$. La formule $\phi_{|\overline{c}}$ est insatisfiable parce que ϕ l'est. Comme $\phi_{|\overline{c}}$ est insatisfiable, il existe une séquence de résolutions R depuis $\phi_{|\overline{c}}$ jusqu'à $\square$ (autrement dit, une réfutation par résolution). Si l'on remplace chaque clause de cette réfutation par l'état dans lequel elle était avant la propagation faite à partir de ϕ, on obtient une séquence de résolutions depuis ϕ jusqu'à une clause qui sous-somme c. On prouve par récurrence sur le nombre de variables utilisées en tant que pivot d'au moins une étape de résolution de R qu'il existe une séquence de transformations depuis ϕ jusqu'à un équivalent Max-SAT contenant c :

— Initialisation de la récurrence (0 et 1 variables utilisées) : Si le nombre de variables utilisées en tant que pivot dans la réfutation est 0, alors il existe une clause vide $\square \in \phi_{|\overline{c}}$ et par conséquent il existe une clause $c_s \in \phi$ qui sous-somme c. Remarquons aussi que si le nombre de variables utilisées en tant que pivot dans la réfutation est 1, alors R contient une seule étape de résolution sur (x) et $(\overline{x})$ et il est donc possible de générer une clause c_s sous-sommant c en appliquant une max-résolution. Si nécessaire, on applique ensuite une séquence finie d'étapes de *split* grâce à l'équivalence $c_s \equiv (c_s \vee \overline{l_1}) \wedge (c_s \vee l_1 \vee \overline{l_2}) \wedge ... \wedge (c_s \vee l_1 \vee ... \vee l_{k-1} \vee \overline{l_k}) \wedge c$ pour obtenir un équivalent Max-SAT de ϕ contenant c (avec l_1, ..., l_k des littéraux et c_s une clause).

— Posons maintenant $k > 1$ et supposons que la propriété soit vraie pour tout $k' < k$ et notons R une réfutation par résolution utilisant k variables en tant que pivot d'au moins une étape de résolution.

  — Soit R est *read-once* et on remplace alors chaque résolution par une max-résolution pour obtenir une transformation finie générant une clause c_s qui sous-somme c. On applique ensuite une séquence finie d'étapes de *split* à partir de c_s pour générer un équivalent contenant c.

  — Soit R n'est pas *read-once* et on applique alors la même méthode que dans le cas *read-once* avec une différence. Comme R n'est pas *read-once*, alors il est possible d'avoir à appliquer une max-résolution sur une (ou deux) clause(s) manquante(s). Dans ce cas, on génère récursivement un remplaçant pour cette clause après avoir effectué la propagation de sa falsification (on propage les littéraux opposés à la clause). Chaque récursion utilise une nouvelle réfutation par résolution avec au moins une variable en moins et la génération d'un remplaçant est donc forcément finie par hypothèse de récurrence (car le nombre de variables décroît). Étant donnée une réfutation R, on a au plus $2|R|$ remplaçants à générer pour cette réfutation donc la transformation globale pour obtenir une clause $c_s \in \phi$ qui sous-somme c est finie. Comme on applique ensuite une séquence finie d'étapes de *split* pour obtenir la clause c, alors la transformation complète générant c est finie.

Pour conclure, étant donnée une clause prémisse manquante c d'une étape de max-résolution mr_i, on est toujours capable de générer un remplaçant pour cette clause sans consommer l'autre prémisse (car la propagation faite pour générer le remplaçant met de côté l'autre clause sans y toucher). On est donc sûr de pouvoir appliquer itérativement chaque étape $MS(P)$ et la transformation complète obtenue est une

max-réfutation finie de ϕ contenant chaque étape de $MS(P)$ dans le même ordre. □

La procédure décrite dans le théorème 4.1 peut être écrite sous la forme d'un algorithme, que nous appellerons *algorithme de génération de remplaçants* et que nous décrivons dans la section suivante.

4.3. Algorithme de génération de remplaçants

L'algorithme de génération de remplaçants est décrit dans l'algorithme 4.1. Celui-ci utilise une sous-procédure décrite dans l'algorithme 4.2 afin de générer un remplaçant pour une clause manquante. Le principe de l'algorithme est de vérifier, pour chaque résolution, si chaque prémisse est actuellement présente dans la formule. Si une prémisse est absente de la formule, on appelle une sous-procédure afin de générer une copie (fonction générer_remplaçant). Quand les deux prémisses sont présentes (après génération d'une copie si nécessaire), on applique la transformation par max-résolution (fonction appliquer_max_résolution). L'algorithme retourne la formule obtenue après application de toutes les transformations effectuées ainsi que la liste de ces transformations (ces transformations sont insérées progressivement dans la liste T au fur et à mesure de leur calcul et application). La transformation globale récupérée à la fin de l'algorithme forme une max-réfutation de la formule initiale.

Algorithme 4.1 Algorithme de génération de remplaçants

Entrée : Formule insatisfiable ϕ, séquence de résolution P déduisant c
Sortie : (ϕ', T) où T est une transformation de ϕ vers ϕ' contenant c

```
1:  T ← ∅
2:  pour tout résolution sur les clauses c1 et c2 de P faire
3:      si c1 ∉ φ alors
4:          (φ, T') ← générer_remplaçant(c1, φ)
5:          T ← T.T'
6:      si c2 ∉ φ alors
7:          (φ, T') ← générer_remplaçant(c2, φ)
8:          T ← T.T'
9:      (φ, T') ← appliquer_max_résolution(c1, c2, φ)
10:     T ← T.T'
11: retourner (φ, T)
```

Pour générer un remplaçant d'une clause prémisse c manquante, on propage la falsification de cette clause (fonction propager) et on calcule une réfutation par résolution P de la formule ainsi obtenue (fonction calculer_réfutation_par_résolution). On annule ensuite la propagation effectuée précédemment (fonction annuler_propagation) pour que la réfutation P soit désormais une séquence de résolutions sur les clauses de la formule courante qui génère une clause qui sous-somme c. On applique ensuite récursivement l'algorithme de génération de remplaçant pour adapter la séquence

de résolutions pour Max-SAT et l'appliquer (fonction algorithme_de_génération_de_-remplaçants). Enfin, si la clause ultimement générée par la transformation n'est pas la clause c mais une clause qui la sous-somme, on récupère cette clause (fonction chercher_clause_sous_somme_de) et on applique la règle du *split* pour obtenir c (fonction déduire_par_split). La fonction retourne la formule obtenue après la génération du remplaçant pour c et la liste des transformations effectuées.

Algorithme 4.2 Sous-procédure : générer_remplaçant

Entrée : Clause c à déduire, formule insatisfiable ϕ
Sortie : (ϕ', T) où T est une transformation de ϕ vers ϕ' contenant c

1: $\phi \leftarrow$ propager$(\overline{c}, \phi)$
2: $P \leftarrow$ calculer_réfutation_par_résolution(ϕ)
3: $(\phi, P) \leftarrow$ annuler_propagation$(\overline{c}, \phi, P)$
4: $(\phi, T) \leftarrow$ algorithme_de_génération_de_remplaçants(ϕ, P)
5: **si** $c \notin \phi$ **alors**
6: $\quad c' \leftarrow$ chercher_clause_sous_somme_de(c, ϕ)
7: $\quad \phi, T' \leftarrow$ déduire_par_split(c, c', ϕ)
8: $\quad T \leftarrow T.T'$
9: **retourner** (ϕ, T)

Dans la prochaine section, on applique l'algorithme de génération de remplaçants sur un exemple de réfutation par résolution non *read-once*.

4.4. Illustration

Dans cette section, on applique l'algorithme de génération de remplaçants sur l'exemple suivant. On utilisera le terme *niveau* pour faire référence à la profondeur actuelle des appels récursifs à la fonction *algorithme_de_génération_de_remplaçants*. Ainsi, chaque étape de la réfutation initiale est dans le niveau 1 et chaque étape d'une réfutation permettant de générer un remplaçant pour une clause prémisse d'une étape du niveau l est dans le niveau $(l+1)$.

Exemple 4.1. *On considère la formule* $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$ *avec la réfutation par résolution de la figure 2.2.*

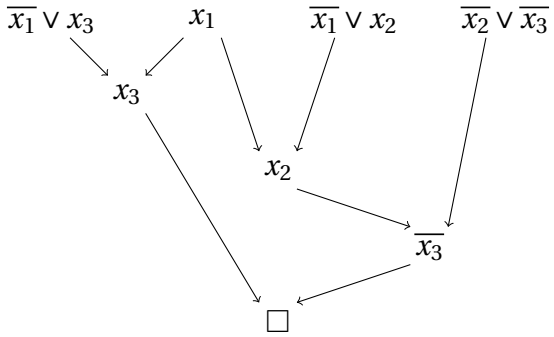


FIGURE 2.2. – Réfutation par résolution (cf page 39)

La figure 2.2 propose une représentation ambiguë de la réfutation par résolution. En effet, les étapes de résolution sont partiellement ordonnées et on pourrait décider que la première résolution soit sur les clauses $(\overline{x_1} \vee x_3)$ *et* (x_1) *tout comme on pourrait considérer qu'elle soit sur les clauses* (x_1) *et* $(\overline{x_1} \vee x_2)$*, les deux choix menant à une exécution différente de l'algorithme. Avant d'exécuter l'algorithme de génération de remplaçants, on va donc proposer un ordre total arbitraire décrit dans la table 4.1. L'application de l'algorithme de génération de remplaçants est résumée dans la table 4.1 ainsi que dans la figure 4.3 et on donne plus de détails ci-après.*

Première itération *La première étape de résolution porte sur les clauses* $(\overline{x_1} \vee x_3)$ *et* (x_1)*. Comme ces deux clauses appartiennent à la formule courante (qui est ici la formule initiale), on peut appliquer la max-résolution sur ces deux clauses et remplacer les clauses* $(\overline{x_1} \vee x_3)$ *et* (x_1) *par les nouvelles clauses* (x_3) *et* $(x_1 \vee \overline{x_3})$*. La formule est maintenant* $\phi = (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3) \wedge (x_1 \vee \overline{x_3})$*.*

Deuxième itération *La deuxième étape de résolution porte sur les clauses* (x_1) *et* $(\overline{x_1} \vee x_2)$*. La clause* (x_1) *n'appartient plus à la formule courante, on doit donc générer un remplaçant pour cette clause.*

Génération du remplaçant pour (x_1) *Pour générer un remplaçant pour* (x_1)*, on propage sa falsification* $\overline{x_1}$*, on calcule une réfutation par résolution sur la formule obtenue et on annule la propagation effectuée pour obtenir une séquence de résolutions permettant de générer la clause* (x_1) *(figure 4.1).*

Première itération (deuxième niveau) *La seule et unique résolution pour générer le remplaçant est sur les clauses* (x_3) *et* $(x_1 \vee \overline{x_3})$ *qui sont dans la formule courante. On applique donc la max-résolution sur ces clauses pour obtenir les clauses* (x_1) *et* $(\overline{x_1} \vee x_3)$*. Le remplaçant de* (x_1) *a donc été généré et la formule courante est* $\phi = (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_1) \wedge (\overline{x_1} \vee x_3)$*.*

Étape	Résolution	Algorithme de génération de remplaçants	Formule courante
1	$(\overline{x_1} \vee x_3) \wedge (x_1) \rightarrow (x_3)$	$(\overline{x_1} \vee x_3) \wedge (x_1) \rightarrow (x_3) \wedge (x_1 \vee \overline{x_3})$	$(\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3) \wedge (x_1 \vee \overline{x_3})$
2	$(x_1) \wedge (\overline{x_1} \vee x_2) \rightarrow (x_2)$	Génération de remplaçant pour (x_1) (Figure 4.1) $(x_3) \wedge (x_1 \vee \overline{x_3}) \rightarrow (x_1) \wedge (\overline{x_1} \vee x_3)$ Remplaçant pour (x_1) généré $(x_1) \wedge (\overline{x_1} \vee x_2) \rightarrow (x_2) \wedge (x_1 \vee \overline{x_2})$	$(\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3) \wedge (x_2) \wedge (x_1 \vee \overline{x_2})$
3	$(x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \rightarrow (\overline{x_3})$	$(x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \rightarrow (\overline{x_3}) \wedge (x_2 \vee x_3)$	$(\overline{x_1} \vee x_3) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3}) \wedge (x_2 \vee x_3)$
4	$(x_3) \wedge (\overline{x_3}) \rightarrow \square$	Génération de remplaçant pour (x_3) (Figure 4.2) $(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3) \rightarrow (\overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$ $\wedge (\overline{x_1} \vee x_2 \vee x_3)$ $(\overline{x_2} \vee x_3) \wedge (x_2 \vee x_3) \rightarrow (x_3)$ Remplaçant pour (x_3) généré $(x_3) \wedge (\overline{x_3}) \rightarrow \square$	$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge \square$

Tableau 4.1. – Application de l'algorithme de génération de remplaçants

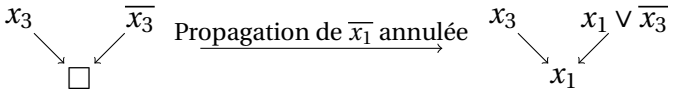


FIGURE 4.1. – Génération d'un remplaçant pour (x_1)

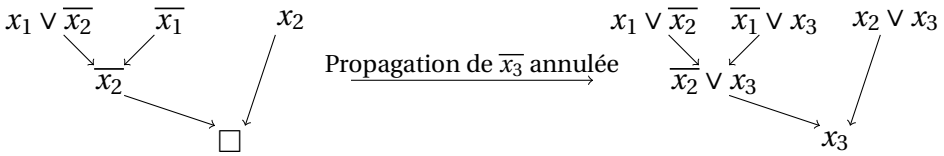


FIGURE 4.2. – Génération d'un remplaçant pour (x_3)

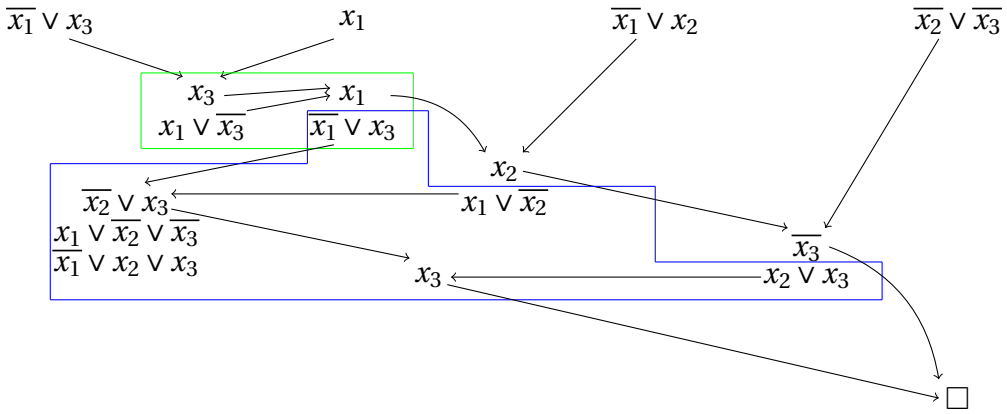


FIGURE 4.3. – Adaptation d'une réfutation par résolution pour Max-SAT

Retour au niveau 1 et fin de la deuxième itération *Les clauses* (x_1) *et* $(\overline{x_1} \vee x_2)$ *sont maintenant dans la formule courante. On applique donc la max-résolution sur ces clauses et on obtient les clauses* (x_2) *et* $(x_1 \vee \overline{x_2})$*. La formule courante est maintenant* $\phi = (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3) \wedge (x_2) \wedge (x_1 \vee \overline{x_2})$.

Troisième itération *La troisième étape de résolution est sur les clauses* (x_2) *et* $(\overline{x_2} \vee \overline{x_3})$*, qui sont dans la formule courante. On peut donc appliquer la max-résolution sur ces clauses et obtenir les clauses* $(\overline{x_3})$ *et* $(x_2 \vee x_3)$*. La formule courante est maintenant* $\phi = (\overline{x_1} \vee x_3) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3}) \wedge (x_2 \vee x_3)$.

Quatrième itération *La quatrième étape de résolution porte sur les clauses* (x_3) *et* $(\overline{x_3})$*. La clause* (x_3) *n'appartient plus à la formule courante, on doit donc générer un remplaçant pour cette clause.*

Génération du remplaçant pour (x_3) *Pour générer un remplaçant pour* (x_3)*, on propage sa falsification* $\overline{x_3}$*, on calcule une réfutation par résolution sur la formule obtenue et on annule la propagation effectuée pour obtenir une séquence de résolutions permettant de générer la clause* (x_3) *(figure 4.2).*

Première itération (second niveau) *La première résolution pour générer le remplaçant est sur les clauses* $(x_1 \vee \overline{x_2})$ *et* $(\overline{x_1} \vee x_3)$*, qui sont dans la formule courante. On applique donc la max-résolution sur ces clauses et on obtient les clauses* $(\overline{x_2} \vee x_3)$*,* $(x_1 \vee \overline{x_2} \vee \overline{x_3})$ *et* $(\overline{x_1} \vee x_2 \vee x_3)$*. La formule courante est maintenant* $\phi = (\overline{x_3}) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$.

Deuxième itération (second niveau) *La deuxième résolution pour générer le remplaçant est sur les clauses* $(\overline{x_2} \vee x_3)$ *et* $(x_2 \vee x_3)$*, qui sont dans la formule courante. On applique donc la max-résolution sur ces clauses et on obtient la clause* (x_3)*. Le remplaçant pour* (x_3) *a été généré et la formule courante est maintenant* $\phi = (\overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_3)$.

Retour au niveau 1 et fin de la quatrième itération *Les deux clauses* (x_3) *et* $(\overline{x_3})$ *sont maintenant dans la formule courante. On applique donc la max-résolution sur ces clauses et on obtient la clause vide* $\square$*. La formule courante est maintenant* $\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge \square$.

Fin de l'exécution et bilan *L'exécution de l'algorithme de génération de remplaçants est terminée. On a pu adapter la réfutation par résolution de la figure 2.2 pour transformer la formule initiale* $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$ *en la formule équivalente* $\phi' = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge \square$ *qui contient une clause vide. La max-réfutation ainsi générée est représentée dans la figure 4.3, la partie verte étant la génération du remplaçant pour la clause* (x_1) *alors que la partie bleue est la génération du remplaçant pour la clause* (x_3).

4.5. Conclusion

Dans ce chapitre, on a présenté une méthode permettant d'adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation. La max-réfutation ainsi obtenue est, par définition, une transformation qui préserve l'équivalence Max-SAT, depuis une formule initiale jusqu'à un équivalent contenant la clause vide. Cette méthode est un premier pas vers la génération pratique de certificats pour le problème Max-SAT.

Les travaux futurs incluent une étude expérimentale pour comparer les performances de l'algorithme de génération de remplaçants par rapport aux travaux proposés dans le chapitre 5 et pour essayer d'améliorer les certificats générés dans le chapitre 6. De plus, on peut observer un effet de retour-arrière dans l'exemple résumé dans la figure 4.3 pour la génération du remplaçant pour la clause (x_1) où l'algorithme annule une règle d'inférence appliquée précédemment pour générer un remplaçant. Par conséquent, il pourrait être intéressant d'étudier une variation de l'algorithme de génération de remplaçants dans laquelle l'algorithme n'est pas appliqué sur toute la réfutation, mais sur une seule branche de la réfutation (ce qui aurait pour conséquent que certains remplaçants seraient générés pour créer des clauses qui n'ont pas été générées du tout).

Dans le prochain chapitre, on propose une méthode alternative permettant d'appliquer n'importe quelle réfutation par résolution pour Max-SAT. Le but est de proposer une méthode avec des garanties théoriques sur la taille de la max-réfutation calculée par rapport à la taille de la réfutation initiale. Cette méthode aura également l'avantage de ne pas nécessiter de calculs d'autres réfutations par résolution.

5. Des réfutations SAT aux réfutations Max-SAT

Sommaire

5.1. Introduction

Dans ce chapitre, on propose une deuxième méthode pour adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation. Cette deuxième méthode, par rapport à la première, a l'avantage de ne pas nécessiter de calculs de réfutations par résolution intermédiaires. De plus, il est possible avec cette méthode de donner des garanties théoriques sur la taille de la max-réfutation obtenue.

Pour adapter n'importe quelle réfutation par résolution en max-réfutation, il faut s'intéresser au cas des clauses non *read-once* de la réfutation, c'est à dire les clauses qui sont utilisées plusieurs fois en tant que prémisse d'une étape de résolution. Pour calculer une max-réfutation à partir de la réfutation, il faudrait pouvoir dupliquer une clause utilisée plusieurs fois afin de disposer d'autant de copies de cette clause que nécessaire. Afin de pouvoir générer des copies d'une clause utilisée plusieurs fois, on va enrichir la max-résolution et utiliser en plus la règle du *split*.

Ce chapitre est découpé en plusieurs parties. Tout d'abord, on présente dans la section 5.2 l'adaptation des réfutations par résolution *tree-like regular* en max-réfutations pour Max-SAT, grâce à l'utilisation de la règle du *split*. Ensuite, on va étendre progressivement cette adaptation aux cas plus généraux en se ramenant à chaque fois à ce cas

de base. Ainsi, on présente successivement l'adaptation des réfutations par résolution *tree-like* en max-réfutations dans la section 5.3, des réfutations par résolution *semi-tree-like* en max-réfutations dans la section 5.4 et on présente enfin le cas général dans la section 5.5. On montre enfin qu'il existe des réfutations dont l'adaptation est exponentielle avec la méthode proposée dans la section 5.6 et on conclut dans la section 5.7.

Les travaux présentés dans ce chapitre ont fait l'objet d'une publication internationale à la conférence *ICTAI 2020* dans l'article *Towards Bridging the Gap Between SAT and Max-SAT Refutations* [PCH20], publication qui a été résumée par une publication francophone à la conférence *JFPC 2021* dans l'article *Des réfutations SAT aux réfutations Max-SAT* [PCH21c].

5.2. Cas *tree-like regular*

Dans cette section, on montre comment il est possible d'adapter une réfutation par résolution *tree-like regular* pour obtenir une max-réfutation dont la taille, exprimée en nombre d'étapes d'inférence, est linéaire par rapport à la taille de la réfutation par résolution. L'idée est d'utiliser la règle du *split* pour dupliquer n'importe quelle clause c qui serait utilisée en tant que prémisse de k ($k > 1$) étapes de résolution différentes, ceci afin de générer de nouvelles clauses sous-sommées par c, clauses qui seront utilisées en tant que prémisses des étapes de résolution dont c était une prémisse.

Tout d'abord, on va montrer que lorsqu'une clause c est utilisée plusieurs fois en tant que prémisse d'une étape de résolution, il existe une variable $x \notin var(c)$ sur laquelle utiliser la règle du *split* pour générer deux clauses $(c \vee x)$ et $(c \vee \overline{x})$, clauses qui vont prendre la place de la clause c en tant que prémisse des étapes de résolution. Dans la suite de ce chapitre, on dit que, étant données une réfutation P, une clause c de P, une branche B de la réfutation depuis c jusqu'à $\square$ et une clause c', on dit que la branche B accepte la substitution de c par c' s'il est possible de remplacer, sur la branche B, la clause c par la clause c' sans affecter la validité de la réfutation.

Lemme 5.1. *Étant donnée une réfutation par résolution tree-like regular non-read-once P et une clause non-read-once c de P, il existe une variable $x \notin var(c)$ et une partition de l'ensemble des branches B depuis c jusqu'à $\square$ en deux sous-ensembles non vides B_1 et B_2 telles que les branches de B_1 acceptent la substitution de c par $(c \vee x)$ et les branches de B_2 acceptent la substitution de c par $(c \vee \overline{x})$.*

Démonstration. Soit P une réfutation par résolution *tree-like regular non-read-once* et c une clause *non-read-once* de P. Il existe un nœud v dans le graphe orienté acyclique de P représentant une étape de résolution sur une variable x tel que v est le premier point de jonction de tous les chemins depuis c jusqu'à $\square$. L'existence de ce point de jonction est garantie par le fait que, dans le pire des cas, les chemins depuis c jusqu'à

$\square$ se réunissent bien à l'étape de résolution générant $\square$. Comme la réfutation est *tree-like*, chaque chemin depuis c jusqu'à $\square$ conduit jusqu'à une et une seule prémisse de l'étape de résolution du nœud v. En effet, l'existence d'un chemin conduisant aux deux prémisses force l'existence d'une clause intermédiaire non-read-once, et donc d'une réfutation qui ne serait pas *tree-like*. On note x la variable éliminée par la résolution du nœud v et on partitionne les chemins depuis c jusqu'à v en deux sous-ensembles B_1 et B_2 contenant respectivement les chemins conduisant à la prémisse contenant le littéral x et à la prémisse contenant littéral $\overline{x}$. Comme P est *regular*, x n'est pas une variable de la clause c et les chemins de B_1 acceptent la substitution de c par $(c \vee x)$ tandis que les chemins de B_2 acceptent la substitution de c par $(c \vee \overline{x})$.

$\square$

Le résultat présenté dans le lemme 5.1 garantit la possibilité que, pour chaque clause utilisée plusieurs fois en tant que prémisse d'une étape de résolution, il est possible d'appliquer la règle du *split* afin de générer deux clauses qui seront utilisées à la place de la clause initiale. On peut donc désormais appliquer cette règle pour remplacer n'importe quelle clause utilisée $k > 1$ fois par deux clauses utilisées $1 \leq k_1 < k$ et $1 \leq k_2 < k$ avec $k = k_1 + k_2$. En appliquant itérativement cette méthode, on arrivera donc à éliminer toutes les utilisations multiples de clauses et on pourra ensuite remplacer chaque étape de résolution par une étape de max-résolution pour obtenir, depuis notre réfutation par résolution *tree-like regular*, une max-réfutation de taille linéaire.

Théorème 5.1. *Étant données une formule insatisfiable ϕ et une réfutation par résolution tree-like regular P de ϕ, il existe une max-réfutation de ϕ contenant $O(|P|)$ étapes d'inférence.*

Démonstration. Soit P une réfutation par résolution *tree-like regular* de ϕ. On pose $T_1 = \emptyset$ et $T_2 = MR(P)$, où $MR(P)$ est la projection Max-SAT de P obtenue en remplaçant chaque étape de résolution par la même étape de max-résolution. Si P est *read-once*, T_2 est une max-réfutation de ϕ contenant exactement $|P|$ étapes d'inférence (ce qui est évidemment en $O(|P|)$). Dans le cas contraire, soit c une clause de la réfutation P utilisée plusieurs fois en tant que prémisse d'une étape de résolution. Grâce au lemme 5.1, il existe une variable $x \notin var(c)$ et une partition des chemins partant de la clause c et allant à $\square$ en deux sous-ensembles non vides, le premier acceptant la substitution de c par $(c \vee x)$ et le second acceptant la substitution de c par $c \vee \overline{x}$. On applique la règle du *split* sur la clause c et sur la variable x pour obtenir les deux clauses $(c \vee x)$ et $(c \vee \overline{x})$ et on remplace la clause c en tant que prémisse par la clause $(c \vee x)$ sur le premier sous-ensemble de chemins partant de c et par $c \vee \overline{x}$ sur le second. En faisant cela, on augmente T_1 en ajoutant l'étape de *split* ainsi faite et on modifie les étapes impliquant la clause c dans T_2 comme décrit ci-dessus.Comme T_2 est une réfutation par résolution *tree-like regular* de $(\phi \backslash c) \wedge (c \vee x) \wedge (c \vee \overline{x})$, il est encore possible de répéter ce traitement sur n'importe quelle autre clause utilisée plusieurs fois, et ceci jusqu'à ce que T_2 devienne *read-once regular*. À partir de cet instant, on aura un couple (T_1, T_2) tel que T_1 soit une séquence d'étapes de *split* transformant ϕ en un équivalent Max-SAT ϕ' et T_2 soit une réfutation par résolution *tree-like regular*

de ϕ'. Ces transformations mises les unes après les autres forment une max-réfutation de ϕ.

Pour démontrer que la taille de la réfutation obtenue est en $O(|P|)$, on vérifie que pour toute clause c de la réfutation utilisée $k \geq 1$ fois en tant que prémisse, il est possible, en au plus $(k-1)$ *splits*, de transformer la réfutation pour que la clause c ne soit plus utilisée dans plusieurs transformations et qu'aucune utilisation multiples d'autres clauses n'ait été ajoutée. Cela peut se démontrer par récurrence sur la valeur de k :

— Si $k = 1$, on a clairement besoin de 0 *split* puisque la clause n'est pas utilisée plusieurs fois.

— Supposons que notre assertion est vraie pour tout $k' < k$ et soit c une clause utilisée k fois. En utilisant le lemme 5.1, on sait qu'il est possible d'utiliser 1 *split* pour remplacer la clause c par deux clauses c_1 et c_2 utilisées respectivement k_1 et k_2 fois avec $k_1, k_2 > 0$ et $k_1 + k_2 = k$. En utilisant notre hypothèse de récurrent sur c_1 et c_2, on sait qu'il est possible de faire notre transformation avec au plus $k_1 - 1$ *splits* pour c_1 et avec au plus $k_2 - 1$ *splits* pour c_2. Il est donc possible de transformer la réfutation pour que la clause c ne soit plus utilisée dans plusieurs transformations et qu'aucune utilisation multiple d'autres clauses n'ait été ajoutée avec au plus $1 + (k_1 - 1) + (k_2 - 1) = k - 1$ *splits*.

Soit $c_1, ..., c_p$ les clauses feuille de P et notons respectivement $k^1, k^2, ..., k^p$ leur nombre d'utilisation respective. Remarquons que $k^1 + k^2 + ... + k^p = |P| + 1$ car P a exactement $2|P|$ prémisses (chacune des $|P|$ étapes d'inférence utilise 2 clauses), ainsi que $|P| - 1$ clauses dérivées par résolution qui sont réutilisées en tant que prémisse (la clause vide est la seule qui n'est pas réutilisée si on ignore le cas trivial ou il existerait une autre clause dérivée inutilisée, cas qui se règle en supprimant la portion inutilisée). En utilisant le résultat du raisonnement par récurrence précédent, on a besoin d'au plus $k^1 - 1 + k^2 - 1 + ... + k^p - 1 \leq |P|$ étapes de *split* pour toutes les clauses feuilles de P et par conséquent, on a $|T_1| \leq |P|$. De l'autre côté, le nombre de max-résolution de T_2 est par construction égal au nombre de résolutions dans P et par conséquent $|T_2| = |P|$. On conclut que la max-réfutation obtenue contient au plus $2|P|$ étapes d'inférence, ce qui est en $O(|P|)$.

□

Exemple 5.1. *On considère la formule* $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$ *avec la réfutation par résolution tree-like regular représentée dans la figure 2.2. On constate que la clause* (x_1) *est utilisée deux fois en tant que prémisse d'une étape de résolution. En explorant les deux branches qui partent de* (x_1)*, on trouve que le point de jonction de ces deux branches est sur la dernière étape de résolution permettant de déduire* □*, étape de résolution qui est sur la variable* x_3*. De plus, la branche de gauche arrive sur la prémisse contenant le littéral* x_3 *alors que la branche de droite arrive sur la prémisse contenant le littéral* $\overline{x_3}$*. Par conséquent, on applique la règle du split sur la clause* (x_1) *et sur la variable* x_3 *pour obtenir les clauses* $(x_1 \vee x_3)$ *et* $(x_1 \vee \overline{x_3})$ *puis on remplace la clause* (x_1) *par* $(x_1 \vee x_3)$ *et* $(x_1 \vee \overline{x_3})$ *sur respectivement la branche de gauche et la branche de*

droite. Enfin, on remplace chaque étape de résolution par une étape de max-résolution pour obtenir la max-réfutation de ϕ représentée dans la figure 5.1.

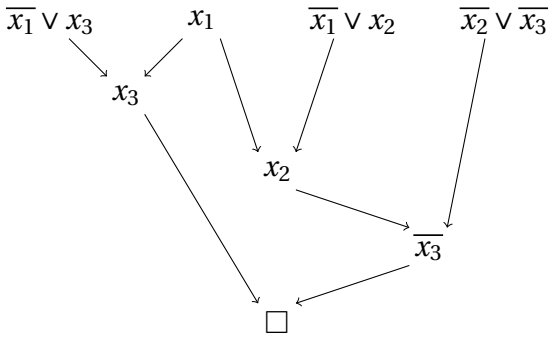


FIGURE 2.2. – Réfutation par résolution *tree-like regular* (cf page 39)

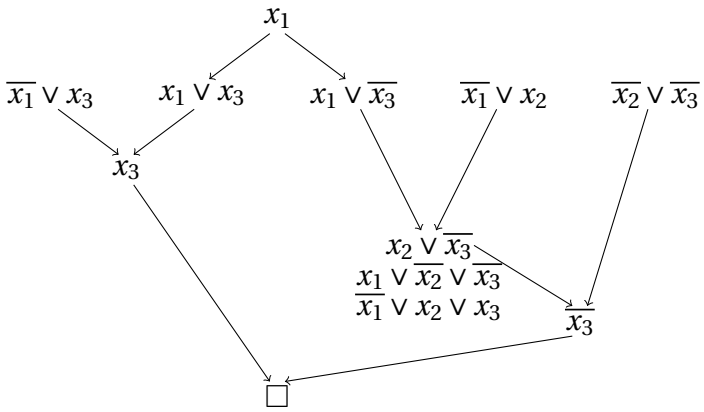


FIGURE 5.1. – Adaptation d'une réfutation par résolution *tree-like regular*

5.3. Cas *tree-like*

Dans la section précédente, on a proposé une adaptation linéaire de n'importe quelle réfutation par résolution *tree-like regular* en une max-réfutation. On propose dans cette section d'étendre cette adaptation aux réfutations *tree-like*. Pour cela, on exhibe simplement une transformation connue permettant d'améliorer n'importe quelle réfutation par résolution *tree-like* pour la rendre *tree-like regular*, et cela sans en augmenter sa taille. Ce résultat a été démontré dans [Urq01] sous la forme du lemme ci-dessous (voir lemme 5.1 de [Urq01]). La preuve, constructive, consiste à supprimer des redondances dans une réfutation *tree-like* afin de la rendre *tree-like regular*. Plus précisément, lorsqu'une branche possède au moins deux résolutions sur la même variable, il est possible de supprimer la première résolution sur cette variable et de mettre à jour le reste de la réfutation, en supprimant potentiellement d'autres résolutions devenues inapplicables.

Lemme 5.2. *[Urq01] Une réfutation par résolution tree-like de taille minimale est regular.*

Exemple 5.2. *On considère la réfutation par résolution tree-like représentée dans la figure 5.2. Cette réfutation n'est pas regular à cause d'une branche contenant une irrégularité à cause de deux résolutions sur la variable x_1. Cependant, il est possible de supprimer cette irrégularité en enlevant la première résolution sur la variable x_1 et en mettant à jour le reste de la réfutation, ce qui donne la réfutation tree-like regular représentée dans la figure 5.3. Remarquons qu'après la transformation, les clauses $(\overline{x_1} \vee \overline{x_3})$ et (x_3) ne sont plus utilisées dans la réfutation.*

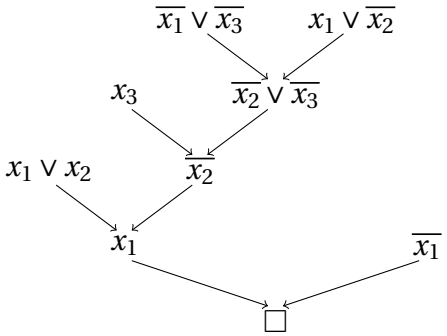


FIGURE 5.2. – Réfutation par résolution avec une irrégularité sur la variable x_1

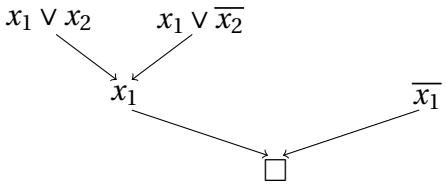


FIGURE 5.3. – Minimisation d'une réfutation *tree-like* pour la rendre *regular*

Comme il est possible de transformer n'importe quelle réfutation par résolution *tree-like* pour la rendre également *regular* sans augmenter sa taille, alors on peut, pour n'importe quelle réfutation par résolution *tree-like*, la rendre *tree-like regular* puis appliquer l'adaptation proposée dans le théorème 5.1 pour l'adapter en une max-réfutation de taille linéaire.

Corollaire 5.1. *Étant données une formule insatisfiable ϕ et une réfutation par résolution tree-like P de ϕ, il existe une max-réfutation de ϕ contenant $O(|P|)$ étapes d'inférence.*

Démonstration. Soit ϕ une formule insatisfiable et P une réfutation par résolution *tree-like* de ϕ. Grâce au lemme 5.2, il existe une réfutation par résolution P_2 *tree-like regular* de ϕ telle que $|P_2| = O(|P|)$. En appliquant le théorème 5.1 sur la réfutation P_2, on obtient une max-réfutation contenant $O(|P_2|) = O(|P|)$ étapes d'inférence.

□

5.4. Cas *semi-tree-like*

Dans la section 5.3, on a proposé une adaptation linéaire de n'importe quelle réfutation par résolution *tree-like* afin d'obtenir une max-réfutation. On propose dans cette section d'étendre cette adaptation linéaire au cas des réfutations *semi-tree-like* définies ci-dessous. Comme indiqué dans la proposition 5.1, cette classe de réfutations étend la classe des réfutations *tree-like*, c'est à dire que chaque réfutation par résolution *tree-like* est aussi *semi-tree-like*.

Définition 5.1 (Réfutation par résolution *semi-tree-like*)**.** *Une réfutation par résolution est dite semi-tree-like si chaque branche de la réfutation contient au plus une clause utilisée plusieurs fois en tant que prémisse d'une étape de résolution.*

Exemple 5.3. *On considère la réfutation par résolution P de la figure 5.4. La réfutation P est clairement semi-tree-like dans chaque branche de la réfutation contient au plus une clause utilisée plusieurs fois. Elle n'est cependant pas tree-like car (x_1), qui est utilisée plusieurs fois, est une clause intermédiaire de la réfutation.*

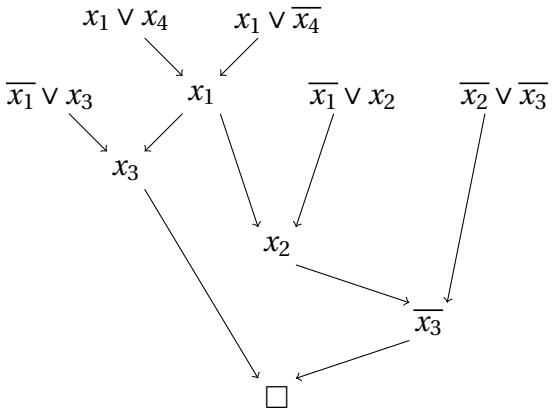


FIGURE 5.4. – Réfutation par résolution *semi-tree-like*

Proposition 5.1. *Toute réfutation par résolution tree-like est aussi semi-tree-like.*

Démonstration. Soit P une réfutation par résolution *tree-like*. Par définition, chaque clause intermédiaire est utilisée maximum une fois en tant que prémisse d'une résolution. Dans chaque branche de la réfutation, la seule clause pouvant être utilisée plusieurs fois est donc la feuille, ce qui implique qu'il y a maximum une clause utilisée plusieurs fois par branche de la réfutation. Par conséquent, P est *semi-tree-like*.

□

Pour étendre nos résultats aux cas des réfutations par résolution *semi-tree-like*, on propose une méthode qui consiste à remarquer qu'une réfutation *semi-tree-like* peut être partitionnée en deux morceaux, le premier étant une séquence *read-once* d'étapes de résolution et le second étant une réfutation par résolution *tree-like*. On peut donc adapter les deux morceaux séparément pour Max-SAT, le premier en remplaçant

chaque résolution par une max-résolution comme dans [HM11] et le second en utilisant le résultat du corollaire 5.1. Après avoir adapté séparément les deux morceaux, on les recombine ensemble pour construire la max-réfutation.

Théorème 5.2. *Étant données une formule insatisfiable ϕ et une réfutation par résolution semi-tree-like P de ϕ, il existe une max-réfutation de ϕ contenant $O(|P|)$ étapes d'inférence.*

Démonstration. Soit P une réfutation par résolution *semi-tree-like*. Comme P est *semi-tree-like*, chaque branche de P contient au plus une clause utilisée plusieurs fois. On partitionne P en deux morceaux P_1 et P_2 de la manière suivante :

— Pour chaque branche contenant une clause c utilisée plusieurs fois, les transformations jusqu'à c sont mises dans P_1 et celles après c sont mises dans P_2.

— Les transformations de toute branche ne contenant aucune clause utilisée plusieurs fois sont mises dans P_2.

Par construction, P_1 est une séquence de résolution dont aucune clause n'est utilisée plusieurs fois, donc il est possible, en remplaçant chaque résolution par une max-résolution, de l'adapter en séquence de max-résolutions P'_1 contenant exactement $|P_1|$ étapes d'inférence (comme dans le cas *read-once* [HM11]). Ensuite, P_2 est une réfutation par résolution *tree-like* car les clauses utilisées plusieurs fois dans P sont des feuilles de P_2. Il est donc possible de l'adapter en une max-réfutation de taille $O(|P_2|)$ en utilisant le résultat du corollaire 5.1. Finalement, il suffit de recombiner P'_1 et P'_2 pour obtenir une max-réfutation contenant au plus $O(|P_1|+|P_2|) = O(|P|)$ étapes d'inférence.

□

Exemple 5.4. *On considère la réfutation par résolution semi-tree-like de l'exemple 5.3 représentée dans la figure 5.4. Pour adapter cette réfutation, on considère d'un côté la résolution sur les clauses $(x_1 \vee x_4)$ and $(x_1 \vee \overline{x_4})$ (qui est une séquence de résolutions read-once) et de l'autre le reste de la réfutation (qui est une réfutation tree-like), on adapte chaque morceau séparément avant de les refusionner et on obtient la max-réfutation représentée dans la figure 5.5.*

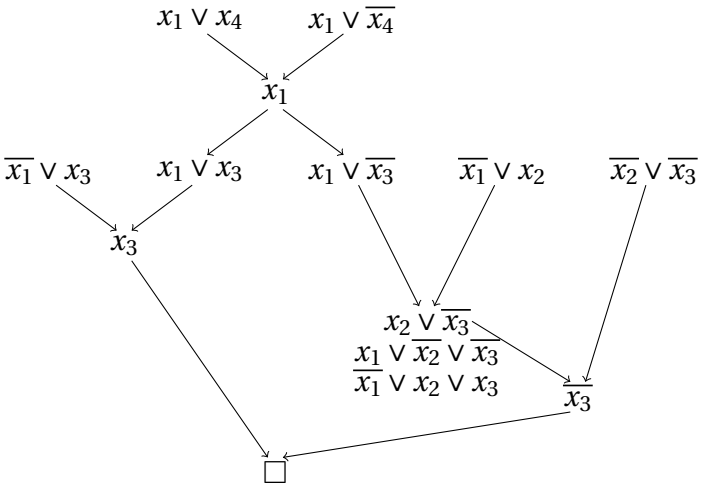


FIGURE 5.5. – Adaptation d'une réfutation *semi-tree-like* pour Max-SAT

5.5. Cas général

Dans les sections 5.2, 5.3 et 5.4, on a proposé des adaptations linéaires pour des classes particulières de réfutations par résolution afin de les rendre valides pour Max-SAT. Dans cette section, on propose une adaptation de n'importe quelle réfutation par résolution afin d'en déduire une max-réfutation. L'idée est simplement de revenir aux cas *tree-like* ou *semi-tree-like* en appliquant une première transformation sur la réfutation. Afin de simplifier la lecture, on rendra dans le lemme 5.3 la réfutation *tree-like*. [1]

Pour revenir au cas *tree-like*, on cherche itérativement la première clause intermédiaire c utilisée $k > 1$ fois et dupliquer la partie de la réfutation qui a permis la génération de c afin de générer k clauses $c_1 = c, c_2, ..., c_k$, chaque clause contenant les mêmes littéraux que la clause c. Les clauses $c_1 = c, c_2, ..., c_k$ vont ensuite se répartir les étapes de résolution dont c était la prémisse (une étape de résolution par clause c_i), et ainsi la clause c ne sera plus utilisée qu'une seule fois. En répétant cette opération, on force la réfutation à devenir *tree-like* car on repousse les clauses utilisées plusieurs fois vers le début de la réfutation. Cependant, effectuer cette transformation pour éliminer une clause intermédiaire utilisée k fois nécessite ici de multiplier par k la taille d'une partie de la réfutation. La transformation globale pour obtenir une réfutation *tree-like* augmente au pire exponentiellement la taille de la réfutation. Pour préciser cette borne supérieure exponentielle, on définit un nouveau paramètre ci-dessous qui est le nombre de multi-utilisations des clauses intermédiaires.

Définition 5.2 (Nombre de multi-utilisations des clauses intermédiaires)**.** *Soit P une réfutation par résolution. Le nombre de multi-utilisations de clauses intermédiaires de*

1. On aurait pu aussi la rendre semi-tree-like, ce qui n'a pas d'impact sur la complexité de la méthode dans le pire des cas.

P, noté $\mu(P)$, est calculé comme suit :

$$\mu(P) = \sum_{\text{clause intermédiaires non-read-once } c} (d^+(c) - 1)$$

où $d^+(c)$ est le nombre d'utilisation de la clause c, c'est à dire le nombre d'étapes de résolution dont c est une clause prémisse.

Lemme 5.3. *Étant données une formule insatisfiable ϕ et une réfutation par résolution P de ϕ, il existe une réfutation par résolution tree-like P de ϕ contenant $O(2^{\mu(P)} \times |P|)$ étapes d'inférence.*

Démonstration. Soit P une réfutation par résolution de ϕ. Itérativement, on considère la première clause intermédiaire utilisée plusieurs fois et on duplique la partie de la réfutation qui a généré la clause c exactement $d^+(c) - 1$ fois. Chaque duplication décrémente le nombre de multi-utilisations des clauses intermédiaires utilisées plusieurs fois et, à chaque duplication, la taille de la preuve est doublée dans le pire des cas. On conclut donc que la taille de la réfutation *tree-like* obtenue est bornée par $O(2^{\mu(P)} \times |P|)$.

□

Maintenant que l'on sait transformer chaque réfutation pour la rendre *tree-like*, on va juste utiliser l'adaptation des réfutations *tree-like* décrite dans la section 5.3.

Théorème 5.3. *Étant données une formule insatisfiable ϕ et une réfutation par résolution P de ϕ, il existe une max-réfutation de ϕ contenant $O(2^{\mu(P)} \times |P|)$ étapes d'inférence.*

Démonstration. Soit P une réfutation par résolution de ϕ. On adapte P pour obtenir une réfutation par résolution *tree-like* de taille $O(2^{\mu(P)} \times |P|)$ grâce au lemme 5.3. Ensuite, grâce au corollaire 5.1, on obtient une max-réfutation de taille $O(2^{\mu(P)} \times |P|)$.

□

Exemple 5.5. *On considère la réfutation par résolution représentée dans la figure 5.6. Cette réfutation n'est pas semi-tree-like car les clauses (x_1) et (x_4) sont deux clauses utilisées plusieurs fois qui sont sur une même branche. Tout d'abord, on duplique la portion de la réfutation ayant généré la clause (x_1) pour obtenir la réfutation par résolution tree-like représentée dans la figure 5.7. Ensuite, on applique l'adaptation présentée dans la section 5.3 pour obtenir la max-réfutation représentée dans la figure 5.8.*

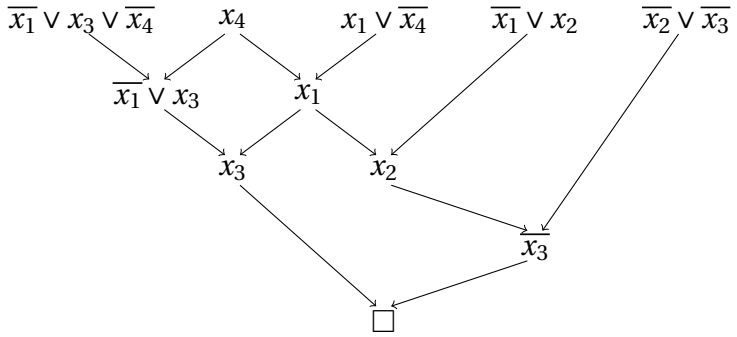


FIGURE 5.6. – Réfutation par résolution non *semi-tree-like*

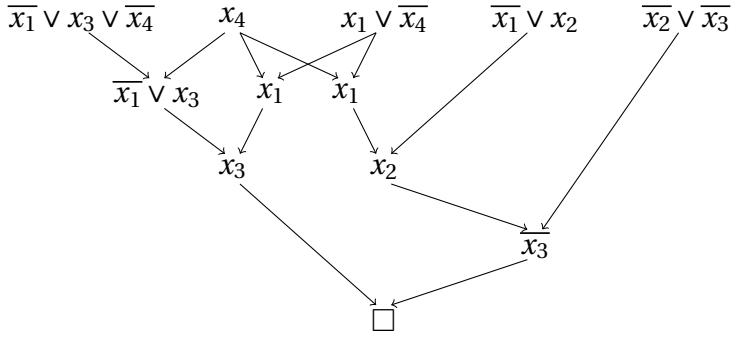


FIGURE 5.7. – Adaptation *tree-like* d'une réfutation non *semi-tree-like*

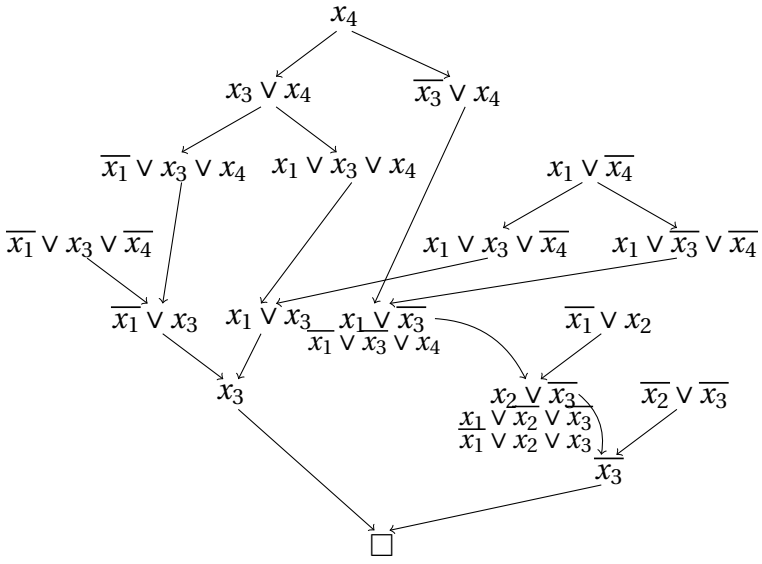


FIGURE 5.8. – Adaptation d'une résolution non *semi-tree-like* en max-réfutation

On termine ce chapitre en exhibant des réfutations par résolution dont l'adaptation proposée dans ce chapitre est de taille exponentielle.

5.6. Motifs en diamant

On présente dans la section suivante un motif de séquence de résolutions que l'on va utiliser pour construire des réfutations dont l'adaptation est exponentielle pour la méthode proposée dans ce chapitre.

Définition 5.3 (Motif en diamant)**.** *Soit A une disjonction de littéraux et $x \notin var(A)$ et $y \notin var(A)$ deux variables distinctes. On définit le motif en diamant (x, y, A) comme la séquence de résolutions représentée dans la figure 5.9.*

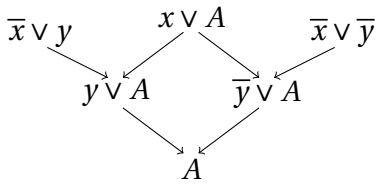


FIGURE 5.9. – Motif en diamant (x, y, A)

FIGURE 5.10. – Représentation simplifiée d'un motif en diamant

On représente de manière simplifiée ce motif dans la figure 5.10. Remarquez que le motif en diamant $(x, y, \square)$ est une réfutation par résolution. Maintenant, imaginons que la clause en haut du motif en diamant $(x, y, \square)$ est elle-même déduite d'un précédent motif en diamant et itérons le même raisonnement pour définir une k-pile de motifs en diamant :

Définition 5.4 (k-pile de motifs en diamant)**.** *Soit $k \geq 1$ un entier naturel et x_i et y_i deux variables distinctes avec $1 \leq i \leq k$. Une k-pile de motifs en diamant est formée par k motifs en diamant (x_i, y_i, A_i) avec $1 \leq i \leq k$ tel que $A_1 = \square$ et $A_i = (x_1 \vee \cdots \vee x_{i-1})$ pour $1 < i \leq k$. Chaque motif en diamant (x_i, y_i, A_i) est empilé sur le motif en diamant $(x_{i-1}, y_{i-1}, A_{i-1})$ (à part celui du bas) tel que la conclusion d'un motif en diamant est la clause du haut du motif en diamant du dessous.*

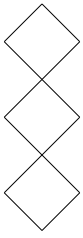


FIGURE 5.11. – Représentation simplifiée d'une 3-pile de motifs en diamant

Une 3-pile de motifs en diamant est représentée dans la figure 5.11. Les k-pile de motifs en diamant sont des réfutations par résolution car elles déduisent la clause vide. La taille d'une k-pile de motifs en diamant P est $|P| = 3k$ et son nombre de multi-utilisations est $\mu(P) = k-1$. En appliquant l'adaptation pour Max-SAT proposée dans ce chapitre, on obtient une max-réfutation de taille au moins 2^{k-1}, ce qui est un cas d'adaptation exponentielle.

Remarquons que l'algorithme de génération de remplaçants, présenté dans le chapitre 4, est capable de calculer une max-réfutation de taille $5k < 2|P|$ pour les k-piles de motifs en diamant, ce qui est linéaire par rapport à la taille de la k-pile de motifs en diamant. En effet, il est possible d'adapter chaque motif en diamant avec au plus 5 étapes de max-résolution, comme indiqué dans la figure 5.12 et dans la table 5.1. Ainsi, malgré l'efficacité théorique de l'adaptation proposée dans ce chapitre sur certaines classes de réfutations, il peut être pertinent de combiner les deux stratégies proposées pour obtenir la meilleure adaptation possible dans le cas général.

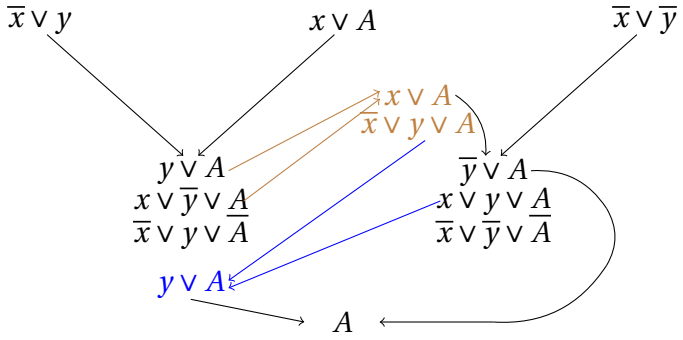


FIGURE 5.12. – Adaptation du motif en diamant (x, y, A) grâce à l'algorithme de génération de remplaçants

Étape	Résolution	Algorithme de génération de remplaçants	Formule courante
1	$(\overline{x} \vee y) \wedge (x \vee A) \rightarrow (y \vee A)$	$(\overline{x} \vee y) \wedge (x \vee A) \rightarrow (y \vee A) \wedge (x \vee \overline{y} \vee A) \wedge (\overline{x} \vee y \vee \overline{A})$	$(\overline{x} \vee \overline{y}) \wedge (y \vee A) \wedge (x \vee \overline{y} \vee A) \wedge (\overline{x} \vee y \vee \overline{A})$
2	$(x \vee A) \wedge (\overline{x} \vee \overline{y}) \rightarrow (\overline{y} \vee A)$	Génération de remplaçant pour $(x \vee A)$ $(y \vee A) \wedge (\overline{x} \vee \overline{y} \vee A) \rightarrow (x \vee A) \wedge (\overline{x} \vee y \vee A)$ Remplaçant pour $(x \vee A)$ généré $(x \vee A) \wedge (\overline{x} \vee \overline{y}) \rightarrow (\overline{y} \vee A) \wedge (x \vee y \vee A) \wedge (\overline{x} \vee \overline{y} \vee \overline{A})$	$(\overline{x} \vee y \vee \overline{A}) \wedge (\overline{x} \vee y \vee A) \wedge (\overline{y} \vee A) \wedge$ $(x \vee y \vee A) \wedge (\overline{x} \vee \overline{y} \vee \overline{A})$
3	$(y \vee A) \wedge (\overline{y} \vee A) \rightarrow (A)$	Génération de remplaçant pour $(y \vee A)$ $(\overline{x} \vee y \vee A) \wedge (x \vee y \vee A) \rightarrow (y \vee A)$ Remplaçant pour $(y \vee A)$ généré $(y \vee A) \wedge (\overline{y} \vee A) \rightarrow (A)$	$(\overline{x} \vee y \vee \overline{A}) \wedge (\overline{x} \vee \overline{y} \vee \overline{A}) \wedge (A)$

Tableau 5.1. – Adaptation du motif en diamant (x, y, A) grâce à l'algorithme de génération de remplaçants

5.7. Conclusion

Dans ce chapitre, on s'est intéressé à adapter n'importe quelle réfutation par résolution afin d'en déduire une max-réfutation. En particulier, on a proposé des adaptations linéaires dans les cas *tree-like regular*, *tree-like* et *semi-tree-like*. Ces résultats sont basés sur l'ajout de la règle du *split* pour adapter les réfutations *tree-like regular*, ce qui permet de transformer ces réfutations pour que plus aucune clause ne soit utilisée plusieurs fois en tant que prémisse d'une étape de résolution. On a aussi étendu notre adaptation au cas général, même si cette fois-ci l'adaptation peut nécessiter une augmentation au pire exponentielle de la taille de la réfutation. Ces résultats sont résumés dans le tableau 5.2, incluant le cas *read-once* qui était déjà connu.

Réfutation par résolution	**Taille de la réfutation Max-SAT**
Read-once	Linéaire
Tree-like regular	Linéaire
Tree-like	Linéaire
Semi-tree-like	Linéaire
Cas général	Exponentielle

Tableau 5.2. – Résultats sur l'adaptation des réfutations par résolution pour Max-SAT

Ces résultats ont permis de supprimer un blocage théorique sur l'adaptation des réfutations par résolution pour Max-SAT. Les conséquences d'une telle adaptation peuvent être notamment de permettre la génération de certificats pour Max-SAT, conséquences que nous explorons dans le chapitre suivant. L'étude du cas exponentiel reste cependant à compléter. En effet, si notre adaptation dans le cas général provoque l'augmentation exponentielle de la taille de la réfutation, il reste à identifier la raison de ce résultat. Est-t-il est impossible de trouver une adaptation qui soit polynomiale pour n'importe quelle réfutation par résolution ? Ou, au contraire, existerait-t-il une adaptation qui soit polynomiale dans le cas général ?

6. MS-Builder : un générateur de certificats pour Max-SAT

Sommaire

6.1. Introduction

Dans ce chapitre, on utilise les travaux du chapitre précédent, sur l'adaptation des réfutations par résolution pour Max-SAT, afin de générer des certificats pour les instances Max-SAT de la littérature. Un certificat pour le problème Max-SAT sera défini comme étant une transformation de la formule initiale en un équivalent contenant une sous-formule satisfiable et des clauses vides, plus un modèle pour la sous-formule satisfiable. Pour générer des certificats pour le problème Max-SAT, on va itérativement faire appel à un oracle SAT pour obtenir une réfutation par résolution, l'adapter en utilisant les travaux du chapitre 5 afin d'obtenir une max-réfutation, et l'appliquer sur la formule courante. Ce traitement sera répété jusqu'à ce que l'appel à l'oracle SAT permette de déduire que la formule courante (en ignorant les clauses vides) est satisfiable, avec pour preuve un modèle pour cette sous-formule.

Ce chapitre est découpé en plusieurs parties. En préambule, on présente dans la section 6.2 une amélioration de l'adaptation des réfutations par résolution pour le cas particulier des clauses unitaires utilisées plusieurs fois dans la réfutation. On définit ensuite ce qu'est un certificat pour le problème Max-SAT ainsi que deux outils, MS-Builder et MS-Checker, permettant respectivement de générer et de vérifier un certificat Max-SAT. Le fonctionnement de MS-Builder est en particulier illustré sur un exemple. Ensuite, on détaille les résultats des expérimentations de MS-Builder et MS-Checker sur les instances de l'*évaluation Max-SAT 2020* [Bac+20]. Enfin, on

conclut dans la section 6.5.

Les travaux présentés dans ce chapitre ont fait l'objet d'une publication internationale à la conférence *SAT 2021* dans l'article *A Proof Builder for Max-SAT* [PCH21a].

6.2. Réparation de la propagation unitaire

Dans cette section, partant du principe que les réfutations par résolution seront obtenues grâce à l'appel à un oracle SAT, on va s'intéresser à apporter une amélioration bien particulière aux réfutations par résolution générées suite à un appel à un oracle SAT. Cette amélioration porte sur l'impact de la propagation unitaire sur la forme de la réfutation par résolution obtenue. Calculer des réfutations par résolution en faisant appel à un oracle SAT peut causer le calcul de réfutations non *read-once* à cause des effets de la propagation unitaire. En effet, détecter une clause unitaire et propager le littéral correspondant peut être vu comme utiliser cette clause unitaire dans plusieurs étapes de résolution, étapes qui correspondent à la propagation unitaire du littéral correspondant. On s'intéressera ici à modifier la réfutation pour que plus aucune clause unitaire ne soit utilisée plusieurs fois en tant que prémisse d'une étape de résolution. Pour cela, lorsqu'une clause unitaire est utilisée dans plusieurs résolutions, on supprime son impact et on le réinjecte à la fin de la réfutation. Cette stratégie fonctionne lorsque la réfutation est dite *basée sur la propagation unitaire*, ce qui correspond au fait que, lorsqu'un littéral est propagé par propagation unitaire, il l'est complètement et la variable correspondante ne réapparaît plus ensuite.

Définition 6.1 (Réfutation basée sur la propagation unitaire)**.** *Soit P une réfutation par résolution. On dit que P est basée sur la propagation unitaire si, pour toute clause unitaire c dans P, toute branche partielle de la réfutation depuis c jusqu'à $\square$ ne contient aucune irrégularité*[1] *sur la variable $x \in c$.*

Exemple 6.1. *La réfutation par résolution de la figure 2.2 est basée sur la propagation unitaire. En particulier, la clause (x_1) est unitaire et utilisée plusieurs fois, mais une fois la variable x_1 éliminée par les deux résolutions sur la clause (x_1), la variable x_1 n'apparaît plus dans la réfutation.*

Pour l'implémentation de l'outil MS-Builder permettant de générer des certificats pour le problème Max-SAT, on utilisera comme hypothèse que les réfutations par résolution calculées sont *basées sur la propagation unitaire*, hypothèse qui se vérifiera dans les expérimentations puisque, sur l'ensemble des expérimentations effectuées, toutes les réfutations par résolution calculée étaient *basées sur la propagation unitaire*. Suivant l'idée présentée avant, on va maintenant proposer une méthode pour transformer n'importe quelle réfutation par résolution *basée sur la propagation unitaire* afin qu'il n'existe plus aucune clause unitaire utilisée dans plusieurs étapes de résolution.

1. On rappelle qu'une irrégularité est une branche de la réfutation contenant plusieurs résolutions sur la même variable.

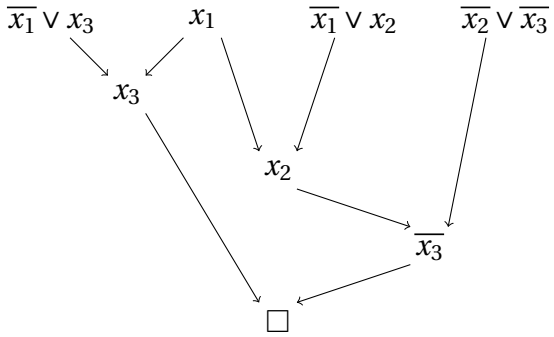


FIGURE 2.2. – Réfutation par résolution basée sur la propagation unitaire (cf page 39)

Théorème 6.1. *Soit ϕ une formule insatisfiable et P une réfutation par résolution de ϕ basée sur la propagation unitaire. Il existe une réfutation par résolution P' de ϕ telle qu'aucune clause utilisée dans plusieurs étapes de résolution n'est unitaire.*

Démonstration. Soit ϕ une formule insatisfiable et P une réfutation par résolution de ϕ *basée sur la propagation unitaire*. On démontre que si P possède $k > 0$ clauses unitaires utilisées dans plusieurs étapes de résolution, alors il existe une réfutation P_2 de ϕ qui possède $k-1$ clauses unitaires utilisées dans plusieurs étapes de résolution. Pour cela, on considère la réfutation P et la dernière clause unitaire c utilisée dans plusieurs étapes de résolution. Posons sans perte de généralité que $c = (x)$. Comme P est *basée sur la propagation unitaire*, alors en supprimant les étapes de résolution contenant la clause c comme prémisse et en remplaçant chaque résolvante d'une étape de résolution disparue par l'autre prémisse de la même résolution, on obtient que la séquence de résolutions qui conduisait à $\square$ conduit désormais à $\overline{x}$. Il suffit de réinjecter une étape de résolution entre la clause c et la clause $c' = \overline{x}$ pour obtenir une réfutation par résolution de ϕ contenant exactement $k-1$ clauses unitaires utilisées dans plusieurs étapes de résolution. En répétant cette transformation, on obtient une réfutation par résolution P' de ϕ telle qu'aucune clause utilisée dans plusieurs étapes de résolution n'est unitaire. $\square$

La transformation proposée dans la preuve permet de garantir, après pré-traitement, que toute les réfutations par résolution ne contiendront aucune clause unitaire problématique. En particulier, si une réfutation par résolution était non *read-once* uniquement à cause des clauses unitaires, cette transformation permet de la rendre *read-once*. On appellera *semi-read-once* cette classe de réfutations.

Définition 6.2 (Réfutation par résolution *semi-read-once*)**.** *Une réfutation par résolution de ϕ est dite semi-read-once si elle est basée sur la propagation unitaire et si chaque clause utilisée en tant que prémisses de plusieurs étapes de résolution est une clause unitaire.*

Exemple 6.2. *La réfutation par résolution de la figure 2.2 est semi-read-once.*

Corollaire 6.1. *Soit ϕ une formule insatisfiable. S'il existe une réfutation semi-read-once de ϕ, alors il existe une réfutation read-once de ϕ.*

Démonstration. Il suffit d'appliquer la transformation de la preuve du théorème 6.1 pour obtenir la réfutation read-once. □

Exemple 6.3. *On considère la réfutation tree-like regular de la figure 2.2. Cette réfutation est semi-read-once. Pour la rendre read-once, on supprime les étapes de résolution impliquant la clause* (x_1) *et on ajoute une étape de résolution impliquant cette clause à la fin de la réfutation, ce qui permet d'obtenir la figure 6.1.*

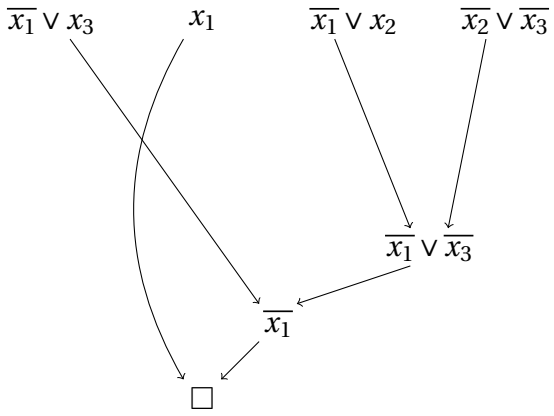


FIGURE 6.1. – Réparation d'une réfutation *semi-read-once* en réfutation *read-once*

6.3. MS-Builder & MS-Checker

Dans cette section, on présente un outil, MS-Builder, permettant de générer des certificats pour le problème Max-SAT. Cet outil est basé sur l'adaptation des réfutations par résolution pour Max-SAT présentée dans le chapitre 5, puis améliorée dans le cas des clauses unitaires dans la section 6.2. On considère ici les certificats pour le problème Max-SAT comme étant la composition d'une transformation depuis la formule initiale vers une formule décomposable en un ensemble de clauses vides et une sous-formule satisfiable, plus un modèle pour la sous-formule satisfiable. Le fonctionnement de MS-Builder peut être décrit par l'algorithme 6.1, dont la correction et la terminaison est garantie par la validité de l'adaptation présentée dans le chapitre 5. MS-Builder reçoit en entrée un fichier contenant une formule, comme dans la figure 6.2, et exporte un certificat, comme dans la figure 6.3.

La formule doit être écrite dans le format standard WCNF (l'ancien ou le nouveau) [Bac+20]. La formule (sous le nouveau format) est constituée de lignes représentant les clauses, chaque ligne représentant une clause commençant par le poids de la clause ('h' si la clause est dure) suivie des littéraux de la clause (le littéral x_i est représenté par le nombre i alors que le littéral $\overline{x_i}$ est représenté par le nombre $-i$. La formule peut aussi contenir des commentaires, qui sont des lignes devant commencer par la lettre 'c'.

Algorithme 6.1 MS-Builder

Entrée : Formule ϕ
Sortie : Certificat Max-SAT c pour ϕ

```
(T, opt) ← (∅, 0)
tant que φ est insatisfiable faire
    RP ← calculer_réfutation_par_résolution(φ)
    MRP ← adapter_réfutation_par_résolution_pour_Max-SAT(RP)
    φ ← appliquer_réfutation_Max-SAT(φ, MRP)
    (φ, opt) ← supprimer_clauses_vides(φ, opt)
    T ← T.MRP
I ← calculer_modèle(φ)
retourner (T, opt, I)
```

Le certificat doit commencer par une séquence de transformations, chaque étape de transformation étant écrite sur une ligne. Une étape de transformation doit commencer par la lettre 't', suivie du nom abrégé de la règle utilisée (par exemple `msres` pour la max-résolution et `split` pour le *split*), puis des prémisses de la transformation (entre '< >'). Pour le *split*, un paramètre, qui est la variable pivot, est spécifié après le nom. Ensuite, le certificat doit contenir une ligne commençant par la lettre 'o' avec l'optimum de la formule. Enfin, il doit se terminer par une ligne commençant par le lettre 'v' avec une interprétation des variables permettant de satisfaire la sous-formule satisfiable obtenue après l'application des transformations. Le certificat peut également contenir des commentaires, qui sont des lignes devant commencer par la lettre 'c'.

```
c formule de l'exemple 6.3
1 -1 3 0
1 1 0
1 -1 2 0
1 -2 -3 0
```

FIGURE 6.2. – Format de la formule

```
t msres < 1 -1 -2 | 1 -2 -3 >
t msres < 1 -1 3 | 1 -1 -3 >
t msres < 1 1 | 1 -1 >
o 1
v 000
```

FIGURE 6.3. – Format du certificat

MS-Checker reçoit en entrée deux fichiers, le premier contenant une formule et le second contenant un certificat Max-SAT pour cette formule. Pour chaque transformation du certificat, MS-Checker vérifie que la transformation est valide et que la ou les prémisse(s) appartiennent bien à la formule courante, puis applique la transformation. Enfin, il vérifie que le nombre de clauses vides calculées est bien égal à l'optimum du certificat et que l'interprétation proposée est bien un modèle pour la sous-formule finale.

Exemple 6.4. *On considère la formule* $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2}) \wedge (\overline{x_3}) \wedge (x_2 \vee x_3)$, *écrite dans le format WCNF dans la figure 6.4.*

```
c formule de l'exemple 6.4
1 -1 3 0
1 1 0
1 -1 2 0
1 -2 -3 0
1 -2 0
1 -3 0
1 2 3 0
```

FIGURE 6.4. – Formule sous format WCNF

Un premier appel à un oracle SAT indique que la formule courante est insatisfiable avec la réfutation par résolution de la figure 6.5, qui est adaptée pour Max-SAT en la réfutation Max-SAT de la figure 6.6.

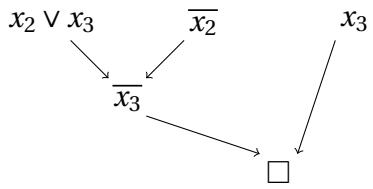


FIGURE 6.5. – Réfutation

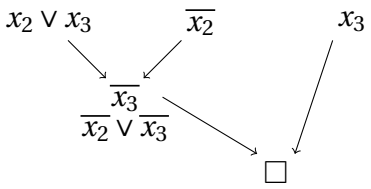


FIGURE 6.6. – Max-réfutation

Après application de la max-réfutation de la figure 6.6, la formule est maintenant $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge \square$. *Un deuxième appel à un oracle SAT indique que la formule courante (sauf les clauses vides) est insatisfiable avec la réfutation par résolution de la figure 2.2, qui est adaptée pour Max-SAT en la réfutation Max-SAT de la figure 6.1.*

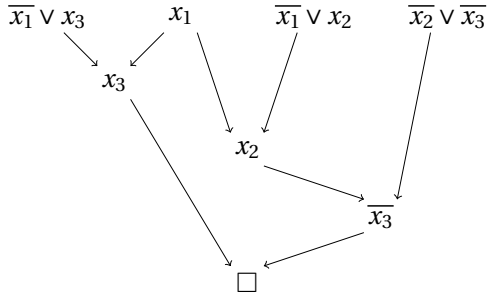


FIGURE 2.2. – Réfutation (cf page 39)

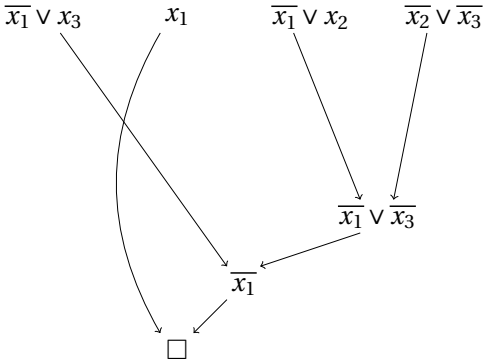


FIGURE 6.1. – Max-réfutation (cf page 93)

Après application de la max-réfutation de la figure 6.1, la formule est maintenant $\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge \square \wedge \square$. *Un troisième et dernier appel à un oracle*

SAT indique que la formule courante (sauf les clauses vides) est satisfiable en posant $x_1 = x_2 = x_3 = 0$. *MS-Builder retourne ici le certificat de la figure 6.7.*

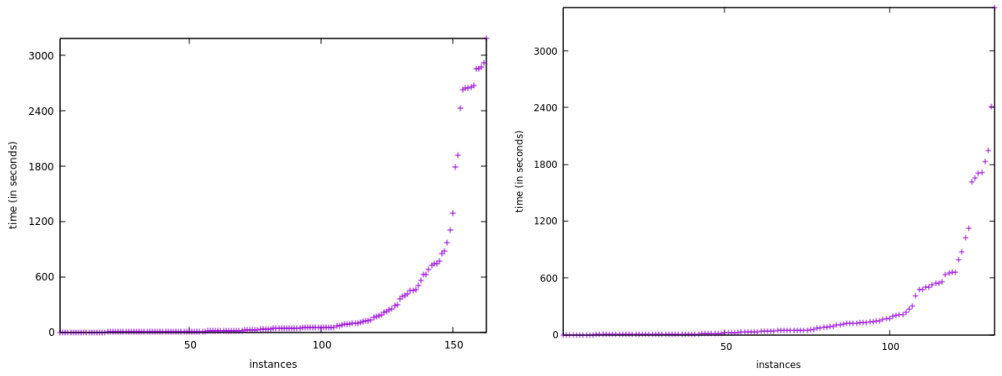
```
c certificat de l'exemple 6.4
c première réfutation read-once
t msres < 1 2 3 | 1 -2 >
t msres < 1 3 | 1 -3 >
c seconde réfutation semi-read-once
t msres < 1 -1 2 | 1 -2 -3 >
t msres < 1 -1 3 | 1 -1 -3 >
t msres < 1 1 | 1 -1 >
o 2
v 000
```

FIGURE 6.7. – Certificat

6.4. Expérimentations

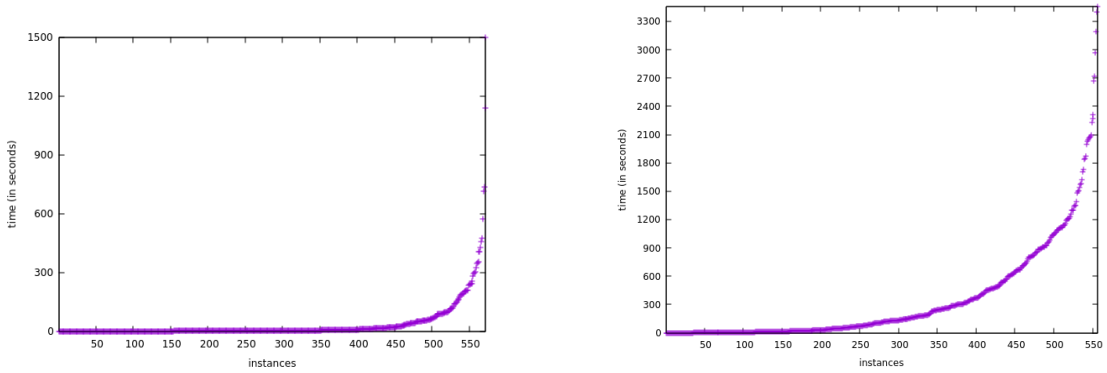
Dans cette section, on présente les résultats des expérimentations de MS-Builder et MS-Checker sur les instances de la compétition Max-SAT 2020 [Bac+20]. Les expérimentations ont été faites sur des serveurs Dell PowerEdge M620 avec des processeurs Intel XeonSilver E5-2609 (2.5⌣2.6 GHz) utilisant le système d'exploitation Ubuntu 18.04. Chaque test a eu un temps limite de 1 heure et une taille mémoire maximum de 16 gigaoctets. Les réfutations par résolution ont été calculées en utilisant le solveur Booleforce [Bie10] et le vérificateur Tracecheck [Bie06].

MS-Builder a réussi à construire des certificats Max-SAT complets pour 163 instances sur 576 instances non pondérées et pour 132 instances sur 600 instances pondérées. Le temps d'exécution pour la construction des certificats est représenté dans la figure 6.8. On peut observer que les instances pondérées semblent plus difficiles à résoudre que les instances non pondérées. MS-Checker a réussi à vérifier 575 certificats (complets ou incomplets) sur 576 dans le cas non pondéré et 557 certificats (complets ou incomplets) sur 600 dans le cas pondéré. La vérification de certificats est bien-sûr plus facile que la construction de certificats sur la plupart des formules. Quelques exceptions existent quand la formule a un nombre important de clauses (initialement ou après l'application de transformations), ce qui peut rendre difficile l'opération linéaire consistant à chercher une clause dans une formule, opération utilisée itérativement dans MS-Checker (une structure de données adaptée à cette opération doit donc être utilisée). Le temps de vérification des certificats est représenté dans la figure 6.9.



Instances non pondérées Instances pondérées

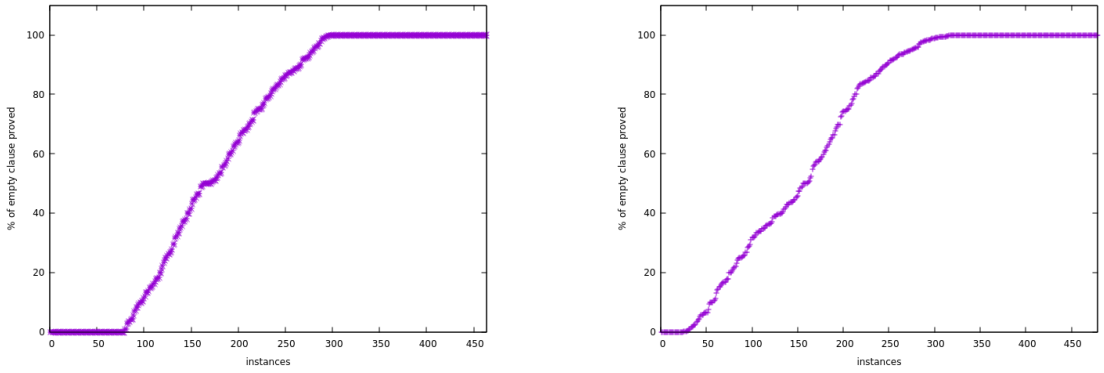
FIGURE 6.8. – Temps d'exécution (en secondes) pour la construction de certificats par instance



Instances non pondérées Instances pondérées

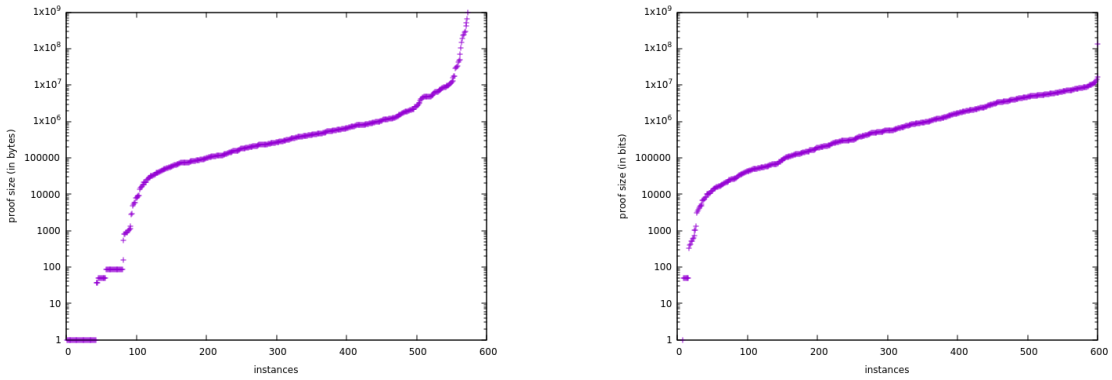
FIGURE 6.9. – Temps d'exécution (en secondes) pour la vérification de certificats par instance

MS-Builder a réussi à construire au moins la moitié du certificat (par rapport à la valeur de l'optimum Max-SAT de la formule) pour 302 instances non pondérées sur 463 et sur 324 instances pondérées sur 489 dont l'optimum Max-SAT est connu, comme illustré dans la figure 6.10 qui représente le pourcentage de clauses vides certifiées par instance. La taille des certificats (complets ou partiels) varient de quelques octets à 1 gigaoctets comme illustré dans la figure 6.11. Remarquez que certaines instances sont si dures qu'aucune clause vide n'a pu être certifiée après 1 heure de temps d'exécution. De l'autre côté, certaines instances faciles avec un optimum Max-SAT de 1 possèdent des certificats très petits.



Instances non pondérées Instances pondérées

FIGURE 6.10. – Pourcentage de clauses vides certifiées par instance



Instances non pondérées Instances pondérées

FIGURE 6.11. – Taille du certificat (échelle logarithmique) calculé par instance

Finalement, on peut observer dans la table 6.1 que les réfutations sont majoritairement *read-once* et *semi-read-once* pour les instances non pondérées. Cependant, il y a beaucoup d'instances pour lesquelles, si les premières réfutations sont de petites tailles et faciles à traiter, elles deviennent progressivement de plus en plus grandes et difficiles à traiter. Les dernières réfutations à traiter dans chaque instance sont souvent les plus difficiles et les plus grandes. Par conséquent, le temps limite d'exécution arrive à souvent son terme pour ces dernières réfutations. En revanche, il y a une grande différence quand on observe la fréquence d'apparition des réfutations *read-once* et *semi-read-once* dans les instances pondérées, qui est plus faible : il y a seulement 56.83 % de réfutations *read-once* ou *semi-read-once* dans les instances pondérées contre 95.8 % dans les instances non pondérées. Cette différence peut représenter une explication sur la difficulté à résoudre les instances pondérées par rapport aux instances non pondérées.

Classe de réfutation par résolution	Instances non pondérées		Instances pondérées	
	Nombre	Pourcentage	Nombre	Pourcentage
read-once	169 239	83.7 %	139 816	36.19 %
semi-read-once	24 556	12.1 %	79 750	20.64 %
tree-like regular	2 879	1.4 %	16 137	4.17 %
tree-like	1 795	0.9 %	108 014	27.96 %
unrestricted	3 799	1.9 %	42 642	11.04 %

Tableau 6.1. – Répartition des classes de réfutation rencontrées dans l'intégralité des tests

6.5. Conclusion

Dans ce chapitre, on a utilisé les travaux du chapitre 5 dans deux outils, MS-Builder et MS-Checker, afin de générer et de vérifier des certificats pour le problème Max-SAT. MS-Builder est basé sur des appels itératifs à un oracle SAT, ce qui permet d'obtenir des réfutations qui seront ensuite adaptées pour Max-SAT et appliquées sur la formule courante. MS-Builder a permis de générer des certificats Max-SAT pour un nombre significatif d'instances de la littérature.

Les travaux futurs concernent l'amélioration de la vitesse d'exécution de l'outil ainsi que de la taille des certificats exportés. Il peut notamment être intéressant d'ajouter l'algorithme de génération de remplaçant, ou une nouvelle méthode, afin d'avoir une deuxième méthode d'adaptation des réfutations par résolution implémentée dans l'outil et d'observer s'il est possible d'améliorer ses performances. Il peut également être pertinent, comme cela a été fait pour SAT, d'essayer de réduire la taille des certificats exportés en réfléchissant aux informations relatives aux certificats qui pourraient être allégées.

Dans le chapitre suivant, on s'intéressera à une problématique plus large que les preuves pour le problème Max-SAT. On se demandera, étant donnée une formule initiale et une information (sous la forme d'une clause ou d'une autre formule), s'il est possible de déduire cette information en utilisant des règles d'inférence qui préservent l'équivalence Max-SAT et, si la réponse est positive, comment générer une telle déduction.

7. Déduction de clauses et de formules dans Max-SAT

Sommaire

7.1. Introduction

Ce chapitre concerne des transformations Max-SAT qui dépassent le cadre strict de la certification d'une solution pour le problème Max-SAT. On s'intéresse ici au problème qui consiste, étant donnée une formule initiale et une information à déduire (sous la forme d'une clause ou d'une formule), à construire des transformations depuis la formule initiale jusqu'à une formule équivalente qui contienne cette information. Le problème étudié est, dans un sens, plus général que le problème Max-SAT, puisqu'il suffit de trouver un moyen de déduire k clauses vides et une impossibilité de déduire une $(k+1)$-ème clause vide pour certifier que l'optimum Max-SAT d'une formule est k.

Ce chapitre est découpé en plusieurs parties. Tout d'abord, on définit dans la section 7.2 le concept d'explicabilité, qui permet de caractériser, à partir d'une formule initiale, les clauses ou formules pouvant être déduites par des transformations qui préservent l'équivalence Max-SAT. Ensuite, on s'intéresse dans la section 7.3 aux systèmes de preuve pour Max-SAT et à leur capacité, étant donnée une clause (ou une formule) déductible à partir d'une formule initiale, à être capable de déduire cette information en utilisant les règles d'inférence du système. Si n'importe quelle information (clause ou formule) déductible à partir d'une formule initiale l'est en utilisant un système

de preuve, on dira qu'il est inférentiellement complet, sinon on dira qu'il est inférentiellement incomplet. Comme la max-résolution est inférentiellement incomplet, on présentera plusieurs systèmes de preuve inférentiellement complets, dont le système **ExC**, basé sur le concept d'explicabilité et qui se compose des règles *symmetric cut* et expansion (définie plus loin). Dans la section 7.4, on présentera l'algorithme d'explication, qui permet, étant donnée une clause ou une formule explicable depuis une formule initiale, d'exhiber une transformation utilisant **ExC** et permettant, à partir de la formule initiale, de déduire une transformation menant à cette clause ou à cette formule. On verra dans la section 7.5 comment l'algorithme d'explication permet, en particulier, de construire des preuves pour le problème Max-SAT. Enfin, on adaptera ces travaux pour le cas partiel pondéré dans la section 7.6 et on conclura dans la section 7.7.

Les travaux présentés dans ce chapitre ont fait l'objet d'une publication internationale à la conférence *ICTAI 2021* dans l'article *Inferring Clauses and Formulas in Max-SAT* [PCH21d].

7.2. Explicabilité des clauses et des formules

Dans cette section, on définit la notion de clauses et de formules explicables. Une clause ou une formule est explicable s'il existe une séquence de transformations préservant l'équivalence Max-SAT, appelée explication, permettant la déduction de la clause ou de la formule à partir d'une formule initiale. On commence cette section en donnant la définition formelle d'une clause explicable. Ensuite, on présente deux résultats intermédiaires dans les propositions 7.1 et 7.2. Le premier démontre que les clauses qui sont sous-sommées par au moins une clause de la formule sont explicables, alors que le second démontre que les clauses qui s'opposent à toutes les clauses de la formule ne le sont pas. Grâce à ces deux résultats, on propose une caractérisation des clauses explicables dans le théorème 7.1, caractérisation qui sera utilisée dans la section 7.4 pour proposer un algorithme permettant de construire des explications de clauses. Enfin, on généralise ces notions pour le cas des formules et on les caractérise dans le théorème 7.2.

Définition 7.1 (Clause explicable)**.** *Étant données une formule ϕ et une clause non tautologique c, on dit que c est explicable dans ϕ s'il existe une formule ϕ' telle que $\phi \equiv c \wedge \phi'$; sinon on dit que c est inexplicable dans ϕ.*

Intuitivement, les clauses explicables représentent l'information qui peut être déduite à partir d'une formule. Si on exhibe une transformation T depuis une formule ϕ jusqu'à une formule équivalente $c \wedge \phi'$, on dira que T est une explication de c dans ϕ. Cette transformation est la preuve que l'information voulue, ici une clause, peut être inférée à partir de la formule. Remarquons que, pour une formule ϕ, n'importe quelle clause $c \in \phi$ est trivialement explicable.

Définition 7.2 (Explication d'une clause)**.** *Soit ϕ une formule et c une clause explicable dans ϕ. Une explication de c dans ϕ est une transformation depuis ϕ jusqu'à un équivalent $c \wedge \phi'$ où ϕ' est une formule.*

Maintenant que l'on a défini les clauses explicables, on va les caractériser afin de proposer, dans la section 7.4, un algorithme pour construire des explications de clauses ou pour réfuter leur explicabilité. La proposition 7.1 affirme que les clauses sous-sommées sont explicables et la proposition 7.2 affirme que les clauses opposées à toutes les clauses de la formule ne le sont pas. Ces deux résultats simples permettent d'énoncer dans le théorème 7.1 une caractérisation des clauses explicables qui sera ensuite utilisée pour proposer un algorithme d'explication de clauses dans la section 7.4.

Proposition 7.1. *Soit ϕ une formule et c une clause non tautologique. Si $\exists c' \in \phi$ tel que c' sous-somme c alors c est explicable dans ϕ.*

Démonstration. Supposons que $\exists c' \in \phi$ tel que c' sous-somme c. Par conséquent, on a $c = c' \vee A$ où $A = a_1 \vee a_2 \vee \cdots \vee a_n$. Soit $\phi_2 = \bigwedge_{1 \leq i \leq n} (c' \bigvee_{1 \leq j < i} a_j \vee \overline{a_i})$. On démontre que $c' \equiv \phi_2 \wedge c$. Pour toute affectation des variables I, on a deux cas possibles :

— Soit I satisfait c' et dans ce cas, il existe un littéral $l \in c'$ qui est satisfait par I. Comme ce littéral est dans chaque clause de $\phi_2 \wedge c$, on a $cout_I(\phi_2 \wedge c) = 0 = cout_I(c')$.

— Soit I falsifie c' et dans ce cas, il y a exactement une clause de $\phi_2 \wedge c$ qui est falsifiée. En effet, si on considère l'affectation de chaque variable $var(a_1), ..., var(a_n)$ de I, par construction, exactement une clause de $\phi_2 \wedge c$ est opposé à toutes les affectations des variables $var(a_1), ..., var(a_n)$ de I. Cette clause est falsifiée par I alors que les autres sont satisfaites et on a donc $cout_I(\phi_2 \wedge c) = 1 = cout_I(c')$.

Par conséquent, on a $\phi \equiv (\phi \setminus \{c'\}) \wedge c' \equiv (\phi \setminus \{c'\}) \wedge \phi_2 \wedge c \equiv \phi' \wedge c$ où $\phi' = (\phi \setminus \{c'\}) \wedge \phi_2$. On en conclut que c est explicable dans ϕ. □

Proposition 7.2. *Soit ϕ une formule et c une clause non tautologique. Si $\forall c' \in \phi$, c' s'oppose à c alors c n'est pas explicable dans ϕ.*

Démonstration. Supposons que $\forall c' \in \phi$, c' s'oppose à c. On considère l'interprétation $I = \{\overline{l} \mid l \in c\}$. Comme chaque clause de ϕ s'oppose à c, I est un modèle de ϕ. Par conséquent, on a $cout_I(\phi) = 0$, et, par définition de I, c est falsifiée par I. En revanche, on a $cout_I(c) = 1$ et pour chaque formule ϕ', $cout_I(c \wedge \phi') > cout_I(\phi)$. On en conclut que c n'est pas explicable dans ϕ. □

Théorème 7.1. *Soit une formule ϕ et une clause c non tautologique. La clause c est explicable dans la formule ϕ si et seulement si $\exists c' \in \phi$, c' qui ne s'oppose pas à c et $\forall x \notin var(c)$, $c \vee x$ et $c \vee \overline{x}$ sont explicables dans ϕ.*

Démonstration. Premièrement, supposons que c est explicable dans ϕ, c'est à dire qu'il existe une formule ϕ' telle que $\phi \equiv \phi' \wedge c$. La contraposée de la proposition 7.2 nous indique que $\exists c' \in \phi, c'$ qui ne s'oppose pas à c. Soit une variable $x \notin var(c)$ et

une interprétation I. Comme $cout_I(c) = cout_I((c \vee x) \wedge (c \vee \overline{x}))$ [LR20b], on a $\phi' \wedge c \equiv \phi' \wedge (c \vee x) \wedge (c \vee \overline{x})$ et on conclut que $\forall x \notin var(c)$, $c \vee x$ et $c \vee \overline{x}$ sont explicables dans ϕ.

Réciproquement, supposons que $\exists c' \in \phi$ tel que c' ne s'oppose pas à c et que $\forall x \notin var(c)$, $c \vee x$ et $c \vee \overline{x}$ sont explicables dans ϕ. Il y a deux cas possibles :

— Soit $\nexists x \notin var(c)$, c'est à dire que c contient toutes les variables de ϕ. Par conséquent, c' ne s'oppose pas à c et surtout, c' sous-somme c. On en déduit grâce à la proposition 7.1 que c est explicable dans ϕ.

— Sinon, on considère la formule M telle que pour toute interprétation I, la clause $(\bigvee_{l \in I} \overline{l})$ apparaît $cout_I(\phi)$ fois dans M. Par construction de M, pour toute interprétation I, on a $cout_I(\phi) = cout_I(M)$ et donc $\phi \equiv M$. Intuitivement, M est la formule maximale équivalente à ϕ, c'est à dire la formule équivalente à ϕ qui contient le nombre maximum de clauses non tautologiques. De manière similaire, on considère les formules maximales équivalentes $(c \vee x)$ et $(c \vee \overline{x})$. On note M_1 (respectivement M_2) la formule telle que pour toute interprétation I, la clause $(\bigvee_{l \in I} \overline{l})$ apparaît $cout_I(c \vee x)$ (respectivement $cout_I(c \vee \overline{x})$) fois dans M_1 (respectivement M_2). Par définition de M_1 (respectivement M_2), on a $M_1 \equiv (c \vee x)$ (respectivement $M_2 \equiv (c \vee \overline{x})$). Comme $(c \vee x)$ (respectivement $(c \vee \overline{x})$) est explicable dans ϕ, on a $M_1 \subseteq M$ (respectivement $M_2 \subseteq M$). De plus, comme $(c \vee x)$ s'oppose à $(c \vee \overline{x})$, on a $M_1 \cap M_2 = \emptyset$. On en déduit que $\phi \equiv M \equiv (M \setminus (M_1 \cup M_2)) \wedge M_1 \wedge M_2 \equiv (M \setminus (M_1 \cup M_2)) \wedge (c \vee x) \wedge (c \vee \overline{x}) \equiv (M \setminus (M_1 \cup M_2)) \wedge c$. Par conséquent, c est explicable dans ϕ.

Ayant démontré les deux sens de l'équivalence, on conclut que c est explicable dans ϕ si et seulement si $\exists c' \in \phi$, c' ne s'oppose pas à c et $\forall x \notin var(c)$, $c \vee x$ et $c \vee \overline{x}$ sont explicables dans ϕ. □

Maintenant, on va étendre la notion d'explicabilité aux formules.

Définition 7.3 (Formule explicable)**.** *Étant données deux formules ϕ_1 et ϕ_2, on dit que ϕ_2 est explicable dans ϕ_1 s'il existe une formule ϕ' telle que $\phi_1 \equiv \phi_2 \wedge \phi'$; sinon on dit que ϕ_2 est inexplicable dans ϕ_1.*

Définition 7.4 (Explication d'une formule)**.** *Soient deux formules ϕ_1 et ϕ_2 telles que ϕ_2 est explicable dans ϕ_1. Une explication de ϕ_2 dans ϕ_1 est une transformation qui préserve l'équivalence Max-SAT depuis ϕ_1 jusqu'à $\phi_2 \wedge \phi'$ où ϕ' est une formule*

Faisons la remarque qu'étant donnée deux formules, dire que la deuxième formule est explicable dans la première revient à dire qu'il est possible d'inférer la deuxième formule à partir de la première. Ainsi, deux formules sont équivalentes si et seulement si la première est explicable dans la seconde et si la seconde est explicable dans la première.

Proposition 7.3. *Deux formules ϕ_1 et ϕ_2 sont équivalentes si et seulement si ϕ_1 est explicable dans ϕ_2 et ϕ_2 est explicable dans ϕ_1.*

Démonstration. Si deux formules ϕ_1 et ϕ_2 sont équivalentes, alors il suffit de prendre la formule $\phi' = \emptyset$ et on a bien $\phi_1 \equiv \phi_2 \wedge \phi'$ et $\phi_2 \equiv \phi_1 \wedge \phi'$.

Si ϕ_1 est explicable dans ϕ_2 et ϕ_2 est explicable dans ϕ_1, alors il existe ϕ' et ϕ'' telles que $\phi_1 \equiv \phi_2 \wedge \phi'$ et $\phi_2 \equiv \phi_1 \wedge \phi''$. Par conséquent, $\phi_1 \equiv \phi_1 \wedge \phi' \wedge \phi''$ et donc $\phi' = \phi'' = \emptyset$ d'où ϕ_1 est Max-SAT-équivalent à ϕ_2. □

Intuitivement, expliquer une formule semble plus difficile qu'expliquer une clause. En effet, si deux clauses c_1 et c_2 sont explicables dans une formule, cela ne veut pas dire que $c_1 \wedge c_2$ est également explicable dans cette formule. Pourtant, on peut démontrer que cette intuition est fausse et qu'expliquer une formule revient à expliquer itérativement chacune de ces clauses.

Théorème 7.2. *Étant donnée deux formules ϕ_1 et $\phi_2 = c_1 \wedge ... \wedge c_m$, ϕ_2 est explicable dans ϕ_1 si et seulement si il existe une suite de formules $\langle \phi'_1, ..., \phi'_m \rangle$ telles que :*
— *Il existe une explication de c_1 depuis ϕ_1 jusqu'à ϕ'_1.*
— *$\forall i \in \{2, ..., m\}$, il existe une explication de c_i depuis $(\phi'_{i-1} \backslash \{c_{i-1}\})$ jusqu'à ϕ'_i.*

Démonstration. Supposons que ϕ_2 est explicable dans ϕ_1. On pose $\phi'_1 = \phi_2$ et $\forall i \in \{2, ..., m\}$, $\phi'_i = \phi'_{i-1} \backslash \{c_{i-1}\}$.
Réciproquement, par transition de $\equiv$, on a $\phi_1 \equiv \phi_2 \wedge (\phi'_m \backslash \{c_m\})$ et on conclut que ϕ_2 est explicable dans ϕ_1. □

Grâce au résultat du théorème 7.2, n'importe quelle méthode capable de construire des explications de clauses sera aussi capable de construire des explications de formules. Dans la section 7.4, on va proposer un algorithme basé sur la preuve du théorème 7.1 afin de construire des explications de clauses. Cet algorithme sera ensuite facilement étendu aux explications de formules en utilisant le résultat du théorème 7.2.

7.3. Systèmes de preuve inférentiellement complets

Dans cette section, on étudie les systèmes de preuve pour Max-SAT et leur complétude ou incomplétude vis-à-vis de l'explication de clauses et de formules. Remarquons tout d'abord que la max-résolution est inférentiellement incomplète [LR20b], c'est à dire qu'il existe des clauses explicables dans une formule qui ne peuvent être expliquées en utilisant uniquement la règle de la max-résolution. On présentera plus tard des systèmes de preuve inférentiellement complets et on étudiera les liens qui les unissent.

Définition 7.5 (Complétude inférentielle)**.** *Soit P un système de preuve pour Max-SAT. On dit que P est inférentiellement complet (ou complet pour l'explication) si, pour toute formule ϕ et pour toute clause explicable c dans ϕ, il existe une explication de c dans ϕ qui utilise uniquement les règles d'inférence de P.*

Comme la max-résolution est inférentiellement incomplète, on va étudier dans cette section plusieurs systèmes de preuve pour Max-SAT. En particulier, on étudiera la complétude inférentielle de ces systèmes et la capacité qu'ils ont à se simuler entre eux. On définit pour cela la simulation réfutationnelle et la simulation inférentielle. La simulation réfutationnelle, couramment utilisée pour comparer deux systèmes de preuve [LR20b ; Her+08], permet d'étudier la capacité qu'a un système de preuve à simuler n'importe quelle réfutation calculée en utilisant un autre système de preuve. Dans ce chapitre, on s'intéresse à des inférences qui dépassent la simple réfutation et qui permettent n'importe quelle transformation. On définit alors également la simulation inférentielle pour étudier la capacité qu'a un système de preuve à simuler n'importe quelle transformation calculée en utilisant un autre système de preuve. Clairement, tout système qui simule inférentiellement un autre système le simule également réfutationnellement.

Définition 7.6 (Simulation réfutationnelle)**.** *Soient P_1 et P_2 deux systèmes de preuve pour Max-SAT. On dit que P_1 r-p-simule (simule polynomiallement la réfutation dans) P_2 s'il existe une fonction calculable polynomiale f telle que pour toute réfutation Π de ϕ dans P_2, $f(\Pi)$ est une transformation de ϕ dans P_1.*

Définition 7.7 (Simulation inférentielle)**.** *Soient P_1 et P_2 deux systèmes de preuve pour Max-SAT. On dit que P_1 i-p-simule (simule polynomiallement l'inférence dans) P_2 s'il existe une fonction calculable polynomiale f telle que pour toute transformation Π de ϕ dans P_2 permettant de déduire une clause c, $f(\Pi)$ est une transformation de ϕ dans P_1 permettant de déduire c.*

Remarque 7.1. *Si un système P_1 i-p-simule un système P_2 et que P_2 i-p-simule P_1, on dira que P_1 et P_2 sont i-p-équivalents.*

Pour faire l'explication de clauses et de formules, on va en particulier s'intéresser au système de preuve, que l'on appellera **ExC**. La règle d'expansion est définie ci-dessous et peut être considérée à la fois comme l'adaptation pour Max-SAT de la règle d'affaiblissement pour SAT, ainsi que comme la généralisation de la règle du *split*. La motivation derrière l'introduction de la règle d'expansion est le résultat démontré dans la proposition 7.1. La règle d'expansion permet d'expliquer, en une seule étape d'inférence, n'importe quelle clause sous-sommée par une autre clause.

Définition 7.8 (Expansion)**.** *Étant donnée une clause $c = (A)$ avec A une disjonction de littéraux et $B = b_1 \vee ... \vee b_k$, la règle d'expansion appliquée sur c et B est définie de la manière suivante :*

$$\frac{c_1 = (A)}{\begin{array}{c} cc_1 = (A \vee \overline{b_1}) \\ cc_2 = (A \vee b_1 \vee \overline{b_2}) \\ \dots \\ cc_k = (A \vee b_1 \vee \dots \vee b_{k-1} \vee \overline{b_k}) \\ c_2 = (A \vee B) \end{array}}$$

Remarque 7.2. *Comme les autres règles d'inférence pour Max-SAT, la règle d'expansion remplace les prémisses par les conclusions.*

Remarque 7.3. *On note k-expansion l'application de la règle d'expansion sur une clause c et une disjonction de littéraux B telles que* $|B| = k$.

Maintenant, on va étudier la relation entre la règle d'expansion et la règle du *split* dans les propositions 7.4 et 7.5, puis on va démontrer que la règle d'expansion préserve l'équivalence Max-SAT dans la proposition 7.6.

Proposition 7.4. *Toute étape de split peut être simulée par une* 1*-expansion.*

Démonstration. Si on considère une transformation par *split* $(A) \vdash (x \vee A) \wedge (\overline{x} \vee A)$, il suffit de poser $B = x$ et l'application de la règle d'expansion sur la clause (A) et la disjonction de littéraux B simule l'application de la règle du *split* sur la clause (A) et sur la variable x. □

Proposition 7.5. *Toute k-expansion peut être simulée par k splits.*

Démonstration. Soit $(A) \vdash (A \vee \overline{b_1}) \wedge ... \wedge (A \vee b_1 \vee ... \vee b_{k-1} \vee \overline{b_k}) \wedge (A \vee B)$ une k-expansion. On applique la règle du *split* k fois pour simuler l'expansion comme ci-dessous :

$$
\begin{aligned}
(A) &\vdash (A \vee \overline{b_1}) \wedge (A \vee b_1) \\
&\vdash (A \vee \overline{b_1}) \wedge (A \vee b_1 \vee \overline{b_2}) \vee (A \vee b_1 \vee b_2) \\
&\vdash (A \vee \overline{b_1}) \wedge (A \vee b_1 \vee \overline{b_2}) \vee (A \vee b_1 \vee b_2 \vee \overline{b_3}) \vee (A \vee b_1 \vee b_2 \vee b_3) \\
&... \\
&\vdash (A \vee \overline{b_1}) \wedge ... \wedge (A \vee b_1 \vee ... \vee b_{k-1} \vee \overline{b_k}) \wedge (A \vee B)
\end{aligned}
$$

□

Proposition 7.6. *La règle d'expansion préserve l'équivalence Max-SAT.*

Démonstration. Comme la règle du *split* préserve l'équivalence Max-SAT [LR20b] et qu'elle peut simuler la règle de l'expansion (proposition 7.5), alors la règle d'expansion préserve l'équivalence Max-SAT. □

Pour construire des explications de clauses et de formules, on définit ci-dessous un système de preuve pour Max-SAT, que l'on nomme **ExC**, et qui est composé du *symmetric cut* et de l'expansion. On montre dans la proposition 7.7 qu'il est inférentiellement complet. Remarquez que ce résultat est aussi valide pour les systèmes de preuve **max-résolution + expansion**, **ResS = max-résolution + split** [LR20b] et **symmetric cut + split** [BL20]. On étudie également les relations entre ces différents systèmes. En particulier, on démontre dans la proposition 7.8 que **ExC** i-p-simule **symmetric cut + split**, **max-résolution** et **ResS**. On observe également qu'en présence de la règle d'expansion, il est possible d'affaiblir la max-résolution en la remplaçant par le *symmetric cut* et cela sans avoir un système de preuve plus faible. Cela peut être déduit du résultat énoncé dans la proposition 7.9 qui démontre que **ExC** est i-p-équivalent à **max-résolution + expansion**, c'est à dire que **ExC** i-p-simule **max-résolution + expansion** et que **max-résolution + expansion** i-p-simule **ExC**. Notez que les résultats des propositions 7.8 et 7.9 restent valides si on restreint le résultat aux simulations linéaires.

Définition 7.9 (ExC)**.** *Le système de preuve **ExC** est composée des deux règles symmetric cut et expansion.*

Proposition 7.7. ***ExC** est inférentiellement complet.*

Démonstration. Le système **ResS**={**max-résolution + split**} a été récemment démontré complet dans [LR20b]. Comme le système **symmetric cut + split** est capable de simuler le système **ResS** [BL20], alors il est également inférentiellement complet. Enfin, comme l'expansion simule le *split*, alors les systèmes **ExC** et **max-résolution + expansion** sont également inférentiellement complets. □

Proposition 7.8. ***ExC** i-p-simule les systèmes suivants :*
- ***max-résolution***
- ***symmetric cut + split***
- ***ResS** = **max-résolution + split***
- ***max-résolution + expansion***

Démonstration. Une étape de max-résolution peut être simulée en utilisant deux étapes d'expansion et une étape de *symmetric cut* comme ci-dessous :

$$\begin{array}{lcll} (x \vee A) & \vdash & (x \vee A \vee \overline{b_1}) \wedge \ldots \wedge (x \vee A \vee B) & \text{[expansion]} \\ (\overline{x} \vee B) & \vdash & (\overline{x} \vee B \vee \overline{a_1}) \wedge \ldots \wedge (\overline{x} \vee B \vee A) & \text{[expansion]} \\ (x \vee A \vee B) \wedge (\overline{x} \vee B \vee A) & \vdash & (A \vee B) & [\textit{symmetric cut}] \end{array}$$

Par conséquent, **ExC** i-p-simule **max-résolution** et **max-résolution + expansion**. Comme indiqué dans la proposition 7.4, chaque étape de *split* peut être simulée par une 1-expansion, donc **ExC** i-p-simule **symmetric cut + split**. En cumulant les deux informations (*split* simulé par une expansion et max-résolution par deux expansions et un *split*), on conclut également que **ExC** i-p-simule **ResS**. □

Proposition 7.9. ***ExC** est i-p-équivalent à **max-résolution + expansion**.*

Démonstration. Comme montré dans la proposition 7.8, **ExC** i-p-simule **max-résolution + expansion**. Réciproquement, **max-résolution + expansion** i-p-simule **ExC** car la règle du *symmetric cut* est un cas particulier de la max-résolution. □

On résume les résultats des propositions 7.8 et 7.9 vis-à-vis des relations entre **ExC** et les autres systèmes étudiés dans la figure 7.1. Dans la prochaine section, on va voir comment il est possible de construire des explications de clauses et de formules en utilisant **ExC**.

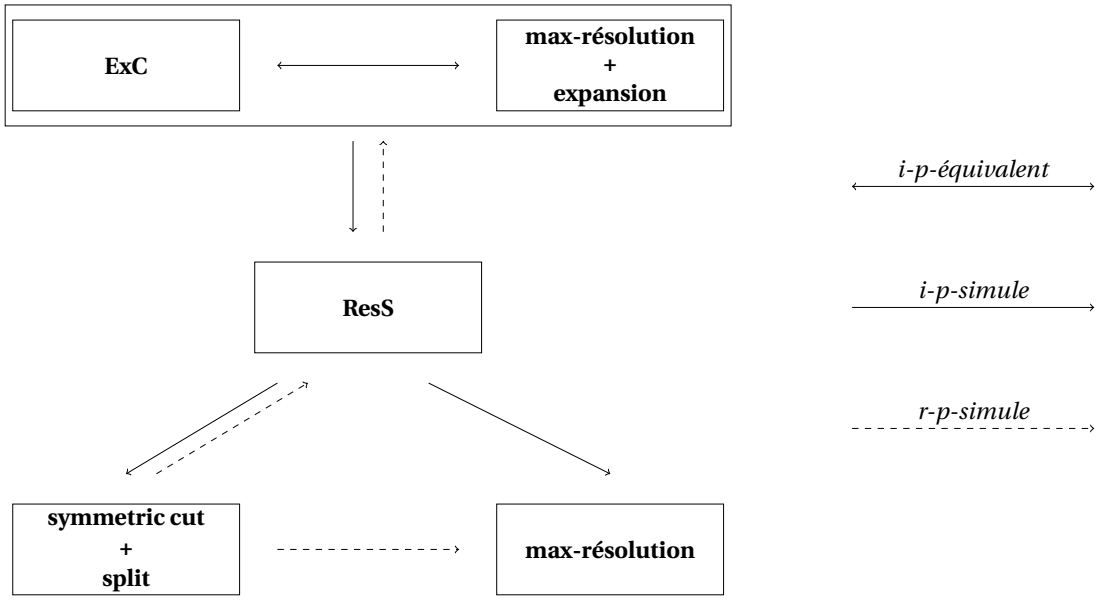


FIGURE 7.1. – Relations entre **ExC** et d'autres systèmes de preuve

7.4. L'algorithme d'explication

Dans cette section, on va utiliser la caractérisation du théorème 7.1 pour déduire une méthode capable de construire des explications de clauses. On généralisera ensuite cette méthode aux formules grâce à la caractérisation du théorème 7.2. L'idée est, étant donnée une formule ϕ et une clause c à expliquer, de construire l'explication de la clause c en construisant depuis sa fin la transformation qui a permis de déduire c à partir de ϕ. Tout d'abord, on démontre le cas de base qui est l'explication des clauses qui sont sous-sommées par d'autres clauses de la formule.

Proposition 7.10. *Soit ϕ une formule et c une clause non-tautologique. Si il existe $c' \in \phi$ tel que c' sous-somme c alors il existe une explication de c dans ϕ utilisant uniquement une étape d'expansion.*

Démonstration. Supposons que il existe $c' \in \phi$ tel que c' sous-somme c. La formule ϕ_2 dans la preuve de la proposition 7.1 correspond, par construction, à l'application de la règle d'expansion sur la clause c' et sur la disjonction de littéraux $B = c \setminus c'$. Cette application de la règle d'expansion est une explication de c dans ϕ. □

Maintenant, on démontre le résultat principal affirmant qu'il est possible d'exhiber une explication de n'importe quelle clause explicable en utilisant **ExC** et on fournit une borne sur le nombre d'étapes d'inférence de ces explications. Ces résultats restent valides, avec une borne supérieure différente, en remplaçant le *symmetric cut* par la max-résolution et/ou l'expansion par le *split*. En remplaçant l'expansion par le *split*, un facteur additionnel de n (n étant le nombre de variables de la formule) doit être ajoutée sur la borne supérieure calculée.

Théorème 7.3. *Soit ϕ une formule et c une clause non-tautologique explicable dans ϕ. Il existe une explication de c dans ϕ utilisant les règles de **ExC** contenant $O(2^n)$ étapes d'inférence.*

Démonstration. Supposons que c est explicable dans ϕ. Il y a deux cas possibles :

— Soit il existe $c' \in \phi$ tel que c' sous-somme c. Dans ce cas, on peut exhiber une explication de c par expansion de c' comme démontré dans la proposition 7.10.

— Sinon, grâce au théorème 7.1, on sait qu'il existe $c' \in \phi$ qui ne s'oppose pas à c et par conséquent, il existe une variable $x \notin var(c)$. Comme, toujours grâce au théorème 7.1, on sait aussi que pour toute variable $x \notin var(c)$, $c \vee x$ et $c \vee \overline{x}$ sont explicables dans ϕ, on peut choisit n'importe quelle variable $x \notin var(c)$ et construire récursivement des explications T_1 et T_2 pour les clauses $c \vee x$ et $c \vee \overline{x}$. Ces appels récursifs sont bornés en profondeur car la taille des clauses est bornée par le nombre de variable n et les appels récursifs sur des clauses de taille n sont forcément des cas de clauses sous-sommées. Après avoir construit des explications pour les clauses $c \vee x$ et $c \vee \overline{x}$, on peut construire une explication pour la clause c en appliquant le symmeric cut comme suit : $(c \vee x) \wedge (c \vee \overline{x}) \vdash c$. Plus précisément, si on note T_3 cette application du *symmetric cut*, alors $T = (T_1, T_2, T_3)$ est une explication de la clause c dans ϕ.

Pour expliquer c, on utilise clairement au plus 2^n applications du *symmetric cut* car dans le pire des cas, chaque clause c' (incluant c) est expliquée par deux clauses $c' \vee x$ et $c' \vee \overline{x}$ (où x est une variable qui n'est pas dans c') jusqu'à avoir des clauses de taille n à expliquer. Dans le pire des cas, chaque clause expliquée grâce à une clause sous-sommée appartenant à la formule nécessite l'application d'une étape de la règle d'expansion, donc on a besoin également d'au plus 2^n expansions. On conclut que si c est explicable alors il existe une explication de c dans ϕ qui utilise les règles de **ExC** et qui contient au maximum 2^{n+1} étapes d'inférence. □

La preuve du théorème 7.3 peut être décrite sous la forme d'un algorithme, que l'on appellera *algorithme d'explication*, et qui permet, étant donnée une clause c et une formule ϕ, d'expliquer la clause c ou de réfuter son explicabilité. On présente l'algorithme ci-dessous avec les notations suivantes :

— $T(\phi)$ est l'application de la transformation T sur la formule ϕ.

— $T_1.T_2$ est la concaténation des transformations T_1 et T_2.

— $expansion(c', c)$ est l'application de l'expansion sur la clause c' pour obtenir la clause c.

On illustre également l'application de l'algorithme d'explication dans l'exemple 7.1.

Exemple 7.1. *On considère la formule $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3) \wedge (\overline{x_1})$ et on cherche à expliquer la clause $c = (x_1)$. On note T et C respectivement la séquence de transformations et l'ensemble des clauses qui doivent encore être expliquées. On construit l'explication de $c = (x_1)$ comme ci-dessous, en considérant un choix lexicographique des variables :*

1. *$T = ()$ et $C = \{(x1)\}$*
   — *(x_1) n'est pas sous-sommée par une clause de ϕ.*

Algorithme 7.1 Algorithme d'explication

Entrée : clause c, formule ϕ
Sortie : (R, T, ϕ') avec $R \in$ {EXPLICABLE, INEXPLICABLE}, T l'explication de c et $\phi' = T(\phi)$

```
1: T ← ∅
2: si ∀c' ∈ φ : c' s'oppose à c alors
3:    retourner (INEXPLICABLE, ∅, φ)
4: si ∃c' ∈ φ tel que c' sous-somme c alors
5:    T ← expansion(c', c)
6:    retourner (EXPLICABLE, T, T(φ))
7: Choisir une variable x ∉ var(c)
8: (E1, T1, φ) ← Expliquer(c ∨ x, φ)
9: (E2, T2, φ) ← Expliquer(c ∨ x̄, φ)
10: si E1 = INEXPLICABLE or E2 = INEXPLICABLE alors
11:    retourner (INEXPLICABLE, ∅, φ)
12: T3 ← ((c ∨ x) ∧ (c ∨ x̄) ⊢ c)
13: T ← T1.T2.T3
14: retourner (EXPLICABLE, T, T3(φ))
```

— (x_1) *n'est pas opposée à toutes les clauses de* ϕ.
— *On choisit la variable* $x_2 \notin (x_1)$ *et on ajoute les clauses* $(x_1 \vee x_2)$ *et* $(x_1 \vee \overline{x_2})$ *à la liste C des clauses à expliquer. L'application du symmetric cut sur ces clauses est ajoutée à* T.

2. $T = (T_1)$ *où* $T_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \vdash (x_1)$ *and* $C = \{(x_1 \vee x_2), (x_1 \vee \overline{x_2})\}$
   — *La clause* $(x_1 \vee x_2)$ *appartient à la formule et est donc immédiatement expliquée.*
3. $T = (T_1)$ *et* $C = \{(x_1 \vee \overline{x_2})\}$
   — $(x_1 \vee \overline{x_2})$ *n'est pas sous-sommée par une clause de* ϕ.
   — $(x_1 \vee \overline{x_2})$ *n'est pas opposée à toutes les clauses de* ϕ.
   — *On choisit la variable* $x_3 \notin (x_1 \vee \overline{x_2})$ *et on ajoute les clauses* $(x_1 \vee \overline{x_2} \vee x_3)$ *et* $(x_1 \vee \overline{x_2} \vee \overline{x_3})$ *à la liste C. L'application du symmetric cut sur ces clauses est ajoutée à* T.
4. $T = (T_2, T_1)$ *où* $T_2 = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \vdash (x_1 \vee \overline{x_2})$ *et* $C = \{(x_1 \vee \overline{x_2} \vee x_3), (x_1 \vee \overline{x_2} \vee \overline{x_3})\}$
   — *La clause* $(x_1 \vee \overline{x_2} \vee x_3)$ *est expliquée par expansion de la clause* $(x_3) \in \phi$ *qui la sous-somme.*
5. $T = (T_3, T_2, T_1)$ *où* $T_3 = (x_3) \vdash (\overline{x_1} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$ *and* $C = \{(x_1 \vee \overline{x_2} \vee \overline{x_3})\}$
   — *La clause* $(x_1 \vee \overline{x_2} \vee \overline{x_3})$ *appartient à la formule et est donc immédiatement expliquée.*
6. $T = (T_3, T_2, T_1)$ *et* $C = \emptyset$. *L'explication de la clause* (x_1) *est complète.*

On conclut que $\phi \vdash (\overline{x_1} \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1) \wedge (\overline{x_1})$ *en appliquant la transformation* T *représentée dans la figure 7.2. Remarquez comment l'explication* T *est construire par la fin (les étapes du bas de la figure ont été calculées avant les étapes du haut). Les clauses* $(\overline{x_1} \vee x_3)$ *et* $(x_1 \vee x_2 \vee x_3)$ *sont des clauses de compensation, essentielles pour préserver l'équivalence Max-SAT.*

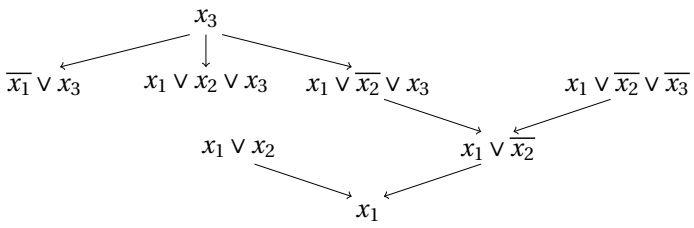


FIGURE 7.2. – Explication de la clause (x_1) dans ϕ

Maintenant que l'on sait expliquer n'importe quelle clause ou réfuter son explicabilité, on va utiliser le résultat du théorème 7.2 pour étendre ce résultat aux formules. Pour cela, on va itérativement expliquer chaque clause de la formule comme énoncé dans le théorème 7.2.

Théorème 7.4. *Soient* ϕ_1 *et* ϕ_2 *deux formules avec* n *variables telles que* ϕ_2 *est explicable dans* ϕ_1 *avec* $m = |\phi_2|$. *Il existe une explication de* ϕ_2 *dans* ϕ_1 *utilisant les règles de* ***ExC*** *et contenant* $O(m \times 2^n)$ *étapes d'inférence.*

Démonstration. On construit une explication de ϕ_2 dans ϕ_1 grâce au résultat du théorème 7.2, c'est à dire en expliquant successivement chaque clause de ϕ_2 dans la formule courante dans laquelle on met progressivement de côté toutes les clauses expliquer (pour ne pas les consommer). Comme démontré dans le théorème 7.3, chaque explication de clause coûte au plus $O(2^n)$ étapes d'inférence et on doit expliquer m clauses. On en déduit que l'explication complète de ϕ_2 dans ϕ_1 contient $O(m \times 2^n)$ étapes d'inférence. □

7.5. Explication et certificats pour le problème Max-SAT

Dans cette section, on va utiliser l'algorithme d'explication pour générer des certificats pour le problème Max-SAT. L'idée est d'utiliser itérativement l'algorithme d'explication pour essayer d'expliquer la clause vide, et cela jusqu'à ce que la clause vide ne soit plus explicable. L'explication de l'ensemble des clauses vides, plus la raison de l'échec pour la génération de la dernière clause vide (qui permet d'en déduire un modèle pour la sous-formule satisfiable finale) est bien un certificat pour le problème Max-SAT.

Théorème 7.5. *Étant donnée une formule* ϕ *avec* n *variables et* m *clauses, on peut déduire* $\phi \vdash \underbrace{\square \wedge \cdots \wedge \square}_{opt(\phi)} \wedge \phi'$ *où* ϕ' *en au plus* $O(m \times 2^n)$ *étapes d'inférence de* ***ExC****.*

Démonstration. On construit la transformation en plusieurs itérations, chaque itération étant l'explication d'une clause vide $\square$, ce qui permet de transformer la formule courante ϕ_i (avec $\phi_0 = \phi$) en un équivalent ϕ_{i+1} et une clause vide $\square$. On a donc $\phi_0 \vdash \square \wedge \phi_1 \vdash \cdots \vdash \underbrace{\square \wedge \cdots \wedge \square}_{s} \wedge \phi_s$ ($s \geq 0$) tel que l'algorithme d'explication échoue à expliquer la clause vide $\square$ dans ϕ_s. Par conséquent, la formule ϕ_s est satisfiable et $s = opt(\phi)$.

Grâce au résultat du théorème 7.3, on peut construire une explication de chacune des s clauses vides en $O(2^n)$ étapes d'inférence. De plus, expliquer une clause vide, par construction de l'algorithme d'explication, n'augmente pas le nombre de variables de la formule et $opt(\phi)$ est inférieur ou égal au nombre initial de clauses m. On construit donc au plus m explications de la clause vide comme dans la preuve du théorème 7.2. On conclut que $\phi \vdash \underbrace{\square \wedge \cdots \wedge \square}_{opt(\phi)} \wedge \phi'$ peut être déduit en $O(m \times 2^n)$ étapes d'inférence de **ExC**. □

Le résultat démontré dans le théorème 7.5 montre que l'algorithme d'explication permet de générer des certificats pour le problème Max-SAT avec une meilleure borne supérieure, en nombre d'étapes d'inférence, que le résultat existant utilisant la max-résolution qui était en $O(m \times n \times 2^n)$ étapes d'inférence [BLM06]. Dans la prochaine section, on va étendre les résultats présentés aux formules pondérées partielles.

7.6. Extension au cas pondéré partiel

Les notions et caractérisations des clauses et des formules explicables se généralisent immédiatement au cas partiel pondéré. On étend les systèmes de preuve proposés dans la section 7.3 en ajoutant deux règles d'inférence, respectivement le *fold* et le *unfold* [BL20], pour dupliquer une clauses pondérées en deux clauses pondérées (dont la somme des poids est égale au poids de la clause d'origine) et pour fusionner deux clauses pondérées ayant les mêmes littéraux. En particulier, on appelle **WExC** (Weighted Explanation Calculus) le système **ExC + fold + unfold**. Les règles du *symmetric cut* et de l'expansion sont naturellement étendues pour le cas partiel pondéré et en pratique, on a simplement besoin du cas particulier de la règle du *symmetric cut* où les deux prémisses ont le même poids.

Définition 7.10 (Fold & Unfold)**.** *Étant donnée une disjonction de littéraux A et deux poids positifs w_1 and w_2, la règle du fold et du unfold sont respectivement définies ci-dessous :*

$$\frac{(c, w_1) \quad (c, w_2)}{(c, w_1 + w_2)} \qquad \frac{(c, w_1 + w_2)}{(c, w_1) \quad (c, w_2)}$$

Pour expliquer une clause souple (c, w), on ignore les poids et on essaie d'expliquer la clause c dans la formule non pondérée. L'explication ainsi obtenue pour la clause c peut être facilement adaptée pour le cas pondérée en utilisant la règle du *unfold* pour

utiliser des clauses de même poids. Si la clause pondérée expliquée est de poids plus faible que la clause recherchée, on ré-applique la même procédure jusqu'à obtenir le poids total souhaité. Ce résultat est démontré dans le théorème 7.6. Expliquer une clause dure (c, ∞) revient exactement au même qu'expliquer une clause non pondérée c en utilisant uniquement la version non pondérée des clauses dures. Enfin, comme toute clause dure peut être vue comme étant une clause souple de poids suffisamment important pour que la falsifier soit plus pénalisant que de falsifier l'ensemble des clauses souples, on peut considérer sans perte de généralité que les formules ont des poids finis, c'est à dire uniquement le cas des formules pondérées.

Théorème 7.6. *Soit ϕ une formule pondérée avec n variables et (c, w) une clause pondérée explicable dans ϕ. Il existe une explication de $(c, w) \in \phi$ utilisant **WExC** et contenant $O(w \times 2^n)$ étapes d'inférence.*

Démonstration. On considère la version non pondérée de la formule ϕ, c'est à dire la formule $\phi' = \{c | (c, w) \in \phi\}$. Comme (c, w) est explicable dans ϕ, alors c est explicable dans ϕ'. On applique l'algorithme d'explication pour obtenir une explication T de c dans ϕ. Soit ξ les prémisses initiales de T. On pose $w' = \min\{w \mid (c, w) \in \phi \text{ and } c \in \xi\}$ et on applique la règle du *unfold* sur les clauses qui appartiennent à ξ dans ϕ et on utilise la version pondérée de la transformation T pour obtenir une explication de (c, w'). On a désormais trois cas possibles :

- si $w' = w$, on a une explication de (c, w) dans ϕ.
- si $w' > w$, on ajoute une étape de *unfold* sur (c, w') pour avoir l'explication de (c, w) dans ϕ.
- si $w' < w$, on répète la même procédure sur $WT(\phi) \setminus \{(c, w')\}$, où WT est l'explication (pondérée) de (c, w') dans ϕ, et cela jusqu'à expliquer des clauses (c, w'_i) telles que $1 \leq i \leq k$ et $\sum_{1 \leq i \leq k} w'_i \geq w$. Ensuite, on fusionne ces clauses en utilisant k étapes de *fold* pour avoir une explication de $(c, \sum_{1 \leq i \leq k} w'_i)$ et on se ramène aux deux précédents cas pour avoir une explication de (c, w) dans ϕ.

Comme montré dans le théorème 7.3, chaque clause non pondérée peut être expliquée en $O(2^n)$ étapes d'inférence et on a besoin de $O(2^n)$ applications de la règle *unfold* pour obtenir l'explication pondérée. Cela est répété k fois où k est borné par w, plus potentiellement $O(1)$ *fold*. Par conséquent, il existe une explication de (c, w) dans ϕ utilisant **WExC** en $O(w \times 2^n)$ étapes d'inférence. □

Théorème 7.7. *Soient ϕ_1 et ϕ_2 deux formules pondérées avec n variables telles que ϕ_2 est explicable dans ϕ_1 avec m le nombre de clauses de ϕ_2 et $w = \max\{w' \mid (c, w') \in \phi_2 \text{ and } w' \neq \infty\}$. Il existe une explication de ϕ_2 dans ϕ_1 utilisant **WExC** contenant $O(m \times w \times 2^n)$ étapes d'inférence.*

Démonstration. On explique itérativement chaque clause de ϕ_2. Chaque clause peut être expliquée en $O(w \times 2^n)$ étapes d'inférence comme montré dans le théorème 7.6 et on a exactement m clauses à expliquer. Par conséquent, il existe une explication de ϕ_2 dans ϕ_1 utilisant **WExC** contenant $O(m \times w \times 2^n)$ étapes d'inférence. □

Définition 7.11 (Plafond d'une formule)**.** *Soit ϕ une formule. Le plafond (roof) d'une formule, noté $rf(\phi)$, est défini comme suit :*

$$rf(\phi) = \sum_{(c,w)\in\phi \mid w\neq\infty} w$$

Théorème 7.8. *Étant donnée une formule ϕ avec n variables et m clauses, on peut déduire $\phi \vdash (\square, opt(\phi)) \wedge \phi'$ avec ϕ' satisfiable en utilisant $O(rf(\phi) \times 2^n)$ étapes d'inférence de **WExC**.*

Démonstration. Comme pour le cas non pondéré, on explique itérativement une nouvelle clause vide jusqu'à ce que cela ne soit pas possible et on fusionne les clauses vides en une seule clause $(\square, opt(\phi))$. Le nombre de clauses vides expliquées $opt(\phi)$ est borné par $rf(\phi)$ et chaque clause vide peut être expliquée en $O(2^n)$ étapes d'inférence de **WExC** comme montré dans le théorème 7.6. Par conséquent, on peut déduire $\phi \vdash (\square, opt(\phi)) \wedge \phi'$ avec ϕ' satisfiable en utilisant $O(rf(\phi) \times 2^n)$ étapes d'inférence de **WExC**. □

7.7. Conclusion

Dans ce chapitre, on s'est intéressé à l'inférence dans Max-SAT. On a ainsi introduit la notion de clauses et de formules explicables, qui peuvent être déduites à partir d'une formule initiale en appliquant des transformations qui préservent l'équivalence Max-SAT. Comme la max-résolution est inférentiellement incomplète, on a proposé un nouveau système d'inférence (**ExC**), composé de la règle du symmetric cut et de la règle d'expansion. On a démontré que **ExC** est inférentiellement complet et on l'a comparé avec d'autres systèmes de preuve dans la littérature. Pour construire des explications de n'importe quelle clause ou formule explicable utilisant **ExC**, on a proposé l'algorithme d'explication, qui est capable de construire des explications ou de réfuter l'explicabilité d'une clause ou d'une formule. L'algorithme d'explication peut aussi être utilisé pour construire des certificats pour le problème Max-SAT. Ces résultats ont été étendus pour le cas des formules pondérées partielles.

Les perspectives de ce travail sont à la fois théoriques et pratiques. On a proposé un couple composé d'un système de preuve et d'un algorithme qui permettent d'inférer n'importe quelle formule explicable en $O(m \times 2^n)$ étapes d'inférence. Est-il possible d'améliorer ce résultat pour inférer plus efficacement des formules explicables ? Ensuite, il est intéressant de se demander si l'inférence de clauses et de formules, ainsi que l'algorithme d'explication, peuvent avoir un intérêt pour la résolution pratique du problème Max-SAT. Par exemple, cela pourrait avoir un impact sur le calcul de bornes inférieures dans les solveurs Max-SAT de type séparation et évaluation, ou permettre de transformer les clauses dans ces solveurs en utilisant des motifs comme ceux utilisés avec la règle de la max-résolution [LMP07].

Conclusion

Conclusion

Dans cette thèse, on s'est intéressé au problème de satisfiabilité maximum. On a tout d'abord introduit deux méthodes pour adapter n'importe quelle réfutation par résolution en une max-réfutation valide pour Max-SAT [PCH21b ; PCH20]. Avant les travaux de cette thèse, il n'existait aucune méthode permettant d'adapter n'importe quelle réfutation par résolution afin d'en déduire une max-réfutation et la question de savoir si cela était possible sans augmenter considérablement la taille de la transformation était une question ouverte [BLM06]. On a en particulier démontré que cela était possible lorsque la réfutation est *tree-like regular*, *tree-like* ou *semi-tree-like* puisque l'une des adaptations proposées y est de taille linéaire. Ces deux adaptations sont basées sur l'augmentation de la max-résolution par l'ajout du *split*, ce qui justifie la pertinence de cette règle pour Max-SAT.

Ensuite, on a utilisé l'adaptation des réfutations par résolution pour Max-SAT qu'on a proposé dans [PCH20] pour générer des certificats pour le problème Max-SAT pour les instances de la littérature. L'outil implémenté pour construire les certificats, MS-Builder, ainsi que le vérificateur de certificats, MS-Checker, ont tous deux été mis à disposition de la communauté scientifique [PCH21a ; Py21]. MS-Builder est en particulier le premier outil de la littérature permettant de générer des certificats pour le problème Max-SAT.

Enfin, on s'est intéressé à comment inférer efficacement une information sous forme de clause ou formule à partir d'une formule initiale donnée en utilisant des règles d'inférence qui préservent l'équivalence Max-SAT [PCH21d]. Pour cela, on a introduit la notion d'explicabilité des clauses et des formules. On a proposé un nouveau système de preuve pour Max-SAT nommé **ExC**, qui se compose de la règle du *symmetric cut* et de la règle de l'expansion, définie dans [PCH21d]. Ce nouveau système de preuves améliore la max-résolution car il est inférentiellement complet et tout traitement effectué par max-résolution peut être également effectué dans ce système. On a également proposé l'algorithme d'explication qui utilise **ExC** et qui permet d'inférer des clauses ou des formules si elles sont explicables.

Les travaux futurs portent notamment sur l'amélioration des travaux énoncés ci-dessus. Tout d'abord, on a proposé deux méthodes afin d'adapter n'importe quelle réfutation par résolution pour en déduire une max-réfutation, mais aucune de ces adaptations n'a été démontrée comme étant polynomiale dans le cas général. Il peut être intéressant d'essayer de démontrer l'existence d'une adaptation polynomiale dans le cas général si cela est possible ou, dans le cas contraire, de fournir une dé-

monstration qu'une telle adaptation n'est pas possible. De plus, les adaptations ont été faites grâce à l'ajout de la règle du *split* et il peut être intéressant de voir dans quels cas cette adaptation est possible en utilisant uniquement la max-résolution.

Ensuite, si on a proposé, pour la première fois, un outil permettant de générer des certificats pour le problème Max-SAT, il convient d'améliorer son efficacité, que ce soit en améliorant l'algorithme utilisé, son implémentation, ou encore en compressant le format des certificats exportés. De plus, MS-Builder utilise uniquement l'adaptation des réfutations par résolution proposée dans le chapitre 5 [PCH20]. Il peut être pertinent d'implémenter aussi l'adaptation proposée dans le chapitre 4 [PCH21b] et de choisir, pour chaque réfutation calculée, la méthode à utiliser pour adapter la réfutation, ceci afin d'améliorer les performances pratiques de l'outil.

Si les travaux présentés ont surtout comme intérêt la génération théorique et pratique de certificats dans Max-SAT, qu'il s'agisse de max-réfutations, de certificats pour le problème Max-SAT ou d'inférence de clauses et de formules, il convient d'essayer d'utiliser ces travaux afin d'améliorer la résolution pratique du problème Max-SAT. Les algorithmes de résolution basés sur l'appel à un oracle SAT utilisent le traitement des cœurs inconsistants par l'ajout d'une contrainte de cardinalité. Dans quels cas serait-il judicieux de remplacer ce traitement par une transformation par max-résolution ? Ce traitement pourrait notamment se faire lorsque la réfutation par résolution du cœur inconsistant est *semi-tree-like*, ce qui généraliserait le traitement effectué dans le cas *read-once* [HM11]. Par ailleurs, l'explication de clauses vides de la formule pourrait aussi être appliquée, par exemple lors de l'estimation de la borne inférieure dans les algorithmes de type séparation et évaluation. Pour des raisons d'efficacité pratique, il conviendrait d'identifier les cas pertinents pour l'application d'une cette explication. Enfin, le problème de satisfiabilité minimum (Min-SAT) [LXM21], dual à Max-SAT, est une autre version d'optimisation du problème SAT et consiste à minimiser le nombre de clauses satisfaites. Il serait intéressant d'étendre nos résultats à Min-SAT ainsi qu'à des formalismes proches comme le problème WCSP (*Weighted Constraint Satisfaction Problem*) [LH05].

Bibliographie

[Abr15] André ABRAMÉ. « Max-résolution et apprentissage pour la résolution du problème de satisfiabilité maximum ». Thèse de doct. Aix-Marseille Université, 2015 (cf. p. 24, 35, 57, 59).

[AH14a] André ABRAMÉ et Djamal HABET. « Efficient Application of Max-SAT Resolution on Inconsistent Subsets ». In : *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP 2014)*. 2014, p. 92-107 (cf. p. 57).

[AH14b] André ABRAMÉ et Djamal HABET. « Local Max-Resolution in Branch and Bound Solvers for Max-SAT ». In : 2014, p. 336-343 (cf. p. 57, 58).

[AH14c] André ABRAMÉ et Djamal HABET. « Maintaining and Handling All Unit Propagation Reasons in Exact Max-SAT Solvers ». In : *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SOCS 2014)*. 2014 (cf. p. 57).

[AH14d] André ABRAMÉ et Djamal HABET. « On the Extension of Learning for Max-SAT ». In : *Proceedings of the 7th European Starting AI Researcher Symposium (STAIRS 2014)*. T. 241. 2014, p. 1-10 (cf. p. 57, 58).

[AH15a] André ABRAMÉ et Djamal HABET. « Ahmaxsat : Description and Evaluation of a Branch and Bound Max-SAT Solver ». In : *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2015), p. 89-128 (cf. p. 57).

[AH15b] André ABRAMÉ et Djamal HABET. « On the Resiliency of Unit Propagation to Max-Resolution ». In : *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*. 2015, p. 268-274 (cf. p. 57).

[AH16] André ABRAMÉ et Djamal HABET. « Learning Nobetter Clauses in Max-SAT Branch and Bound Solvers ». In : *Proceedings of IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2016)*. 2016, p. 452-459 (cf. p. 57).

[AKS02] Manindra AGRAWAL, Neeraj KAYAL et Nitin SAXENA. « Primes is in P ». In : *Annals of Mathematics* 160 (sept. 2002) (cf. p. 25).

[Ale+02] Michael ALEKHNOVICH, Jan JOHANNSEN, Toniann PITASSI et al. « An exponential separation between regular and general resolution ». In : *Proceedings on 34th Annual ACM Symposium on Theory of Computing*. ACM, 2002, p. 448-456 (cf. p. 42).

[Ale+07] Michael ALEKHNOVICH, Jan JOHANNSEN, Toniann PITASSI et al. « An Exponential Separation between Regular and General Resolution ». In : *Theory of Computing* 3.1 (2007), p. 81-102 (cf. p. 42).

[AMP04] Teresa ALSINET, Felip MANYÀ et Jordi PLANES. « A Max-SAT Solver with Lazy Data Structures ». In : *Advances in Artificial Intelligence - IBERAMIA 2004, Proceedings*. Sous la dir. de Christian LEMAÎTRE, Carlos A. Reyes GARCIA et Jesùs A. GONZÀLEZ. T. 3315. Lecture Notes in Computer Science. Springer, 2004, p. 334-342 (cf. p. 58).

[AG13] C. ANSÓTEGUI et J. GABÀS. « Solving (Weighted) Partial MaxSAT with ILP ». In : *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR 2013*. 2013, p. 403-409 (cf. p. 53).

[Ans+12] Carlos ANSÓTEGUI, Maria BONET, Joel GABÀS et al. « Improving SAT-Based weighted MaxSAT solvers ». In : *Theory and Applications of Satisfiability Testing - SAT 2009*. T. 7514. Oct. 2012, p. 86-101 (cf. p. 53).

[ABL09a] Carlos ANSÓTEGUI, Maria BONET et Jordi LEVY. « On Solving MaxSAT Through SAT ». In : *Proceedings of the 2009 Conference on Artificial Intelligence Research and Development : Proceedings of the 12th International Conference of the Catalan Association for Artificial Intelligence*. Juil. 2009, p. 284-292 (cf. p. 53).

[ABL09b] Carlos ANSÓTEGUI, Maria BONET et Jordi LEVY. « Solving (Weighted) Partial MaxSAT through Satisfiability Testing ». In : *Theory and Applications of Satisfiability Testing - SAT 2009*. T. 5584. Juin 2009, p. 427-440 (cf. p. 53).

[ABL10] Carlos ANSÓTEGUI, Maria BONET et Jordi LEVY. « A new algorithm for Weighted Partial MaxSAT ». In : jan. 2010 (cf. p. 53).

[ABL13] Carlos ANSÓTEGUI, Maria Luisa BONET et Jordi LEVY. « SAT-based MaxSAT algorithms ». In : *Artificial Intelligence* 196 (2013), p. 77-105 (cf. p. 53).

[ALM08] Josep ARGELICH, Chu Min LI et Felip MANYÀ. « A Preprocessor for MaxSAT Solvers ». In : *Theory and Applications of Satisfiability Testing - SAT 2008*. T. 4996. Lecture Notes in Computer Science. 2008, p. 15-20 (cf. p. 60).

[Arg+08] Josep ARGELICH, Chu Min LI, Felip MANYÀ et al. « The First and Second Max-SAT Evaluations ». In : *Journal of Satisfiability Boolean Modeling and Computation* 4.2-4 (2008), p. 251-278. URL : https://maxsat-evaluations.github.io (cf. p. 49).

[AL19] Albert ATSERIAS et Massimo LAURIA. « Circular (Yet Sound) Proofs ». In : *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Sous la dir. de Mikolás JANOTA et Inês LYNCE. T. 11628. Lecture Notes in Computer Science. Springer, 2019, p. 1-18 (cf. p. 60).

[AS09] Gilles AUDEMARD et Laurent SIMON. « Predicting Learnt Clauses Quality in Modern SAT Solvers ». In : *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*. 2009, p. 399-404 (cf. p. 36).

[AS18] Gilles AUDEMARD et Laurent SIMON. « On the Glucose SAT Solver ». In : *Journal of Artificial Intelligence Tools* 27.1 (2018) (cf. p. 36).

[Bac+17] Fahiem BACCHUS, Antti HYTTINEN, Matti JÄRVISALO et al. « Reduced cost fixing in maxsat ». In : *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming - CP 2017*. T. 10416. 2017, p. 641-651 (cf. p. 57).

[Bac+20] Fahiem BACCHUS, Matti JÄRVISALO, Jeremias BERG et al. *MaxSAT Evaluation 2020*. 2020. URL : https://maxsat-evaluations.github.io/2020/ (cf. p. 64, 90, 93, 96).

[BR99] Nikhil BANSAL et Venkatesh RAMAN. « Upper Bounds for MaxSat : Further Improved ». In : *International Symposium on Algorithms and Computation (ISAAC)*. T. 1741. Jan. 1999, p. 247-258 (cf. p. 47, 48).

[BIW04] Eli BEN-SASSON, Russell IMPAGLIAZZO et Avi WIGDERSON. « Near Optimal Separation Of Tree-Like And General Resolution ». In : *Combinatorica* 24 (sept. 2004), p. 585-603 (cf. p. 40, 41).

[BBP20] Jeremias BERG, Fahiem BACCHUS et Alex POOLE. « Abstract Cores in Implicit Hitting Set MaxSat Solving ». In : *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing - SAT 2020*. 2020 (cf. p. 57).

[BP10] Daniel Le BERRE et Anne PARRAIN. « The Sat4j library, release 2.2 ». In : *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), p. 59-64 (cf. p. 49).

[Bie06] Armin BIERE. *TraceCheck*. 2006. URL : http://fmv.jku.at/tracecheck/ (cf. p. 39, 96).

[Bie08] Armin BIERE. « PicoSAT Essentials ». In : *Journal on Satisfiability, Boolean Modeling and Computation* 4.2-4 (2008), p. 75-97 (cf. p. 38).

[Bie10] Armin BIERE. *Booleforce*. 2010. URL : http://fmv.jku.at/booleforce/ (cf. p. 39, 96).

[Bie+99a] Armin BIERE, Alessandro CIMATTI, Edmund M. CLARKE et al. « Symbolic Model Checking Using SAT Procedures instead of BDDs ». In : *Proceedings of the 36th Conference on Design Automation*. ACM Press, 1999, p. 317-320 (cf. p. 31).

[Bie+99b] Armin BIERE, Alessandro CIMATTI, Edmund M. CLARKE et al. « Symbolic Model Checking without BDDs ». In : *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Proceedings*. T. 1579. Lecture Notes in Computer Science. Springer, 1999, p. 193-207 (cf. p. 31).

[Bie+20] Armin BIERE, Katalin FAZEKAS, Mathias FLEURY et al. « CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020 ». In : *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Sous la dir. de Tomas BALYO, Nils FROLEYKS, Marijn HEULE et al. T. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, p. 51-53 (cf. p. 36).

[Bie+09] Armin BIERE, Marijn HEULE, Hans van MAAREN et al., éd. *Handbook of Satisfiability*. T. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009 (cf. p. 31, 41).

[Bon+18] Maria Luisa BONET, Sam BUSS, Alexey IGNATIEV et al. « MaxSAT Resolution With the Dual Rail Encoding ». In : *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Sous la dir. de Sheila A. MCILRAITH et Kilian Q. WEINBERGER. AAAI Press, 2018, p. 6565-6572 (cf. p. 60).

[BL20] Maria Luisa BONET et Jordi LEVY. « Equivalence Between Systems Stronger Than Resolution ». In : *Theory and Applications of Satisfiability Testing – SAT 2020*. T. 12178. Lecture Notes in Computer Science. 2020 (cf. p. 48, 49, 63, 106, 107, 112).

[BLM06] María Luisa BONET, Jordi LEVY et Felip MANYÀ. « A complete calculus for MAX-SAT ». In : *Theory and Applications of Satisfiability Testing - SAT 2006*. T. 4121. Août 2006, p. 240-251 (cf. p. 39, 48, 60, 62, 63, 112, 116).

[BLM07] María Luisa BONET, Jordi LEVY et Felip MANYÀ. « Resolution for Max-SAT ». In : *Artificial Intelligence*. T. 171. 2007, p. 606-618 (cf. p. 48).

[BF98] Brian BORCHERS et Judith FURMAN. « A Two-Phase Exact Algorithm for MAX-SAT and Weighted MAX-SAT Problems ». In : *Journal of Combinatorial Optimization* 2.4 (1998), p. 299-306 (cf. p. 58).

[BGS99] Laure BRISOUX, Éric GRÉGOIRE et Lakhdar SAIS. « Improving Backtrack Search for SAT by Means of Redundancy ». In : *Proceedings of Foundations of Intelligent Systems, 11th International Symposium, ISMIS '99*. T. 1609. Lecture Notes in Computer Science. 1999, p. 301-309 (cf. p. 37).

[BB92] M. BURO et H. Kleine BÜNING. « Report on a SAT competition ». Technical Report, University of Paderborn. 1992 (cf. p. 37).

[CHA20] Mohamed Sami CHERIF, Djamal HABET et André ABRAMÉ. « Understanding the power of Max-SAT resolution through UP-resilience ». In : t. 289. 2020 (cf. p. 58).

[Cla+01] Edmund M. CLARKE, Armin BIERE, Richard RAIMI et al. « Bounded Model Checking Using Satisfiability Solving ». In : *Formal Methods in System Design* 19.1 (2001), p. 7-34 (cf. p. 31).

[CKL04] Edmund M. CLARKE, Daniel KROENING et Flavio LERDA. « A Tool for Checking ANSI-C Programs ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*. T. 2988. Lecture Notes in Computer Science. Springer, 2004, p. 168-176 (cf. p. 31).

[Coo71] Stephen A. COOK. « The Complexity of Theorem-Proving Procedures ». In : *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. 1971, p. 151-158 (cf. p. 25, 31).

[Dar+07] Sylvain DARRAS, Gilles DEQUEN, Laure DEVENDEVILLE et al. « On Inconsistent Clause-Subsets for Max-SAT Solving ». In : *Principles and Practice of Constraint Programming (CP 2007)*. T. 4741. Lecture Notes in Computer Science. 2007, p. 225-240 (cf. p. 57, 58).

[Dav13] Jessica DAVIES. « Solving MAXSAT by Decoupling Optimization and Satisfaction ». Thèse de doct. University of Toronto, 2013 (cf. p. 54).

[DB11] Jessica DAVIES et Fahiem BACCHUS. « Solving MAXSAT by Solving a Sequence of Simpler SAT Instances ». In : *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*. CP'11. 2011, p. 225-239 (cf. p. 54, 57).

[DB13a] Jessica DAVIES et Fahiem BACCHUS. « Exploiting the Power of mip Solvers in maxsat ». In : *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing - SAT 2013*. T. 7962. 2013, p. 166-181 (cf. p. 57).

[DB13b] Jessica DAVIES et Fahiem BACCHUS. « Postponing optimization to speed up MAXSAT ». In : *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming - CP 2013*. T. 8124. 2013, p. 247-262 (cf. p. 57).

[DLL62] Martin DAVIS, George LOGEMANN et Donald LOVELAND. « A machine program for theorem-proving ». In : *Communications of the Association for Computing Machinery* 5 (1962), p. 394-397 (cf. p. 33, 34).

[DP60] Martin DAVIS et Hilary PUTNAM. « A Computing Procedure for Quantification Theory ». In : *Journal of the Association for Computing Machinery* 7 (juil. 1960), p. 201-215 (cf. p. 34).

[DD03] Gilles DEQUEN et Olivier DUBOIS. « kcnfs : An Efficient Solver for Random k-SAT Formulae ». In : *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*. T. 2919. Lecture Notes in Computer Science. 2003, p. 486-501 (cf. p. 37).

[ES03] Niklas EÉN et Niklas SÖRENSSON. « An Extensible SAT-solver ». In : *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*. T. 2919. Lecture Notes in Computer Science. 2003, p. 502-518 (cf. p. 36, 38, 54).

[Fil+20] Yuval FILMUS, Meena MAHAJAN, Gaurav SOOD et al. « MaxSAT Resolution and Subcube Sums ». In : *SAT*. 2020, p. 295-311 (cf. p. 60, 63).

[FM06] Zhaohui FU et Sharad MALIK. « On Solving the Partial MAX-SAT Problem ». In : *Theory and Applications of Satisfiability Testing - SAT 2006*. 2006 (cf. p. 49, 51).

[Gel08] Allen Van GELDER. « Verifying RUP Proofs of Propositional Unsatisfiability ». In : *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008*. 2008 (cf. p. 39).

[GN03] Evguenii I. GOLDBERG et Yakov NOVIKOV. « Verification of Proofs of Unsatisfiability for CNF Formulas ». In : *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003)*. IEEE Computer Society, 2003, p. 10886-10891 (cf. p. 39).

[GSC97] Carla P. GOMES, Bart SELMAN et Nuno CRATO. « Heavy-Tailed Distributions in Combinatorial Search ». In : *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*. Sous la dir. de Gert SMOLKA. T. 1330. Lecture Notes in Computer Science. Springer, 1997, p. 121-135 (cf. p. 38).

[Gom+00] Carla P. GOMES, Bart SELMAN, Nuno CRATO et al. « Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems ». In : *Journal of Automated Reasoning* 24.1/2 (2000), p. 67-100 (cf. p. 38).

[GSK98] Carla P. GOMES, Bart SELMAN et Henry A. KAUTZ. « Boosting Combinatorial Search Through Randomization ». In : *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98*. 1998, p. 431-437 (cf. p. 38).

[Hak85] Armin HAKEN. « The Intractability of Resolution ». In : *Theoretical Compututer Science* 39 (1985), p. 297-308 (cf. p. 38).

[HLO07] Federico HERAS, Javier LARROSA et Albert OLIVERAS. « MiniMaxSat : A New Weighted Max-SAT Solver ». In : *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Proceedings*. T. 4501. Lecture Notes in Computer Science. 2007, p. 41-55 (cf. p. 57).

[HLO08] Federico HERAS, Javier LARROSA et Albert OLIVERAS. « MiniMaxSAT : An Efficient Weighted Max-SAT solver ». In : *Journal of Artificial Intelligence Research* 31 (2008), p. 1-32 (cf. p. 57).

[HM11] Federico HERAS et Joao MARQUES-SILVA. « Read-Once resolution for Unsatisfiability-Based Max-SAT Algorithms ». In : *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22*. 2011 (cf. p. 61, 83, 117).

[HMM11] Federico HERAS, Antonio MORGADO et João MARQUES-SILVA. « Core-Guided Binary Search Algorithms for Maximum Satisfiability ». In : *Proceedings of the National Conference on Artificial Intelligence (AAAI 2011)*. Jan. 2011, p. 36-41 (cf. p. 53).

[HMM12] Federico HERAS, Antonio MORGADO et João MARQUES-SILVA. « Improvements to Core-Guided Binary Search for MaxSAT ». In : *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing - SAT 2012*. T. 7317. 2012 (cf. p. 53).

[Her+08] Philipp HERTEL, Fahiem BACCHUS, Toniann PITASSI et al. « Clause Learning Can Effectively P-Simulate General Propositional Resolution ». In : *AAAI*. 2008, p. 283-290 (cf. p. 105).

[HJW13a] Marijn HEULE, Warren A. Hunt JR. et Nathan WETZLER. « Trimming while checking clausal proofs ». In : *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, p. 181-188 (cf. p. 39).

[HJW13b] Marijn HEULE, Warren A. Hunt JR. et Nathan WETZLER. « Verifying Refutations with Extended Resolution ». In : *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. Sous la dir. de Maria Paola BONACINA. T. 7898. Lecture Notes in Computer Science. Springer, 2013, p. 345-359 (cf. p. 39).

[HJW15] Marijn HEULE, Warren A. Hunt JR. et Nathan WETZLER. « Expressing Symmetry Breaking in DRAT Proofs ». In : *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Proceedings*. Sous la dir. d'Amy P. FELTY et Aart MIDDELDORP. T. 9195. Lecture Notes in Computer Science. Springer, 2015, p. 591-606 (cf. p. 39).

[HK17] Marijn J. H. HEULE et Oliver KULLMANN. « The science of brute force ». In : *Communications of the ACM* 60.8 (2017), p. 70-79 (cf. p. 31).

[HKM16] Marijn J. H. HEULE, Oliver KULLMANN et Victor W. MAREK. « Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer ». In : *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Proceedings*. Sous la dir. de Nadia CREIGNOU et Daniel Le BERRE. T. 9710. Lecture Notes in Computer Science. Springer, 2016, p. 228-245 (cf. p. 31).

[HB19] Randy HICKEY et Fahiem BACCHUS. « Speeding Up Assumption-Based SAT ». In : *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing - SAT 2019*. T. 11628. 2019, p. 164-182 (cf. p. 57).

[IBM09] IBM. *IBM CPLEX optimizer*. 2009. URL : https://www.ibm.com/products/ilog-cplex-optimization-studio (cf. p. 54).

[IMM17] Alexey IGNATIEV, António MORGADO et João MARQUES-SILVA. « On Tackling the Limits of Resolution in SAT Solving ». In : *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Sous la dir. de Serge GASPERS et Toby WALSH. T. 10491. Lecture Notes in Computer Science. Springer, 2017, p. 164-183 (cf. p. 60).

[IMM19] Alexey IGNATIEV, António MORGADO et João MARQUES-SILVA. « RC2 : an Efficient MaxSAT Solver ». In : *Journal on Satisfiability, Boolean Modeling and Computation* 11 (2019), p. 53-64 (cf. p. 53).

[IM95] Kazuo IWAMA et Eiji MIYANO. « Intractability of Read-Once Resolution ». In : *Proceedings of the Tenth Annual Structure in Complexity Theory Conference*. 1995, p. 29-36 (cf. p. 40).

[JW90] Robert G. JEROSLOW et Jinchang WANG. « Solving Propositional Satisfiability Problems ». In : *Annals of Mathematics and Artificial Intelligence* 1 (1990), p. 167-187 (cf. p. 37).

[KS99] Henry A. KAUTZ et Bart SELMAN. « Unifying SAT-based and Graph-based Planning ». In : *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*. Morgan Kaufmann, 1999, p. 318-325 (cf. p. 31).

[KL14] Boris KONEV et Alexei LISITSA. « A SAT Attack on the Erdős Discrepancy Conjecture ». In : *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Proceedings*. Sous la dir. de Carsten SINZ et Uwe EGLY. T. 8561. Lecture Notes in Computer Science. Springer, 2014, p. 219-226 (cf. p. 31).

[KL15] Boris KONEV et Alexei LISITSA. « Computer-aided proof of Erdős discrepancy properties ». In : *Artificial Intelligence* 224 (2015), p. 103-118 (cf. p. 31).

[Kos+12] Miyuki KOSHIMURA, Tong ZHANG, Hiroshi FUJITA et al. « QMaxSAT : A Partial Max-SAT Solver system description ». In : *Journal on Satisfiability, Boolean Modeling and Computation* 8 (jan. 2012) (cf. p. 49, 50).

[Küg10] Adrian KÜGEL. « Improved Exact Solver for the Weighted MAX-SAT Problem ». In : *POS-10. Pragmatics of SAT*. T. 8. EPiC Series in Computing. 2010, p. 15-27 (cf. p. 57).

[LH05] Javier LARROSA et Federico HERAS. « Resolution in Max-SAT and its relation to local consistency in weighted CSPs ». In : *IJCAI International Joint Conference on Artificial Intelligence - IJCAI 2005*. Jan. 2005, p. 193-198 (cf. p. 48, 117).

[LR20a] Javier LARROSA et Emma ROLLON. « Augmenting the Power of (Partial) MaxSat Resolution with Extension ». In : *Proceedings of the AAAI Conference on Artificial Intelligence*. 2020 (cf. p. 63).

[LR20b] Javier LARROSA et Emma ROLLON. « Towards a Better Understanding of (Partial Weighted) MaxSAT Proof Systems ». In : *Proceedings of Theory and Applications of Satisfiability Testing - SAT 2020*. T. 12178. Lecture Notes in Computer Science. 2020, p. 218-232 (cf. p. 49, 63, 103-107).

[LA97] Chu Min LI et ANBULAGAN. « Heuristics Based on Unit Propagation for Satisfiability Problems ». In : *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97*. 1997, p. 366-371 (cf. p. 37).

[Li+08] Chu Min LI, Felip MANYÀ, Nouredine Ould MOHAMEDOU et al. « Transforming Inconsistent Subformulas in MaxSAT Lower Bound Computation ». In : *Principles and Practice of Constraint Programming, 14th International Conference (CP 2008)*. T. 5202. Lecture Notes in Computer Science. 2008, p. 582-587 (cf. p. 57).

[Li+10] Chu Min LI, Felip MANYÀ, Nouredine Ould MOHAMEDOU et al. « Resolution-based lower bounds in MaxSAT ». In : *Constraints* 15 (2010), p. 456-484 (cf. p. 57).

[LMP05] Chu Min LI, Felip MANYÀ et Jordi PLANES. « Exploiting Unit Propagation to Compute Lower Bounds in Branch and Bound Max-SAT Solvers ». In : *Principles and Practice of Constraint Programming (CP 2005)*. T. 3709. Lecture Notes in Computer Science. 2005, p. 403-414 (cf. p. 57, 58).

[LMS16] Chu Min LI, Felip MANYÀ et Joan Ramon SOLER. « A Clause Tableau Calculus for MaxSAT ». In : *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*. 2016, p. 766-772 (cf. p. 60).

[Li+09] Chu-Min LI, Felip MANYÀ, Nouredine MOHAMEDOU et al. « Exploiting Cycle Structures in Max-SAT ». In : *Theory and Applications of Satisfiability Testing - (SAT 2009)*. T. 5584. 2009, p. 467-480 (cf. p. 57).

[LMP06] Chu-Min LI, Felip MANYÀ et Jordi PLANES. « Detecting Disjoint Inconsistent Subformulas for Computing Lower Bounds for Max-SAT ». In : *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*. 2006, p. 86-91 (cf. p. 57, 58).

[LMP07] Chu-Min LI, Felip MANYÀ et Jordi PLANES. « New inference rules for Max-SAT ». In : *Journal of Artificial Intelligence Research (JAIR)* 30 (2007), p. 321-359 (cf. p. 57, 114).

[LXM21] Chu-Min LI, Fan XIAO et Felip MANYÀ. « A resolution calculus for MinSAT ». In : *Logic Journal of the IGPL* 29.1 (2021), p. 28-44 (cf. p. 117).

[Lia+16] Jia Hui LIANG, Vijay GANESH, Pascal POUPART et al. « Exponential Recency Weighted Average Branching Heuristic for SAT Solvers ». In : *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. Sous la dir. de Dale SCHUURMANS et Michael P. WELLMAN. AAAI Press, 2016, p. 3434-3440 (cf. p. 37).

[Lib00] Paolo LIBERATORE. « On the complexity of choosing the branching literal in DPLL ». In : *Artificial Intelligence* 116.1-2 (2000), p. 315-326 (cf. p. 36).

[LSL08] Han LIN, Kaile SU et Chu Min LI. « Within-problem Learning for Efficient Lower Bound Computation in Max-SAT Solving ». In : *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*. 2008, p. 351-356 (cf. p. 57, 58).

[Liu+16] Yanli LIU, Chu Min LI, Kun HE et al. « Breaking Cycle Structure to Improve Lower Bound for Max-SAT ». In : *Frontiers in Algorithmics, 10th International Workshop, FAW 2016*. T. 9711. Lecture Notes in Computer Science. 2016, p. 111-124 (cf. p. 57).

[MF] Hans van MAAREN et John FRANCO. *SAT Competitions*. URL : http://www.satcompetition.org/ (cf. p. 33).

[MMP09] Vasco MANQUINHO, Joao MARQUES-SILVA et Jordi PLANES. « Algorithms for Weighted Boolean Optimization ». In : *Theory and Applications of Satisfiability Testing - SAT 2009*. T. 5584. Avr. 2009 (cf. p. 53).

[MM08] Joao MARQUES-SILVA et Vasco MANQUINHO. « Towards More Effective Unsatisfiability-Based Maximum Satisfiability Algorithms ». In : *Theory and Applications of Satisfiability Testing – SAT 2008*. T. 4996. Mai 2008 (cf. p. 53).

[MP07] Joao MARQUES-SILVA et Jordi PLANES. « On Using Unsatisfiability for Solving Maximum Satisfiability ». In : *ArXiv* (2007) (cf. p. 53).

[MP08] Joao MARQUES-SILVA et Jordi PLANES. « Algorithms for Maximum Satisfiability Using Unsatisfiable Cores ». In : avr. 2008, p. 408-413 (cf. p. 53).

[MML12] R. MARTINS, Vasco MANQUINHO et I. LYNCE. « On partitioning for maximum satisfiability ». In : *20th European Conference on Artificial Intelligence - ECAI 2012*. T. 242. Jan. 2012, p. 913-914 (cf. p. 53).

[MML14] Ruben MARTINS, Vasco M. MANQUINHO et Inês LYNCE. « Open-WBO : A Modular MaxSAT Solver ». In : *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*. T. 8561. Lecture Notes in Computer Science. 2014, p. 438-445 (cf. p. 53).

[MB19] Sibylle MÖHLE et Armin BIERE. « Backing Backtracking ». In : *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*. Sous la dir. de Mikolás JANOTA et Inês LYNCE. T. 11628. Lecture Notes in Computer Science. Springer, 2019, p. 250-266 (cf. p. 36).

[MDM14a] Antonio MORGADO, Carmine DODARO et Joao MARQUES-SILVA. « Core-Guided MaxSAT with Soft Cardinality Constraints ». In : *Principles and Practice of Constraint Programming - CP 2014*. Slides. 2014 (cf. p. 50, 51, 55).

[MDM14b] Antonio MORGADO, Carmine DODARO et Joao MARQUES-SILVA. « Core-Guided MaxSAT with Soft Cardinality Constraints ». In : *Principles and Practice of Constraint Programming - CP 2014*. T. 8656. Sept. 2014, p. 564-573 (cf. p. 53).

[Mor+13] Antonio MORGADO, Federico HERAS, Mark LIFFITON et al. « Iterative and core-guided MaxSAT solving : A survey and assessment ». In : *Constraints* 18 (oct. 2013), p. 478-534 (cf. p. 53).

[Mos+01] Matthew W. MOSKEWICZ, Conor F. MADIGAN, Ying ZHAO et al. « Chaff : Engineering an Efficient SAT Solver ». In : *Proceedings of the 38th Design Automation Conference, DAC 2001*. ACM, 2001, p. 530-535 (cf. p. 36-38).

[NR] Alexander NADEL et Vadim RYVCHIN. « Chronological Backtracking ». In : *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Sous la dir. d'Olaf BEYERSDORFF et Christoph M. WINTERSTEIGER. T. 10929. Lecture Notes in Computer Science. Springer, p. 111-121 (cf. p. 36).

[NB14] Nina NARODYTSKA et Fahiem BACCHUS. « Maximum Satisfiability Using Core-Guided MaxSAT Resolution ». In : *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. 2014, p. 2717-2723 (cf. p. 53).

[NR00] Rolf NIEDERMEIER et Peter ROSSMANITH. « New Upper Bounds for Maximum Satisfiability ». In : *Journal of Algorithms* 36.1 (2000), p. 63-88 (cf. p. 47, 48).

[PRB18] Tobias PAXIAN, Sven REIMER et Bernd BECKER. « Dynamic Polynomial Watchdog Encoding for Solving Weighted MaxSAT ». In : *Theory and Applications of Satisfiability Testing – SAT 2018*. 2018 (cf. p. 50).

[Py21] Matthieu PY. *MS-Builder*. 2021. URL : https://pageperso.lis-lab.fr/matthieu.py/en/software.html (cf. p. 64, 116).

[PCH20] Matthieu PY, Mohamed Sami CHERIF et Djamal HABET. « Towards Bridging the Gap Between SAT and Max-SAT Refutations ». In : *32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020*. IEEE, 2020, p. 137-144 (cf. p. 18, 77, 116, 117).

[PCH21a] Matthieu PY, Mohamed Sami CHERIF et Djamal HABET. « A Proof Builder for Max-SAT ». In : *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 2021 (cf. p. 18, 91, 116).

[PCH21b] Matthieu PY, Mohamed Sami CHERIF et Djamal HABET. « Computing Max-SAT Refutations using SAT Oracles ». In : *33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*. IEEE, 2021 (cf. p. 18, 67, 116, 117).

[PCH21c] Matthieu PY, Mohamed Sami CHERIF et Djamal HABET. « Des réfutations SAT aux réfutations Max-SAT ». In : *Journées Francophones de Programmation par Contraintes (JFPC)*. 2021 (cf. p. 18, 77).

[PCH21d] Matthieu PY, Mohamed Sami CHERIF et Djamal HABET. « Inferring Clauses and Formulas in Max-SAT ». In : *33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021*. IEEE, 2021 (cf. p. 18, 101, 116).

[Qui50] Willard Van Orman QUINE. *Methods of Logic*. Harvard University Press, 1950 (cf. p. 34).

[RRR98] V. RAMAN, B. RAVIKUMAR et S. S. RAO. « A Simplified NP-Complete MAX-SAT Problem ». In : *Information Processing Letters* 65 (1998), p. 1-6 (cf. p. 45).

[Rob65] John Alan ROBINSON. « A Machine-Oriented Logic Based on the Resolution Principle ». In : *Journal of the Association for Computing Machinery* 12 (1965), p. 23-41 (cf. p. 33, 38, 48).

[SBJ16] Paul SAIKKO, Jeremias BERG et Matti JÄRVISALO. « LMHS : A SAT-IP Hybrid MaxSAT Solver ». In : *Theory and Applications of Satisfiability Testing - SAT 2016*. T. 9710. Lecture Notes in Computer Science. 2016, p. 539-546 (cf. p. 57).

[Sil99] João P. Marques SILVA. « The Impact of Branching Heuristics in Propositional Satisfiability Algorithms ». In : *Proceedings of Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99*. T. 1695. Lecture Notes in Computer Science. 1999, p. 62-74 (cf. p. 37).

[SS96] João P. Marques SILVA et Karem A. SAKALLAH. « Conflict Analysis in Search Algorithms for Satisfiability ». In : IEEE Computer Society, 1996, p. 467-469 (cf. p. 36).

[SS99] João P. Marques SILVA et Karem A. SAKALLAH. « GRASP : A Search Algorithm for Propositional Satisfiability ». In : *IEEE Transactions of Computers* 48.5 (1999), p. 506-521 (cf. p. 35, 36).

[Urq01] Alasdair URQUHART. « The Complexity of Propositional Proofs ». In : *Current Trends in Theoretical Computer Science, Entering the 21th Century*. Sous la dir. de Gheorghe PAUN, Grzegorz ROZENBERG et Arto SALOMAA. 2001, p. 332-342 (cf. p. 42, 80, 81).

[Urq08] Alasdair URQUHART. « Regular and General Resolution : An Improved Separation ». In : *Proceedings of Theory and Applications of Satisfiability Testing - SAT 2008*. T. 4996. Lecture Notes in Computer Science. 2008, p. 277-290 (cf. p. 42).

[Urq11] Alasdair URQUHART. « A Near-Optimal Separation of Regular and General Resolution ». In : *SIAM Journal of Computing* 40.1 (2011), p. 107-121 (cf. p. 42).

[WF93] Richard J. WALLACE et Eugene C. FREUDER. « Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems ». In : *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Sous la dir. de David S. JOHNSON et Michael A. TRICK. T. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1993, p. 587-615 (cf. p. 58).

[Wat17] Dimitri WATEL. *Chapitre 1 : Complexité des problèmes de décision*. 2017. URL : http://dimitri.watel.free.fr/teaching/mpro_ica/courses/01_ComplexiteProblemDecision-FR.pdf (cf. p. 21).

[WHJ14] Nathan WETZLER, Marijn HEULE et Warren A. Hunt JR. « DRAT-trim : Efficient Checking and Trimming Using Expressive Clausal Proofs ». In : *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Proceedings*. Sous la dir. de Carsten SINZ et Uwe EGLY. T. 8561. Lecture Notes in Computer Science. Springer, 2014, p. 422-429 (cf. p. 39).

[Zha97] Hantao ZHANG. « SATO : An Efficient Propositional Prover ». In : *Automated Deduction - CADE-14, 14th International Conference on Automated Deduction*. Sous la dir. de William MCCUNE. T. 1249. Lecture Notes in Computer Science. 1997, p. 272-275 (cf. p. 35, 37).

[ZS00] Hantao ZHANG et Mark E. STICKEL. « Implementing the Davis-Putnam Method ». In : *Journal of Automatic Reasoning* 24 (2000), p. 277-296 (cf. p. 37).

Résumé

Dans cette thèse, on s'intéresse au problème de satisfiabilité maximum, ou problème Max-SAT, qui consiste, étant donnée une formule propositionnelle sous forme normale conjonctive, à trouver une interprétation des variables de la formule permettant de satisfaire le plus de clauses possible, ou, de manière équivalente, de falsifier le moins de clauses possible. Le système de preuve le plus utilisé dans Max-SAT est basé sur la règle d'inférence par max-résolution qui est l'adaptation pour Max-SAT de la règle de résolution utilisée pour le problème SAT. La règle de résolution déduit une nouvelle clause à partir de deux clauses qui s'opposent, ce qui permet notamment de certifier qu'une formule est insatisfiable en déduisant progressivement de nouvelles clauses jusqu'à en déduire une contradiction, représentée sous la forme d'une clause vide (on parle alors de *réfutation par résolution*). L'adaptation des réfutations par résolution pour Max-SAT sans en augmenter considérablement la taille est un problème ouvert depuis l'introduction de la max-résolution.

On propose dans cette thèse deux méthodes pour adapter n'importe quelle réfutation par résolution en réfutation valide pour Max-SAT, à laquelle on se réfère sous le terme *max-réfutation*. Une autre contribution de cette thèse est la construction de certificats qui permettent de démontrer l'optimalité d'une solution pour le problème Max-SAT. Pour générer de tels certificats, on utilise les max-réfutations que l'on est désormais capables de générer à partir des réfutations par résolution. Comme une max-réfutation permet de certifier qu'au moins une clause d'une formule doit être falsifiée, il est alors possible de certifier, en utilisant k max-réfutations, qu'au moins k clauses de cette même formule doivent être falsifiées. Cette séquence de max-réfutations, à laquelle on ajoute une interprétation des variables permettant de falsifier exactement k clauses de la formule, forme un certificat pour le problème Max-SAT. Enfin, on s'intéresse au problème qui consiste, étant donnée une formule initiale et une information donnée (clause ou formule), à inférer cette information par des règles d'inférence qui préservent l'équivalence Max-SAT. Comme la max-résolution est incomplète pour l'inférence dans Max-SAT, on propose un nouveau système de preuve ainsi qu'un algorithme permettant de construire n'importe quelle inférence de clauses ou de formules, ou de certifier qu'une telle inférence ne peut exister.

Mots clés : Intelligence Artificielle, Programmation par Contraintes, Problème Max-SAT, Problème SAT, Inférence.