Optimisation booléenne multiobjectif : complexité sous contraintes compilées et résolution via SAT

THÈSE

présentée et soutenue publiquement le 26 juin 2015

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Emmanuel Lonca

Composition du jury

Rapporteurs :	Olivier Bailleux	Université de Bourgogne
	Jin-Kao Hao	Université d'Angers
Examinateurs :	Frédéric Koriche	Université d'Artois
	Daniel Le Berre	Université d'Artois, directeur de thèse
	Pierre Marquis	Université d'Artois, co-directeur de thèse
	Patrice Perny	Université Pierre et Marie Curie

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.univ-artois.fr

Mis en page avec la classe thloria.

Remerciements

J'adresse mes remerciements à toutes les personnes m'ayant permis de manière plus ou moins directe de survivre à cette thèse. En premier lieu, je tiens bien évidemment à remercier mes directeurs et encadrants, Anne, Daniel et Pierre. Anne, merci pour ta gentillesse, et pour m'avoir rassuré en me soutenant que si, si, les meilleures thèses sont celles qui sont accomplies en trois ans et demi. Daniel, je te remercie pour le savant mélange de liberté et de coups de cravache que tu m'as adressés, qui m'ont fait évoluer du jeune étudiant de master recherche que j'étais, pour devenir un (toujours jeune) apprenti chercheur. Pierre, je te remercie pour la rigueur que tu as tenté de m'inculquer dans mon travail. À ce propos, je pense que je n'oublierai jamais le jour où, en lisant le premier paragraphe de cette thèse, tu m'as montré à quel point je pouvais l'améliorer en le formulant autrement (démonstration qui était aussi valable pour les centaines de paragraphes suivants). Cet épisode douloureux aura cependant eu le mérite de faciliter la vie de tous les lecteurs présents et futurs de ce manuscrit, ce qui inclut les deux rapporteurs de cette thèse, Olivier Bailleux et Jin-Kao Hao, que je remercie très chaleureusement pour avoir bien voulu lire ma prose en avant-première et m'avoir prodigué quelques ultimes conseils.

Je remercie aussi mes camarades de galère doctorale, qui en plus de devoir tenter eux-mêmes de survivre à leur thèse, ont réussi à m'apporter un grand soutien, en particulier dans la dernière ligne droite. Collègues, vu le nombre de blagounettes que je vous avais faites subir durant ces dernières années, je ne pense vraiment pas que ces attentions étaient nécessaires pour vous permettre de rejoindre le paradis des thésards, mais merci quand même.

Je remercie également tous les autres collègues du CRIL et de Navarre, avec qui j'ai eu l'occasion de travailler sur des problématiques de recherche ou pour des enseignements. En ce qui concerne la recherche, je tiens d'ailleurs à remercier mes collaborateurs pour m'avoir appris à travailler en trois-huit en buvant des litres de café.

Enfin, puisqu'il n'est à mon avis pas humainement possible de survivre à une thèse sans activités extérieures, je tiens à remercier les clubs sportifs et culturels qui m'ont permis de tenir. En premier lieu, je tiens à adresser mes plus chaleureux remerciements à la section « sports et loisirs » maintenue par certains de mes collègues du CRIL, dont les nombreuses activités du soir m'ont apporté la sérénité dont j'avais besoin avant de retourner au charbon le lendemain. Je tiens aussi à remercier les quizz hebdomadaires organisés dans un café arrageois (celui qui a un nom d'arbre) qui m'ont permis de conserver un minimum de culture générale et de contact avec le monde extérieur.

Et merci à ceux que j'ai oublié de remercier (y compris moi, sans qui rien de tout cela n'aurait été possible), aussi. C'est pas contre vous, mais c'est pas forcément évident de se souvenir de toutes les personnes qu'on a à remercier quand il y en a tellement.

Cette thèse est dédiée à ceux
qui durant ces dernières années
ont eu la patience de me supporter
ou le bon sens de ne plus s’y essayer.

Table des matières

Introduction générale **1**

Partie I Optimisation booléenne multiobjectif : complexité sous contraintes compilées

Chapitre 1 Problèmes de décision **7**

1.1 Calculabilité et complexité algorithmique des problèmes de décision 8
1.1.1 Machine de Turing 8
1.1.2 Hiérarchie polynomiale 11
1.2 Problème CNF-SAT 17
1.2.1 Syntaxe de la logique propositionnelle 17
1.2.2 Sémantique de la logique propositionnelle 20
1.2.3 Complexité de la détermination de la cohérence d'une formule 21
1.3 Conclusion du chapitre 23

Chapitre 2 Problèmes d'optimisation **25**

2.1 Optimisation monocritère 26
2.1.1 Notion de préférence 26
2.1.2 Fonctions de coût et d'utilité 28
2.2 Optimisation multicritère 30
2.2.1 Généralités 30
2.2.2 « Agréger puis comparer » *vs.* « comparer puis agréger » 31
2.2.3 Propriétés des fonctions d'agrégation 32
2.2.4 Prise en compte du potentiel 36
2.2.5 Fonctions d'agrégations de critères usuelles 36
2.3 Conclusion du chapitre 43

Chapitre 3 Compilation de connaissances : le langage NNF et ses sous-langages **45**
3.1 Définitions du langage NNF et de ses sous-langages 46
3.1.1 Le langage NNF et ses propriétés 46
3.1.2 Diagrammes de décision binaires 49
3.1.3 DNNF structurées 52
3.2 Choisir un langage de compilation 53
3.2.1 Requêtes 53
3.2.2 Transformations 54
3.2.3 Concision des langages étudiés 58
3.3 Conclusion du chapitre 58

Chapitre 4 Optimisation sous contraintes NNF **61**
4.1 La requête OPT 62
4.2 OPT dans le cadre de la somme des coûts et du leximax 64
4.2.1 Fonctions objectifs linéaires 65
4.2.2 Fonctions objectifs générales 70
4.2.3 Résultats de *fixed-parameter tractability* 72
4.3 Raffinement de la frontière entre optimisation simple et complexe 74
4.3.1 Agrégation via opérateurs OWA 74
4.3.2 Extensions des fonctions linéaires 77
4.3.3 Fonctions sous-modulaires 78
4.4 Conclusion du chapitre 79

Partie II Optimisation booléenne multiobjectif : résolution via SAT

Chapitre 5 Prouveurs pseudo-booléens **83**
5.1 Approches complètes *vs.* incomplètes 84
5.2 De la résolution aux prouveurs CDCL 87
5.2.1 Principe de résolution de Robinson 87
5.2.2 Algorithme DP 90
5.2.3 Algorithme DPLL 93

5.2.4 Apprentissage de clauses : les prouveurs CDCL . . . 95
5.3 Des prouveurs CDCL aux prouveurs SAT modernes . . . 99
5.3.1 Heuristiques de choix de variable . . . 99
5.3.2 Redémarrages . . . 101
5.3.3 Nettoyage des clauses apprises . . . 101
5.3.4 Structures « watched two literals » . . . 103
5.4 Prouveurs et contraintes pseudo-booléennes . . . 105
5.4.1 Contraintes pseudo-booléennes . . . 105
5.4.2 Plans coupes et résolution généralisée . . . 106
5.4.3 Intégration des contraintes pseudo-booléennes dans les prouveurs CDCL . . . 108
5.4.4 Encodages de contraintes de cardinalité par des clauses . . . 110
5.5 Détection de contraintes de cardinalité . . . 112
5.5.1 Détection statique de contraintes *AtMost*-1 et *AtMost*-2 . . . 113
5.5.2 Détection sémantique de contraintes AtMost-k . . . 116
5.5.3 Détection sémantique : preprocessing *vs.* inprocessing . . . 121
5.5.4 Résultats expérimentaux . . . 122
5.5.5 Conclusion à propos des approches proposées . . . 124
5.6 Conclusion du chapitre . . . 124

Chapitre 6 Optimisation par prouveurs pseudo-booléens **127**
6.1 Optimisation linéaire incrémentale par contrainte de borne . . . 128
6.2 Cas des fonctions d'optimisation pseudo-booléennes non linéaires . . . 131
6.2.1 Réduction vers des fonctions linéaires par ajout de contraintes . . . 131
6.2.2 Optimisation par énumération des solutions optimales d'une minorante . . . 133
6.3 Génération de contraintes paresseuses pour l'optimisation . . . 135
6.3.1 Motivation . . . 135
6.3.2 Optimisation par contraintes de borne dynamiques . . . 138
6.3.3 Expérimentations et résultats . . . 140
6.3.4 Analyse des résultats . . . 144
6.3.5 Conclusions à propos des approches proposées . . . 144
6.4 De la correction des prouveurs . . . 147
6.5 Conclusion du chapitre . . . 148

Conclusion générale **151**

Annexe A Contributions publiées dans le cadre de cette thèse **155**

Table des figures 157

Liste des tableaux 159

Liste des algorithmes 160

Bibliographie 161

Introduction générale

L'aide à la décision a pour but d'assister un décideur humain dans ses choix, voir même de prendre des décisions de manière automatique (dans ce cas, on parle plutôt de *décision automatique*). L'intérêt d'employer de telles techniques s'est imposée avec la volonté de traiter des problèmes toujours plus complexes, dépendant d'une quantité de données toujours plus importante. En effet, cette complexité et cet important volume de données empêchent un décideur humain de trouver une *bonne* solution parmi l'ensemble des solutions existantes d'un problème donné, voir même tout simplement de trouver une solution à ce problème en un temps raisonnable.

On pourrait par exemple citer les *problèmes de gestion de dépendances logicielles* (en anglais, *software dependency management problems*), nés du recours à la programmation modulaire, tels que le problème de gestion de dépendances de paquets GNU/Linux [MANCINELLI *et al.* 2006]. Le système GNU/Linux est composé d'un ensemble de logiciels (noyaux, librairies, interfaces graphiques, logiciels utilisateurs, ...) appelés *paquets*. Un paquet peut exister en plusieurs *versions*. Par exemple, $\texttt{firefox}_2$ et $\texttt{firefox}_3$ représentent le paquet `firefox` en versions 2 et 3.

Or, il existe de nombreuses restrictions sur ces paquets, telles que les *dépendances* (l'installation d'un paquet nécessite l'installation d'autres paquets), et les incompatibilités (l'installation d'un paquet implique qu'un certain nombre d'autres paquets ne soient pas installés). Un problème de gestion de dépendances est ainsi défini de la manière suivante :

- étant donné l'ensemble des paquets $\mathcal{X}$ ainsi que les contraintes qui les lient ;
- étant donné une *requête*, c'est-à-dire un ensemble de paquets à installer $\mathcal{X}_+$ et un ensemble de paquets à retirer $\mathcal{X}_-$;
- existe-t-il un ensemble de paquets $\mathcal{X}_f$ respectant la requête ($\mathcal{X}_+ \subseteq \mathcal{X}_f$ et $\mathcal{X}_f \cap \mathcal{X}_- = \varnothing$), tel que $\mathcal{X}_f$ satisfasse l'ensemble des contraintes (dépendances, incompatibilités) ? Si oui, en citer un.

Comme nous le verrons par la suite, ce problème est dit *combinatoire*, c'est-à-dire que dans le pire des cas, il faudra tester un nombre d'ensembles de paquets exponentiel dans le nombre de paquets de $\mathcal{X}$ (en $O(2^{|\mathcal{X}|})$) afin de résoudre un problème donné. Étant donné qu'à l'heure où ces lignes sont écrites, certaines distributions GNU/Linux atteignent 37500 paquets (cas de Debian), le nombre de possibilités à tester est potentiellement de 10^{1000}, et donc bien trop important pour un décideur humain.

La nécessité de l'aide à la décision apparaît comme encore plus criante lorsqu'on souhaite obtenir une *bonne* solution d'un problème combinatoire, voir même *une des meilleures* selon un *critère* donné. Dans ce cas, il ne suffit pas de trouver une solution réalisable, mais de trouver une solution réalisable telle que nulle autre solution ne soit considérée comme meilleure. On passe alors d'un *problème de décision* (déterminer l'existence d'une solution) à un *problème d'optimisation monocritère* (déterminer une des meilleures solutions possibles selon mon critère, si une solution existe). Dans le cadre d'un problème de gestion de dépendances, on pourrait par exemple souhaiter que le nombre de paquets qui ne sont pas à jour (c'est-à-dire, un paquet installé pour lequel il existe une version postérieure qui n'est pas installée) soit minimal, et se retrouver ainsi dans le cadre de l'optimisation monocritère.

Un décideur peut aussi souhaiter considérer plusieurs critères à la fois, et ainsi faire passer le problème de décision initial concernant l'existence d'une solution à un problème d'*optimisation multicritère*. La difficulté de ce type de problèmes provient du fait que les critères sont généralement antagonistes (les bonnes solutions selon un critère vont être généralement mauvaises sur au moins un des autres critères considérés), et qu'il n'existe donc pas de solution absolument meilleure que toutes les autres. Dans ce cas, il s'agit plutôt de déterminer une solution qui offre un bon *compromis* entre les critères.

Ce document est divisé en deux parties. Dans la première, nous présentons dans un premier temps les notions et résultats théoriques de l'état de l'art en ce qui concerne les problématiques de décision et d'optimisation. Nous présentons notamment des résultats de la théorie de la complexité, en particulier ceux centrés sur les concepts utiles à notre étude, ainsi que la logique propositionnelle et les différents langages qui lui sont associés et qui sont employés usuellement pour encoder de l'information. Nous

présentons aussi dans cette première partie les principales méthodes d'agrégation de critères disponibles dans l'état de l'art. Une fois l'ensemble de ces préliminaires formels décrits et expliqués, nous présentons nos résultats théoriques concernant l'optimisation de diverses fonctions quand les contraintes sont compilées. Un ensemble de langages propositionnels usuels en pour la compilation de contrainte est considéré.

Dans un second temps, ce document traite de considérations pratiques pour l'optimisation de fonctions objectifs sous contraintes exprimées dans un fragment standard, celui des formules sous forme normale conjonctive ; nous étendons aussi notre étude aux formules créées par une conjonction de contraintes pseudo-booléennes, dont les clauses des formules CNF sont des cas particuliers. Nous présentons tout d'abord brièvement les problématiques issues de la naissance des prouveurs SAT, les logiciels capables de déterminer de manière automatique si un problème modélisé en tant que formule sous forme normale conjonctive admet une solution réalisable. Nous présenterons ensuite nos contributions à ces prouveurs ; celles-ci proposent des prétraitements de l'information initiale (le problème) dans le but d'améliorer l'efficacité de la résolution des problèmes concernés par les prouveurs. Après cela, nous présentons la manière dont les prouveurs SAT sont utilisés pour fournir des algorithmes de résolution de problèmes d'optimisation de fonctions sous contraintes booléennes, puis nous détaillons les travaux que nous avons réalisés dans le but de rendre l'utilisation des prouveurs dans ces algorithmes d'optimisation plus intelligente.

Première partie

Optimisation booléenne multiobjectif : complexité sous contraintes compilées

Chapitre 1

Problèmes de décision

Sommaire

1.1 Calculabilité et complexité algorithmique des problèmes de décision 8

1.1.1 Machine de Turing 8

1.1.2 Hiérarchie polynomiale 11

1.2 Problème CNF-SAT 17

1.2.1 Syntaxe de la logique propositionnelle 17

1.2.2 Sémantique de la logique propositionnelle 20

1.2.3 Complexité de la détermination de la cohérence d’une formule 21

1.3 Conclusion du chapitre 23

Dans ce chapitre, nous étudions les problèmes de décision, notamment en ce qui concerne leur complexité théorique. Nous présentons ensuite le langage que nous allons employer pour modéliser les problèmes, la logique propositionnelle, et plus particulièrement les formules sous forme normale conjonctive qui permettent de représenter simplement les problèmes concrets que l'on souhaite résoudre de manière automatisée.

Un problème de décision est un problème dont le but est de décider de l'existence ou non d'une solution réalisable dans l'espace des choix possibles. Typiquement, il peut être formulé sous la forme d'une expression « mon problème admet-il oui ou non une solution ? ». La résolution de tels problèmes ne s'applique pas uniquement aux problèmes de décision « purs », dans le sens où certains problèmes, comme les problèmes d'optimisation, peuvent être résolus en décidant de l'existence ou non d'une solution pour une succession de problèmes de décision.

Il est important de noter que certains problèmes sont théoriquement plus difficiles à résoudre que d'autres, la « difficulté » correspondant au nombre d'opérations à appliquer en fonction de la taille des données en entrée dans le pire des cas pour résoudre le problème. Pour cela, on classe tout d'abord les problèmes selon qu'ils sont calculables (existe-t-il un programme capable de résoudre le problème ?) ou non. Ensuite, parmi les problèmes calculables, une échelle de complexité définie récursivement, appelée la *hiérarchie polynomiale* permet de classer les problèmes des plus faciles aux plus difficiles à résoudre théoriquement.

1.1 Calculabilité et complexité algorithmique des problèmes de décision

Il existe un grand nombre de problèmes de décision, dont la difficulté de résolution semble différer. Dans cette section, nous présentons les bases de la théorie de la calculabilité et de la complexité afin de définir plusieurs classes de difficulté théoriques définies dans l'état de l'art. Ces classes peuvent être définies en utilisant un modèle théorique de calcul bien connu, la *machine de Turing* [TURING 1936].

1.1.1 Machine de Turing

Une machine de Turing est un modèle théorique permettant de représenter de manière abstraite une machine de calcul, comme un ordinateur. Bien que ce modèle fut inventé avant le premier ordinateur, celui-ci le représente de manière assez peu éloignée des machines que nous connaissons aujourd'hui. En effet, la machine de Turing est notamment basée sur l'abstraction d'un ruban servant à conserver des données (la mémoire de nos machines modernes), et d'une fonction de transition, qui pourrait être considérée comme le cerveau de la machine (le processeur de nos machines modernes).

De manière mathématique, une définition de la machine de Turing est la suivante.

Définition 1.1 (Machine de Turing) *Une machine de Turing est un septuplet* $(Q, \Gamma, \Sigma, B, q_0, F, \delta)$ *tel que :*

- Q *est un ensemble fini d'états ;*
- Γ *est l'alphabet fini du ruban ;*
- $\Sigma \subseteq \Gamma$ *est l'alphabet des symboles d'entrée ;*
- $B \in \Gamma \setminus \Sigma$ *est un symbole particulier, dit* blanc *;*
- $q_0 \in Q$ *est l'état initial ;*
- $F \subseteq Q$ *est l'ensemble des états finaux ;*
- $\delta : (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, \rightarrow\})$ *est la « fonction » de transition.*

Concrètement, une machine de Turing peut être vue comme un ruban infini, divisé en cases, chacune contenant un symbole de l'alphabet Γ. En plus de ce ruban, il faut garder à l'esprit l'état courant ϵ (la

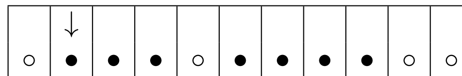
configuration courante du processeur), ainsi que la case courante, qui est habituellement représentée à l'aide d'une *tête de lecture/écriture* pour le ruban.

Au lancement d'un calcul sur la machine de Turing, on écrit le mot d'entrée (une succession de symboles de l'alphabet des symboles d'entrée Σ) sur le ruban, on place la tête de lecture sur le premier symbole du mot, et l'état courant est initialisé à l'état initial, q_0. Les cases du ruban non utilisées contiennent le caractère blanc B.

La suite du calcul est composé d'étapes, dont le déroulement est le suivant :

1. on lit le symbole γ sous la tête de lecture ;
2. s'il n'existe pas de couple (ϵ, γ) admettant une image dans la fonction de transition, alors le calcul s'arrête, et la réponse est celle écrite sur le ruban ;
3. on récupère le triplet $(\epsilon', \gamma', t) = \delta(\epsilon, \gamma)$;
4. on écrit sous la tête de lecture/écriture le symbole γ' ;
5. si $t = \leftarrow$ (resp. $\rightarrow$), on déplace la tête de lecture d'une case vers la gauche (resp. droite) ;
6. on met à jour l'état courant, qui est maintenant ϵ' ; s'il s'agit d'un état final ($\epsilon' \in F$), le calcul s'arrête, et la réponse est celle écrite sur le ruban.

On présente ci-dessous une machine de Turing capable de calculer la somme de deux entiers positifs. L'alphabet est composé de deux symboles, $\bullet$ et $\circ$, ce dernier étant le caractère blanc. Le nombre n sera codé comme une succession de n fois le caractère $\bullet$, suivi du caractère $\circ$. Ainsi, l'entrée pour la somme des nombres 3 et 4 est codée de la manière suivante :

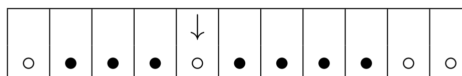


La table de transition suivante permet de faire calculer à cette machine la somme des deux nombres en entrée, en considérant que l'état initial est l'état 0. L'ensemble des états finaux est composé du seul état 3.

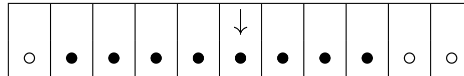
ϵ	γ	ϵ'	γ'	t
0	$\bullet$	0	$\bullet$	$\rightarrow$
0	$\circ$	1	$\bullet$	$\rightarrow$
1	$\bullet$	1	$\bullet$	$\rightarrow$
1	$\circ$	2	$\circ$	$\leftarrow$
2	$\circ$	3	$\circ$	$\leftarrow$
3	HALT			

La table de transition permet en fait à la machine de Turing de « déplacer » le symbole $\bullet$ final à la place du symbole $\circ$ placé entre les deux nombres. Le déroulement est le suivant :

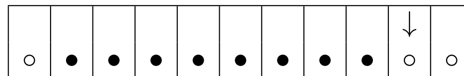
1. tant qu'on est dans l'état 0 (parcours du premier nombre) et qu'on lit $\bullet$, on écrit $\bullet$, on se déplace vers la droite, et on reste dans l'état 0 ;



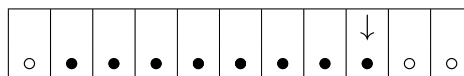
2. une fois le parcours terminé, on se retrouve dans le cas où on lit le caractère ∘ alors qu'on se trouve toujours dans l'état 0 ; on écrit le caractère •, on se déplace vers la droite, et on va dans l'état 1 (parcours du deuxième nombre) ;



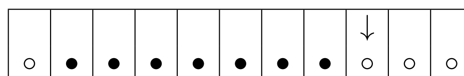
3. tant qu'on est dans l'état 1 et qu'on lit le caractère •, on écrit le caractère •, on se déplace vers la droite, et on reste dans l'état 1 ;



4. une fois le parcours terminé, on se retrouve dans le cas où on lit le caractère ∘ alors qu'on se trouve toujours dans l'état 1 ; on écrit le caractère ∘, on se déplace vers la gauche, et on va dans l'état 2 ;



5. il ne reste alors qu'à supprimer le dernier caractère •, qui est placé sous la tête de lecture ; on écrit le caractère ∘, et on va dans l'état 3, qui est un état final : le calcul s'arrête et on renvoie le contenu du ruban, qui contient bien $4 + 3 = 7$ fois le caractère •.



La machine de Turing est un outil très puissant en théorie de la calculabilité et de la complexité. Ce modèle simple permet par exemple de montrer que certains problèmes ne sont pas calculables, c'est-à-dire qu'il n'existe pas de machine de Turing permettant pour toute entrée de donner la réponse attendue. Ce résultat provient de la preuve qu'il n'est pas possible de créer un programme `HALT`, prenant en paramètres un programme et une entrée, qui répond 1 si le programme en paramètre s'arrête sur l'entrée, et qui répond 0 sinon [TURING 1936].

Ce résultat nous permet déjà de séparer les problèmes selon deux classes distinctes : les problèmes *décidables* (ou calculables), et les problèmes *indécidables*. Parmi les problèmes décidables, ces mêmes machines permettent aussi de tracer une frontière entre les problèmes dits *traitables* (comme le problème 2-SAT, [COOK 1971]), et ceux qui ne le sont pas (comme les problèmes k-SAT pour $k > 2$).

Pour tous les problèmes décidables, il existe donc une machine de Turing capable de calculer une réponse à un problème pour toute entrée. En revanche, on peut séparer les machines de Turing en deux familles :

– les machines de Turing déterministes, pour lesquelles la fonction de transition δ est une application (pour chaque couple (ϵ, γ), il existe au plus une image par δ) ;
– les machines de Turing non déterministes, pour lesquelles cette condition d'unicité d'image n'est pas nécessaire (ainsi, on peut considérer les machines déterministes comme un sous-ensemble des machines non déterministes).

Le non-déterminisme apparaît ici comme un obstacle, puisqu'il implique de potentiellement devoir tester, à chaque étape, l'intégralité des transitions disponibles, chose qui n'est pas vraie pour les machines déterministes. Plus précisément, une machine de Turing non-déterministe résout un problème de décision si pour toute instance positive de ce problème au moins une séquence de transitions aboutit à la réponse « oui » et si pour toute instance négative toutes les séquences d'exécution aboutissent à la réponse « non ». On dit d'un problème pour lequel il existe une machine de Turing déterministe qui le décide en temps polynomial qu'il est traitable, contrairement aux problèmes décidables pour lesquels il n'en existe pas. Cette séparation a par la suite été étendue afin d'avoir un tri plus fin des problèmes, qui sont maintenant classés sur une échelle de complexité appelée la *hiérarchie polynomiale*.

1.1.2 Hiérarchie polynomiale

La complexité d'un problème peut être déterminée par deux mesures sur les machines de Turing capables de résoudre ce problème. Ces deux mesures se basent sur la notation $\mathcal{O}(f(n))$, qui signifie que dans le pire des cas, le nombre d'étapes ou de cases nécessaires au déroulement de l'algorithme est asymptotiquement majoré par $f(n)$ à un facteur constant près. Plus formellement, on dit qu'une fonction $g(n)$ est en $\mathcal{O}(f(n))$, où n est la taille du mot d'entrée, si et seulement si il existe une constante $c < \infty$ telle que :

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = c.$$

La première de ces deux mesures se base sur le « temps de résolution », matérialisé dans le cadre d'une machine de Turing par le nombre d'étapes nécessaires à sa résolution.

Définition 1.2 (Complexité temporelle) *Un problème appartient à la classe de complexité* $\mathsf{TIME}(f(n))$ *si et seulement si il existe une machine de Turing déterministe capable de le résoudre en un nombre d'étapes en* $\mathcal{O}(f(n))$ *pour un mot de taille* n*.*

La seconde mesure se base sur l'espace nécessaire à la résolution du problème par la machine de Turing, c'est-à-dire le nombre de cases du ruban utilisées pour la résolution du problème. Il est à noter que la complexité spatiale joue le rôle de borne inférieure pour la complexité temporelle, puisque qu'il faut au minimum un nombre n d'étapes pour écrire dans n cases du ruban.

Définition 1.3 (Complexité spatiale) *Un problème appartient à la classe de complexité* $\mathsf{SPACE}(f(n))$ *si et seulement si il existe une machine de Turing déterministe capable de le résoudre en utilisant un nombre de cases du ruban en* $\mathcal{O}(f(n))$ *pour un mot de taille* n*.*

À partir de ces mesures sur les machines de Turing, des classes de problèmes ont été définies afin d'évaluer la complexité théorique des problèmes de décision. Ainsi, les moins difficiles à résoudre se retrouvent être ceux pour lesquels une machine de Turing déterministe est capable de calculer une réponse pour chacune des entrées possibles en temps polynomial. Ces problèmes sont regroupés dans la classe P [GAREY & JOHNSON 1979, PAPADIMITRIOU 1994].

Définition 1.4 (Classe P) *La classe* P *est définie comme l'ensemble des problèmes de décision pouvant être résolus par une machine de Turing déterministe en temps polynomial dans la taille de l'entrée.*

Bien qu'un nombre important de problèmes appartiennent à la classe P (on pourrait par exemple citer les problèmes de gestion de dépendances logicielles [SYRJÄNEN 1999, ARGELICH *et al.* 2010, MANCINELLI *et al.* 2006] pour lesquels aucun paquet n'est en conflit avec un autre), il en existe cependant un certain nombre pour lesquels aucune machine déterministe s'exécutant en temps polynomial

n'existe et d'autres encore pour lesquels aucune machine déterministe n'a pu être déterminée (par exemple, les problèmes de gestion de dépendances logicielles en considérant cette fois des conflits). De ce fait, il est nécessaire de considérer des classes de complexité plus générales, comme la classe NP [GAREY & JOHNSON 1979, PAPADIMITRIOU 1994].

Définition 1.5 (Classe NP) *La classe* NP *est définie comme l'ensemble des problèmes de décision pouvant être résolus par une machine de Turing non déterministe en temps polynomial dans la taille de l'entrée.*

Cette classe est définie de la même manière que la classe P, à la différence près que le déterminisme de la machine de Turing n'est pas requis [COOK 1971]. De ce fait, on obtient l'inclusion $\mathsf{P} \subseteq \mathsf{NP}$. En revanche, l'inclusion inverse n'ayant jamais été démontrée ni infirmée, on ne sait pas si les classes P et NP sont confondues ; cependant, la conjecture est que $\mathsf{P} \neq \mathsf{NP}$.

Les problèmes de la classe NP correspondent en fait à des problèmes pour lesquels il existe une machine de Turing déterministe dont la complexité temporelle est *simplement exponentielle* dans la taille de l'entrée [LEWIS & PAPADIMITRIOU 1998], c'est-à-dire dont le nombre d'étapes est en $\mathcal{O}(a^{f(n)})$, où a est une constante finie strictement supérieure à 1 et $f(n)$ est un polynôme de n (précisons que les problèmes pour lesquels le nombre d'étapes n'est pas *simplement* exponentiel, ce qui est par exemple le cas d'un nombre d'étapes en $\mathcal{O}(a^{f(n)^{f'(n)}})$, font partie d'une classe plus générale que NP).

En pratique, les problèmes de la classe NP sont aussi ceux pour lesquels, étant donné un énoncé du problème et une solution potentielle, il est possible de déterminer en temps polynomial si la solution potentielle du problème, qui doit être de taille polynomiale dans la taille de l'entrée, est effectivement une solution du problème. On dit que les problèmes de la classe NP sont ceux admettant un *problème certificat* dans P, c'est-à-dire un problème de décision prenant en entrée l'instance du problème de la classe NP et un certificat potentiel, et qui retourne « oui » si et seulement si il s'agit effectivement d'un certificat pour l'instance.

Exemple 1.1 (Certificat polynomial) *La figure 1.1 montre un exemple de problème de gestion de dépendances au format CUDF [*TREINEN & ZACCHIROLI *2009]. Dans cet exemple :*

- *l'ensemble des couples (paquet,version) considérés (on parle d'*unités installables*) est* $\{a_1, a_2, b_1, b_2, b_3, c_1, d_1, e_1, e_2, e_3, e_4, f_1, g_1, h_1, i_1, j_1\}$ *;*
- *les unités installables présentes à l'état initial sont* a_1*,* b_1*,* c_1 *et* d_1 *;*
- *les paquets existants en plusieurs versions ne peuvent être installés que sous une unique version,* a_2 *ne peut être installé en même temps que le paquet* d*,* b_3 *ne peut être installé en même temps que le paquet* c*,* j_1 *ne peut être installé en même temps qu'un des paquets* b*,* c *ou* d *;*
- e_1 *requiert l'installation des paquets* f*,* g*,* h *et* i *;* e_2 *requiert l'installation des paquets* f*,* g*,* h*,* a *en version supérieure à 1 et* b *en version supérieure à 1 ;* e_3 *requiert l'installation des paquets* f*,* g*,* a *en version supérieure à 1 et* b *en version supérieure à 2 ;* e_4 *requiert l'installation des paquets* f*,* g*,* h *;*
- *à l'état final, le paquet* e *doit être installé.*

Supposons que l'on dispose d'un ensemble de paquets dont on souhaite déterminer si oui ou non cet ensemble est une solution au problème de la figure 1.1. Il suffit pour cela de vérifier, pour chacune des contraintes du problème, si elle est violée ou non en parcourant l'ensemble des paquets de la solution potentielle, ce qui peut se faire en temps polynomial dans la taille du problème. Or, cet ensemble est solution du problème si et seulement si aucune des contraintes n'est violée ; on dispose alors ici d'un certificat dans P *pour notre problème, qui appartient donc à la classe de complexité* NP.

Une autre manière de prouver l'appartenance d'un problème de décision Φ à la classe NP consiste à montrer qu'on peut réduire ce problème à un autre problème de décision Ψ de la classe NP en temps polynomial. On cherche dans ce cas à montrer l'existence d'une *réduction fonctionnelle polynomiale* de Φ

```
package: a
version: 1
conflicts: a
installed: true

package: a
version: 2
conflicts: a, d

package: b
version: 1
conflicts: b
installed: true

package: b
version: 2
conflicts: b

package: b
version: 3
conflicts: b, c

package: c
version: 1
installed: true

package: d
version: 1
installed: true

package: e
version: 1
depends: f, h, i
conflicts: e

package: e
version: 2
depends: f, h, a > 1, b > 1
conflicts: e

package: e
version: 3
depends: f, a > 1, b > 2
conflicts: e

package: e
version: 4
depends: f, j
conflicts: e

package: f
version: 1

package: g
version: 1
conflicts: b, c, d

package: h
version: 1

package: i
version: 1

request: install package e
install: e
```

FIGURE 1.1 – Illustration d'un problème de gestion de dépendances au format CUDF.

vers Ψ, telle que les instances acceptées pour Φ (on parle *d'instances positives*) soient exactement celles acceptées pour Ψ, modulo la transformation induite par la réduction fonctionnelle. Ce raisonnement est valable pour la classe NP, ainsi que pour les autres classes de complexité que nous présentons dans cette section. Fournir une réduction fonctionnelle permet aussi de borner la complexité d'un problème ; en effet, en partant d'un problème Ψ, montrer qu'il existe un problème Φ pour lequel il existe une réduction polynomiale vers Ψ démontre que Φ est au plus aussi difficile que Ψ.

Définition 1.6 (Réduction fonctionnelle polynomiale) *Soient deux problèmes Φ et Ψ, et un algorithme f qui prend en entrée des instances de Φ et qui retourne des instances de Ψ. f est appelée réduction fonctionnelle polynomiale de Φ vers Ψ si et seulement si :*
- *f est un algorithme en temps polynomial ;*
- *ϕ est une instance positive de Φ si et seulement si $f(\phi)$ est une instance positive de Ψ.*

Étant donné une classe de complexité C, on définit aussi la classe coC des problèmes dont le problème complémentaire appartient à la classe C. Un problème de décision est dit complémentaire d'un autre si et seulement si les entrées donnant lieu à la réponse « oui » de l'un sont celles donnant lieu à la réponse « non » de l'autre, et vice-versa.

Définition 1.7 (Classe de complexité complémentaire) *Soit* C *une classe de complexité. La classe* coC *est définie comme l'ensemble des problèmes pour lesquels le problème complémentaire appartient à la classe* C*.*

Exemple 1.2 (Preuve d'appartenance à une classe de complexité complémentaire) *Le problème P de gestion de dépendances logicielles défini plus haut correspond au problème de confirmer ou infirmer l'existence d'un ensemble d'unités installables tel que toutes les contraintes du problème soient satisfaites. On définit le problème complémentaire P' comme le problème logiquement inverse de P, c'est-à-dire de déterminer que pour tout ensemble d'unités installables, il existe au moins une contrainte qui n'est pas satisfaite. Puisque P est dans* NP*, on conclut que P' est dans* coNP*.*

En ce qui concerne la classe P et son complémentaire, on a en fait P = coP. En revanche, on conjecture que ceci est faux en ce qui concerne la classe NP (cette nouvelle conjecture est en fait impliquée par la précédente, P $\neq$ NP). Ainsi, les trois classes que nous venons de définir permettent de déduire les inclusions suivantes :
- P $\subseteq$ NP
- P $\subseteq$ coNP

Ces inclusions nous indiquent que tout problème de la classe NP ou de la classe coNP est au moins aussi difficile qu'un problème de la classe P ; on dit que les problèmes appartenant à la classe NP ou à la classe coNP sont *P-difficiles*.

Définition 1.8 (C-difficulté) *Soit* C *une classe de complexité. On dit qu'un problème est C-difficile si et seulement si il est au moins aussi difficile que tous les problèmes de la classe* C *(il existe une réduction fonctionnelle polynomiale de tout problème de la classe C vers le problème en question).*

Exemple 1.3 (Réduction fonctionnelle polynomiale, preuve de C-difficulté) *On considère une formule* CNF *contenant p clauses $\alpha_1 \wedge \ldots \wedge \alpha_p$ et n variables. Nous décrivons ici un algorithme permettant de transformer le problème de rechercher un modèle dans une formule* CNF *(problème décrit à la section 1.2.2) en un problème de gestion de dépendances logicielles.*

Nous allons considérer $2n$ unités installables, chacune correspondant à un littéral de la formule CNF *; l'unité i correspond au littéral positif de la variable i, l'unité $-i$ correspond au littéral négatif de*

la variable i. Ainsi, pour toute variable i de la formule, on définit l'unité installable de package i et de version 1 comme étant en conflit avec l'unité installable de package $-i$ et de version 1, et vice versa.

Ensuite, pour chaque clause α_i, on ajoute des dépendances pour chacun des littéraux l qui la composent : pour tout littéral l, on ajoute à l'unité installable $-l$, de version 1, une dépendance correspondant à la disjonction des unités installables correspondant aux autres littéraux de la clause.

Cet algorithme transforme en temps polynomial un problème quelconque de satisfaction de formules CNF *en un problème de gestion de dépendances logicielles, tel que toute solution d'un problème peut être transformée en temps polynomial en une solution de l'autre. On a donc ici une réduction polynomiale qui prouve que le problème de gestion de dépendances logicielles est au moins aussi difficile à résoudre que le problème de satisfaction d'une formule* CNF*. De ce fait, on conclut que le problème de gestion de dépendances logicielles est* NP*-difficile.*

Nous disposons donc d'un outil nous permettant de borner la complexité d'un problème « par le haut » (les réductions fonctionnelles polynomiales), et permettant de la borner « par le bas » (la notion de C-difficulté). Cet encadrement nous permet de déterminer les problèmes les plus difficiles d'une classe, et donc d'introduire la notion de C-complétude, définie formellement de la façon suivante.

Définition 1.9 (C-complétude) *Un problème Φ appartenant à* C *est complet pour sa classe de complexité* C *si et seulement si pour tout problème $\Psi \in C$, il existe une réduction fonctionnelle polynomiale de Ψ vers Φ. Un tel problème est dit C-complet.*

Exemple 1.4 (Preuve de C-complétude) *Dans les exemples précédents, nous avons montré que le problème de gestion de dépendances logicielles était inclus dans la classe de complexité* NP*, et qu'il était aussi* NP*-difficile. Ce problème fait donc partie des problèmes les plus difficiles de la classe* NP*, et est donc* NP*-complet.*

Historiquement, le premier problème dont la NP-complétude a été prouvée est le problème de satisfaction de formules CNF, communément appelé « problème SAT » [COOK 1971].

Nous avons présenté trois classes de complexité (P, NP, coNP), ainsi que des outils permettant de comparer la complexité de deux problèmes donnés. Cependant, il existe des problèmes pour lesquels il n'existe pas de machine de Turing non déterministe pouvant les résoudre en temps polynomial, ce qui indique qu'il sont plus difficiles que les problèmes de la classe NP. C'est pourquoi il est nécessaire de définir d'autres classes de complexité. À cette fin, nous allons encore une fois nous appuyer sur les machines de Turing, et plus précisément sur la notion de machine de Turing à oracle [TURING 1939].

Définition 1.10 (Machine de Turing à oracle) *Une machine de Turing à oracle est une machine de Turing (déterministe ou non) à laquelle 3 états sont ajoutées : $q_?$ qui sert à interroger l'oracle, q_{oui} et q_{non} qui représentent la réponse de l'oracle. Un langage oracle A est également défini et un ruban de la machine, appelé bande de l'oracle, lui est réservé. Cet oracle permet de décider si un élément x appartient à A en une étape de calcul.*

Un oracle A permet donc de décider en une étape de calcul si un mot appartient à un langage A (donc, de fournir une réponse à un problème de décision) en temps constant. Ce langage peut appartenir à une classe de complexité quelconque : de ce fait, supposer disposer d'un oracle pour la classe NP permet de résoudre n'importe quel problème de décision de la classe NP en une unique étape de calcul. On définit alors de nouvelles classes de complexité paramétrées par un oracle.

Définition 1.11 (Classe C^A) *Soit* C *une classe de complexité quelconque déterministe ou non.* C^A *est la classe des langages décidés dans un temps borné comme dans* C*, par le même type de machine que* C *mais munie d'un oracle A.*

À partir de ces machines, une échelle de complexité a été bâtie. Cette échelle est nommée *hiérarchie polynomiale*. Cette hiérarchie est construite de manière récursive, chaque nouvelle classe de complexité étant définie à partir de la précédente, dont les problèmes sont plus simples.

Définition 1.12 (Hiérarchie polynomiale) *La hiérarchie polynomiale [*STOCKMEYER *1976], est une classe de complexité (nommée* PH*) définie récursivement à partir de ses sous-classes à l'aide des machines de Turing avec oracle de la manière suivante :*
- $\Delta_0^{\mathsf{P}} = \Sigma_0^{\mathsf{P}} = \Pi_0^{\mathsf{P}} = \mathsf{P}$ *;*
- $\Delta_{k+1}^{\mathsf{P}} = \mathsf{P}^{\Sigma_k^{\mathsf{P}}}$ *;*
- $\Sigma_{k+1}^{\mathsf{P}} = \mathsf{NP}^{\Sigma_k^{\mathsf{P}}}$ *;*
- $\Pi_{k+1}^{\mathsf{P}} = \mathsf{co}\Sigma_{k+1}^{\mathsf{P}}$.

La hiérarchie polynomiale est illustrée à la figure 1.2 ; sur cette figure, la zone en gris représente les classes considérées dans cette thèse. On a en particulier $\Sigma_1^{\mathsf{P}} = \mathsf{NP}$ et $\Pi_1^{\mathsf{P}} = \mathsf{coNP}$. La classe Δ_2^{P} est aussi fréquemment désignée via sa définition, c'est-à-dire P^{NP}. En ce qui concerne les processus d'optimisation, nous verrons par la suite que de nombreux algorithmes sont basés sur la résolution d'un certain nombre de problèmes de décision de la classe NP, suivie de la résolution d'un problème de la classe coNP : ces problèmes sont répertoriés comme appartenant à la classe D^{P} [PAPADIMITRIOU & YANNAKAKIS 1984], qui est elle-même incluse dans la classe Δ_2^{P}. Une illustration de la classe D^{P} est donnée à la figure 1.3.

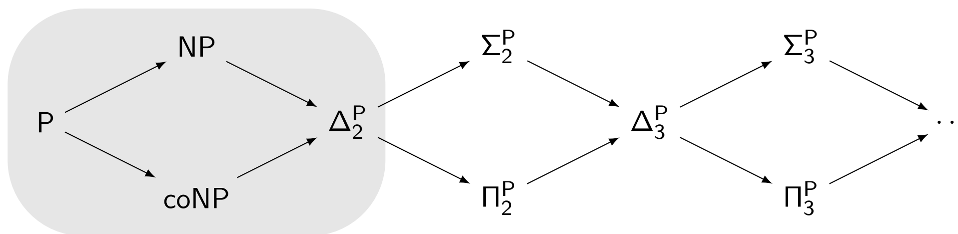


FIGURE 1.2 – Illustration de la hiérarchie polynomiale. En gris, la zone correspondant aux niveaux 0 et 1 de la hiérarchie polynomiale.

La conjecture présentée plus haut (il existerait certains problèmes de la classe NP ne pouvant être résolus par une machine de Turing déterministe en temps polynomial) est répertoriée dans la littérature comme la *conjecture* $\mathsf{P}\neq\mathsf{NP}$, et fait figure de question fondamentale en théorie de la complexité. Sa réalisation (ou non) aurait un impact majeur aussi bien théoriquement qu'en pratique, puisque s'il était avéré qu'elle soit fausse, on aurait un effondrement de la hiérarchie polynomiale dès sa base. Cela signifie que tous les problèmes qui la composent se révéleraient aussi difficiles les uns que les autres. Bien que ceci puisse être vu comme un résultat désirable, cela poserait de nombreux problèmes en terme de sécurité. En effet, certains domaines, comme la cryptographie, sont basés sur la difficulté théorique de problèmes de décision : de nombreuses méthodes connues, comme l'algorithme RSA [RIVEST *et al.* 1983], perdraient tout leur intérêt si la conjecture se révélait fausse.

La nature récursive de la hiérarchie polynomiale implique qu'il est nécessaire de connaître un problème de la classe NP (c'est-à-dire, Σ_1^P) qui n'appartienne pas à la classe P (donc, un problème NP-complet) afin d'obtenir la première étape inductive. Le premier problème dont la NP-complétude a été prouvée est le problème de satisfaction de formules CNF, communément appelé « problème SAT » [COOK 1971].

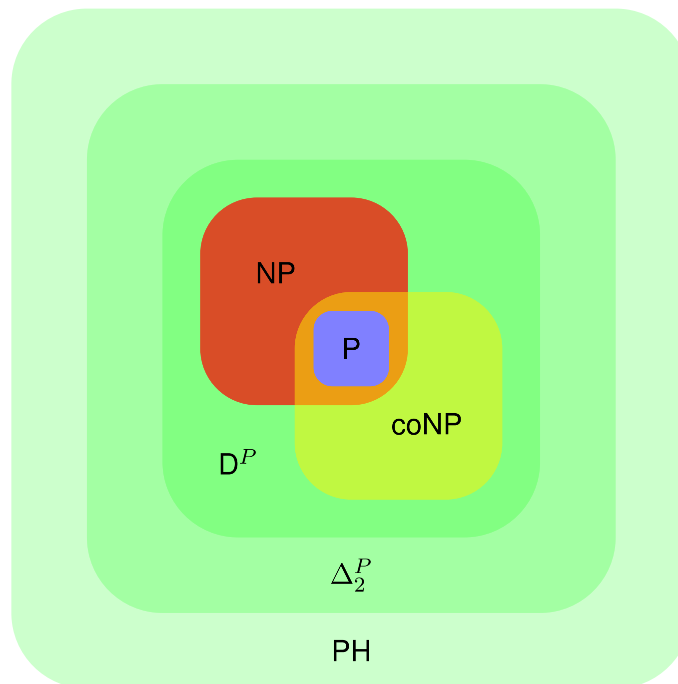


FIGURE 1.3 – Illustration des deux premiers niveaux de la hiérarchie polynomiale.

1.2 Problème CNF-SAT

Le problème SAT prend en paramètre des formules appelées *formules* CNF, qui sont des formules particulières d'un langage de la *logique propositionnelle* [BOOLE 1854].

1.2.1 Syntaxe de la logique propositionnelle

Afin de pouvoir représenter des informations, une formule propositionnelle contient un certain nombre de variables. Ces variables ne peuvent prendre que deux valeurs (appelées *valeurs de vérité*, usuellement représentées par *vrai* et *faux*, ou par les valeurs numériques 0 et 1). Cet ensemble de valeurs, bien que semblant restreint, permet par exemple de représenter de manière immédiate des faits. Par exemple, dans le cadre d'un problème de gestion de dépendances, on pourrait définir une variable par unité installable ; dans ce cas, la valeur *vrai* pourrait signifier que l'unité est installée sur la machine de l'utilisateur, quand la valeur *faux* signifierait qu'elle ne l'est pas. Dans l'exemple de la figure 1.1, on aurait ainsi 16 variables propositionnelles.

Définition 1.13 (Variable propositionnelle) *Une variable propositionnelle, ou atome, ou variable booléenne, est une variable pouvant prendre les valeurs* vrai *(pouvant être représentée par* 1 *ou* $\top$*) ou* faux *(pouvant être représentée par* 0 *ou* $\bot$*).*

Les relations entre les variables sont codées en utilisant des connecteurs logiques. Nous verrons par la suite que ces connecteurs s'appliquent en fait à des formules, et que les variables sont elles-mêmes des formules.

Définition 1.14 (Connecteur logique) *Un connecteur logique est un opérateur permettant de combiner un ensemble donné de formules. Le nombre de formules qu'un connecteur est capable de combiner est* l'arité *du connecteur.*

$$\begin{array}{ll} & \neg e_1 \vee (f_1 \wedge h_1) \\ \wedge & \neg e_2 \vee (f_1 \wedge h_1 \wedge a_2 \wedge (b_2 \vee b_3)) \\ \wedge & \neg e_3 \vee (f_1 \wedge a_2 \wedge b_2) \\ \wedge & \neg e_4 \vee (f_1 \wedge j_1) \\ \wedge & \neg a_1 \vee \neg a_2 \\ \wedge & \neg b_1 \vee \neg(b_2 \vee b_3 \vee g_1) \\ \wedge & \neg b_2 \vee \neg(b_3 \vee g_1) \\ \wedge & \neg b_3 \vee \neg(c_1 \vee g_1) \\ \wedge & \neg c_1 \vee \neg g_1 \\ \wedge & \neg d_1 \vee \neg(a_2 \vee g_1) \\ \wedge & \neg e_1 \vee \neg(e_2 \vee e_3 \vee e_4) \\ \wedge & \neg e_2 \vee \neg(e_3 \vee e_4) \\ \wedge & \neg e_3 \vee \neg e_4 \end{array}$$

(a) une formule propositionnelle

$$\begin{array}{ll} & \neg e_1 \vee (f_1 \wedge h_1) \\ \wedge & \neg e_2 \vee (f_1 \wedge h_1 \wedge a_2 \wedge (b_2 \vee b_3)) \\ \wedge & \neg e_3 \vee (f_1 \wedge a_2 \wedge b_2) \\ \wedge & \neg e_4 \vee (f_1 \wedge j_1) \\ \wedge & \neg a_1 \vee \neg a_2 \\ \wedge & \neg b_1 \vee (\neg b_2 \wedge \neg b_3 \wedge \neg g_1) \\ \wedge & \neg b_2 \vee (\neg b_3 \wedge \neg g_1) \\ \wedge & \neg b_3 \vee (\neg c_1 \wedge \neg g_1) \\ \wedge & \neg c_1 \vee \neg g_1 \\ \wedge & \neg d_1 \vee (\neg a_2 \wedge \neg g_1) \\ \wedge & \neg e_1 \vee (\neg e_2 \wedge \neg e_3 \wedge \neg e_4) \\ \wedge & \neg e_2 \vee (\neg e_3 \wedge \neg e_4) \\ \wedge & \neg e_3 \vee \neg e_4 \end{array}$$

(b) transformation de la formule propositionnelle en NNF en utilisant les lois de De Morgan

FIGURE 1.4 – Une formule propositionnelle et la formule NNF correspondante après application des lois de De Morgan.

Parmi les connecteurs usuels, on trouve :

– $\neg$, l'opérateur de *négation*, d'arité 1 ;
– $\vee$, l'opérateur de *disjonction*, d'arité 2 ;
– $\wedge$, l'opérateur de *conjonction*, d'arité 2.

Les opérateurs de conjonction et de disjonction peuvent être étendus pour gérer une arité (finie) quelconque sans perte de généralité. On considère aussi deux opérateurs d'arité 0 : $\top$ et $\bot$, qui représentent les valeurs de vérité, respectivement *vrai* et *faux*. Nous disposons maintenant de variables et de connecteurs nous permettant de définir ce qu'est une *formule propositionnelle*.

Définition 1.15 (Formule propositionnelle) *Une formule propositionnelle est une formule définie récursivement de la façon suivante :*

– *une variable propositionnelle est une formule propositionnelle ;*
– *les opérateurs $\top$ et $\bot$, d'arité zéro, sont des formules propositionnelles ;*
– *soit $\otimes$ un opérateur d'arité n et $\{\Phi_1, \ldots, \Phi_n\}$ un ensemble de n formules propositionnelles ; $\otimes(\Phi_1, \ldots, \Phi_n)$ est une formule propositionnelle.*

La figure 1.4(a) montre un exemple de formule propositionnelle construite sur la morphologie (*i.e.* l'ensemble des connecteurs) $\{\neg, \vee, \wedge\}$. Une telle formule est en fait un circuit disposant d'une racine, dans la mesure où les formules propositionnelles sont définies de manière récursive et qu'une même formule peut être utilisée sans restriction dans la composition de multiples autres formules. Pour une sous-formule enracinée en un nœud N, on définit $Var(N)$ comme l'ensemble des variables composant cette sous-formule. Lorsqu'une formule ne contient que les opérateurs $\neg$, $\vee$, $\wedge$, $\top$, et $\bot$, et que les occurrences de l'opérateur de négation n'apparaissent qu'en tant que parent d'une variable propositionnelle, on dit que la formule est une formule NNF (*Negation Normal Form*) [DARWICHE 1999, DARWICHE 2001]. La définition d'une telle formule est donnée dans la littérature de la manière suivante.

Définition 1.16 (Formule NNF) *Soit* PS *un ensemble fini de variables propositionnelles. Une formule de* $\mathsf{NNF}_{\mathsf{PS}}$ *est un graphe dirigé, acyclique où chaque feuille est étiquetée par* vrai *($\top$ ou 1),* faux *($\bot$ ou 0), x ou $\neg x$, où $x \in$ PS ; chaque nœud interne est étiqueté par $\wedge$ ou $\vee$ et peut avoir un nombre arbitraire*

de fils. La taille d'une formule $\Phi \in \mathsf{NNF}_{\mathsf{PS}}$*, notée* $|\Phi|$*, est le nombre de nœuds de la formule* Φ*. La hauteur de* Φ *est la longueur du chemin le plus long entre la racine de la formule et une de ses feuilles.*

Par la suite, sauf ambiguïté, on simplifiera les notations en employant NNF plutôt que $\mathsf{NNF}_{\mathsf{PS}}$ lorsque cela est possible.

Les formules NNF peuvent être considérées sans perte de généralité. En effet, quand la morphologie du langage considéré est $\{\neg, \vee, \wedge\}$, toute formule propositionnelle Φ peut être transformée en une formule NNF Φ' en temps polynomial dans la taille de Φ, en utilisant les lois de De Morgan (Φ et Ψ sont deux formules propositionnelles) :

$$\neg(\Phi \vee \Psi) \rightarrow (\neg\Phi) \wedge (\neg\Psi)$$
$$\neg(\Phi \wedge \Psi) \rightarrow (\neg\Phi) \vee (\neg\Psi).$$

Il suffit d'appliquer ces règles de transformation pour tous les nœuds étiquetés par l'opérateur de négation, considérés dans l'ordre topologique (*i.e.* de la racine vers les feuilles) qui seraient parents d'un nœud $\vee$ ou $\wedge$. Un exemple de transformation en utilisant ces règles est donné aux figures 1.4(a) et 1.4(b).

À la fin de ce processus de transformation, on peut fusionner chacun des opérateurs de négation avec la variable propositionnelle qu'il admet comme fils. On se retrouve alors avec trois types de feuilles : $\top$, $\bot$, ou une structure composée d'une variable et potentiellement d'une négation (x ou $\neg x$). Cette structure est appelée *littéral*.

Définition 1.17 (Littéral) *Soit* PS *un ensemble fini de variables propositionnelles. L'ensemble des littéraux de* PS *est défini comme l'union de l'ensemble des variables de* PS *et de l'ensemble de leur négation :* $\{x, \neg x \mid x \in \mathsf{PS}\}$*. Pour une variable* $x \in \mathsf{PS}$*,* x *est le littéral positif,* $\neg x$ *est le littéral négatif, et* x *(resp.* $\neg x$*) est le littéral complémentaire de* $\neg x$ *(resp.* x*).*

Dans une formule NNF, si une variable n'apparaît que sous la forme de son littéral positif, ou bien si elle n'apparaît que sous la forme de son littéral négatif, on dit alors que ce littéral est *pur*.

Les formules NNF permettent donc de représenter toutes les formules de la logique propositionnelle sur $\{\neg, \vee, \wedge\}$, modulo une transformation polynomiale dans la taille de la formule initiale. Ce langage englobe donc un très grand nombre de formules, dont certaines pour lesquelles leur structure particulière offre des propriétés calculatoires intéressantes, et d'autres qui n'admettent pas ces propriétés mais qui permettent de représenter un problème concret de manière aisée. Parmi ces dernières familles de formules, on peut notamment identifier les formules sous forme normale conjonctive, ou formules CNF (*Conjonctive Normal Form*). Une formule CNF est définie comme une conjonction de clauses, c'est-à-dire une conjonction de disjonctions de littéraux.

Définition 1.18 (Clause, formule CNF) *Une clause est une disjonction de littéraux. Une formule* CNF *est une formule composée d'une conjonction de clauses.*

La formule $(a \vee \neg b) \wedge (b \vee \neg c)$ est un exemple de formule CNF. Parmi les formules NNF, on trouve aussi les formules sous forme normale disjonctive, ou formules DNF (*Disjonctive Normal Form*). Ces formules apportent des propriétés intéressantes, telle que le test de cohérence en temps polynomial (concept présenté dans la section suivante), mais ne permettent malheureusement pas une modélisation aisée. Une formule DNF est une disjonction de termes, c'est-à-dire une disjonction de conjonctions de littéraux.

Définition 1.19 (Terme, formule DNF) *Un* terme*, ou* cube*, est une conjonction de littéraux. Une formule* DNF *est une formule composée d'une disjonction de termes.*

Par exemple, la formule $(a \wedge \neg b) \vee (b \wedge \neg c)$ est une formule DNF. Nous avons introduit les concepts syntaxiques nécessaires pour nous permettre de présenter maintenant la sémantique de la logique propositionnelle.

$$\begin{aligned}
\omega(\top) &= 1 \\
\omega(\bot) &= 0 \\
\omega(\neg\Phi) &= 1 - \omega(\Phi) \\
\omega(\Phi_1 \vee \ldots \vee \Phi_n) &= \max(\{\omega(\Phi_i) \mid i = 1, \ldots, n\}) \\
\omega(\Phi_1 \wedge \ldots \wedge \Phi_n) &= \min(\{\omega(\Phi_i) \mid i = 1, \ldots, n\}) \\
\omega(\Phi \Rightarrow \Psi) &= \omega(\neg\Phi \vee \Psi) \\
\omega(\Phi \Leftrightarrow \Psi) &= \omega((\neg\Phi \vee \Psi) \wedge (\Phi \vee \neg\Psi))
\end{aligned}$$

TABLE 1.1 – Sémantique des opérateurs usuels de la logique propositionnelle.

1.2.2 Sémantique de la logique propositionnelle

Les formules propositionnelles sont employées dans le but de représenter des problèmes concrets. De ce fait, il est nécessaire d'avoir une sémantique permettant de faire le lien entre l'expression syntaxique du problème par une formule et la réalité du problème modélisé.

En logique propositionnelle, les formules sont évaluées à une valeur de vérité dépendante de la valeur de vérité associée à chacune de leurs variables. Ainsi, en ne considérant que les formules NNF, il suffit de faire remonter les valeurs de vérité depuis les feuilles jusqu'à la racine en associant à chacun des nœuds internes la valeur de vérité associée à chacun des opérateurs, en fonction de leurs fils. La valeur de vérité associée à la racine est alors la valeur de vérité de la formule. Une affectation de l'ensemble des variables propositionnelles est appelée une interprétation.

Définition 1.20 (Interprétation) *Une interprétation sur un ensemble de variables propositionnelles* PS *est une application de* PS *dans* $\{0, 1\}$*, l'ensemble des valeurs de vérité.*

Étant donné une interprétation ω et une formule Φ, on note $\omega(\Phi)$ la valeur de vérité donné à la formule Φ par l'interprétation ω. La sémantique des opérateurs usuels nous permet de remonter les valeurs de vérité dans les nœuds internes de la formule, et est présenté dans la table 1.1, dans laquelle nous avons ajouté les opérateurs d'implication ($\Rightarrow$) et d'équivalence ($\Leftrightarrow$), fréquemment utilisés dans le but d'alléger les notations.

Une fois la valeur de vérité de la racine de la formule connue, on connaît alors la valeur de vérité de la formule pour l'interprétation considérée. Si cette valeur est 1 (pour *vrai*), alors l'interprétation est un *modèle* de la formule.

Définition 1.21 (Modèle) *Soit ω une interprétation sur* PS *et Φ une formule propositionnelle dont les variables sont incluses dans* PS. *ω est un* modèle *de Φ si et seulement si $\omega(\Phi) = 1$. Toute interprétation qui n'est pas un modèle est un* contre-modèle *de Φ.*

On note $\Omega(\Phi) = \{\omega \mid \omega(\Phi) = 1\}$ l'ensemble des modèles d'une formule Φ. Si une formule n'admet aucun contre-modèle, cette formule est valide. À l'inverse, si aucun modèle n'existe, la formule est incohérente. Une formule admettant au moins un modèle est cohérente.

Par exemple, soit Φ la formule de la figure 1.4(b), et soient deux interprétations ω_1 et ω_2.

$$\begin{array}{llllllllllllllll}
\omega_1 = \{ & a_1 & \neg a_2 & b_1 & \neg b_2 & \neg b_3 & c_1 & d_1 & e_1 & \neg e_2 & \neg e_3 & \neg e_4 & f_1 & \neg g_1 & h_1 & i_1 & \} \\
\omega_2 = \{ & a_1 & a_2 & \neg b_1 & b_2 & \neg b_3 & \neg c_1 & d_1 & \neg e_1 & e_2 & \neg e_3 & \neg e_4 & f_1 & \neg g_1 & h_1 & \neg i_1 & \}
\end{array}$$

L'interprétation ω_1 permet de satisfaire la formule (on a $\omega_1(\Phi) = 1$). En revanche, l'interprétation ω_2 est un contre-modèle, puisque la formule implique qu'au moins un des littéraux a_1 et a_2 est falsifié, ce qui n'est pas le cas pour cette interprétation.

Définition 1.22 (Validité) *Soit Φ une formule propositionnelle. Si Φ admet au moins un modèle, alors Φ est dite* cohérente. *Si toute interprétation de Φ est modèle, alors Φ est dite* valide. *En revanche, si toute interprétation est un contre-modèle, alors Φ est dite* incohérente.

Dans l'exemple précédent, Φ est cohérente (elle admet au moins un modèle, ω_1), mais Φ n'est pas valide (elle admet au moins un contre-modèle, ω_2).

Si deux formules admettent exactement les mêmes modèles, elles représentent le même problème. On dit alors que ces formules sont équivalentes. C'est par exemple le cas pour les formules des figures 1.4(a) et 1.4(b).

Définition 1.23 (Équivalence de formules) *Soit Φ et Ψ deux formules. Φ et Ψ sont équivalentes, noté $\Phi \equiv \Psi$, si et seulement si l'ensemble des modèles de Φ est égal à l'ensemble des modèles de Ψ.*

Il existe une notion plus faible que l'équivalence, permettant toutefois de déduire l'existence de modèles d'une formule de l'existence de modèles d'une autre formule. Cette relation est connue sous le nom d'*équi-satisfiabilité*.

Définition 1.24 (Équi-satisfiabilité) *Soient Φ et Ψ deux formules. Ces formules sont dites* équi-satisfiables *(ou équivalentes pour la cohérence) si et seulement si Φ et Ψ sont soit toutes les deux cohérentes, soit toutes les deux incohérentes.*

Un exemple de formules équi-satisfiables est donné dans la section suivante.

1.2.3 Complexité de la détermination de la cohérence d'une formule

Nous allons maintenant étudier la complexité du problème de décision consistant à déterminer si un modèle existe pour une formule NNF, c'est-à-dire si cette formule est cohérente.

Tout d'abord, on met en évidence une transformation permettant de traduire n'importe quelle formule NNF en une formule CNF. Bien que cet algorithme s'exécute en temps polynomial, le fait qu'il ajoute des variables propositionnelles auxiliaires implique qu'on ne puisse parler de formules équivalentes, mais de formules équi-satisfiables. Cet algorithme est connu sous le nom de *transformation de Tseitin* [TSEITIN 1968]. Il est à noter qu'on peut aussi transformer une formule NNF quelconque en formule CNF sans ajouter de variables, mais via un algorithme exponentiel en temps et en espace, en appliquant les règles suivantes depuis les parents des feuilles jusqu'à la racine :

- si le nœud N considéré et son père P sont tous deux des nœuds $\wedge$ (resp. $\vee$), on peut les « fusionner » en attachant les feuilles de N à P et en supprimant le nœud N ; la hauteur de la sous-formule enracinée en P est alors diminuée de 1 ;
- sinon, on traduit N afin qu'il passe d'un nœud $\wedge$ (resp. $\vee$) à un nœud $\vee$ (resp. $\wedge$) en utilisant la distributivité de $\vee$ sur $\wedge$ (resp. de $\wedge$ sur $\vee$) :

$$\begin{aligned}(a_1 \wedge \ldots \wedge a_n) \vee (b_1 \wedge \ldots \wedge b_p) &\rightarrow \bigwedge_{\substack{i \in 1,\ldots,n \\ j \in 1,\ldots,p}} (a_i \vee b_j) \\ (a_1 \vee \ldots \vee a_n) \wedge (b_1 \vee \ldots \vee b_p) &\rightarrow \bigvee_{\substack{i \in 1,\ldots,n \\ j \in 1,\ldots,p}} (a_i \wedge b_j)\,;\end{aligned}$$

  de ce fait, l'opérateur de N deviendra nécessairement le même que celui de son père P, et la hauteur de P pourra être diminué de 1 à la prochaine itération.

Une fois cet algorithme exécuté, on obtient soit une CNF, soit une DNF. Si on souhaite obtenir une CNF (resp. DNF) et que notre formule est une DNF (resp. CNF), il suffit d'appliquer la règle correspondante parmi les deux présentées ci-dessus pour obtenir la formule de notre choix.

Étant donné la complexité importante de l'algorithme sans ajout de variables, on préfèrera la transformation de Tseitin. Cette transformation vise à remplacer la valeur de vérité de chacun des nœuds internes par une nouvelle variable en ajoutant des clauses au problème initial pour fixer les équivalences. En considérant les nœuds dans l'ordre topologique inverse, on se retrouve alors à remplacer des nœuds dont les fils sont uniquement des littéraux : soit des littéraux de la formule initiale, soit des littéraux provenant de variables additionnelles apparues en transformant une sous-formule lors d'une précédente transformation de Tseitin. Les règles sont les suivantes (x est la nouvelle variable).

$$\begin{array}{lcl} x \Leftrightarrow l_1 \vee \ldots \vee l_n & \rightarrow & (\neg x \vee l_1 \vee \ldots \vee l_n) \wedge (x \vee \neg l_1) \wedge \ldots \wedge (x \vee \neg l_n) \\ x \Leftrightarrow l_1 \wedge \ldots \wedge l_n & \rightarrow & (x \vee \neg l_1 \vee \ldots \vee \neg l_n) \wedge (\neg x \vee l_1) \wedge \ldots \wedge (\neg x \vee l_n) \end{array}$$

Lorsque tous les nœuds de la formule initiale ont été traités, il reste alors uniquement l'ensemble des clauses utilisées pour fixer les équivalences entre les sous-formules et les variables additionnelles. En ajoutant simplement une clause unitaire contenant le littéral positif de la variable générée à partir du remplacement de la racine de la formule initiale, on obtient la transformation de Tseitin Φ' de la formule Φ. Il est toutefois à noter que cet ajout final n'est pas nécessaire lorsque, avant cet ajout, la formule est déjà sous forme CNF.

Exemple 1.5 *Considérons une sous-formule de la figure 1.4(b) :*

$$\neg e_2 \vee (f_1 \wedge g_1 \wedge h_1 \wedge a_2 \wedge (b_2 \vee b_3)).$$

Tout d'abord, on introduit une variable x_1 équivalente à la formule $(b_2 \vee b_3)$:

$$\begin{array}{ll} & \neg e_2 \vee (f_1 \wedge g_1 \wedge h_1 \wedge a_2 \wedge x_1) \\ \wedge & \neg x_1 \vee b_2 \vee b_3 \\ \wedge & x_1 \vee \neg b_2 \\ \wedge & x_1 \vee \neg b_3. \end{array}$$

Ensuite, on introduit une variable x_2 équivalente à la formule $(f_1 \wedge g_1 \wedge h_1 \wedge a_2 \wedge x_1)$:

$$\begin{array}{ll} & \neg e_2 \vee x_2 \\ \wedge & \neg x_1 \vee b_2 \vee b_3 \\ \wedge & x_1 \vee \neg b_2 \\ \wedge & x_1 \vee \neg b_3 \\ \wedge & x_2 \vee \neg f_1 \vee \neg g_1 \vee \neg h_1 \vee \neg a_2 \vee \neg x_1 \\ \wedge & \neg x_2 \vee f_1 \\ \wedge & \neg x_2 \vee g_1 \\ \wedge & \neg x_2 \vee h_1 \\ \wedge & \neg x_2 \vee a_2 \\ \wedge & \neg x_2 \vee x_1. \end{array}$$

Dans cet exemple, il n'est pas nécessaire d'ajouter une variable supplémentaire pour la formule $\neg e_2 \vee x_2$, puisque c'est une clause. Il n'est donc pas non plus nécessaire d'introduire la clause unitaire finale.

Les formules Φ et Φ' ne sont pas seulement équi-satisfiables ; en effet, l'ensemble des modèles de Φ' correspond en fait exactement à l'ensemble des modèles de Φ une fois retirées les affectations des variables auxiliaires. De ce fait, la transformation de Tseitin est non seulement une réduction fonctionnelle polynomiale de toute formule propositionnelle (on peut l'étendre pour gérer les opérateurs de négation) vers CNF pour le problème de satisfiabilité, mais aussi pour d'autres problèmes, comme par exemple

celui de l'énumération de modèles. Cependant, réduire de tels problèmes de décision sur le langage NNF vers une de ses sous-classes, CNF, ne permet pas de les résoudre plus facilement. En effet, le problème de satisfiabilité de formule CNF, CNF-SAT (quelquefois abrégé en *problème SAT*), est un problème NP-complet. Comme nous l'avons noté précédemment, il s'agit même du premier problème à avoir été prouvé comme faisant partie de cette classe de complexité [COOK 1971].

Le langage des formules CNF est tout à fait adapté à une représentation simple des caractéristiques d'un problème. En effet, il permet de représenter un énoncé comme un ensemble de contraintes, telles que des implications logiques (si a est vrai, alors b et c doivent aussi l'être, ce qui permet de coder des dépendances entre unités installables), des exclusions mutuelles (si a est vrai, alors ni b ni c ne doivent l'être, ce qui permet de coder des conflits entre unités installables), de forcer des faits (a doit être vrai ; permet de coder des requêtes). De ce fait, bien qu'il existe des sous-ensembles du langage NNF qui autorisent le test de satisfiabilité en temps polynomial dans la taille de la formule (comme les formules DNNF [DARWICHE & MARQUIS 2002], les formule Horn [MINOUX 1988, DALAL 1992, RAUZY 1995] ou Horn-renommables, les formules 2-CNF [COOK 1971]), le fait que l'encodage d'un problème dans ces langages soit complexe fait que s'intéresser au problème CNF-SAT reste intéressant en pratique.

1.3 Conclusion du chapitre

Dans ce chapitre, nous avons présenté le concept de problème de décision. Nous avons tout d'abord étudié leur complexité, après avoir noté que certains d'entre eux ne sont pas calculables. Nous avons ensuite présenté un problème fondamental pour la théorie de la décision dès lors que l'on considère des problèmes pour lesquels il ne semble pas exister d'algorithmes de résolution polynomial, le problème de satisfiabilité de formule CNF. Nous avons vu que la complexité des problèmes de décision était supposée s'étaler sur une échelle, appelée la hiérarchie polynomiale. Nous avons présenté un certain nombre de procédés permettant de classer un problème de décision sur cette échelle, tels que les machines de Turing, ou les réductions fonctionnelles polynomiales.

Nous avons ensuite considéré en détail le problème SAT, problème fondamental pour la hiérarchie polynomiale. Ce problème est connu comme étant le problème de référence pour les problèmes de la classe de complexité NP. Nous avons vu que le langage CNF permettait d'exprimer de manière simple bon nombre de problèmes combinatoires, et avons ainsi justifié toute l'importance portée à la résolution efficace de tels problèmes.

Dans le chapitre suivant, nous allons considérer non plus des problèmes de décision, mais des problèmes d'optimisation. Ces problèmes sont plus complexes, dans la mesure où ne cherche plus une solution quelconque, mais une solution qui est considérée meilleure que les autres selon un unique critère, ou bien même un ensemble fini de critères.

Chapitre 2

Problèmes d'optimisation

Sommaire

2.1 Optimisation monocritère **26**
- 2.1.1 Notion de préférence 26
- 2.1.2 Fonctions de coût et d'utilité 28

2.2 Optimisation multicritère **30**
- 2.2.1 Généralités 30
- 2.2.2 « Agréger puis comparer » *vs.* « comparer puis agréger » 31
- 2.2.3 Propriétés des fonctions d'agrégation 32
- 2.2.4 Prise en compte du potentiel 36
- 2.2.5 Fonctions d'agrégations de critères usuelles 36

2.3 Conclusion du chapitre **43**

Dans le chapitre précédent, nous avons présenté les problèmes de décision, c'est-à-dire les problèmes pour lesquels il s'agit de déterminer si une solution existe ou non pour un problème donné. Dans ce chapitre, nous nous intéressons aux problèmes d'optimisation, où il s'agit de déterminer une des meilleures solutions d'un problème selon un ou plusieurs critères donnés.

Nous présentons tout d'abord la notion de relation de préférence entre solutions (ou *alternatives*). Ce type de relation nous permet de comparer les alternatives deux à deux selon un critère donné ; cette notion est la base de l'optimisation monocritère, puisque déterminer qu'une solution est au moins aussi préférée que n'importe quelle autre selon un unique critère indique qu'elle est optimale pour ce critère.

Dans la suite, nous nous intéressons à la problématique de l'optimisation lorsque plusieurs critères sont considérés. Passer du cadre monocritère au cadre multicritère apporte de nombreux problèmes ; nous traiterons notamment dans ce chapitre de ceux liés à la commensurabilité entre critères (le fait de pouvoir comparer ou agréger des critères, ce qui suppose normalement qu'ils représentent des quantités comparables), ainsi que de ceux liés à la notion de compromis (le fait de déterminer que des alternatives sont « globalement » bonnes quand elles ne satisfont pas de manière égale tous les critères).

Nous traiterons finalement de différentes méthodes de l'état de l'art employées pour l'agrégation de scores (coûts ou utilités) ; nous donnerons dans un premier temps les propriétés attendues de ces agrégateurs (« bonnes » propriétés), puis nous présenterons à proprement parler les agrégateurs que nous considérons dans nos travaux.

2.1 Optimisation monocritère

Un problème d'optimisation dans le cadre de l'optimisation monocritère peut être formulé par une question : « mon problème admet-il oui ou non une solution, et si oui retourner *une des meilleures* selon mon critère de choix ». Les problèmes d'optimisation sont au moins aussi difficiles que les problèmes de décision, puisque répondre à un problème d'optimisation permet en même temps de répondre au problème de décision associé ; c'est par exemple le cas pour 2-SAT, qui est un problème de décision dans P, alors que le problème MAX-2-SAT est lui NP-difficile.

2.1.1 Notion de préférence

Afin de pouvoir définir ce qu'est un critère de choix, nous présentons tout d'abord le concept de *relation de préférence*. Une relation de préférence $\succeq$ est une relation binaire sur l'ensemble Ω des alternatives d'un problème (dans notre cas, des modèles d'une formule propositionnelle), c'est-à-dire un sous-ensemble de $\Omega \times \Omega$. Ainsi, $\omega_1 \succeq \omega_2$ signifie que le décideur préfère (de manière non stricte, c'est-à-dire que l'indifférence est possible) le modèle ω_1 au modèle ω_2. Dans le but de simplifier les notations et les définitions, nous allons en fait considérer trois relations pour l'expression des préférences pour un critère :

- la relation de préférence $\succeq$;
- la relation de préférence stricte $\succ$, correspondant à $\succeq$ dont on a retiré les couples composés de deux alternatives pour lesquelles le décideur est indifférent : $\omega_1 \succ \omega_2 \Leftrightarrow (\omega_1 \succeq \omega_2) \wedge (\omega_2 \not\succeq \omega_1)$;
- la relation d'indifférence $\sim$, correspondant justement aux alternatives considérées comme équivalentes en terme d'intérêt du décideur : $\omega_1 \sim \omega_2 \Leftrightarrow (\omega_1 \succeq \omega_2) \wedge (\omega_2 \succeq \omega_1)$.

Dans la suite, on considérera que la relation $\succeq$ est *totale*, ce qui signifie que pour tout couple $(\omega_1, \omega_2) \in \Omega^2$, on a $\omega_1 \succeq \omega_2$, ou $\omega_2 \succeq \omega_1$. Notons toutefois que cette hypothèse n'implique pas que les relations $\succ$ et $\sim$ soient aussi totales, et qu'il est même impossible que $\sim$ et $\succ$ soient toutes les deux totales : si $\omega_1 \sim \omega_2$, alors on a ni $\omega_1 \succ \omega_2$, ni $\omega_2 \succ \omega_1$; d'un autre côté, si $\omega_1 \succ \omega_2$, alors on a ni $\omega_1 \sim \omega_2$, ni $\omega_2 \sim \omega_1$.

Les relations de préférence (strictes ou non) et les relations d'indifférence peuvent admettre un certain nombre de propriétés définies pour les relations binaires, permettant de les classifier :

– *réflexivité* : $\forall \omega \in \Omega, \omega \succeq \omega$;
– *irréflexivité* : $\forall \omega \in \Omega, \omega \not\succeq \omega$;
– *transitivité* : $\forall \omega_1, \omega_2, \omega_3 \in \Omega, ((\omega_1 \succeq \omega_2) \wedge (\omega_2 \succeq \omega_3)) \Rightarrow \omega_1 \succeq \omega_3$;
– *symétrie* : $\forall \omega_1, \omega_2 \in \Omega, \omega_1 \succeq \omega_2 \Rightarrow \omega_2 \succeq \omega_1$;
– *asymétrie* : $\forall \omega_1, \omega_2 \in \Omega, \omega_1 \succeq \omega_2 \Rightarrow \omega_2 \not\succeq \omega_1$.

Un certain nombre de types de relations binaires ont été définies en fonction de ces propriétés :

– un *préordre* est une relation réflexive et transitive ;
– un *ordre strict* est une relation transitive et irréflexive ;
– une *relation d'équivalence* est une relation réflexive, symétrique et transitive.

La relation binaire $\succeq$ définie au-dessus satisfait les propriétés de réflexivité, puisqu'elle n'exclut pas l'indifférence du décideur, et de transitivité. Cette relation est donc un préordre, c'est-à-dire une relation qui permet de représenter un ordre sur l'ensemble des éléments où les égalités sont possibles. Contrairement à la relation $\succeq$, la relation $\succ$, qui est elle aussi transitive, est irréflexive ; une telle relation permet de coder des préférences mais pas l'indifférence : c'est une relation d'ordre strict. Enfin, dans les cas que nous étudierons, la relation $\sim$ est réflexive, symétrique, et transitive : une relation respectant ces trois propriétés est une relation d'équivalence. Lorsque les relations $\succeq$, $\succ$ et $\sim$ sont respectivement un préordre, un ordre, et une relation d'équivalence, on dit alors qu'on se situe dans le cadre des *hypothèses fortes de rationalité du décideur*.

Définition 2.1 (Hypothèses fortes de rationalité du décideur [BARBA-ROMERO & POMEROL 1997]) *Des préférences respectent les hypothèses fortes de rationalité du décideur si et seulement si :*

– *l'intersection entre $\sim$ et $\succ$ est vide ;*
– *$\sim$ est réflexive et transitive ;*
– *$\succ$ est asymétrique ;*
– *$\succeq$ est transitive.*

Bien que ces hypothèses semblent raisonnables, elles ne sont pas toujours satisfaites. On peut citer par exemple le célèbre problème des tasses de café.

Exemple 2.1 (Problème des tasses à café [LUCE 1956]) *Trouvez un décideur qui préfère une tasse de café avec un morceau de sucre à une tasse de café avec cinq morceaux de sucre. Maintenant préparez 401 tasses de café avec $(1+\frac{i}{100})x$ grammes de sucre pour $i = 0, \ldots, 400$, où x est le poids d'un morceau de sucre. Il est évident qu'il sera indiffèrent entre la tasse i et la tasse $i+1$, pour tout $i \neq 400$, mais il ne sera probablement pas indifférent entre les tasses $i = 0$ et $i = 400$.*

Dans le problème des tasses de café, la relation $\sim$ n'est donc pas transitive. Dans ce cas, on se situe dans le cadre moins restrictif des *hypothèses faibles de rationalité du décideur* [BARBA-ROMERO & POMEROL 1997].

Définition 2.2 (Hypothèses faibles de rationalité du décideur) *Des préférences respectent les hypothèses faibles de rationalité du décideur si et seulement si :*

– *l'intersection entre $\sim$ et $\succ$ est vide ;*
– *$\succ$ est asymétrique ;*
– *$\succeq$ est transitive.*

Dans ce cas, la relation de préférence permet de modéliser un *semi-ordre*, aussi appelé *quasi-ordre* [PIRLOT & VINCKE 1997]. Cette structure permet d'introduire un *seuil d'indifférence*, c'est-à-dire de considérer comme équivalentes des solutions proches, sans pour autant devoir considérer deux solutions comme équivalentes en procédant « de proche en proche ».

Une relation de préférence nous permet de sélectionner, parmi l'ensemble des solutions d'un problème, une solution qui est la meilleure pour le critère associé ; c'est-à-dire une solution telle qu'aucune autre ne lui est strictement préférée selon la relation de préférence considérée. On parle alors de *solution optimale*.

Définition 2.3 (Solution optimale) *Soit Ω l'ensemble des solutions d'un problème, et $\succeq$ une relation de préférence associée à un critère. $\omega^* \in \Omega$ est une solution optimale du problème pour le critère associé à $\succeq$ si et seulement si :*

$$\forall \omega \in \Omega \setminus \{\omega^*\},\ \omega \not\succ \omega^*.$$

Sous les hypothèses fortes de rationalité du décideur, il nous est possible d'exprimer les préférences de manière numérique, en associant un score à chacune des alternatives, via le recours aux fonctions d'utilité.

2.1.2 Fonctions de coût et d'utilité

Sous les hypothèses fortes de rationalité du décideur, la relation $\succeq$ est un préordre total. Cette condition est suffisante [FISHBURN 1970] pour représenter la relation de préférence considérée à l'aide d'une *fonction d'utilité* ou d'une *fonction de coût*.

Définition 2.4 (Fonction d'utilité, fonction de coût) *Une fonction d'utilité $u : \Omega \mapsto \mathbb{R}$ modélise une relation de préférence $\succeq$ si et seulement si :*

$$\forall \omega_1, \omega_2 \in \Omega,\ u(\omega_1) \geq u(\omega_2) \Leftrightarrow \omega_1 \succeq \omega_2.$$

Une fonction de coût $c : \Omega \mapsto \mathbb{R}$ modélise une relation de préférence $\succeq$ si et seulement si :

$$\forall \omega_1, \omega_2 \in \Omega,\ c(\omega_1) \leq c(\omega_2) \Leftrightarrow \omega_1 \succeq \omega_2.$$

On remarque que si l'on dispose d'une fonction d'utilité (resp. de coût), alors on dispose aussi d'une fonction de coût (resp. d'utilité) : il suffit pour cela de définir la fonction $c(\omega) = -u(\omega)$ (resp. $u(\omega) = -c(\omega)$). De ce fait, dans la suite, on parlera indifféremment de fonction d'utilité ou de fonction de coût.

Lorsque la structure à modéliser par une fonction objectif (on désignera par *fonction objectif* une fonction de coût que l'on souhaite minimiser ou bien une fonction d'utilité que l'on souhaite maximiser) est un quasi-ordre total $\succeq$, on peut se ramener à un préordre total *via* la trace $\succeq^{\pm}$ de $\succeq$ [BOUYSSOU *et al.* 2013], définie par :

$$\omega_1 \succeq^{\pm} \omega_2 \Leftrightarrow \forall \omega_3 \in \Omega, (\omega_2 \succeq \omega_3 \Rightarrow \omega_1 \succeq \omega_3) \wedge (\omega_3 \succeq \omega_1 \Rightarrow \omega_3 \succeq \omega_2).$$

Une façon classique de représenter des fonctions de coût ou d'utilité dans le cadre de la logique propositionnelle consiste à employer des *fonctions pseudo-booléennes*.

Définition 2.5 (Fonction pseudo-booléenne [BOROS & HAMMER 2002]) *Une fonction pseudo-booléenne est une fonction de l'ensemble des interprétations sur un ensemble de variables booléennes dans l'ensemble des nombres réels :*

$$f : \mathbb{B}^n \mapsto \mathbb{R}.$$

Une telle fonction peut être représentée de manière concise en utilisant une base propositionnelle pondérée, munie d'un opérateur d'agrégation.

Définition 2.6 (Base propositionnelle pondérée) *Une base (propositionnelle) pondérée est un ensemble de couples* $\{(\phi_i, c_i)\}$ *où les* ϕ_i *sont des formules propositionnelles et les* c_i *sont des nombres réels. On définit la fonction Val :* $(\phi_i, c_i), \omega \mapsto \mathbb{R}$ *qui retourne* c_i *si* $\phi_i \wedge \omega$ *est cohérent, et* 0 *sinon.*

Interprétée de cette manière, une base pondérée représente un ensemble de poids : si une formule ϕ_i de la base est cohérente avec une interprétation ω, alors le coût de cette interprétation ω pour ϕ_i est c_i ; dans le cas contraire, le coût est nul. Pour chaque modèle ω d'une formule propositionnelle, on peut alors obtenir un vecteur de coûts $(\text{Val}((\phi_1, c_1), \omega), \ldots, \text{Val}((\phi_n, c_n), \omega))$.

Définition 2.7 (Vecteur de coûts) *Soient une formule* Φ *et une base pondérée* $\phi = \{(\phi_i, w_i) \mid i \in 1, \ldots, n\}$. *Le vecteur de coûts associé à un modèle ω de* Φ *est défini par :*

$$(\mathit{Val}((\phi_1, c_1), \omega), \ldots, \mathit{Val}((\phi_n, c_n), \omega)).$$

Une fois un vecteur de coûts obtenu, il ne reste qu'à agréger les coûts issus de ce vecteur en utilisant un opérateur d'agrégation. Le plus souvent, c'est la somme qui est utilisée à cette fin ; dans ce cas, on peut envisager d'employer, par souci de concision, le formalisme suivant pour représenter notre fonction objectif :

$$f_\phi(\omega) = c_1\phi_1 + \ldots + c_n\phi_n \; .$$

Par exemple, imaginons que nous recherchions une solution à la formule de la figure 1.4(a) (page 18) représentant l'exemple de la figure 1.1 de la page 13 qui minimise le nombre de paquets à retirer (on dit qu'un paquet est retiré s'il est installé dans au moins une version dans l'état initial, mais qu'aucune de ses versions n'est installée dans l'état final). Dans ce but, on peut créer une fonction pseudo-booléenne de coût (que l'on souhaite donc minimiser) dont la valeur est égale au nombre de paquets retirés : $c(\omega) = 1.(\neg a_1 \neg a_2) + 1.(\neg b_1 \neg b_2 \neg b_3) + 1.(\neg c_1) + 1.(\neg d_1)$. Parmi les modèles de la formule, on retrouve les deux interprétations suivantes :

$$\begin{array}{l} \omega_1 = \{\ a_1 \ \neg a_2 \ b_1 \ \neg b_2 \ \neg b_3 \ c_1 \ d_1 \ e_1 \ \neg e_2 \ \neg e_3 \ \neg e_4 \ f_1 \ \neg g_1 \ h_1 \ i_1 \ \} \\ \omega_2 = \{\ \neg a_1 \ a_2 \ \neg b_1 \ b_2 \ \neg b_3 \ \neg c_1 \ d_1 \ \neg e_1 \ e_2 \ \neg e_3 \ \neg e_4 \ f_1 \ \neg g_1 \ h_1 \ \neg i_1 \ \} \; . \end{array}$$

On remarque que $c(\omega_1) = 0$, alors que $c(\omega_2) = 1$. De ce fait, ω_2 ne peut être une solution optimale pour notre critère, puisque ω_1 a un coût inférieur. En revanche, ω_1 est une solution optimale pour le critère, puisque aucune autre solution ne peut retirer moins de zéro paquets.

En fonction de la base pondérée utilisée, on peut distinguer un certain nombre de *familles de représentation* de la fonction objectif :

- la famille $\mathcal{L}$ des *représentations linéaires* des bases pondérées dont les ϕ_i sont des littéraux ;
- la famille $\mathcal{Q}$ des *représentations quadratiques* des bases pondérées dont les ϕ_i sont des cubes de taille 2 ;
- la famille $\mathcal{P}$ des *représentations polynomiales* des bases pondérées dont les ϕ_i sont des cubes de longueur quelconque ;
- et finalement, la famille $\mathcal{G}$ des *représentations pseudo-booléennes générales* des bases pondérées dont les ϕ_i sont des formules propositionnelles.

Il est à noter que toute fonction pseudo-booléenne admet une représentation polynomiale [Boros & Hammer 2002]. En effet, on peut associer à toute fonction pseudo-booléenne une base pondérée composée d'un couple pour chaque modèle de la formule correspondante et de lui associer le coût global attendu. Ceci n'est cependant pas réalisable en pratique dans la majorité des cas du fait du nombre

exponentiel de modèles qu'une formule peut admettre. Il est ainsi d'usage d'utiliser une représentation plus concise, bien que potentiellement plus difficile à exploiter.

L'optimisation monocritère avec une fonction objectif basée sur une base pondérée peut aussi être interprétée comme un processus d'optimisation multicritère, où les critères sont binaires. En effet, en considérant chaque ϕ_i comme un critère pouvant être soit totalement satisfait, soit totalement violé (on exclut toute valeur intermédiaire), le vecteur de coûts correspond en fait à l'ensemble des coûts engendrés par la solution considérée, pour l'ensemble des critères définis. Dans ce cas, l'agrégation effectuée à partir de ce vecteur cherche à dégager une valeur de coût globale en fonction de chacun des coûts individuels. Bien que ce formalisme exclue tout critère non binaire, et bien que comme nous le verrons par la suite, la somme n'est que rarement indiquée comme opérateur d'agrégation dans le cadre de la synthèse de critères, il s'agit ici d'un exemple d'optimisation multicritère.

2.2 Optimisation multicritère

Dans la section précédente, nous avons présenté les concepts permettant de déterminer une solution optimale selon un unique critère. Malheureusement, les préférences d'un décideur peuvent être complexes à représenter à l'aide d'un seul critère de choix. Dans certains cas, la solution attendue doit respecter au mieux un ensemble d'attentes formant autant de critères de choix, pouvant être conjointement contradictoire dans le sens où une alternative qui serait jugée plutôt bonne pour un critère ne sera pas nécessairement jugée aussi satisfaisante pour les autres critères. Il s'agit alors de dégager une solution qui satisfait au mieux, de manière globale, les attentes du décideur.

2.2.1 Généralités

Dans le cas d'une fonction objectif associant l'ensemble des alternatives (qui sont des modèles dans le cadre de la logique propositionnelle) représentées de manière explicite à une valeur, le problème d'optimiser selon plusieurs critères ne pose pas de difficulté particulière : on code l'ordre (au sens non strict) attendu directement dans la fonction objectif en affectant aux modèles des valeurs cohérentes avec cet ordre. En revanche, lorsqu'on souhaite définir une fonction objectif tenant compte de plusieurs critères de manière plus concise, il peut se révéler judicieux de créer une fonction pour chaque critère, et de raisonner ensuite à l'aide de chacune d'elles. Cependant, un certain nombre de problèmes se posent alors.

Tout d'abord, l'ensemble des critères considérés est généralement contradictoire, au sens où une alternatives considérée comme très bonne pour un des critères peut être jugée mauvaise pour d'autres critères considérés par le décideur ; dans le cas contraire, où tous les critères iraient dans le même sens, il serait alors simplement nécessaire qu'une solution soit optimale pour l'un d'eux pour la savoir globalement satisfaisante, voir même optimale pour l'ensemble des critères. Un cas classique d'ensemble de critères contradictoire est le rapport qualité/prix : généralement, ce qui n'est pas cher est de mauvaise qualité. En revanche, plus un produit est de bonne qualité, plus il risque d'être cher comparé à ceux qui sont jugés de moins bonne qualité. Une bonne approche dans ce cas est de tenter de rechercher une solution de compromis entre la qualité et le prix.

Un autre problème pouvant se poser concerne la commensurabilité des critères. En effet, les relations de préférence associées à chacun des critères ne peuvent pas toujours être représentées à l'aide de fonctions de coût ou d'utilité dont les échelles où le nombre de graduations sont les mêmes (ce qui peut poser le problème d'agréger un critère binaire avec un critère pouvant prendre une infinité de valeurs). Un autre problème pouvant se poser est le cas où il n'y a tout simplement aucune cohérence à comparer ces différents coûts, comme par exemple dans le cas où ces valeurs concernent des unités différentes.

Finalement, se pose la question, en présence de deux vecteurs de coûts calculés à partir de deux alternatives différentes que l'on souhaite comparer, de la méthode à employer pour les différencier. Il existe deux façons de procéder : pour la première, on calcule une valeur de coût global pour chacun des vecteurs, puis on compare les deux valeurs obtenues ; pour la deuxième, on calcule un vecteur dans lequel chaque valeur correspond à la comparaison des deux alternatives pour un critère, puis on agrège ces valeur afin de déterminer quelle solution choisir.

2.2.2 « Agréger puis comparer » *vs.* « comparer puis agréger »

On considère disposer d'un ensemble de n critères $\{1, \ldots, n\}$ et des n fonctions de coût $c_1, \ldots, c_n$ qui leur sont associées. On souhaite définir une relation de préférence $\succeq_c$ qui synthétise l'ensemble des relations de préférence induites par $c_1, \ldots, c_n$. Cette relation doit à la fois tenir compte de la différence entre deux alternatives pour un critère donné, mais doit aussi être capable d'agréger les n critères. De ce fait, deux méthodes se dégagent pour comparer deux alternatives :

– donner à chacune d'elle une valeur globale (agréger) et comparer ces valeurs globales (comparer) : c'est l'approche dite AC, pour « agréger et comparer » ;
– créer un vecteur de n éléments exprimant les différences entre les deux alternatives pour chacun des n critères (comparer), et agréger les valeurs de ce vecteur (agréger) : c'est l'approche CA, pour « comparer et agréger ».

Approche AC

Tout d'abord, on considère l'approche « agréger puis comparer » (AC), qui associe à chaque vecteur de coûts un coût global. Une illustration de cette approche est donnée à la figure 2.1.

$$c(\omega) = \oplus(c_1(\omega), \ldots, c_n(\omega)) \ .$$

Cette approche semble très naturelle pour certains domaines dans lesquels on cherche justement à composer un coût (ou une utilité) global depuis un certain nombre de coûts partiels. On peut citer comme exemples les problèmes pour lesquels on cherche à optimiser la rentabilité économique de différentes alternatives impliquant un certain nombre de coûts différents, les problèmes pour lesquels on chercherait à déterminer la satisfaction globale d'un client en fonction de sa satisfaction pour différents éléments d'une prestation, ...

Un avantage important à procéder de cette manière en terme de représentation du problème d'optimisation réside dans le fait que chaque alternative se voit attribuer une valeur en fonction des scores qu'elle fait prendre aux différentes fonctions objectifs. De ce fait, cela revient en quelque sorte à générer un metacritère $c(\omega) = \oplus(c_1(\omega), \ldots, c_n(\omega)$, et à optimiser ensuite selon cet unique critère. Il est cependant à noter que la nouvelle fonction objectif ne fait pas nécessairement partie de la même famille de représentation, ce qui peut complexifier le processus d'optimisation.

Cependant, agréger des critères de natures différentes n'est pas toujours aisé si ces critères ne sont pas commensurables. En effet, même en partant de l'hypothèse réductrice que tout critère est exprimé en tant qu'une fonction de coût ou d'utilité, rien n'indique qu'ils soient systématiquement commensurables. Il se peut en effet que ces critères représentent des unités différentes, ou bien qu'ils aient des croissances différentes : certains objectifs peuvent par exemple être représentés par des fonctions strictement convexes d'un nombre de littéraux positifs, quand d'autres peuvent être des fonctions strictement concaves de ce même ensemble ; de ce fait, ajouter un littéral positif à un modèle n'aura pas le même impact sur les coûts pour les deux critères considérés, si ce littéral fait partie des paramètres des deux fonctions.

Afin d'illustrer ce problème de commensurabilité, considérons le problème de choisir un objet avec le meilleur rapport qualité/prix, en supposant que la qualité est donnée par une note allant de zéro à dix

points, quand le prix est lui exprimé en euros. Comment dégager une valeur d'agrégation de ces valeurs ? Quel prix vaut un point de qualité ? Passer d'une note de qualité de deux à trois points coûte-t-il le même prix que de passer de huit à neuf points ? Il n'existe malheureusement pas de réponse simple à cette question. Bien que des travaux soient menés dans le but de réduire l'impact du problème de commensurabilité entre critères (on peut par exemple citer [BENFERHAT *et al.* 2014]), nous ne traiterons pas dans cet ouvrage des méthodes dont le but est de tenter de déterminer des fonctions objectif commensurables ; le lecteur intéressé pourra par exemple se diriger vers [BOUYSSOU *et al.* 2006] pour obtenir des explications à ce propos.

Une autre approche, qui n'agrège pas directement les utilités des fonctions monocritère, a été proposée pour tenter de contourner le problème de la commensurabilité : il s'agit de l'approche CA.

Approche CA

L'approche « comparer puis agréger » (CA) tente de palier ce problème de commensurabilité de critères en prévenant la comparaison directe de critères différents. Cette approche est illustrée à la figure 2.1.

Étant donné deux vecteur de coûts $(c_{1,1}, \ldots, c_{1,n})$ et $(c_{2,1}, \ldots, c_{2,n})$ correspondant à deux alternatives différentes, on compare les deux alternatives pour chacun des critères pris séparément. On obtient alors un vecteur de valeurs de comparaison intermédiaires, et c'est sur ce vecteur là que l'on procède à l'agrégation, afin de déterminer si une des alternatives est préférée à l'autre, et si oui laquelle.

$$(\oplus_1(c_{1,1}, c_{2,1}), \ldots, \oplus_n(c_{1,n}, c_{2,n}))$$

En utilisant cette approche, il n'y a ainsi plus de comparaison directe de scores obtenus sur des critères différents, contrairement à l'approche AC : les scores d'un critère sont comparés avec les scores d'autres alternatives, mais sur le même critère. Cette phase de comparaison n'implique donc aucun problème de commensurabilité, dans la mesure où ces scores sont issus de la même fonction objectif, et donc considérés comme comparables.

En revanche, l'agrégation des indices de préférence du vecteur suppose d'être capable de les comparer. De ce fait, il faut les considérer commensurables, ce qui encore une fois n'est pas nécessairement vrai. En effet, reprenons l'exemple du rapport qualité/prix illustrant l'approche AC, et supposons que nous comparions deux alternatives dont les notes de qualité sur dix points et les coûts monétaires sont donnés par les couples $(5, 150)$ et $(7, 200)$. On pourrait, par exemple, utiliser comme fonctions de comparaison des critères les différences de points et de coûts monétaires, ce qui nous donnerait $(-2, -50)$ comme vecteur d'indices partiels de comparaison. On voit dans ce cas que les valeurs que l'on doit agréger représentent encore une fois une quantité de points, et une quantité d'euros : le problème de l'agrégation de quantités non commensurables se pose toujours.

L'approche CA est en fait en quelque sorte une reformulation du problème de l'agrégation de préférences individuelles étudié en choix social, problème pour lequel il a été démontré de nombreux résultats négatifs qui se répercutent sur l'optimisation multicritère dans le cadre de l'approche CA [ARROW 2012, ARROW *et al.* 1986, PERNY 1992] qui ne seront pas détaillés ici.

2.2.3 Propriétés des fonctions d'agrégation

Les approches « agréger puis comparer » et « comparer puis agréger » définies dans la section précédente reposent toutes les deux sur une phase d'agrégation de valeurs : agrégation de coûts pour l'approche AC, agrégation d'indices de comparaison partiels en ce qui concerne l'approche CA. De nombreuses fonctions ont été étudiées dans la littérature afin de réaliser ces agrégations de la manière la plus juste possible.

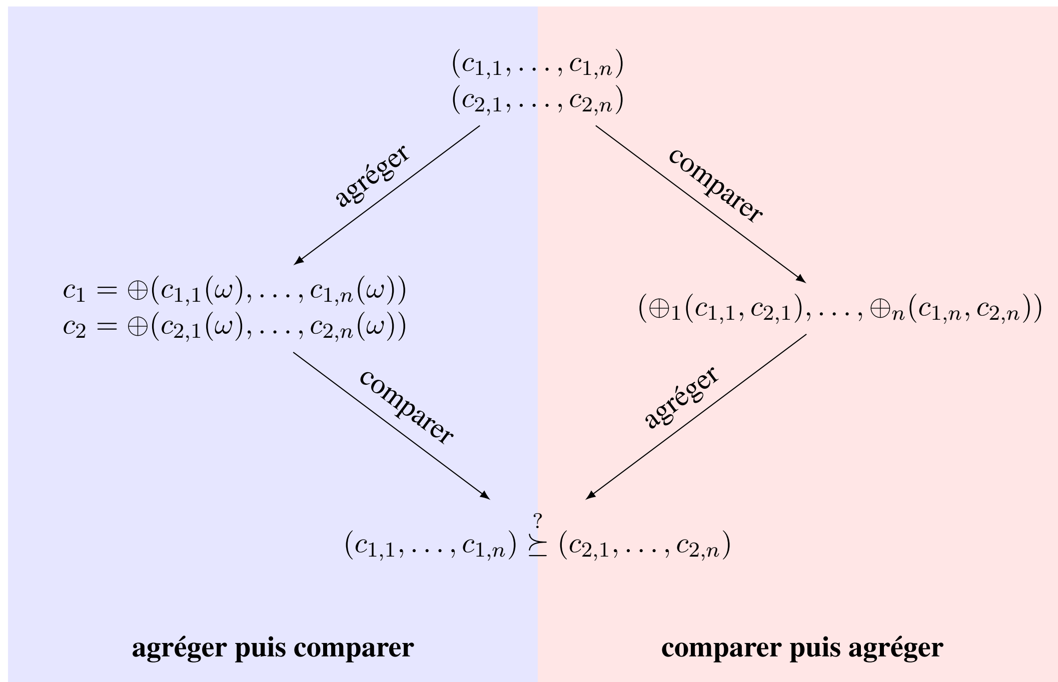


FIGURE 2.1 – Illustration des méthodes « agréger puis comparer » (AC) et « comparer puis agréger » (CA).

Ces fonctions satisfont ou non selon les cas un certain nombre de propriétés mathématiques admises comme souhaitables pour que l'agrégation soit considérée comme juste. Dans cette section, nous donnons les définitions de ces propriétés et expliquons en quoi elles sont souhaitables.

Par convention, nous supposons ici que les valeurs à agréger correspondent à des coûts (coûts impliqués par une fonction objectif pour l'approche AC, coûts de choisir une alternative plutôt qu'une autre en ce qui concerne l'approche CA) pouvant prendre l'ensemble des valeurs réelles de l'intervalle $[0, 1]$. Notons toutefois que ceci s'effectue sans perte de généralité puisque toute fonction prenant des valeurs dans un intervalle fini (ce qui est le cas dans le cadre d'une fonction pseudo-booléenne, puisque le nombre fini de valeurs qu'elles peuvent admettre fait qu'on peut choisir une valeur tendant vers l'infini à la place de l'infini lui-même) peuvent être adaptées via une transformation linéaire pour admettre l'ensemble de ses valeurs dans l'intervalle $[0, 1]$.

Dans la suite, nous utilisons pour simplifier le formalisme associé à l'approche « agréger puis comparer », c'est-à-dire que nous considérons agréger des critères plutôt que des indices de préférence partiels, bien que la plupart des sujets abordés ici concernent aussi l'approche CA.

Unanimité pour les valeurs extrêmes, idempotence

Une fonction d'agrégation satisfait la propriété *d'unanimité pour les valeurs extrêmes* si et seulement si :

$$\oplus(0, \ldots, 0) = 0, \; \oplus(1, \ldots, 1) = 1.$$

L'agrégation doit permettre de dégager une valeur synthétisant les valeurs à agréger. De ce fait, une alternative jugée tout à fait mauvaise sur tous les critères doit être aussi jugée globalement tout à fait mauvaise. Le même raisonnement s'applique pour une alternative qui serait jugée tout à fait bonne pour chacun des critères.

Dans le cas où les critères sont parfaitement commensurables, ce raisonnement devrait aussi être appliqué à chacune des graduations possibles dans le jugement des alternatives : si cela est le cas, la fonction d'agrégation satisfait la propriété *d'idempotence* :

$$\oplus(a, \ldots, a) = a.$$

Il est bien entendu évident que la propriété d'idempotence implique la propriété d'unanimité pour les valeurs extrêmes : il suffit de prendre $a = 0$ et $a = 1$.

Monotonie, Pareto-dominance

Dans le cadre monocritère, une solution est préférée à une autre si et seulement si elle est préférée selon l'unique critère considéré. Dans le cadre multicritère, lorsque de multiples alternatives génèrent des vecteurs de coûts identiques excepté pour un unique critère, ce critère devient de fait le seul sur lequel seront jugées ces alternatives pour les départager. De ce fait, dans une telle situation, on s'attend naturellement à ce que la solution choisie soit une de celles qui sont les meilleures pour ce critère. On dit dans ce cas que le vecteur de coûts de la solution à choisir *domine au sens de Pareto* (ou *Pareto-domine*) le vecteur de coûts de l'autre solution. Cette relation de dominance s'étend aux cas où un vecteur de coûts domine un autre pour tous les critères pour lesquels les deux solutions ne sont pas jugées équivalentes. Si un vecteur de coûts domine au sens de Pareto tous les autres, on dit qu'il est *optimal au sens de Pareto* (ou *Pareto-optimal*). On note $\succeq_P$ la relation associée à la dominance de Pareto, et on nomme *front de Pareto* l'ensemble des solutions Pareto-optimales. Une illustration du front de Pareto est donné à la figure 2.2.

L'ensemble des fonctions d'agrégation respectant les comparaisons impliquées par la dominance de Pareto comporte notamment les fonctions respectant la propriété de *stricte monotonie* vis à vis de chacun des critères [KOSTREVA *et al.* 2004] :

$$a'_i < a_i \Rightarrow \oplus(a_1, \ldots, a'_i, \ldots, a_n) < \oplus(a_1, \ldots, a_i, \ldots, a_n).$$

On définit aussi la propriété de *monotonie* (au sens large) vis à vis de chacun des critères :

$$a'_i < a_i \Rightarrow \oplus(a_1, \ldots, a'_i, \ldots, a_n) \leq \oplus(a_1, \ldots, a_i, \ldots, a_n).$$

Comme nous le verrons dans la suite, il existe des fonctions qui satisfont la monotonie, mais qui ne satisfont pas la monotonie stricte. Cela est dû au fait que ces fonctions ne tiennent pas toujours compte de tous les coûts du vecteur pour effectuer leurs agrégations.

Neutralité

Il est souhaitable qu'une fonction d'agrégation ait le même comportement quel que soit l'ordre dans lequel on lui fournit les coûts à agréger. En effet, comme noté plus haut dans cette section, le but de l'agrégation doit être de dégager une valeur synthétisant les valeurs à agréger. Or, la synthèse de coûts ne doit normalement pas être dépendante de l'ordre dans lequel ces coûts sont considérés.

Une fonction d'agrégation pour laquelle cet ordre n'influe pas satisfait la propriété de *neutralité*, définie formellement par :

$$\oplus(a_1, \ldots, a_n) = \oplus(a_{\sigma(1)}, \ldots, a_{\sigma(n)}) \text{ pour tout permutation } \sigma \text{ de } \{1, \ldots, n\}.$$

Parmi les fonctions neutres, on trouve les fonctions *convexes au sens de Schur* ; ce sont les fonctions qui en plus de satisfaire la neutralité, satisfont aussi la propriété suivante :

$$\oplus(a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n) \leq \oplus(a_1, \ldots, a_i - \epsilon, \ldots, a_j + \epsilon, \ldots, a_n), \ \forall \epsilon \in [0, a_j - a_i].$$

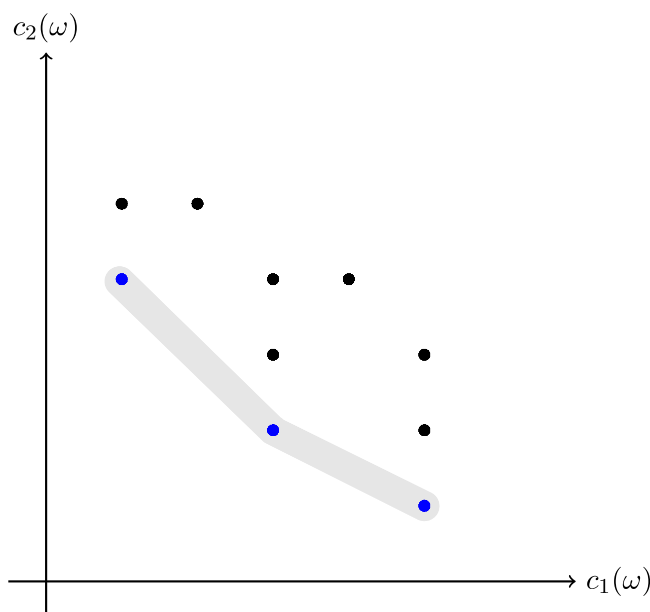


FIGURE 2.2 – Illustration du front de Pareto (ensemble des solutions non Pareto-dominées). Chacun des axes représente une fonction de coût, les points noirs et bleus représentent les solutions du problème. Les points bleus sont les points du front de Pareto, illustré en gris.

Si l'inégalité est stricte, on dit que la fonction est *fortement convexe au sens de Schur*. La Schur-convexité implique la *dominance au sens de Lorenz* (ou *Lorenz-dominance*) [GOLDEN & PERNY 2010]. La dominance de Lorenz se base sur le vecteur de Lorenz, défini comme un vecteur de n valeurs telle que la valeur d'indice i est la somme des i plus petites valeurs du vecteur de coûts de l'alternative ($c_{(1)} \leq \ldots \leq c_{(n)}$ est une permutation ordonnée de $\{c_1, \ldots, c_n\}$) :

$$L(\omega) = (c_{(1)}, c_{(1)} + c_{(2)}, \ldots, c_{(1)} + \ldots + c_{(n)})$$

Un vecteur domine ainsi un autre vecteur (strictement) au sens de Lorenz si et seulement si son vecteur de Lorenz domine (strictement) le vecteur de Lorenz du deuxième vecteur de coûts au sens de Pareto. La relation associée $\succeq_L$ est définie par :

$$c_1 = (c_{1,1}, \ldots, c_{1,n}) \succeq_L c_2 = (c_{2,1}, \ldots, c_{2,n}) \Leftrightarrow L(c_1) \succeq_P L(c_2).$$

La relation de dominance stricte est de ce fait définie de la façon suivante :

$$c_1 = (c_{1,1}, \ldots, c_{1,n}) \succ_L c_2 = (c_{2,1}, \ldots, c_{2,n}) \Leftrightarrow L(c_1) \succ_P L(c_2).$$

Les fonctions d'agrégation respectant le préordre engendré par la dominance de Lorenz permettent de respecter un principe très important en ce qui concerne l'agrégation de critères, le *principe des transferts de Pigou-Dalton* [DALTON 1920, LUSS 1999].

Ce principe formalise que l'on gagne à transférer un coût d'un critère très mauvais vers un critère meilleur. Cette idée permet de privilégier les alternatives pour lesquelles les coûts sont relativement équilibrés plutôt que celles qui sont jugées très mauvaises sur certains critères alors qu'elles sont jugées très satisfaisantes pour d'autres ; ce principe semble souhaitable dans la mesure où l'agrégation vise à rechercher un compromis. Par exemple, si on considère deux vecteurs de coûts $c_1 = (2, 5, 3)$ et $c_2 = (2, 4, 4)$, on voit qu'il est possible d'obtenir c_2 à partir de c_1 en transférant un coût de 1 depuis

le deuxième coût vers le troisième. Puisque dans c_1 le deuxième coût est plus important que le troisième, alors le respect du principe de Pigou-Dalton implique que la solution dont le vecteur de coûts est c_2 doit être préférée à celle dont le vecteur est c_1.

2.2.4 Prise en compte du potentiel

Dans la section précédente, nous avons vu qu'il est jugé souhaitable qu'une fonction d'agrégation respecte le principe des transferts de Pigou-Dalton, afin d'obtenir des solutions équilibrées, impliquant ainsi une notion de compromis. Afin de respecter ce principe, il est nécessaire de faire en sorte que les fonctions objectifs représentent le coût réellement engendré par une solution comparée à une autre.

Or, il existe des cas dans lesquels certains critères ne peuvent être pleinement satisfaits par aucune des alternatives réalisables du problème ; dit d'une autre façon, bien qu'une fonction de coût soit définie de manière à ce que ses images soient dans l'intervalle $[0, 1]$, il existe des problèmes pour lesquels aucune alternative ne permettent d'obtenir un coût nul.

En effet, considérons la formule de la figure 1.4(a) (page 18), qui modélise le problème de gestion de dépendances de la figure 1.1 (page 13) né du souhait d'installer le paquet `e`. Considérons le problème d'optimisation monocritère « installer le moins de paquets possible ». Pour toute solution, on devra installer au moins deux nouveaux paquets (le minimum étant atteint en installant le paquet `e` et le paquet `f`, respectivement en versions 3 et 1). Ainsi, si on ne considère plus le coût réel en terme de nombre de paquets, mais le surcoût occasionné par une solution qui installerait plus de deux paquets, on considérera qu'une solution qui installe uniquement deux paquets sera optimale pour ce critère. Dans la littérature, une telle fonction de surcoût est nommée *fonction de regret* [SAVAGE 1951, LOOMES & SUGDEN 1982].

Ainsi, si on souhaite considérer le potentiel de chacun des critères, il est nécessaire de calculer la valeur optimale pour chacun des critères, et donc d'effectuer n appels à un problème d'optimisation monocritère. Le vecteur composé des n valeurs optimales représente le *point idéal* [STEUER & CHOO 1983, MARLER & ARORA 2004] du problème d'optimisation multicritère. Il est à noter que ce *point idéal* ne correspond bien souvent pas une alternative réalisable.

Définition 2.8 (Point idéal) *Soit Ω l'ensemble des alternatives réalisables, et soit $c_1, c_2, ..., c_n$ les fonctions de coût associées aux n critères. Le point idéal $(c_1^*, c_2^*, ..., c_n^*) \in [0, 1]^n$ est le vecteur tel que :*

$$\forall i \in \{1, 2, ..., n\},\ c_i^* = \min(\{c_i(\omega) \mid \omega \in \Omega\}).$$

Le calcul de ce point idéal, utile pour les agrégations tenant compte du potentiel des alternatives pour chacun des critères, est un surcoût en terme de temps de calcul qui peut être très lourd. De ce fait, il est important de déterminer si une fonction d'agrégation nécessite réellement son calcul. En effet, si l'agrégation se résume par exemple à effectuer une moyenne des coûts intermédiaires, ce calcul n'est pas nécessaire, puisque considérer des fonctions de regret en lieu et place des fonctions de coût ne changera pas l'ordre induit par les valeurs des agrégations. Une illustration du point idéal d'un problème d'optimisation bicritère est présenté à la figure 2.3.

Nous avons maintenant défini les principales propriétés attendues pour les fonctions d'agrégation. Nous présentons maintenant un ensemble de fonctions employées de manière usuelle pour l'agrégation de critères.

2.2.5 Fonctions d'agrégations de critères usuelles

Dans cette section, nous présentons quelques fonctions utilisées de manière usuelle pour l'agrégation des critères, ou des indices de préférence partiels en ce qui concerne l'approche « comparer puis agréger ».

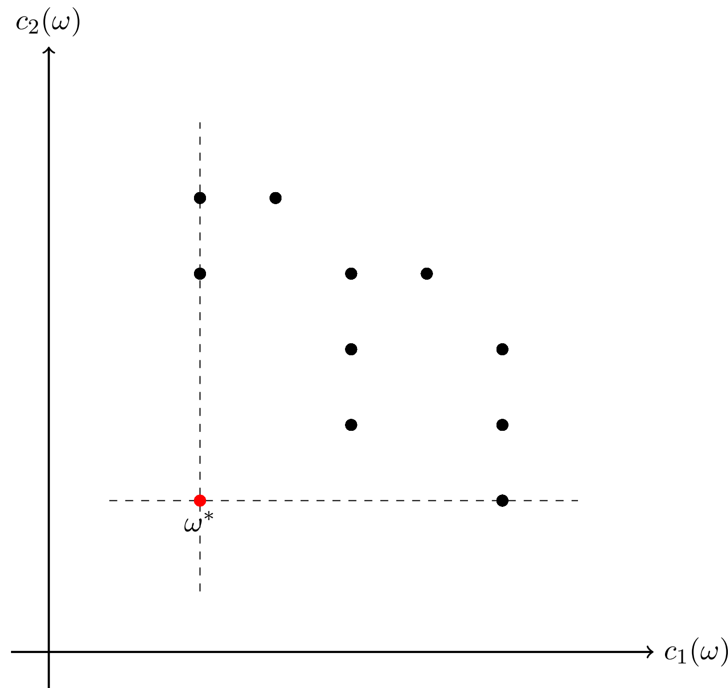


FIGURE 2.3 – Illustration du point idéal (en rouge) pour un problème bicritère. Chacun des axes représente une fonction de coût, les points noirs représentent les solutions du problèmes.

Sommes et moyennes pondérées

Soient $w_1, \ldots, w_n \in [0, 1]$ des poids tels que $w_1 + \ldots + w_n = 1$. Les sommes (ou moyennes) pondérées sont définies par :

$$\oplus(c_1, \ldots, c_n) = \sum_{i=1}^{n}(w_i c_i).$$

Ces fonctions d'agrégation respectent les propriétés d'idempotence et de stricte monotonie ; une solution optimale pour la somme pondérée est de ce fait Pareto-optimale [BOUYSSOU *et al.* 2006]. En revanche, la relation de préférence induite par la somme des coûts ne respecte pas la dominance de Lorenz, et ne suit donc pas le principe des transferts de Pigou-Dalton. De ce fait, en supposant des poids tous égaux à $1/n$, un vecteur de coûts $(0, 1)$ sera jugé équivalent à un vecteur de coûts $(0.5, 0.5)$, ce qui va à l'encontre du principe d'un choix d'une solution équilibrée sur l'ensemble de ses critères ; ce phénomène peut être observé à la figure 2.4. On note que dans le cas particulier où tous les poids sont égaux, la moyenne pondérée respecte la propriété de neutralité.

Il est aussi à noter que dans le cas fréquent où chacun des critères est représenté par une somme de variables, utiliser la somme pondérée comme agrégateur permet de se retrouver dans le cadre de l'optimisation monocritère. De plus, agréger en utilisant la somme pondérée ne nécessite pas de prendre en compte le potentiel des critères, puisque cela reviendrait à retirer la valeur constante $\sum_{i=1}^{n} c_i^*$ du coût global pour toutes les alternatives.

Ordre lexicographique

Cette méthode n'est pas à proprement parler une fonction d'agrégation, puisqu'elle n'effectue pas une synthèse des scores selon chacun des critères. Imposer un ordre lexicographique revient à comparer

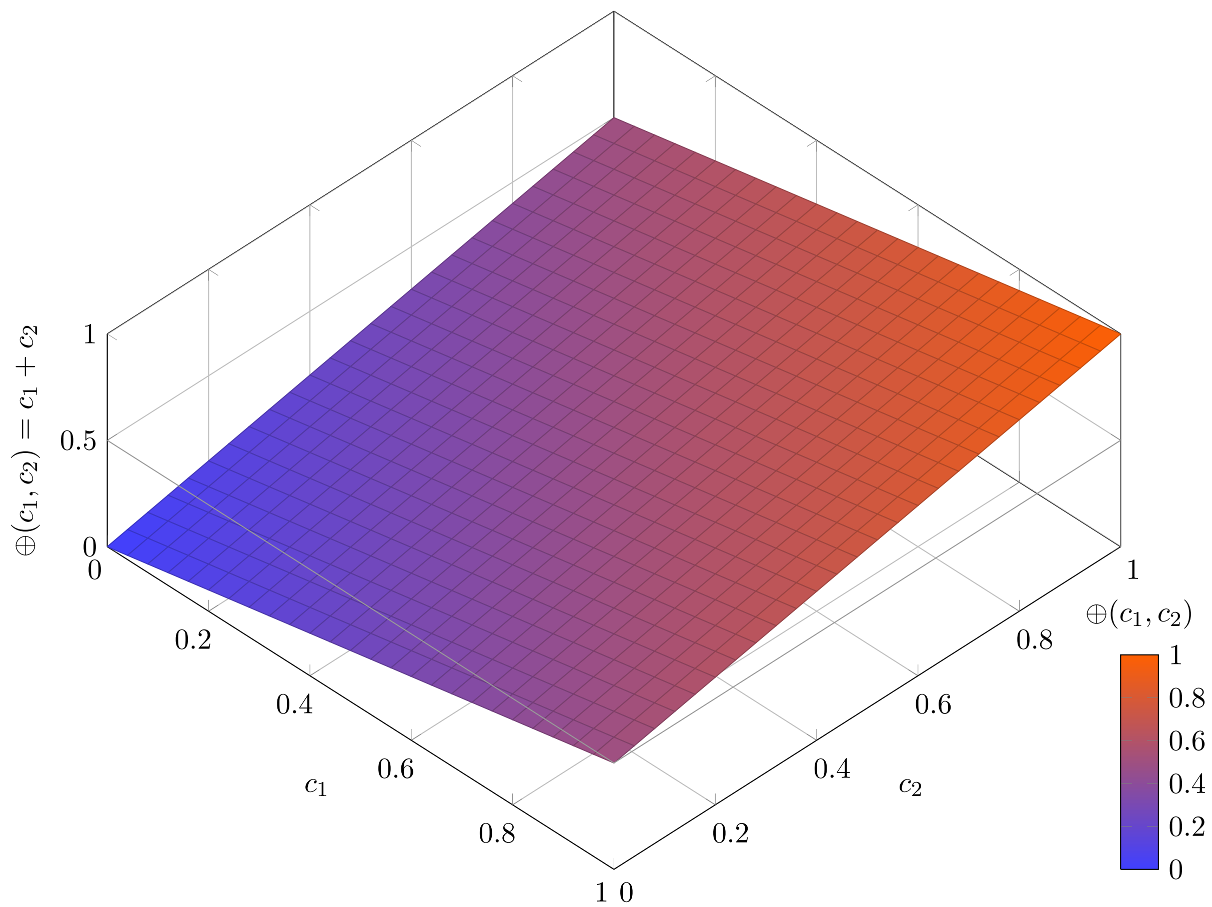


FIGURE 2.4 – Illustration d'une agrégation de deux critères en utilisant la moyenne pondérée.

les alternatives critère par critère, en ne gardant à chaque étape que les solutions optimales pour le critère considéré. Plus précisément, on calcule l'ensemble des solutions optimales en utilisant l'algorithme suivant :

1. déterminer l'ensemble des solutions optimales pour le critère 1 (le plus important) ;
2. parmi l'ensemble des solutions optimales pour le critère 1, déterminer l'ensemble des solutions optimales pour le critère 2 ;
3. ...
4. parmi l'ensemble des solutions optimales pour le critère $n-1$, déterminer l'ensemble des solutions optimales pour le critère n (le moins important).

L'utilisation de cette méthode n'a évidemment de sens que dans le cas où une hiérarchie stricte est présente entre les critères. Par exemple, dans le cadre d'un problème de gestion de dépendances logicielles, on peut souhaiter avant toute chose s'assurer de retirer le moins de paquets possible, de manière à minimiser le nombre de fonctionnalités qui disparaîtraient du système. On pourrait ensuite utiliser d'autres critères pour départager les solutions qui retirent le moins de paquets.

Notons que lorsque les coûts sont bornés par une valeur, on peut réduire le problème de l'optimisation avec ordre lexicographique à celui de la somme pondérée [PHILLIPS 1987] : il s'agit dans ce cas de déterminer les poids de telle manière que pour tout $i \in \{1, \ldots, n-1\}$, on ait le rapport w_{i+1}/w_i qui soit supérieur à la valeur de coût maximale pour le critère w_i, de manière à empêcher la compensation entre critères.

L'optimisation par ordre lexicographique assure que la solution optimale est Pareto-optimale ; cependant, par définition, les propriétés découlant de la neutralité ne sont pas assurées. Encore une fois, la prise en compte du potentiel des alternatives sur les critères n'est pas utile.

L'opérateur $\max$ et ses dérivés

Les opérateurs de moyenne pondérée ont l'inconvénient de ne pas privilégier les solutions équilibrées, ceci étant dû au fait que l'agrégation additive de ces opérateurs impose une compensation entre les critères. Une solution à ce problème est d'employer des opérateurs considérant le coût le plus important des vecteurs correspondant aux coûts de chacun des critères pour une alternative.

L'opérateur $\max$ est la plus petite des *t-conormes*, qui sont les fonctions d'agrégation $\oplus$ non décroissantes, neutres, et associatives telles que $\oplus(0, \ldots) = 0$ et $\oplus(c_1, \ldots, c_n) \geq \max\{c_i \mid i \in \{1, \ldots, n\}\}$ [FODOR & ROUBENS 1994, MIZUMOTO 1989]. En plus des propriétés issues des t-conormes, l'opérateur $\max$ satisfait les propriétés d'idempotence et de monotonie.

Bien qu'il satisfasse la propriété de monotonie, il ne satisfait pas la monotonie *stricte*. En effet, le classement des vecteurs de coûts n'étant réalisé qu'en fonction du coût le plus élevé, deux vecteurs différant d'un coût qui n'est pas le coût maximal sont jugés équivalents. De ce fait, une solution obtenue grâce à l'opérateur $\max$ n'est pas nécessairement Pareto-optimale.

L'opérateur $\max$ retourne *la statistique d'ordre* n (la statistique d'ordre i est la $i^{\text{ème}}$ plus petite valeur d'un ensemble). Il est toutefois possible de raffiner l'opérateur $\max$ dans le but de retourner des solutions Pareto-optimales en utilisant le leximax. Le leximax est le pendant neutre de l'ordre lexicographique, dans la mesure où il lui correspond une fois les vecteurs de coûts triés du plus important au plus faible. En effet, un algorithme pour récupérer *l'unique* vecteur de coût qui sera préféré aux autres (mais qui peut correspondre à plusieurs alternatives) est le suivant :

1. déterminer l'ensemble des solutions optimales pour la statistique d'ordre n (le $n^{\text{ème}}$ plus grand coût, c'est-à-dire le $\max$) ;
2. parmi l'ensemble des solutions optimales pour la statistique d'ordre n, déterminer l'ensemble des solutions optimales pour la statistique d'ordre $n-1$;

3. ...
4. parmi l'ensemble des solutions optimales la statistique d'ordre 2, déterminer l'ensemble des solutions optimales la statistique d'ordre 1 (le $\min$).

Le leximax est un agrégateur disposant de propriétés très intéressantes :
- il satisfait la monotonie stricte, et il est donc cohérent avec la Pareto-dominance ;
- il respecte aussi l'ordre induit par la dominance de Lorenz, et respecte donc ainsi le principe des transferts de Pigou-Dalton.

En revanche, comme toute t-conorme, une solution leximax-optimale peut coûter globalement très cher en terme de somme de coûts : $(0.99, 0.99, 0.99)$ sera ainsi préférée à $(0, 0, 1)$, ce qui semble très peu naturel. De ce fait, il semble intéressant d'aboutir à un compromis entre cet agrégateur et la somme pondérée, ce qui est permis par les opérateurs de somme pondérée ordonnée présentés à la section suivante.

Finalement, dans le cas où le $\max$ porte sur des fonctions de regrets dans le but de prendre en compte le potentiel des alternatives, on parle alors de *norme de Tchebycheff* [STEUER & CHOO 1983, WIERZBICKI 1986] au point idéal.

Opérateurs de somme pondérée ordonnée

Les opérateurs de moyenne (ou somme) pondérée ordonnée (OWA pour *Ordered Weighted Average operators*, ou bien encore OWS pour *Ordered Weighted Sum operators*) [YAGER 1988] sont définis comme étant une somme où les critères sont pondérés en fonction de leur rang. Plus précisément, un poids est attribué à la statistique de rang 1, un autre à celle de rang 2, et ainsi de suite jusqu'au rang n. Formellement, la définition d'un OWA est la suivante :

$$\text{OWA}_w(c_1, \ldots, c_n) = \sum_{i=1}^{n} w_i c_{(i)}$$

où $w = (w_1, \ldots, w_n)$ tel que la somme des w_i soit égale à 1, et où l'ensemble des $c_{(i)}$ est une permutation de l'ensemble des c_i telle que $c_{(1)} \leq \ldots \leq c_{(n)}$.

Les OWA étant une famille d'agrégateurs très générale, ils contiennent un certain nombre de fonctions que nous avons déjà étudiées précédemment :
- si $w = (1/n, \ldots, 1/n)$, on se retrouve dans le cas de la moyenne arithmétique ;
- si $w = (0, \ldots, 0, 1)$, il s'agit de l'opérateur $\max$;
- plus généralement, les opérateurs de somme pondérée ordonnée sont capables de représenter toutes les statistiques de rang i.

L'ensemble des opérateurs OWA satisfait les propriétés d'idempotence, de monotonie et de neutralité. Si les poids du vecteur w sont croissants ($w_1 \leq \cdots \leq w_n$), on voit bien que l'agrégation agit comme un compromis entre la somme pondérée et le leximax. En effet, plus un coût est important, et plus il sera amplifié, puisque son poids sera plus grand que celui associé aux poids plus faibles. Cependant, bien que les coûts moins importants aient moins d'influence, il agissent tout de même si le poids qui leur est associé n'est pas nul. Si les poids sont strictement croissants ($w_1 < \ldots < w_n$), l'opérateur OWA est strictement monotone : dans ce cas, l'ordre des alternatives respecte le principe des transferts de Pigou-Dalton, ce qui fait de ces opérateurs des cibles de choix en terme de *bonnes* propriétés. Dans l'exemple de la figure 2.5, on voit d'ailleurs que les solutions équilibrées sont préférées à celles dont la somme des coûts est la même, mais dont les écarts entre les coûts sont plus importants.

Les opérateurs que nous avons présentés jusqu'à présent possèdent l'inconvénient de ne pas tenir compte des synergies qui pourraient exister entre certains critères, comme par exemple le fait qu'un coût important pour un critère implique forcément qu'un autre aura lui aussi un coût important. Afin qu'un

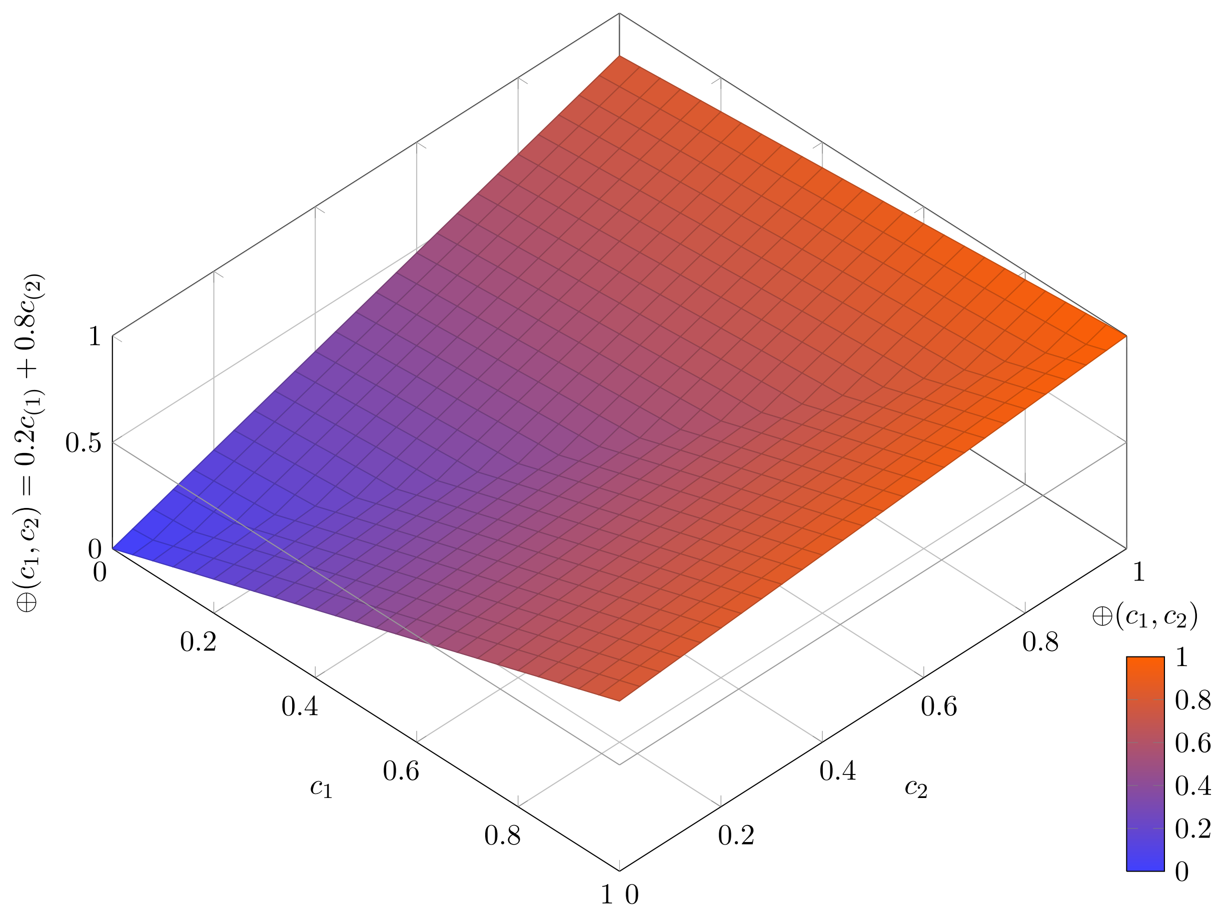


FIGURE 2.5 – Illustration d'une agrégation de deux critères en utilisant un opérateur de somme pondérée ordonnée.

décideur puisse définir des règles en terme d'agrégation de critères en tenant compte de ces synergies, il est nécessaire d'employer des agrégateurs plus généraux, comme les intégrales floues.

Intégrales de Sugeno et de Choquet

Les intégrales de Sugeno [SUGENO 1974] et de Choquet [CHOQUET 1953, MUROFUSHI & SUGENO 1991] sont basées sur la notion de *mesure floue*, aussi appelée *capacité*.

Définition 2.9 (Mesure floue) *Une mesure floue sur un ensemble N est une fonction μ de l'ensemble des parties de N dans $[0,1]$ telle que :*
- $\mu(\emptyset) = 0$ *et* $\mu(N) = 1$ *;*
- $A \subset B \Rightarrow \mu(A) \leq \mu(B)$.

Une mesure floue représente l'importance de chacune des coalitions de critères, et permet donc de cette façon de gérer les synergies, positives ou négatives, qui pourraient exister entre certains critères. Dans le cas où aucune synergie n'apparaît, une mesure floue est dite *additive* ; elle est formellement définie de la façon suivante :

$$\forall A \in \mathcal{P}(N),\ \mu(A) = \sum_{i \in A} \mu(\{i\}).$$

Un autre cas de capacité particulière est la *capacité cardinale*, ou *symétrique* : il s'agit en fait d'une capacité telle que la valeur ne dépend que de la cardinalité de l'ensemble de critères considérés, et non pas des critères eux-mêmes (on a $\mu(A) = \mu(B)$ si $|A| = |B|$).

Ces capacités sont ensuite utilisées par des intégrales employées comme agrégateurs, une intégrale représentant d'une certaine façon une moyenne dans le cas discret. Dans la suite de cette section, on reprend la notation $(.)$ pour désigner une permutation de l'ensemble des critères $N = \{1, \dots, n\}$ telle que $c_{(1)} \leq \dots \leq c_{(n)}$. On définit aussi les ensembles A_i basés sur la permutation $(.)$ considérée, où $A_i = \{(i), \dots, (n)\}$.

On distingue deux types d'intégrales utilisées comme opérateurs d'agrégation, dont l'intégrale de Sugeno [SUGENO 1974] :

$$\mathcal{S}_\mu(c_1, \dots, c_n) = \max\{\min(c_{(i)}, \mu(A_i)) \mid i \in N\}.$$

L'utilisation de cette intégrale possède toutefois le même inconvénient que les autres méthodes basées sur l'opérateur $\max$, à savoir que la valeur obtenue par l'agrégation ignore en fait $n-1$ critères. De ce fait, on utilisera plutôt une autre intégrale floue, l'intégrale de Choquet [MUROFUSHI & SUGENO 1991] (basée sur des travaux de [CHOQUET 1953]), qui est basée sur une somme ordonnée :

$$\mathcal{C}_\mu(c_1, \dots, c_n) = \sum_{i=1}^{n} (c_{(i)} - c_{(i-1)})\mu(A_i),\ \text{avec } c_{(0)} = 0.$$

L'intégrale de Choquet, comme celle de Sugeno, est idempotente et monotone. L'intégrale de Choquet définit en fait une classe d'agrégateurs très générale contenant de nombreux opérateurs de notre étude :
- une intégrale de Choquet basée sur une capacité additive correspond à une somme pondérée dont les poids w_i sont égaux à $\mu(\{i\})$;
- tout opérateur OWA est une intégrale de Choquet dont la capacité est définie par $\mu(A) = \sum_{i=0}^{|A|-1} w_{n-j}$; réciproquement, toute intégrale de Choquet dont la capacité est symétrique est un opérateur OWA ;
- de ce fait, les intégrales de Choquet contiennent aussi les moyennes arithmétiques et les statistiques d'ordre, y compris le $\max$.

Exemple 2.2 (Exemples d'agrégation par des intégrales floues) *Soit μ une capacité telle que :*

$$\begin{array}{lll} \mu(\emptyset) = 0 & & \\ \mu(\{1\}) = 0.4 & \mu(\{2\}) = 0.4 & \mu(\{3\}) = 0.3 \\ \mu(\{1,2\}) = 0.7 & \mu(\{1,3\}) = 0.7 & \mu(\{2,3\}) = 0.6 \\ \mu(\{1,2,3\}) = 1. & & \end{array}$$

Soit un vecteur de coûts $c = (0.3, 0.6, 0.5)$. La valeur de l'intégrale de Sugeno de c pour la capacité μ est la suivante :

$$\begin{aligned} \mathcal{S}_\mu(0.3, 0.6, 0.5) &= \max(\min(0.3, \mu(\{1,2,3\})), \min(0.5, \mu(\{2,3\})), \min(0.6, \mu(\{2\}))) \\ &= \max(0.3, 0.5, 0.4) \\ &= 0.5\ . \end{aligned}$$

La valeur de l'intégrale de Choquet de c pour la capacité μ est la suivante :

$$\begin{aligned} \mathcal{C}_\mu(0.3, 0.6, 0.5) &= (0.3 - 0)\mu(\{1,2,3\}) + (0.5 - 0.3)\mu(\{2,3\}) + (0.6 - 0.5)\mu(\{2\}) \\ &= 0.3 + 0.12 + 0.04 \\ &= 0.46\ . \end{aligned}$$

La difficulté de l'utilisation de l'intégrale de Choquet est due à la détermination d'une « bonne » capacité, puisqu'il faut réussir à déterminer les synergies qui existent entre tous les sous-ensembles de critères possibles. Néanmoins, il est possible de définir des capacités (dites *k-additives*) ne tenant compte des synergies que pour des coalitions de k critères au maximum. Le cas $k = 2$ a notamment été mis en valeur dans des travaux récents [Clivillé *et al.* 2007, Berrah & Clivillé 2007, Grabisch & Labreuche 2008].

2.3 Conclusion du chapitre

Dans ce chapitre, nous avons présenté plusieurs notions clés concernant la thématique de l'optimisation, qui nous seront utiles par la suite. Nous nous sommes tout d'abord intéressés à la problématique de l'optimisation monocritère, en présentant les relations de préférence. Nous avons vu que dans de nombreux cas, on pouvait déterminer une fonction mathématique associant un coût (ou une utilité) à toute solution, de telle sorte que l'ordre donné par les valeurs de cette fonction soit cohérent avec l'ordre imposé par la relation de préférence.

Nous avons ensuite étudié différentes familles de représentation de fonctions objectifs, en les définissant comme des bases pondérées munies d'un opérateur d'agrégation. Nous avons noté que ce formalisme permet notamment de représenter les fonctions objectifs définies usuellement dans les travaux concernant l'optimisation sous contraintes exprimées par des formules de logique propositionnelle.

Nous avons ensuite présenté les bases de l'optimisation multicritère, notamment les approches « agréger puis comparer » (AC) et « comparer puis agréger » (CA). Finalement, nous avons présenté les propriétés attendues d'une agrégation de critères, avant de présenter des opérateurs d'agrégation fréquemment utilisés, notamment les opérateurs de somme pondérée ordonnée et l'intégrale de Choquet.

Dans le chapitre suivant, nous présentons un certain nombre de langages propositionnels utilisés dans le cadre de la compilation de connaissances, afin de disposer de tous les préliminaires utiles à la présentation de notre première contribution au quatrième chapitre.

Chapitre 3

Compilation de connaissances : le langage NNF et ses sous-langages

Sommaire

3.1 Définitions du langage NNF et de ses sous-langages **46**
3.1.1 Le langage NNF et ses propriétés 46
3.1.2 Diagrammes de décision binaires 49
3.1.3 DNNF structurées 52
3.2 Choisir un langage de compilation **53**
3.2.1 Requêtes 53
3.2.2 Transformations 54
3.2.3 Concision des langages étudiés 58
3.3 Conclusion du chapitre **58**

La compilation de connaissances apparaît depuis quelques années comme une direction majeure vers laquelle s'orienter pour attaquer les problèmes intraitables [KORICHE *et al.* 2013, AMILHASTRE *et al.* 2014]. L'idée derrière l'approche par compilation est de traduire un problème d'un langage source permettant une modélisation aisée, comme CNF, vers un langage cible permettant de répondre à des *requêtes* en temps polynomial. La phase de traduction étant généralement complexe, ce processus sied particulièrement aux situations où un nombre important d'appels à des algorithmes de complexité théorique élevée pour le langage initial, mais faible pour le langage cible, est nécessaire pour traiter un problème, ou une succession de problèmes portant sur les mêmes contraintes.

En ce qui concerne l'optimisation, l'intérêt potentiel de la compilation apparaît rapidement, de par l'existence d'algorithmes basés sur l'architecture « SAT incrémental » (étudiée plus en détail dans un chapitre ultérieur). En effet, en supposant qu'un langage cible permette de définir une contrainte de borne (retirer les modèles de coûts supérieur à k) en temps polynomial (pour le peu que la valeur de la borne soit polynomiale en la taille de l'entrée), et aussi de déterminer la cohérence d'une formule en temps polynomial, on obtient un algorithme d'optimisation s'exécutant en temps polynomial : tant que la formule est cohérente, on extrait un modèle, et on retire les modèles ayant un coût supérieur pour la fonction objectif considérée ; le dernier modèle obtenu est optimal.

L'intérêt de la compilation de connaissances apparaît encore plus évident dans le cadre des problèmes qui sont déterminés par un ensemble de contraintes et par un état initial, ce qui est par exemple le cas du problème de gestion de dépendances logicielles. Plus précisément, les contraintes sont les mêmes pour tout le monde (peu importe l'utilisateur, deux paquets mutuellement exclusifs ne peuvent être installés ensemble), mais chaque utilisateur possède une configuration initiale (*i.e.* un ensemble de paquets installés) différents. Puisque de nombreux langages cibles permettent d'encoder ces situations initiales en temps polynomial dans la base de contraintes compilée, une unique phase de compilation est nécessaire pour donner la possibilité à une multitude d'utilisateurs d'obtenir des algorithmes polynomiaux à des requêtes de complexité élevée avant que la formule initiale n'ait été traduite.

Dans ce chapitre, nous présentons l'ensemble des langages pour lesquels nous avons considéré des requêtes d'optimisation. Pour la plupart de ces langages, la définition donnée est issue de la carte de compilation de [DARWICHE & MARQUIS 2002].

3.1 Définitions du langage NNF et de ses sous-langages

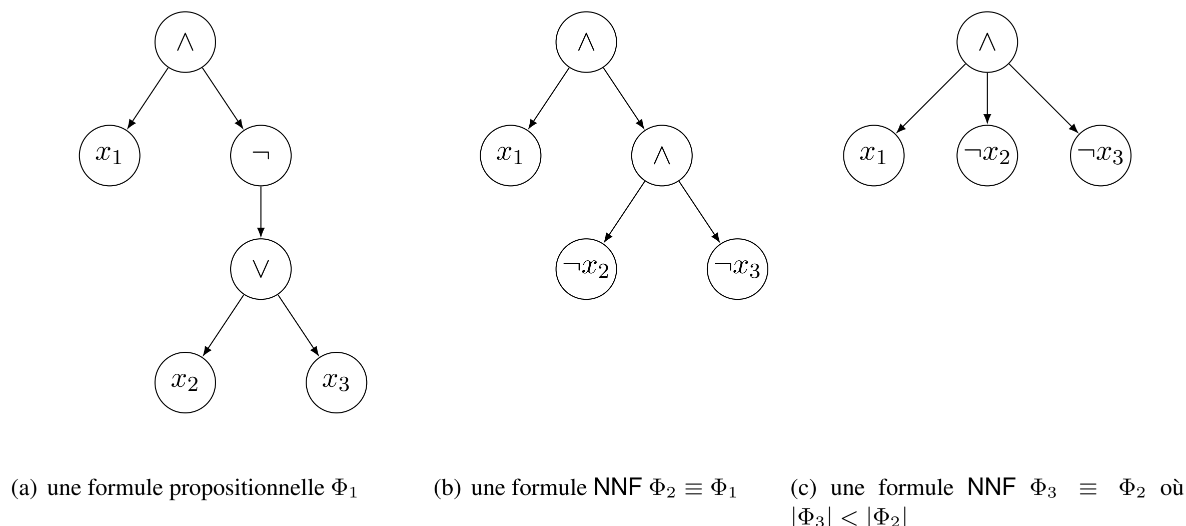
3.1.1 Le langage NNF et ses propriétés

Tout d'abord, on considère une forme normale pour l'ensemble des formules de la logique propositionnelle, les formules NNF, définie dans [DARWICHE 1999, DARWICHE 2001], dont la définition a été présentée à la section 1.2.1. On rappelle aussi que toute formule propositionnelle sur PS construite sur la morphologie $\{\neg, \vee, \wedge\}$ peut être écrite comme une formule NNF en temps polynomial en appliquant de manière récursive les lois de De Morgan (voir figures 3.1(a) et 3.1(b)) de la racine jusqu'au feuilles pour « descendre » les négations au niveau des feuilles :

$$\neg(x_1 \wedge \ldots \wedge x_n) \equiv \neg x_1 \vee \ldots \vee \neg x_n$$
$$\neg(x_1 \vee \ldots \vee x_n) \equiv \neg x_1 \wedge \ldots \wedge \neg x_n.$$

Enfin, comme dans le cadre de la logique propositionnelle, l'associativité des opérateurs de conjonction et de disjonction permet de considérer des nœuds d'arité quelconques. Il peut être utile dans une optique de gain d'espace, après application des lois de De Morgan, de vérifier (à nouveau) qu'un nœud $\wedge$ (resp. $\vee$) n'a pas pour fils un autre nœud $\wedge$ (resp. $\vee$), comme le montrent les figures 3.1(b) et 3.1(c).

Le langage NNF, très général, n'est évidemment pas un langage intéressant pour la compilation, puisque qu'il inclut le langage CNF qui ne dispose lui même pas des propriétés attendues en terme de



(a) une formule propositionnelle Φ_1 (b) une formule NNF $\Phi_2 \equiv \Phi_1$ (c) une formule NNF $\Phi_3 \equiv \Phi_2$ où $|\Phi_3| < |\Phi_2|$

FIGURE 3.1 – Mise sous forme NNF d'une formule quelconque.

requêtes traitables. Plus précisément, CNF est le sous-langage de NNF des formules qui satisfont les propriétés *flatness* et *simple disjunction*.

Définition 3.1 (f-NNF) *Une formule* NNF *satisfait la propriété* flatness *(aplatissement) si et seulement si sa hauteur est de deux au plus. L'ensemble des formules* NNF *qui satisfont cette propriété forment le langage* f-NNF.

Parmi ces formules, on retrouve donc les formules CNF, dont une définition a déjà été donnée à la section 1.2.1. Les formules CNF sont des formules *flat* qui satisfont aussi la propriété de *simple disjunction*.

Définition 3.2 (CNF) *Une formule* NNF *satisfait la propriété* simple disjunction *(disjonction simple) si et seulement si les fils des nœuds* $\vee$ *sont des feuilles qui ne partagent pas de variables. L'ensemble des formules* f-NNF *qui satisfont cette propriété forme le langage* CNF.

Parmi les formules f-NNF, on retrouve aussi d'autres formules bien connues, les formules sous forme normale disjonctive, qui sont les formules f-NNF qui satisfont la propriété *simple conjunction*. Ces formules ont elles-aussi été définies à la section 1.2.1.

Définition 3.3 (DNF) *Une formule* NNF *satisfait la propriété* simple conjunction *(conjonction simple) si et seulement si les fils des nœuds* $\wedge$ *sont des feuilles qui ne partagent pas de variables. L'ensemble des formules* f-NNF *qui satisfont cette propriété forme le langage* DNF.

Il est à noter que les propriétés de *simple disjunction* et de *simple conjunction* imposent que les nœuds non terminaux ne partagent pas de variables. Or, si cela est le cas, il est possible de transformer la formule considérée en temps polynomial en une formule équivalente pour laquelle les occurrences multiples ont disparu. En effet, il suffit de laisser au maximum une seule occurrence d'un littéral présent à plusieurs reprises, puis de remplacer le nœud $\vee$ (resp. $\wedge$) par un nœud $\top$ (resp. $\bot$) si deux littéraux opposés apparaissent.

Il est aussi à noter que toute formule $\Phi \in$ f-NNF peut être écrite simplement comme une formule CNF ou une formule DNF. En effet, si Φ a une hauteur de 0 (Φ est un littéral) ou 1 (Φ est une clause ou

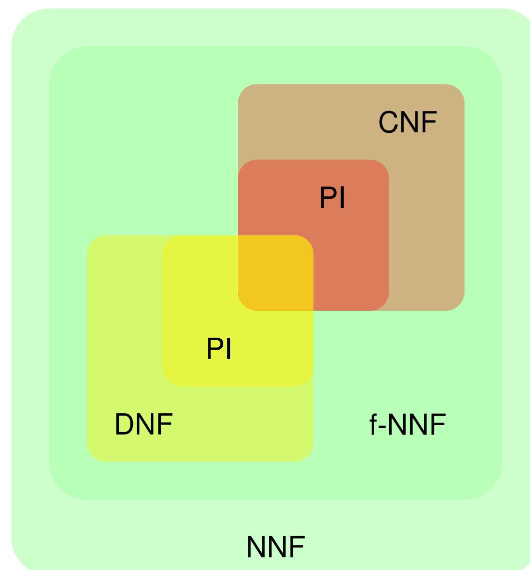


FIGURE 3.2 – Relations d'inclusion entre les sous-langages de f-NNF. La zone centrale correspond exactement aux formules de hauteur inférieure ou égale à 1.

un cube), alors Φ est à la fois une formule CNF et une formule DNF. En revanche, si Φ a une hauteur de deux, il suffit alors de faire en sorte que la racine $\wedge$ (resp. $\vee$) n'ait pas de nœud $\wedge$ (resp. $\vee$) parmi ses fils, en faisant « remonter » les fils de ce nœud fils pour qu'ils deviennent fils de la racine (on exploite ici les propriétés de distributivité de $\wedge$ sur $\vee$ et de $\vee$ sur $\wedge$).

Bien que CNF n'admette pas de bonnes propriétés, un de ses sous-langage ayant de bien meilleures propriétés a été beaucoup étudié dans la littérature : il s'agit du langage des impliqués premiers, que l'on note usuellement PI [MARQUIS 2000].

Définition 3.4 (PI) *Le langage* PI *est le sous-ensemble des formules du langage* CNF *pour lesquelles :*
- *toute clause impliquée par la formule est sous-sommée par une clause de la formule (l'ensemble des littéraux de la deuxième clause est inclus dans la première) ;*
- *aucune clause de la formule n'est sous-sommée par une autre clause de la formule.*

De manière duale, le langage IP est un raffinement du langage DNF.

Définition 3.5 (IP) *Le langage* IP *est le sous-ensemble des formules du langage* DNF *pour lesquelles :*
- *tout cube impliquant la formule sous-somme un terme de la formule (l'ensemble des littéraux du deuxième terme est inclus dans le premier) ;*
- *aucun terme de la formule n'est sous-sommée par un autre terme de la formule.*

Ce langage clôt la liste des langages satisfaisant la propriété *flatness* de notre étude. Il est à noter que toute formule de hauteur inférieure ou égale à 1 (c'est-à-dire, un cube ou un terme), appartient soit à PI, soit à IP. Une représentation des relations d'inclusion entre les sous-langages de f-NNF est donnée à la figure 3.2.

Nous nous intéressons maintenant aux langages pour lesquels un entrelacement de nœuds $\wedge$ et $\vee$ peut exister. Les définitions de ces langages sont basées sur trois propriétés basées sur les nœuds $\wedge$ ou $\vee$. Tout d'abord, nous présentons la propriété de décomposabilité.

Définition 3.6 (DNNF) *Une formule* NNF *satisfait la propriété* decomposability *(décomposabilité) si et seulement si, pour chaque nœud* $\wedge$*, il n'existe pas deux nœuds fils partageant une ou plusieurs variables. Le langage des formules* NNF *satisfaisant la propriété de décomposabilité est nommé* DNNF.

Comme nous le verrons par la suite, cette propriété très importante apporte le test de la cohérence d'une formule en temps polynomial. C'est d'ailleurs grâce à cette propriété que le test de cohérence est traitable pour les formules DNF (DNF est un sous-ensemble de DNNF).

Nous présentons maintenant la propriété de déterminisme.

Définition 3.7 (d-NNF) *Une formule* NNF *satisfait la propriété* determinism *(déterminisme) si et seulement si, pour chaque nœud* $\vee$*, il n'existe pas deux nœuds fils dont la conjonction est cohérente. Le langage des formules* NNF *satisfaisant la propriété de déterminisme est nommé* d-NNF.

Cette propriété, seule, n'apporte pas de résultats intéressants ; en revanche, en conjonction avec la décomposabilité, le déterminisme permet d'apporter des propriétés intéressantes, comme celle de rendre traitable le comptage de modèle ou le test d'implication de formules. De ce fait, nous considérons aussi le langage d-DNNF, le sous-langage de DNNF qui satisfait la propriété de déterminisme en plus de satisfaire la propriété de décomposabilité.

Enfin, la troisième propriété utile à l'identification de nos sous-ensembles de NNF est la propriété *smoothness*.

Définition 3.8 (s-NNF) *Une formule* NNF *satisfait la propriété* smoothness *(uniformité) si et seulement si, pour chaque nœud* $\vee$*, il n'existe pas deux nœuds fils qui portent sur des ensembles de variables différents. Le langage des formules* NNF *satisfaisant la propriété de smoothness est nommé* s-NNF.

Il est à noter qu'apporter la propriété *smoothness* à une formule NNF peut être réalisé en temps polynomial, au prix toutefois d'une augmentation de la taille de la formule. De plus, considérer le sous-ensemble de CNF qui satisfait *smoothness* n'a pas de sens, puisque cela entre en contradiction avec la propriété *simple disjunction*. Enfin, il n'est pas nécessaire de différencier les formules DNF qui satisfont *smoothness* et les formules IP qui satisfont aussi cette propriété, puisque dans les deux cas, on obtient le langage MODS. En effet, si deux termes portent sur les mêmes variables, soit ils sont contradictoires, soit ils sont identiques à l'ordre des littéraux près. Dans ce cas, il suffit de laisser une unique occurrence par sous-ensembles de cubes équivalents pour apporter la propriété de déterminisme, et ainsi obtenir une formule MODS.

Définition 3.9 (MODS) MODS *est le sous-langage de* DNF *dont les formules satisfont aussi les propriétés* determinism *et* smoothness.

Les formules de ce langage correspondent en fait à la disjonction des modèles d'une formule Φ, où les littéraux x pour lesquels la polarité n'influe pas sur la cohérence (c'est-à-dire que $\Phi \wedge x \equiv \Phi \wedge \neg x$) peuvent être omis. De ce fait, $\top$ est une formule de MODS. Une représentation des relations d'inclusion entre les sous-langages de DNNF est donnée à la figure 3.3.

Nous présentons maintenant la famille des langages qui sont basés sur des nœuds particuliers appelés *nœuds décision*, la famille des diagrammes de décision binaires.

3.1.2 Diagrammes de décision binaires

La carte de compilation de [DARWICHE & MARQUIS 2002] s'intéresse aussi aux langages dérivés des diagrammes de décision binaires issus de [LEE 1959] (et popularisés par [AKERS 1978] et [BRYANT 1986]),

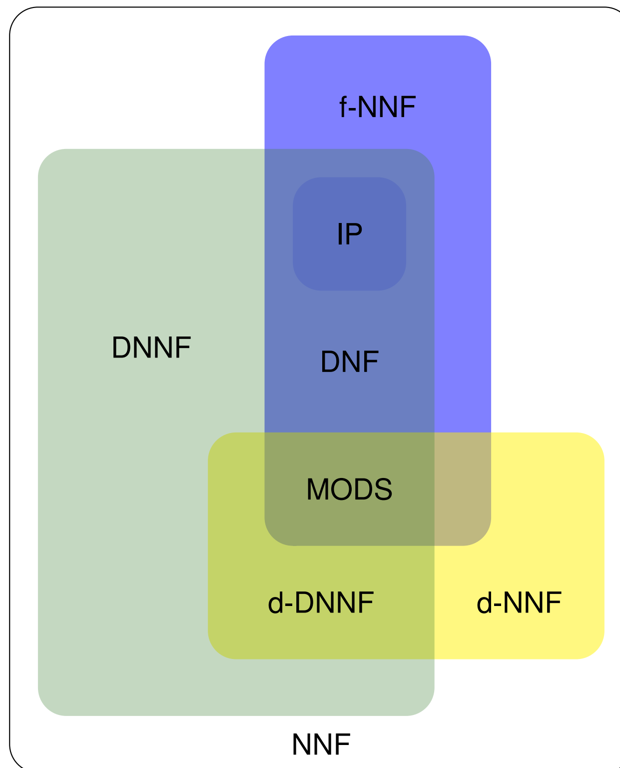


FIGURE 3.3 – Relations d'inclusion entre les sous-langages de DNNF.

qui a étudié la complexité de certains algorithmes opérant sur les diagrammes dans le cas où un ordre sur les variables est imposé lors de la création des diagrammes.

L'idée de ces diagrammes en ce qui concerne la compilation de connaissances est d'utiliser l'identité de Shannon [SHANNON 1949] pour décomposer une formule propositionnelle :

$$\Phi \equiv (x \wedge \Phi_{|x}) \vee (\neg x \wedge \Phi_{|\neg x})$$

où $x \in \Phi$ et $\Phi_{|x}$ représente Φ où les occurrences du littéral x dans Φ sont remplacées par $\top$, et celles de $\neg x$ par $\bot$.

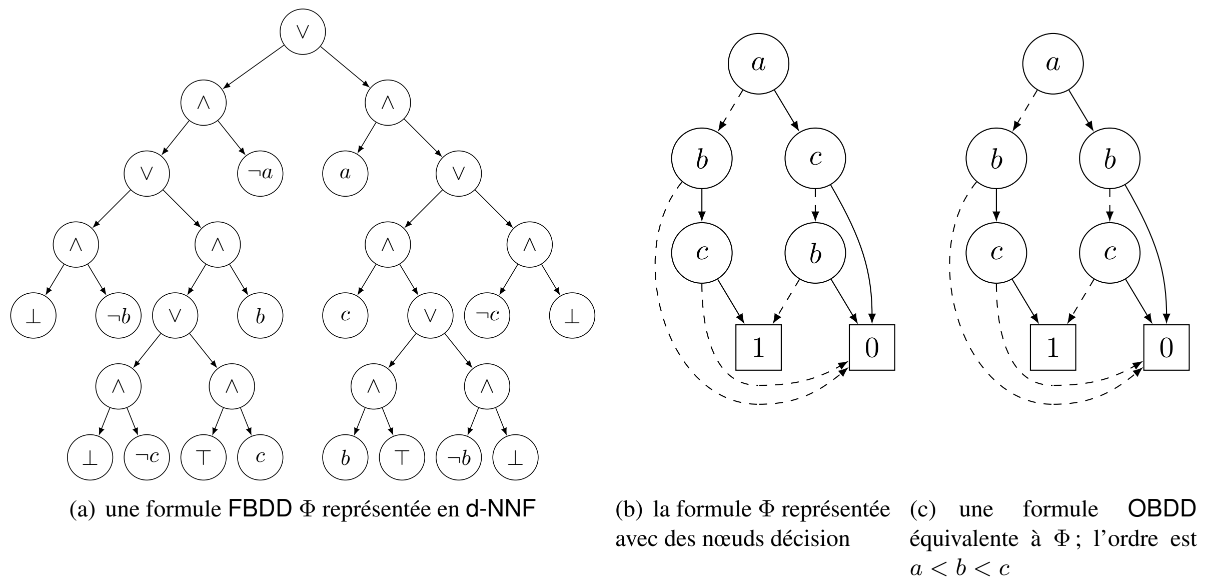
Pour représenter cette identité, la notion de *nœud décision* a été introduite [BRYANT 1986].

Définition 3.10 (Nœuds décision) *Dans une formule* NNF*, un nœud décision N est un nœud étiqueté par $\top$, $\bot$, ou un nœud $\vee$ de la forme $(x \wedge N') \vee (\neg x \wedge N'')$, où x est une variable, et où N' et N'' sont des nœuds décision. La variable x associée au nœud N, appelée variable décision, est notée dVar(N).*

Lorsque tous les nœuds d'une formule NNF sont des nœuds décision (il suffit en fait que la racine soit un tel nœud), alors la formule est un diagramme de décision binaire. On nomme le langage de l'ensemble de ces formules BDD (pour *Binary Decision Diagrams*).

Définition 3.11 (BDD) *Le langage* BDD *est le langage contenant toutes les formules* NNF *dont la racine est un nœud décision.*

Les formules BDD admettent une notation simplifiée, illustrée aux figures 3.4(a) et 3.4(b). Étant donné un nœud décision N de la forme $(x \wedge N') \vee (\neg x \wedge N'')$ où x est le littéral positif, on représente un nœud étiqueté x, relié par deux arcs aux nœuds N' et N''. L'arc correspondant au choix du littéral



(a) une formule FBDD Φ représentée en d-NNF

(b) la formule Φ représentée avec des nœuds décision

(c) une formule OBDD équivalente à Φ ; l'ordre est $a < b < c$

FIGURE 3.4 – Nœuds décision, formules FBDD et formules OBDD

positif, ici celui reliant N et N' est représenté par un trait plein. Celui entre N et N'', en revanche, et dessiné en pointillés.

La propriété de déterminisme est évidemment satisfaite par les formules BDD : la définition des nœuds décision assurant que les nœuds $\vee$ ont exactement deux fils logiquement contradictoires. De ce fait, BDD est un sous-langage de d-NNF. Toutefois, ces formules ne sont pas décomposables dans la mesure où rien n'interdit la présence de deux nœuds décision sur un chemin entre la racine et une feuille. Le sous-langage de BDD qui satisfait la propriété de décomposabilité est le langage FBDD.

Définition 3.12 (FBDD) FBDD *est le langage formé par l'intersection des langages* BDD *et* DNNF.

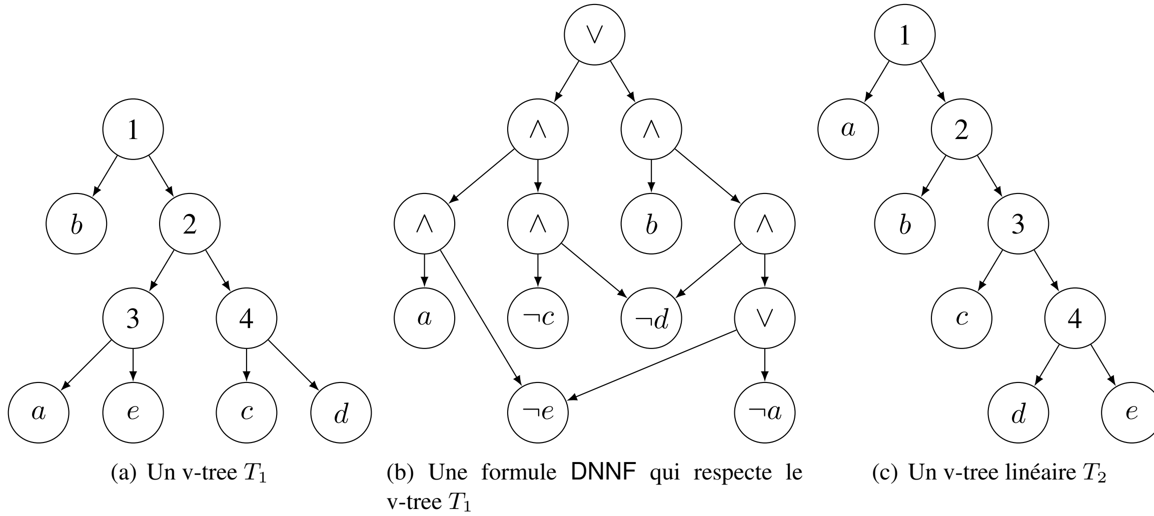
Ces formules sont les *Free Binary Decision Diagrams* présentés par exemple par [GERGOV & MEINEL 1993] dans le cadre de la vérification formelle. Ils sont usuellement définis comme les BDD satisfaisant la propriété *read-once*, c'est-à-dire ceux pour lesquels il n'existe pas deux nœuds décision portant sur la même variable sur un chemin entre la racine de la formule et une de ses feuilles.

Finalement, la dernière famille de diagrammes de décision binaires que nous étudions se base sur un ordre des variables ; il s'agit du langage OBDD (pour *Ordered BDD*).

Définition 3.13 (OBDD) *Soit $<$ un ordre strict et total sur* PS. $\mathsf{OBDD}_<$ *est l'ensemble des formules* FBDD *telles que pour tout couple de nœuds décision (N, N'), si N est un ancêtre de N' alors dVar$(N) <$ dVar(N'). L'ensemble des langages* $\mathsf{OBDD}_<$ *est nommé* OBDD.

Une illustration de formule OBDD est représentée à la figure 3.4(c). Contrairement à la figure 3.4(b) où selon les chemins entre la racine et une feuille, les nœuds décision de la variable b peuvent être des ancêtres ou des descendants de ceux de la variable c, l'ordre $a < b < c$ est imposé pour le diagramme ordonné.

Nous verrons par la suite que cet ordre imposé assure l'existence d'algorithmes pour des opérations qui peuvent se révéler très intéressantes pour la compilation de connaissances.



(a) Un v-tree T_1

(b) Une formule DNNF qui respecte le v-tree T_1

(c) Un v-tree linéaire T_2

FIGURE 3.5 – Exemples de v-tree

3.1.3 DNNF structurées

Dans [PIPATSRISAWAT & DARWICHE 2008], les auteurs généralisent la notion d'ordre utilisée pour le langage OBDD et l'appliquent dans le cadre des formules DNNF. Cette généralisation s'appuie sur la notion de v-tree, un arbre binaire dont les feuilles sont les variables de PS.

Définition 3.14 (v-tree) *Un v-tree pour un ensemble de variables* PS *est un arbre strictement binaire (les seuls nœuds n'ayant pas exactement deux fils sont les feuilles) dont les nœuds des feuilles correspondent exactement à l'ensemble des variables de* PS*.*

Pour la définition suivante, on introduit les deux notations :

- soit N un nœud de v-tree, Vars(N) est l'ensemble des variables apparaissant dans les descendants de N ;
- Soit N un nœud d'un v-tree, N^l est le fils gauche de N, N^r est le fils droit.

Définition 3.15 (SDNNF, d-SDNNF) *Une formule* DNNF *(resp.* d-DNNF*) respecte un v-tree T si et seulement si pour chaque nœud conjonction $N = \Phi_1 \wedge \Phi_2$, il existe un nœud t de T tel que Vars$(\Phi_1) \subseteq$ Vars(t^l) et Vars$(\Phi_2) \subseteq$ Vars(t^r). L'ensemble des formules* DNNF *(resp.* d-DNNF*) qui respecte un v-tree T est noté* DNNF$_T$ *(resp.* d-DNNF$_T$*). L'ensemble des formules appartenant à un langage* DNNF$_T$ *(resp.* d-DNNF$_T$*) est noté* SDNNF *(resp.* d-SDNNF*).*

La figure 3.5 (adaptée de [PIPATSRISAWAT & DARWICHE 2008]) présente deux exemples de v-tree, ainsi qu'une formule DNNF qui respecte un v-tree.

Il est intéressant de noter qu'étant donné un v-tree T, toute formule DNF peut être transformée en temps polynomial dans le produit des tailles de la formule et du v-tree en formule DNNF$_T$: il suffit pour cela de décomposer les nœuds conjonctifs de manière à représenter le v-tree [PIPATSRISAWAT & DARWICHE 2008]. De la même manière, une formule de MODS peut être réduite à une formule d-DNNF$_T$ en temps polynomial peu importe le v-tree T.

Le v-tree de la figure 3.5(c) possède une structure particulière (en peigne), et représente en fait un ordre total pour les variables ; on dit qu'un tel v-tree est linéaire.

Définition 3.16 (v-tree linéaire) *Un v-tree T est dit linéaire si et seulement si tout nœud de T qui n'est pas une feuille admet au moins un fils qui est une feuille.*

Un v-tree linéaire T représente en fait un ordre total $<$. Ceci implique que toute formule $\mathsf{OBDD}_<$ peut être réécrite en temps polynomial dans le produit des tailles de T et de la formule comme une formule $\mathsf{d\text{-}DNNF}_T$ [PIPATSRISAWAT & DARWICHE 2008].

Nous avons présenté dans cette section l'ensemble des langages considérés dans notre étude sur la complexité algorithmique des requêtes d'optimisation sous contraintes. Dans la section suivante, nous présentons les différentes propriétés qu'offre chacun de ces langages, ainsi qu'une étude de leur concision relative.

3.2 Choisir un langage de compilation

Dans cette section, nous expliquons les critères à prendre en compte pour le choix d'un langage de compilation. Nous ne nous intéressons pas ici aux algorithmes de compilation permettant de passer d'un langage à un autre, mais aux propriétés de ces différents langages : les requêtes et transformations de la formule permises (ou pas) en temps polynomial, ainsi que la concision de ces différents langages ; chaque requête est ainsi vue aussi comme une propriété offerte ou non par chacun des langages considérés. Les résultats de cette section sont majoritairement issus de [DARWICHE & MARQUIS 2002] et de [PIPATSRISAWAT & DARWICHE 2008]. Dans cette section, lorsque nous emploierons le terme langage, il s'agira d'un des langages présentés dans la section précédente.

3.2.1 Requêtes

Comme nous l'avons rappelé dans le chapitre introductif de cette partie, l'intérêt de la compilation réside dans la possibilité de rendre traitables certaines requêtes pour lesquelles il n'existe a priori pas d'algorithme polynomial pour les langages de représentation usuels tels que CNF.

Les deux premières requêtes considérées sont la cohérence et la validité d'une formule.

Définition 3.17 (CO, VA) *Un langage L satisfait* CO *(resp.* VA*) si et seulement si existe un algorithme en temps polynomial qui associe à chaque formule Φ de L la valeur* 1 *si elle est cohérente (resp. valide), et* 0 *sinon.*

En terme d'optimisation, la propriété CO est fondamentale, au sens où l'optimisation est au moins aussi difficile que la cohérence pour un langage donné. En effet, s'il est possible de retourner une solution optimale selon un ou plusieurs critère ou de déterminer qu'il n'y a pas de solution, alors il est aussi possible de décider de la cohérence de cette formule dans le même temps.

Définition 3.18 (CE) *Un langage L satisfait* CE *(Clausal Entailment) si et seulement si il existe un algorithme en temps polynomial qui associe à chaque couple (Φ, γ), où Φ est une formule de L et γ est une clause, la valeur* 1 *si $\Phi \models \gamma$ et* 0 *sinon.*

La propriété CE a notamment des applications dans le test d'équivalence de formules CNF. En disposant d'une forme compilée satisfaisant CE de chacune de ces formules, vérifier que la forme compilée de la première formule implique toutes les clauses de la deuxième, et *vice-versa* est suffisant pour prouver l'équivalence. En l'absence de formes compilées de chacune des deux formules CNF, il est toutefois plus efficace de tester l'équivalence *via* des structures *miter* [BRAND 2003, MARQUES-SILVA & GLASS 1999] que de passer par des phases de compilation.

On considère aussi la propriété duale, à savoir vérifier si un cube implique une formule.

Définition 3.19 (IM) *Un langage L satisfait* IM *(Implicant) si et seulement si il existe un algorithme en temps polynomial qui associe à chaque couple (Φ, γ), où Φ est une formule de L et γ est un cube, la valeur* 1 *si $\gamma \models \Phi$ et* 0 *sinon.*

On considère ensuite l'implication et l'équivalence de formules du même langage.

Définition 3.20 (EQ, SE) *Un langage L satisfait* EQ *(resp.* SE*, Sentential Entailment) si et seulement si il existe un algorithme en temps polynomial qui associe à chaque couple* (Φ_1, Φ_2) *de formules de L la valeur* 1 *si* $\Phi_1 \equiv \Phi_2$ *(resp.* $\Phi_1 \models \Phi_2$*) et* 0 *sinon.*

Il est à noter que SE est une propriété plus forte que CE (puisque une clause est une formule pouvant être transformée en temps polynomial en une formule équivalente appartenant à n'importe quel langage de notre étude) et EQ (puisque tester $\Phi_1 \models \Phi_2$ et $\Phi_2 \models \Phi_1$ permet de tester l'équivalence de ces deux formules). De ce fait, un langage L satisfaisant la propriété SE satisfait aussi CE et EQ.

On considère aussi le comptage de modèles, une requête utile par exemple pour l'inférence dans les réseaux bayesiens [LITTMAN *et al.* 2001, BACCHUS *et al.* 2003, SANG *et al.* 2005, DARWICHE 2009] et pour la résolution de problèmes de planification [KAUTZ *et al.* 1992, DOMSHLAK & HOFFMANN 2007]. Des travaux récents ont d'ailleurs mis en valeur l'utilité de la compilation pour le comptage de modèles [KORICHE *et al.* 2013].

Définition 3.21 (CT) *Un langage L satisfait* CT *(model CounTing) si et seulement si il existe un algorithme en temps polynomial qui associe à chaque formule Φ de L un nombre entier positif ou nul qui représente le nombre de modèles de Φ sur $Var(\Phi)$ (en supposant $PS = Var(\Phi)$).*

Enfin, on considère l'énumération de modèles. Bien qu'un langage satisfaisant cette propriété permette d'optimiser une fonction en un temps polynomial dans la taille de l'ensemble des modèles, cette propriété se révèle assez peu intéressante pour cet objectif, dans la mesure où l'ensemble des modèles d'une formule a souvent un cardinal exponentiel dans la taille de la formule.

Définition 3.22 (ME) *Un langage L satisfait* ME *(Model Enumeration) si et seulement si il existe un algorithme qui liste pour toute formule Φ de L, en temps polynomial dans la taille de Φ et dans le nombre de modèles de Φ, l'ensemble des modèles de Φ.*

Le tableau 3.1 indique parmi les langages de notre étude, lesquels satisfont ou ne satisfont pas les requêtes considérées dans cette section.

3.2.2 Transformations

Nous présentons maintenant un ensemble de transformations présentes dans l'état de l'art. Contrairement aux *requêtes*, dont le but est d'obtenir des informations sans altérer la formule, une *transformation* est un mécanisme dont le but est justement de modifier la formule pour refléter un changement dans les contraintes ou dans la modélisation du problème.

Dans le cadre des problèmes de configuration, il peut s'agir par exemple d'inscrire la sélection d'un module, ou bien d'ajouter un nouveau module. Or, la phase de compilation peut rendre difficile une transformation autorisée en temps polynomial par une formule CNF. C'est notamment le cas de l'ajout d'une nouvelle clause. Ainsi, il conviendra, si la formule peut changer régulièrement, de considérer un langage pour la compilation qui permet de telles modifications pour un coût maîtrisé.

Nous présentons tout d'abord la propriété de conditionnement d'une formule par un cube [DARWICHE 1999].

Définition 3.23 (Conditionnement d'une formule par un cube) *Soit Φ une formule propositionnelle et γ un cube cohérent. Le conditionnement de Φ par γ, noté $\Phi_{|\gamma}$ est la formule obtenue en remplaçant les occurrences du littéral x de Φ par $\top$ (resp. $\bot$) si x (resp. $\neg x$) apparaît dans γ.*

L	CO	VA	CE	IM	EQ	SE	CT	ME
NNF	○	○	○	○	○	○	○	○
s-NNF	○	○	○	○	○	○	○	○
f-NNF	○	○	○	○	○	○	○	○
CNF	○	✓	○	✓	○	○	○	○
PI	✓	✓	✓	✓	✓	✓	○	✓
DNNF	✓	○	✓	○	○	○	○	✓
DNF	✓	○	✓	○	○	○	○	✓
IP	✓	✓	✓	✓	✓	✓	○	✓
MODS	✓	✓	✓	✓	✓	✓	✓	✓
SDNNF	✓	○	✓	○	○	○	○	✓
DNNF$_T$	✓	○	✓	○	○	○	○	✓
d-DNNF	✓	✓	✓	✓	?	○	✓	✓
d-SDNNF	✓	✓	✓	✓	?	○	✓	✓
d-DNNF$_T$	✓	✓	✓	✓	✓	✓	✓	✓
d-NNF	○	○	○	○	○	○	○	○
BDD	○	○	○	○	○	○	○	○
FBDD	✓	✓	✓	✓	?	○	✓	✓
OBDD	✓	✓	✓	✓	✓	○	✓	✓
OBDD$_<$	✓	✓	✓	✓	✓	✓	✓	✓

TABLE 3.1 – Complexité des requêtes pour les langages considérés. ✓ signifie « satisfait », ○ signifie « ne satisfait pas si P≠NP », ? signifie « problème ouvert ». Ce tableau est basé sur celui de [DARWICHE & MARQUIS 2002]. Notons qu'il existe un algorithme probabiliste en temps polynomial pour EQ à partir de formules d-DNNF, ce qui est bien souvent suffisant en pratique.

Le conditionnement correspond à la notion de *restriction* dans la littérature sur les fonctions booléennes. La propriété associée au conditionnement est définie comme suit.

Définition 3.24 (CD) *Un langage L satisfait* CD *(Conditioning) si et seulement si il existe un algorithme en temps polynomial qui associe à chaque couple (Φ, γ), où Φ est une formule de L et γ est un cube, une formule de L qui est logiquement équivalente à $\Phi_{|\gamma}$.*

Le conditionnement étant assez simple à réaliser en général, il est traité en temps polynomial par l'ensemble des langages de notre étude. Cependant, il est d'une grande aide dans la réalisation d'un compilateur vers un langage satisfaisant la propriété de décomposabilité, puisqu'il permet l'utilisation de la décomposition de Shannon.

L'oubli, aussi appelé *marginalisation* ou *élimination des termes médians* a des utilités dans divers domaines, comme la planification ou la révision de croyances [LANG & MARQUIS 2010].

Définition 3.25 (Oubli) *Soit Φ une formule propositionnelle et X un sous-ensemble de variables de* PS*. L'oubli de X dans Φ, noté $\exists X.\Phi$, est une formule qui ne mentionne aucune variable de X et telle que pour toute formule Ψ qui ne mentionne aucune variable de X, on a $\Phi \models \Psi$ précisément quand $\exists X.\Phi \models \Psi$.*

Les propriétés associées à l'oubli sont les suivantes.

Définition 3.26 (FO, SFO) *Un langage L satisfait* FO *(Forgetting) si et seulement si il existe un algorithme en temps polynomial qui associe à tout couple (Φ, X), où Φ est une formule de L et X est un sous-ensemble de* PS*, une formule de L équivalente $\exists X.\Phi$. Si la propriété est valable pour un singleton X, L satisfait* SFO *(Singleton Forgetting).*

La conjonction et la disjonction permettent d'ajouter des contraintes ou des alternatives à une formule. Elles permettent aussi la construction de formules par un compilateur selon une approche dite ascendante (*bottom-up*).

Définition 3.27 ($\wedge$C, $\vee$C) *Un langage L satisfait* $\wedge$C *(resp.* $\vee$C*) si et seulement si il existe un algorithme en temps polynomial qui associe à tout ensemble fini de formules $\Phi_1, \ldots, \Phi_n$ de L une formule de L logiquement équivalente à $\Phi_1 \wedge \ldots \wedge \Phi_n$ (resp. $\Phi_1 \vee \ldots \vee \Phi_n$).*

Définition 3.28 ($\wedge$BC, $\vee$BC) *Un langage L satisfait* $\wedge$BC *(resp.* $\vee$BC*) si et seulement si il existe un algorithme en temps polynomial qui associe à tout couple de formules (Φ_1, Φ_2) de L une formule de L logiquement équivalente à $\Phi_1 \wedge \Phi_2$ (resp. $\Phi_1 \vee \Phi_2$).*

Finalement, on considère la négation d'une formule.

Définition 3.29 ($\neg$C) *Un langage L satisfait* $\neg$C *si et seulement si il existe un algorithme en temps polynomial qui associe à chaque formule Φ de L une formule de L équivalente à $\neg\Phi$.*

Le tableau 3.2 indique parmi les langages de notre étude, lesquels admettent un algorithme en temps polynomial pour chacune des transformations que nous avons considérées.

L	CD	FO	SFO	$\wedge$C	$\wedge$BC	$\vee$C	$\vee$BC	$\neg$C
NNF	✓	○	✓	✓	✓	✓	✓	✓
s-NNF	✓	○	✓	✓	✓	✓	✓	✓
f-NNF	✓	○	✓	●	●	●	●	✓
CNF	✓	○	✓	✓	✓	●	✓	●
PI	✓	✓	✓	●	●	●	✓	●
DNNF	✓	✓	✓	○	○	✓	✓	●
DNF	✓	✓	✓	●	✓	✓	✓	●
IP	✓	●	●	●	✓	●	●	●
MODS	✓	✓	✓	●	✓	●	●	●
SDNNF	✓	✓	✓	○	○	✓	✓	○
DNNF$_T$	✓	✓	✓	○	✓	✓	✓	○
d-DNNF	✓	○	○	○	○	○	○	?
d-SDNNF	✓	●	?	●	○	●	○	?
d-DNNF$_T$	✓	●	?	●	✓	●	?	?
d-NNF	✓	○	✓	✓	✓	✓	✓	✓
BDD	✓	○	✓	✓	✓	✓	✓	✓
FBDD	✓	●	○	●	○	●	○	✓
OBDD	✓	●	✓	●	○	●	○	✓
OBDD$_<$	✓	●	✓	●	✓	●	✓	✓

TABLE 3.2 – Complexité des transformations pour les langages considérés. ✓ signifie « satisfait », ○ signifie « ne satisfait pas si P$\neq$NP », ● signifie « ne satisfait pas », ? signifie « problème ouvert ». Ce tableau est basé sur celui de [DARWICHE & MARQUIS 2002] et inclut des résultats récents de [BOVA *et al.* 2014].

3.2.3 Concision des langages étudiés

Un autre point à prendre un considération lors du choix d'un langage de compilation est la concision du langage. En effet, une formule compilée peut avoir une taille exponentielle dans la taille de la formule initiale. De ce fait, il peut être judicieux, après avoir déterminé quelles requêtes et quelles transformations sont souhaitées pour le langage cible, de choisir le langage le plus succinct possible.

La notion de concision employée dans [DARWICHE & MARQUIS 2002] s'appuie sur la taille des formules, en considérant qu'un langage est au moins aussi succinct qu'un autre si le premier permet de représenter toutes les formules du deuxième sans que la taille n'explose.

Définition 3.30 (Concision) *Soient L_1 et L_2 deux langages. On dit que L_1 est au moins aussi succinct que L_2, qu'on note $L_1 \leq L_2$, s'il existe un polynôme p tel que pour toute formule Ψ de L_2, il existe une formule équivalente Φ de L_1 telle que $|\Phi| \leq p(|\Psi|)$.*

On ne considérera ici que les langages pour lesquels la propriété CO (test de cohérence en temps polynomial) est satisfaite, puisque non seulement ces langages sont les seuls à être considérés comme étant de « bons » langages cibles, mais aussi parce que ce sont les seuls pour lesquels on puisse potentiellement avoir des algorithmes en temps polynomial pour l'optimisation.

Sans surprise, on se rend compte que plus un langage offre de bonnes propriétés, et plus il est sujet à être gourmand en espace. Si on considère les langages descendants des diagrammes de décision binaires, on remarque ce comportement.

$$\mathsf{NNF} < \mathsf{DNNF} < \mathsf{d\text{-}DNNF} < \mathsf{FBDD} < \mathsf{OBDD} < \mathsf{OBDD}_T < \mathsf{MODS}$$

En revanche, en ce qui concerne le langage FBDD, il n'apparaît pas comme un bon choix de langage cible si on considère uniquement les propriétés assimilées aux requêtes, puisque celui-ci est strictement moins succinct que d-DNNF, qui possède les mêmes propriétés.

Cette différence de concision se retrouve aussi entre les diagrammes de décision binaires et les formules décomposables structurées,

$$\mathsf{SDNNF} < \mathsf{OBDD}\ ;\ \mathsf{DNNF}_T < \mathsf{OBDD}_T$$

tout comme pour les formules DNNF et DNF.

$$\mathsf{NNF} < \mathsf{DNNF} < \mathsf{DNF} < \mathsf{IP} < \mathsf{MODS}$$

Pour les requêtes, choisir DNF plutôt que DNNF est donc un mauvais choix, dans la mesure où ces deux langages offrent précisément les mêmes requêtes. En revanche, l'avantage apporté par DNF quant aux transformations peut être un facteur déterminant pour le choix du langage de compilation.

3.3 Conclusion du chapitre

Dans ce chapitre, nous avons décrit un certain nombre de langages de compilation de l'état de l'art, en considérant des sous-langages de NNF. Nous avons rappelé les requêtes et transformations usuelles offertes ou pas par ces langages, ainsi que les résultats théoriques quant à leur concision relative.

Nous nous intéressons tout particulièrement aux langages satisfaisant la requête CO (ceux pour lesquels l'optimisation pourrait être réalisée en temps polynomial), c'est-à-dire ceux pour lesquels le test de cohérence est en temps polynomial. Les langages satisfaisant cette propriété sont DNNF et ses sous-langages, ainsi que PI. Il est à noter que dans le cas où les contraintes d'un problème peuvent évoluer, il est judicieux de considérer les langages, s'il en existe, qui satisfont les propriétés liées aux

transformations qui permettent de faire évoluer la forme compilée en temps polynomial. Parmi les transformations qui pourraient être utiles à cette fin, on peut notamment penser à la conjonction bornée, qui peut par exemple permettre d'ajouter un nombre fixé de contraintes en temps polynomial ; les langages satisfaisant à la fois CO et $\wedge$BC sont DNF et les DNNF structurées (SDNNF). Il faut toutefois noter que ces langages sont moins succincts que la plupart des autres langages que nous avons considérés dans notre étude.

Dans le chapitre suivant, nous étudions la complexité de la requête d'optimisation pour des contraintes compilées dans différents langages de ce chapitre, cela pour un large spectre de fonctions objectifs.

Chapitre 4

Optimisation sous contraintes NNF

Sommaire

4.1 La requête OPT 62

4.2 OPT dans le cadre de la somme des coûts et du leximax 64

4.2.1 Fonctions objectifs linéaires 65

4.2.2 Fonctions objectifs générales 70

4.2.3 Résultats de *fixed-parameter tractability* 72

4.3 Raffinement de la frontière entre optimisation simple et complexe 74

4.3.1 Agrégation via opérateurs OWA 74

4.3.2 Extensions des fonctions linéaires 77

4.3.3 Fonctions sous-modulaires 78

4.4 Conclusion du chapitre 79

Les problèmes d'optimisation sous contraintes étant généralement des problèmes combinatoires, ils sont bien souvent NP-difficiles. Cependant, comme nous l'avons noté en préambule du chapitre précédent, certains problèmes d'optimisation sont définis par un ensemble de contraintes qui n'évolue pas ou peu, au contraire de la fonction objectif qui diffère d'un utilisateur à l'autre. Dans ce cas, compiler au prix d'une complexité élevée l'ensemble des contraintes dans un langage propositionnel permettant ensuite de procéder à des requêtes d'optimisation en temps polynomial dans la taille d'une fonction objectif non connue à l'avance peut être intéressant ; cela permet à la fois de meilleures performances lors de phases d'interaction avec un utilisateur, mais aussi d'économiser du temps de calcul une fois le temps de compilation amorti par les gains obtenus lors des phases d'optimisation.

Exemple 4.1 (Compilation de problème de gestion de dépendances logicielles) *On considère un ensemble de n problèmes de gestion de dépendances logicielles portant sur les mêmes paquets : un ensemble de n utilisateurs souhaite installer des paquets qu'ils ne possèdent pas, en retirant un minimum de paquets de leur machine.*

Si les relations entre les différents paquets sont encodées par une formule CNF*, le problème d'optimisation considéré est* NP*-difficile, et le temps de calcul correspondant aux requêtes des n utilisateurs correspondra donc au temps nécessaire au traitement de n problèmes* NP*-difficiles.*

En revanche, si les contraintes concernant les relations entre paquets sont compilées dans un langage permettant à la fois, pour un utilisateur, d'y indiquer en temps polynomial les paquets installés sur sa machine (on pense ici à la propriété de conditionnement) et d'y exécuter en temps polynomial la recherche d'une solution pour son problème d'optimisation (la fonction objectif dépend de l'état initial de l'utilisateur), alors la compilation permet d'économiser du temps de calcul si le nombre d'utilisateurs est suffisamment important (tout en permettant aux utilisateurs d'obtenir une solution optimale à leur problème de manière plus rapide).

Dans ce chapitre, nous présentons une étude de la requête d'optimisation sur les formules NNF. Plus précisément, nous étudions la complexité des processus d'optimisation sur les sous-langages de NNF présentés dans le chapitre précédent pour différentes familles de fonctions objectifs $\mathcal{F}$. Nous nous intéressons tout particulièrement au langage DNNF pour la compilation des contraintes, qui permet d'optimiser en temps polynomial un certain nombre de fonctions objectifs et pour lequel il existe des compilateurs capables de traduire des problèmes réels représentés par une formule CNF en formule DNNF, tels que `c2d`[1] [DARWICHE 2002] et `Dsharp`[2] [MUISE *et al.* 2012].

Nous définissons une nouvelle requête dans l'esprit de celles présentées dans le chapitre précédent, OPT, pour traiter de l'optimisation. Contrairement aux requêtes que nous avons présentées précédemment, OPT est une requête paramétrée par une famille $\mathcal{F}$ de fonctions objectifs.

Notons que l'approche que nous considérons ici diffère de celles étudiées auparavant pour l'optimisation sous contraintes compilées (citons par exemple [BAHAR *et al.* 1993, WILSON 2005, SANNER & MCALLESTER 2005, FARGIER & MARQUIS 2007, KADIOGLU & SELLMANN 2008, KATSIRELOS *et al.* 2011]) dans la mesure où l'objectif de ces travaux étaient de compiler à la fois les contraintes du problème et la fonction objectif, alors que dans notre cas, les fonctions objectifs ne sont pas connues au moment de la compilation des contraintes.

4.1 La requête OPT

Comme nous allons le vérifier dans la suite de ce chapitre, la complexité d'un problème d'optimisation dépend du langage considéré pour la représentation des contraintes. Cette complexité dépend aussi

1. `http://reasoning.cs.ucla.edu/c2d/`
2. `http://www.haz.ca/research/dsharp/`

bien évidemment de la fonction objectif considérée.

Pour rappel, nous avons défini les fonctions objectifs pseudo-booléennes à la section 2.1.2 en partant de la notion de base propositionnelle pondérée, décrite comme un ensemble $\phi = \{(\phi_i, c_i) \mid i = 1, \ldots, n\}$ où les ϕ_i sont des formules propositionnelles et les c_i sont des nombres réels exprimant des coûts. Pour un modèle, on obtient de cette base un vecteur composé de n réels, où chaque élément correspond à un couple (ϕ_i, c_i) de la base. La valeur de cet élément est c_i si le modèle est cohérent avec la formule ϕ_i ; cette valeur est 0 sinon (ce qui correspond à la fonction Val définie à la page 29).

Nous considérons ici que toute formule de la base est un critère binaire, et que la méthode « agréger puis comparer » est employée pour le processus d'optimisation. Il faut donc agréger les valeurs du vecteur issu de la base pondérée et du modèle considérés de manière à dégager un coût global.

Ainsi, nos fonctions objectifs sont composées :

- d'une base pondérée $\phi = \{(\phi_i, c_i) \mid i \in 1, \ldots, n\}$, qui nous donne les différents coûts induits par une solution ω *via* la fonction Val ;
- d'une relation de préférence $\succeq$ permettant de comparer des vecteurs de n coûts.

Les solutions optimales d'un problème seront alors les solutions pour lesquelles la valeur issue de l'agrégation des coûts du vecteur ne sera dominée par aucune autre valeur correspondant à une solution. Ce formalisme est suffisamment expressif pour permettre la représentation de toute fonction pseudo-booléenne (*i.e.* de $\mathbb{B}^m$ dans $\mathbb{R}$). Par exemple, en considérant la somme comme fonction d'agrégation, si la base pondérée porte sur des formules composées uniquement d'un littéral, on se retrouve dans le cadre de l'optimisation sous fonction pseudo-booléenne linéaire ; s'il s'agit de cubes de taille deux au plus, il s'agit d'optimisation sous fonction pseudo-booléenne quadratique ; si les ϕ_i sont des formules NNF, on peut représenter des fonctions pseudo-booléennes quelconques.

Pour rappel, nous considérons un certain nombre de *familles de représentation* en fonction du langage utilisé pour représenter les ϕ_i de la base pondérée :

- la famille $\mathcal{L}$ des *représentations linéaires* des bases pondérées dont les ϕ_i sont des littéraux ;
- la famille $\mathcal{Q}$ des *représentations quadratiques* des bases pondérées dont les ϕ_i sont des cubes de taille 2 au plus ;
- la famille $\mathcal{P}$ des *représentations polynomiales* des bases pondérées dont les ϕ_i sont des cubes de longueur quelconque ;
- et finalement, la famille $\mathcal{G}$ des *représentations pseudo-booléennes générales* des bases pondérées dont les ϕ_i sont des formules NNF.

Notons que les représentations linéaires ne permettent pas de représenter l'ensemble des fonctions pseudo-booléennes, mais seulement celles qui sont additives. Nous allons aussi considérer des sous-familles en fonction du signe des littéraux et des poids utilisés dans les bases pondérées. Étant donné une famille $\mathcal{F}$, $\mathcal{F}^+$ désignera les bases pondérées de $\mathcal{F}$ dont les littéraux sont tous positifs ; $\mathcal{F}_+$ désignera les bases pondérées de $\mathcal{F}$ dont les poids sont tous positifs ou nuls. Enfin, on considérera aussi les sous-familles pour lesquelles ces deux conditions sont respectées : $\mathcal{F}^+_+$.

Exemple 4.2 (Famille de bases pondérées) *Les bases pondérées suivantes sont considérées dans notre étude :*

- $\{(x_1, 1), (\neg x_2, -4)\} \in \mathcal{L}$ *;*
- $\{(x_1, 1), (\neg x_2 \wedge x_3, 4)\} \in \mathcal{Q}_+$ *;*
- $\{(x_1, 1), (x_2 \wedge x_3 \wedge x_4, -4)\} \in \mathcal{P}^+$ *;*
- $\{(x_1, 1), (x_2 \wedge (x_3 \vee (\neg x_3 \wedge x_4)), 3)\} \in \mathcal{G}$.

Dans [ROSENBERG 1975], l'auteur montre que toute fonction polynomiale pseudo-booléenne basée sur une somme peut être réécrite comme une fonction quadratique en temps polynomial sans changer l'ensemble des contraintes, mais seulement en ajoutant de nouvelles variables. Cette transformation ne

conserve cependant pas le signe des littéraux ; or, étant donné que nous considérons des sous-familles dont des conditions portent sur la polarité des littéraux de PS, nous devons dissocier les familles $\mathcal{P}$ et $\mathcal{Q}$.

Exemple 4.3 (Réduction d'une fonction de $\mathcal{P}$ vers $\mathcal{Q}$) *Soit la fonction pseudo-booléenne polynomiale définie par :*

$$f(x_1, x_2, x_3, x_4) = x_1x_2 + 2x_1x_2x_3 + 3x_2x_3x_4.$$

Cette fonction est équivalente pour l'optimisation à la fonction quadratique f' définie ci-dessous quand la nouvelle variable booléenne x_5 est libre :

$$f'(x_1, x_2, x_3, x_4, x_5) = 8x_1x_2 - 14x_1x_5 - 14x_2x_5 + 21x_5 + 2x_3x_5 + 3x_4x_5.$$

Cette réduction depuis les représentations polynomiales montre que la famille des représentations quadratiques est suffisante pour représenter n'importe quelle fonction pseudo-booléenne. En effet, la famille des représentations polynomiales permet de représenter une fonction pseudo-booléenne comme une somme d'interprétations (exprimées par des cubes) pondérées par les valeurs que la fonction doit donner pour ces interprétations.

Étant donné un langage propositionnel L (NNF ou un de ses sous-langages), un langage de représentation de bases pondérées $\mathcal{F}$ et une relation de préférence $\succeq$, on peut définir la requête OPT pour l'optimisation sous formules de L étant donné $\mathcal{F}$ et $\succeq$.

Définition 4.1 (Requête OPT) *Soient L un langage, $\mathcal{F}$ un langage de représentation de fonctions objectifs pseudo-booléennes, et $\succeq$ une relation de préférence. L satisfait la propriété* OPT *pour $\mathcal{F}$ et $\succeq$ si il existe un algorithme qui associe pour toute formule Φ de L et pour toute base pondérée ϕ de $\mathcal{F}$ une solution optimale de Φ pour $\succeq$ si Φ est cohérente, et « pas de solution » sinon, ceci en temps polynomial dans la taille de Φ et de ϕ. On dit alors que L satisfait* OPT$[\succeq, \mathcal{F}]$.

Bien entendu, il est évident qu'un langage ne permettant pas de déterminer la cohérence d'une formule en temps polynomial ne sera pas une cible intéressante pour les problématiques d'optimisation, puisque le test de cohérence est lui-même « inclus » dans la recherche d'un modèle optimal.

Proposition 4.2 *Soient L un langage, $\mathcal{F}$ un langage de représentation de fonctions objectifs pseudo-booléennes, et $\succeq$ une relation de préférence. Si L ne satisfait pas* CO*, L ne satisfait pas* OPT$[\succeq, \mathcal{F}]$.

Preuve *Si L satisfait* OPT*$[\succeq, \mathcal{F}]$, alors L retourne une solution en temps polynomial s'il en existe une « pas de solution » sinon, et donc L satisfait* CO.

De ce fait, nous allons restreindre notre étude aux langages DNNF ainsi qu'à ses sous-langages, en plus du langage PI qui permet lui-aussi de tester la cohérence d'une formule en temps polynomial (voir la figure 4.1).

Nous démontrons parmi les langages L de NNF et les langages de représentation de fonctions objectifs $\mathcal{L}$, $\mathcal{Q}$, $\mathcal{P}$ et $\mathcal{G}$, pour quels couples $(L, \mathcal{F})$ la requête OPT est satisfaite, pour plusieurs relations de préférence usuelles $\succeq$.

4.2 OPT dans le cadre de la somme des coûts et du leximax

Nous considérons dans un premier temps les deux relations de préférence suivantes :

- $\succeq_\Sigma$, qui étant donné deux interprétations, donne la préférence à celle dont la somme des coûts induits par les bases pondérées est inférieure à la somme des coûts induits par la deuxième interprétation ;

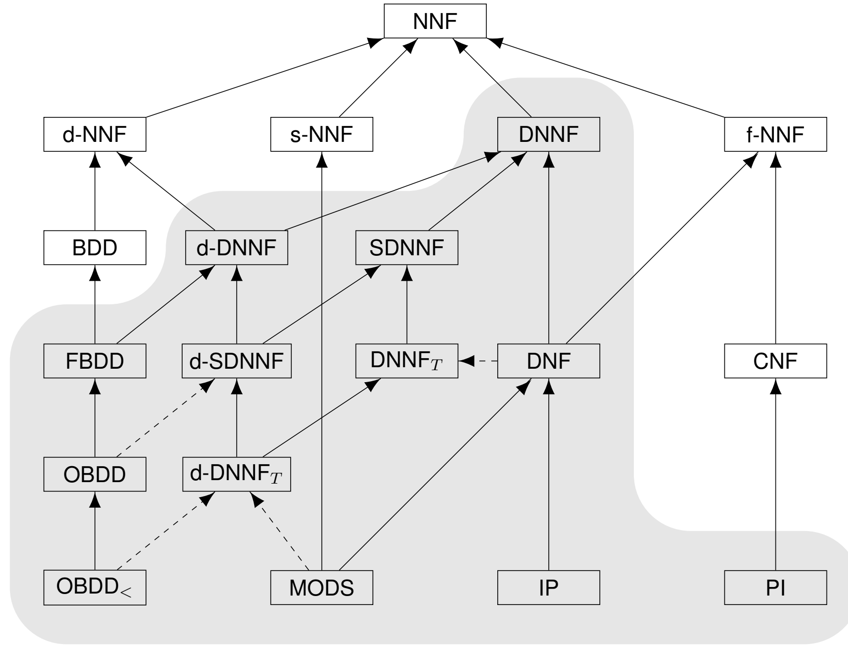


FIGURE 4.1 – Le langage NNF et ses sous-langages. Une flèche pleine $A \to B$ signifie que le langage A est un sous-langage de B. Une flèche en pointillés signifie qu'il existe une traduction en temps polynomial du langage origine vers le langage extrémité qui respecte l'équivalence. Les langages dans la zone grise sont ceux qui satisfont la propriété CO. Tableau basé sur celui de [DARWICHE & MARQUIS 2002]

– $\succeq_{\text{LEX}}$, qui étant donné deux interprétations, donne la préférence à celle dont le vecteur de coûts unitaires induits par les bases pondérées domine le vecteur de coût de la deuxième interprétation selon le leximax.

En ce qui concerne les langages de représentation de fonctions objectifs, nous débutons notre étude par la famille la plus spécifique que nous considérons, à savoir la famille $\mathcal{L}$ des représentations linéaires.

4.2.1 Fonctions objectifs linéaires

Nous avons restreint notre étude aux langages permettant de tester la cohérence d'une formule en temps polynomial, à savoir les langages DNNF et PI. Cependant, une réduction depuis le problème *minimal hitting set* implique des résultats négatifs en ce qui concerne le langage PI.

Proposition 4.3 PI *ne satisfait ni* OPT$[\succeq_\Sigma, \mathcal{L}_+^+]$, *ni* OPT$[\succeq_{\text{LEX}}, \mathcal{L}_+^+]$, *sauf si* P=NP.

Preuve *Toute formule Krom positive* Φ (i.e. *une formule* CNF *composée uniquement de clauses binaires dans lesquelles les littéraux sont tous positifs) peut être écrite en temps polynomial comme une formule* PI *positive. La* NP*-difficulté de l'optimisation dans ce cas pour la somme et le leximax vient du problème* minimal hitting set *(ensemble intersectant minimal), où le cardinal des ensembles est de taille 2. Soit C un ensemble d'ensembles c_i (où $|c_i| = 2$) dont les éléments sont inclus dans un ensemble de référence E. Un* ensemble intersectant *de C est un ensemble h d'éléments de E tel que $E \cap c_i \neq \varnothing$. Le problème de déterminer s'il existe un ensemble intersectant h de C contenant au plus k éléments a été démontré* NP*-complet [*KARP *1972]. Clairement, un tel ensemble intersectant existe si et seulement si une solution optimale ω^* du problème d'optimisation défini par la contrainte dure $\phi = \wedge_{c_i \in C}(\vee_{x \in c_i} x)$ (une formule Krom positive) et la base pondérée $\{(x, 1) \mid x \in \mathsf{PS}\}$ (qui appartient à $\mathcal{L}_+^+$), telle que $f(\omega^*) \leq k$ quand $\succeq = \succeq_\Sigma$, et $f(\omega^*)$ contient au plus k fois le coût 1 quand $\succeq = \succeq_{\text{LEX}}$.*

Ce résultat est négatif bien que les hypothèses sur la famille de représentation soient les plus restrictives parmi celles que nous considérons, ce qui implique que le résultat est aussi négatif pour toutes les autres familles de fonctions objectifs de notre étude en ce qui concerne PI. De ce fait, nous considérons dans la suite uniquement le langage DNNF et ses sous-langages pour la compilation des contraintes ; langages pour lesquels nous verrons qu'il existe un certain nombre de familles de fonctions objectifs pouvant être optimisées en temps polynomial dans la taille de l'entrée. Afin de démontrer ces résultats, nous devons dans un premier temps définir des notions qui nous seront utiles à la description d'un algorithme capable de retourner une solution optimale lorsque les contraintes sont exprimées par une formule DNNF et une fonction objectif représentée par une base $\mathcal{L}$ agrégée par la somme ou le leximax. Nous introduisons tout d'abord la notion *d'interprétation partielle*.

Définition 4.4 (Interprétation partielle) *Soit $V \subseteq \mathsf{PS}$ un ensemble de variables propositionnelles. Une interprétation partielle ω_V est un ensemble de couples $\{(v, \{0,1\}) \mid v \in V\}$ tel que chaque $v \in V$ apparaît dans un et un seul couple. Une interprétation partielle peut aussi être vue comme le cube $(\bigwedge_{(v,0)\in\omega_V} \neg v) \wedge (\bigwedge_{(v,1)\in\omega_V} v)$.*

Le terme « partielle » vient du fait qu'en ajoutant à une telle interprétation des couples $\{(v, \{0,1\}) \mid v \in \mathsf{PS} \setminus V\}$, on obtient une interprétation « complète ». Dans ce cas, on dit que l'interprétation créée à partir de l'interprétation partielle ω_V est une *extension* de ω_V. Si toutes les extensions de ω_V sont des modèles, l'interprétation partielle ω_V est alors un *générateur de modèles*.

Définition 4.5 (Générateur de modèles) *Soit ω_V une interprétation partielle telle que l'ensemble des extensions de ω_V soit un sous-ensemble des modèles d'une formule Φ. On dit que ω_V est un générateur de modèles de Φ.*

En nous appuyant sur ces notions, nous allons maintenant décrire un algorithme capable de retourner en temps polynomial une solution optimale pour une fonction pseudo-booléenne linéaire, pour $\succeq = \succeq_\Sigma$ et $\succeq = \succeq_{\mathrm{LEX}}$, quand la formule est sous forme DNNF(et qu'une solution existe). Cet algorithme est basé sur plusieurs lemmes ; le premier indique qu'étant donné un générateur de modèles dont au moins une extension est une solution optimale pour une fonction $f \in \mathcal{L}$, on peut obtenir en temps polynomial une solution optimale pour f.

Lemme 4.6 *Soit f une fonction de $\mathcal{L}$. Soit ω_V un générateur de modèles pour une formule NNF Φ dont au moins une extension est une solution optimale de f. Étant donné une fonction d'optimisation composée de f et de la relation de préférence $\succeq_\Sigma$ ou $\succeq_{\mathrm{LEX}}$, on peut générer en temps polynomial une solution optimale de f.*

Preuve *Étant donné un générateur de modèles ω_V de Φ dont au moins une extension est une solution optimale de f et une base pondérée linéaire $\{(l_i, w_i) \mid i \in 1, \dots, n\}$, il est facile de calculer un modèle optimal qui est une extension de ω_V, que l'agrégateur soit $\succeq_\Sigma$ ou $\succeq_{\mathrm{LEX}}$. Pour chaque littéral l_i dont la variable v_i n'est pas assignée dans ω_V,*

- *si $w_i < 0$, on ajoute $(v_i, 0)$ à ω_V si l_i est un littéral négatif, ou $(v_i, 1)$ si l_i est un littéral positif ;*
- *si $w_i \geq 0$, on ajoute $(v_i, 0)$ à ω_V si l_i est un littéral positif, ou $(v_i, 1)$ si l_i est un littéral négatif ;*

Ceci permet de minimiser chacun des poids unitaires, ce qui permet de minimiser la valeur de la fonction objectif puisque la somme et le leximax sont monotones non décroissants. On ajoute ensuite un couple dans ω_V pour chacune des variables qui n'y apparaît pas encore de manière à obtenir une interprétation complète.

Exemple 4.4 (Extension optimale d'un générateur de modèle) *Soit le problème de minimiser l'agrégation par la somme (ou le leximax) de la base $\{(a,1),(b,1),(c,1),(d,-1)\} \in \mathcal{L}$ étant donné le générateur de modèles $\omega_V = \{(a,1),(b,1),(e,1)\}$ dont au moins une extension est optimale et l'ensemble*

de variables du problème est $\{a, b, c, d, e, f\}$. En ce qui concerne les littéraux c et d qui apparaissent dans la fonction objectif, on dispose du choix de leurs valeurs de vérité, puisqu'elles ne font pas partie du générateur de modèles. Étant donné que la somme (ou le leximax) est un opérateur monotone non décroissant, il s'agit, pour minimiser la fonction objectif, de choisir les valeur de vérité minimisant les coûts unitaires ; c'est-à-dire d'ajouter les couples $(c, 0)$ et $(d, 1)$ au générateur de modèle pour obtenir une interprétation partielle dont toute extension est un modèle minimal pour la fonction objectif. Finalement, pour étendre cette interprétation partielle en une interprétation complète, il suffit d'ajouter un couple (f, w), avec $w = 0$ ou $w = 1$.

L'idée de l'algorithme d'optimisation est de parcourir une formule dans l'ordre topologique inverse, et de démontrer que pour tout nœud interne de la formule, on peut déterminer une interprétation partielle qui peut être étendue en un modèle optimal de la sous-formule. Une fois remonté à la racine, la formule complète sera alors considérée, et un optimum pour la formule pourra donc être déterminé en temps polynomial, comme l'assure le lemme précédent.

Étant donné que nous allons considérer la formule dans l'ordre topologique inverse, nous devons nous intéresser en premier lieu aux feuilles de la formule.

Lemme 4.7 *Soit Φ une formule* NNF *telle que $|\Phi| = 1$. On peut, en temps polynomial, calculer un générateur de modèles dont l'ensemble des extensions est égal à l'ensemble des modèles de Φ quand il en existe, ou déterminer que la formule est incohérente.*

Preuve *Il existe quatre types de formules* NNF *Φ telles que $|\Phi| = 1$:*
- *si $\Phi = \top$, les modèles de Φ sont les extensions du générateur $\{\}$;*
- *si $\Phi = \bot$, Φ n'admet pas de modèle, et donc pas de générateur ;*
- *si $\Phi = x$ ($x \in PS$), les modèles de Φ sont les extensions du générateur $\{(x, 1)\}$;*
- *enfin, si $\Phi = \neg x$ ($x \in PS$), les modèles de Φ sont les extensions du générateur $\{(x, 0)\}$.*

Nous nous intéressons maintenant aux nœuds $\wedge$. Dans la mesure où nous considérons DNNF et ses sous-langages, ces nœuds satisfont la propriété de décomposabilité, ce qui permet d'assurer que les modèles pour une sous-formule dont la racine est un nœud $\wedge$ sont les unions des modèles des fils de la racine.

Lemme 4.8 *Soit $\Phi = \wedge_{i=1}^{k} \Phi_i$ une formule dont la racine est un nœud $\wedge$ décomposable. Supposons, pour chaque fils Φ_i de Φ, que l'on connaisse un générateur de modèles dont au moins une extension est un modèle optimal pour une fonction f composée d'une base $\phi \in \mathcal{F}$ et d'un agrégateur monotone, ou que l'on sache que ce nœud est la racine d'une formule incohérente. On peut alors calculer en temps polynomial dans le nombre de ses fils un générateur de modèles correspondant à Φ dont au moins une extension est un modèle optimal de f.*

Preuve *Si un des fils Φ_i est incohérent, alors Φ est aussi incohérent, et n'admet donc pas de générateur de modèles. Dans le cas contraire, le fait que Φ ait pour racine un nœud $\wedge$ décomposable assure que l'union des générateurs de modèles des nœuds fils soit un générateur de modèles pour Φ. De plus, étant donné que f est monotone, les modèles optimaux de Φ sont les modèles qui minimisent la valeur de f pour les variables de Φ_1, de Φ_2, etc... De ce fait, le générateur de modèles obtenu par l'union de ceux des Φ_i contient aussi un modèle optimal de f sous Φ.*

Exemple 4.5 (Union de générateurs de modèles) *Soit $\Phi = \Phi_1 \wedge \Phi_2$ avec $\Phi_1 = a \vee \neg b$ et $\Phi_2 = c \vee (d \wedge \neg e)$ une formule* DNNF. *Soit $\{(a, 1), (b, 1), (d, 1)\} \in \mathcal{L}$ une base pondérée et f la fonction correspondant à l'agrégation par la somme (ou le leximax) des coûts de cette base. $\{(b, 0)\}$ est un*

générateur de modèles pour Φ_1 et $\{(c,1)\}$ est un générateur de modèles pour Φ_2. Pour chacun de ces générateurs de modèles, au moins une de leurs extensions est un modèle minimal pour f.

Φ peut être vue comme l'union de deux problèmes distincts, dans la mesure où Φ_1 et Φ_2 ne partagent aucune variable et que l'on peut considérer que la fonction objectif peut être séparée en deux en suivant la même décomposition sur les variables. De ce fait, les solutions optimales pour Φ sont celles qui sont des unions de solutions optimales de Φ_1 et de Φ_2, ce qui est en particulier vrai pour celles qui sont des extensions des générateurs de modèles $\{(b,0)\}$ et $\{(c,1)\}$. Cela assure que $\{(b,0),(c,1)\}$ est un générateur de modèles pour Φ dont au moins une extension est un modèle optimal de f.

Enfin, le dernier type de nœuds à considérer pour que notre algorithme puisse traiter toute formule DNNF est le nœud $\vee$.

Lemme 4.9 *Soit $\Phi = \vee_{i=1}^{k}\Phi_i$ une formule dont la racine est un nœud $\vee$. Supposons, pour chaque fils Φ_i de Φ, que l'on connaisse un générateur de modèles dont une extension est un modèle optimal pour $f \in \mathcal{F}$ ou que l'on sache que ce nœud est la racine d'une formule incohérente. On peut alors sélectionner parmi les générateurs de modèles des fils de Φ une interprétation partielle de Φ dont les extensions sont des modèles de Φ et qui contient au moins une solution optimale pour une fonction objectif composée d'une base de $\mathcal{L}$ et de la relation de préférence $\succeq_\Sigma$ ou $\succeq_{\mathrm{LEX}}$.*

Preuve *L'incohérence de Φ peut être facilement déterminée étant donné qu'elle correspond au fait que chacun des Φ_i est lui-même incohérent, c'est-à-dire qu'il n'admette pas de générateur de modèles. Si Φ est cohérent, alors l'ensemble de ses modèles est égal à l'union des modèles des fils de la racine de Φ, puisque la racine est un nœud $\vee$. De ce fait, tout modèle optimal de Φ étant donné une fonction objectif composée d'une base de $\mathcal{L}$ et de la relation de préférence $\succeq_\Sigma$ ou $\succeq_{\mathrm{LEX}}$ est aussi un modèle optimal pour un de ses fils. Cela signifie qu'il existe au moins un fils dont le générateur de modèles est une interprétation partielle dont les extensions sont des modèles de Φ et dont au moins l'une de ces extensions est un modèle optimal. Pour choisir un générateur de modèles pour Φ, il suffit alors de calculer en temps polynomial la meilleure solution parmi celles des générateurs des fils.*

Exemple 4.6 (Choix d'un générateur de modèle pour un nœud $\vee$) *Soit $\Phi = \Phi_1 \vee \Phi_2$ une disjonction de formules DNNF telle que $\Phi_1 = a \wedge \neg b$ et $\Phi_2 = \neg a \wedge b$. Soit $\{(a,2),(b,1)\} \in \mathcal{L}$ une base pondérée et f l'agrégation de ses coûts par la somme (ou le leximax). $\{(a,1),(b,0)\}$ et $\{(a,0),(b,1)\}$ sont deux générateurs de modèles de respectivement Φ_1 et Φ_2, et chacun d'eux admet un modèle optimal parmi ses extensions.*

L'ensemble des modèles de Φ étant l'union des modèles de Φ_1 et Φ_2, on peut calculer un modèle minimal pour chacune de ces sous-formules afin de déterminer un modèle optimal pour Φ. Pour Φ_1, le seul modèle optimal est $\{(a,1),(b,0)\}$ et donne la valeur 2 pour la somme des coûts ($(0,2)$ pour le leximax) quand le seul modèle optimal de Φ_2 est $\{(a,0),(b,1)\}$ qui donne la valeur 1 pour la somme des coûts ($(0,1)$ pour le leximax). On peut de ce fait choisir $\{(a,0),(b,1)\}$ comme générateur de modèles pour Φ, puisque toutes ses extensions sont modèles de Φ et que l'une d'elle possède une extension qui minimise f.

Les quatre lemmes précédents sont suffisants pour prouver de la proposition suivante, concernant la possibilité d'optimiser en temps polynomial dans la taille de l'entrée une fonction composée d'une base de $\mathcal{L}$ et d'un agrégateur $\succeq_\Sigma$ ou $\succeq_{\mathrm{LEX}}$, quand les contraintes sont exprimées par une formule DNNF. Le processus d'optimisation qui en découle, lorsque $\succeq_\Sigma$ est considéré, est illustré par l'algorithme 4.1 et la figure 4.2.

Proposition 4.10 DNNF *satisfait* $\mathrm{OPT}[\succeq_\Sigma, \mathcal{L}]$ *et* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{L}]$.

Preuve *Preuve par induction : appliquer les lemmes 4.6, 4.7, 4.8, 4.9 dans l'ordre topologique inverse de la formule.*

Exemple 4.7 (Optimisation d'une formule DNNF par une fonction objectif linéaire) *On considère la formule de la figure 4.2. Nous décrivons ici le déroulement de l'algorithme sur cette formule.*

On considère tout d'abord l'ensemble des feuilles de la formule (ce qui est ici cohérent avec l'ordre topologique inverse), et on associe à chacune de ces feuilles le générateur de modèles qui contient le littéral de cette feuille. On considère ensuite les nœuds de conjonction ; on associe à chacun de ces nœuds le générateur de modèles correspondant à l'union des générateurs de ses nœuds fils.

Le prochain nœud dans l'ordre topologique inverse est le nœud $\vee$ *qui n'est pas la racine. Son générateur de modèles doit être choisi parmi ceux de ses fils, et doit être celui qui permet de minimiser la fonction objectif. Or, les deux générateurs des nœuds fils du nœud* $\vee$ *permettent d'obtenir la même valeur minimale (égale à* 1*) ; on choisit arbitrairement de remonter le générateur* $\{a, \neg b\}$*. Finalement, pour le nœud* $\vee$ *racine, les deux générateurs en concurrence sont* $\{a, \neg b\}$ *et* $\{a, c\}$*. Ici, toutes les extensions du deuxième vont donner un coût minimal de* 2*, ce qui est supérieur à la valeur obtenue par l'extension du générateur de modèles* $\{a, \neg b\}$ *qui minimise la fonction objectif, à savoir* $\{a, \neg b, \neg c\}$*, qui donne une valeur de* 1 *à cette fonction.*

Algorithme 4.1 : Optimisation d'une formule DNNF par une fonction objectif $f \in (\succeq_\Sigma, \mathcal{L})$

Entrées : une formule $\Phi \in$ DNNF, une fonction objectif $f \in \mathcal{L}$
Sorties : Un modèle de Φ optimal pour f

```
1  pour chaque nœud N de Φ, pris dans l'ordre topologique inverse faire
2  |  si N = ⊥ alors Gen(N) ← nil ;
3  |  si N = ⊤ alors Gen(N) ← ∅ ;
4  |  si N = v or N = ¬v with v ∈ PS alors Gen(N) ← {N} ;
5  |  si N = N_1 ∧ ... ∧ N_k alors
6  |  |  si ∃i ∈ {1,...,n} tel que Gen(N_i) = nil alors
7  |  |  |  Gen(N) ← nil;
8  |  |  sinon
9  |  |  |  Gen(N) ← ⋃_{i=1}^{n} Gen(N_i);
10 |  |  fin
11 |  fin
12 |  si N = N_1 ∨ ... ∨ N_k alors
13 |  |  si ∀i ∈ {1,...,n}, Gen(N_i) = nil alors
14 |  |  |  Gen(N) ← nil ;
15 |  |  sinon
16 |  |  |  Gen(N) ← Gen(N_i) tel que
17 |  |  |  min(f|Gen(N_i)) ≤ min(f|Gen(N_j)) pour j ∈ {1,...,n} ;
18 |  |  fin
19 |  fin
20 fin
21 retourner une extension de Gen(R_Φ) qui minimise f, où R_Φ est la racine de Φ;
```

Cette proposition est bien entendu applicable à tous les sous-langages de DNNF, ainsi qu'à toutes les sous-familles de $\mathcal{L}$; elle conclut l'étude de complexité concernant OPT[$\succeq_\Sigma, \mathcal{L}$] et OPT[$\succeq_{\text{LEX}}, \mathcal{L}$]. Il

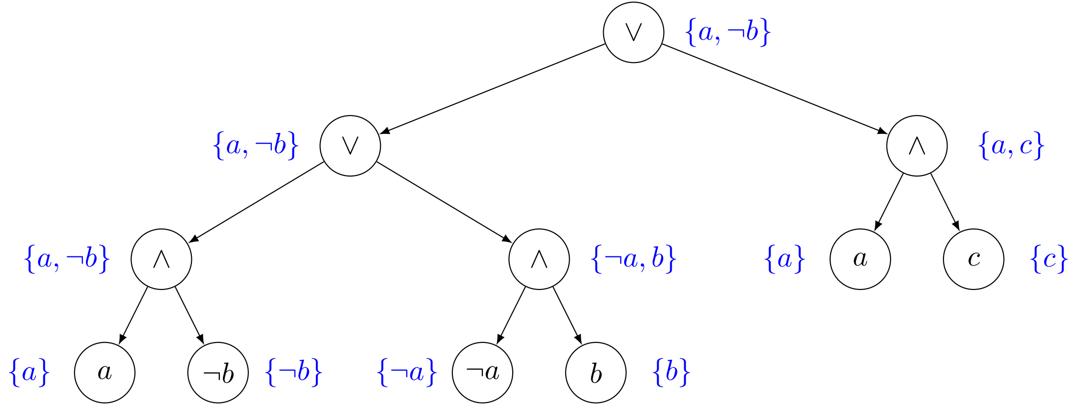


FIGURE 4.2 – Optimisation de la formule DNNF $\Phi = ((a \wedge \neg b) \vee (\neg a \wedge b)) \vee (a \wedge c)$ par la fonction objectif linéaire composée de la base pondérée $\phi = \{(a, 1), (b, 1), (c, 1)\}$ et de la relation de préférence $\succeq_\Sigma$, en appliquant l'algorithme 4.1. Les générateurs de modèles remontés sont en bleu.

est à noter que les propriétés sur lesquelles s'appuie cet algorithme ont déjà été utilisées dans le cadre d'autre travaux traitant de compilation de connaissances [DARWICHE & MARQUIS 2004, KIMMIG *et al.* 2012] ; l'utilisation que nous en faisons pour l'optimisation n'a cependant à notre connaissance pas été réalisée par le passé.

Nous allons donc maintenant nous intéresser à la requête OPT lorsque $\succeq_\Sigma$ ou $\succeq_{\text{LEX}}$ est concerné, mais cette fois-ci pour les familles de représentations qui sont plus générales que $\mathcal{L}$.

4.2.2 Fonctions objectifs générales

Le langage $\{\top\}$ auquel on fait référence par la suite est le langage contenant uniquement la tautologie $\top$, et nous permet de définir OPT pour l'optimisation sans contrainte. Puisque tous les langages de notre étude contiennent la formule $\top$, montrer que l'optimisation est NP-difficile sous $\top$ permet de montrer que le problème est NP-difficile pour chacun des langages que nous considérons.

En ce qui concerne les familles de représentation de notre étude plus générales que $\mathcal{L}$, les résultats théoriques sont beaucoup moins encourageants. Cependant, en posant des conditions très restrictives à la fois sur le langage de représentation des contraintes et sur les familles de fonctions objectifs considérées, il existe des cas pour lesquels l'optimisation est réalisable en temps polynomial. C'est par exemple le cas pour le langage DNF, lorsque les familles de fonctions objectifs considérées sont closes par restriction et permettent la minimisation sans contrainte en temps polynomial.

Proposition 4.11 *Soit $\mathcal{F}$ une famille de fonctions objectif close par restriction telle que* OPT$[\oplus, \mathcal{F}]$ *est satisfaite par le langage* $\{\top\}$. DNF *satisfait* OPT$[\oplus, \mathcal{F}]$.

Preuve *La preuve est donnée par l'algorithme suivant, qui s'exécute en temps polynomial. Pour chaque terme cohérent γ de la formule* DNF*, calculer en temps polynomial la solution minimale de la fonction objectif $f|\gamma$ (la restriction de f par γ) sous contrainte $\top$. Retourner une solution minimale pour f parmi les solutions minimales obtenues à partir de chacun des termes cohérents.*

Le langage NNF satisfaisant la requête de conditionnement, il est immédiat que le fait que la famille $\mathcal{G}$ des fonctions objectif représentées par l'agrégation de NNF pondérées est close par restriction. De plus, dans le cas où les poids et les littéraux des bases pondérées sont tous positifs, il est aisé de voir qu'une agrégation avec un opérateur monotone non décroissant tel que la somme ou le leximax permet

la minimisation en temps polynomial ; il suffit en effet d'affecter toutes les variables à la valeur de vérité *faux* pour obtenir une solution optimale. De ce fait, la proposition précédente implique le corollaire suivant.

Corollaire 4.12 DNF *satisfait* $\mathrm{OPT}[\succeq_\Sigma, \mathcal{G}^+_+]$ *et* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{G}^+_+]$.

Dans [DARWICHE & MARQUIS 2002], le langage DNF n'était pas considéré comme un langage de choix pour la compilation compte tenu des requêtes considérées, puisque celui-ci n'apportait pas de requêtes supplémentaires comparé à d-DNNF tout en étant strictement moins succinct. En revanche, on voit ici que dans le cadre de l'optimisation, le choix du langage DNF peut être tout à fait judicieux, tout du moins lorsque l'on dispose de fortes hypothèses quant à la représentation de la fonction objectif.

En revanche, relâcher une des conditions sur les bases pondérées suffit à rendre le problème théoriquement difficile, même dans le cas où les formules de ces bases ne sont que des cubes de taille deux.

Proposition 4.13 *Quand $\top$ peut être représenté dans le langage L, L ne satisfait ni* $\mathrm{OPT}[\succeq_\Sigma, \mathcal{Q}^+]$*, ni* $\mathrm{OPT}[\succeq_\Sigma, \mathcal{Q}_+]$*, ni* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{Q}^+]$*, ni* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{Q}_+]$.

Preuve

- $\mathrm{OPT}[\succeq_\Sigma, \mathcal{Q}_+]$ *et* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{Q}_+]$. *On considère le problème de décision suivant. Étant donné un ensemble fini de termes de taille 2 $S = \{\gamma_1, \ldots, \gamma_n\}$, on souhaite déterminer s'il existe une interprétation qui satisfait au moins k termes de S. Ce problème est connu comme étant* NP-*complet. Or, on peut associer en temps polynomial ce problème avec le problème d'optimiser la base pondérée $\{(\neg l_{i,1} \wedge l_{i,2}, 1), (l_{i,1} \wedge \neg l_{i,2}, 1), (\neg l_{i,1} \wedge \neg l_{i,2}, 1) \mid (\gamma_i = l_{i,1} \wedge l_{i,2}) \in S\}$ sous la contrainte $\top$, où l'agrégateur est $\succeq_\Sigma$ ou $\succeq_{\mathrm{LEX}}$. En effet, une solution optimale ω^* de ce problème est telle que $f(\omega^*) < n - k$ quand l'agrégateur est $\succeq_\Sigma$, et contient au moins k zéros (soit au plus $n - k$ uns) quand l'agrégateur est $\succeq_{\mathrm{LEX}}$, si et seulement si il existe une interprétation (ici, ω^*) qui satisfait au moins k termes de S.*
- $\mathrm{OPT}[\succeq_\Sigma, \mathcal{Q}^+]$ *et* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{Q}^+]$. *Considérons le problème de rechercher un ensemble intersectant minimal où le cardinal des ensembles est deux, de la même manière que dans la preuve de la proposition 4.3. On associe une variable x à chaque élément de l'ensemble référence $E = \cup_{c_i \in C} c_i$ ($|E| = n$), telle que x signifie « l'ensemble x n'est pas sélectionné dans l'ensemble intersectant ». On peut associer en temps polynomial en le cardinal de C, la collection de m sous-ensembles c_i de E et la constante k au problème d'optimiser sans contrainte $\top$ la base $\{(x_{i,1} \wedge x_{i,2}, 2), (x_{i,1}, -1), (x_{i,2}, -1) \mid c_i = \{x_{i,1}, x_{i,2}\} \in C\}$, pour l'agrégateur $\succeq_\Sigma$ ou l'agrégateur $\succeq_{\mathrm{LEX}}$. Pour chaque $c_i = \{x_{i,1}, x_{i,2}\} \in C$ et chaque interprétation ω, la « contribution » à $f(\omega)$ de la restriction de ω sur $\{x_{i,1}, x_{i,2}\}$ concédée par la base $\{(x_{i,1} \wedge x_{i,2}, 2), (x_{i,1}, -1), (x_{i,2}, -1)\}$ associée à c_i consiste en une agrégation des valeurs 2,-1 et 0. Quand $x_{i,1}$ et $x_{i,2}$ sont tous deux satisfaits (le résultat n'est pas un ensemble intersectant), les coûts obtenus sont les valeurs 2, -1, -1 ; quand un seul d'entre eux est satisfait, les valeurs obtenues sont 0, 0, et -1 ; finalement, quand aucun des deux n'est satisfait, les coûts sont 0,0 et 0. Par construction, l'interprétation pour laquelle tous les coûts sont de 0 est un ensemble intersectant (équivalent à E). Quand une interprétation correspond à un ensemble intersectant, le vecteur de coûts associé est composé des coûts 0 et -1. Ainsi, la solution optimale ω^* du problème* $\mathrm{OPT}[\succeq_{\mathrm{LEX}}, \mathcal{Q}^+]$ *décrit plus haut est telle que $f(\omega^*)$ contient le plus de -1 possible, sans contenir de 2. De ce fait, C possède un ensemble intersectant de taille maximale k si et seulement si $n + f(\omega^*) \leq k$ pour $\succeq_\Sigma$, et si et seulement si $f(\omega^*)$ contient au moins $n - k$ coûts -1 pour $\succeq_{\mathrm{LEX}}$.*

De plus, poser des conditions très restrictives sur les familles de représentation non linéaires n'est pas suffisant pour assurer que l'optimisation ne soit pas NP-difficile. En effet, lorsque le langage de représen-

tation des contraintes considéré n'est pas DNF ou un de ses sous-langages, le problème d'optimisation est là encore théoriquement intraitable, même pour des bases de $\mathcal{Q}$.

Proposition 4.14 OBDD$_<$ *ne satisfait ni* OPT$[\succeq_\Sigma, \mathcal{Q}^+_+]$*, ni* OPT$[\succeq_{\text{LEX}}, \mathcal{Q}^+_+]$ *;* PI *ne satisfait ni* OPT$[\succeq_\Sigma, \mathcal{Q}^+_+]$*, ni* OPT$[\succeq_{\text{LEX}}, \mathcal{Q}^+_+]$.

Preuve *Nous avons prouvé que L ne satisfaisait ni* OPT$[\succeq_\Sigma, \mathcal{Q}_+]$*, ni* OPT$[\succeq_{\text{LEX}}, \mathcal{Q}_+]$ *à partir du moment où $\top$ pouvait être représenté dans L ; ce qui est le cas pour* OBDD$_<$ *et* PI*. Nous montrons maintenant que concernant* OBDD$_<$ *et* PI*, il existe une réduction du problème d'optimisation* OPT$[\succeq_\Sigma, \mathcal{Q}_+]$ *vers* OPT$[\succeq_\Sigma, \mathcal{Q}^+_+]$*. Pour cela, il suffit d'introduire pour chaque littéral négatif $\neg x$ appartenant à la base pondérée une nouvelle variable n_x telle que $\neg x \Leftrightarrow n_x$. La conjonction des formules $\neg x \Leftrightarrow n_x$ peut être représentée en temps polynomial par une formule* OBDD$_<$ *ou* PI *; il reste alors à remplacer la contrainte $\top$ par cette formule pour obtenir un problème d'optimisation dont les solutions optimales sont les mêmes que celles du problème original.*

Le fait que ces résultats soient majoritairement négatifs nous a poussé à rechercher la ou les causes de la complexité de l'optimisation hors du cadre des fonctions linéaires. Il apparaît qu'une de ces causes est la taille de la fonction objectif. De ce fait, nous avons de nouveau étudié la complexité théorique des problèmes d'optimisation, mais en considérant cette fois-ci que la taille des fonctions objectif était bornée.

4.2.3 Résultats de *fixed-parameter tractability*

Dans la section précédente, nous avons considéré des problèmes qui admettent deux paramètres, à savoir une formule propositionnelle Φ représentant des contraintes, et une fonction objectif f représentée par une base pondérée ϕ et paramétrée par un agrégateur de coûts $\oplus$. Les résultats obtenus jusqu'à lors étaient valables pour des fonctions objectifs de taille quelconque. Dans cette section, on considère cette fois que le cardinal $|\phi|$ des bases pondérées est cette fois-ci borné, et on s'intéresse une nouvelle fois à la complexité des problèmes d'optimisation de notre étude. Un problème appartenant à la classe de complexité P une fois un de ses paramètres p borné est dit *fixed parameter tractable* [Fellows & Langston 1987] pour le paramètre p.

Il faut toutefois être prudent avec cette notion, puisqu'elle permet de considérer comme traitable un très grand nombre de problèmes, une fois le bon paramètre considéré comme borné. Par exemple, on considérant le nombre de variables n d'une formule CNF comme borné, le problème de vérifier la cohérence d'une CNF devient traitable. En effet, énumérer les 2^n interprétations devient « facile » puisque 2^n est alors une constante ; de ce fait, vérifier qu'il existe parmi ces interprétations une qui soit modèle de la formule devient tout aussi facile.

Néanmoins, nous allons considérer de tels résultats en admettant que le cardinal de la représentation sous forme de bases pondérées est borné. Cela peut avoir du sens dans certains cas, notamment lorsque la difficulté pratique du calcul vient de la combinatoire des contraintes plutôt que de la fonction objectif elle-même. Si un langage L satisfait OPT pour une relation de préférence $\succeq$ et une famille de bases pondérées $\mathcal{F}$ données lorsque le cardinal des bases $|\phi|$ est considéré borné, on dira alors que « L satisfait OPT$_{|\phi|}[\succeq, \mathcal{F}]$ ».

Sous cette nouvelle hypothèse, nos premiers résultats sont positifs en ce qui concerne la famille de représentation $\mathcal{P}$, dans le sens où l'optimisation passe de NP-difficile à la classe P.

Proposition 4.15 *Soit L un langage qui satisfait les propriétés* CD *et* CO*. L satisfait* OPT$_{|\phi|}[\succeq_\Sigma, \mathcal{P}]$ *et* OPT$_{|\phi|}[\succeq_{\text{LEX}}, \mathcal{P}]$.

Preuve *Soit $\Phi \in L$ et $\{(t_i, w_i) \mid i = \{1, \dots, n\}\}$ une base pondérée de $\mathcal{P}$. Il existe 2^n ensembles de la forme $\{t_i^* \mid i = \{1, \dots, n\}\}$ où chaque t_i^* est soit t_i, soit une clause équivalente à $\neg t_i$ obtenue comme disjonction des négations des littéraux de t_i. À chacun de ces ensembles est associé un vecteur de n réels $(v_1, \dots, v_n)$ où chaque v_i est égal à w_i quand t_i est dans l'ensemble, et à zéro sinon. Quand n est borné, le nombre de ces ensembles est donc lui aussi borné. De plus, chacun de ces ensembles peut être vu comme une formule* CNF *α qui contient au plus n clauses de taille supérieure à 1. Chaque clause de α contient au plus m littéraux, ce qui implique que α peut être traduite en une formule* DNF *α' équivalente à α en temps $O(m^n)$, qui contient au plus $O(m^n)$ termes cohérents. Quand L satisfait* CD *et* CO*, on peut facilement vérifier si $\alpha' \land \Phi$ est cohérent en déterminant si il existe un terme consistent t dans α' tel que $\Phi_{|t}$ est cohérent. Lors de ce processus énumératif, tous les α' tels que $\alpha' \land \Phi$ est incohérent sont simplement ignorés. Pour les α' restants, il est aisé de calculer la valeur $f(\omega_{\alpha'})$ où $\omega_{\alpha'}$ correspond à n'importe quel modèle de α', et donc aussi de n'importe quel modèle de α. Par construction, cette valeur est égale à l'agrégation ($\succeq_\Sigma$ ou $\succeq_{\text{LEX}}$) appliquée au vecteur $(v_1, \dots, v_n)$ associé à α. Il suffit alors de mémoriser à chaque étape du processus énumératif la solution préférée à toutes les autres, et de la retourner à la fin du processus (si il n'existe pas de solution, on retourne ce résultat).*

Néanmoins, lorsque l'on considère la famille $\mathcal{G}$ des représentations générales, la condition sur le cardinal de l'ensemble de bases pondérées n'est cette fois pas suffisante pour obtenir un algorithme polynomial pour l'optimisation (si P$\neq$NP.

Proposition 4.16 *Quand $\top$ peut être représenté en temps polynomial dans un langage L, L ne satisfait ni $\text{OPT}_{|\phi|}[\succeq_\Sigma, \mathcal{G}^+]$, ni $\text{OPT}_{|\phi|}[\succeq_{\text{LEX}}, \mathcal{G}^+]$, ni $\text{OPT}_{|\phi|}[\succeq_\Sigma, \mathcal{G}_+]$, ni $\text{OPT}_{|\phi|}[\succeq_{\text{LEX}}, \mathcal{G}_+]$ pour $|\phi| \geq 2$.*

Preuve *On considère le problème (*NP*-complet) de déterminer la cohérence de la formule* CNF *$\psi^+ \land \psi^-$, où toutes les clauses de ψ^+ sont positives (i.e. ne contiennent que des littéraux positifs), et où toutes les clauses de ψ^- sont négatives.*

- $\text{OPT}_{|\phi|}[\succeq_\Sigma, \mathcal{G}^+]$ *et* $\text{OPT}_{|\phi|}[\succeq_{\text{LEX}}, \mathcal{G}^+]$. *On peut associer en temps polynomial toute formule $\psi^+ \land \psi^-$ au problème d'optimiser la base $\{(\psi^+, -1), (\neg\psi^-, 1)\}$ (cette base est dans $\mathcal{G}^+$) sous la contrainte $\top$ pour les agrégateurs $\succeq_\Sigma$ et $\succeq_{\text{LEX}}$. Or, $\psi^+ \land \psi^-$ est cohérent si et seulement si l'optimisation par la somme renvoie -1 ou l'optimisation par le leximax retourne le vecteur $(-1, 0)$.*
- $\text{OPT}_{|\phi|}[\succeq_\Sigma, \mathcal{G}^+]$ *et* $\text{OPT}_{|\phi|}[\succeq_{\text{LEX}}, \mathcal{G}^+]$. *On peut associer en temps polynomial toute formule $\psi^+ \land \psi^-$ au problème d'optimiser la base $\{(\neg\psi^+, 1), (\neg\psi^-, 1)\}$ (cette base est dans $\mathcal{G}^+$) sous la contrainte $\top$ pour les agrégateurs $\succeq_\Sigma$ et $\succeq_{\text{LEX}}$. Or, $\psi^+ \land \psi^-$ est cohérent si et seulement si l'optimisation par la somme renvoie 0 ou l'optimisation par le leximax retourne le vecteur $(0, 0)$.*

De plus, en dehors du cas où on considère le langage cible DNF, le fait d'imposer de très fortes restrictions sur la famille $\mathcal{G}$ ne suffit pas pour obtenir de meilleurs résultats, même en bornant $|\phi|$.

Corollaire 4.17 $\text{OBDD}_<$ *et* PI *ne satisfont pas les propriétés* $\text{OPT}_{|\phi|}[\succeq_\Sigma, \mathcal{G}_+^+]$ *et* $\text{OPT}_{|\phi|}[\succeq_{\text{LEX}}, \mathcal{G}_+^+]$*, pour $|\phi| \geq 2$.*

Preuve *Dans la preuve de la proposition 4.14, nous présentons une réduction de* $\text{OPT}[\succeq_\Sigma, \mathcal{Q}_+]$ *vers* $\text{OPT}[\succeq_\Sigma, \mathcal{Q}_+^+]$ *quand la formule à optimiser est un* $\text{OBDD}_<$*. La même réduction est applicable pour passer de* $\text{OPT}[\succeq_\Sigma, \mathcal{G}_+]$ *vers* $\text{OPT}[\succeq_\Sigma, \mathcal{G}_+^+]$ *et de* $\text{OPT}[\succeq_{\text{LEX}}, \mathcal{G}_+]$ *vers* $\text{OPT}[\succeq_{\text{LEX}}, \mathcal{G}_+^+]$*, aussi bien en ce qui concerne* $\text{OBDD}_<$ *que* PI*.*

Le langage SDNNF (et ses sous-langages dont OBDD fait partie) est un langage considéré comme intéressant pour la compilation de contraintes dans la mesure où il autorise la conjonction bornée. Cette propriété permet d'ajouter « efficacement » en nombre borné des contraintes une fois la formule compilée. Or, dans notre cadre, nous pouvons tirer parti de cette propriété lorsque le nombre de bases pondérées est borné pour obtenir un algorithme d'optimisation polynomial.

Proposition 4.18 *Quand chaque ϕ_i est une formule* $\mathsf{OBDD}_<$*,* $\mathsf{OBDD}_<$ *satisfait* $\mathrm{OPT}_{|\phi|}[\succeq_\Sigma, \mathcal{G}]$ *et* $\mathrm{OPT}_{|\phi|}[\succeq_{\mathrm{LEX}}, \mathcal{G}]$.

Preuve *L'approche consiste à remplacer chaque* $\mathsf{OBDD}_<$ ϕ_i *qui n'est pas un littéral par une variable* n_{ϕ_i} *et à remplacer la formule* Φ *par la formule* $\Phi \wedge (\wedge_{i=1}^{n} n_{\phi_i} \Leftrightarrow \phi_i)$*. Cette formule* $\mathsf{OBDD}_<$ *peut être calculée en* $O(|\Phi|).(2max_{i=1}^{n}|\phi_i| + 1)^{n-1}$ *puisque* $\mathsf{OBDD}_<$ *satisfait les propriétés* $\neg$C *et* $\wedge$BC*, et que la représentation en* $\mathsf{OBDD}_<$ *de* $n_{\phi_i} \Leftrightarrow \phi_i$ *peut être calculée en temps polynomial, peu importe l'ordre* $<$*.*

Les résultats obtenus dans cette section sont synthétisés à la table 4.1. Comme le montre ce tableau, nous avons mis en évidence deux scénarios dans lesquels l'optimisation peut s'effectuer en temps polynomial :

– en considérant une fonction objectif linéaire, et une formule DNNF ;
– en considérant une fonction objectif appartenant à une famille close par restriction qui permet l'optimisation sans contrainte en temps polynomial, et une formule DNF.

Nous montrons dans les sections suivantes que les cas d'optimisation polynomiale que nous avons mis en évidence ne peuvent pas être facilement généralisés à d'autres familles de fonctions objectifs.

	$\mathcal{G}$	$\mathcal{G}^+$	$\mathcal{G}_+$	$\mathcal{G}_+^+$	$\{\mathcal{P},\mathcal{Q}\}$	$\{\mathcal{P},\mathcal{Q}\}^+$	$\{\mathcal{P},\mathcal{Q}\}_+$	$\{\mathcal{P},\mathcal{Q}\}_+^+$	$\mathcal{L}$	$\mathcal{L}^+$	$\mathcal{L}_+$	$\mathcal{L}_+^+$
DNNF	$\times$	$\times$	$\times$	$\times$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\surd$	$\surd$	$\surd$
$(*)$ DNNF$_T$,OBDD$_<$	$\times$	$\times$	$\times$	$\times$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\surd$	$\surd$	$\surd$
DNF, IP, MODS	$\times$	$\times$	$\times$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
PI	$\times$	$\times$	$\times$	$\times$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$

TABLE 4.1 – Complexité de la requête OPT pour les langages pour lesquels le test de cohérence s'effectue en temps polynomial, quand les coûts de la base pondérée sont agrégés par la somme ou le leximax. $\surd$ signifie « temps polynomial », $\oslash$ signifie « temps polynomial si $|\phi|$ est borné, temps exponentiel sous l'hypothèse P $\neq$ NP sinon » et $\times$ signifie « temps exponentiel sous l'hypothèse P $\neq$ NP ».
$(*)$ $\mathsf{OBDD}_<$ (resp. DNNF_T) satisfait $\mathrm{OPT}_{|\phi|}[\oplus,\mathcal{G}]$ si $|\phi|$ est borné pour $\oplus = \succeq_\Sigma$ et $\oplus = \succeq_{\mathrm{LEX}}$ quand chaque ϕ_i est dans $\mathsf{OBDD}_<$ (resp. DNNF_T).

4.3 Raffinement de la frontière entre optimisation simple et complexe

Dans cette section, nous considérons un ensemble de fonctions ou d'agrégateurs plus généraux que ceux considérés jusqu'à lors. Nous dessinons une frontière entre l'optimisation simple (dans P) et l'optimisation complexe (NP-difficile). Nous montrons dans un premier temps que considérer un opérateur plus général que la somme et le leximax peut rendre l'optimisation difficile.

4.3.1 Agrégation via opérateurs OWA

Dans cette section, nous démontrons que la minimisation d'une fonction objectif représentée dans $\mathcal{L}$ d'un opérateur OWA est NP-difficile. On notera OWA$_W$ l'opérateur OWA défini par le vecteur de poids $W = (w_1, \ldots, w_n)$, et $\succeq_{\mathrm{OWA}_w}$ la relation de préférence associée.

On observe d'abord qu'il n'est pas possible d'utiliser le même algorithme que pour l'optimisation de fonctions linéaires quand l'agrégation est une somme, comme le montre l'exemple suivant. Soit $\Phi = \Phi_1 \wedge \Phi_2$ une formule d-DNNF telle que $\Phi_1 = (a \wedge \neg b) \vee (\neg a \wedge b)$ et $\Phi_2 = (c \wedge \neg d) \vee (\neg c \wedge d)$. Soit $\phi = \{(a, 2), (\neg a, 1), (b, 1), (\neg b, 0), (c, 2), (\neg c, 1), (d, 1), (\neg d, 0)\}$ une base pondérée et $\oplus$ un opérateur OWA qui agrège $n = 8$ coûts via le vecteur de poids $W = (0, 0, 0, 0, 0, \frac{1}{3}, \frac{2}{3}, 0)$ (c'est-à-dire, un tiers fois la statistique d'ordre $n - 2$ plus deux tiers fois la statistique d'ordre $n - 1$). Considérons l'optimisation

de ϕ avec l'agrégateur OWA_W pour les modèles de Φ_1. Pour minimiser la fonction objectif, on pourrait choisir le modèle $\{a, \neg b, \neg c, \neg d\}$. De façon similaire, en considérant les modèles de Φ_2, on pourrait choisir le modèle $\{\neg a, \neg b, c, \neg d\}$. Cependant, si on considère les contraintes de Φ, minimiser la fonction objectif nécessite de choisir le modèle $\{\neg a, b, \neg c, d\}$, qui n'est cohérent ni avec $\{a, \neg b, \neg c, \neg d\}$, ni avec $\{\neg a, \neg b, c, \neg d\}$; ceci montre bien que dans le cadre de l'optimisation de coûts agrégés par un opérateur OWA, même dans le cas où les contraintes sont exprimées par une formule d-DNNF, le processus d'optimisation n'est pas séparable, et ne semble donc pas pouvoir être réalisé par un algorithme procédant de manière inductive en considérant les nœuds dans l'ordre topologique inverse.

Afin de prouver la complexité de ce problème, nous allons présenter une réduction polynomiale de $\text{OPT}[\succeq_\Sigma, \mathcal{Q}_+]$ vers $\text{OPT}[\succeq_{\text{OWA}_W}, \{\top\}]$, et donc prouver la NP-difficulté du problème d'optimiser avec un opérateur de somme pondérée ordonnée quelconque comme agrégateur.

Tout d'abord, considérons le cas simple de l'optimisation d'une base contenant un unique cube de taille 2 ($\phi = \{(l_1 \wedge l_2, c)\}$). On agrège avec la somme. On note que la valeur de la fonction f considérée est égale à c si et seulement si à la fois l_1 et l_2 sont évalués à 1 (*vrai*). Une autre façon de voir le problème est de dire que f prend la valeur c pour une interprétation ω si et seulement si le minimum des valeurs de vérité de l_1 et l_2 est égal à 1 :

$$f(\omega) = c \Leftrightarrow \min(\{l_1, l_2\}) = 1 \ (1).$$

Or, comme nous l'avons vu précédemment, l'opérateur $\min$ est inclus dans l'ensemble des opérateurs OWA, comme tous les opérateurs permettant de récupérer la statistique de rang $i \in \{1, \ldots, n\}$. De ce fait, il est possible de réécrire (1) de la façon suivante :

$$f(\omega) = c \Leftrightarrow \text{OWA}_{[1,0]}(l_1, l_2) = 1 \ (2).$$

En formalisant (2) comme un problème d'optimisation d'une base pondérée dont les coûts unitaires sont agrégés, le problème devient le suivant :

$$\text{min. } \text{OWA}_{[1,0]}(\{(l_1, c), (l_2, c)\}) \ (3).$$

Il est aussi possible de faire apparaître les littéraux complémentaires de l_1 et l_2. Ceci nous sera utile dans la suite de notre preuve :

$$\text{min. } \text{OWA}_{[0,0,1,0]}(\{(l_1, c), (l_2, c), (\sim l_1, 0), (\sim l_2, 0)\}) \ (4).$$

Nous avons donc mis en évidence une réduction polynomiale de $\text{OPT}[\succeq_\Sigma, \mathcal{Q}_+]$ vers $\text{OPT}[\succeq_{\text{OWA}_W}, \{\top\}]$ lorsque la taille de la base pondérée du problème original est de cardinal égal à 1. Nous allons maintenant généraliser ce résultat à des bases pondérées initiales de cardinal quelconque.

L'idée derrière cette généralisation est de regrouper les valeurs de vérité concernant un même cube, ceci malgré l'ordre qui est imposé par l'opérateur OWA. Afin de réussir à regrouper ces valeurs de vérité, il faut que les coûts qu'elles engendrent soient proches, peu importe les valeurs de vérité prises par les variables. Considérons le problème de minimisation de somme suivant :

$$\text{min. } f_\Sigma(\omega) = \Sigma(\{(l_{1,1} \wedge l_{1,2}, 1), (l_{2,1} \wedge l_{2,2}, 2)\}) \ (5).$$

Considérons maintenant le problème de minimisation d'agrégation par un opérateur OWA suivant :

$$\text{min. } f_{\text{OWA}}(\omega) = \text{OWA}_{[0,0,0,0,\frac{1}{2},0,\frac{1}{2},0]}$$
$$(\{(l_{1,1}, 1), (l_{1,2}, 1), (\sim l_{1,1}, 0), (\sim l_{1,2}, 0), (l_{2,1}, 12), (l_{2,2}, 12), (\sim l_{2,1}, 10), (\sim l_{2,2}, 10)\}) \ (6).$$

On remarque que, pour toute interprétation ω, on a $f_{\text{OWA}}(\omega) = \frac{1}{2}(f_\Sigma(\omega) + 10)$, et que les deux problèmes de minimisation ont donc les mêmes solutions minimales. Ceci s'explique par la composition des vecteurs de coûts générés par la base de (6) pour chacune des solutions :

- les quatre coûts les plus faibles sont nuls : ils correspondent aux quatre bases qui sont évalués à *faux* et n'ajoutent donc aucun coût à $f_{\text{OWA}}(\omega)$;
- les cinquième et sixième coûts les plus faibles correspondent respectivement à $1 * \min(\{l_{1,1}, l_{1,2}\})$ et $1 * \max(\{l_{1,1}, l_{1,2}\})$; puisqu'ils sont respectivement associés par l'opérateur OWA aux poids $\frac{1}{2}$ et 0, la valeur qu'ils ajoutent à $f_{\text{OWA}}(\omega)$ est de $\frac{1}{2}$ fois la valeur associée à la base $(l_{1,1} \wedge l_{1,2}, 1)$ de (5) ;
- les septième et huitième coûts les plus faibles correspondent respectivement à $10{+}2{*}\min(\{l_{2,1}, l_{2,2}\})$ et $10{+}2{*}\max(\{l_{1,1}, l_{1,2}\})$; puisqu'ils sont respectivement associés par l'opérateur OWA aux poids $\frac{1}{2}$ et 0, la valeur qu'ils ajoutent à $f_{\text{OWA}}(\omega)$ est de $\frac{1}{2}$ fois la valeur associée à la base $(l_{1,1} \wedge l_{1,2}, 2)$ de (5) à laquelle on a ajouté la valeur 10.

On voit bien dans cet exemple qu'il est possible de faire en sorte que les coûts associés aux littéraux d'un même cube de taille deux soient successifs une fois le tri des coûts opéré : il suffit pour cela que les poids associés à chacun des littéraux d'un cube soient tous deux strictement inférieurs ou strictement supérieurs aux autres poids du problème. Puisque la valeur maximale d'un coût est borné, il est possible de considérer des constantes suffisamment grandes pour que ceci soit possible. On a donc la proposition suivante.

Proposition 4.19 *Soient*
- $\phi_1 = \{(l_{1,1} \wedge l_{1,2}, c_1), \ldots, (l_{n,1} \wedge l_{n,2}, c_n)\} \in \mathcal{Q}_+$ *et*
- $\phi_2 = \{(\sim l_{1,1}, K^1 + c_1), (\sim l_{1,2}, K^1 + c_1), \ldots, (\sim l_{n,1}, K^n + c_n), (\sim l_{n,2}, K^n + c_n),$
  $(l_{1,1}, K^1 + c_1), (l_{1,2}, K^1 + c_1), \ldots, (l_{n,1}, K^n + c_n), (l_{n,2}, K^n + c_n)\} \in \mathcal{L}_+$

deux bases pondérées, tel que $K > \max(\{c_i \mid i \in 1, \ldots, n\})$. *Les solutions optimales du problème de minimiser* $\Sigma(\phi_1)$ *sont exactement celles du problème de minimiser* OWA$_W(\phi_2)$*, où* W *est un vecteur de poids composé de* $2|\phi_1|$ *fois le poids 0 et* $|\phi_1|$ *fois les poids* $\frac{1}{|\phi_1|}$ *et* 0.

Preuve *Les vecteurs de coûts issus de la base* ϕ_2 *sont composés des valeurs suivantes :*
- $2|\phi_1|$ *fois la valeur zéro, puisque pour un cube* $l_{i,1} \wedge l_{i,2}$*, deux littéraux parmi* $\{l_{i,1}, l_{i,2}, \sim l_{i,1}, \sim l_{i,2}\}$ *seront falsifiés ;*
- $K^1 + c_1 * \min(\{l_{1,1}, l_{1,2}\})$ *et* $K^1 + c_1 * \max(\{l_{1,1}, l_{1,2}\})$*, puisque par construction,* $K^1 > 0$ *et* $K^1 + c_i < K^{1+p}$ *pour tout coût* c_i *et tout entier strictement positif* p *;*
- $\ldots$
- $K^n + c_n * \min(\{l_{n,1}, l_{n,2}\})$ *et* $K^n + c_n * \max(\{l_{n,1}, l_{n,2}\})$*, puisque par construction,* $K^n > K^{n-p} + c_i$ *pour tout coût* c_i *et tout entier strictement positif* p.

En considérant les poids associés par l'opérateur OWA, on se retrouve alors avec une valeur de $\frac{K^1 * c_1}{|\phi_1|} * \max(\{l_{1,1}, l_{1,2}\}) + \ldots + \frac{K^n * c_n}{|\phi_1|} * \max(\{l_{n,1}, l_{n,2}\})$*, c'est-à-dire une transformation linéaire de la valeur de la fonction d'optimisation pour la somme.*

Ce qui implique directement le corollaire suivant.

Corollaire 4.20 *Quand* $\top$ *peut être représenté dans un langage* L*,* L *ne satisfait pas* $\mathsf{OPT}[\succeq_{OWA_W}, \mathcal{L}]$.

Ce résultat suggère qu'il est difficile d'optimiser en temps polynomial avec des agrégateurs plus généraux que la somme et le leximax, même quand la base pondérée fait partie de la famille $\mathcal{L}$. Dans la suite, nous nous intéressons à des extensions minimes des fonctions objectifs linéaires en considérant toutefois un opérateur pour lequel nous possédons des résultats positifs ($\succeq_\Sigma$), et nous montrons que l'optimisation pour ces extensions est NP-difficile même quand on se limite à des langages de représentation de contraintes restreints.

4.3.2 Extensions des fonctions linéaires

Redéfinissons d'abord les fonctions linéaires agrégées par la somme de la manière suivante.

Définition 4.21 *La famille de fonction $\mathcal{L}_\Sigma$ est définie inductivement de la manière suivante :*

1. *si l est un littéral, $l \in \mathcal{L}_\Sigma$;*
2. *si $f \in \mathcal{L}_\Sigma$ et $w \in \mathbb{R}$, $w \cdot f \in \mathcal{L}_\Sigma$;*
3. *si $f_1, f_2 \in \mathcal{L}_\Sigma$, $f_1 + f_2 \in \mathcal{L}_\Sigma$.*

Dans cette définition, les deux opérateurs « constructeurs » autorisés ($+$ et $\cdot$) sont monotones. De ce fait, on peut se demander s'il n'est pas possible d'étendre cette définition à d'autres opérateurs monotones tout en continuant de permettre l'optimisation en temps polynomial pour certains langages de contraintes. On définit de ce fait la famille des fonctions linéaires étendues, où est autorisé l'ajout d'un opérateur monotone d'arité quelconque.

Définition 4.22 *La famille de fonction $\mathcal{L}_{\Sigma,+}$ est définie inductivement de la manière suivante :*

1. *si l est un littéral, $l \in \mathcal{L}_{\Sigma,+}$;*
2. *si $f \in \mathcal{L}_{\Sigma,+}$ et $w \in \mathbb{R}$, $w \cdot f \in \mathcal{L}_{\Sigma,+}$;*
3. *si $f_1, f_2 \in \mathcal{L}_{\Sigma,+}$, $f_1 + f_2 \in \mathcal{L}_{\Sigma,+}$;*
4. *si $f_1, \dots, f_k \in \mathcal{L}_{\Sigma,+}$ et $\otimes$ est un opérateur monotone d'arité k, $\otimes(f_1, \dots, f_k) \in \mathcal{L}_{\Sigma,+}$.*

On montre que le fait d'ajouter un opérateur monotone, même unaire, ne permet malheureusement plus d'assurer une optimisation en temps polynomial quel que soit le langage de contraintes contenant $\top$.

Proposition 4.23 *Soit L un langage contenant $\top$. L'optimisation d'une fonction de $\mathcal{L}_{\Sigma,+}$ sous contraintes dans L est* NP*-difficile.*

Preuve *On considère dans un premier temps le cas où l'opérateur $\otimes$ est d'arité $k = 1$. Soit $\psi = (l_{1,1} \vee l_{1,2} \vee l_{1,3}) \wedge \dots \wedge (l_{n,1} \vee l_{n,2} \vee l_{n,3})$ une formule 3-*CNF*. Soit L un langage tel que $\top \in L$. On peut associer en temps polynomial à ψ le problème de minimisation sous contrainte $\top$ de $f(\omega) = \Sigma_{i=1}^{n} \otimes (l_{i,1} + l_{i,2} + l_{i,3})$ où $\otimes(x) = \min(1, x)$ est une fonction unaire monotone, et où $f \in \mathcal{L}_{\Sigma,+}$. Il existe une instantiation complète des variables satisfaisant au moins k clauses de ψ si et seulement si le problème d'optimisation renvoie une solution optimale dont la valeur est au moins égale à k. Or, ceci correspond au problème MIN-SAT, connu comme étant* NP*-difficile [*KOHLI et al. *1994]. La preuve s'étend aux opérateurs $\otimes$ monotones d'arité $k > 1$ en considérant $\otimes(x, \dots) = \min(1, x)$.*

On remarque d'ailleurs que ce résultat tient toujours en ajoutant une condition de « décomposabilité » sur la fonction objectif elle-même. On définit les fonctions linéaires étendues décomposables de la manière suivante.

Définition 4.24 *La famille de fonction D-$\mathcal{L}_{\Sigma,+}$ est définie inductivement de la manière suivante :*

1. *si l est un littéral, $l \in$ D-$\mathcal{L}_{\Sigma,+}$;*
2. *si $f \in$ D-$\mathcal{L}_{\Sigma,+}$ et $w \in \mathbb{R}$, $w \cdot f \in$ D-$\mathcal{L}_{\Sigma,+}$;*
3. *si $f_1, f_2 \in$ D-$\mathcal{L}_{\Sigma,+}$ et $Vars(f_1) \cap Vars(f_2) = \emptyset$, $f_1 + f_2 \in$ D-$\mathcal{L}_{\Sigma,+}$;*
4. *si $f_1, \dots, f_k \in$ D-$\mathcal{L}_{\Sigma,+}$, $Vars(f_1) \cap \dots \cap Vars(f_k) = \emptyset$, et $\otimes$ est un opérateur monotone d'arité k, $\otimes(f_1, \dots, f_k) \in$ D-$\mathcal{L}_{\Sigma,+}$.*

Bien que la condition de décomposabilité de la contrainte permette l'optimisation en temps polynomial pour les fonctions linéaires, la proposition suivante montre qu'il n'est pas suffisant de l'appliquer aux fonctions linéaires étendues pour permettre leur optimisation en temps polynomial lorsque la contrainte est une formule OBDD. On voit d'ailleurs dans la preuve que cette insuffisance est due au fait qu'il est possible de créer des chaînes d'équivalence en temps polynomial en utilisant une formule OBDD.

Proposition 4.25 *L'optimisation d'une fonction de D-$\mathcal{L}_{\Sigma,+}$ sous contraintes dans* OBDD$_<$ *est* NP-*difficile.*

Preuve *On considère dans un premier temps le cas où l'opérateur $\otimes$ est d'arité $k = 1$. Soit $\psi = (l_{1,1} \vee l_{1,2}) \wedge \ldots \wedge (l_{n,1} \vee l_{n,2})$ une formule 2-*CNF*. On peut associer en temps polynomial à ψ le problème de minimisation de $f(\omega) = \Sigma_{i=1}^{n} \otimes (x_{i,1} + x_{i,2})$ sous contrainte* OBDD$_<$ *$\psi' = \wedge_{i=1}^{n} \wedge_{j=1}^{m} (x_{i,j} \Leftrightarrow n_{i,j}(x_i))$ où $n_{i,j}(x_i) = x_i$ quand $l_{i,j}$ est un littéral positif, $n_{i,j}(x_i) = \neg x_i$ quand $l_{i,j}$ est un littéral négatif, $\otimes(x) = \min(1, x)$ est une fonction unaire monotone. Dans ce problème, on a bien $f \in$ D-$\mathcal{L}_{\Sigma,+}$ et la propriété de décomposabilité de la fonction est assurée par le fait que chaque variable n'apparaît qu'une unique fois dans la fonction objectif. L'ordre $<$ choisi pour ψ' est supposé respecter $x_i < x_{i,j} < x_{i+1}$ pour que la construction de la contrainte se fasse en temps polynomial. Il existe une assignation complète des variables satisfaisant au moins k clauses de ψ si et seulement si le problème d'optimisation renvoie une solution optimale dont la valeur est au moins égale à k. Or, ceci correspond au problème 2-MIN-SAT, connu comme étant* NP-*difficile [*KOHLI et al. *1994]. La preuve s'étend aux opérateurs $\otimes$ monotones d'arité $k > 1$ en considérant $\otimes(x_1, \ldots) = \min(1, x)$.*

Cette proposition clôt notre étude tendant à montrer que l'optimisation sur les sous-langages de DNNF en temps polynomial est calculatoirement difficile dès que l'on sort du cas linéaire. En revanche, comme nous l'avons vu plus haut, si une contrainte est sous forme DNF et que l'on sait optimiser une fonction d'une famille $\mathcal{F}$ close par conditionnement, alors l'optimisation est réalisable en temps polynomial. Nous présentons maintenant le dernier point de ce chapitre, qui traite de la minimisation des fonctions sous-modulaires.

4.3.3 Fonctions sous-modulaires

Dans cette section, nous nous intéressons à la famille des fonctions sous-modulaires, une famille de fonctions très générales, abondamment étudiées dans la littérature (voir par exemple [FUJISHIGE 2005] pour un *survey paper*, [ORLIN 2009] et [FUJISHIGE & ISOTANI 2011] dans le cas non contraint, ou encore [GOEL *et al.* 2010] pour l'optimisation sous contraintes).

On dit qu'une fonction pseudo-booléenne $f : \{0,1\}^n \rightarrow \mathbb{R}$ est sous-modulaire si et seulement si pour toute paire de vecteurs $x_1, x_2 \in \{0,1\}^n$, on a $f(x_1 \wedge x_2) + f(x_1 \vee x_2) \leq f(x_1) + f(x_2)$, où $x_1 \wedge x_2$ (resp. $x_1 \vee x_2$) est le vecteur dont les composantes d'indice i ont pour valeur le minimum (resp. le maximum) entre les composantes d'indice i de x_1 et x_2. Dans la suite, on note $\mathcal{S}$ la famille des fonctions sous-modulaires.

Malheureusement, bien que ces fonctions puissent être minimisées en temps polynomial lorsqu'aucune contrainte n'est présente [FUJISHIGE 2005], il n'en va pas de même lorsque les contraintes sont exprimées en DNNF, voir même en OBDD.

Proposition 4.26 *L'optimisation d'une fonction de $\mathcal{S}$ sous contraintes dans* OBDD$_<$ *est* NP-*difficile.*

Preuve *Soit $\psi = (l_{1,1} \vee l_{1,2}) \wedge \ldots \wedge (l_{n,1} \vee l_{n,2})$ une formule 2-*CNF*. On peut associer en temps polynomial à ψ le problème de minimisation de $f(\omega) = \Sigma_{i=1}^{n} \otimes (x_{i,1} + x_{i,2})$ sous contrainte* OBDD$_<$ *$\psi' = \wedge_{i=1}^{n} \wedge_{j=1}^{m} (x_{i,j} \Leftrightarrow n_{i,j}(x_i))$ où $n_{i,j}(x_i) = x_i$ quand $l_{i,j}$ est un littéral positif, $n_{i,j}(x_i) = \neg x_i$*

quand $l_{i,j}$ est un littéral négatif, $\otimes(x) = \min(1, x)$. La fonction f est une somme de fonctions de budget additives, qui sont sous-modulaires : f est donc aussi sous-modulaire. L'ordre $<$ choisi pour ψ' est supposé respecter $x_i < x_{i,j} < x_{i+1}$ pour que la construction de la contrainte se fasse en temps polynomial. Il existe une assignation complète des variables satisfaisant au moins k clauses de ψ si et seulement si le problème d'optimisation renvoie une solution optimale dont la valeur est au moins égale à k. Or, ceci correspond encore au problème 2-MIN-SAT.

En revanche, le salut quant à l'optimisation sans contraintes vient encore une fois du langage DNF. Pour démontrer cela, on montre dans un premier temps que la famille $\mathcal{S}$ des fonctions sous-modulaires est close par restriction.

Lemme 4.27 *La famille $\mathcal{S}$ des fonctions sous-modulaires est close par restriction.*

Preuve *Pour une fonction pseudo-booléenne $f : \{0,1\}^n \to \mathbb{R}$ et un ensemble de variables propositionnelles $U = \{u_1, \ldots, u_t\} \in$ PS, la dérivée de f par rapport à U d'ordre t est la fonction pseudo-booléenne définie par les règles inductives suivantes :*

$$
\begin{aligned}
\frac{\partial f}{\partial U}(x) &= f(x) && \textit{if } U = \emptyset \\
\frac{\partial f}{\partial U}(x) &= f(x \vee u_i^1) - f(x \wedge u_i^0) && \textit{if } U = \{u_i\} \\
\frac{\partial f}{\partial U}(x) &= \frac{\partial f}{\partial \{u_1\}} \left(\frac{\partial f}{\partial (U \setminus \{u_1\})}(x) \right) && \textit{sinon.}
\end{aligned}
$$

*où u_i^1 est le vecteur de taille n composé de 0 sauf pour la i-ème valeur qui est égale à 1 ; de manière symétrique, u_i^0 est le vecteur de taille n composé de 1 sauf pour la i-ème valeur qui est égale à 0. Or, une fonction objectif pseudo-booléenne f est sous-modulaire si et seulement si l'ensemble de ses dérivées d'ordre deux sont non positives [*FUJISHIGE *2005]. Par construction, on a $\frac{\partial f}{\partial U}(x \vee u) \leq 0$, ce qui complète la preuve.*

De ce fait, nous avons en notre possession tous les éléments nécessaires pour appliquer la proposition 4.12, et ainsi déduire que l'optimisation de fonctions sous-modulaires sous contraintes dans DNF peut s'effectuer en temps polynomial.

Proposition 4.28 *L'optimisation d'une fonction de $\mathcal{S}$ sous contraintes dans* DNF *est dans* P.

Preuve *Directement en considérant le lemme 4.27 ($\mathcal{S}$ close par restriction), le fait que l'on sache optimiser sans contrainte une fonction sous-modulaire [*FUJISHIGE *2005], et la proposition 4.12 qui dit que ces deux conditions sont suffisantes pour permettre l'optimisation de fonctions sous-modulaires sous contraintes dans* DNF *en temps polynomial.*

4.4 Conclusion du chapitre

Dans ce chapitre, nous avons étudié la complexité de l'optimisation sous contraintes DNNF ; les résultats obtenus sont résumés à la table 4.2.

Nous avons déterminé que lorsque le langage DNNF est employé pour représenter les contraintes d'un problème, l'optimisation d'une fonction linéaire sous ces contraintes s'effectue en temps polynomial. En revanche, nous avons aussi démontré que le résultat de traitabilité ne s'étend pas à des classes de fonctions à peine plus générales que les fonctions linéaires.

OPT	$\{\top\}$	DNF	DNNF, $\text{OBDD}_<$
fonctions linéaires	**dans** P	**dans** P	**dans** P
fonctions sous-modulaires	dans P	**dans** P	NP**-difficile**
fonctions quadratiques	NP-difficile	NP-difficile	NP-difficile
fonctions polynomiales	NP-difficile	NP-difficile	NP-difficile
fonctions OWA	NP**-difficile**	NP**-difficile**	NP**-difficile**

TABLE 4.2 – Résumé des résultats de complexité concernant l'optimisation d'un ensemble de familles de fonctions objectifs pseudo-booléennes. Les résultats en gras découlent de nos contributions.

Nous avons aussi démontré que le langage DNF pouvait être une cible intéressante pour l'optimisation de fonctions sous contraintes. Bien que celui-ci n'ait pas été considéré jusqu'à présent comme un langage de choix pour la compilation de formules propositionnelles, il s'avère qu'il peut être intéressant pour l'optimisation. En effet, étant donnée une famille de fonctions objectifs, il est suffisant de savoir que toute fonction de cette famille puisse être optimisée en temps polynomial sans contrainte et que cette famille soit close par restriction pour pouvoir optimiser toute fonction de cette famille sous contraintes DNF.

Cependant, beaucoup de problèmes combinatoires ne peuvent s'exprimer naturellement et de manière concise en utilisant le formalisme DNNF (et *a fortiori* en utilisant le formalisme DNF). Bien que nous ayons proposé un certain nombre de scénarios concrets pour lesquels la compilation de formule NNF en formule DNNF a un sens pour l'optimisation, il est aussi nécessaire d'être capable de procéder à l'optimisation de fonctions pseudo-booléennes lorsque le langage d'expression des contraintes n'est pas décomposable. C'est la problématique à laquelle nous nous intéressons dans les chapitres suivants.

Deuxième partie

Optimisation booléenne multiobjectif : résolution via SAT

Chapitre 5

Prouveurs pseudo-booléens

Sommaire

5.1 Approches complètes *vs.* incomplètes 84
5.2 De la résolution aux prouveurs CDCL 87
5.2.1 Principe de résolution de Robinson 87
5.2.2 Algorithme DP 90
5.2.3 Algorithme DPLL 93
5.2.4 Apprentissage de clauses : les prouveurs CDCL 95
5.3 Des prouveurs CDCL aux prouveurs SAT modernes 99
5.3.1 Heuristiques de choix de variable 99
5.3.2 Redémarrages 101
5.3.3 Nettoyage des clauses apprises 101
5.3.4 Structures « watched two literals » 103
5.4 Prouveurs et contraintes pseudo-booléennes 105
5.4.1 Contraintes pseudo-booléennes 105
5.4.2 Plans coupes et résolution généralisée 106
5.4.3 Intégration des contraintes pseudo-booléennes dans les prouveurs CDCL . 108
5.4.4 Encodages de contraintes de cardinalité par des clauses 110
5.5 Détection de contraintes de cardinalité 112
5.5.1 Détection statique de contraintes *AtMost*-1 et *AtMost*-2 113
5.5.2 Détection sémantique de contraintes AtMost-k 116
5.5.3 Détection sémantique : preprocessing *vs.* inprocessing 121
5.5.4 Résultats expérimentaux 122
5.5.5 Conclusion à propos des approches proposées 124
5.6 Conclusion du chapitre 124

Dans le premier chapitre, nous avons présenté le problème de satisfaction de formules CNF, ou problème SAT. Nous avons indiqué que ce problème est non seulement un problème de référence en ce qui concerne les problèmes de la classe NP du fait qu'il fut le premier problème montré NP-complet [COOK 1971], mais aussi qu'il est intéressant d'un point de vue pratique étant donné que le codage d'un problème de la classe NP est plutôt naturel quand on le représente à l'aide d'un ensemble de contraintes en formules CNF.

Ces observations ont conduit à un grand nombre de travaux [BIERE *et al.* 2009] lors des dernières décennies concernant l'implémentation de logiciels capables de résoudre ces problèmes de manière efficace en pratique ; ces logiciels sont appelés *prouveurs* SAT, ou bien encore *solveurs* SAT.

On peut séparer les prouveurs en deux grandes catégories : les prouveurs *complets*, qui répondent en temps fini, bien qu'exponentiel dans la taille de la formule en entrée, au problème de cohérence d'une formule CNF. L'autre catégorie est celle des prouveurs dits *incomplets*, dont le but est de tenter de déterminer en un temps court (temps fixé ou polynomial en nombre d'étapes dans la taille de la formule) si un modèle existe, mais pour lesquels il n'existe aucune garantie de réponse dans le sens où le programme peut cesser son exécution sans avoir fourni de résultat concernant la cohérence de la formule.

Dans ce chapitre, nous allons tout d'abord présenter les solveurs actuels, en débutant par les prouveurs incomplets. Nous présentons ensuite les approches complètes en procédant à l'historique des méthodes ayant conduit aux prouveurs efficaces d'aujourd'hui, les *solveurs CDCL* (pour *Conflict Driven Clause Learning*) ; l'idée de ces algorithmes est d'apprendre des contraintes au fil de la recherche d'une preuve de cohérence ou d'incohérence dans le but de diminuer le temps nécessaire à la résolution du problème.

Nous présentons ensuite une extension de ces solveurs modernes qui tire profit de contraintes plus générales que les clauses, les *contraintes de cardinalité*. Nous présentons finalement une nouvelle contribution concernant la recherche de contraintes de cardinalité « cachées » dans une formule CNF dans le but d'accélérer la vitesse de résolution de problèmes gérés par des prouveurs tirant parti de l'utilisation de ces contraintes plus générales que les clauses.

5.1 Approches complètes *vs.* incomplètes

Les prouveurs complets implémentent des algorithmes assurant, pour une formule CNF donnée, que l'on sache en temps fini si il existe ou non un modèle (le prouveur répond respectivement « SAT » ou « UNSAT »). Cependant, en considérant que les classes de complexité P et NP ne sont pas égales, on sait que ces algorithmes ne peuvent s'exécuter en temps polynomial. L'idée derrière les approches incomplètes est justement de proposer un prouveur qui pourrait s'exécuter en un temps plus faible que celui nécessaire pour un algorithme complet (temps fixé ou nombre d'étapes polynomial dans la taille de la formule), bien que ce programme n'assure pas de déterminer à coup sûr la réponse au problème posé. Pour cela, ces approches recherchent un modèle en effectuant un parcours efficace, bien que non exhaustif, de l'espace de recherche.

Dans la littérature, on trouve un certain nombre d'approches incomplètes, parmi lesquelles on peut citer les algorithmes « génétiques » [JONG & SPEARS 1989, HAO & DORNE 1994, HOLLAND 1975], la « survey propagation » [BRAUNSTEIN *et al.* 2005], et la technique la plus utilisée de nos jours, la recherche locale [LI *et al.* 2012, LI & LI 2012, BALINT & MANTHEY 2013] que nous présentons brièvement à titre d'exemple.

La procédure de la recherche locale débute par le choix aléatoire d'une interprétation (complète) de la formule. En partant de cette interprétation, le but est de procéder de proche en proche en sélectionnant des interprétations *voisines* (cette notion de voisinage est présentée dans le paragraphe suivant) qui falsifient de moins en moins de clauses. Dans le cas où on trouve une interprétation qui ne falsifie aucune

contrainte, cette interprétation est alors un modèle de la formule.

Les interprétations dites voisines d'une interprétation I sont les interprétations I' qui sont identiques à I à part pour un littéral l. Dit d'une autre manière, si l'interprétation I est définie par les littéraux $\{a_1, \ldots, a_k, l, b_1, \ldots, b_p\}$, alors l'interprétation I' définie par $\{a_1, \ldots, a_k, \neg l, b_1, \ldots, b_p\}$ est une interprétation voisine de I. Dans cet exemple, on dit que l'on est passé de l'interprétation I à l'interprétation I' en *flippant* le littéral l.

Dans l'algorithme de la recherche locale, on peut se trouver dans deux cas de figure lorsque l'on considère l'ensemble des interprétations I' voisines de l'interprétation courante I :

– soit une des interprétations voisines satisfait un plus grand nombre de contraintes que l'interprétation I ; dans ce cas, la nouvelle interprétation courante de l'algorithme est une des voisines de I qui minimise le nombre de contraintes falsifiées ; si ce nombre est nul, alors l'algorithme s'arrête : l'interprétation est un modèle de la formule ;
– soit il n'existe aucune interprétation voisine I' telle que le nombre de contraintes falsifiées est strictement inférieur : on se retrouve dans ce que l'on appelle un *minimum local*.

L'algorithme 5.1 décrit le processus de recherche locale, où la stratégie d'échappement des minima locaux consiste à partir d'une nouvelle interprétation choisie aléatoirement.

Algorithme 5.1 : Recherche Locale

Entrées : Une formule CNF Φ, un nombre de tests maximal M.
Sorties : `SAT` si l'algorithme a pu déterminer que la formule Φ est cohérente, `UNKNOWN` sinon.

```
1  tant que vrai faire
2  |  m ← 0;
3  |  I ← interprétationAléatoire(Φ);
4  |  MinimumLocal ← faux ;
5  |  tant que m < M ET NON MinimumLocal faire
6  |  |  m ← m + 1;
7  |  |  voisines ← voisinesDe(I);
8  |  |  I' ← minFalsifiées(voisines,Φ);
9  |  |  si minimumLocalAtteint(I,I') alors MinimumLocal ← vrai;
10 |  |  I ← I' ;
11 |  |  si falsifiées(I,Φ) = 0 alors retourner SAT ;
12 |  fin
13 |  si m = M alors retourner UNKNOWN;
14 fin
```

Exemple 5.1 (Recherche locale) *Soit la formule* CNF $\Phi = (a \vee b) \wedge (b \vee c) \wedge (a \vee c)$. *On définit la fonction f qui associe à toute interprétation le nombre de clauses de Φ qu'elle falsifie. On choisit comme interprétation de départ $\omega_0 = \{\neg a, \neg b, \neg c\}$; cette interprétation n'est pas modèle de la formule (on a $f(\omega_0) = 3$). On considère alors les trois interprétations voisines de ω_0 :*

$$\begin{array}{ccc} \omega_1 = \{a, \neg b, \neg c\} & \omega_2 = \{\neg a, b, \neg c\} & \omega_3 = \{\neg a, \neg b, c\} \\ f(\omega_1) = 1 & f(\omega_2) = 1 & f(\omega_3) = 1. \end{array}$$

Parmi les interprétations voisines de ω_0, on voit qu'il en existe au moins une qui falsifie moins de contraintes. On sélectionne alors une des interprétations voisines qui satisfait le plus de clauses, par exemple ω_1 : on flippe de ce fait la variable a. On s'intéresse ensuite aux interprétations voisines de ω_1 :

$$\begin{array}{ccc} \omega_{1,1} = \{\neg a, \neg b, \neg c\} & \omega_{1,2} = \{a, b, \neg c\} & \omega_{1,3} = \{a, \neg b, c\} \\ f(\omega_{1,1}) = 3 & f(\omega_{1,2}) = 0 & f(\omega_{1,3}) = 0. \end{array}$$

Encore une fois, on s'aperçoit que nous n'avons pas atteint de minimum local, puisqu'il existe des interprétations voisines pour lesquelles le nombre de clauses falsifiées est inférieur à 1. On remarque du même coup qu'il existe des interprétations qui satisfont toutes les clauses, et qui sont donc modèles, comme par exemple $\omega_{1,2}$.

Nous discutons maintenant d'un certain nombres d'approches ayant été proposées pour lutter contre les effets des minima locaux.

Exemple 5.2 (Minimum local) *Soit la formule* CNF $\Phi = (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c) \wedge (a \vee b) \wedge (a \vee c) \wedge (b \vee c)$ *et soit l'interprétation* $\omega_0 = \{a, b, \neg c\}$. *On a alors* $f(\omega_0) = 1$. *Or, il n'existe pas d'interprétation voisine qui falsifie moins de une contrainte :*

$$\begin{array}{ccc} \omega_1 = \{\neg a, b, \neg c\} & \omega_2 = \{a, \neg b, \neg c\} & \omega_3 = \{a, b, c\} \\ f(\omega_1) = 1 & f(\omega_2) = 1 & f(\omega_3) = 3. \end{array}$$

Nous avons donc atteint un minimum local. Il est à noter que pour cette formule, on serait tombé dans un minimum local peu importe l'interprétation initiale et la séquence de flips à laquelle on aurait procédé, puisque la formule est incohérente. De plus, puisqu'il n'existe aucune interprétation qui falsifie moins de contraintes que ce minimum local, celui-ci est qualifié de minimum global.

Les minima locaux bloquent notre algorithme, dans la mesure où on ne peut plus sélectionner une interprétation voisine telle que la solution s'améliore. De ce fait, il est nécessaire d'employer des techniques d'échappement, qui permettent de s'éloigner de l'espace de recherche courant. À cette fin, un certain nombre de stratégies ont été proposées.

Parmi les méthodes étudiées pour s'échapper des minima locaux, un certain nombre d'approches ajoutent une part d'aléatoire dans le choix d'un voisin, en forçant de temps à autre à ne pas faire un « coup optimal » lors du choix d'une interprétation voisine. Parmi ce groupe d'approches, on peut par exemple citer *Random Walk* [SELMAN & KAUTZ 1993], dans laquelle on effectue avec une probabilité p ($p \in]0, 1[$) un flip aléatoire parmi l'ensemble des variables référencées par les clauses falsifiées à la place du coup optimal. Cette méthode a été adaptée quelques années plus tard pour donner la stratégie *novelty* [MCALLESTER *et al.* 1997], dans laquelle l'étape de choix du littéral à flipper doit choisir deux des littéraux donnant les interprétations voisines qui falsifient le plus petit nombre de clauses. Ainsi, si le meilleur choix est de flipper l_1, alors si l_2 (le deuxième meilleur choix) a été flippé plus récemment que l_1, on flippe l_1. Dans le cas où c'est l_1 qui a été flippé le plus récemment, alors on flippe l_2 avec une probabilité p ($p \in]0, 1[$) et l_1 avec une probabilité $1 - p$. L'intérêt de cette approche, comparée à *Random Walk*, et que même lorsqu'un flip non optimal est effectué, ce flip est tout de même « satisfaisant ». L'approche *novelty* a elle-même été adaptée par la suite à de multiples reprises [MCALLESTER *et al.* 1997, HOOS & STÜTZLE 1999].

Une autre approche étudiée est l'approche dite *tabou* [GLOVER 1989, GLOVER 1990, MAZURE *et al.* 1997]. Cette approche se base sur une file de taille constante qui contient les derniers littéraux flippés. Ainsi, lors du choix de l'interprétation voisine, on ne considère pas les interprétations que l'on pourrait obtenir en flippant un littéral de la file tabou. Lorsque la variable à flipper est déterminée, on la met alors dans la file, et on défile un élément si la taille maximale associée à la file tabou a été dépassée. Cette approche est très efficace pour échapper aux minima locaux dans la mesure où la taille de cette file est adéquate à l'instance, chose qui est malheureusement difficile à déterminer. Les approches à base de liste tabou ont aussi été adaptées avec succès dans le cadre de l'optimisation : on peut notamment trouver des exemples pour la recherche de *maximum diversity problem* [WANG *et al.* 2014] ou de *maximal independent set* [JIN & HAO 2015].

Enfin, les approches dites de *recherche locale dynamique* associent un compteur à chacune des clauses, et utilisent ces compteurs pour échapper aux minima locaux [MORRIS 1993]. Lorsqu'une contrainte est falsifiée par une interprétation, on incrémente son compteur. Lors du choix d'une interprétation voisine, on ne regarde plus simplement le nombre de clauses falsifiées, mais on cherche l'interprétation voisine pour laquelle la somme des compteurs associés aux clauses falsifiées est minimale. Cette approche a été améliorée par la suite par différents travaux, une des stratégies de recherche locale dynamique les plus efficaces étant SAPS [HUTTER *et al.* 2002]. Les solveurs de recherche locale sont toujours étudiés et améliorés, comme en attestent les prouveurs récents tels que CDLS [AUDEMARD *et al.* 2009], Sattime [LI & LI 2012] et Sparrow [BALINT & MANTHEY 2013].

Bien que les stratégies d'échappement de minima locaux que nous venons de présenter aient grandement amélioré la recherche locale, il est un problème qu'elles ne pourront jamais régler de manière totale, qui est la preuve de l'incohérence d'une formule (bien que certains prouveurs basés sur la recherche locale puisse déterminer l'incohérence comme CLS [FANG & RUML 2004], GUNSAT [AUDEMARD & SIMON 2007] ou Ranger [PRESTWICH & LYNCE 2006], encore une fois de manière incomplète). Cependant, comme nous allons le voir par la suite, être capable de prouver l'incohérence est un point primordial lors de la recherche (complète) d'une solution optimale d'un problème d'optimisation en utilisant un prouveur SAT. De ce fait, nous allons maintenant étudier les approches permettant de détecter de manière systématique les formules pour lesquelles il n'existe aucun modèle : les approches complètes.

5.2 De la résolution aux prouveurs CDCL

Comme nous l'avons noté précédemment dans ce chapitre, les approches de résolution de problèmes de cohérence de formule CNF peuvent être séparées selon qu'elles soient complètes ou non. Dans cette section, nous nous intéressons aux approches complètes, c'est-à-dire celles qui déterminent à coup sûr, en temps fini, si une formule CNF est cohérente ou incohérente.

Notre présentation sera effectuée de manière chronologique, en décrivant les adaptations qui ont permis de passer des premiers solveurs complets, d'architectures plutôt simples, aux solveurs actuels, bien plus efficaces.

Comme nous allons le voir, les premières approches remarquables étaient des approches « syntaxiques », dans le sens où les algorithmes ne s'intéressaient qu'à la structure de la formule CNF considérée et non à la signification des variables et contraintes qu'elle contient. Historiquement, l'approche complète la plus ancienne à avoir des répercussions dans les algorithmes actuels est celle basée sur le principe de résolution de Robinson, que nous étudions ici.

5.2.1 Principe de résolution de Robinson [ROBINSON 1965]

L'approche de Robinson cherche à déterminer l'incohérence d'une formule en appliquant trois règles de transformation jusqu'à obtenir un des deux cas d'arrêt de l'algorithme.

La première de ces trois règles est la règle de *sous-sommation* aussi que l'on trouve aussi sous son nom anglais dans la littérature, à savoir règle de *subsumption*. Le but de cette règle est de retirer une clause s'il en existe une autre qui est plus forte, dans le sens où tous les cubes interdits par la première sont aussi interdits par la deuxième. Plus formellement, la définition de sous-sommation pour une clause peut être donnée de la façon suivante.

Définition 5.1 (Clause sous-sommée) *Soit c_1 et c_2 deux clauses. On dit que la clause c_1 sous-somme la clause c_2 si et seulement si l'ensemble des littéraux de c_2 inclut l'ensemble des littéraux de c_1.*

En partant de cette définition, il est aisé de se rendre compte que si c_1 sous-somme c_2 (on suppose que chaque littéral apparaît au plus une fois par clause), l'ensemble des modèles de c_1 est inclus dans

l'ensemble des modèles de c_2 (c_2 est une conséquence logique de c_1, noté $c_1 \models c_2$) : en effet, si c_1 est satisfaite, alors il existe au moins un de ses littéraux qui est évalué à vrai ; or, cela implique qu'un des littéraux de c_2 est lui aussi évalué à vrai (puisque les littéraux de c_1 sont aussi des littéraux de c_2), ce qui implique finalement que la clause c_2 est elle aussi satisfaite.

La deuxième règle du principe de Robinson est la règle de fusion. Cette règle vise à supprimer les occurrences redondantes d'un littéral dans une clause :

$$(a_1 \vee \ldots a_n \vee \mathbf{x} \vee b_1 \vee \ldots \vee b_p \vee \mathbf{x} \vee c_1 \vee \ldots \vee c_p)$$
$$\equiv$$
$$(a_1 \vee \ldots a_n \vee \mathbf{x} \vee b_1 \vee \ldots \vee b_p \vee c_1 \vee \ldots \vee c_p).$$

Il est évident qu'un littéral redondant peut être retiré d'une clause, puisque cela ne changera pas l'ensemble (au sens mathématique) des littéraux de la clause, et ne changera donc pas l'ensemble des interprétations qui satisfont cette clause. Notons toutefois que cette observation n'est pas valable pour certains types de contraintes plus générales que nous étudierons dans la suite de ce document, comme les contraintes de cardinalité.

La dernière règle du principe de Robinson est la règle de résolution. Cette règle va permettre de générer une nouvelle clause depuis deux clauses du problème qui contiennent chacune une des polarités d'une variable. Autrement dit, si on dispose des clauses $c_1 = (a \vee b_1 \vee \ldots \vee b_n)$ et $c_2 = (\neg a \vee c_1 \ldots \vee c_p)$, alors on voit que la variable a apparaît via son littéral positif dans c_1 et via son littéral négatif dans c_2. De ce fait, on peut conclure qu'il va falloir satisfaire au moins un des littéraux b_i $(i \in 1, \ldots, n)$ ou un des littéraux c_j $(j \in 1, \ldots, p)$ pour trouver un modèle de la formule Φ contenant les clauses c_1 et c_2. En effet, si tous les littéraux b_i et c_j sont falsifiés, alors on va pouvoir réduire c_1 et c_2 à $c'_1 = (a)$ et $c'_2 = (\neg a)$, ce qui implique que la formule n'est pas cohérente (il faudrait affecter la variable a aux deux valeurs de vérité en même temps pour espérer détecter un modèle, ce qui est impossible). De ce fait, en présence de c_1 et c_2, on peut déduire la clause $c_3 = (b_1 \vee \ldots \vee b_n \vee c_1 \ldots \vee c_p)$. Nous venons d'appliquer le principe de résolution avec la variable a pour déduire c_3 depuis c_1 et c_2. De façon formelle, la règle de résolution est définie de la manière suivante.

Définition 5.2 (Règle de résolution) *Soient $c_1 = (x \vee \beta_1)$ et $c_2 = (\neg x \vee \beta_2)$ deux clauses ayant en commun une variable x présente dans les deux clauses sous la forme de littéraux complémentaires, la clause $c = (\beta_1 \vee \beta_2)$ peut être obtenue par la suppression de toutes les occurrences du littéral x et $\neg x$ dans respectivement c_1 et c_2. Cette opération, notée $\eta[x, c_1, c_2]$, est appelée résolution et la clause produite est appelée résolvante de c_1 et c_2 sur x.*

Dans le cas où une résolvante contient deux littéraux opposés d'une même variable, la clause obtenue est alors tautologique, c'est-à-dire satisfaite pour toute interprétation ; elle peut de ce fait être retirée du problème.

Le principe de résolution de Robinson suit un algorithme très simple une fois ces trois règles définies. En effet, il suffit d'appliquer les règles tant que cela est possible, et tant qu'aucune clause de la formule n'est vide. Si une clause vide est déduite, alors on a déterminé que la formule était incohérente (on a déduit une contrainte qui ne peut être satisfaite). En revanche, s'il est impossible d'appliquer une des trois règles, alors on a prouvé qu'il existe un modèle pour la formule considérée (si une incohérence existe, le théorème de Herbrand indique que l'on peut nécessairement la déduire). Ce programme est décrit par l'algorithme 5.2.

Exemple 5.3 (Résolution de Robinson) *Considérons la formule $\Phi_1 = (a \vee b) \wedge (b \vee c) \wedge (a \vee c)$. On ne peut ni appliquer la règle de sous-sommation, ni la règle de fusion, ni la règle de résolution : cette formule est cohérente.*

Algorithme 5.2 : Résolution de Robinson

Entrées : Une formule CNF Φ.
Sorties : `SAT` si la formule Φ est cohérente, `UNSAT` sinon.

1 *règleAppliquée* $\leftarrow$ `vrai` ;
2 **tant que** *règleAppliquée* **faire**
3 $\quad \Phi' \leftarrow \Phi$;
4 $\quad$ **tant que** $\exists x$ *tel que* $\Phi' = (x \vee x \vee x_1 \vee \ldots \vee x_p) \wedge \alpha_1 \wedge \ldots \wedge \alpha_n$ **faire**
5 $\quad\quad \Phi' \leftarrow (x \vee x_1 \vee \ldots \vee x_p) \wedge \alpha_1 \wedge \ldots \wedge \alpha_n$;
6 $\quad$ **fin**
7 $\quad$ **tant que** $\Phi' = (x_1 \vee \ldots \vee x_p) \wedge (x_1 \vee \ldots \vee x_p \vee \ldots \vee x_k) \wedge \alpha_1 \wedge \ldots \wedge \alpha_n$ **faire**
8 $\quad\quad \Phi' \leftarrow (x_1 \vee \ldots \vee x_p) \wedge \alpha_1 \wedge \ldots \wedge \alpha_n$;
9 $\quad$ **fin**
10 $\quad$ **si** $\Phi' = (\neg x \vee \alpha_1) \wedge (x \vee \alpha_2) \wedge \ldots \wedge \alpha_n$ **alors**
11 $\quad\quad \Phi' \leftarrow (\alpha_1 \vee \alpha_2) \wedge \ldots \wedge \alpha_n$;
12 $\quad$ **fin**
13 $\quad$ **si** $\Phi = \Phi'$ **alors**
14 $\quad\quad$ *règleAppliquée* $\leftarrow$ `faux` ;
15 $\quad$ **sinon**
16 $\quad\quad \Phi \leftarrow \Phi'$;
17 $\quad$ **fin**
18 $\quad$ **si** $\Phi' = () \wedge \alpha_1 \ldots \wedge \alpha_n$ **alors retourner** `UNSAT` ;
19 **fin**
20 **retourner** `SAT` ;

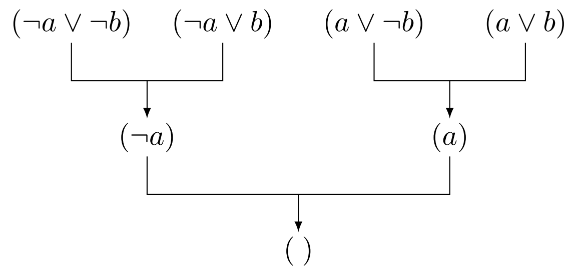


FIGURE 5.1 – Illustration d'un arbre de résolution pour une formule incohérente.

Considérons maintenant la formule $\Phi_2 = (\neg a \vee \neg b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (a \vee b)$. On peut déduire la clause $(\neg a)$ en effectuant la résolution avec la variable b depuis les clauses $(\neg a \vee \neg b)$ et $(\neg a \vee b)$, et ainsi retirer ces deux dernières puisqu'elles sont sous-sommées par la clause déduite. De plus, on peut aussi déduire la clause (a) en appliquant la résolution avec la variable b depuis les clauses $(a \vee \neg b)$ et $(a \vee b)$ (que l'on retire puisqu'elles sont maintenant sous-sommées par la clause (a)). Finalement, on peut déduire la clause vide depuis les clauses (a) et $(\neg a)$ en appliquant la règle de résolution avec la variable a, et ainsi prouver que la formule Φ_2 est incohérente.

Toute clause générée par l'algorithme peut être représentée par un arbre de résolution. C'est un arbre dont les feuilles sont représentées en haut et la racine en bas, où les feuilles sont des clauses du problème initial, et où tout nœud non terminal représente une clause obtenue par la règle de la résolution à partir des deux clauses qui sont représentées par les fils du nœud considéré (on applique la règle de fusion sur cette clause si nécessaire). En créant un arbre de résolution pour la formule Φ_2 de l'exemple précédent, on peut ainsi représenter de manière visuelle la preuve de l'incohérence de la formule. Un tel arbre correspondant à l'exemple précédent est donné à la figure 5.1.

Il est important de noter que cet algorithme termine toujours en temps fini, en ayant déterminé si la formule initiale est cohérente ou non : c'est bien une approche complète pour déterminer la cohérence ou l'incohérence d'une formule CNF. On dit aussi que cette méthode est complète pour la réfutation de formules CNF.

Cependant, bien que cet algorithme soit complet, il faut noter qu'il est de manière générale bien peu efficace du fait du nombre exponentiel de clauses qu'il peut générer (il requiert dans le pire des cas un espace exponentiel dans la taille de la formule initiale), bien que les algorithmes basés sur la règle de la résolution puissent être efficaces dans certains cas particuliers [CHATALIC & SIMON 2000]. Puisque c'est le nombre exponentiel de résolvantes qui pose problème, des améliorations de cet algorithme ont été proposées dans le but de réduire ce nombre lors de la recherche, dont le célèbre algorithme de Davis et Putnam.

5.2.2 Algorithme DP [DAVIS & PUTNAM 1960]

L'algorithme de Davis et Putnam, nommé algorithme DP dans la littérature, est un raffinement de l'approche de Robinson dans la mesure où il génère de manière générale moins de résolvantes. L'idée de cet algorithme est de retirer à chaque étape une variable x du problème en effectuant toute les résolutions possibles sur cette variable, ce qui permet ensuite de supprimer toutes les clauses du problème dans lesquelles la variable x apparaît (la formule initiale et la formule générée sont équivalentes pour la cohérence).

Exemple 5.4 (Élimination de variable par énumération des résolvantes) *Soit une formule* CNF *:*

$$\Phi = (a \vee b) \wedge (a \vee c \vee d) \wedge (\neg a \vee \neg b \vee d) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c).$$

On utilise la règle de résolution avec la variable a pour toute paire de clauses telles que le littéral a apparaît dans une clause et le littéral $\neg a$ apparaît dans l'autre. On obtient les résolvantes $(b \vee \neg b \vee d)$, $(b \vee \neg c)$, $(\neg b \vee c \vee d)$ et $(c \vee \neg c \vee d)$; on remarque par ailleurs que la première et la dernière sont tautologiques (toujours vraies) puisqu'elles contiennent, toutes les deux, deux littéraux opposés, et que l'on peut donc les supprimer. On peut alors remplacer dans Φ les clauses contenant la variable a par les deux résolvantes restantes pour obtenir la formule Φ', qui est équivalente à ϕ en ce qui concerne la cohérence de la formule :

$$\Phi' = (b \vee \neg c) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee \neg c).$$

L'algorithme DP se termine lorsque un des deux cas d'arrêts prévus apparaît :
– comme dans le cas de l'algorithme de Robinson, si la clause vide est générée alors la formule est prouvée incohérente, et l'algorithme peut s'arrêter ;
– si la formule est vide (c'est-à-dire qu'elle ne contient plus aucune clause), alors il n'existe plus aucune contrainte et la formule est de ce fait cohérente.

Il est aisé de vérifier que l'algorithme s'arrête forcément en temps fini : puisque l'on enlève une variable du problème à chaque étape, alors la formule ne comptera nécessairement plus aucun littéral à partir d'un certain moment (au maximum, au bout d'un nombre d'étapes égal au nombre de variables de la formule initiale) : dans ce cas, soit elle ne contiendra plus aucune clause, soit elle sera composée de clauses vides.

Dans l'algorithme DP, sont aussi présentes deux règles dont le but est d'améliorer le temps de calcul et l'espace nécessaire au déroulement de l'algorithme en évitant de générer des résolvantes inutiles : la règle d'élimination des littéraux purs et la règle de propagation unitaire.

La règle des littéraux purs s'applique lorsqu'une variable apparaît dans une formule CNF sous une unique polarité (soit la variable x n'apparaît que sous la forme du littéral x, soit elle n'apparaît que sous la forme du littéral $\neg x$). On choisit alors d'éliminer la variable x en énumérant ses résolvantes. Or, l'ensemble des résolvantes générées à partir de la variable x est vide (on ne peut pas appliquer la règle de la résolution avec la variable x étant donné qu'elle n'apparaît que sous un unique littéral). De ce fait, éliminer la variable x en énumérant ses résolvantes revient simplement dans ce cas à supprimer les clauses qui contiennent cette variable.

Exemple 5.5 (Élimination des littéraux purs) *Soit la formule* CNF *de l'exemple précédent :*

$$\Phi = (a \vee b) \wedge (a \vee c \vee d) \wedge (\neg a \vee \neg b \vee d) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c).$$

Dans cette formule, on note que la variable d n'apparaît que sous son littéral positif. On choisit alors d'éliminer la variable d en ajoutant l'ensemble des résolvantes générées à partir de cette variable. Or, cet ensemble est vide puisque la variable d n'apparaît que sous un littéral. De ce fait, cette phase d'élimination revient simplement à supprimer les clauses contenant la variable d, et ainsi obtenir la formule Φ', équivalente à Φ pour la cohérence :

$$\Phi' = (a \vee b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c).$$

La seconde règle, bien plus fondamentale de par l'importance qu'elle a aujourd'hui encore dans les prouveurs modernes, est la règle de la propagation unitaire. Cette règle s'applique lorsqu'une clause ne contient plus qu'un littéral x. Dans ce cas, il est possible de retirer toutes les clauses contenant ce littéral x (en pratique, cela ne concerne que la clause unitaire puisqu'elle sous-somme toute clause contenant le littéral x) et de retirer les littéraux opposés $\neg x$ dans les autres clauses. Ceci s'explique par le fait que l'élimination de cette variable par l'énumération des résolvantes génère exactement une clause c' par clause c contenant le littéral $\neg x$ (puisqu'il n'existe qu'une clause contenant le littéral x), et que cette

clause c' correspond à la clause c à laquelle on a retiré le littéral $\neg x$. De plus, étant donné que cette phase peut réduire la taille d'au moins une clause en la remplaçant par elle-même privée de $\neg x$, elle peut impliquer de nouvelles phases de propagation unitaire, et ainsi créer une cascade de propagations unitaires.

Exemple 5.6 (Propagation unitaire) *Soit la formule* CNF *:*

$$\Phi_1 = (a) \wedge (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee d).$$

On remarque que Φ_1 contient la clause unitaire (a) : on peut alors retirer toutes les clauses contenant le littéral a et retirer toutes les occurrences du littéral $\neg a$; on obtient alors une formule équi-satisfiable Φ_2 :

$$(b) \wedge (c) \wedge (\neg b \vee \neg c \vee d).$$

Cette première propagation a créé deux nouvelles clauses unitaires. On applique successivement les règles précédentes pour les deux littéraux concernés, et on obtient alors la formule CNF *Φ_3 :*

$$(d).$$

Finalement, en appliquant une nouvelles fois ces règles pour le littéral unitaire d, on obtient une formule CNF *qui ne contient plus aucune contrainte, et qui est donc cohérente.*

En ce qui concerne l'algorithme DP à proprement parler, on applique les règles d'élimination des littéraux purs et de propagation unitaire lorsque c'est possible. Lorsque ces règles ne sont pas applicables, on applique la règle de la résolution pour éliminer une variable. Cette succession d'étapes est appliquée jusqu'à obtenir un cas d'arrêt, c'est-à-dire une clause vide ou une formule ne contenant aucune clause. L'algorithme DP est décrit par l'algorithme 5.3.

Algorithme 5.3 : Algorithme DP

Entrées : Une formule CNF Φ.
Sorties : `SAT` si la formule Φ est cohérente, `UNSAT` sinon.

```
1  tant que vrai faire
2  |  Φ ← appliquerRèglePropagationUnitaire(Φ) ;
3  |  Φ ← appliquerRègleLittérauxPurs(Φ) ;
4  |  si contientClauseVide(Φ) alors retourner UNSAT ;
5  |  si neContientPasDeClauses(Φ) alors retourner SAT ;
6  |  v ← sélectionnerVariable(Φ) ;
7  |  résolvantes ← ensembleRésolvantes(Φ,v) ;
8  |  Φ ← retirerClausesContenantVariable(Φ,v) ;
9  |  Φ ← Φ ∪ résolvantes ;
10 fin
```

Exemple 5.7 (Algorithme DP) *On considère la formule* CNF *Φ_1 définie par :*

$$\Phi_1 = (\neg a \vee b) \wedge (a \vee b) \wedge (a \vee c \vee d) \wedge (\neg a \vee \neg b \vee d) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c) \wedge (b \vee c).$$

On élimine la variable pure d :

$$\Phi_2 = (\neg a \vee b) \wedge (a \vee b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c) \wedge (b \vee c).$$

En l'absence de clause unitaire et de littéral pur, on enlève une variable du problème, ici a, en remplaçant les clauses qui la contiennent par les résolvantes issues de toute déduction possible par une résolution sur la variable a :

$$\Phi_3 = (b) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c) \wedge (b \vee c).$$

Aucun littéral pur n'apparaît dans la formule Φ_3. En revanche, il existe une clause unitaire : la clause (b). On applique alors les règles de la propagation unitaire pour le littéral b :

$$\Phi_4 = (\neg c).$$

Une nouvelle clause unitaire apparaît : la clause $(\neg c)$. En appliquant les règles de propagation pour le littéral unitaire $\neg c$, la dernière contrainte de la formule Φ_4 disparaît : la formule devient donc vide de toute contrainte. Ce cas d'arrêt implique que la formule Φ_1 initiale est cohérente.

Bien que cet algorithme ait été proposé dans le but d'améliorer l'approche de Robinson en guidant les étapes de résolution, il s'agit encore une fois d'un algorithme requérant dans le pire des cas un espace exponentiel dans la taille de la formule initiale. Or, la théorie de la complexité ainsi que les résultats de Cook [COOK 1971] nous apprennent que le problème de cohérence d'une formule CNF est NP-complet et qu'il existe donc un algorithme n'ayant besoin que d'un espace polynomial pour s'exécuter.

Nous présentons maintenant l'algorithme DPLL, une approche « sémantique » complète dérivée de l'algorithme DP, qui s'exécute en espace polynomial.

5.2.3 Algorithme DPLL [DAVIS *et al.* 1962]

Deux ans après l'algorithme DP, l'algorithme DPLL est proposé. C'est une évolution de l'algorithme DP dans laquelle l'espace mémoire requis n'est plus exponentiel, mais polynomial dans la taille de la formule.

Cet algorithme emprunte à l'algorithme DP ses cas d'arrêt (clause vide, ou formule ne contenant aucune contrainte), ses règles d'élimination des littéraux purs et de propagation unitaire, ainsi que la volonté de supprimer une variable du problème lorsque aucune des deux règles citées ci-dessus n'est applicable. En revanche, la façon dont l'algorithme procède pour « supprimer » une variable est tout à fait différente.

La procédure permettant de retirer une variable du problème est issue de la méthode de Quine [QUINE 1950] et est nommée *séparation*. Cette séparation est basée sur un constat simple : soit Φ une formule et x une variable de Φ ; Φ est cohérente si et seulement si $\Phi_{|x} \wedge x$ ou $\Phi_{|\neg x} \wedge \neg x$ est cohérente. De ce fait, au moment de la séparation, on va par exemple commencer par vérifier si $\Phi_{|\neg x}$ (Φ conditionnée par $\neg x$) est cohérente ; si oui, alors Φ est cohérente. En revanche, si $\Phi_{|\neg x}$ n'est pas cohérente, on va alors tester la cohérence de $\Phi_{|x}$, qui sera alors équivalente à la cohérence de Φ.

En appliquant ce procédé de séparation de manière récursive, on voit que l'on va potentiellement tester l'ensemble des interprétations possibles, mais en un espace polynomial. En effet, ce calcul récursif correspond au parcours en profondeur d'un arbre binaire, où chaque nœud non terminal correspond à une variable et admet deux fils : un pour le littéral négatif, un pour le littéral positif.

Cependant, dans le cas général, cet algorithme élague l'arbre de recherche de manière à ne pas visiter des branches inutiles. En effet, dans le cas où un littéral pur est présent, on sait qu'il est suffisant de tester la cohérence de la formule après propagation de ce littéral pur pour tester la cohérence de la formule. De ce fait, en supposant une formule Φ et un littéral pur x, on ne va pas effectuer l'opération de séparation sur x à proprement parler (on ne visite que le fils du nœud correspondant au littéral pur ; l'autre branche est dite *fermée*). De la même façon, on ferme toute branche correspondant à un littéral dont le littéral opposé apparaît dans une clause unitaire.

Notons que dans le cadre d'une méthode de résolution « sémantique », les règles appliquées lors de la présence d'une clause unitaire peuvent être expliquées autrement que dans le cas d'un algorithme « syntaxique ». Ici, plutôt que de justifier la propagation unitaire de manière « syntaxique » en s'appuyant sur la règle de résolution, on peut simplement réagir sur le sens de la formule considérée et sur les affectations qui peuvent ou non conduire à découvrir une interprétation qui satisfait ou non une formule. En effet, d'un point de vue sémantique, le problème SAT est résolu si et seulement si il existe un littéral évalué à vrai dans chacune des clauses par une interprétation. Or, si on considère une clause unitaire, il existe une unique façon de la satisfaire, qui est de faire en sorte que le littéral unitaire x de cette clause soit évalué à vrai. De ce fait, il ne sert à rien de considérer l'arbre de recherche découlant du choix impliquant l'évaluation de x à faux, qui conduit nécessairement à un conflit ; ceci implique qu'en propageant le littéral x, on s'assure de considérer le sous-arbre qui contient tous les modèles potentiels ; cela va de plus permettre de ne plus considérer les clauses contenant le littéral x et va empêcher les clauses contenant le littéral opposé d'être satisfaites grâce à lui, exactement comme si ce littéral n'était pas présent. On arrive alors aux mêmes implications que celles de la propagation unitaire définie de manière « syntaxique » : on peut oublier les clauses contenant x, et aussi ne plus considérer le littéral opposé, exactement comme si on remplaçait chacune des clauses le contenant par une version identique à l'exception près qu'elle ne contiendrait plus ce littéral opposé.

Finalement, l'algorithme DPLL peut être présenté très simplement de manière récursive, comme le montre l'algorithme 5.4.

Algorithme 5.4 : Algorithme DPLL

Entrées : Une formule CNF Φ.
Sorties : `SAT` si la formule Φ est cohérente, `UNSAT` sinon.

1 $\Phi \leftarrow$ `appliquerRèglePropagationUnitaire(`Φ`)` ;
2 $\Phi \leftarrow$ `appliquerRègleLittérauxPurs(`Φ`)` ;
3 **si** `contientClauseVide(`Φ`)` **alors retourner** `UNSAT` ;
4 **si** `neContientPasDeClauses(`Φ`)` **alors retourner** `SAT` ;
5 *v* $\leftarrow$ `sélectionnerVariable(`Φ`)` ;
6 *res* $\leftarrow$ `DPLL(`$\Phi_{|\neg v}$`)` ;
7 **si** *res* $=$ `SAT` **alors**
8 | **retourner** `SAT`
9 **fin**
10 **retourner** `DPLL(`$\Phi_{|v}$`)` ;

L'algorithme DPLL admet aussi un comportement très intéressant dans la mesure où, lorsqu'une formule est cohérente, il est capable de retourner un modèle en plus de déterminer qu'il en existe un. Il suffit pour cela d'adapter l'algorithme afin qu'il retourne les choix qu'il a faits lors des étapes de séparation lorsque la cohérence a été déterminée.

Exemple 5.8 (Algorithme DPLL) *La figure 5.2 illustre le déroulement de l'algorithme DPLL sur la formule* $\Phi = (\neg a \vee b) \wedge (a \vee b) \wedge (\neg b \vee c) \wedge (a \vee \neg b \vee \neg c)$.

On choisit dans un premier temps d'affecter la variable a *à faux, c'est-à-dire de propager le littéral* $\neg a$*. Cela crée la clause unitaire* (b) *; de ce fait, on va propager* b *et fermer la branche correspondante à* $\neg b$*. On se retrouve alors avec la formule* $(c) \wedge (\neg c)$*. On propage alors le littéral unitaire* c *et on ferme la branche correspondante à* $\neg c$ *(notons que l'on aurait pu de la même façon propager* $\neg c$*, qui est lui aussi unitaire). On tombe alors sur un conflit : la clause* $(\neg c)$ *devient vide après propagation du littéral* c*.*

Puisque nous avons vérifié ou fermé toute les branches depuis que nous avons propagé le littéral

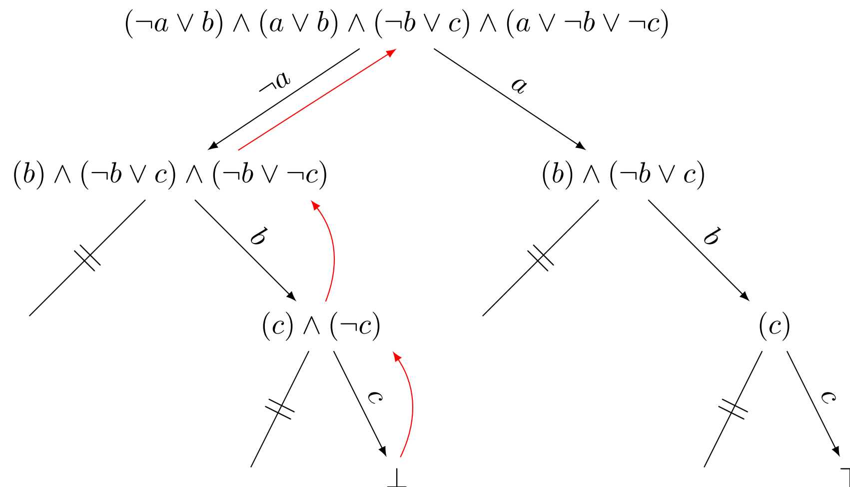


FIGURE 5.2 – Illustration de l'algorithme DPLL. Les flèches barrées sont des branches fermées (propagation unitaire), les flèches rouges représentent des retours-arrière.

$\neg a$, on sait qu'il n'existe pas de modèle pour la formule $\Phi_{|\neg a}$. On revient alors à la racine (on nomme retour-arrière*, ou* backtrack*, le fait de remonter dans l'arbre de recherche) et on propage le littéral a. La formule se résume alors à la* CNF *$(b) \vee (\neg b \vee c)$. Une fois propagé le littéral b, notre formule ne contient plus que la clause (c), ce qui permet de déduire la cohérence de la formule une fois la dernière application de la règle de propagation unitaire effectuée.*

De plus, au moment où nous avons démontré la cohérence de la formule, nous avions effectué les propagations des littéraux a et b et c. On peut ainsi conclure que le cube $a \wedge b \wedge c$ est un impliquant de la formule Φ.

La technique de *backtrack* utilisée dans l'algorithme DPLL n'est cependant pas optimale dans la mesure où lorsqu'un conflit est obtenu, elle ne procède à aucune analyse des causes de ce conflit. De ce fait, elle n'annule que le dernier littéral choisi alors que l'affectation ayant conduit au conflit peut être antérieure : le *backtrack* est dit chronologique. De plus, le *backtrack* de DPLL n'empêche en rien qu'un conflit ayant la même cause se produise à nouveau plus tard dans la recherche de la solution. Dans la section suivante, nous présentons l'algorithme CDCL, qui permet en analysant les causes d'un conflit de s'affranchir de ces deux problèmes.

5.2.4 Apprentissage de clauses : les prouveurs CDCL

Afin de montrer les inconvénients du *backtrack* chronologique de l'algorithme DPLL et de présenter l'approche CDCL (*Conflict Driven Clause Learning*, terme apparu dans [LAWRENCE 2004]), nous allons dans un premier temps définir un certain nombre de notions. Dans cette section, nous omettons la règle des littéraux purs décrite précédemment. Nous verrons par la suite que cette règle n'est plus utilisée dans les solveurs actuels, dans la mesure où elle ne permet pas d'employer des structures de données particulières qui ont apporté des gains très importants dans les performances des prouveurs contemporains.

Nous présentons dans un premier temps la notion de *niveau de décision*, qui permet de noter à quel étape de l'algorithme DPLL ou CDCL un littéral est affecté, que ce soit par un choix ou une application de la propagation unitaire.

Définition 5.3 (Niveau de décision) *Un littéral x est au niveau de décision n si ce littéral est le n-ième littéral affecté (hors propagation unitaire ; ce littéral est alors appelé* littéral décision*) ou si ce littéral a été propagé (par la propagation unitaire) suite à la n-ième affectation de littéral. Les littéraux propagés avant la première décision (qui existent seulement si la formule contient des clauses unitaires) appartiennent au niveau de décision 0.*

Maintenant que nous avons défini cette mesure, nous pouvons décrire le déroulement des algorithmes DPLL et CDCL (qui sont tous deux des algorithmes de recherche en profondeur) en utilisant les *séquences de décisions-propagations*.

Définition 5.4 (Séquence de décisions-propagations) *Soit une séquence de décisions $\langle x_1, \dots, x_n \rangle$, et $y_{0,1}, \dots, y_{0,p_0}$ les propagations au niveau de décision 0. On note $y_{i,1}, \dots, y_{i,p_i}$ les littéraux propagés après l'affectation du littéral x_i. La séquence de décisions-propagations de $\langle x_1, \dots, x_n \rangle$ est*

$$\{\langle y^0_{0,1}, \dots, y^0_{0,p_0} \rangle, \langle (x^1_1), y^1_{1,1}, \dots, y^1_{1,p_i} \rangle, \dots, \langle (x^n_n), y^n_{n,1}, \dots, y^n_{n,p_n} \rangle\},$$

où x^j_i désigne le littéral x_i et indique qu'il apparaît au niveau de décision j.

L'exemple 5.9 présente une séquence de décisions-propagations pour une formule CNF et une séquence de décisions donnée.

Exemple 5.9 (Séquence de décisions-propagations) *Soit la formule* CNF Φ *composée des clauses :*

$$\begin{aligned}
\alpha_1 &= (x_1 \vee \neg x_2)\\
\alpha_2 &= (\neg x_2 \vee \neg x_3 \vee \neg x_4)\\
\alpha_3 &= (x_3 \vee x_5 \vee x_6)\\
\alpha_4 &= (\neg x_4 \vee \neg x_6 \vee x_9)\\
\alpha_5 &= (\neg x_6 \vee x_7)\\
\alpha_6 &= (\neg x_6 \vee \neg x_8)\\
\alpha_7 &= (\neg x_7 \vee x_8 \vee x_{10})\\
\alpha_8 &= (\neg x_9 \vee \neg x_{10})\\
\alpha_9 &= (\neg x_{11} \vee x_{12})\\
\alpha_{10} &= (\neg x_{11} \vee \neg x_{12})
\end{aligned}$$

La séquence de décisions-propagations $\mathcal{I} = \{\langle (x^1_2), x^1_1 \rangle, \langle (x^2_4), \neg x^2_3 \rangle, \langle (x^3_{11}), \neg x^3_{12} \rangle,$ $\langle (\neg x^4_5), x^4_6, x^4_7, \neg x^4_8, x^4_{10}, x^4_9, \neg x^4_{10} \rangle\}$ est obtenue par la séquence de décisions $\langle x_2, x_4, x_{11}, \neg x_5 \rangle$.

Dans l'exemple 5.9, la séquence de décisions ayant produit la séquence de décisions-propagations (que nous nommerons aussi interprétation partielle dans la suite) $\mathcal{I} = \{\langle (x^1_2), x^1_1 \rangle, \langle (x^2_4), \neg x^2_3 \rangle,$ $\langle (x^3_{11}), \neg x^3_{12} \rangle, \langle (\neg x^4_5), x^4_6, x^4_7, \neg x^4_8, x^4_{10}, x^4_9, \neg x^4_{10} \rangle\}$ conduit à un conflit, puisqu'elle propage à la fois le littéral x_{10} et le littéral $\neg x_{10}$. Cette propagation des deux littéraux opposés d'une variable, en plus d'entrer à l'encontre du fait qu'une variable ne puisse avoir qu'une unique valeur de vérité (*vrai* ou *faux* dans notre cas), génère en fait une clause vide lors de l'application de l'algorithme DPLL. En effet, le fait de pouvoir propager les deux littéraux implique qu'une clause contenant chacun de ces littéraux ne contient plus qu'un littéral, celui de la variable x_{10}. Cela implique qu'en choisissant un des littéraux pour la propagation unitaire, le littéral opposé va disparaître dans la clause qui n'a pas encore été considérée, et ainsi rendre cette clause vide.

Comme nous le verrons par la suite, une explication de l'obtention de ce conflit est que les littéraux x_4 et x_6 sont tous les deux satisfaits (on dit que la clause $\neg x_4 \vee \neg x_6$ est un *nogood*), puisque la propagation de ces deux littéraux implique une cascade de propagations unitaires entraînant les propagations des

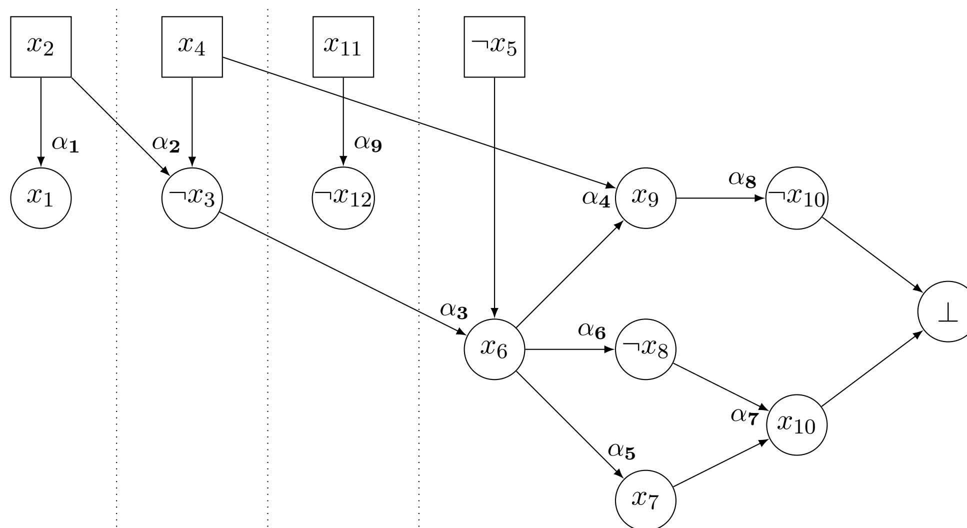


FIGURE 5.3 – Exemple de graphe d'implication obtenu à partir de l'interprétation partielle de l'exemple 5.9. L'interprétation partielle falsifie la formule car elle falsifie la clause α_8.

littéraux x_{10} et $\neg x_{10}$. De plus, on sait que x_4 et x_6 sont propagés en conséquence des décisions prises (x_2, x_4, x_{11} et $\neg x_5$), mais il est possible qu'ils soient aussi propagés par d'autres séquences de décision. De ce fait, on considérant $\neg x_4 \vee \neg x_6$ comme *nogood* plutôt que $\neg x_2 \vee \neg x_4 \vee \neg x_{11} \vee x_5$, on s'assure non seulement de ne pas pouvoir affecter à l'avenir les littéraux de la décision ayant conduit au conflit courant, mais va aussi empêcher de considérer toutes les interprétations qui ne satisfont pas ce *nogood*. De plus, si on avait considéré le *nogood* issu de la séquence de décisions, on aurait ainsi tenu compte de x_{11}, qui est nullement la cause du conflit étant donné que les variables x_{11} et x_{12} font partie d'un problème « à part », dans la mesure où les contraintes les concernant ne concernent pas les autre variables du problème.

De ce fait, en considérant le *nogood* $\neg x_4 \vee \neg x_6$, on implique que lors de l'affectation de la variable x_4 à vrai, il aurait fallu propager le littéral $\neg x_6$. Or, en suivant l'algorithme DPLL, on aurait simplement défait la dernière décision (ici, affecter x_5 à faux) lorsque le conflit serait survenu, et on aurait affecté x_5 à vrai, ce qui n'empêche pas un conflit basé sur le même *nogood* de se produire à nouveau. En partant de ce constat, on voit que l'analyse des sources d'un conflit obtenu peut améliorer le temps de résolution d'un problème en empêchant de parcourir des branches de l'arbre de recherche aboutissant à des conflits qui pourraient être anticipés. C'est l'intégration de cette analyse de conflits qui a donné naissance aux prouveurs actuels, les solveurs CDCL.

L'analyse d'un conflit se base sur un graphe dérivé de la séquence de décisions-propagations ayant conduit au conflit, en ajoutant une information pour chaque propagation, à savoir la clause qui est responsable de cette propagation. Ce graphe est nommé *graphe d'implication*.

Définition 5.5 (Graphe d'implication) *Soit une séquence de décisions-propagations $\mathcal{I}$. Le graphe d'implication de $\mathcal{I}$ est un graphe orienté dans lequel tous les littéraux de $\mathcal{I}$ sont représentés par un nœud et où un arc (x, y) étiqueté α apparaît pour chaque propagation unitaire par la clause α d'un littéral y lors de l'affectation ou la propagation d'un littéral x. Si deux littéraux contradictoires sont propagés, on ajoute un nœud $\bot$ et un arc entre chacun des littéraux contradictoires et le nouveau nœud $\bot$.*

Exemple 5.10 (Graphe d'implication) *La figure 5.3 représente le graphe d'implication de la séquence de décisions-propagations de l'exemple 5.9.*

Dans le graphe d'implication, un certain nombre de nœuds appartenant au dernier niveau de décision

sont remarquables, dans la mesure où ils « dominent » le conflit. Plus formellement, un nœud UIP (ou un littéral UIP, en référence au littéral qui étiquette un nœud UIP), est défini de la façon suivante.

Définition 5.6 (UIP (*Unique Implication Point*)) *Soit $\mathcal{G}$ le graphe d'implication associé à une interprétation partielle conflictuelle. Soit $x \in \mathcal{G}$ le nœud associé au littéral choisi au dernier niveau de décision, et z le nœud de conflit (étiqueté par $\perp$ dans le graphe de la figure 5.3). Un nœud y est un nœud UIP si et seulement si tous les chemins de x vers z passent par le nœud y.*

Dans le graphe de la figure 5.3, il existe deux littéraux UIP : $\neg x_5$ et x_6. Les littéraux UIP sont numérotés de 1 à p, où le 1-UIP (ou *first UIP*) est le plus proche du nœud conflit, quand le dernier (p-UIP) est le nœud décision du dernier niveau de décision.

On utilise ensuite ce graphe dans le but de rechercher un *nogood*. Pour cela, on considère tout d'abord les deux clauses à partir desquelles ont été propagés les deux littéraux contradictoires, puis on applique la règle de résolution entre elles. On applique alors la règle de résolution pour éliminer chacun des littéraux du dernier niveau de décision (en effectuant la résolution entre la clause courante et la clause ayant impliqué le littéral à supprimer), à l'exception du littéral 1-UIP, dont la nature implique qu'il fera nécessairement son apparition lors des résolutions successives. On obtient finalement une clause conséquence de la formule (puisque obtenue par la règle de résolution appliquée entre des clauses de la formule) qui est falsifiée par l'interprétation courante : cette clause est un *nogood*, dont un seul littéral appartient au dernier niveau de décision (le littéral 1-UIP).

Il est à noter que l'on peut appliquer la règle de résolution sur la variable du nœud 1-UIP dans le but de faire apparaître d'autres littéraux UIP. En pratique, cela ne permet cependant pas aux prouveurs récents de gagner du temps pour le test de cohérence d'une formule.

Exemple 5.11 (Calcul d'un *nogood*) *On calcule un* nogood *à partir du graphe d'implication de la figure 5.3, dont le conflit porte sur la variable x_{10} et dont le littéral 1-UIP est le littéral $\neg x_6$:*

$$\begin{aligned}
\sigma_1 = \eta[x_{10}, \alpha_7, \alpha_8] &\quad = \quad \neg x_7 \vee x_8 \vee \neg x_9 \\
\sigma_2 = \eta[x_9, \sigma_1, \alpha_4] &\quad = \quad \neg x_4 \vee \neg x_6 \vee \neg x_7 \vee x_8 \\
\sigma_3 = \eta[x_7, \sigma_2, \alpha_5] &\quad = \quad \neg x_4 \vee \neg x_6 \vee x_8 \\
\sigma_4 = \eta[x_8, \sigma_3, \alpha_6] &\quad = \quad \neg x_4 \vee \neg x_6.
\end{aligned}$$

Dans la mesure où ce *nogood* ne contient qu'un unique littéral du dernier niveau de décision, celui-ci nous apprend qu'un littéral (le first-UIP) aurait dû être propagé avant de prendre la dernière décision. Ainsi, si on décide d'ajouter le *nogood* à l'ensemble de clauses du problème (la formule obtenue est équivalente), il faut alors remonter au niveau de décision auquel la nouvelle clause aurait dû propager son littéral (le niveau auquel la clause était *assertive*). Puisque ce niveau de décision peut être situé bien en amont de la recherche, on dit dans ce cas que le retour-arrière est *non chronologique* (on parle de *backjumping* et non plus de *backtracking*).

La conjonction de l'apprentissage de clauses et du *backjumping* intégrés à l'algorithme DPLL permet d'obtenir l'algorithme CDCL, à savoir l'algorithme à la base des prouveurs SAT modernes. Notons que dans l'algorithme CDCL, l'expression des conditions d'arrêt est différente de celle de l'algorithme DPLL. Le cas d'arrêt positif (formule cohérente) se produit lorsque toutes les variables sont affectées sans l'apparition de conflit (plus précisément, lorsque l'on affecte la dernière variable alors que la phase de propagations unitaires précédente n'a pas débouché sur un conflit). Le cas d'arrêt négatif apparaît lui lorsque la clause générée par l'analyse de conflit est vide, ce qui revient à déduire la clause vide par résolution, puisque l'analyse de conflit déduit une clause en appliquant la règle de résolution sur des clauses du problème, ou bien encore à ne pouvoir remettre en cause aucune décision. CDCL est donné par l'algorithme 5.5.

Algorithme 5.5 : Algorithme CDCL

Entrées : Une formule CNF Φ

Sorties : `vrai` si la formule `SAT` est cohérente, `UNSAT` sinon.

```
1  dl ← 0 ;                                              // niveau de décision
2  I ← ∅ ;                                          // interprétation partielle
3  tant que vrai faire
4  |  clauseConflictuelle ← propagationUnitaire(Φ, I) ;
5  |  si clauseConflictuelle = NIL alors
6  |  |  lit ← sélectionnerLittéral(Φ, I) ;
7  |  |  dl ← dl + 1 ;
8  |  |  I ← I ∪ {⟨(lit^dl)⟩} ;
9  |  |  si toutesVariablesAffectées(Φ, I) alors retourner SAT ;
10 |  sinon
11 |  |  nvlClause ← analyserConflit(I, clauseConflictuelle) ;
12 |  |  si dl = 0 alors retourner UNSAT ;
13 |  |  Φ ← Φ ∧ nvlClause ;
14 |  |  dl' ← niveauAssertif(nvlClause, I) ;
15 |  |  I ← I \ {⟨(x^p), ...⟩ | p > dl'} ;
16 |  fin
17 fin
```

Le passage de l'algorithme DPLL vers les prouveurs CDCL a été effectué en plusieurs étapes, parmi lesquelles on peut citer les prouveurs GRASP [SILVA & SAKALLAH 1996], SATO [ZHANG 1997], ou bien encore Relsat [BAYARDO JR. & SCHRAG 1997].

Dans la section suivante, nous décrivons les améliorations qui, une fois ajoutées aux prouveurs CDCL, permettent d'obtenir les prouveurs SAT dits « modernes ».

5.3 Des prouveurs CDCL aux prouveurs SAT modernes

Dans cette section, nous présentons un certain nombre d'améliorations ayant permis aux prouveurs SAT de gagner en efficacité. Certaines d'entre elles permettent d'améliorer les prouveurs CDCL, mais aussi les algorithmes que nous avons présentés antérieurement (DP, DPLL). Il est du reste à noter que parmi les améliorations que nous présentons, certaines ont été proposées et utilisées avant l'apparition des prouveurs CDCL. Nous débutons d'ailleurs par la présentation d'heuristiques de choix de variables, qui tentent de répondre à un problème apparu dès l'algorithme DP.

5.3.1 Heuristiques de choix de variable

Les algorithmes DP, DPLL, et CDCL contiennent une instruction de choix de variable. Or, ce choix de variable n'est pas anodin, dans la mesure où choisir une variable plutôt qu'une autre peut conduire à suivre un arbre de recherche totalement différent, notamment en ce qui concerne sa taille [LI & ANBULAGAN 1997a].

Prenons l'exemple d'une formule CNF Φ telle que $\Phi = \Phi_1 \wedge \Phi_2$, où Φ_1 est cohérente et Φ_2 n'est pas cohérente, et telle que Φ_1 et Φ_2 portent sur des variables différentes. Dans le cadre de l'algorithme DPLL, si les choix de variables se portent d'abord sur les variables de Φ_1, puis sur celles de Φ_2, alors on va devoir prouver l'incohérence de Φ_2 pour chacun des modèles de Φ_1. En revanche, si les premières variables

choisies sont les variables de Φ_2, alors il suffira de montrer l'incohérence de Φ_2 une unique fois pour démontrer l'incohérence de Φ.

L'ensemble des approches considérées pour opérer un « bon » choix de variable se base sur des heuristiques, étant donné que la recherche d'un choix de variable optimal est NP-difficile [LIBERATORE 2000]. Ces heuristiques peuvent être décomposées selon trois familles : les heuristiques dites « syntaxiques », les heuristiques de type *look-ahead*, et les heuristiques *look-back*.

Les premières heuristiques proposées ont été les heuristiques « syntaxiques ». Ces heuristiques tirent leur nom du fait qu'elles ne se basent que sur l'étude de la formule pour proposer une variable à choisir pour l'étape courante de l'algorithme considéré : en effet, dans ces heuristiques, le choix est effectué de manière à satisfaire le maximum de clauses possible une fois l'affectation actée. On peut par exemple citer les heuristique MOM [GOLDBERG 1979], BOHM [BURO & BÜNING 1992] et JW [JEROSLOW & WANG 1990, HOOKER & VINAY 1995, BARTH & STADTWALD 1995].

Après les approches syntaxiques sont apparues les heuristiques de type *look-ahead*, à savoir des heuristiques qui tentent d'anticiper le comportement du prouveur une fois les affectations effectuées. Parmi les heuristiques de type *look-ahead*, on peut notamment citer les heuristiques basées sur la propagation unitaire [LI & ANBULAGAN 1997a], qui anticipent les propagations unitaires en cascade pour sélectionner la variable à affecter. On peut aussi citer l'heuristique BSH utilisée dans le prouveur kcnfs [DEQUEN & DUBOIS 2003] qui tente de déterminer quelles variables font partie du *backbone* de la formule, à savoir les variables x pour lesquelles x ou $\neg x$ est conséquence logique de la formule.

Finalement, les heuristiques ayant le plus de succès dans les prouveurs actuels sont les heuristiques dites *look-back*, c'est-à-dire les heuristiques qui prennent en compte les conflits passés dans le but de déterminer quelles sont les variables les plus « importantes » du problème. Nous parlons ici uniquement d'heuristiques de choix de variable, et non de comportement *look-back* au sens général qui peut être associé aux prouveurs dont la recherche est conditionnée par des informations apprises lors de la recherche mais dont le choix de variable de décision ne tient pas nécessairement compte, comme dans le cas de Relsat [BAYARDO JR. & SCHRAG 1997].

La première approche *look-back*, proposée par [BRISOUX *et al.* 1999], propose de donner un poids à chaque clause de la formule, et d'augmenter ce poids lorsqu'un conflit apparaît en falsifiant cette clause. Lors du choix d'une variable, considérer celles qui apparaissent dans des clauses ayant des poids importants permet alors de considérer des variables qui sont probablement importantes dans le problème dans le sens où elles font partie de contraintes difficiles à satisfaire. Apparue par la suite, et à la base de la plupart des heuristiques de choix de variables employées actuellement, l'heuristique VSIDS [MOSKEWICZ *et al.* 2001] propose quant à elle de donner un poids à chacune des variables, appelé *activité*. Cette valeur est mise à jour lors de l'analyse de conflit : si une variable apparaît comme cause du conflit, alors son activité est augmentée. Ainsi, considérer les variables avec une forte activité permet de considérer les variables qui sont sources de conflit, celles qui semblent être les plus contraintes. Puisque l'activité est censée refléter l'apparition récente d'une variable dans un conflit, les activités des variables sont fréquemment diminuées de sorte à s'assurer que les variables pour lesquelles cette mesure est la plus importante correspondent à celles ayant conduit à un ou plusieurs conflits récemment.

Enfin, dans le cadre de CDCL notamment, il semble important de considérer la valeur de vérité qui sera associée à la variable choisie. En effet, bien que cela ne soit pas nécessaire dans le cadre de DP et que le retour-arrière de DPLL minimise les conséquences d'un tel choix, le fait que CDCL utilise un *backtrack* non chronologique fait que les deux valeurs d'un littéral ne seront pas nécessairement considérées. De ce fait, différentes heuristiques ont vu le jour telles que des heuristiques syntaxiques se basant sur le nombre d'occurrences positives et négatives de la variable, ainsi que la taille des clauses dans lesquelles apparaissent ces variables [JEROSLOW & WANG 1990]. On peut aussi citer le *progress saving* [PIPATSRISAWAT & DARWICHE 2007], qui consiste à affecter à une variable la dernière valeur qui lui a été attribuée, dans le but de se focaliser sur un espace de recherche (on parle d'intensification de

la recherche).

5.3.2 Redémarrages

Les heuristiques de choix de variable permettent de sélectionner une variable (ou un littéral, si on considère aussi l'heuristique de choix de polarité) dans le but d'obtenir un arbre de recherche permettant une recherche efficace [LI & ANBULAGAN 1997a], en particulier en ce qui concerne les premiers choix [GOMES *et al.* 2000].

On peut se rendre compte a posteriori que les choix initiaux ne semblent pas avoir été de bons choix, par exemple si les littéraux concernés ont un mauvais score pour l'heuristique considérée. Or, les algorithmes comme CDCL procèdent à des retours-arrière qui ne remontent pas nécessairement suffisamment haut dans l'arbre de recherche pour remettre en cause ces choix de manière rapide. Dans ce cas, un moyen de remettre en cause ces choix est de reprendre la recherche à zéro après un certain nombre de conflits, en gardant les informations rassemblées jusqu'à lors, telles que les clauses apprises et les données associées aux heuristiques (compteurs, ...). Cela permet non seulement de remettre en cause les choix initiaux qui ne semblent pas prometteurs (et de lutter contre le phénomène de « *heavy tail* » identifié dans [GOMES *et al.* 2000]), mais aussi de produire des graphes d'implication différents dans le but d'apprendre des clauses plus intéressantes que celles apprises jusqu'à présent [BIERE 2008b]. Les différentes stratégies de redémarrage employées peuvent être décomposées en deux familles : les stratégies statiques et les stratégies dynamiques.

Les stratégies statiques tirent leur nom du fait que le nombre de conflits à atteindre avant de redémarrer le prouveur ne dépend pas de la recherche en cours, mais est décidé avant même le début de la recherche. Parmi ces stratégies, on peut noter les redémarrages à intervalle fixe [RYAN 2004, MOSKEWICZ *et al.* 2001, GOLDBERG & NOVIKOV 2002] (on redémarre tous les p conflits, $p > 0$), les redémarrages basés sur une suite géométrique [WALSH 1999, EÉN & SÖRENSSON 2003a] (le $n^{\text{ème}}$ redémarrage est effectué $ab^{f(n)-1}$ conflits après le $(n-1)^{\text{ème}}$ redémarrage). En ce qui concerne un certain nombre de prouveurs récents (tels que [HUANG 2007, PIPATSRISAWAT & DARWICHE 2007, SÖRENSSON & EÉN 2009]), les stratégies de redémarrage statiques employées sont basées sur des suites mathématiques qui permettent d'alterner les redémarrages rapides et les redémarrages longs, comme les stratégies à base de suite de Luby [LUBY *et al.* 1993] ou *Inner-Outer* [BIERE 2008b].

Les stratégies dynamiques, introduites par [KAUTZ *et al.* 2002], utilisent quant à elles les informations obtenues lors de la recherche comme paramètre pour savoir si un redémarrage du prouveur devrait être effectué. Parmi les approches étudiées pour les redémarrages dynamiques, on peut par exemple citer celles basées sur les changements de polarité des variables lors d'une nouvelle affectation [BIERE 2008a], celle de Glucose [AUDEMARD & SIMON 2009a] qui prend en compte les niveaux de décision atteints lors de la recherche [AUDEMARD & SIMON 2009b], ou encore la hauteur parcourue lors des retours-arrière [HAMADI *et al.* 2009].

5.3.3 Nettoyage des clauses apprises

L'algorithme CDCL a permis d'augmenter considérablement l'efficacité des prouveurs grâce à l'apprentissage de clauses. Cependant, le nombre de clauses générées par cet algorithme peut être très important, et ces clauses peuvent ne plus être utiles au fur et à mesure que la recherche progresse. Or, gérer un grand nombre de clauses peut conduire à un ralentissement du prouveur, en particulier en ce qui concerne les phases de propagation unitaire [SILVA & SAKALLAH 1996, EÉN & SÖRENSSON 2003a]. Pour palier ce problème, les prouveurs modernes embarquent un mécanisme permettant de supprimer à certains moments une partie des clauses apprises qui sont considérées comme devenues peu utiles.

Encore une fois, étant donné que l'on ne peut savoir simplement si une clause sera de nouveau utile ou non à l'avenir, les stratégies de nettoyage de la base de clauses apprises sont basées sur des heuristiques. La première de ces heuristiques, implémentée dans le solveur Minisat [EÉN & SÖRENSSON 2003a], s'appuie sur l'heuristique de choix de variables VSIDS décrite précédemment. De la même manière que pour les variables, on associe une *activité* à chacune des clauses apprises, cette activité étant augmentée lorsque la clause concernée est utilisée par le processus d'analyse de conflit. Encore une fois, puisque cette activité doit représenter les clauses utilisées il y a peu de temps, l'ensemble des activités est diminué fréquemment de manière à ce que les clauses considérées récemment aient une activité plus importante. Ainsi, lorsque la phase de nettoyage de clauses est invoquée, les clauses apprises ayant les activités les plus faibles sont supprimées.

La deuxième mesure ayant apporté des gains importants en ce qui concerne le nettoyage de la base de clauses apprises est le LBD (pour *Literal Block Distance*), proposée par [AUDEMARD & SIMON 2009b]. Cette mesure est déterminée de la façon suivante :

- lorsqu'une clause est générée, le score de LBD qui lui est associé est égal au nombre de niveaux de décision de l'interprétation partielle ayant conduit au conflit représenté dans la clause générée (si la clause contient trois littéraux du niveau de décision p et un littéral du niveau de décision q, alors son LBD est de 2) ;
- lorsqu'une clause apprise est utilisée ultérieurement lors d'une phase de propagation unitaire, on calcule son LBD pour l'interprétation partielle courante, et on associe cette valeur à la clause si elle est inférieure à la valeur précédente.

Dans [AUDEMARD & SIMON 2009b], les auteurs montrent que les clauses dont le LBD est faible sont importantes pour la recherche. Ainsi, lors de la phase de nettoyage de la base de clauses apprises, celles dont le LBD est important sont supprimées.

Finalement, la mesure PSM d'une clause proposée dans [AUDEMARD *et al.* 2011] se base sur le *progress saving* [PIPATSRISAWAT & DARWICHE 2007] présenté précédemment. Cette mesure correspond, pour l'ensemble des littéraux retenus par le *progress saving*, au nombre de ces littéraux que la clause contient (c'est-à-dire qu'une clause falsifiée a un PSM de 0, une clause utilisée par la propagation unitaire a un PSM de 1, ...). Ainsi, le fait que des clauses aient un PSM faible semble indiquer que ces clauses auront de grandes chances de propager un littéral ou d'être falsifiées, et donc d'être utiles lors de la recherche. En revanche, le fait qu'une clause ait un PSM élevé indique que cette clause sera probablement satisfaite par plusieurs littéraux, et ne sera donc probablement que peu utile pour la suite. Cette mesure étant fortement dynamique (d'un moment à l'autre, une clause peut passer de « peu importante » à « très importante »), elle permet un état intermédiaire entre clause apprise conservée et clause apprise supprimée, appelé *clause gelée* [AUDEMARD *et al.* 2011]. Une clause gelée est une clause détachée du solveur (qui ne participe pas à la recherche), mais qui est conservée au cas où elle serait de nouveau utile plus tard (chose qui n'est pas possible avec le LBD et VSIDS qui ne permettent pas à une clause détachée de changer de score). Ainsi, dans [AUDEMARD *et al.* 2011] une clause générée est active. Lors d'une phase de nettoyage, une clause peut changer d'état :

- si une clause active a un PSM trop important, elle est gelée ;
- si une clause gelée a un PSM suffisamment faible, elle redevient active ;
- si une clause est gelée depuis un certain nombre de phases de nettoyages, elle est supprimée.

Enfin, il est important, comme dans le cas des redémarrages, de trouver une fréquence de nettoyage qui soit efficace. En effet, des nettoyages trop fréquents font perdre l'avantage de l'apprentissage, quand un nettoyage trop peu fréquent où trop peu important conduit à réduire l'efficacité du nettoyage. De plus, il est important que la taille de la base de clauses soit de plus en plus importante, afin de conserver la complétude du prouveur, puisque un seuil sur le nombre de clauses pourrait empêcher de déduire une preuve qui nécessiterait plus de clauses que ce seuil. À l'heure actuelle, les approches majoritairement utilisées sont celles proposées par [EÉN & SÖRENSSON 2003a] et [AUDEMARD & SIMON 2009a]. Dans la pre-

mière approche, le $n^{\text{ème}}$ nettoyage est opéré lorsque le nombre de clauses apprises atteint $\frac{1}{3}p.(\frac{11}{10})^{n-1}$, où p est le nombre de clauses initial. En ce qui concerne la deuxième approche, une suite statique donne le nombre de conflits qui déclenche le nettoyage : le $n^{\text{ème}}$ nettoyage est opéré après $20000 + 500(n-1)$ conflits depuis le début de la recherche.

5.3.4 Structures « watched two literals »

Comme nous l'avons noté précédemment, la règle de propagation unitaire permet d'améliorer la recherche d'un modèle en exploitant le fait qu'un littéral présent dans une clause unitaire implique qu'un modèle composé de l'interprétation partielle courante contient nécessairement ce littéral. De plus, nous avons aussi vu que la recherche de l'algorithme CDCL se base sur les séquences de décisions-propagations, jusqu'à ce qu'une propagation génère un conflit. De ce fait, dans cet algorithme, il est simplement nécessaire de connaître, lors de la propagation d'un littéral (que cette propagation soit intervenue lors de la propagation d'un littéral unitaire ou bien d'une décision) si une clause devient conflictuelle (tous ses littéraux sont évalués à faux par l'interprétation courante) ou assertive (tous ses littéraux sauf un sont évalués à faux, le dernier littéral faisant référence à une variable non propagée). De ce constat est née une structure permettant de s'assurer qu'une clause est soit falsifiée, soit assertive, de manière plus efficace qu'en auscultant l'ensemble de ses littéraux.

Cette structure proposée par [ZHANG & MALIK 2002], nommée « watched two literals », permet de ne regarder que deux littéraux de chacune des clauses lors des phases de propagations de littéraux, et est de ce fait qualifiée de *paresseuse* (c'est une évolution de l'approche *head-tail* proposée dans [ZHANG & STICKEL 2000], généralisée par [VAN GELDER 2002]). L'idée derrière l'utilisation de cette structure est que regarder deux littéraux d'une clause est suffisant pour déterminer que cette clause est assertive, falsifiée, ou qu'elle se trouve dans un état où au moins deux littéraux de la clause ne sont pas évaluables compte tenu de l'interprétation courante.

Prenons l'exemple d'une clause $x_1 \vee \ldots \vee x_n$ dont les deux littéraux pointés ne sont pas affectés.

$\downarrow$	$\downarrow$				
x_1	x_2	$\ldots$	x_i	$\ldots$	x_n

Propagation d'un littéral positif de la clause

Supposons tout d'abord qu'un littéral de cette clause soit propagé à vrai, et que la clause soit donc satisfaite. Si ce littéral est pointé, alors on sait que la clause est satisfaite par l'interprétation courante. Bien que cette information puisse sembler importante, l'algorithme CDCL ne la considère pas : en effet, sa condition d'arrêt quand la formule est cohérente intervient uniquement lorsqu'une interprétation partielle est devenue complète sans qu'un conflit ne soit présent. De ce fait, apprendre qu'une clause est satisfaite est inutile pour l'algorithme, et on peut alors ignorer cette nouvelle information.

De même, si un littéral de la clause propagé à vrai n'est pas pointé, le fait que l'on ne se rende pas compte que la clause est devenue satisfaite n'est pas important, puisque cette information est inutile pour le prouveur, qui ne s'intéresse qu'aux clauses assertives et falsifiées. De ce fait, les cas qui nous intéressent concernent seulement les configurations où une propagation falsifie un littéral de la clause.

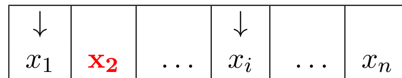
Propagation d'un littéral négatif de la clause

Supposons maintenant qu'un littéral de la clause soit propagé à faux, et que ce littéral ne soit pas pointé.

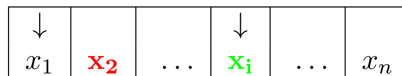
$\downarrow$	$\downarrow$				
x_1	x_2	$\ldots$	$\mathbf{x_i}$	$\ldots$	x_n

Dans ce cas, la clause n'est toujours ni assertive, ni falsifiée, puisqu'il existe toujours deux littéraux libres. De ce fait, le fait de ne pas savoir que ce littéral a été falsifié n'est pas important pour l'algorithme CDCL, qui n'a pas besoin de connaître cette information.

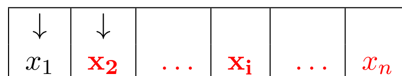
Considérons maintenant que le littéral propagé à faux soit pointé par la structure paresseuse. Dans ce cas, il est nécessaire de rechercher un autre littéral à pointer, afin de déterminer s'il existe ou non un deuxième littéral non affecté. Lors de cette recherche, trois cas de figures peuvent se présenter. Si un tel littéral existe (disons le littéral x_i), on se retrouve dans le cas dans lequel nous étions avant cette propagation, c'est-à-dire dans le cas où deux littéraux non affectés sont pointés.



Supposons qu'il n'existe pas un tel littéral, mais que lors de la recherche d'un nouveau littéral à pointer, on se rende compte qu'un littéral fait partie de l'interprétation courante. Dans ce cas, on apprend que la clause est satisfaite, et qu'elle ne peut donc être ni assertive, ni falsifiée.



Considérons maintenant le dernier cas : les littéraux qui ne sont pas pointés sont tous falsifiés. Dans ce cas, on sait qu'il existe un unique littéral non affecté dans la clause : ce littéral doit donc être propagé.



On place alors le littéral à propager dans la file des propagations à effectuer. Si le littéral opposé à celui-ci est déjà présent dans la file, on se retrouve alors dans le cas où la clause que l'on considère est falsifiée. Cela sera détecté grâce au système des pointeurs, puisqu'il n'est possible, dans ce cas de figure, que les deux littéraux pointés soient falsifiés.

Intégration des « watched two literals » dans les prouveurs CDCL

Afin d'intégrer ces structures de données dans les prouveurs CDCL, il est nécessaire de maintenir un tableau pour chacun des littéraux de la formule. Le tableau associé à chacun des littéraux x stocke l'ensemble des clauses dans lesquelles le littéral x est pointé. De ce fait, lors de la propagation du littéral $\neg x$, on sait qu'il ne sera nécessaire de ne traiter que les clauses auxquelles le tableau associé au littéral x fait référence. Dans le cas où on devrait mettre à jour les pointeurs d'une clause, il est bien entendu nécessaire de mettre aussi à jour ces tableaux de références de clauses.

Nous devons aussi noter que de telles structures vont apporter certaines restrictions à notre prouveur, en empêchant de pouvoir considérer des règles ou des heuristiques nécessitant de connaître *l'état complet* du solveur à un moment donné. C'est le cas des heuristiques « syntaxiques » , qui doivent par exemple compter le nombre de littéraux pour sélectionner une variable ; c'est aussi le cas de la règle des littéraux purs (puisque déterminer qu'un littéral est pur nécessite de vérifier que le littéral opposé apparaît zéro fois). Cependant, la structure de « watched two literals » ayant apporté des gains de temps très sensibles aux prouveurs CDCL modernes, ces restrictions apparaissent finalement négligeables comparé à l'amélioration apportée.

Finalement, nous devons aussi noter que l'emploi de telle structure n'implique aucun traitement supplémentaire lors du *backtrack* : il n'est pas nécessaire de modifier les pointeurs d'une clause en cas de remontée dans l'arbre de recherche.

5.4 Prouveurs et contraintes pseudo-booléennes

Dans les sections précédentes, nous avons présenté l'historique d'un ensemble de prouveurs SAT, depuis l'algorithme de Robinson jusqu'au prouveurs CDCL. Nous avons notamment présenté un certain nombre d'améliorations ayant permis aux prouveurs CDCL d'évoluer jusqu'à obtenir le titre de solveurs *modernes*.

Or, toutes ces améliorations sont aussi applicables à une famille de prouveurs plus généraux, appelés *prouveurs pseudo-booléens*. Ces prouveurs permettent de considérer des contraintes plus générales que les clauses, appelées *contraintes pseudo-booléennes*. Dans cette section, nous présentons les contraintes pseudo-booléennes, et nous présentons les adaptations qui permettent d'utiliser l'algorithme CDCL avec toutes ses améliorations pour gérer de telles contraintes, et ainsi passer de *prouveur SAT moderne* à *prouveur PB (Pseudo-Booléen)* moderne.

5.4.1 Contraintes pseudo-booléennes

Les contraintes pseudo-booléennes sont des contraintes de la forme :

$$\sum_{i=1}^{n} a_i x_i \; R \; k,$$

telles que :

- les x_i sont des littéraux de la formule, le littéral négatif de x_i étant remplacé par $1 - x_i$;
- les a_i sont des poids entiers, positifs ou négatifs ;
- R est un opérateur de comparaison d'entiers parmi $>, \geq, <, \leq, =$;
- k est un entier relatif.

Ainsi, afin de déterminer si une telle contrainte est satisfaite par une interprétation, il suffit de comparer la valeur de la fonction pseudo-booléenne $\sum_{i=1}^{n} a_i x_i$ avec la valeur de k, et de regarder si l'assertion définie par l'opérateur est vérifiée ou non. Dans la suite, nous considérons typiquement l'opérateur $\leq$, dans la mesure où toute contrainte pseudo-booléenne peut être écrite en utilisant cet opérateur, à l'exception des contraintes de la forme $\sum_{i=1}^{n} a_i x_i = k$ qui nécessitent de représenter à la fois la contrainte $\sum_{i=1}^{n} a_i x_i \geq k$ et la contrainte $\sum_{i=1}^{n} a_i x_i \leq k$, et qui requièrent donc deux contraintes basées sur l'opérateur $\leq$ pour être représentées.

Dans les contraintes pseudo-booléennes, on peut dénombrer un certain nombre de cas particuliers. Parmi ceux-ci, on trouve par exemple les contraintes de cardinalité, qui sont des contraintes pseudo-booléennes dans lesquelles tous les poids sont égaux à 1 :

$$\sum_{i=1}^{n} x_i \leq k.$$

Lorsqu'elle est exprimée avec l'opérateur $\leq$ pour une cardinalité de k, on dit d'une telle contrainte de cardinalité que c'est une contrainte *AtMost-k*. Les contraintes de cardinalité forment aussi un surensemble des clauses, dans la mesure où les interprétations satisfaisant une clause $x_1 \vee \ldots \vee x_n$ correspondent exactement aux interprétations qui satisfont la contrainte de cardinalité $x_1 + \ldots + x_n \geq 1$. En revanche, l'encodage d'une contrainte de cardinalité en clause est moins évident. Nous présenterons un certain nombre d'encodages connus, ultérieurement dans ce chapitre.

L'utilisation de contraintes de cardinalité permet d'obtenir des systèmes de preuve plus puissants que la résolution, comme les *plans coupes* [GOMORY 1958, CHVÁTAL 1973]. Cela signifie que la démonstration de l'incohérence d'une formule peut être réalisée en un nombre d'étapes inférieur à celui de la résolution [COOK *et al.* 1987], mais aussi qu'il est possible d'apprendre des contraintes plus contraignantes lors d'un processus d'analyse de conflit.

5.4.2 Plans coupes et résolution généralisée

Le systèmes de preuve dit des *plans coupes* [GOMORY 1958] (ou *cutting planes*) se base sur des équations linéaires sur les variables booléennes et non sur des clauses ; il est de ce fait nécessaire de considérer les clauses comme étant des contraintes pseudo-booléennes. Par exemple, la clause $x_1 \vee x_2 \vee \neg x_3$ sera considérée comme étant la contrainte $x_1 + x_2 + (1 - x_3) \geq 1$, que l'on peut aussi noter comme une contrainte *AtMost* : $-x_1 - x_2 + x_3 \leq 0$.

Au contraire de la résolution, qui n'autorise qu'une règle ($(A \vee x) \wedge (B \vee \neg x) \models (A \vee B)$), le système de preuve des plans coupes en autorise quatre.

Tout d'abord, on peut trivialement introduire les bornes des variables booléennes (x est une variable booléenne).

$$\frac{}{0 \leq x \leq 1}$$

Il est aussi possible de réaliser la somme de deux équations ; c'est cette règle qui va permettre aux plans coupes de réaliser efficacement la règle de résolution, et donc de lui permettre d'être un système de preuve au moins aussi puissant que la résolution seule.

$$\frac{\sum_{i=1}^{n} a_i x_i \leq k_1 \quad \sum_{i=1}^{n} b_i x_i \leq k_2}{\sum_{i=1}^{n} (a_i + b_i) x_i \leq k_1 + k_2}$$

En effet, considérons deux clauses, $x_1 \vee x_2 \vee x_3$ et $\neg x_1 \vee x_5 \vee x_6$. Une fois traduites sous formes de contraintes pseudo-booléennes, ces clauses deviennent $x_1 + x_2 + x_3 \geq 1$ et $-x_1 + x_4 + x_5 \geq 0$; en sommant ces deux contraintes, on obtient $x_2 + x_3 + x_4 + x_5 \geq 1$, ce qui est bien équivalent à la clause $x_2 \vee x_3 \vee x_4 \vee x_5$ que l'on aurait obtenue en appliquant la règle de résolution sur les deux clauses initiales.

Il est aussi possible de multiplier ou de diviser une équation par une constante entière.

$$\frac{\sum_{i=1}^{n} a_i x_i \leq k}{\sum_{i=1}^{n} c \cdot a_i x_i \leq c \cdot k}$$

En ce qui concerne la division, si une valeur est non entière, on remplace alors cette valeur par son arrondi à l'entier supérieur.

$$\frac{\sum_{i=1}^{n} c \cdot a_i x_i \leq k}{\sum_{i=1}^{n} a_i x_i \leq \lceil \frac{k}{c} \rceil}$$

Contrairement à la résolution, on ne cherche plus ici à dériver la clause vide pour obtenir un conflit, mais à dériver l'équation $0 \geq 1$. Comme nous venons de la voir, les plans coupes forment un système de preuves au moins aussi puissant que la résolution, puisqu'ils sont capables d'émuler efficacement sa seule règle. Cependant, on connaît aussi des problèmes, comme les instances de pigeons/pigeonniers, pour lesquelles il existe une preuve de longueur polynomiale dans le nombre de contraintes pour les plans coupes, quand une preuve utilisant la résolution est nécessairement de longueur exponentielle dans la taille de l'instance [COOK *et al.* 1987]. Des résultats théoriques récents indiquent que les plans coupes sont aussi très efficaces en ce qui concerne l'espace mémoire nécessaire pour réaliser une preuve [GALESI *et al.* 2014].

Exemple 5.12 (Exemple de preuve par le système *cutting planes* pour le problème des pigeons) *Soit le problème de mettre $n + 1 = 3$ pigeons dans $n = 2$ pigeonniers, sachant que l'on ne peut mettre plus d'un pigeon par pigeonnier. On définit $n^2 + n = 6$ variables $x_{i,j}$, où $x_{i,j}$ est à* vrai *si et seulement si le pigeon i est dans le pigeonnier j. On code le problème de la façon suivante.*

$$\begin{array}{lcl} -x_{1,1} - x_{2,1} - x_{3,1} & \geq & -1 \\ -x_{1,2} - x_{2,2} - x_{3,2} & \geq & -1 \\ x_{1,1} + x_{1,2} & \geq & 1 \\ x_{2,1} + x_{2,2} & \geq & 1 \\ x_{3,1} + x_{3,2} & \geq & 1 \end{array}$$

Dans ce codage, les deux premières contraintes impliquent qu'un pigeonnier contient au plus un pigeon ; la première est par exemple équivalente à $x_{1,1} + x_{2,1} + x_{3,1} \leq 1$*. Les trois dernières contraintes imposent que chaque pigeon doit être placé dans un pigeonnier.*

En utilisant la résolution comme système de preuve, toute preuve de l'incohérence de ce problème est exponentielle dans le nombre de contraintes. Cependant, en considérant les plans coupes et en sommant l'ensemble des contraintes, on obtient directement la preuve d'incohérence.

$$\begin{array}{rcccccccl} (& -x_{1,1} & & -x_{2,1} & & -x_{3,1} & &) \geq & -1 \\ +\,(& & -x_{1,2} & & -x_{2,2} & & -x_{3,2} &) \geq & -1 \\ +\,(& x_{1,1} & +x_{1,2} & & & & &) \geq & 1 \\ +\,(& & & x_{2,1} & +x_{2,2} & & &) \geq & 1 \\ +\,(& & & & & x_{3,1} & +x_{3,2} &) \geq & 1 \\ \hline & & & & & & 0 & \geq & 1 \end{array}$$

Dans les solveurs PB modernes dits *cutting planes* [CHAI & KUEHLMANN 2005, DIXON 2004, SHEINI & SAKALLAH 2006, LE BERRE & PARRAIN 2010], le système de preuve employé est en fait la résolution généralisée [HOOKER 1988] et non les plans coupes. En ce qui concerne la résolution généralisée, la règle de la résolution est remplacée par l'utilisation de la règle de la multiplication sur les deux contraintes initiales puis l'utilisation de la règle de l'addition sur les deux contraintes obtenues de manière à ce que le littéral que l'on souhaite éliminer disparaisse, comme le montre l'exemple ci-dessous.

Exemple 5.13 (Résolution généralisée) *Soient les deux contraintes* $2x_1 + x_2 + x_3 \geq 2$ *et* $-x_1 - x_2 - x_3 \geq -1$*. Supposons que lors de l'analyse de conflit, on souhaite générer une nouvelle contrainte qui ne contient pas la variable* x_1*, en utilisant les deux contraintes que nous avons données. Afin que la somme de deux contraintes fasse disparaître la variable* x_1*, il est nécessaire que cette variable ait le même poids dans la contrainte où elle apparaît positivement et celle où elle apparaît négativement. Dans notre exemple, il est nécessaire de multiplier la deuxième contrainte par deux pour que cette condition soit vérifiée.*

$$\begin{array}{rcccl} (& 2x_1 & x_2 & x_3 &) \geq \; 2 \\ +\,2\,(& -x_1 & -x_2 & -x_3 &) \geq \; -1 \\ \hline & & -x_2 & -x_3 & \geq \; 0 \end{array}$$

Ainsi, la contrainte obtenue par la résolution généralisée sur x_1 *depuis* $2x_1 + x_2 + x_3 \geq 2$ *et* $-x_1 - x_2 - x_3 \geq -1$ *est la contrainte* $-x_2 - x_3 \geq 0$*.*

Le système de preuve des plans coupes *p-simule* la résolution [COOK *et al.* 1987] (i.e. une preuve pour la résolution peut être traduite en temps polynomial en preuve pour la résolution généralisée ; la résolution généralisée est en fait équivalente à la résolution lorsque des clauses sont considérées), et il est équivalent aux plans coupes quand des contraintes pseudo-booléennes sont considérées. De ce fait, afin que ces prouveurs aient la capacité d'utiliser la puissance du système de preuve *cutting planes*, il est nécessaire que les informations du problème connues sous forme de contraintes PB ne soient pas « diluées » par un encodage de ces contraintes en clauses (quelques-uns de ces codages sont présentés

plus loin dans ce chapitre). Dans le cas contraire, il faut être capable de découvrir que de telles contraintes sont cachées derrière les clauses, et les recomposer, lors d'une phase de prétraitement par exemple. De telles méthodes sont elles-aussi décrites plus loin dans ce chapitre.

Enfin, à l'heure actuelle, les prouveurs pseudo-booléens basés sur la résolution généralisée sont généralement bien moins efficaces que leurs homologues basés sur la résolution, alors même que le système de preuves est strictement plus puissant. Ceci est dû à l'inefficacité de certains calculs comme ceux de plus petits communs multiples, nécessaires pour déterminer les coefficients minimaux par lesquels multiplier les contraintes lors de la résolution généralisée, dans le but éviter que les coefficients grandissent de façon trop importante au fur et à mesure de la recherche.

5.4.3 Intégration des contraintes pseudo-booléennes dans les prouveurs CDCL

Adaptation de la procédure d'analyse de conflits

Dans la section précédente, nous avons mis en lumière le fait que les systèmes de preuve par résolution et par résolution généralisée requièrent l'utilisation de structures de données différentes. En effet, si l'on souhaite utiliser la résolution comme système de preuve, les contraintes que l'on doit considérer doivent être des clauses, alors que dans le cas de la résolution généralisée, les contraintes sont vues comme des équations linéaires, ou encore comme des contraintes pseudo-booléennes.

Or, la conversion d'un type de contrainte vers un autre n'est pas toujours évident. En effet, alors qu'il est simple de transformer une clause en une contrainte de cardinalité équivalente (il suffit de considérer que la somme des valeurs des littéraux, en remplaçant les littéraux négatifs $\neg l$ par $1 - l$, doit être supérieure ou égale à 1), utiliser la règle de la résolution sur des contraintes pseudo-booléennes lors de l'analyse de conflit n'est pas trivial [ALOUL *et al.* 2002a].

Une façon de procéder, qui est notamment celle utilisée dans Sat4j, est de considérer une clause qui est conséquence de la contrainte pseudo-booléenne considérée dans le graphe d'implication [LE BERRE & PARRAIN 2010]. Une façon de construire une telle clause consiste à y ajouter l'ensemble des littéraux qui étaient falsifiés dans la contrainte initiale au moment où elle est devenue assertive, et finalement d'y insérer le littéral propagé.

Exemple 5.14 (Résolution avec une contrainte pseudo-booléenne) *Soit* $\Phi = (a + b \geq 1) \wedge (-b + 2c + 2d \geq 2) \wedge (-c - d \geq -1)$*. Considérons l'interprétation partielle* $\mathcal{I} = \{\langle(\neg a^1), b^1, c^1, d^1, \neg d^1\rangle\}$ *menant à un conflit portant sur la valeur de la variable* d*. Lors de l'analyse de conflit, on va tout d'abord chercher à appliquer la résolution entre deux contraintes ayant propagé des valeurs opposées pour la variable* d*. Dans notre cas, il s'agit des deux dernières contraintes de la formule ; or, bien que la dernière soit en fait la clause* $\neg c \vee \neg d$ *exprimée sous la forme d'une inéquation, il n'en est pas de même en ce qui concerne la contrainte* $-b + 2c + 2d \geq 2$*. Cependant, si on regarde quelles affectations ont poussé à propager le littéral* d *en raison de cette contrainte, on voit que la raison est l'affectation de la variable* b *à vrai (une fois* b *propagé à vrai, la seule façon de satisfaire cette contrainte est de propager à la fois* c *et* d *à vrai ; une discussion à propos des multiples propagations des contraintes pseudo-booléennes intervient dans la section suivante), et donc que la clause* $\neg b \vee d$*, qui est une conséquence logique de la contrainte PB* $-b + 2c + 2d \geq 2$*, aurait été suffisante pour propager* d*. De ce fait, lors de la première étape de l'analyse de conflit, on va simplement effectuer la résolution entre les clauses* $\neg b \vee d$ *et* $\neg c \vee \neg d$*, et la première résolvante est alors la clause* $\neg b \vee \neg c$*.*

Adaptation des structures paresseuses *watched two literals*

Comme nous l'avons vu précédemment, les structures paresseuses de type *watched two literals* ont permis des gains conséquents dans les prouveurs CDCL, en permettant de détecter les clauses assertives

(les clauses non satisfaites dont un seul littéral n'est pas affecté) et les clauses falsifiées sans pour autant inspecter l'ensemble d'une formule. En ce qui concerne les contraintes pseudo-booléennes, il n'est pas possible de détecter l'apparition de contraintes assertives ou falsifiées uniquement en inspectant deux littéraux seulement. En revanche, il est possible d'adapter ces structures en pointant des littéraux [CHAI & KUEHLMANN 2005, DIXON 2004].

Considérons tout d'abord les contraintes de cardinalité, c'est-à-dire les contraintes pseudo-booléennes dont les poids associés aux littéraux sont unitaires. Soit une contrainte de cardinalité $x_1 + \ldots + x_n \geq k$:

- cette contrainte est falsifiée s'il ne reste plus suffisamment de littéraux à affecter pour que k littéraux soient évalués à vrai, ce qui revient au cas où $n - k + 1$ littéraux sont falsifiés ;
- de ce fait, cette contrainte est assertive dans le cas où $n - k$ littéraux sont falsifiés.

Cette observation est cohérente avec les cas étudiés pour une clause, dans la mesure où une clause $\gamma = x_1 \vee \ldots \vee x_n$ est une contrainte de cardinalité α particulière : $x_1 + \ldots + x_n \geq 1$. En effet, γ est falsifiée si tous ses n littéraux sont falsifiés, ce qui correspond bien à $n - 1 + 1$ littéraux falsifiés pour α. De la même façon, γ est assertive si $n - 1$ de ses littéraux sont falsifiés, ce qui correspond bien aux $n - 1$ littéraux devant être falsifiés pour que α soit assertive.

Or, dans le cas des clauses, on pointait $k + 1 = 2$ littéraux pour vérifier qu'au moment où un de ces $k + 1$ littéraux était affecté à faux, il restait ou non k littéraux non affectés, et ainsi décider si la clause est assertive ou falsifiée. De la même manière, dans le cadre des contraintes de cardinalité, il suffit de pointer $k + 1$ littéraux, peu importe le nombre k apparaissant dans la contrainte, afin de déterminer lors d'une propagation si la contrainte considérée est devenue assertive ou falsifiée. Notons toutefois qu'une contrainte de cardinalité peut propager plusieurs littéraux. Supposons que parmi les n littéraux x_i, $n - k$ soient falsifiés (la contrainte est donc assertive), et que les k littéraux restants ne soient pas affectés. Dans ce cas, l'unique manière de satisfaire cette contrainte consiste à affecter l'ensemble des k littéraux restants à vrai : il faut donc propager ces k littéraux.

Exemple 5.15 (Multiples propagations d'une contrainte de cardinalité) *Soit la contrainte de cardinalité $x_1 + x_2 + x_3 + x_4 \geq 2$. Supposons que les littéraux x_1 et x_2 soient affectés à faux. L'unique façon de satisfaire la contrainte sous ces hypothèses est de propager à la fois les littéraux x_3 et x_4.*

Considérons maintenant les contraintes pseudo-booléennes générales. La difficulté dans la détection du fait qu'une contrainte de cardinalité soit devenue assertive ou falsifiée est que le nombre de littéraux impliquant cet état n'est pas fixe. En effet, la contrainte $(n - 1)x_1 + x_2 + \ldots + x_n \geq n - 1$ peut être satisfaite de deux façons différentes :

- si le littéral x_1 est évalué à vrai, alors la contrainte est satisfaite, peu importe les valeurs associées aux autre littéraux x_i : un unique littéral peut donc être suffisant pour satisfaire la contrainte ;
- en revanche, si le littéral x_1 est falsifié, alors l'ensemble des littéraux $x_2, \ldots, x_n$ doivent être évalués à vrai pour que la contrainte soit satisfaite : il peut donc être nécessaire que $n - 1$ littéraux soient affectés à vrai pour que la contrainte soit satisfaite.

De ce fait, afin de détecter le fait qu'une contrainte pseudo-booléenne devient assertive ou falsifiée, il est nécessaire de considérer les poids associés à chacun des littéraux en plus de considérer le degré k de la contrainte. Ainsi, pour les contraintes pseudo-booléennes, il est nécessaire de détecter quand la somme des coefficients des littéraux non affectés additionnée à la somme des coefficients correspondant aux littéraux satisfaits est inférieure à k pour déterminer qu'elle est falsifiée, et quand elle est égale à k pour déterminer que la contrainte est assertive. Au moment de choisir le nombre de pointeurs que devra contenir la contrainte, il faut considérer les sommes des p poids les plus petits associés au littéraux afin d'être certain de déterminer le nombre maximal de littéraux non affectés pouvant provoquer le fait qu'une contrainte PB devienne assertive ou falsifiée. De la même manière que pour les contraintes de cardinalité, il est nécessaire d'ajouter au nombre maximal de littéraux à inspecter pour déterminer qu'une

contrainte PB devient assertive un pointeur supplémentaire, afin de détecter qu'une propagation effectuée implique une ou plusieurs propagations.

Il est à noter qu'en plus de pouvoir propager plusieurs littéraux à un niveau de décision, une contrainte PB peut aussi propager des littéraux à plusieurs niveaux de décision, comme le montre l'exemple suivant.

Exemple 5.16 (Multiples propagations d'une contrainte pseudo-booléenne) *Soit la contrainte PB* $4x_1 + 4x_2 + x_3 + x_4 + x_5 \geq 6$*. Supposons que la littéral* x_1 *soit affecté à faux. Dans ce cas, il est nécessaire de propager le littéral* x_2*, puisque dans le cas contraire, la somme des poids des littéraux non affectés serait égale à 3, ce qui est insuffisant pour satisfaire la contrainte.*

De plus, considérons qu'après cette propagation, le littéral x_3 *est affecté à faux. Dans l'interprétation partielle courante, on a donc* $x_1 = x_3 = \bot$ *et* $x_2 = \top$*, ce qui implique qu'il reste à affecter des littéraux dont la somme des poids est au moins égale à 2 dans le but de satisfaire la contrainte. Or, les deux seuls littéraux encore non affectés,* x_4 *et* x_5*, ont justement une somme de poids égale à deux : il faut donc propager ces littéraux au niveau de décision courant. De ce fait, lors de la construction de l'interprétation courante, on voit que la contrainte considérée dans l'exemple a propagé des littéraux à plusieurs niveaux de décision, ce qui n'est pas possible avec les contraintes de cardinalité.*

5.4.4 Encodages de contraintes de cardinalité par des clauses

Dans cette section, nous présentons un certain nombre d'encodages connus pour la traduction de contraintes de cardinalité en clauses. Le but de cette présentation succincte est d'introduire la section suivante, qui traitera de la détection de contraintes de cardinalité dans une formule CNF, de manière à pouvoir utiliser les systèmes de preuves plus puissants que la résolution (et notamment la résolution généralisée) de manière efficace, en s'appuyant sur des contraintes pseudo-booléennes plutôt que des clauses. Notons que les encodages décrits dans la suite sont considérés arc-cohérents, dans le sens où une fois une contrainte de cardinalité encodée comme une conjonction de clauses, l'affectation de littéraux dans la formule CNF provoque la propagation des littéraux qui auraient été propagés en effectuant les mêmes affectation au niveau de la contrainte de cardinalité.

Au delà des trois encodages que nous détaillons dans cette section, beaucoup d'autres ont été proposés pour les contraintes AtMost-1 ([WALSH 2000, GENT & PROSSER 2002, GENT & NIGHTINGALE 2004, ANSÓTEGUI & MANYÀ 2004, KLIEBER & KWON 2007, FRISCH & GIANNAROS 2010, HÖLLDOBLER & NGUYEN 2013, BARAHONA *et al.* 2014] et pour les contraintes AtMost-k avec $k > 1$ [ALOUL *et al.* 2002b, BAILLEUX & BOUFKHAD 2003, SINZ 2005], certains utilisant des structures complexes [EÉN & SÖRENSSON 2006, ASÍN *et al.* 2009, BEN-HAIM *et al.* 2012]. Une étude récente approfondit l'efficacité pratique de ces encodages dans le contexte du problème MaxSAT [MARTINS *et al.* 2012].

Nous débutons par la présentation du *naïve encoding*. Pour une contrainte AtMost-1 $\sum_{i=1}^{n} x_i \leq 1$, l'idée derrière le *naïve encoding*, aussi appelé *pairwise encoding* ou *binomial encoding* (ou même *direct encoding* dans la communauté CP [WALSH 2000]), est d'exclure toute paire de littéraux qui ne peuvent être satisfaits de manière simultanée, et ce de façon explicite : $\bigwedge_{i=1}^{n} \bigwedge_{j>i}^{n} (\neg x_i \vee \neg x_j)$. De cette manière, l'encodage d'une contrainte portant sur n littéraux nécessite $\frac{n(n-1)}{2}$ clauses.

Exemple 5.17 (Encodage de la contrainte $\sum_{i=1}^{n} x_i \leq 1$ avec le *binomial encoding*) *Quand* $n = 10$*, la contrainte de cardinalité* $\sum_{i=1}^{n} x_i \leq 1$ *peut s'écrire avec les 10 variables et les 45 clauses suivantes*

en utilisant le binomial encoding.

$\neg x_1 \vee \neg x_2$	$\neg x_1 \vee \neg x_3$	$\neg x_1 \vee \neg x_4$	$\neg x_1 \vee \neg x_5$
$\neg x_1 \vee \neg x_6$	$\neg x_1 \vee \neg x_7$	$\neg x_1 \vee \neg x_8$	$\neg x_1 \vee \neg x_9$
$\neg x_1 \vee \neg x_{10}$	$\neg x_2 \vee \neg x_3$	$\neg x_2 \vee \neg x_4$	$\neg x_2 \vee \neg x_5$
$\neg x_2 \vee \neg x_6$	$\neg x_2 \vee \neg x_7$	$\neg x_2 \vee \neg x_8$	$\neg x_2 \vee \neg x_9$
$\neg x_2 \vee \neg x_{10}$	$\neg x_3 \vee \neg x_4$	$\neg x_3 \vee \neg x_5$	$\neg x_3 \vee \neg x_6$
$\neg x_3 \vee \neg x_7$	$\neg x_3 \vee \neg x_8$	$\neg x_3 \vee \neg x_9$	$\neg x_3 \vee \neg x_{10}$
$\neg x_4 \vee \neg x_5$	$\neg x_4 \vee \neg x_6$	$\neg x_4 \vee \neg x_7$	$\neg x_4 \vee \neg x_8$
$\neg x_4 \vee \neg x_9$	$\neg x_4 \vee \neg x_{10}$	$\neg x_5 \vee \neg x_6$	$\neg x_5 \vee \neg x_7$
$\neg x_5 \vee \neg x_8$	$\neg x_5 \vee \neg x_9$	$\neg x_5 \vee \neg x_{10}$	$\neg x_6 \vee \neg x_7$
$\neg x_6 \vee \neg x_8$	$\neg x_6 \vee \neg x_9$	$\neg x_6 \vee \neg x_{10}$	$\neg x_7 \vee \neg x_8$
$\neg x_7 \vee \neg x_9$	$\neg x_7 \vee \neg x_{10}$	$\neg x_8 \vee \neg x_9$	$\neg x_8 \vee \neg x_{10}$
$\neg x_9 \vee \neg x_{10}$			

Le *nested encoding* utilise quant à lui des variables auxiliaires, dans le but de diminuer le nombre de clauses en séparant de manière récursive chaque contrainte en deux contraintes plus petites :

$$\sum_{i=1}^{n} x_i \leq 1 =$$
$$(y + (\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor -1} x_i) \leq 1) \wedge (\neg y + (\sum_{i=\lfloor \frac{n}{2} \rfloor -1}^{n} x_i) \leq 1).$$

Cet encodage est un encodage récursif, dans le sens où en présence d'une contrainte à encoder, on en génère en fait deux, qui sont elles mêmes encodées via cet encodage, tant que $n > 4$. Pour $n = 4$, le *naïve encoding* nécessite 6 clauses, tout comme le *nested encoding*, qui nécessite en revanche plus de variables. Ainsi, on arrête de séparer les contraintes dès qu'elle contiennent au plus 4 littéraux. De cette manière, le *nested encoding* requiert $3n - 6$ clauses.

Exemple 5.18 (Encodage de la contrainte $\sum_{i=1}^{n} x_i \leq 1$ avec le *nested encoding*) *Quand $n = 10$, la contrainte de cardinalité $\sum_{i=1}^{n} x_i \leq 1$ peut s'écrire avec les 13 variables et les 24 clauses suivantes en utilisant le* nested encoding.

$\neg x_1 \vee \neg x_2$	$\neg x_1 \vee \neg x_3$	$\neg x_1 \vee \neg y_2$
$\neg x_2 \vee \neg x_3$	$\neg x_2 \vee \neg y_2$	$\neg x_3 \vee \neg y_2$
$\neg x_4 \vee \neg x_5$	$\neg x_4 \vee \neg y_1$	$\neg x_4 \vee y_2$
$\neg x_5 \vee \neg y_1$	$\neg x_5 \vee y_2$	$\neg y_1 \vee y_2$
$\neg x_6 \vee \neg x_7$	$\neg x_6 \vee \neg x_8$	$\neg x_6 \vee \neg y_3$
$\neg x_7 \vee \neg x_8$	$\neg x_7 \vee \neg y_3$	$\neg x_8 \vee \neg y_3$
$\neg x_9 \vee \neg x_{10}$	$\neg x_9 \vee y_1$	$\neg x_9 \vee y_3$
$\neg x_{10} \vee y_1$	$\neg x_{10} \vee y_3$	$y_1 \vee y_3$

À l'heure actuelle, le meilleur encodage de contraintes AtMost-1 connu pour un grand nombre de variables (à partir de $n > 47$ [MANTHEY *et al.* 2012]) est le *two product encoding* [CHEN 2010]. Pour n variables dans la contrainte, deux entiers $p = \lfloor \sqrt{n} \rfloor$ et $q = \lceil \frac{n}{p} \rceil$ sont utilisés comme indices de ligne et de colonne. Les variables x_i sont placées dans une matrice de telle sorte que chaque variable soit définie par un unique couple d'indices (r_s, c_t) grâce aux clauses $\neg x_i \vee r_s$ et $\neg x_i \vee c_t$, où $s = \lfloor \frac{i-1}{q} \rfloor + 1$ et $t = ((i-1) \bmod q) + 1)$. De la même manière que l'encodage précédent, deux nouvelles contraintes de cardinalité sont générées pour une contrainte encodée ; elles sont encodées avec le *two product encoding* tant que $n > 4$, puis avec le *binomial encoding* une fois que $n \leq 4$.

Exemple 5.19 (Encodage de la contrainte $\sum_{i=1}^{n} x_i \leq 1$ avec le *two product encoding*) *La contrainte de cardinalité $\sum_{i=1}^{n} x_i \leq 1$ peut s'écrire avec les 17 variables et les 26 clauses suivantes en utilisant le* two product encoding, *comme le montre la figure 5.4.*

encodage	clauses nécessaires	variables nécessaires
binomial encoding	49 995 000	10 000
nested encoding	29 610	13 726
two product encoding	20 229	10 232

TABLE 5.1 – Nombres de variables et de clauses nécessaires pour encoder la contrainte $\sum_{i=1}^{10000} x_i \leq 1$ pour le *binomial encoding*, le *nested encoding*, et le *two product encoding*.

Bien que sur l'exemple à dix variables, le *two product encoding* soit moins performant que le *nested encoding*, il n'en va pas de même lorsque le nombre de variables est plus important, comme le montre la table 5.1 qui indique le nombre de clauses et de variables nécessaires pour encoder une contrainte AtMost-1 à 10 000 variables pour les trois encodages que nous avons présentés.

Notons finalement que des travaux ont été réalisés concernant la représentation de contraintes pseudo-booléennes à l'aide de clauses [BAILLEUX *et al.* 2009]. Nous présentons maintenant une contribution publiée dans les actes de la conférence d'audience internationale SAT [BIERE *et al.* 2014], dans laquelle nous proposons des méthodes statiques et sémantiques de détection de contraintes de cardinalité.

5.5 Détection de contraintes de cardinalité

Nous avons noté précédemment que pour tirer bénéfice du système de preuve par résolution généralisée, il faut disposer de contraintes de cardinalité, puisque dans le cas contraire la résolution généralisée est équivalente à la résolution classique. Nous avons dans cette optique proposé différentes approches dans le but de recouvrer des contraintes qui auraient été diluées par un encodage sous forme de clauses, ou bien dans le but de découvrir des contraintes qui n'auraient pas été déterminées par l'expert ayant encodé le problème.

Dans la suite de ce chapitre, nous présentons nos contributions en terme d'approches permettant de détecter des contraintes, puis nous montrons expérimentalement que ces approches permettent de résoudre des jeux d'essai pour lesquels il n'existait à l'heure actuelle, à notre connaissance, aucun outil capable de fournir une preuve d'incohérence.

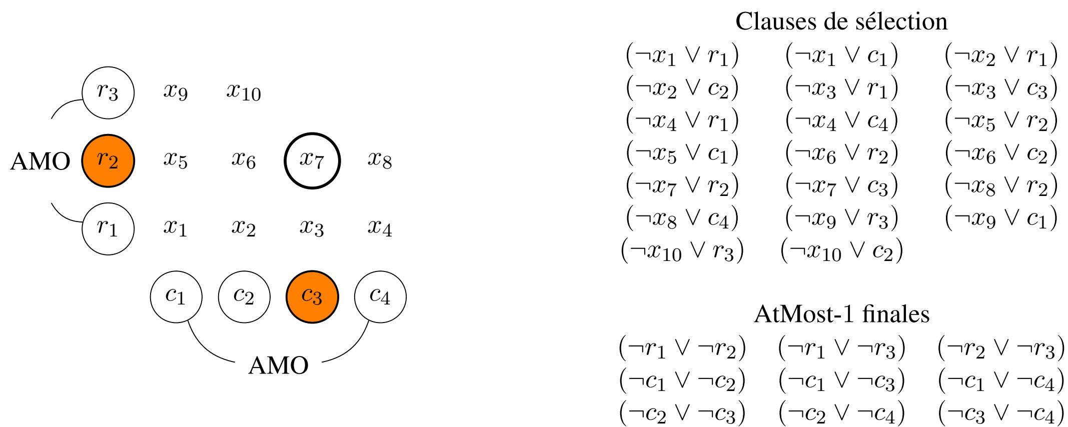


FIGURE 5.4 – Encodage de la contrainte *AtMost*-1 $\sum_{i=1}^{10} x_i \leq 1$ avec le *two product encoding*, et deux contraintes *AtMost*-1 auxiliaires, $r_1 + r_2 + r_3 \leq 1$ et $c_1 + c_2 + c_3 + c_4 \leq 1$.

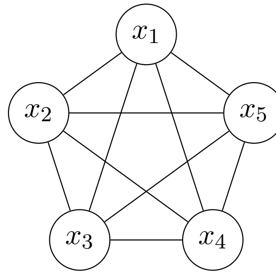


FIGURE 5.5 – Illustration d'une contrainte *AtMost*-1 quand les clauses binaires sont vues comme un NAND graphe.

5.5.1 Détection statique de contraintes *AtMost*-1 et *AtMost*-2

La détection du *naïve encoding* des contraintes *AtMost*-1 peut être réalisé par une analyse syntaxique de la formule, plus précisément en recherchant des cliques dans le *NAND graph* (NAG) de la formule (un graphe non orienté dans lequel deux littéraux sont reliés si leurs littéraux complémentaires appartiennent à une même clause binaire de la formule). Dans [ANSÓTEGUI 2004, ANSÓTEGUI *et al.* 2007], les auteurs modifient les solveurs Chaff [MOSKEWICZ *et al.* 2001] et Satz [LI & ANBULAGAN 1997a, LI & ANBULAGAN 1997b] pour reconnaître ces contraintes en utilisant la propagation unitaire et la recherche locale. Une structure de données spécifique, basée sur ce NAG, est souvent intégrée dans les solveurs SAT modernes pour réduire la mémoire utilisée. Depuis un tel graphe, on peut extraire les contraintes AtMost-1 par une analyse syntaxique. La détection du *naïve encoding* des contraintes AtMost-2 peut être effectuée en s'intéressant aux clauses ternaires.

Les outils 3MCARD [VAN LAMBALGEN 2006], LINGELING [BIERE 2013] et SBSAT [WEAVER 2012] sont capables de reconnaître de telles contraintes en procédant à une analyse syntaxique ; nous présentons ci-dessous leurs procédures, en plus de notre nouvelle méthode.

3MCARD et SBSAT ne restreignent pas leur recherche aux clauses d'une certaine taille, mais considèrent la formule dans son ensemble. SBSAT construit des OBDDs en utilisant les clauses qui partagent des littéraux, et détecte les contraintes en fusionnant et analysant ces OBDDs [WEAVER 2012]. 3MCARD construit un graphe à partir des clauses binaires, et augmente la taille de la contrainte courante en collectant de nouvelles clauses. Seul Lingeling possède des méthodes spécialement conçues pour la détection des contraintes AtMost-1 et AtMost-2.

Détection du pairwise encoding

Détecter la structure du *pairwise encoding* dans un NAG est facile : si une clique est présente, alors les littéraux des nœuds correspondants forment une contrainte AtMost-1. Puisque la recherche d'une clique de taille k dans un graphe est NP-difficile [KARP 1972], une phase de prétraitement ne devrait pas effectuer une recherche exhaustive de cliques.

L'algorithme glouton de recherche de cliques implémenté dans LINGELING itère sur l'ensemble des littéraux n qui n'ont pas encore été inclus dans une contrainte AtMost-1. Pour chaque n, l'ensemble S de littéraux candidats est initialisé avec n. Ensuite, chaque littéral l qui apparaît nié dans une clause binaire $\neg n \vee \neg l$ (l et n sont connectés dans le graphe) est ajouté de manière gloutonne dans l'ensemble S après avoir vérifié que pour chaque littéral k ajouté précédemment dans S, l et k sont reliés dans le graphe. L'ensemble S finalement obtenu est une clique (voir figure 5.5 pour un exemple), et représente donc une contrainte AtMost-1 ($\sum_{l \in S} l \leq 1$), non triviale dans le cas où $|S| > 2$ [BIERE 2013].

Détection du *nested encoding*

Considérons le *nested encoding* de la contrainte AtMost-1 $x_1 + x_2 + x_4 + x_5 \leq 1$, où la contrainte est divisée en $x_1 + x_2 + x_3 \leq 1$ et $\neg x_3 + x_4 + x_5 \leq 1$. Ces deux contraintes sont représentées en CNF par les 6 clauses $(\neg x_1 \vee \neg x_2), (\neg x_1 \vee \neg x_3), (\neg x_2 \vee \neg x_3), (x_3 \vee \neg x_4), (x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)$. Puisque nous ne disposons pas de la contrainte binaire $(\neg x_1 \vee \neg x_4)$, la méthode présentée au-dessus ne peut être appliquée pour détecter cette contrainte. Nous présentons ici une nouvelle façon de détecter ces contraintes.

Les deux contraintes issues de la division de la contrainte originale peuvent être reconnues par la méthode de la section précédente (leurs littéraux forment une clique dans le NAG). De ce fait, on détecte qu'il existe une contrainte AtMost-1 qui porte sur le littéral x_3, et une autre qui porte sur $\neg x_3$. Par sommation, on obtient la contrainte recherchée.

Pour chaque variable v, toutes les paires de AtMost-1 (A,B) telles que $v \in A$ et $\neg v \in B$ sont sommées et simplifiées. L'étape de simplification vérifie qu'il n'y a pas de littéraux dupliqués (dans ce cas, les littéraux doivent être falsifiés puisque leur poids est supérieur au degré), ou de littéraux complémentaires (ici, tous les littéraux de la contrainte $(A + B)$ à l'exception des littéraux complémentaires doivent être falsifiés, puisque $x + \neg x = 1$ implique que le degré doit être réduit de 1, c'est-à-dire nul). La contrainte simplifiée est finalement ajoutée à S puis retournée par l'algorithme.

Puisque le *nested encoding* peut être encodé de manière récursive, l'algorithme peut être appelé de multiples fois pour trouver ces contraintes de manière récursive. Pour éviter de considérer plusieurs fois les mêmes couples de contraintes, on peut ne considérer que les couples pour lesquels un des deux éléments est une contrainte qui a été découverte à l'itération précédente.

Détection du *two product encoding*

Le *two product encoding* a une structure récursive similaire à celle du *nested encoding* ; sa structure est cependant plus complexe. Nous détaillons donc cet algorithme plus en détail dans cette section. La figure 5.4 illustre une contrainte AtMost-1 encodée *via* le *two product encoding*.

Pour chaque littéral concerné, $x_1, ..., x_{10}$ dans l'exemple de la figure 5.4, deux implications sont utilisées pour fixer les indices de ligne et de colonne. Par exemple, puisque x_7 appartient à la deuxième ligne et la quatrième colonne, on ajoute les contraintes $x_7 \Rightarrow r_2$ et $x_7 \Rightarrow c_3$. Afin d'empêcher deux indices de ligne ou deux indices de colonne d'être sélectionnés simultanément, on ajoute des contraintes AtMost-1 sur les c_i et sur les r_i. Ces contraintes sont ajoutées avec le *pairwise encoding* si leur taille est faible, ou avec le *two product encoding* si leur taille est plus importante, d'où la nécessité d'une étape récursive dans l'algorithme.

Dans notre exemple, les contraintes de sélection d'indices pour x_7, $x_7 \Rightarrow c_3$ et $x_7 \Rightarrow r_2$, impliquent les sélecteurs c_3 et r_2. De plus, les implications $c_3 \Rightarrow \neg c_2$ et $\neg c_2 \Rightarrow (\neg x_2 \wedge \neg x_6 \wedge \neg x_{10})$ montrent par transitivité que $x_7 \Rightarrow (\neg x_{10} \wedge \neg x_6 \wedge \neg x_2)$. Puisque toutes les implications sont construites sur des clauses binaires, le raisonnement inverse tient aussi : $x_6 \Rightarrow \neg x_7$ et $x_2 \Rightarrow \neg x_7$; on peut alors déduire $x_6 + x_7 \leq 1$ et $x_2 + x_7 \leq 1$. Cependant, la contrainte $x_2 + x_6 \leq 1$ ne peut être déduite *via* les colonnes et leurs littéraux c_2 et c_3, mais elles peuvent l'être en utilisant les lignes (avec les littéraux r_1 et r_2). Les mêmes raisonnements peuvent être produits sur les lignes.

Plus généralement, étant donnée une contrainte AtMost-1 R, où l'opposé d'un littéral $r_i \in R$ implique un littéral $\neg x_i$ ($\neg r_i \Rightarrow \neg x_i$), tel que ce littéral $\neg x_i$ implique un littéral b_i qui appartient à une autre contrainte AtMost-1 C ($\neg b_i \in C$), alors, en utilisant R comme contrainte de ligne et C comme contrainte de colonne, une contrainte AtMost-1 portant sur x_i peut être construite en cherchant les littéraux x_j manquants.

Pour chaque littéral r_i de la contrainte de ligne R, les littéraux x_i impliqués par $\neg r_i$ peuvent être

regroupés en tant que candidats à la formation d'une ligne dans la représentation *two product*. On ne considère ici que les littéraux x_i qui impliquent un littéral différent pour la contrainte C, de manière à ce que chacun des littéraux d'une ligne correspondent à des colonnes différentes dans la matrice. Les littéraux d'une ligne forment à eux seuls une contrainte *AtMost*-1, qui sera étendue lors de l'étude de la ligne suivante.

Tout ceci correspond exactement à la construction de l'encodage : si un des éléments de la nouvelle contrainte est satisfait, les sélecteurs d'indices de ligne et de colonne correspondants le seront aussi. Ceci impliquera que les autres sélecteurs d'indices devront être falsifiés, tout comme les littéraux auxquels ils sont associés.

À notre connaissance, aucun algorithme capable de détecter de telles contraintes n'a encore été proposé. Nous présentons maintenant l'algorithme 5.6, qui est capable de détecter une approximation de l'ensemble de ces contraintes. Notons que le terme *approximation* concerne ici non seulement le nombre de contraintes détectées, qui peut être plus faible que celui attendu, mais aussi l'expressivité des contraintes obtenues dans la mesure où il n'est pas assuré que les contraintes détectées soient maximales (plus précisément, on n'assure pas qu'il n'existe pas un littéral qui pourrait être ajouté à une contrainte détectée de manière à la renforcer).

Algorithme 5.6 : ExtractTwoProductEncoding

Entrées : L'ensemble de contraintes *AtMost*-1 S
Sorties : L'ensemble de contraintes S étendu

```
1  pour chaque R ∈ S faire
2  |  r = min(R) ;
3  |  l = min{l | ¬r ⇒ ¬l} ;
4  |  pour chaque c tel que ¬l ⇒ ¬c faire
5  |  |  pour chaque C ∈ {S_C | S_C ∈ S et c ∈ S_C} faire
6  |  |  |  S ← BuildAtMost-1(S,R,C) ;
7  |  |  fin
8  |  fin
9  fin
10 retourner S
```

La construction d'une nouvelle contrainte *AtMost*-1 encodée avec le *two product encoding* passe par la recherche de deux contraintes *AtMost*-1 R et C qui seront les contraintes de ligne et de colonne, et qui doivent contenir respectivement les littéraux r et c, qui doivent être utilisés en tant que sélecteurs de ligne et de colonne par des littéraux l (algorithme 5.6).

Nous procédons de la façon suivante : pour chaque contrainte R, on choisit un littéral r que l'on considère comme un sélecteur de ligne. Ensuite, on choisit le littéral l qui fera partie de notre nouvelle contrainte *AtMost*-1. Afin de réduire le temps de calcul, le littéral r choisi est le plus petit littéral de R et le littéral l est le plus petit littéral tels que $\neg r \Rightarrow \neg l$ (algorithme 5.6, lignes 2–3). Finalement, on sélectionne la contrainte C qui contient les sélecteurs de colonne.

Pour chaque paire de contraintes *AtMost*-1 R et C, une nouvelle contrainte AtMost-1 peut être bâtie en collectant tous les littéraux x_i. Cette condition peut être vérifiée en recherchant les littéraux impliqués par le complément du littéral de sélection de colonne c : $\neg c \Rightarrow \neg x_i$. De plus, un littéral x_i doit impliquer un sélecteur de ligne $r \in R$ (algorithme 5.7, lignes 2–4). Pour assurer cette propriété, un ensemble auxiliaire de littéraux `hitSet` est utilisé pour sauvegarder tous les littéraux sélecteurs de ligne (donc, de R) durant l'analyse de chacune des colonnes. Si, pour une colonne c, et un littéral x_i, un *nouveau* sélecteur $t \in$ `hitSet` est trouvé (algorithme 5.7, ligne 6), alors l'ensemble `hitSet` des littéraux est mis

Algorithme 5.7 : BuildAtMost-1

```
   Entrées : L'ensemble de contraintes AtMost-1 S, une contrainte de ligne R, une contrainte de
             colonne C
 1 A = ∅ ;                                          // AtMost-1 basée sur R et C
 2 pour chaque k ∈ C faire
 3 |   hitSet = R ;       // pour parcourir chaque littéral une unique fois
 4 |   pour chaque x_i tel que ¬k ⇒ ¬x_i, x_i ∉ A faire
 5 |   |   pour chaque t tel que ¬x_i ⇒ ¬t faire
 6 |   |   |   if t ∈ hitSet then
 7 |   |   |   |   hitSet ← hitSet \ {t} ;
 8 |   |   |   |   A ← A ∪ {x_i} ;
 9 |   |   fin
10 |   fin
11 fin
12 S ← S ∪ A ;
```

à jour en retirant t, et la contrainte *AtMost*-1 en construction est mise à jour en ajoutant x_i (algorithme 5.7, lignes 8–9).

Les étapes de la détection d'une contrainte sont synthétisées à la figure 5.6.

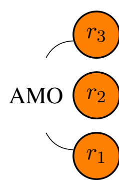
Détection de contraintes *AtMost*-2

Pour un faible nombre de littéraux et un degré faible, par exemple $k = 2$, le *naïve encoding* est compétitif. Nous présentons donc une méthode pour extraire ces contraintes depuis des clauses ternaires.

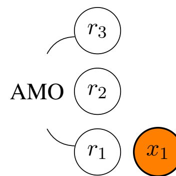
De manière similaire à l'extraction syntaxique des contraintes *AtMost*-1, on analyse les contraintes ternaires à l'aide d'un algorithme glouton. Nous considérons d'abord un littéral initial n qui n'apparaît pas dans une contrainte *AtMost*-2 déjà extraite, on considère toutes les clauses ternaires contenant $\neg n$, et l'ensemble des littéraux candidats S est initialisé avec l'ensemble des littéraux qui apparaissent niés au moins deux fois dans ces clauses. Si, à un moment dans l'algorithme, cet ensemble contient moins de 4 littéraux, la recherche est vaine et on passe à un nouveau littéral initial. Sinon, on teste chaque triplet de littéraux de S et on regarde s'il existe dans la formule une clause ternaire les liant. Si ce test échoue, on retire de l'ensemble des candidats un des trois littéraux. Si $|S| \geq 4$ et tous les triplets de S sont soutenus par une clause, la contrainte $\sum_{l \in S} l \leq 2$ est ajoutée.

5.5.2 Détection sémantique de contraintes AtMost-k

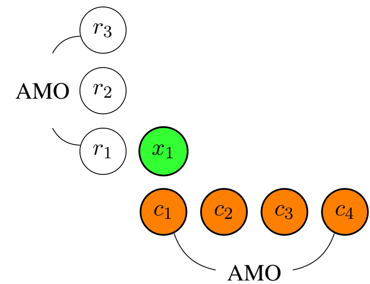
Une autre approche qui peut être suivie pour détecter des contraintes de cardinalité est d'utiliser la propagation unitaire, dans l'esprit de [Le Berre 2001]. L'avantage d'utiliser une approche « sémantique » plutôt qu'une approche « syntaxique » est que cela nous permet de détecter des contraintes sans nécessiter une approche spécifique pour chacun des encodages, au prix de procéder à des phases de propagation unitaires plutôt que de parcourir un NAG. Cette approche nous donne la possibilité de détecter des contraintes tant que l'encodage utilisé préserve l'arc-cohérence par propagation unitaire (dit autrement, il faut que la sélection de k littéraux d'une contrainte ayant un degré de k provoque la propagation des littéraux opposés à ceux de la contrainte qui ne sont pas sélectionnés). De plus, notre algorithme est capable de détecter des contraintes de cardinalité qui n'aurait pas été connues lors de la phase d'encodage du problème. Toutefois, les variables additionnelles ajoutées pour certains encodages peuvent altérer la recherche et nous faire découvrir des contraintes tronquées (comme le montre l'exemple 5.20).



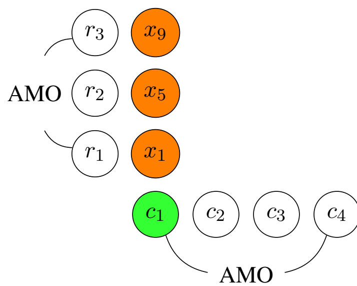
(a) Choix d'une contrainte de ligne.



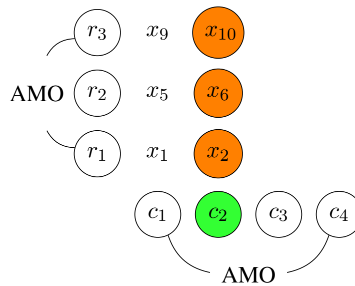
(b) Détermination d'un littéral de la matrice.



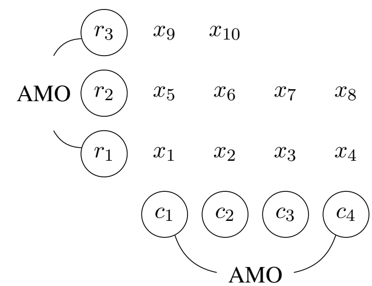
(c) Détermination d'une contrainte de colonne.



(d) Détermination des littéraux de la matrice correspondant au premier sélecteur de colonne.



(e) Détermination des littéraux de la matrice correspondant au deuxième sélecteur de colonne.



(f) Détermination des littéraux de la matrice correspondant aux sélecteurs de colonne restants.

FIGURE 5.6 – Détection de la contrainte AtMost-1 $\sum_{i=1}^{10} x_i \leq 1$ encodée via le two product encoding.

Le déroulement général de notre algorithme est le suivant : en partant d'une contrainte $\sum_{i=1}^{n} l_i \leq k$, on tente de l'étendre en ajoutant de nouveaux littéraux m tels que $(\sum_{i=1}^{n} l_i) + m \leq k$.

Notre contribution est un algorithme qui détecte des contraintes de cardinalité dans les CNF en utilisant la propagation unitaire, de telle sorte que ces contraintes ne puissent plus être étendues par des littéraux.

Extension d'une contrainte de cardinalité par un littéral

L'idée de notre algorithme est la suivante : étant donnée une clause $cl = l_1 \vee l_2 \vee \ldots \vee l_n$, nous souhaitons vérifier que cette contrainte fait partie d'une contrainte de cardinalité $cc = \sum_{i=1}^{n} \neg l_i + \sum_j m_j \leq n-1$. En effet, nous savons que $cl = l_1 \vee l_2 \vee \ldots \vee l_n \equiv \sum_{i=1}^{n} l_i \geq 1 \equiv \sum_{i=1}^{n} \neg l_i \leq n-1 = cc'$, et nous cherchons donc les littéraux m_j pour étendre cc'.

Retournons à notre exemple du *nested encoding* en considérant la CNF $\alpha = \neg x_1 \vee \neg x_2$,$\neg x_1 \vee \neg x_3$, $\neg x_2 \vee \neg x_3$,$x_3 \vee \neg x_4$, $x_3 \vee \neg x_5$,$\neg x_4 \vee \neg x_5$. La clause $\neg x_1 \vee \neg x_2$ représente la contrainte de cardinalité $x_1 + x_2 \leq 1$. Si nous propageons un des deux littéraux x_1 ou x_2 dans α, on remarque que dans les deux cas, les littéraux $\neg x_3, \neg x_4, \neg x_5$ sont propagés. Ceci implique que l'on peut étendre $x_1 + x_2 \leq 1$ par un des littéraux x_3, x_4 ou x_5, c'est-à-dire que les contraintes $x_1 + x_2 + x_3 \leq 1$, $x_1 + x_2 + x_4 \leq 1$,$x_1 + x_2 + x_5 \leq 1$ sont des conséquences de la formule α. Plus généralement, si toutes les combinaisons valides maximales de littéraux de cc' impliquent un littéral $\neg m$, m peut être ajouté à cc'. Nous exploitons ici la propriété suivante.

Proposition 5.7 *Soit α une CNF. On note $\alpha(S)$ l'ensemble des littéraux propagés dans α sous les hypothèses S. Soit $cc = \sum_{i=1}^{n} l_i \leq k$ une contrainte de cardinalité, $L = \{l_i \mid 1 \leq i \leq n\}$ et $L_k = \{S | [S \subseteq L] \wedge [|S| = k]\}$. Si $\alpha \models cc$ et $\forall S \in L_k, \alpha(S) \models \neg m$, alors $\alpha \models (\sum_{i=1}^{n} l_i) + m \leq k$.*

Preuve *Soit ω un modèle de α ,$\alpha \models \sum_{i=1}^{n} l_i \leq k$ et $\forall S \in L_k, \alpha(S) \models \neg m$. Supposons que ω ne soit pas un modèle de $\alpha \wedge (\sum_{i=1}^{n} l_i) + m \leq k$. Cela implique qu'au moins $k+1$ littéraux de $\{l_1, ..., l_n\}$ sont satisfaits. Puisque $\alpha \models \sum_{i=1}^{n} l_i \leq k$, m doit être lui aussi satisfait, ce qui est contradictoire avec l'hypothèse de départ $\forall S \in L_k, \alpha(S) \models \neg m$.*

Si plusieurs littéraux sont candidats à l'extension, il n'est pas correct de tous les ajouter d'une traite à cc'. Dans notre exemple, x_3, x_4 et x_5 sont candidats à l'extension de $x_1 + x_2 \leq 1$, mais la contrainte $x_1 + x_2 + x_3 + x_4 + x_5 \leq 1$ n'est pas conséquence de α. Nous devons aussi faire attention aux clauses unitaires de la formule originale : ces littéraux sont par définition candidats à l'extension, mais si nous en ajoutons un à notre contrainte, seuls les littéraux de ce type pourront ensuite étendre notre contrainte.

Considérons la formule $\neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_3 \vee \neg x_2, \neg x_4$ et supposons que nous choisissions la clause $\neg x_1 \vee \neg x_2$ comme contrainte de cardinalité de départ ($x_1 + x_2 \leq 1$). Deux littéraux sont candidats à l'extension : x_3 et x_4. Si nous choisissons x_4, la contrainte ne sera pas plus expressive étant donné que nous savons que la variable x_4 doit être falsifiée. Notons aussi que si la propagation unitaire conduit à un conflit, il n'est pas nécessaire de filtrer les candidats étant donné que tous les littéraux sont impliqués par une formule incohérente. Dans ce cas, nous devons simplement ne pas sélectionner pour l'extension un littéral dont le complémentaire se trouve déjà dans la contrainte de cardinalité. Cette vérification est effectuée à la ligne 1 de l'algorithme 5.8.

L'algorithme 5.8 exploite la proposition 5.7 pour trouver le complément d'un littéral qui peut étendre une contrainte de cardinalité. L'ensemble `candidats` garde l'ensemble des candidats (les littéraux dont la négation peut étendre la contrainte). À chaque phase de propagation unitaire, seuls les littéraux propagés sont gardés. Une fois ces phases passées, seuls les littéraux propagés à chaque étape sont présents dans `candidats`, ces littéraux peuvent donc étendre la contrainte de cardinalité.

Algorithme 5.8 : SearchCandidates

Entrées : une CNF α, une contrainte de cardinalité $\sum_{i=1}^{n} l_i \leq k$
Sorties : un ensemble de candidats m tels que $(\sum_{i=1}^{n} l_i) + \neg m \leq k$

```
1 candidats ← {v_i|v_i ∈ Vars(α)} ∪ {¬v_i|v_i ∈ Vars(α)} \ {¬l_i};
2 pour chaque S ⊆ {l_i} tel que |S| = k faire
3     propagés ← propagationUnitaire(α, S) ;
4     si ⊥ ∉ propagés alors
5         candidats ← candidats ∩ propagés ;
6         si candidats = ∅ alors
7             retourner ∅;
8 fin
9 retourner candidats ;
```

Lemme 5.8 *Soit α une CNF et $cc = \sum_{i=1}^{n} l_i \leq k$ une contrainte de cardinalité telle que $\alpha \models cc$. $\forall m \in$ SearchCandidates(α,cc), $\alpha \models (\sum_{i=1}^{n} l_i) + \neg m \leq k$.*

Extension maximale d'une contrainte de cardinalité

En pratique, nous n'allons pas apprendre toutes les contraintes que nous allons découvrir, mais uniquement celles qui ne peuvent plus être étendues. De plus, si une contrainte correspond en fait à une clause, nous la gardons sous forme clausale. L'algorithme 5.8 calcule l'ensemble des littéraux propagés par tous les ensembles L_k. Si l'ensemble retourné par cet algorithme est vide, on ne peut plus étendre notre contrainte ; s'il est en revanche non vide, il est possible d'étendre notre contrainte avec un des littéraux de l'ensemble.

Lemme 5.9 *Soit α une CNF. $\forall c \in \alpha$, $\alpha \models$ ExpandCardFromClause(α, c).*

De manière itérative, nous recherchons un littéral candidat à l'extension, nous l'ajoutons à la contrainte, et répétons ces étapes tant qu'un candidat est trouvé. À partir de la seconde itération, il n'est plus nécessaire de calculer tous les ensembles L_k, puisque certains auront été calculés lors d'itérations antérieures. Plus précisément, pour trouver le $n^{ème}$ littéral d'une contrainte, nous avons déjà calculé $\binom{n-1}{k}$ des $\binom{n}{k}$ phases de propagations nécessaires pour l'appel courant de l'algorithme 5.8. Les seules phases manquantes sont celles où figure parmi les hypothèses le dernier candidat ajouté. Ce processus est décrit dans l'algorithme 5.9.

Grâce à cette remarque, nous proposons l'algorithme 5.10 qui calcule efficacement des contraintes de cardinalité maximales.

Le calcul des $\binom{n}{k}$ phases de propagation unitaire est la partie difficile de l'algorithme.

Lemme 5.10 *Soit α une CNF avec n variables et l littéraux. Soit c une clause de α de taille $|c| = k + 1$. ExpandCardFromClause(α, c) a une complexité temporelle en $O(\binom{n}{k} \times l)$.*

Retrait des clauses dominées

La dernière étape dans notre approche consiste à détecter les clauses qui sont conséquences de contraintes découvertes, et à les retirer dans une optique d'efficacité du solveur. Le but est d'obtenir une formule équivalente à l'originale en retirant un maximum de clauses qui sont devenues inutiles suite à la découverte de contraintes de cardinalité.

Algorithme 5.9 : RefineCandidates

Entrées : une CNF α, une contrainte $\sum_{i=1}^{n} l_i + l_{new} \leq k$, un ensemble de littéraux L
Sorties : un ensemble de candidats m tels que $(\sum_{i=1}^{n} l_i) + l_{new} + \neg m \leq k$

1 candidats $\leftarrow L$;
2 **pour chaque** $\mathsf{S}' = \mathsf{S} \cup \{l_{new}\}$ *tel que* $\mathsf{S} \subseteq \{l_i\}$ *et* $|S| = k - 1$ **faire**
3 | propagés $\leftarrow$ `propagationUnitaire`(α, S') ;
4 | **si** $\perp \notin$ propagés **alors**
5 | | candidats $\leftarrow$ candidats $\cap$ propagés ;
6 | | **si** candidats $= \emptyset$ **alors**
7 | | | **retourner** $\emptyset$;
8 **fin**
9 **retourner** candidats ;

Algorithme 5.10 : ExpandCardFromClause

Entrées : une CNF α, une clause c
Sorties : une contrainte de cardinalité cc ou c

1 cc $\leftarrow \sum_{l \in c} \neg l \leq |c| - 1$;
2 candidats $\leftarrow$ `SearchCandidates`(α, cc) ;
3 **tant que** candidats $\neq \emptyset$ **faire**
4 | sélectionner m dans candidats ;
5 | $cc \leftarrow \sum_{l_i \in cc} l_i + \neg \mathsf{m} \leq |c| - 1$;
6 | candidats $\leftarrow$ `RefineCandidates`$(\alpha, \mathsf{cc}, \text{candidats} \setminus \{\mathsf{m}\})$;
7 **fin**
8 **si** $|cc| > |c|$ **alors retourner** cc ;
9 **sinon retourner** c ;

Nous utilisons pour cela la règle décrite dans [BARTH 1993], utilisée dans 3MCard [VAN LAMBALGEN 2006], pour déterminer si une clause (exprimée sous la forme d'une contrainte AtMost-1) est dominée par une contrainte découverte. Cette règle indique que, étant données deux contraintes de cardinalité, $L \geq d$ domine $L' \geq d'$ si et seulement si $|L \setminus L'| \leq d - d'$. Ainsi, afin de tenter d'étendre une contrainte sous forme clausale, on utilise cette règle pour déterminer si elle n'est pas dominée par une contrainte révélée. Dans ce cas, nous ne traitons pas cette contrainte et nous la retirons du problème. Nous retirons aussi du problème les clauses qui ont été étendues.

Un autre point important pour éviter la redondance de contraintes est de considérer en premier les clauses les plus petites comme candidates à l'extension. De ce fait, nous découvrons des contraintes dont les degrés sont croissants, et une contrainte découverte ne peut donc pas être conséquence d'une contrainte découverte antérieurement si les littéraux sont communs.

Puisque les nouvelles contraintes sont conséquences de la formule, et que les contraintes enlevées sont conséquences de contraintes ajoutées, l'algorithme assure que la nouvelle formule est logiquement équivalente à l'originale. D'où la proposition suivante.

Proposition 5.11 *Soit α une CNF et k un entier donné.*

$$\alpha \equiv \textit{DetectConstraintsInCNF}(\alpha, k)$$

Dans notre exemple sur le *nested encoding*, notre approche fonctionne comme suit. Tout d'abord, nous tentons d'étendre $x_1 + x_2 \leq 1$. Nous déterminons que cette contrainte peut être étendue en $x_1 +$

Algorithme 5.11 : DetectConstraintsInCNF

Entrées : une CNF α et une borne k

Sorties : une formule $\phi \equiv \alpha$ contenant des contraintes de cardinalité de degré $\leq k$ et des clauses

```
1 φ ← ∅ ;
2 pour chaque clause c ∈ α de taille |c| croissante telle que |c| ≤ k + 1 faire
3 │ si aucune contrainte révélée cc ∈ φ ne domine c alors
4 │ │ φ ← φ ∪ ExpandCardFromClause(α, c) ;
5 │ fin
6 fin
7 retourner φ ;
```

$x_2 + x_3 \leq 1$ ou $x_1 + x_2 + x_4 + x_5 \leq 1$. Supposons que la deuxième contrainte est trouvée. La CNF est réduite en enlevant les clauses dominées par cette contrainte, à savoir $\neg x_1 \vee \neg x_2$ et $\neg x_4 \vee \neg x_5$. On cherche ensuite à étendre $x_1 + \neg x_3 \leq 1$, on obtient la contrainte $x_1 + x_2 + x_3 \leq 1$. Ceci nous permet de retirer la clause dominée $\neg x_2 \vee \neg x_3$. Ensuite, $\neg x_3 + x_4 \leq 1$ est étendue en $\neg x_3 + x_4 + x_5 \leq 1$. Les clauses restantes, toutes dominées, sont retirées de la formule, et l'algorithme s'arrête puisqu'il n'y a plus de clauses à étudier. Notons que si la première contrainte découverte avait été $x_1 + x_2 + x_3 \leq 1$, l'algorithme n'aurait pas été capable de révéler $x_1+x_2+x_4+x_5 \leq 1$ car toutes les clauses contenant $\neg x_1$ auraient été retirées. De même, en ce qui concerne les encodages utilisant des variables additionnelles, le fait de considérer ces variables pour l'extension d'une contrainte peut mener à découvrir une contrainte « tronquée », comme le montre l'exemple ci-dessous.

Exemple 5.20 (Détection de contrainte tronquée à cause d'une variable additionnelle) *Supposons que l'on encode la contrainte $x_1 + x_2 + x_3 + x_4 \leq 1$ en utilisant le nested encoding en ajoutant pour cela la variable y_1 :*

$$x_1 + x_2 + y_1 \leq 1,\ x_3 + x_4 - y_1 \leq 0.$$

L'encodage final en conjonction de clauses est le suivant :

$$\begin{array}{ccc} \neg x_1 \vee \neg x_2 & \neg x_1 \vee \neg y_1 & \neg x_2 \vee \neg y_1 \\ \neg x_3 \vee \neg x_4 & \neg x_3 \vee y_1 & \neg x_4 \vee y_1. \end{array}$$

En étendant la première clause, et donc en partant de la contrainte $x_1 + x_2 \leq 1$, on pourra choisir un littéral pour l'extension parmi y_1, x_3, ou x_4. Si on choisit x_3, on pourra par la suite étendre $x_1+x_2+x_3 \leq 1$ en $x_1 + x_2 + x_3 + x_4 \leq 1$ et ainsi retrouver la contrainte initiale. En revanche, si on cherche à étendre $x_1 + x_2 + y_1 \leq 1$, on ne peut pas découvrir une contrainte au moins aussi générale que la contrainte initiale (puisque $x_1 + x_2 + y_1 + x_3 + x_4 \leq 1$ n'est pas conséquence de la conjonction de clauses).

En ce qui concerne la complexité de l'algorithme, le pire cas est atteint lorsque l'on cherche à étendre toutes les clauses, d'où la proposition suivante.

Proposition 5.12 *Soit α une CNF avec n variables, m clauses and l littéraux. Soit k un entier tel que $0 < k \leq n$ et $m_k \leq m$ le nombre de clauses de taille $\leq k + 1$ dans α. DetectConstraintsInCNF(α, k) a une complexité temporelle en $O(m_k \times \binom{n}{k} \times l)$.*

5.5.3 Détection sémantique : preprocessing *vs.* inprocessing

Dans les sections précédentes, nous avons considéré la détection de contraintes de cardinalité lors d'un prétraitement. Nous avons aussi étudié la possibilité que ce traitement soit réalisé pendant la recherche, ce qui n'a malheureusement pas conduit à des résultats encourageants.

Nous nous sommes en particulier intéressés aux clauses apprises, dans le but de déterminer si ces clauses étaient oui ou non des conséquences de contraintes de cardinalité. Or, nous nous sommes rendu compte que cette approche est moins efficace sur nos jeux d'essai que celle à base de prétraitement.

Prenons par exemple le cas d'une instance du problème des pigeons/pigeonniers. Au cours de la recherche, le prouveur sans prétraitement arrive à un conflit pour lequel la raison est calculée en appliquant la résolution généralisée pour l'ensemble de contraintes $\{x_4 - x_2 \geq 0, x_4 - x_3 \geq 0, x_1 + x_2 + x_3 \geq 2\}$. À partir de ces données, le prouveur détermine que la raison du conflit est $2x_4 + x_1 \geq 2$. Cette contrainte ne peut pas être étendue, mais elle indique en revanche que le littéral x_4 est conséquence de la formule.

Si les contraintes de cardinalité avaient été détectées, le même conflit aurait été expliqué par les contraintes $x_1 + x_2 + x_3 \geq 2$ et $x_4 - x_2 - x_3 \geq 2$, dont le prouveur aurait déduit la nouvelle contrainte $x_4 + x_1 \geq 2$. De ce fait, le prouveur déduirait de ce conflit que non seulement x_4 est conséquence de la formule, mais aussi que x_1 est lui-aussi conséquence. Cette observation est très importante, dans la mesure où elle montre qu'il n'est pas suffisant d'utiliser notre approche de détection sémantique, telle qu'elle est actuellement définie, au cours de l'analyse de conflits ; ceci justifie son emploi lors d'un prétraitement. On pourrait justement tenter d'améliorer cette méthode dans le but qu'elle fonctionne au cours de la recherche, lorsque cela est nécessaire, afin d'économiser du temps de calcul.

5.5.4 Résultats expérimentaux

Les résultats expérimentaux montrent que les méthodes que nous avons proposées détectent un nombre important de contraintes de cardinalité. Notre analyse empirique a été réalisée sur des jeux d'essai académiques, comme des grilles de Sudoku ou des instances du problème des pigeons, pour lesquels nous connaissons le nombre de contraintes à découvrir dans la CNF, et qui sont simples à résoudre en utilisant la résolution généralisée quand les contraintes sont des contraintes de cardinalité. Nos expérimentations ont été effectuées sur des machines dont les processeurs sont des Intel Xeon (@2.66GHz) avec 32Go de RAM ; les temps limites d'exécution ont été fixés à 900s.

L'approche statique est implémentée dans la dernière version de Lingeling [BIERE 2013]. L'approche syntaxique associée à la reconnaissance du *two product encoding*, ainsi que l'approche sémantique, ont été implémentées dans Riss [MANTHEY 2012]. Puisque Riss ne sait pas tirer avantage des contraintes de cardinalité, nous l'avons utilisé comme un préprocesseur rapide pour fournir le nouveau problème à Sat4j [LE BERRE & PARRAIN 2010], qui utilise la résolution généralisée. Cela nous permet de vérifier si les contraintes détectées sont suffisantes pour résoudre ces *benchmarks*. Nous avons aussi comparé nos approches avec SBSAT [3].

Problème des pigeons

Ces *benchmarks* sont connus pour être extrêmement difficiles pour les solveurs basés sur la résolution [HAKEN 1985]. Pour $n+1$ pigeons et n pigeonniers, le problème est d'affecter à chaque pigeon un pigeonnier en ayant un pigeon par case au maximum. Chaque variable booléenne $x_{i,j}$ représente l'information « le pigeon i est dans le pigeonnier j ». Le problème est représenté à l'aide de $n+1$ clauses $\bigvee_{j=1}^{n} x_{i,j}$ et n contraintes de cardinalité $\sum_{j=1}^{n+1} x_{i,j} \leq 1$. Nous avons généré ces *benchmarks* pour n allant de 10 à 15 par pas de 1, et n allant de 25 à 200 par pas de 25 en utilisant six encodages différents : *binomial*, *product*, *binary*, *ladder*, *commander* et *sequential*. Les résultats sont donnés à la figure 5.7.

Comme nous pouvions nous y attendre, Sat4j est incapable de résoudre la plupart des *benchmarks* quand les problèmes sont sous forme clausale (dans ce cas, la résolution généralisée est équivalente à

3. Nous avons tenté d'inclure 3MCard dans nos résultats, mais il n'a malheureusement pas été capable de lire ou de résoudre la plupart de nos fichiers de test.

Prétraitement Solver	#inst.	Lingeling Lingeling	Synt.(Riss) Sat4jCP	Sem.(Riss) Sat4jCP	∅ SBSAT	∅ Sat4jCP
Pairwise	14	**14 (3s)**	13 (244s)	14 (583s)	6 (0s)	1 (196s)
Binary	14	3 (398s)	2 (554s)	**7 (6s)**	6 (7s)	2 (645s)
Sequential	14	0 (0s)	**14 (50s)**	**14 (40s)**	10 (6s)	1 (37s)
Product	14	0 (0s)	**14 (544s)**	11 (69s)	6 (25s)	2 (346s)
Commander	14	1 (3s)	7 (0s)	**14 (40s)**	9 (187s)	1 (684s)
Ladder	14	0 (0s)	11 (505s)	11 (1229s)	**12 (26s)**	1 (36s)

FIGURE 5.7 – Six encodages du problème des pigeons/pigeonniers : nombre d'instances résolues et somme des temps de calcul (prétraitement et prouveur) pour les instances résolues entre parenthèses.

Prétraitement Solver	#inst.	Lingeling Lingeling	Synt.(Riss) Sat4jCP	Sem.(Riss) Sat4jCP	∅ SBSAT	∅ Sat4jCP
Sgen unsat	13	0 (0s)	**13 (0s)**	**13 (0s)**	9 (614s)	4 (126s)
Fixed bandwidth	23	2 (341s)	**23 (0s)**	**23 (0s)**	**23 (1s)**	13 (1800s)
Rand. orderings	168	16 (897s)	**168 (7s)**	**168 (8s)**	99 (2798s)	69 (3541s)
Rand. 4-reg.	126	6 (1626s)	**126 (4s)**	**126 (5s)**	84 (2172s)	49 (3754s)

FIGURE 5.8 – Plusieurs familles de problèmes hautement combinatoires : nombre d'instances résolues et somme des temps de calcul (prétraitement et solver) pour les instances résolues entre parenthèses.

la résolution standard). La détection sémantique découvre beaucoup de contraintes et permet au solveur de résoudre plus d'instances que les autres solveurs (ceci est particulièrement vrai pour les encodages *pairwise*, *sequential* et *commander*). Notons que l'instance pairwise pour $n = 200$ contient $402\,000$ variables et 4 020 201 clauses, ce qui semble démontrer que notre approche résiste au passage à l'échelle. L'analyse statique associée à la reconnaissance du *two product encoding* se débrouille elle-aussi très bien sur la plupart des encodages, et est la meilleure pour le *two product encoding*, comme attendu. Les encodages *binary* et *ladder* sont plus difficiles à détecter avec nos approches. SBSAT est bien moins efficace (excepté pour le *ladder*). Lingeling n'est efficace que sur le *pairwise* encoding, comme nous l'avions prévu.

Benchmarks hautement combinatoires

Ces *benchmarks* incohérents de *balanced block design* sont décrits dans [SPENCE 2010, VAN GELDER & SPENCE 2010] ; ils contiennent des contraintes de cardinalité AtMost-2. Nous utilisons les *benchmarks* soumis à la compétition SAT09 (alors appelés *sgen*) qui sont les plus petites instances incohérentes de *benchmarks* hautement combinatoires, ainsi que des *benchmarks* fournis par Mladen Miksa et Jakob Nordström de KTH [MIKSA & NORDSTRÖM 2014]. Nos deux approches permettent de récupérer toutes les contraintes et de résoudre de manière quasiment instantanée tous ces *benchmarks*, comme montré à la figure 5.8.

Les solveurs Lingeling et Sat4j sans prétraitement, qui n'utilisent pas la détection de contraintes de cardinalité, montrent que ces fichiers de test sont très difficiles pour les solveurs actuels (ce comportement a été confirmé lors des compétitions SAT).

Benchmarks de Sudoku

Les *benchmarks* de Sudoku contiennent uniquement des contraintes de cardinalité $= 1$ représentées par une clause et une contrainte *AtMost*-1. Ces instances sont triviales à résoudre ; leur intérêt vient du fait que les différentes contraintes de cardinalité partagent un grand nombre de littéraux, ce qui pourrait gêner nos opérations de prétraitement. Nous utilisons deux instances de grilles vides $n \times n$, pour $n = 9$ et $n = 16$. La grille contient $n^2 \times 4$ contraintes AtMost-1 : pour $n = 9$ (resp. 16), il y a 324 (resp. 1024) contraintes (toutes encodées via le *pairwise encoding*).

Les contraintes *AtMost*-1 détectées par l'approche syntaxique contiennent bien toutes 9 (resp. 16) littéraux, cependant certaines sont manquantes : 300/324 (resp. 980/1024) contraintes ont été découvertes par cette approche. L'approche sémantique, quant à elle, découvre toutes les contraintes. Cela vient du fait que toutes les contraintes possèdent au moins une clause binaire qui leur est propre, et qui n'est donc pas retirée quand une autre contrainte a été trouvée ; cette observation est importante quand elle est mise en perspective avec le problème de contraintes oubliées pointé à la section traitant du retrait des clauses dominées. Cette clause est ensuite étendue pour donner la contrainte à laquelle elle appartient.

5.5.5 Conclusion à propos des approches proposées

Nous avons présenté deux approches capables de retrouver des contraintes de cardinalité dans des CNF. La première est basée sur une analyse syntaxique des clauses ternaires et du *NAND graph*, une structure de données utilisée dans certains solveurs pour gérer les contraintes binaires de manière efficace, et permet de récupérer des contraintes *AtMost*-1 et *AtMost*-2. La seconde, basée sur la propagation unitaire dans la formule, est un algorithme générique capable de retrouver des contraintes *AtMost*-k de tailles quelconques. Nous montrons que ces deux approches sont capables de retrouver des contraintes dans des problèmes en CNF connus pour être hautement combinatoires. Nos expérimentations suggèrent que l'approche syntaxique est particulièrement utile pour les contraintes AtMost-1 et AtMost-2, quand l'approche sémantique semble plus robuste au sens où elle permet de récupérer des contraintes de degré arbitraire.

Nos approches permettent de résoudre les plus petits *benchmarks* UNSAT non résolus de la compétition SAT2009 (sgen). Ces jeux d'essai ont tous été résolus dans la seconde par la version de Sat4j basée sur la résolution généralisée une fois que les contraintes *AtMost*-2 ont été mises à jour (ceci avait déjà été remarqué dans [WEAVER 2012]). La différence entre nos deux approches est néanmoins visible avec le *benchmark-challenge* de [VAN GELDER & SPENCE 2010], qu'aucun solveur n'avait pu résoudre en l'espace d'une journée. Cette instance est résolue en une seconde, après avoir révélé les 22 contraintes *AtMost*-2 et les 20 contraintes *AtMost*-3 grâce à l'approche sémantique (l'approche syntaxique n'est pas capable de retrouver les contraintes *AtMost*-3).

Nous avons aussi été capables de révéler des contraintes de cardinalité sur de grands *benchmarks* applicatifs. Cependant, l'utilisation de ces informations pour améliorer le temps de calcul des solveurs est laissée à un travail futur. Vérifier que ces contraintes étaient connues au moment de l'encodage, ou déterminer que ces contraintes étaient « cachées » est un problème ouvert. Une question intéressante, bien qu'en dehors du cadre de ce travail, concerne la puissance du système de preuve résultant de la combinaison de notre approche sémantique (règles d'extension et de domination) et de la résolution généralisée.

5.6 Conclusion du chapitre

Dans ce chapitre, nous avons étudié les prouveurs SAT modernes. Nous avons dans un premier temps exposé l'historique de ces programmes, avant de décrire les composants clés des prouveurs dits mod-

ernes.

Nous avons en particulier décrit les prouveurs pseudo-booléens, pour lesquels on peut utiliser un système de preuves plus puissant que celui de la résolution, que l'on retrouve dans la majorité des prouveurs SAT complets actuels. Ce système de preuve, la résolution généralisée, permet de fournir en théorie des preuves d'incohérence de manière bien plus efficace, à condition que les contraintes utilisées soient plus expressives que de simples clauses. Or, ces contraintes plus générales, les contraintes pseudo-booléennes, sont malheureusement quelquefois diluées par un encodage sous forme de clauses, ou bien sans doute aussi quelquefois des conséquences de la formule que l'utilisateur n'aurait pas pu imaginer (ce qui est difficile à vérifier).

Nous avons présenté une de nos contributions, à savoir des algorithmes capables de découvrir des contraintes de cardinalité dans une formule CNF. Nous avons montré que les approches présentées permettent un gain d'efficacité très important en ce qui concerne certaines familles d'instances, conduisant même à résoudre des *benchmarks* qui n'avaient à l'heure actuelle pas encore été résolus par des outils (à notre connaissance).

Bien que les prouveurs actuels basés sur la résolution généralisée soient aujourd'hui globalement en retrait des prouveurs classiques en ce qui concerne leur efficacité, nos approches sont importantes dans la mesure où, en plus d'avoir permis de fournir un outil plus performant que les autres dans certains cas particuliers, elles permettent d'améliorer des prouveurs qui sont théoriquement plus efficaces que les meilleurs prouveurs d'aujourd'hui. Ceci est déjà vrai pour les problématiques de décision, mais c'est *a fortiori* encore plus vrai en ce qui concerne les problématiques d'optimisation. En effet, concernant les problèmes de décision, l'utilisation d'un système de preuve plus puissant n'aura *a priori* pas autant d'impact sur une instance cohérente que sur une instance incohérente. Or, en ce qui concerne l'optimisation sur les formules CNF, les preuves basées sur les prouveurs SAT contiennent toujours une preuve d'incohérence, comme nous le verrons dans le chapitre suivant. De ce fait, obtenir des preuves de l'incohérence d'un problème de décision plus rapidement est fort utile en pratique.

Chapitre 6

Optimisation par prouveurs pseudo-booléens

Sommaire

6.1 Optimisation linéaire incrémentale par contrainte de borne 128
6.2 Cas des fonctions d'optimisation pseudo-booléennes non linéaires 131
6.2.1 Réduction vers des fonctions linéaires par ajout de contraintes 131
6.2.2 Optimisation par énumération des solutions optimales d'une minorante . . 133
6.3 Génération de contraintes paresseuses pour l'optimisation 135
6.3.1 Motivation 135
6.3.2 Optimisation par contraintes de borne dynamiques 138
6.3.3 Expérimentations et résultats 140
6.3.4 Analyse des résultats 144
6.3.5 Conclusions à propos des approches proposées 144
6.4 De la correction des prouveurs 147
6.5 Conclusion du chapitre 148

Dans la première partie de ce document, nous nous sommes intéressés à la complexité algorithmique de l'optimisation pour un certain nombre de langages de représentation de contraintes, et nous avons montré que la traduction d'un problème formalisé en CNF vers un langage comme DNNF pouvait être intéressante pour un nombre non négligeable de fonctions objectifs.

En revanche, dans le cas où l'on ne procéderait pas à un nombre important de requêtes d'optimisation concernant un même ensemble de contraintes, le coût de la compilation est un frein à la traduction d'une formule CNF vers un formalisme permettant l'optimisation de façon peu complexe. Dans ce cas, la meilleure piste reste de procéder à l'optimisation de la fonction objectif considérée en gardant le formalisme initial dans lequel les contraintes ont été fournies.

Dans ce chapitre, nous traitons donc de l'optimisation de fonctions objectifs lorsque les contraintes sont données sous forme CNF, ou bien lorsqu'elles sont données sous forme de contraintes pseudo-booléennes (PB). Nous présentons d'abord l'algorithme général d'optimisation d'une fonction linéaire sous contraintes PB. Nous présentons ensuite un certain nombre d'encodages permettant de considérer les problèmes d'optimisation de certaines fonctions non linéaires en fournissant une réduction vers des problèmes d'optimisation de fonctions linéaires (ces processus ajoutant des contraintes pseudo-booléennes). Nous étudions aussi une autre technique d'optimisation de fonctions non linéaires, qui propose d'énumérer les solutions optimales d'une fonction linéaire minorant la fonction objectif non linéaire initiale. Finalement, nous présentons une de nos contributions publiée dans la revue RIA [LE BERRE & LONCA 2014], dans laquelle nous étudions les gains potentiels d'une modification de l'architecture des prouveurs CDCL que nous avons codée dans Sat4j.

6.1 Optimisation linéaire incrémentale par contrainte de borne

D'années en années, le succès des approches basées sur SAT dans de nombreux domaines comme la vérification de matériel [BIERE 2009], la vérification de logiciels [KROENING 2009], la planification [RINTANEN 2009] a contribué à la création de prouveurs SAT robustes, efficaces et pensés pour être réutilisables dans des applications tierces. Dans de nombreux cas, ces applications utilisent l'interface incrémentale des prouveurs SAT proposée par Minisat [EÉN & SÖRENSSON 2003a, EÉN & SÖRENSSON 2003b], qui permet de résoudre avec un unique prouveur des séquences de problèmes SAT partageant des clauses. Cela permet d'utiliser les prouveurs SAT comme des boîtes noires, c'est-à-dire sans se préoccuper de leur fonctionnement interne (bien que l'état interne de ces prouveurs puisse évoluer au fur et à mesure de leur utilisation, notamment en ce qui concerne la base des clauses apprises, mais sans le contrôle de l'utilisateur). Si cette approche semble limitée à la résolution de problèmes de décision, elle est aussi utilisée avec succès dans le cadre de problèmes d'optimisation en variables booléennes, tels que l'optimisation pseudo-booléenne [ROUSSEL & MANQUINHO 2009] (optimisation d'une fonction pseudo-booléenne linéaire) ou MAXSAT [LI & MANYÀ 2009] (formule CNF incohérente dont on cherche un sous-ensemble de cardinal maximal de contraintes tel que la conjonction des contraintes est cohérente).

En ce qui concerne la minimisation de fonctions pseudo-booléennes linéaires, l'approche consiste à déterminer un majorant initial de la valeur minimale recherchée en calculant un modèle, puis à ajouter une contrainte de borne au problème initial de manière à supprimer toutes les solutions dont les valeurs pour la fonction objectif sont supérieures ou égales à celle calculée pour le modèle obtenu précédemment. On relance ensuite le prouveur, et on observe sa réponse quant à la cohérence de la formule à laquelle la contrainte de borne a été ajoutée. Si le prouveur trouve un modèle, alors on sait grâce à la nouvelle contrainte que celui-ci donne une valeur inférieure à la fonction objectif que le modèle trouvé lors de l'itération précédente ; on met alors à jour la contrainte de borne avec la valeur obtenue pour le nouveau modèle, et on relance le prouveur à la recherche d'un modèle. En revanche, dès que le prouveur retourne

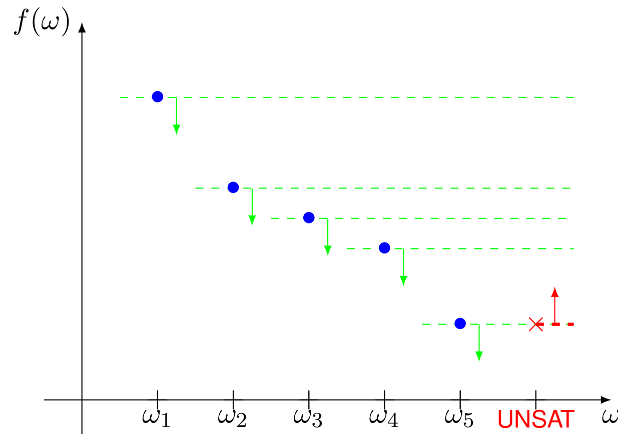


FIGURE 6.1 – Optimisation (minimisation) d'une fonction pseudo-booléenne linéaire par renforcement de contrainte de borne

une preuve de l'incohérence de la formule initiale conjointe à la contrainte de borne, on sait que le dernier modèle calculé est un modèle minimal pour la fonction objectif. L'algorithme 6.1 décrit ce processus SAT, . . ., SAT, UNSAT ; cet algorithme est illustré par la figure 6.1.

La grande force de cette approche est que l'ensemble des clauses (ou contraintes, dans le cas d'un prouveur pseudo-booléen) apprises lors de l'exécution du prouveur peut être conservé d'un appel à l'autre, ce qui permet de conserver des informations importantes qui n'ont pas besoin d'être calculées de nouveau. En effet, les clauses apprises sont des conséquences de la formule, dans la mesure où elles sont obtenues par résolution (ou résolution généralisée) à partir d'un sous-ensemble de contraintes de la formule. Or, lorsqu'on ajoute une contrainte de borne, où lorsqu'on met à jour une contrainte de borne en diminuant la valeur de la borne k, on ne fait que contraindre la formule plus qu'elle ne l'était (tous les modèles de la nouvelle formule étaient aussi des modèles pour la formule précédente). De ce fait, les conséquences de la formule précédente sont aussi des conséquences de la nouvelle formule, ce qui est en particulier le cas pour les contraintes apprises lors de la recherche.

Algorithme 6.1 : Optimisation par renforcement

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, et une fonction objectif à minimiser f

sortie : un modèle de ϕ minimal pour f, ou UNSAT si le problème n'admet pas de solution.

```
1 ⟨réponse,certificat⟩ ← estSatisfiable (φ);
2 si réponse est UNSAT alors
3 |  retourner UNSAT
4 fin
5 répéter
6 |  m ← certificat ;
7 |  ⟨réponse,certificat⟩ ← estSatisfiable (φ ∪ {f < f (m)});
8 jusqu'à (réponse est UNSAT);
9 retourner m ;
```

Plus formellement, soit ϕ une conjonction de contraintes booléennes (clauses, contraintes de cardinalité, contraintes pseudo-booléennes) et f une fonction linéaire de l'ensemble des interprétations sur

PS dans $\mathbb{N}$. Il est possible de calculer un modèle de ϕ minimisant f, en utilisant l'algorithme 6.1. Le problème d'optimisation pseudo-booléen est alors remplacé par un nombre fini de problèmes de décision pseudo-booléens. On distingue deux approches principales pour l'implémentation de cet algorithme : celle qui réutilise des prouveurs capables de raisonner avec des contraintes pseudo-booléennes nativement (des prouveurs pseudo-booléens) et celle qui réutilise des prouveurs SAT classiques et traduit la contrainte pseudo-booléenne $f < f(m)$ en CNF en utilisant des encodages que nous avons présentés au chapitre précédent. Nous présentons ces différentes approches dans le cadre de MAXSAT ; elles sont cependant tout à fait applicables au cadre de l'optimisation pseudo-booléenne linéaire.

La première approche est utilisée dans Sat4j [LE BERRE & PARRAIN 2010]. Son principal intérêt est de représenter de manière compacte ces contraintes. Sat4j traduit les problèmes MAXSAT en problèmes d'optimisation pseudo-booléens. Les résultats de Sat4j lors de la compétition MAXSAT 2009 [4] le situent dans l'état de l'art, malgré des performances sur la résolution de contraintes clausales (compétition SAT [5]) bien en deçà des autres prouveurs (dues principalement au choix du langage utilisé et au support de contraintes génériques).

La seconde approche est utilisée par QMAXSAT [KOSHIMURA *et al.* 2012], l'un des meilleurs prouveurs MAXSAT lors des récentes compétitions MAXSAT (2010-2012), en traduisant les contraintes de cardinalité (il ne gère pas les poids) sous forme de CNF [6]. QMAXSAT utilise les prouveurs SAT comme des boîtes noires : en 2012, la version utilisant Glucose 2 s'est révélée plus efficace que la version utilisant Minisat sur les instances dites « industrielles » avec contraintes dures (*Industrial Partial Max Sat*), alors que la version utilisant Minisat s'est révélée plus efficace sur les instances dites « fabriquées » avec contraintes dures (*Crafted Partial Max Sat*). Cela montre l'intérêt de l'approche boîte noire : il suffit de choisir parmi les divers prouveurs disponibles celui qui fonctionne le mieux sur le type de problème à résoudre.

Il est aussi possible de rechercher de manière dichotomique à la fois un majorant et un minorant [WHITTEMORE *et al.* 2001]. En ce qui concerne la recherche dichotomique, des majorants (résultats SAT) et des minorants (résultats UNSAT) sont calculés jusqu'à obtention de la solution optimale quand le majorant est égal au minorant. Le fait qu'une réponse UNSAT est souvent plus coûteuse à obtenir en pratique qu'une réponse SAT et que les clauses apprises ne puissent pas être gardées quand une réponse UNSAT est trouvée rend souvent l'approche dichotomique moins performante en pratique que l'approche linéaire, bien que nous ayons montré dans [LE BERRE *et al.* 2012] (une de nos contributions, publiée dans les actes de la conférence RFIA) que certaines familles de *benchmarks* cohérents se résolvent aussi bien par minimisation de la borne supérieure qu'en énumérant les coûts possibles du plus faible au plus important.

Une autre approche permettant la réutilisation des prouveurs SAT pour résoudre des problèmes d'optimisation existe, basée elle aussi sur la traduction de contraintes pseudo-booléennes en CNF : l'optimisation basée sur la détection de noyau incohérent (*unsat core guided MAXSAT solvers*) [FU & MALIK 2006a, MORGADO *et al.* 2012, ANSÓTEGUI *et al.* 2013b]. Cette approche est spécifique à la résolution de problèmes MAXSAT, mais peut permettre de résoudre d'autres problèmes d'optimisation par traduction vers MAXSAT.

L'optimisation incrémentale par contrainte de borne offre de manière générale de très bons résultats en ce qui concerne l'optimisation de fonctions objectifs linéaires basée sur un prouveur SAT. Or, dans le cas des techniques basées sur le principe SAT incrémental, le prouveur est arrêté puis relancé à de multiples reprises pour obtenir une solution ; il paraît cependant plus intéressant dans ce contexte de permettre

4. `http://maxsat.ia.udl.cat:81/09/`

5. `http://www.satcompetition.org/2009/`

6. De nombreux travaux concernent la traduction des contraintes de cardinalité et des contraintes pseudo-booléennes en CNF (voir par exemple [FRISCH & GIANNAROS 2010, ASÍN *et al.* 2011, ABÍO *et al.* 2012] pour un aperçu des travaux récents).

au prouveur de poursuivre sa recherche quand il trouve une solution, plutôt que de le laisser s'arrêter et commencer la résolution d'un nouveau problème de décision. Très rapidement des algorithmes de type évaluation et séparation ont étendu les approches SAT classiques : c'est la cas de Bsolo [MANQUINHO *et al.* 1997] par exemple qui étend GRASP [SILVA & SAKALLAH 1999] à la résolution des problèmes d'optimisation. Une approche similaire a été développée pour étendre les prouveurs de type Chaff [FU & MALIK 2006b]. Cela nécessite l'introduction au niveau du prouveur SAT d'un mécanisme permettant de pouvoir recevoir de nouvelles contraintes durant la recherche. C'est ce qui se passe par exemple dans le cadre des prouveurs SMT [BARRETT *et al.* 2009], ou lors de la génération paresseuse de contraintes en CSP [OHRIMENKO *et al.* 2009] : un prouveur SAT est adapté de manière à pouvoir recevoir des clauses pendant la recherche. Dans le cadre des problèmes d'optimisation qui nous intéressent, on pourrait souhaiter créer de manière paresseuse des contraintes de borne, c'est-à-dire ajouter des contraintes de cardinalité ou des contraintes pseudo-booléennes pendant la recherche. Nous traitons ce sujet à la section 6.3.

Finalement, bien que les approches basées sur la réutilisation de prouveurs SAT soient souvent très compétitives, elles ne sont pas toujours les plus adaptées. Les approches de type *Branch & Bound* fonctionnent généralement très bien sur les instances MAXSAT aléatoires ou fabriquées : akmaxsat [KUEGEL 2012] était le meilleur prouveur MAXSAT dans ces catégories en 2011 et 2012.

6.2 Cas des fonctions d'optimisation pseudo-booléennes non linéaires

Les approches présentées ci-dessus nécessitent d'ajouter des contraintes à chaque itération pour borner la fonction objectif. Or, puisque les prouveurs pseudo-booléens ne sont capables de ne gérer que des contraintes linéaires, ceci implique que l'algorithme d'optimisation par contrainte de borne, tel que présenté, n'est pas capable d'optimiser selon une fonction pseudo-booléenne qui n'est pas linéaire.

Dans cette section, nous présentons deux approches connues permettant de s'affranchir de cette limite. La première consiste en une réduction du problème d'optimisation non linéaire initial vers un problème d'optimisation, dans lequel de nouvelles contraintes permettent de « linéariser » la fonction objectif. La deuxième approche consiste, elle, en l'optimisation d'une fonction objectif linéaire minorant la fonction objectif initiale, et en l'énumération des solutions optimales jusqu'à l'obtention d'un cas d'arrêt signifiant qu'une des solutions optimales du problème initial a été extraite par le prouveur lors de l'énumération des solutions optimales de la fonction minorante.

6.2.1 Réduction vers des fonctions linéaires par ajout de contraintes

Il est possible de réduire un problème d'optimisation (sans contraintes) pour un nombre important de fonctions non linéaires à des problèmes d'optimisation de fonctions linéaires sous contraintes. L'exemple ci-dessous l'illustre dans le cas de la minimisation de la fonction $\max$.

Exemple 6.1 (Linéarisation de fonction objectif) *Soit le problème de minimiser la fonction f suivante, sans contraintes, tel que pour $i \in \{1, \ldots, n\}$, f_i est une fonction linéaire.*

$$\begin{array}{rl} \textit{minimiser} & f(\omega) = \max(f_1(\omega), \ldots, f_n(\omega)) \\ \textit{tel que} & \top \end{array}$$

Dire qu'un entier k est le maximum des f_i implique que l'ensemble des f_i est inférieur ou égal à k. De ce fait, on peut réduire notre problème initial sans contraintes au problème suivant, en introduisant une

variable entière k.

$$\begin{array}{rl} \textit{minimiser} & f(\omega) = k \\ \textit{tel que} & f_1(\omega) \leq k \\ & \dots \\ & f_n(\omega) \leq k \end{array}$$

Or, bien que le langage CNF n'offre pas de bonnes propriétés en ce qui concerne les requêtes que nous avons présentées dans le chapitre traitant de la compilation de formules propositionnelles, il permet néanmoins d'ajouter des clauses (et donc des contraintes pseudo-booléennes puisque des encodages polynomiaux en temps existent) en temps polynomial. Il est à noter que bien que les variables entières ne soient pas gérées nativement, il est possible de considérer leur représentation binaire, où une variable booléenne est associée à chacun des bits et est pondérée par la puissance de deux correspondante.

Exemple 6.2 (Codage d'une variable entière) *Soit le problème de minimisation de l'exemple précédent, où k est une variable entière.*

$$\begin{array}{rl} \textit{minimiser} & f(\omega) = k \\ \textit{tel que} & f_1(\omega) \leq k \\ & \dots \\ & f_n(\omega) \leq k \end{array}$$

En supposant que k soit bornée par 2^p, il est possible de l'exprimer grâce à p variables binaires b_i comme la somme $2^0 b_0 + \ldots + 2^{p-1} b_{p-1}$. De ce fait, on peut réduire le problème précédent de manière à obtenir un problème équivalent dont les variables sont toutes binaires.

$$\begin{array}{rl} \textit{minimiser} & f(\omega) = 2^0 b_0 + \ldots + 2^{p-1} b_{p-1} \\ \textit{tel que} & f_1(\omega) \leq 2^0 b_0 + \ldots + 2^{p-1} b_{p-1} \\ & \dots \\ & f_n(\omega) \leq 2^0 b_0 + \ldots + 2^{p-1} b_{p-1} \end{array}$$

Cette technique d'ajout de variables permet d'encoder en temps linéaire un grand nombre d'agrégations de fonctions linéaires. On peut par exemple citer [Bouveret & Lemaître 2009] dans lequel les auteurs introduisent des encodages pour la maximisation du leximin (ou, de manière symétrique, la minimisation du leximax). Dans ce travail, une contrainte *AtLeast* (au moins p contraintes parmi n sont satisfaites) est proposée, qui permet d'encoder le leximin en partant du raisonnement suivant :

- la valeur maximale k_n pour les n critères exprimés par des fonctions $f_1(\omega), \ldots, f_n(\omega)$ peut être minimisée en utilisant la contrainte *AtLeast* pour imposer que n contraintes parmi l'ensemble de n contraintes $\{(f_1(\omega) \leq k_n), \ldots, (f_n(\omega) \leq k_n)\}$ soient satisfaites ;
- la valeur maximale k_{n-1} pour $n-1$ parmi les n critères peut être minimisée en utilisant la contrainte *AtLeast* pour imposer que $n-1$ contraintes parmi l'ensemble de n contraintes $\{(f_1(\omega) \leq k_{n-1}), \ldots, (f_n(\omega) \leq k_{n-1})\}$ soient satisfaites ;
- ...
- la valeur maximale k_1 pour 1 parmi les n critères peut être minimisée en utilisant la contrainte *AtLeast* pour imposer que 1 contrainte parmi l'ensemble de n contraintes $\{(f_1(\omega) \leq k_1), \ldots, (f_n(\omega) \leq k_1)\}$ soit satisfaite.

Une fois ces contraintes définies, les auteurs optimisent successivement k_n, puis k_{n-1}, et ainsi de suite jusqu'à k_1 pour obtenir une solution minimale pour le leximax. On remarque qu'en ajoutant ces contraintes, les auteurs réussissent en fait à trier les valeurs des différentes fonctions objectif $f_i(\omega)$ grâce aux variables k_i. De ce fait, en choisissant une constante K suffisamment grande, les optimisations successives peuvent être traitées en une étape en considérant la minimisation de la fonction objectif

$\sum_{i=1}^{n} K^i k_i$. On se retrouve ici avec l'expression d'un opérateur OWA croissant représentant le leximax, où les K^i sont les poids (non normalisés, au sens où leur somme est supérieure à 1) et les k_i sont les valeurs des fonctions monocritère triées. De ce fait, bien que les auteurs ne l'aient pas noté, leur encodage est aussi valable pour tout opérateur de somme pondérée ordonnée à partir du moment où les poids sont croissants (respect du principe des transfert de Pigou-Dalton [DALTON 1920, LUSS 1999]).

En ce qui concerne les opérateurs OWA généraux, il n'existe pas à notre connaissance dans la littérature d'encodages polynomiaux dans le nombre de critères pour linéariser la fonction objectif. Cependant, si le nombre n de critères est faible, on peut entreprendre de résoudre les $n!$ problèmes d'optimisation correspondant aux $n!$ ordres possibles sur les valeurs des fonctions monocritère et de retourner la solution minimale parmi celles obtenues pour les $n!$ problèmes. Pour imposer un ordre sur deux critères linéaires, il suffit d'ajouter une contrainte $f_i(\omega) < f_j(\omega)$; en répétant cette opération $n-1$ fois, on impose ainsi un ordre sur l'ensemble des critères.

6.2.2 Optimisation par énumération des solutions optimales d'une fonction minorante

Dans le but de traiter de l'optimisation de fonctions non linéaires f, une approche introduite par [GONZALES *et al.* 2008] propose d'utiliser une fonction minorante linéaire $f_{\min}$ et d'énumérer les solutions du problème de contraintes de la moins coûteuse à la plus coûteuse pour $f_{\min}$, jusqu'à obtenir un cas d'arrêt. Cette approche a le mérite de ne pas nécessiter de linéarisation exponentielle en taille ou en espace dans le nombre de fonctions monocritère dans le cas où la fonction objectif est une agrégation de critères, et peut se révéler de ce fait plus efficace qu'une linéarisation lors de la considération de fonctions objectifs complexes, comme les intégrales de Choquet.

L'idée de cet algorithme est qu'en énumérant les solutions ω en partant de la moins coûteuse pour la fonction minorante $f_{\min}$, on va déterminer :

- des minorants pour les solutions optimales de la fonction f : les valeurs des $f_{\min}(\omega)$ (puisque $f_{\min}$ minore f) ;
- des majorants pour les solutions optimales de la fonction f : les valeurs des $f(\omega)$.

Le cas d'arrêt sera alors atteint lorsque le calcul d'un $f_{\min}(\omega)$ minimal (une fois retirées du problème les solutions déjà considérées) donne une valeur supérieure à un $f(\omega)$ calculé précédemment. En effet, lorsque la valeur pour la minorante atteint un seuil k, on sait que la valeur pour la fonction objectif ne descendra pas en dessous de k par la suite ; or, si parmi l'ensemble des solutions énumérées, l'une d'elle au moins donne une valeur inférieure ou égale à k à la fonction f, alors on sait qu'on ne trouvera pas de solution meilleure, et on pourra alors conclure que les solutions parmi celles que l'on a énumérées qui minimisent la fonction f sont des solutions optimales.

La faiblesse de cette approche provient du fait qu'il est difficile de déterminer une fonction minorante qui soit une bonne approximation de la fonction objectif, *a fortiori* dans le cas booléen. Dans notre contribution publiée dans les actes de la conférence RFIA [LE BERRE *et al.* 2012], nous avons cherché à obtenir une preuve d'optimalité en utilisant cette approche pour des problèmes de minimisation de norme de Tchebycheff. Cependant, sur tous les tests que nous avons effectués, nous avons observé que le comportement de la moyenne était celui d'une fonction par paliers : de nombreuses solutions du problème avaient une même valeur moyenne pour les critères. L'algorithme devait donc parcourir toutes les solutions pour ces paliers, sans se rapprocher de la fonction objectif. Nous supposons que c'est le fait de se situer dans un cadre booléen qui avait induit ce comportement (peu de valeurs distinctes pour la moyenne). Des pistes restent cependant à explorer quant à la détermination une fonction minorante linéaire plus adaptée.

Algorithme 6.2 : Optimisation par fonction minorante linéaire

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, une fonction objectif à minimiser f et une fonction $f_{\min}$ telle que $f_{\min} \leq f$

sortie : un modèle de ϕ minimal pour f, ou UNSAT si le problème n'admet pas de solution.

1 $\omega \leftarrow$ `modèleOptimal` (ϕ, f) ;
2 **si** $\omega =$ UNSAT **alors retourner** UNSAT;
3 $\omega_{\min} \leftarrow \omega$;
4 **tant que** $f_{\min}(\omega_{\min}) > f(\omega)$ **faire**
5 $\quad \phi \leftarrow \phi \cup \{\neg\omega\}$;
6 $\quad \omega \leftarrow$ `modèleOptimal` (ϕ, f) ;
7 $\quad$ **si** $f_{\min}(\omega) < f_{\min}(\omega_{\min})$ **alors** $\omega_{\min} \leftarrow \omega$;
8 **fin**
9 **retourner** $\omega_{\min}$;

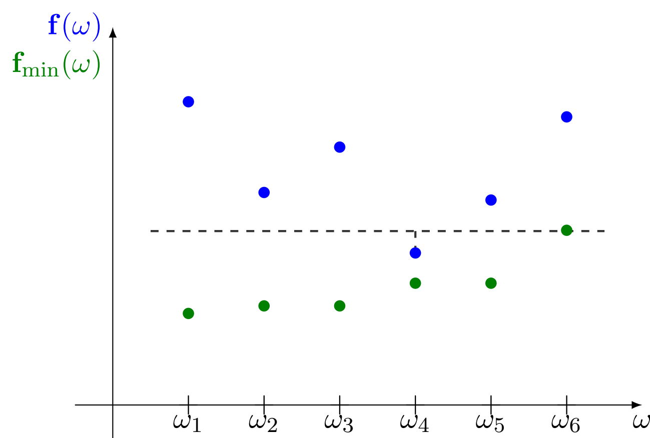


FIGURE 6.2 – Optimisation d'une fonction pseudo-booléenne non linéaire (en bleu) par énumération des solution optimales d'une fonction minorante linéaire (en vert).

6.3 Génération de contraintes paresseuses pour l'optimisation

Les résultats mitigés que nous avons obtenus lors de différents essais pour l'optimisation nous ont poussé à nous intéresser plus particulièrement à l'approche qui donnait les meilleurs résultats, à savoir l'optimisation par contrainte de borne présentée plus haut. Encore une fois, bien que ce type d'optimisation soit initialement réservé aux fonctions linéaires, il permet via des astuces de codage de considérer des fonctions bien plus générales, contenant par exemple l'ensemble des opérateurs de somme pondérée ordonnées convexes, et peut être de ce fait considéré lors de l'agrégation de critères. Nous présentons dans ce chapitre une de nos contributions publiée dans la revue RIA [Le Berre & Lonca 2014], dans laquelle nous tentons d'améliorer les performances des techniques SAT incrémentales, en fournissant une interface permettant à un logiciel externe d'obtenir une communication plus privilégiée avec le prouveur que lors de son utilisation comme une boîte noire.

Les prouveurs « spécialisés » pour des problèmes proches de SAT (MAXSAT, QBF, pseudo-booléen) développés il y a une dizaine d'année ont tendance à disparaître au profit de ceux qui réutilisent sans modification les prouveurs SAT (c'est particulièrement évident ces dernières années pour la résolution du problème MAXSAT [Fu & Malik 2006a, Morgado *et al.* 2012, Ansótegui *et al.* 2013b, Martins *et al.* 2014]). Ce qui nous intéresse ici est d'évaluer l'impact (positif ou négatif) d'une communication privilégiée avec le prouveur SAT par rapport à une approche boîte noire sur une plate-forme commune, et de définir précisément les conditions nécessaires pour permettre cette communication sans remettre en cause le fonctionnement du prouveur.

L'idée d'étudier le comportement des prouveurs SAT quand les clauses sont ajoutées dynamiquement durant la recherche n'est pas nouvelle : [Hooker 1993] présentait déjà une étude sur le sujet il y a plus de vingt ans. Notre étude est originale sur deux aspects. Tout d'abord, nous étudions le cas des prouveurs CDCL, qui, comme nous l'avons vu au chapitre précédent, sont conçus de manière bien différente des prouveurs SAT DPLL [Davis & Putnam 1960, Davis *et al.* 1962] étudiés par Hooker. De plus, nous nous intéressons ici à l'ajout de contraintes de cardinalité et de contraintes pseudo-booléennes, pas uniquement de clauses.

Nous avons intégré l'ajout de contraintes à la volée sur la plate-forme d'optimisation booléenne libre Sat4j [Le Berre & Parrain 2010], et intégré un algorithme d'optimisation par contrainte de borne paresseuse. Nous avons comparé l'approche boîte noire utilisée jusqu'à présent dans Sat4j et l'approche paresseuse sur l'ensemble des problèmes d'optimisation issus des compétitions de prouveurs pseudo-booléens (PB10) et MAXSAT (MAXSAT10). On n'observe pas de différence flagrante de performance (en terme de nombre d'instances résolues et de temps d'exécution) entre les deux approches, ce qui s'explique à la fois par les particularités des *benchmarks* de ces compétitions et par la spécificité du processus d'optimisation. On observe cependant des comportements particuliers, notamment en terme de nombre de solutions intermédiaires rencontrées lors du calcul de la valeur optimale de la fonction objectif sur certaines instances.

Nous présentons tout d'abord l'observation qui a motivé cette étude : l'amélioration significative des performances de l'énumération de solutions basée sur la production de clauses paresseuses. Nous détaillons ensuite les conditions nécessaires pour l'apprentissage de contraintes dynamiques, puis nous présentons ensuite les résultats expérimentaux et concluons par quelques perspectives.

6.3.1 Motivation

Nous utilisons les prouveurs Abscon [Merchez *et al.* 2001] et Sat4j [Le Berre & Parrain 2010] pour illustrer les forces et les faiblesses des approches SAT et CSP pour la résolution de divers problèmes académiques à nos étudiants de master en informatique de l'Université d'Artois. Sat4j-CSP traduit les problèmes CSP en SAT par l'encodage support pour les contraintes binaires [Gent 2002] et direct

[WALSH 2000] pour les contraintes n-aires. Afin de fournir des fonctionnalités similaires à Abscon, il était nécessaire d'ajouter l'énumération de solutions à Sat4j-CSP. Sat4j permet d'énumérer les solutions d'une CNF en « bloquant » chaque solution trouvée à l'aide d'une clause. Cette approche n'est généralement pas envisageable sur des instances issues de problèmes réels, car il faut ajouter autant de clauses que de modèles, et chaque clause contient l'ensemble des variables du problème (voir [MORGADO & SILVA 2005] par exemple pour plus de détails) [7]. Cependant, cette approche fonctionne relativement bien sur les problèmes jouets que nous utilisons en enseignement. L'architecture de Sat4j a évolué récemment pour lui permettre à terme d'intégrer un prouveur SMT. Parmi les évolutions, on retrouve la possibilité d'ajouter une clause conflictuelle à certains moments clés de la recherche (lorsqu'une solution a été trouvée par exemple). Afin de tester cette fonctionnalité, nous avons décidé de réaliser un énumérateur de solutions basé sur cette production de clauses à la volée. Cette nouvelle approche permet d'énumérer les 14200 solutions d'un problème de 12 reines en 113 secondes sur un MacBook 2009 contre 593 secondes pour l'ancienne approche.

Nous avons évalué les deux approches sur les instances SAT utilisées dans [MORGADO & SILVA 2005] : CBS, UF et FLAT. La principale difficulté était de trouver des instances cohérentes avec un nombre raisonnable de solutions (de quelques centaines à quelques centaines de milliers). CBS (*Controlled Backbone Size*) correspond à des instances dont la taille du *backbone* (littéraux identiques dans tous les modèles) est fixe pour des instances 3-SAT à 100 variables booléennes [SINGER *et al.* 2000] (nous avons utilisé les plus grandes instances contenant 449 clauses). UF (*Uniform Formulas*) correspond aux instances cohérentes 3-SAT aléatoires de SATLIB (de 20 à 250 variables booléennes). FLAT correspond à des problèmes de coloriage de graphes 3-coloriables (de 20 à 600 variables booléennes). Le tableau 6.1 résume les résultats de cette expérimentation, effectuée sur des machines à base de processeurs Intel Quad-core XEON X5550 2,66 GHz équipées de 32 Go de mémoire avec un temps limite de deux minutes. Pour chaque approche, on comptabilise comme résolues les instances pour lesquelles le prouveur a pu énumérer toutes les solutions de la CNF dans le temps imparti.

Dans les trois cas, l'approche par ajout de clauses paresseuses permet d'obtenir de bien meilleurs résultats que l'approche par clauses bloquantes. Si on considère les temps de calcul sur les instances résolues par les deux approches (voir le tableau 6.2), on se rend compte qu'en moyenne l'approche « à la volée » énumère les solutions environ dix fois plus rapidement. Les médianes relativement proches nous apprennent que plus les instances sont difficiles, plus le gain est important, ce qui semble naturel. Par ailleurs, nous avons lancé les mêmes expérimentations, avec un temps limite de vingt minutes : bien que les deux prouveurs profitent de l'augmentation du temps qui leur est imparti (voir le tableau 6.3), la différence d'efficacité entre les deux approches est encore plus flagrante, comme le montre le tableau 6.4 ; ce qui confirme que plus le nombre de modèles est important, plus le gain de l'approche « à la volée » est important dans ce contexte.

Pour conclure, lors de nos expérimentations, l'approche « à la volée » a été capable d'énumérer jusqu'à 735 847 solutions en deux minutes, quand l'autre approche n'a pas pu dépasser 67 637 solutions.

Ces résultats encourageants nous ont conduit à généraliser ce principe au cadre des contraintes de cardinalité et des contraintes pseudo-booléennes, dans le but d'améliorer les performances des prouveurs pseudo-booléens et MAXSAT de Sat4j. Nous présentons tout d'abord l'approche dite « boîte noire » utilisée par de nombreux prouveurs (dont Sat4j) pour résoudre des problèmes d'optimisation en variables booléennes. Nous présentons ensuite l'approche « boîte grise », qui permet une plus forte communication avec le prouveur SAT, pour permettre la génération de contraintes à la volée.

7. Il est possible de bloquer une solution en n'utilisant que les littéraux décision comme indiqué dans [GEBSER *et al.* 2007], ce qui permet de réduire la taille des clauses. Quand cette approche est utilisée avec un prouveur apprenant des clauses, il faut faire attention à ne pas éliminer les littéraux propagés par les clauses apprises.

catégorie	#inst.	boîte noire	à la volée
CBS	5000	3622	**4724**
UF	3700	2991	**3300**
FLAT	1700	1058	**1232**

TABLE 6.1 – Comparaison boîte noire *vs* ajout de clauses à la volée pour l'énumération de solutions en nombre d'instances dont les solutions ont toutes été énumérées — temps limite de 2 minutes

catégorie	boîte noire	à la volée
CBS	10 373 (770)	**1029** (**647**)
UF	3653 (**94**)	**423** (103)
FLAT	13 168 (2009)	**1319** (**1010**)

TABLE 6.2 – Comparaison des temps de calcul moyens (et médians) par séries, en ms, pour l'énumération de solutions — temps limite de 2 minutes, cas des instances résolues par les deux prouveurs

catégorie	#inst.	boîte noire	à la volée
CBS	5000	4076	**4881**
UF	3700	3110	**3408**
FLAT	1700	1151	**1675**

TABLE 6.3 – Comparaison boîte noire *vs* ajout de clauses à la volée pour l'énumération de solutions en nombre d'instances dont les solutions ont toutes été énumérées — temps limite de 20 minutes

catégorie	boîte noire	à la volée
CBS	53 239 (1045)	**1584** (**694**)
UF	18 264 (**101**)	**667** (121)
FLAT	39 730 (2447)	**1672** (**1013**)

TABLE 6.4 – Comparaison des temps de calcul moyens (et médians) par séries, en ms, pour l'énumération de solutions — temps limite de 20 minutes, cas des instances résolues par les deux prouveurs

6.3.2 Optimisation par contraintes de borne dynamiques

Afin de mieux comprendre les enjeux et difficultés de l'ajout de contraintes à la volée dans un prouveur CDCL, le lecteur est invité à retourner au chapitre précédent, pour se remémorer en particulier les sections traitant de l'apprentissage de contraintes (section 5.2.4, page 95), mais aussi les sections traitant de la propagation de contraintes, en particulier en ce qui concerne les contraintes pseudo-booléennes (section 5.4.3, page 108).

Ajouter des contraintes à la volée : l'approche boîte grise

Lorsque l'on souhaite ajouter des contraintes de borne à la volée dans le prouveur, on se trouve dans une situation différente de celle de l'apprentissage de clauses : la contrainte ajoutée est falsifiée [8], mais la décision ayant conduit à cette falsification n'est pas nécessairement la dernière à avoir été prise [9]. Il est donc nécessaire de dépiler toutes les propagations et décisions jusqu'au premier niveau où la contrainte est falsifiée de manière à ce que l'état du prouveur soit cohérent avec l'approche CDCL : un conflit doit être traité dès qu'il est détecté. L'algorithme 6.3 présente un prouveur de type CDCL capable d'opérer avec différents types de contraintes, avec ajout de contraintes à la volée lorsqu'une solution est trouvée. On note ligne 11 une fonction `défaireJusqueFalsif` qui a pour but de dépiler les décisions jusqu'au niveau de celle provoquant la falsification de la contrainte. Cette fonction agit de manière spécifique pour chaque type de contrainte :

clause une clause est falsifiée si et seulement si tous ses littéraux sont falsifiés, il suffit donc de dépiler toutes les décisions jusqu'au dernier niveau de décision de la clause. On se retrouve ici dans un cas similaire à un retour arrière lors de l'apprentissage d'une clause après analyse de conflit.

cardinalité une contrainte de cardinalité de n littéraux et de degré k est falsifiée si et seulement si $n - k + 1$ littéraux sont falsifiés. Il faut donc détecter les $n - k + 1$ premières falsifications de littéraux.

pseudo-booléenne une contrainte pseudo-booléenne de n littéraux et de degré k est falsifiée si et seulement si $\sum_{i=1}^{n} w_i \times l_i \equiv \sum w_i - \sum_{l_j=0} w_j < k$. Il faut donc détecter le premier niveau de décision pour lequel la somme des poids des coefficients falsifiés dépasse $\sum w_i - k$.

On peut remarquer que dans le cas des contraintes de cardinalité et des contraintes pseudo-booléennes, l'ordre dans lequel les littéraux sont falsifiés est important. Ce n'est pas le cas pour les clauses puisqu'il n'y a qu'une seule façon de falsifier une clause : falsifier tous ses littéraux.

L'ajout de contraintes falsifiées à chaque modèle trouvé implique que l'algorithme termine toujours par un épuisement des décisions, qui correspond à un problème incohérent dans le cadre classique. Dans notre cas, l'apparition d'un conflit sans prise de décision correspond soit à un problème incohérent soit à la preuve d'optimalité de la dernière solution trouvée. De plus, si une solution a été trouvée avant le temps limite, elle fournit un majorant de la valeur optimale.

Notons enfin que les prouveurs CDCL modernes effacent les clauses apprises durant la recherche. Il est donc nécessaire que les contraintes de borne soient gérées spécifiquement, pour éviter qu'elles ne soient effacées. On notera de plus qu'il n'est pas nécessaire de garder plus d'une contrainte de borne, car ces contraintes sont de plus en plus fortes (c'est aussi le cas dans l'approche par boîte noire [LE BERRE & PARRAIN 2010]).

Si maintenir l'invariant que chaque conflit est analysé dès qu'il est détecté est obligatoire pour garantir le bon fonctionnement d'un prouveur CDCL lors de l'ajout de contraintes à la volée, nous allons aussi

8. Dans le cas contraire, se pose le problème de l'arrêt de l'algorithme, qui pourrait générer indéfiniment la même solution.

9. Ce qui était le cas dans notre étude préliminaire d'énumération de modèles, car la clause contenait toutes les variables. C'est aussi le cas si l'on se restreint aux seuls littéraux décision.

Algorithme 6.3 : Optimisation par ajout de contrainte de borne dynamique

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, et une fonction d'objectif à minimiser f

sortie : un modèle de ϕ minimisant f, ou UNSAT si le problème est insatisfiable.

```
1  satisfiable ← FALSE;
2  dynConflit ← NIL;
3  conflit ← propager ();
4  si conflit = NIL alors
5  |  si décider() = NIL alors
6  |  |  dynConflit ← genContrainteBorne() ;
7  |  |  si dynConflit = NIL alors
8  |  |  |  retourner ⟨SAT,certificat⟩ ;
9  |  |  fin
10 |  |  défaireJusqueFalsif (dynConflit);
11 |  |  satisfiable ← TRUE;
12 |  |  conflit ← dynConflit ;
13 |  fin
14 fin
15 si conflit ≠ NIL alors
16 |  si pileDecisionsVide() alors
17 |  |  si satisfiable alors
18 |  |  |  retourner ⟨OPTIMALE,certificat⟩ ;
19 |  |  sinon
20 |  |  |  retourner ⟨UNSAT,∅⟩ ;
21 |  |  fin
22 |  fin
23 |  clause ← analyser (conflit);
24 |  défaireDécisionPourPropager (clause);
25 |  si conflit = dynConflit alors
26 |  |  propager (dynConflit);
27 |  |  dynConflit ← NIL;
28 |  fin
29 |  apprendre (clause);
30 fin
```

voir qu'en fonction du type de contrainte ajoutée certaines vérifications doivent être faites afin de garantir l'efficacité du prouveur.

Propager les contraintes ajoutées à la volée

Il est possible que la contrainte ajoutée puisse propager des valeurs de vérité lorsque l'on défait des décisions après l'analyse de conflits. Si la contrainte est une contrainte de cardinalité ou une contrainte pseudo-booléenne, il est possible que cette contrainte propage plusieurs littéraux voire des littéraux à des niveaux de décision différents, comme nous l'avons décrit à la section 5.4.3 (page 108) ; ce point étant primordial pour la génération de nos contraintes à la volée, nous le détaillons à nouveau ici.

Considérons par exemple la contrainte $l_1 + l_2 + ... + l_n \leq 0$ produite comme contrainte de borne pour un modèle qui satisfait un unique littéral (l_j) de la fonction objectif. Après ajout de cette contrainte, tous les littéraux qu'elle contient doivent être propagés à faux au niveau de décision 0, puisque qu'un modèle doit dorénavant contenir $\neg l_1$, $\neg l_2$, ..., et $\neg l_n$. En revanche, si on considère uniquement la clause issue de l'analyse de conflits, nous ne pouvons propager que le littéral $\neg l_j$, et non les littéraux $\neg l_i$ tels que $i \neq j$ ($i \in \{1, 2, ..., n\}$). Il est donc nécessaire de permettre à la nouvelle contrainte de propager des valeurs de vérité. C'est le but de la fonction `propager` ligne 27 dans l'algorithme 6.3.

Il existe de plus un problème spécifique aux contraintes pseudo-booléennes : celles-ci ont la particularité de pouvoir propager des valeurs à divers niveaux de décision, alors qu'une clause ne propage qu'une valeur à un unique niveau de décision, et qu'une contrainte de cardinalité peut propager plusieurs valeurs de vérité là aussi à un unique niveau de décision. L'ajout de contraintes pseudo-booléennes implique donc de remonter au premier niveau de propagation, qui peut être le niveau de propagation 0 si le poids d'un littéral est nécessaire à la satisfaction de la contrainte : $5x_1 + 3x_2 + x_3 \leq 3$ par exemple implique $\neg x_1$ au niveau de décision 0. Cette contrainte peut être falsifiée soit en satisfaisant x_1, soit en satisfaisant x_2 et x_3. Dans le premier cas, la propagation se fera effectivement au niveau de décision 0. Dans le second, $\neg x_1$ ne sera pas propagé.

Cet exemple met l'accent sur la difficulté de gérer des contraintes pseudo-booléennes à la volée. Le fait qu'une contrainte puisse être falsifiée de plusieurs manières induit une difficulté dans l'analyse de ces contraintes. Nous n'avons cependant pas trouvé de solution élégante pour régler ce problème à l'heure actuelle.

Dans le cas où la contrainte est une simple clause, une seule valeur de vérité peut être propagée (le seul littéral non falsifié), et cette valeur sera propagée par la clause issue de l'analyse de conflits. En effet, si une clause apprise propage une valeur de vérité quand les décisions sont dépilées, c'est qu'il s'agit d'une clause contenant un seul littéral au niveau de décision courant (*First UIP*). Donc la procédure d'analyse de conflits va retourner exactement cette clause, ou une clause simplifiée [SÖRENSSON & BIERE 2009] dont le littéral propagé sera exactement celui qui serait propagé par la clause ajoutée à la volée. Il n'y a donc pas de propagation supplémentaire possible lors de la génération de clauses conflictuelles à la volée.

Pour résumer, il est important de noter ici que l'on doit différencier le cas des clauses qui ne peuvent propager qu'un littéral, qui sera de toute façon propagé lors de l'analyse de conflits, des contraintes de cardinalité ou des contraintes pseudo-booléennes qui peuvent propager plusieurs littéraux, qui ont donc un pouvoir de propagation plus important que les clauses habituellement générées par analyse de conflit.

6.3.3 Expérimentations et résultats

Nous avons intégré notre algorithme d'optimisation par contrainte de borne dynamique à la plateforme Sat4j, dans les prouveurs d'optimisation pseudo-booléenne et MAXSAT. Il est à noter qu'aucun de ces prouveurs n'effectue de prétraitement. Nos expérimentations ont été réalisées sur des machines

catégorie	#inst.	b. noire	b. grise
BIGINT-LIN	532	**125** (57)	115 (57)
SMALLINT-LIN	699	**270** (33)	266 (33)
SMALLINT-NLC	409	273 (0)	**275** (0)

TABLE 6.5 – Boîte noire *vs* à la volée - instances PB10 résolues (et UNSAT)

équipées de processeurs Intel XEON 3,0 GHz avec 2 Go de mémoire, moins puissantes que celles utilisées en section 6.3.1 mais qui correspondent à l'environnement d'exécution de la compétition pseudo-booléenne PB10. En plus des résultats de performance globale, nous souhaitions vérifier les hypothèses concernant les caractéristiques d'une approche par contraintes de borne dynamique par rapport à une approche boîte noire :

– Le temps passé entre la dernière solution trouvée et la preuve d'incohérence finale devrait être plus réduit avec une approche dynamique, puisque l'on bénéficie de l'analyse de conflits
– La distance entre deux solutions est potentiellement plus réduite dans le cadre d'une approche dynamique, ce qui pourrait dans certains cas demander d'énumérer énormément de solutions proches avant de trouver une solution optimale.

Nous présentons maintenant les résultats de nos expérimentations sur un ensemble de *benchmarks* PB et MAXSAT.

Optimisation pseudo-booléenne

Nous avons utilisé des benchmarks issus de la compétition PB10 [10] afin de comparer les résultats de nos deux approches, en laissant un temps limite de trente minutes à nos prouveurs pour trouver une solution optimale (soit le même temps limite que celui de la compétition). Le tableau 6.5 présente le nombre d'instances résolues par nos deux algorithmes (le nombre d'instances prouvées incohérentes est indiqué entre parenthèses).

Concernant ces *benchmarks*, on remarque un léger avantage à l'utilisation de l'algorithme utilisant le prouveur SAT en tant que boîte noire. Cependant, ces chiffres sont à nuancer légèrement car ces résultats globaux cachent le fait que chaque approche est capable de résoudre des problèmes que l'autre approche ne peut pas résoudre : 12 instances sont uniquement résolues par l'approche externe, et 6 instances sont uniquement résolues par l'approche interne. Le cas pathologique de l'approche à base de contrainte de borne dynamique est d'énumérer un nombre de solutions beaucoup plus important que l'approche boîte noire, ce qui fait perdre du temps, mais aussi beaucoup de mémoire. C'est par exemple le cas pour l'instance `normalized-mps-v2- 20-10-glass4.opb`, pour laquelle le prouveur « boîte noire » trouve la solution optimale au bout de 50 secondes en ayant énuméré 16 solutions intermédiaires, alors que le prouveur utilisant l'autre approche « meurt » car il demande trop de mémoire, après avoir calculé 8715 solutions intermédiaires. Ce cas pathologique n'est présent que rarement dans nos expérimentations, bien que cette dernière approche ait tendance à énumérer plus de solutions avant de trouver la valeur optimale, comme le montre la figure 6.3. On voit par exemple sur cette figure qu'il est possible de résoudre, en énumérant au maximum 50 solutions, environ 620 *benchmarks* avec l'approche « boîte grise », alors qu'on peut en résoudre dans les mêmes conditions environ 645 avec l'approche « boîte noire ». En termes quantitatifs, l'algorithme d'optimisation classique énumère en moyenne 5 solutions quand l'algorithme utilisant des contraintes de borne dynamiques énumère en moyenne 17 solutions. Le nombre médian de solutions énumérées est de 2 dans les deux cas, ce qui montre que la différence entre les nombres de solutions intermédiaires parcourues par nos algorithmes est d'autant plus importante que

10. `http://www.cril.univ-artois.fr/PB10/`

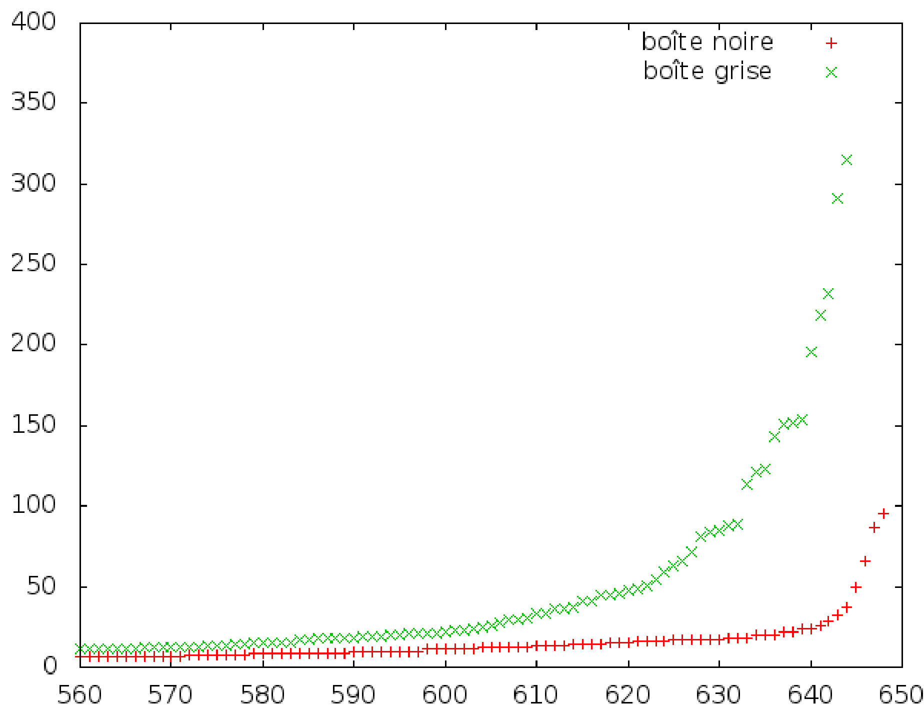


FIGURE 6.3 – Distribution des solutions énumérées pour PB (graphe partiel) — un point (x, y) signifie qu'on peut résoudre x instances en énumérant pour chacune d'elles un maximum de y solutions intermédiaires

le nombre de solutions énumérées est important.

La figure 6.4 représente la distribution des temps des prouveurs sous forme de « cactus », c'est-à-dire en triant les temps par ordre croissant. Les prouveurs peuvent avoir des temps différents sur des instances individuelles, mais globalement, la distribution des temps est quasi-identique dans les deux cas. La figure 6.5 représente de la même manière la distribution du temps nécessaire à la preuve de l'optimalité, c'est-à-dire à la résolution du dernier problème de décision, correspondant à une formule incohérente. Là encore, si les temps individuels par instance varient, les distributions de temps ne sont pas dissociables.

Optimisation MAXSAT

Nous avons comparé les deux approches dans le cadre de l'optimisation MAXSAT sur les instances de la compétition MAXSAT10, avec un temps limite de vingt minutes (soit le même que lors de la compétition). Les résultats sont présentés dans le tableau 6.6. Ici encore, les résultats sont très proches, avec un très léger avantage pour l'approche « boîte noire ». Encore une fois, cette légère différence est en partie imputable à une énumération d'un trop grand nombre de solutions concernant l'autre algorithme pour certaines instances. Dans ces jeux de test, 31 instances sont résolues par l'approche externe et non par l'interne, et 8 autres sont dans le cas inverse. De la même manière que l'algorithme d'optimisation par contrainte de borne dynamique énumère de manière générale plus de solutions dans le cadre de l'optimisation pseudo-booléenne, on observe un comportement similaire pour MAXSAT (voir la figure 6.6). En termes de statistiques, l'approche « boîte noire » énumère en moyenne 7 solutions quand l'approche « boîte grise » en énumère 13. Encore une fois, une étude du nombre médian de modèles découverts lors

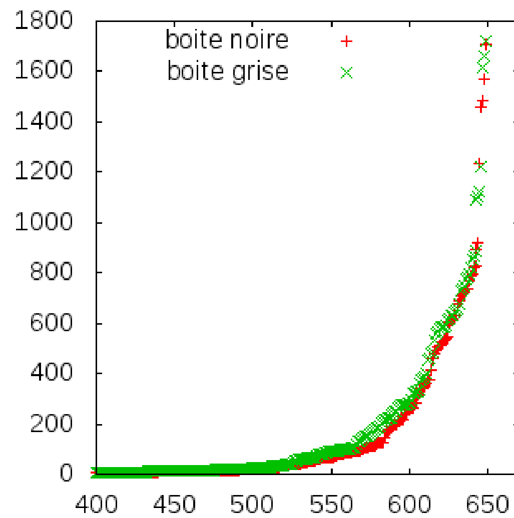


FIGURE 6.4 – Distribution des temps de résolution des instances PB (graphe partiel) — un point (x, y) signifie qu'on peut résoudre x instances en fixant un temps de résolution limite de y secondes.

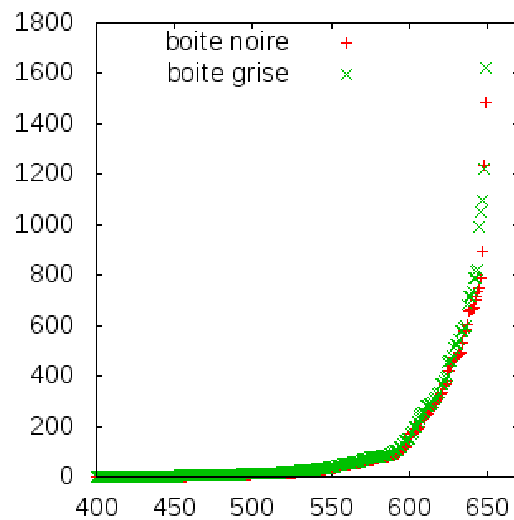


FIGURE 6.5 – Temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée (PB) — un point (x, y) signifie qu'on peut prouver l'optimalité pour x instances en fixant un temps de résolution limite de y secondes.

catégorie	#inst.	b. noire	b. grise
ms_crafted	167	**2**	**2**
ms_industrial	77	**8**	5
ms_random	300	**0**	**0**
pms_crafted	385	**190**	187
pms_industrial	497	**270**	258
pms_random	240	**26**	**26**
wms_crafted	149	**43**	**43**
wms_random	200	**16**	**16**
wpms_crafted	378	**146**	142
wpms_industrial	132	**36**	35
wpms_random	150	**29**	**29**

TABLE 6.6 – Boîte noire *vs* à la volée - instances MAXSAT10 résolues

de la recherche (4 pour la première, 5 pour la deuxième) nous montre que plus le nombre de solutions énumérées est important, plus la différence entre les deux approches sur ce point l'est aussi. Une fois encore, comme le montrent les figures 6.7 et 6.8, on ne peut départager de manière globale nos deux approches ni sur la distribution des temps de calcul des solutions optimales, ni sur la distribution des temps de la preuve d'optimalité.

6.3.4 Analyse des résultats

À la lumière de ces expérimentations, l'approche par apprentissage de bornes à la volée ne se révèle pas plus efficace que l'approche incrémentale ou boîte noire. Ces résultats sont décevants compte tenu des résultats initiaux obtenus sur l'énumération de solutions. Il nous semble donc important de mettre en avant quelques faits qui peuvent expliquer ces résultats. Tout d'abord, dans nos deux problèmes d'optimisation, les prouveurs trouvent une première solution très proche de l'optimale dans plus de la moitié des cas, ce qui réduit *de facto* le nombre maximal de solutions qui seront énumérées (nombre de solutions intermédiaires médian des séries de test variant de 2 à 5 selon les problèmes et les approches). Il faut mettre cela en balance avec les milliers de solutions générées dans le cadre de l'énumération. Les cas (peu nombreux) pour lesquels seul le prouveur « boîte noire » est capable de trouver une solution optimale sont souvent liés à une grande différence en terme de nombre de solutions/contraintes générées. En énumération de solutions, les deux approches doivent générer exactement le même nombre de solutions. Enfin, nous avons essayé différentes heuristiques, stratégies de redémarrages, stratégies de minimisation de clauses : les résultats individuels des prouveurs changent légèrement, mais les tendances restent identiques.

6.3.5 Conclusions à propos des approches proposées

Nous avons étudié l'impact d'une communication privilégiée avec un prouveur SAT générique permettant l'ajout de contraintes à la volée dans le cadre de la résolution de problèmes d'optimisation en variables booléennes. Nous avons montré que si l'ajout dynamique de clauses se fait sans difficulté, l'ajout de contraintes de cardinalité ou de contraintes pseudo-booléennes nécessite quelques précautions. Plus spécifiquement, ces contraintes peuvent propager plusieurs littéraux quand elles sont intégrées au prouveur, alors qu'une clause propage exactement un littéral, qui sera dans tous les cas propagé lors de l'analyse de conflit.

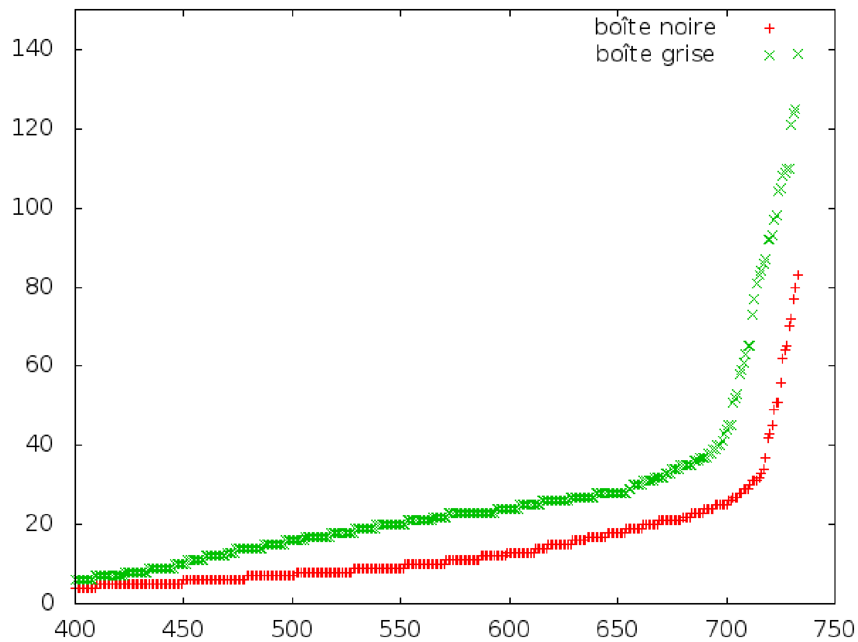


FIGURE 6.6 – Distribution des solutions énumérées pour MAXSAT (graphe partiel) — un point (x, y) signifie qu'on peut résoudre x instances en énumérant pour chacune d'elles un maximum de y solutions intermédiaires

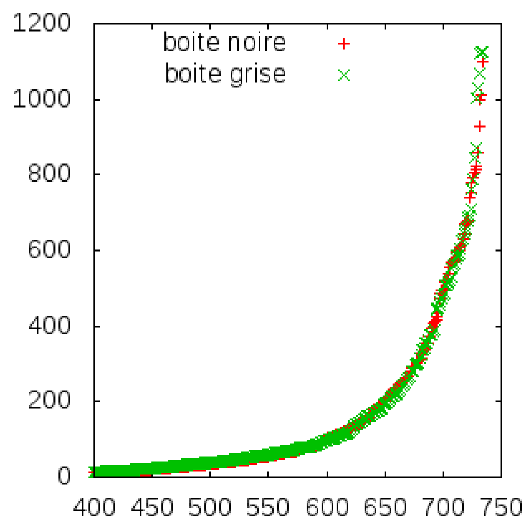


FIGURE 6.7 – Distribution des temps de résolution des instances MAXSAT (graphe partiel) — un point (x, y) signifie qu'on peut résoudre x instances en fixant un temps de résolution limite de y secondes.

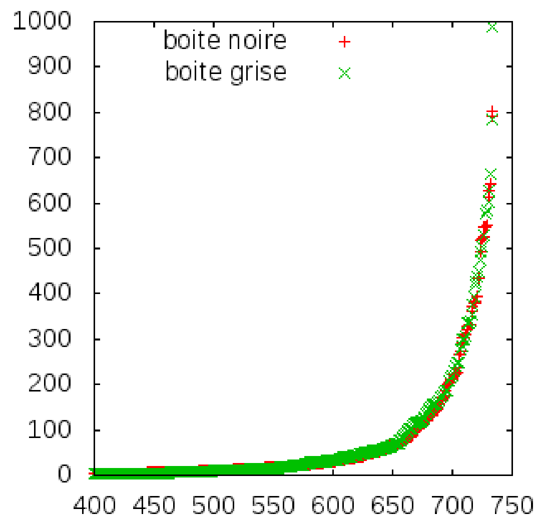


FIGURE 6.8 – Temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée (MAXSAT) — un point (x, y) signifie qu'on peut prouver l'optimalité pour x instances en fixant un temps de résolution limite de y secondes.

Les résultats expérimentaux sont assez décevants : on ne remarque pas de différence flagrante d'efficacité, comme on avait pu le constater sur le cas de l'énumération de solutions qui avait motivé cette étude. Ces résultats peuvent s'expliquer par la nature des *benchmarks* ou du prouveur utilisés : nombre de *benchmarks* sont résolus après production de quelques solutions intermédiaires. Cela signifie d'une part que le prouveur résout ces problèmes en trouvant rapidement une solution proche de l'optimum mais aussi qu'il a tendance à ne pas résoudre les problèmes pour lesquels il faut énumérer un nombre important de solutions intermédiaires. Il nous est cependant difficile de réaliser ces expérimentations sur une autre plateforme car l'approche par ajout de borne dynamique demande une connaissance avancée du prouveur.

Il faut aussi noter que dans le cadre de l'optimisation, un changement dans l'exploration de l'espace de recherche peut avoir un impact important sur le nombre de solutions intermédiaires à explorer avant de trouver la solution optimale. Nos expérimentations montrent dans le cas de MAXSAT que l'approche par ajout de borne dynamique à tendance à énumérer plus de solutions intermédiaires que l'approche boîte noire. Il n'est cependant pas clair à l'heure actuelle si cela est dû à l'approche d'ajout de contraintes à la volée, qui force le prouveur à continuer de chercher des solutions dans l'espace de recherche courant ou s'il s'agit d'illustrations de la faiblesse de la propagation des contraintes pseudo-booléennes mentionnée dans la section précédente.

Nous devons aussi prendre en compte l'évolution des prouveurs SAT ces dernières années. Ils utilisent des stratégies de redémarrage rapides, voire très rapides, car le coût d'un redémarrage est devenu très faible [VAN DER TAK *et al.* 2011]. Le modèle théorique derrière ces algorithmes de type CDCL [PIPATSRISAWAT & DARWICHE 2011, ATSERIAS *et al.* 2011] fonctionne sur l'hypothèse d'un redémarrage après apprentissage de chaque clause dérivée par analyse de conflits : certains prouveurs s'en rapprochent quand ils utilisent une suite de Luby [LUBY *et al.* 1993] (sans facteur). En ajoutant à cela que ces prouveurs bénéficient d'une interface incrémentale, il semble que le gain d'une analyse de conflit sur les contraintes ajoutées à la volée soit quasi nul, car elle demande une attention particulière pour les contraintes de cardinalité et surtout les contraintes pseudo-booléennes.

Nous avons identifié une faiblesse dans notre approche lors de l'apprentissage de contraintes pseudo-

booléennes. Ces contraintes pouvant propager des littéraux à divers niveaux de décision, il est possible que la contrainte apprise à la volée doive propager dès le niveau de décision 0 des littéraux, donc indépendemment de l'affectation partielle courante des variables. Cela nous pousse à penser qu'une solution intermédiaire consisterait à détecter ce genre de contraintes et à redémarrer le prouveur au lieu d'effectuer une analyse de conflits. Il s'agirait d'une solution qui combine en quelque sorte les deux approches présentées dans ce travail.

6.4 De la correction des prouveurs

Dans ce chapitre, nous avons présenté des outils capables de résoudre automatiquement des problèmes de décision, et nous avons proposé des approches dans le but d'améliorer ces logiciels. Or, bien que ces logiciels soient écrits à partir d'algorithmes considérés comme corrects, la correction des logiciels eux-mêmes est bien plus difficile à assurer, vu le nombre de lignes de code qui les composent, les différences pratiques qui existent entre la description d'un algorithme et sa réalisation concrète (la réalisation d'un logiciel n'est pas une simple traduction du pseudocode d'un algorithme), ou bien encore les effets de bord créés par l'utilisation de bibliothèques dont seul le rédacteur peut avoir connaissance dans le cas où la documentation serait inaccessible ou incomplète.

De ce fait, la nécessité de procédures de test dans le but de détecter un ou plusieurs bogues est avérée. Or, les méthodes manuelles de test de logiciel à la recherche de bogues sont des processus longs, sujets à erreurs. De plus, ces méthodes sont aussi dans beaucoup de cas impossibles à employer de manière la plus exhaustive possible, dans la mesure où les problèmes nécessitant un logiciel de résolution sont bien souvent des problèmes dont la nature des données d'entrée implique une combinatoire pour l'algorithme de résolution sous-jacent. La conjonction de ces difficultés est connue dans la littérature comme le *problème de l'oracle* [WEYUKER 1982], c'est-à-dire l'impossibilité de déterminer efficacement si la sortie d'un programme est correcte ; en d'autres termes, il est difficile de vérifier que la sortie du programme est conforme à ce qu'aurait renvoyé l'algorithme pour l'entrée fournie. D'un autre côté, les procédures de *test redondant* s'appuient sur des logiciels équivalents et vérifient pour un ensemble de *benchmarks* si le logiciel en test renvoie les mêmes résultats que les logiciels équivalents. Bien que réalisable dans certains cas, cette approche requiert cependant de disposer de logiciels équivalents, en nombre le plus important possible, et que ces logiciels soient eux-mêmes corrects.

Les *tests métamorphiques* [CHEN *et al.* 1998] ont été proposés dans le but de pallier le problème de l'oracle. L'idée derrière cette technique est de générer des cas de test à partir de tests existants, en utilisant pour cela des règles, appelées *règles métamorphiques*. Ces règles permettent, pour un cas de test et la réponse attendue à ce test, de générer à la fois un nouveau test ainsi que la réponse attendue pour ce nouveau test, ce qui rend l'oracle inutile. En plus de régler cette problématique, la génération de tests métamorphiques peut être hautement automatisée pour générer un grand nombre de cas de test différents à partir du moment où l'on dispose d'un ensemble de règles suffisamment important. Ces procédures de test ont d'ailleurs montré leur efficacité dans plusieurs domaines, tels que les programmes numériques [CHEN *et al.* 2002], la théorie des graphes [CHEN *et al.* 2004], ou bien encore les applications orientées service [CHAN *et al.* 2007].

En marge de l'étude des problématiques d'optimisation auxquels cette thèse est dévolue, nous nous sommes intéressés à la problématique des tests métamorphiques pour un certain nombre de problèmes, incluant le problème de la vérification de la cohérence d'une formule CNF par un prouveur SAT, mais aussi des problèmes de gestion de dépendances logicielles pour les prouveurs acceptant le formalisme CUDF (présenté brièvement dans le premier chapitre). Ces travaux ont donné lieu à publication dans la revue d'audience internationale *Software Testing, Verification and Reliability* [SEGURA *et al.* 2015]. Ces travaux font suite aux travaux initiés par d'autres chercheurs de la communauté SAT sur le test de

prouveurs [BRUMMAYER *et al.* 2010, ARTHO *et al.* 2013].

Contrairement aux approches précédentes basées sur la génération automatique de milliers de *benchmarks* aléatoires ou structurés qui permettaient de tester la réponse (SAT ou UNSAT) d'un prouveur, l'approche à base de règles métamorphique est plus complète : elle permet de connaître l'ensemble des modèles des formules générées, et ainsi de vérifier qu'un prouveur est capable de déterminer l'ensemble des modèles, plutôt que de simplement vérifier s'il est capable d'en détecter un, le cas échéant.

Dans notre contribution, nous avons défini cinq règles métamorphiques :

- disjonction avec une nouvelle variable, ce qui correspond à l'ajout, dans une clause du problème initial, du littéral positif d'une variable n'existant pas dans le problème initial ;
- disjonction avec une nouvelle variable niée, ce qui correspond à l'ajout, dans une clause du problème initial, du littéral négatif d'une variable n'existant pas dans le problème initial ;
- disjonction avec une variable existante, ce qui correspond à l'ajout, dans une clause du problème initial, du littéral positif d'une variable apparaissant dans le problème initial ;
- disjonction avec une variable existante, ce qui correspond à l'ajout, dans une clause du problème initial, du littéral négatif d'une variable apparaissant dans le problème initial ;
- conjonction avec une nouvelle clause.

Ainsi, l'idée est de partir d'une formule dont on connaît l'ensemble des modèles, par exemple une tautologie, et d'appliquer un certain nombre de règles de manière successive, en calculant à chaque fois le nouvel ensemble de solutions. Il est toutefois à noter que le nombre de modèles peut croître de manière exponentielle, et que l'on doit ainsi prendre garde à ne pas générer des instances dont le nombre de solutions est trop important, au risque de voir le temps de calcul nécessaire à leur génération exploser au bout d'un certain nombre d'applications de règles métamorphiques. Cependant, il a été observé que les résultats incorrects étaient de manière générale détectés sur des exemples de taille raisonnable [SEGURA *et al.* 2010, SEGURA *et al.* 2011], ce qui semble indiquer que générer un grand nombre de tests de taille raisonnable, résolus de manière rapide par les logiciels inspectés, est une meilleure stratégie que de passer un temps important à générer un faible nombre de *benchmarks* complexes.

Une fois les tests générés, il ne reste ensuite plus qu'à vérifier que les réponses des logiciels sont conformes à l'ensemble des solutions générées depuis les règles métamorphiques. En ce qui concerne les prouveurs SAT, par exemple, nous avons énuméré les modèles de la formule CNF en ajoutant une clause bloquante pour chaque modèle trouvé, de manière à l'exclure. Il est intéressant de noter que cette approche permet aussi de tester des requêtes d'optimisation sur des formules CNF, puisque l'on dispose de l'ensemble des modèles. Il suffit pour cela de déterminer parmi les modèles lesquels sont optimaux, puis de vérifier la sortie du prouveur (on peut aussi procéder à l'énumération des solutions afin de vérifier l'ordre des solutions fournies par le prouveur).

L'intérêt de ce travail est évident dans la mesure où les cas de test générés ont permis de découvrir des bogues dans des prouveurs testés, y compris dans des prouveurs SAT et sur certaines approches que nous avions développées dans le logiciel p2cudf. De plus, étant donné que les cas de test sont de petite taille, ces bogues ont pu être confirmés par les développeurs des logiciels eux-mêmes quand ils ne l'étaient pas encore dans la littérature ou des rapports.

6.5 Conclusion du chapitre

Dans ce chapitre, nous avons traité de l'utilisation d'un prouveur pseudo-booléen pour l'optimisation de problèmes formulés sous contraintes pseudo-booléennes. Nous avons noté qu'en l'état actuel des choses, sur les problèmes sur lesquels nous avons travaillé, les approches évoluées telles que par exemple, l'utilisation d'une fonction minorante ou la recherche dichotomique de bornes supérieures et inférieures n'étaient pas suffisamment efficaces. De ce fait, nous nous sommes intéressés à l'approche à base de

contraintes de bornes.

Cette technique est basée sur l'architecture dite *SAT incrémental*, c'est-à-dire qu'elle est normalement basée sur un algorithme qui utilise le prouveur comme une boîte noire, dans la mesure où elle ne se préoccupe pas de son état interne. Nous avons fourni un mécanisme permettant une communication privilégiée avec le prouveur dans le but de permettre des gains de temps en évitant de devoir redémarrer le prouveur de zéro entre deux itérations successives. Cette approche a apporté des gains très importants en ce qui concerne l'énumération de solutions, ce qui n'a malheureusement pas été le cas en ce qui concerne le point qui nous intéresse, à savoir l'optimisation.

Enfin, nous avons étudié des approches permettant de détecter des bogues dans des logiciels. En effet, dans le cadre de cette thèse, nous avons développé de multiples algorithmes, que nous avons intégrés dans des prouveurs et dont nous avons vérifié les performances. Or, la frontière entre l'algorithmique et le codage d'une approche est suffisamment épaisse pour qu'en partant d'un algorithme correct, l'approche codée puisse ne pas se révéler exempte de bogues. Nous nous sommes de ce fait aussi intéressés à des méthodes de test, basées sur des règles métamorphiques, qui nous ont permis de détecter des erreurs dans des logiciels, y compris dans certains programmes dans lesquelles nous intégrions nos algorithmes.

Conclusion générale

Dans le cadre de cette thèse, nous nous sommes intéressés à la problématique de l'aide à la décision, une discipline ayant pour but d'aider un décideur humain à effectuer des choix, voire même de prendre des décisions de manière automatique. La nécessité d'employer de telles techniques s'est imposée avec la volonté de traiter des problèmes toujours plus complexes, dépendant d'une quantité de données toujours plus importante. En effet, cette complexité et cet important volume de données empêchent souvent un décideur humain de déterminer une *bonne* solution dans l'ensemble des solutions existantes d'un problème donné, voire même tout simplement de trouver une solution en un temps raisonnable. La nécessité de l'aide à la décision apparaît comme encore plus criante lorsqu'on souhaite obtenir *la meilleure* solution selon un *critère* donné. Dans ce cas, il ne suffit pas de trouver une solution réalisable, mais de trouver une solution réalisable telle que nulle autre solution ne soit considérée comme meilleure selon le critère donné. On passe alors d'un *problème de décision* (déterminer l'existence d'une solution) à un *problème d'optimisation monocritère* (déterminer la meilleure solution possible selon un critère, si une telle solution existe). Un décideur peut aussi souhaiter considérer plusieurs critères, et ainsi faire passer le problème de décision initial concernant l'existence d'une solution à un problème d'*optimisation multicritère*. La difficulté de ce type de problèmes provient du fait que les critères sont généralement contradictoires (les bonnes solutions selon un critère vont être généralement mauvaises sur au moins un des autres critères considérés), et qu'il n'existe donc pas de solution absolument meilleure que toutes les autres. Dans ce cas, il s'agit plutôt de trouver une solution qui constitue un bon *compromis*. Lors de cette thèse, l'objectif était de considérer des problématiques allant du problème de décision aux problèmes d'optimisation complexes, d'un point de vue théorique mais aussi pratique.

Notre étude a concerné deux aspects de l'optimisation booléenne. Nous avons d'abord étudié la complexité de l'optimisation booléenne sous contraintes compilées pour plusieurs familles de représentation de fonctions objectifs et plusieurs langages de compilation. L'objectif est de déterminer s'il est possible d'obtenir des algorithmes d'optimisation polynomiaux en temps après la phase de compilation, alors qu'il n'existe aucun algorithme pouvant s'exécuter en temps polynomial quand les contraintes initiales ne sont pas compilées. Dans les cas où la réponse est positive, bien que l'étape de compilation soit généralement coûteuse, elle peut être amortie et devenir rentable quand un grand nombre de requêtes est effectué par la suite. Nous avons montré qu'il existe des problèmes concrets pour lesquels la compilation est une piste tout à fait intéressante pour des problèmes d'optimisation. Dans ce cadre, nous avons montré que le langage DNNF, pour lequel il existe des compilateurs capables de répondre en temps raisonnable pour un certain nombre d'instances, est un bon choix de langage cible quand la fonction à optimiser est linéaire ; nous avons cependant démontré que dès lors que l'on souhaite considérer des fonctions plus générales, l'optimisation devient de suite difficile. Nous avons aussi démontré des résultats intéressants concernant le langage DNF, qui n'était jusqu'à présent pas considéré comme un bon choix de langage cible. Nous avons prouvé qu'on peut optimiser toute fonction objectif sous contrainte DNF à partir du moment où elle satisfait deux propriétés raisonnables, satisfaites par des familles de fonctions très intéressantes, comme les fonctions sous-modulaires. La première des deux hypothèses est le fait que la fonction puisse être restreinte à un certain nombre de variables, c'est-à-dire que ses propriétés ne changent pas lorsque l'on affecte une valeur à une variable avant de procéder à l'optimisation. Cette propriété attendue est satisfaite par un grand nombre de fonctions objectifs. La deuxième propriété est que la fonction en question puisse être optimisée en temps polynomial lorsqu'aucune contrainte n'est imposée, ce qui est encore une fois peu restrictif, dans la mesure où ajouter des contraintes, peu importe le langage (DNNF, DNF, ...), complexifie les requêtes. Malheureusement, cette dernière hypothèse n'est pas respectée par un grand nombre de fonctions d'agrégation de critères, ce qui exclut *de facto* la compilation de formules propositionnelles pour diminuer la complexité de la recherche d'un compromis quand un agrégateur très général est utilisé.

Le deuxième aspect étudié a conduit, lui, à des contributions pratiques pour l'optimisation quand les contraintes sont des formules CNF, ou des formules plus générales construites comme des conjonctions de contraintes pseudo-booléennes. Ce formalisme permet souvent d'exprimer simplement les contraintes d'un problème, contrairement aux langages cibles de compilation étudiés dans la première partie ; cependant, le simple fait de rechercher une solution au problème étant combinatoire pour ce formalisme, il en va de même pour l'optimisation. De ce fait, nous nous sommes intéressés aux algorithmes de décision et d'optimisation basés sur l'utilisation d'un prouveur SAT dans le but de fournir des améliorations en pratique concernant ces tâches complexes. Puisque les algorithmes d'optimisation passent par la résolution de problèmes de décision, nous nous sommes dans un premier temps intéressés à l'amélioration des performances des prouveurs SAT et PB. Nous avons notamment proposé une approche permettant de découvrir des contraintes pseudo-booléennes dans des formules dans lesquelles elles sont cachées (ces contraintes sont des conséquences d'un ensemble de contraintes du problème), dans le but de pouvoir les employer pour construire une preuve d'incohérence d'une formule de manière plus efficace, utilisant pour cela le système de preuve dit de la *résolution généralisée*. Notre approche permet des gains de performance impressionnants pour résoudre des problèmes jouets. En revanche, pour beaucoup de problèmes applicatifs, cette approche n'est malheureusement pas utile dans la mesure où les prouveurs capables de tirer parti de la résolution généralisée ne sont à l'heure actuelle pas aussi efficaces que les prouveurs CDCL originaux basés sur le système de preuve construit autour de la seule règle de *résolution*. Toutefois, des travaux théoriques ayant montré que la résolution généralisée forme un système de preuves au moins aussi puissant que la résolution, il est à espérer que le manque d'efficacité des prouveurs tirant parti des contraintes de cardinalité soit à imputer à leur jeunesse. De ce fait, une perspective de travail naturelle pourrait être d'étudier le cœur de ces prouveurs dans le but de fournir des implémentations dont l'efficacité se rapprocherait au plus près de celle des prouveurs CDCL traditionnels. Notons aussi que notre approche de détection procède à un prétraitement, et que des contraintes pour lesquelles du temps de calcul a été consacré à leur détection peuvent ne pas être utiles à la résolution du problème ; une amélioration possible pourrait être de considérer la détection de contraintes durant la recherche, *a priori* au moment où un conflit apparaît sur une clause.

Nous nous sommes ensuite intéressés à l'utilisation des prouveurs SAT pour l'optimisation. Dans un premier temps, nous avons testé l'efficacité de méthodes alternatives proposées récemment comme l'optimisation basée sur l'énumération des solutions optimales d'une fonction minorant la fonction objectif, cette méthode semblant prometteuse pour les fonctions non linéaires ce qui est le cas de beaucoup de fonctions d'agrégation de critères. Malgré l'ingéniosité de la méthode proposée, celle-ci s'est révélée inefficace dans notre cas. Notre hypothèse est que cette approche s'adapte mal aux problèmes en variables entières, en particulier quand un grand nombre de solutions ont la même valeur de coût, mais aussi qu'il est difficile de trouver une fonction minorante qui consistue une bonne approximation de la fonction objectif pour le type de fonctions que nous avons retenu.

Nous avons alors étudié la structure incrémentale des prouveurs d'optimisation, qui appellent les prouveurs SAT pour déterminer si une solution existe pour un coût inférieur à une certaine valeur, et qui déterminent ainsi un majorant et un minorant, ce qui permet finalement d'obtenir une solution optimale. Plus précisément, une première solution est calculée, qui fournit un premier majorant. Cette borne est ensuite mise à jour quand une solution meilleure est trouvée. De manière naïve, cette approche incrémentale relance le prouveur à chaque mise à jour de la contrainte de borne, en gardant toutefois une partie des informations apprises jusqu'à lors. Nous avons proposé une approche permettant d'établir une connexion privilégiée entre les prouveurs de cohérence et les algorithmes les employant, dans le but qu'elle permette de ne pas redémarrer le prouveur SAT lors du processus d'optimisation. Cette approche permet en fait de générer des contraintes à la volée, en faisant croire au prouveur qu'elles étaient présentes depuis son lancement, ce qui permet de générer les contraintes de borne sans avoir à cesser l'exécution du prouveur. Bien que nos résultats concernant cette étude montrent des gains très importants pour l'énumération

de solutions (dont l'algorithme de base effectue un grand nombre d'appels au prouveur SAT), elle n'a malheureusement pas apporté d'amélioration en ce qui concerne l'optimisation. Nous avons cependant pointé, concernant notre approche, une faiblesse dans la gestion des contraintes les plus générales considérées, à savoir les contraintes pseudo-booléennes. À l'heure actuelle, nous n'avons réussi à déterminer ni une meilleure façon de gérer ces contraintes, ni une preuve que ceci était impossible : il s'agit *de facto* d'une perspective de recherche qui mérite selon nous d'être étudiée.

Enfin, nous sommes partis du constat que les prouveurs que nous utilisons ne sont potentiellement pas exempts de bogues. Nous avons dans ce but travaillé sur la vérification de prouveurs, en particulier sur les règles métamorphiques permettant de générer un grand nombre de cas de test de petite taille pour le format CNF (parmi d'autres). Notre travail n'a pas été vain, dans la mesure où les fichiers de test générés à partir de notre approche ont permis de détecter des bogues dans plusieurs logiciels, incluant des prouveurs SAT.

Annexe A

Contributions publiées dans le cadre de cette thèse

Article publiés dans un journal d'audience internationale

- Sergio Segura, Amador Durán, Ana B. Sánchez, Daniel Le Berre, Emmanuel Lonca, Antonio Ruiz-Cortés, « Automated Metamorphic Testing of Variability Analysis Tools » dans *Software Testing, Verification and Reliability* (STVR), 25(2) :138–163, 2015.

Articles publiés dans les actes d'une conférence internationale

- Armin Biere, Daniel Le Berre, Emmanuel Lonca, Norbert Manthey, « Detecting cardinality constraints in CNF » dans *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing* (SAT'14).

Articles publiés dans un journal d'audience nationale

- Daniel le Berre, Emmanuel Lonca, « Réutiliser ou adapter les prouveurs SAT pour l'optimisation booléenne » dans *Revue d'Intelligence Artificielle* (RIA), 28(5) :615–636, 2014. .

Articles publiés dans les actes d'une conférence nationale

- Armin Biere, Daniel le Berre, Emmanuel Lonca, Norbert Manthey, « Détection de contraintes de cardinalité dans les CNF » dans *Actes de la 10ème édition des Journées Francophones de Programmation par Contraintes* (JFPC'14).
- Daniel le Berre, Emmanuel Lonca, « Réutiliser ou adapter les prouveurs SAT pour l'optimisation booléenne » dans *Actes de la 9ème édition des Journées Francophones de Programmation par Contraintes* (JFPC'13).
- Daniel Le Berre, Emmanuel Lonca, Pierre Marquis, Anne Parrain, « Calcul de solutions équilibrées Pareto optimales : application au problème de gestion des dépendances logicielles » dans *Actes de la 8ème édition des Journées Francophones de Programmation par Contraintes* (JFPC'12).
- Daniel Le Berre, Emmanuel Lonca, Pierre Marquis, Anne Parrain, « Optimisation multicritère pour la gestion de dépendances logicielles : utilisation de la norme de Tchebycheff » dans *Actes de la 18ème édition de Reconnaissance des Formes et Intelligence Artificielle* (RFIA'12).

Table des figures

Chapitre 1 : Problèmes de décision

1.1 Illustration d'un problème de gestion de dépendances au format CUDF. 13
1.2 Illustration de la hiérarchie polynomiale. 16
1.3 Illustration des deux premiers niveaux de la hiérarchie polynomiale. 17
1.4 Une formule propositionnelle et la formule NNF correspondante après application des lois de De Morgan. 18

Chapitre 2 : Problèmes d'optimisation

2.1 Illustration des méthodes « agréger puis comparer » (AC) et « comparer puis agréger » (CA). 33
2.2 Illustration du front de Pareto. 35
2.3 Illustration du point idéal pour un problème bicritère. 37
2.4 Illustration d'une agrégation de deux critères en utilisant la moyenne pondérée. 38
2.5 Illustration d'une agrégation de deux critères en utilisant un opérateur de somme pondérée ordonnée. 41

Chapitre 3 : Compilation de connaissances : le langage NNF et ses sous-langages

3.1 Mise sous forme NNF d'une formule quelconque. 47
3.2 Relations d'inclusion entre les sous-langages de f-NNF. 48
3.3 Relations d'inclusion entre les sous-langages de DNNF. 50
3.4 Nœuds décision, formules FBDD et formules OBDD 51
3.5 Exemples de v-tree 52

Chapitre 4 : Optimisation sous contraintes NNF

4.1 Le langage NNF et ses sous-langages. 65
4.2 Optimisation d'une formule DNNF par une fonction objectif linéaire. 70

Chapitre 5 : Prouveurs pseudo-booléens

5.1 Illustration d'un arbre de résolution pour une formule incohérente. 90
5.2 Illustration de l'algorithme DPLL. 95
5.3 Exemple de graphe d'implication 97
5.4 Encodage d'une contrainte *AtMost*-1 avec le *two product encoding* 112
5.5 Illustration d'une contrainte *AtMost*-1 quand les clauses binaires sont vues comme un NAND graphe. 113
5.6 Détection d'une contrainte AtMost-1 encodée via le two product encoding 117

5.7 Résultats expérimentaux : détection de contraintes de cardinalité (PHP) 123
5.8 Résultats expérimentaux : détection de contraintes de cardinalité (SGEN Unsat) 123

Chapitre 6 : Optimisation par prouveurs pseudo-booléens
6.1 Optimisation (minimisation) d'une fonction pseudo-booléenne linéaire par renforcement de contrainte de borne 129
6.2 Optimisation d'une fonction pseudo-booléenne non linéaire par énumération des solutions optimales d'une fonction minorante linéaire 134
6.3 Boîte noire *vs* à la volée - distribution des solutions énumérées pour PB 142
6.4 Boîte noire *vs* à la volée - distribution des temps de résolution des instances PB 143
6.5 Boîte noire *vs* à la volée - temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée (PB) 143
6.6 Boîte noire *vs* à la volée - distribution des solutions énumérées pour MAXSAT 145
6.7 Boîte noire *vs* à la volée - distribution des temps de résolution des instances MAXSAT . 145
6.8 Boîte noire *vs* à la volée - temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée (MAXSAT) 146

Liste des tableaux

1.1 Sémantique des opérateurs usuels de la logique propositionnelle. 20

Chapitre 3 : Compilation de connaissances : le langage NNF et ses sous-langages

3.1 Complexité des requêtes pour les langages considérés. 55
3.2 Complexité des transformation pour les langages considérés. 57

Chapitre 4 : Optimisation sous contraintes NNF

4.1 Complexité de la requête OPT pour les langages satisfiant CO 74
4.2 Résumé des résultats de complexité pour l'optimisation sous contrainte DNNF 80

Chapitre 5 : Prouveurs pseudo-booléens

5.1 Nombres de variables et de clauses nécessaires pour encoder une contrainte de cardinalité pour différents encodages. 112

Chapitre 6 : Optimisation par prouveurs pseudo-booléens

6.1 Boîte noire *vs* à la volée - Comparaison boîte noire *vs* ajout de clauses à la volée pour l'énumération de solutions (2 min) 137
6.2 Boîte noire *vs* à la volée - Comparaison des temps de calcul moyens et médians pour l'énumération de solutions (2 min) 137
6.3 Boîte noire *vs* à la volée - Comparaison boîte noire *vs* ajout de clauses à la volée pour l'énumération de solutions (20 min) 137
6.4 Boîte noire *vs* à la volée - Comparaison des temps de calcul moyens et médians pour l'énumération de solutions (20 min) 137
6.5 Boîte noire *vs* à la volée - instances PB10 résolues (et UNSAT) 141
6.6 Boîte noire *vs* à la volée - instances MAXSAT10 résolues 144

Liste des Algorithmes

Chapitre 4 : Optimisation sous contraintes NNF
4.1 Optimisation d'une formule DNNF par une fonction objectif linéaire. 69

Chapitre 5 : Prouveurs pseudo-booléens
5.1 Recherche Locale 85
5.2 Résolution de Robinson 89
5.3 Algorithme DP 92
5.4 Algorithme DPLL 94
5.5 Algorithme CDCL 99
5.6 Détection syntaxique du two product encoding 115
5.7 Détection syntaxique du two product encoding – fonction auxiliaire 116
5.8 Recherche initiale de candidats à l'extension d'une contrainte de cardinalité 119
5.9 Raffinement de l'ensemble des candidats à l'extension d'une contrainte de cardinalité . . . 120
5.10 Détection de contrainte de cardinalité maximale depuis une clause. 120
5.11 Détection des contraintes de cardinalité maximales depuis une CNF. 121

Chapitre 6 : Optimisation par prouveurs pseudo-booléens
6.1 Optimisation par renforcement 129
6.2 Optimisation par fonction minorante linéaire 134
6.3 Optimisation par ajout de contrainte de borne dynamique 139

Bibliographie

[ABÍO *et al.* 2012] Ignasi ABÍO, Robert NIEUWENHUIS, Albert OLIVERAS, Enric RODRÍGUEZ-CARBONELL, et Valentin MAYER-EICHBERGER. « A New Look at BDDs for Pseudo-Boolean Constraints ». *J. Artif. Intell. Res. (JAIR)*, 45 :443–480, 2012.

[AKERS 1978] Sheldon B. AKERS. « Binary decision diagrams ». *Computers, IEEE Transactions on*, 100(6) :509–516, 1978.

[ALOUL *et al.* 2002a] Fadi A ALOUL, Arathi RAMANI, Igor MARKOV, et Karem SAKALLAH. « PBS : a backtrack-search pseudo-boolean solver and optimizer ». Dans *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pages 346–353, 2002.

[ALOUL *et al.* 2002b] Fadi A. ALOUL, Arathi RAMANI, Igor L. MARKOV, et Karem A. SAKALLAH. « Generic ILP versus specialized 0-1 ILP : an update ». Dans Pileggi et Kuehlmann [PILEGGI & KUEHLMANN 2002], pages 450–457.

[AMILHASTRE *et al.* 2014] Jérôme AMILHASTRE, Hélène FARGIER, Alexandre NIVEAU, et Cédric PRALET. « Compiling CSPs : A Complexity Map of (Non-Deterministic) Multivalued Decision Diagrams ». *International Journal on Artificial Intelligence Tools*, 23(4), 2014.

[ANSÓTEGUI & MANYÀ 2004] Carlos ANSÓTEGUI et Felip MANYÀ. « Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables ». Dans Hoos et Mitchell [HOOS & MITCHELL 2005], pages 1–15.

[ANSÓTEGUI *et al.* 2007] Carlos ANSÓTEGUI, Jose LARRUBIA, Chu Min LI, et Felip MANYÀ. « Exploiting multivalued knowledge in variable selection heuristics for SAT solvers ». *Ann. Math. Artif. Intell.*, 49(1-4) :191–205, 2007.

[ANSÓTEGUI *et al.* 2010] Carlos ANSÓTEGUI, Maria Luisa BONET, et Jordi LEVY. « A New Algorithm for Weighted Partial MaxSAT ». Dans Fox et Poole [FOX & POOLE 2010].

[ANSÓTEGUI *et al.* 2012] Carlos ANSÓTEGUI, Maria Luisa BONET, Joel GABÀS, et Jordi LEVY. « Improving SAT-Based Weighted MaxSAT Solvers ». Dans Milano [MILANO 2012], pages 86–101.

[ANSÓTEGUI *et al.* 2013a] Carlos ANSÓTEGUI, Maria Luisa BONET, Joel GABÀS, et

Jordi LEVY. « Improving WPM2 for (Weighted) Partial MaxSAT ». Dans Schulte [SCHULTE 2013], pages 117–132.

[ANSÓTEGUI *et al.* 2013b] Carlos ANSÓTEGUI, Maria Luisa BONET, et Jordi LEVY. « SAT-based MaxSAT algorithms ». *Artif. Intell.*, 196 :77–105, 2013.

[ANSÓTEGUI 2004] Carlos José ANSÓTEGUI. « *Complete SAT solvers for Many-Valued CNF Formulas* ». PhD thesis, University of Lleida, 2004.

[ARGELICH *et al.* 2010] Josep ARGELICH, Daniel Le BERRE, Inês LYNCE, João P. Marques SILVA, et Pascal RAPICAULT. « Solving Linux Upgradeability Problems Using Boolean Optimization ». Dans Lynce et Treinen [LYNCE & TREINEN 2010], pages 11–22.

[ARROW *et al.* 1986] Kenneth Joseph ARROW, Michael D INTRILIGATOR, Werner HILDENBRAND, et Hugo SONNENSCHEIN. *Handbook of mathematical economics*. North-Holland, 1986.

[ARROW 2012] Kenneth J ARROW. *Social choice and individual values*, volume 12. Yale University Press, 2012.

[ARTHO *et al.* 2013] Cyrille ARTHO, Armin BIERE, et Martina SEIDL. « Model-Based Testing for Verification Back-Ends ». Dans Veanes et Viganò [VEANES & VIGANÒ 2013], pages 39–55.

[ASÍN *et al.* 2009] Roberto ASÍN, Robert NIEUWENHUIS, Albert OLIVERAS, et Enric RODRÍGUEZ-CARBONELL. « Cardinality Networks and Their Applications ». Dans Kullmann [KULLMANN 2009], pages 167–180.

[ASÍN *et al.* 2011] Roberto ASÍN, Robert NIEUWENHUIS, Albert OLIVERAS, et Enric RODRÍGUEZ-CARBONELL. « Cardinality Networks : a theoretical and empirical study ». *Constraints*, 16(2) :195–221, 2011.

[ATSERIAS *et al.* 2011] Albert ATSERIAS, Johannes Klaus FICHTE, et Marc THURLEY. « Clause-Learning Algorithms with Many Restarts and Bounded-Width Resolution ». *J. Artif. Intell. Res. (JAIR)*, 40 :353–373, 2011.

[AUDEMARD & SIMON 2007] Gilles AUDEMARD et Laurent SIMON. « GUNSAT : A Greedy Local Search Algorithm for Unsatisfiability ». Dans Veloso [VELOSO 2007], pages 2256–2261.

[AUDEMARD & SIMON 2009a] Gilles AUDEMARD et Laurent SIMON. « GLUCOSE : a solver that predicts learnt clauses quality ». *SAT Competition*, pages 7–8, 2009.

[AUDEMARD & SIMON 2009b] Gilles AUDEMARD et Laurent SIMON. « Predicting Learnt Clauses Quality in Modern SAT Solvers ». Dans Boutilier [BOUTILIER 2009], pages 399–404.

[AUDEMARD *et al.* 2009] Gilles AUDEMARD, Jean-Marie LAGNIEZ, Bertrand MAZURE, et Lakhdar SAIS. « Learning in Local Search ».

Dans *ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence, Newark, New Jersey, USA, 2-4 November 2009*, pages 417–424, 2009.

[AUDEMARD *et al.* 2011] Gilles AUDEMARD, Jean-Marie LAGNIEZ, Bertrand MAZURE, et Lakhdar SAIS. « On Freezing and Reactivating Learnt Clauses ». Dans Sakallah et Simon [SAKALLAH & SIMON 2011], pages 188–200.

[BACCHUS *et al.* 2003] Fahiem BACCHUS, Shannon DALMAO, et Toniann PITASSI. « Algorithms and complexity results for# SAT and Bayesian inference ». Dans *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 340–351. IEEE, 2003.

[BAHAR *et al.* 1993] R. Iris BAHAR, Erica A. FROHM, Charles M. GAONA, Gary D. HACHTEL, Enrico MACII, Abelardo PARDO, et Fabio SOMENZI. « Algebraic decision diagrams and their applications ». Dans *Proceedings of the International Conference Computer-Aided Design (ICCAD)*, pages 188–191, 1993.

[BAILLEUX & BOUFKHAD 2003] Olivier BAILLEUX et Yacine BOUFKHAD. « Efficient CNF Encoding of Boolean Cardinality Constraints ». Dans Rossi [ROSSI 2003], pages 108–122.

[BAILLEUX *et al.* 2009] Olivier BAILLEUX, Yacine BOUFKHAD, et Olivier ROUSSEL. « New Encodings of Pseudo-Boolean Constraints into CNF ». Dans Kullmann [KULLMANN 2009], pages 181–194.

[BAJCSY 1993] Ruzena BAJCSY, éditeur. *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*. Morgan Kaufmann, 1993.

[BALINT & MANTHEY 2013] Adrian BALINT et Norbert MANTHEY. « Boosting the Performance of SLS and CDCL Solvers by Preprocessor Tuning ». Dans Le Berre [LE BERRE 2014], pages 1–14.

[BALINT *et al.* 2013] Adrian BALINT, Anton BELOV, Marijn HEULE, et Matti JÄRVISALO, éditeurs. *Proceedings of SAT Competition 2013 ; Solver and Benchmark Descriptions*, volume B-2013-1. University of Helsinki, Department of Computer Science Series of Publications B, 2013.

[BARAHONA *et al.* 2014] Pedro BARAHONA, Steffen HÖLLDOBLER, et Van-Hau NGUYEN. « Representative Encodings to Translate Finite CSPs into SAT ». Dans Simonis [SIMONIS 2014], pages 251–267.

[BARAL *et al.* 2007] Chitta BARAL, Gerhard BREWKA, et John S. SCHLIPF, éditeurs. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 de *Lecture Notes in Computer Science*. Springer, 2007.

[BARBA-ROMERO & POMEROL 1997] Sergio BARBA-ROMERO et Jean-Charles POMEROL. *Decisiones multicriterio : fundamentos teóricos y utilización práctica*. Universidad de Alcalá, 1997.

[BARRETT *et al.* 2009] Clark W. BARRETT, Roberto SEBASTIANI, Sanjit A. SESHIA, et Cesare TINELLI. Satisfiability Modulo Theories. Dans Biere et al. [BIERE *et al.* 2009], pages 825–885.

[BARTH & STADTWALD 1995] Peter BARTH et Im STADTWALD. « A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization ». Rapport technique, Max-Planck-Institut für Informatik, 1995.

[BARTH 1993] Peter BARTH. « Linear 0-1 Inequalities and Extended Clauses ». Dans Voronkov [VORONKOV 1993], pages 40–51.

[BAYARDO JR. & SCHRAG 1997] Roberto J. BAYARDO JR. et Robert SCHRAG. « Using CSP Look-Back Techniques to Solve Real-World SAT Instances ». Dans Kuipers et Webber [KUIPERS & WEBBER 1997], pages 203–208.

[BEN-HAIM *et al.* 2012] Yael BEN-HAIM, Alexander IVRII, Oded MARGALIT, et Arie MATSLIAH. « Perfect Hashing and CNF Encodings of Cardinality Constraints ». Dans Cimatti et Sebastiani [CIMATTI & SEBASTIANI 2012], pages 397–409.

[BENFERHAT *et al.* 2014] Salem BENFERHAT, Sylvain LAGRUE, et Julien ROSSIT. « Sum-based weighted belief base merging : From commensurable to incommensurable framework ». *Int. J. Approx. Reasoning*, 55(9) :2083–2108, 2014.

[BERRAH & CLIVILLÉ 2007] Lamia BERRAH et Vincent CLIVILLÉ. « Towards an aggregation performance measurement system model in a supply chain context ». *Computers in Industry*, 58(7) :709–719, 2007.

[BIERE & GOMES 2006] Armin BIERE et Carla P. GOMES, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 de *Lecture Notes in Computer Science*. Springer, 2006.

[BIERE *et al.* 2009] Armin BIERE, Marijn HEULE, Hans van MAAREN, et Toby WALSH, éditeurs. *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BIERE *et al.* 2013] Armin BIERE, Amir NAHIR, et Tanja E. J. VOS, éditeurs. *Hardware and Software : Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 de *Lecture Notes in Computer Science*. Springer, 2013.

[BIERE *et al.* 2014] Armin BIERE, Daniel Le BERRE, Emmanuel LONCA, et Norbert MANTHEY. « Detecting Cardinality Constraints in CNF ». Dans Sinz et Egly [SINZ & EGLY 2014], pages 285–301.

[BIERE 2008a] Armin BIERE. « Adaptive Restart Strategies for Conflict Driven SAT Solvers ». Dans Büning et Zhao [BÜNING & ZHAO 2008], pages 28–33.

[BIERE 2008b] Armin BIERE. « PicoSAT Essentials ». *JSAT*, 4(2-4) :75–97, 2008.

[BIERE 2009] Armin BIERE. Bounded Model Checking. Dans Biere et al. [BIERE *et al.* 2009], pages 457–481.

[BIERE 2013] Armin BIERE. « Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013 ». Dans Balint et al. [BALINT *et al.* 2013], pages 51–52.

[BOOLE 1854] George BOOLE. *An investigation of the laws of thought : on which are founded the mathematical theories of logic and probabilities*, volume 2. Walton and Maberly, 1854.

[BOROS & HAMMER 2002] Endre BOROS et Peter L. HAMMER. « Pseudo-Boolean optimization ». *Discrete Applied Mathematics*, 123(1-3) :155–225, 2002.

[BOUTILIER 2009] Craig BOUTILIER, éditeur. *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009.

[BOUVERET & LEMAÎTRE 2009] Sylvain BOUVERET et Michel LEMAÎTRE. « Computing leximin-optimal solutions in constraint networks ». *Artif. Intell.*, 173(2) :343–364, 2009.

[BOUYSSOU *et al.* 2006] Denis BOUYSSOU, Thierry MARCHANT, Marc PIRLOT, Alexis TSOUKIAS, et Philippe VINCKE. *Evaluation and decision models with multiple criteria : Stepping stones for the analyst*. Springer, 2006.

[BOUYSSOU *et al.* 2013] Denis BOUYSSOU, Didier DUBOIS, Henri PRADE, et Marc PIRLOT. *Decision Making Process : Concepts and Methods*. John Wiley & Sons, 2013.

[BOVA *et al.* 2014] Simone BOVA, Florent CAPELLI, Stefan MENGEL, et Friedrich SLIVOVSKY. « Expander CNFs have Exponential DNNF Size ». *CoRR*, abs/1411.1995, 2014.

[BRAND 2003] Daniel BRAND. Verification of large synthesized designs. Dans *The Best of ICCAD*, pages 65–72. Springer, 2003.

[BRAUNSTEIN *et al.* 2005] Alfredo BRAUNSTEIN, Marc MÉZARD, et Riccardo ZECCHINA. « Survey propagation : An algorithm for satisfiability ». *Random Struct. Algorithms*, 27(2) :201–226, 2005.

[BRISOUX *et al.* 1999] Laure BRISOUX, Éric GRÉGOIRE, et Lakhdar SAIS. « Improving Backtrack Search for SAT by Means of Redundancy

». Dans Ras et Skowron [RAS & SKOWRON 1999], pages 301–309.

[BRUMMAYER *et al.* 2010] Robert BRUMMAYER, Florian LONSING, et Armin BIERE. « Automated Testing and Debugging of SAT and QBF Solvers ». Dans Strichman et Szeider [STRICHMAN & SZEIDER 2010], pages 44–57.

[BRYANT 1986] Randal E. BRYANT. « Graph-Based Algorithms for Boolean Function Manipulation ». *IEEE Trans. Computers*, 35(8) :677–691, 1986.

[BÜNING & ZHAO 2008] Hans Kleine BÜNING et Xishun ZHAO, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 de *Lecture Notes in Computer Science*. Springer, 2008.

[BURO & BÜNING 1992] Michael BURO et H Kleine BÜNING. *Report on a SAT competition*. Fachbereich Math.-Informatik, Univ. Gesamthochschule, 1992.

[CHAI & KUEHLMANN 2005] Donald CHAI et Andreas KUEHLMANN. « A fast pseudo-boolean constraint solver ». *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(3) :305–317, 2005.

[CHAN *et al.* 2007] W. CHAN, S. CHEUNG, et K. LEUNG. « A Metamorphic Testing Approach for Online Testing of Service-Oriented Software Applications ». *Int. J. Web Service Res.*, 4(2) :61–81, 2007.

[CHATALIC & SIMON 2000] Philippe CHATALIC et Laurent SIMON. « ZRES : The Old Davis-Putman Procedure Meets ZBDD ». Dans McAllester [MCALLESTER 2000], pages 449–454.

[CHEN *et al.* 1998] Tsong Y CHEN, Shing C CHEUNG, et SM YIU. « Metamorphic testing : a new approach for generating next test cases ». Rapport technique, Technical Report HKUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[CHEN *et al.* 2002] Tsong Yueh CHEN, Jianqiang FENG, et T. H. TSE. « Metamorphic Testing of Programs on Partial Differential Equations : A Case Study ». Dans *26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life : Development and Redevelopment, 26-29 August 2002, Oxford, England, Proceedings*, pages 327–333. IEEE Computer Society, 2002.

[CHEN *et al.* 2004] TY CHEN, DH HUANG, TH TSE, et Zhi Quan ZHOU. « Case studies on the selection of useful relations in metamorphic testing ». Dans *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, 2004.

[CHEN 2010] Jing-Chao CHEN. « A new SAT encoding of the at-most-one constraint ». Dans *In Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation*, 2010.

[CHOQUET 1953] Gustave CHOQUET. « Theory of capacities ». Dans *Annales de l'Institut Fourier*, volume 5, 1953.

[CHVÁTAL 1973] Vašek CHVÁTAL. « Edmonds polytopes and a hierarchy of combinatorial problems ». *Discrete mathematics*, 4(4) :305–337, 1973.

[CIMATTI & SEBASTIANI 2012] Alessandro CIMATTI et Roberto SEBASTIANI, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 de *Lecture Notes in Computer Science*. Springer, 2012.

[CLIVILLÉ *et al.* 2007] Vincent CLIVILLÉ, Lamia BERRAH, et Gilles MAURIS. « Quantitative expression and aggregation of performance measurements based on the MACBETH multi-criteria method ». *International Journal of Production economics*, 105(1) :171–189, 2007.

[COBHAM 1965] Alan COBHAM. « The intrinsic computational difficulty of functions ». Dans *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30, 1965.

[COOK *et al.* 1987] William COOK, Collette R COULLARD, et Gy TURÁN. « On the complexity of cutting-plane proofs ». *Discrete Applied Mathematics*, 18(1) :25–38, 1987.

[COOK 1971] Stephen A COOK. « The complexity of theorem-proving procedures ». Dans *Proceedings of the third annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[DALAL 1992] Mukesh DALAL. « Efficient Propositional Constraint Propagation ». Dans Swartout [SWARTOUT 1992], pages 409–414.

[DALTON 1920] Hugh DALTON. « The measurement of the inequality of incomes ». *The Economic Journal*, pages 348–361, 1920.

[DARWICHE & MARQUIS 2002] Adnan DARWICHE et Pierre MARQUIS. « A Knowledge Compilation Map ». *J. Artif. Intell. Res. (JAIR)*, 17 :229–264, 2002.

[DARWICHE & MARQUIS 2004] Adnan DARWICHE et Pierre MARQUIS. « Compiling propositional weighted bases ». *Artif. Intell.*, 157(1-2) :81–113, 2004.

[DARWICHE 1999] Adnan DARWICHE. « Compiling Knowledge into Decomposable Negation Normal Form ». Dans Dean [DEAN 1999], pages 284–289.

[DARWICHE 2001] Adnan DARWICHE. « Decomposable negation normal form ». *J. ACM*, 48(4) :608–647, 2001.

[DARWICHE 2002] Adnan DARWICHE. « A Compiler for Deterministic, Decomposable Negation Normal Form ». Dans Dechter et Sutton [DECHTER & SUTTON 2002], pages 627–634.

[DARWICHE 2009] Adnan DARWICHE. *Modeling and reasoning with Bayesian networks*. Cambridge University Press, 2009.

[DAVIS & PUTNAM 1960] Martin DAVIS et Hilary PUTNAM. « A Computing Procedure for Quantification Theory ». *J. ACM*, 7(3) :201–215, 1960.

[DAVIS *et al.* 1962] Martin DAVIS, George LOGEMANN, et Donald W. LOVELAND. « A machine program for theorem-proving ». *Commun. ACM*, 5(7) :394–397, 1962.

[DEAN 1999] Thomas DEAN, éditeur. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*. Morgan Kaufmann, 1999.

[DECHTER & SUTTON 2002] Rina DECHTER et Richard S. SUTTON, éditeurs. *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*. AAAI Press / The MIT Press, 2002.

[DECHTER 2000] Rina DECHTER, éditeur. *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 de *Lecture Notes in Computer Science*. Springer, 2000.

[DEQUEN & DUBOIS 2003] Gilles DEQUEN et Olivier DUBOIS. « kcnfs : An Efficient Solver for Random k-SAT Formulae ». Dans Giunchiglia et Tacchella [GIUNCHIGLIA & TACCHELLA 2004], pages 486–501.

[DIXON 2004] Heidi DIXON. « *Automating pseudo-boolean inference within a dpll framework* ». PhD thesis, University of Oregon, 2004.

[DOMSHLAK & HOFFMANN 2007] Carmel DOMSHLAK et Jörg HOFFMANN. « Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. ». *J. Artif. Intell. Res.(JAIR)*, 30 :565–620, 2007.

[EDMONDS 1965a] Jack EDMONDS. « Minimum partition of a matroid into independent subsets ». *J. Res. Nat. Bur. Standards Sect. B*, 69 :67–72, 1965.

[EDMONDS 1965b] Jack EDMONDS. « Paths, trees, and flowers ». *Canadian Journal of Mathematics*, 17(3) :449–467, 1965.

[EÉN & SÖRENSSON 2003a] Niklas EÉN et Niklas SÖRENSSON. « An Extensible SAT-solver ». Dans Giunchiglia et Tacchella [GIUNCHIGLIA & TACCHELLA 2004], pages 502–518.

[EÉN & SÖRENSSON 2003b] Niklas EÉN et Niklas SÖRENSSON. « Temporal induction by incremental SAT solving ». *Electr. Notes Theor. Comput. Sci.*, 89(4) :543–560, 2003.

[EÉN & SÖRENSSON 2006] Niklas EÉN et Niklas SÖRENSSON. « Translating Pseudo-Boolean Constraints into SAT ». *JSAT*, 2(1-4) :1–26, 2006.

[FANG & RUML 2004] Hai FANG et Wheeler RUML. « Complete Local Search for Propositional Satisfiability ». Dans McGuinness et Ferguson [MCGUINNESS & FERGUSON 2004], pages 161–166.

[FARGIER & MARQUIS 2007] Hélène FARGIER et Pierre MARQUIS. « On Valued Negation Normal Form Formulas ». Dans Veloso [VELOSO 2007], pages 360–365.

[FELLOWS & LANGSTON 1987] Michael R. FELLOWS et Michael A. LANGSTON. « Nonconstructive advances in polynomial-time complexity ». *Information Processing Letters*, 26(3) :157 – 162, 1987.

[FIGUEIRA *et al.* 2005] José FIGUEIRA, Salvatore GRECO, et Matthias EHRGOTT. *Multiple criteria decision analysis : state of the art surveys*. Springer, 2005.

[FIKES & LEHNERT 1993] Richard FIKES et Wendy G. LEHNERT, éditeurs. *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993*. AAAI Press / The MIT Press, 1993.

[FISHBURN 1970] Peter C FISHBURN. « Utility theory for decision making ». Rapport technique, DTIC Document, 1970.

[FODOR & ROUBENS 1994] Janos C FODOR et MR ROUBENS. *Fuzzy preference modelling and multicriteria decision support*, volume 14. Springer Science & Business Media, 1994.

[FOX & GOMES 2008] Dieter FOX et Carla P. GOMES, éditeurs. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*. AAAI Press, 2008.

[FOX & POOLE 2010] Maria FOX et David POOLE, éditeurs. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[FRISCH & GIANNAROS 2010] Alan M. FRISCH et Paul A. GIANNAROS. « SAT Encodings of the AT-Most-k Constraint : Some Old, Some New, Some Fast, Some Slow ». Dans *Proceedings of the The 9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010)*, 2010.

[FU & MALIK 2006a] Zhaohui FU et Sharad MALIK. « On Solving the Partial MAX-SAT Problem ». Dans Biere et Gomes [BIERE & GOMES 2006], pages 252–265.

[FU & MALIK 2006b] Zhaohui FU et Sharad MALIK. « Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search ». Dans Hassoun [HASSOUN 2006], pages 852–859.

[FUJISHIGE & ISOTANI 2011] Satoru FUJISHIGE et Shigueo ISOTANI. « A submodular function minimization algorithm based on the minimum-norm base ». *Pacific Journal of Optimization*, 7(1) :3–17, 2011.

[FUJISHIGE 2005] S. FUJISHIGE. *Submodular Functions and Optimization*. Elsevier, 2005.

[GALESI *et al.* 2014] Nicola GALESI, Pavel PUDLÁK, et Neil THAPEN. « The space complexity of cutting planes refutations ». *Electronic Colloquium on Computational Complexity (ECCC)*, 21 :138, 2014.

[GAREY & JOHNSON 1979] M. R. GAREY et David S. JOHNSON. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GEBSER *et al.* 2007] Martin GEBSER, Benjamin KAUFMANN, André NEUMANN, et Torsten SCHAUB. « Conflict-Driven Answer Set Enumeration ». Dans Baral et al. [BARAL *et al.* 2007], pages 136–148.

[GENT & NIGHTINGALE 2004] Ian P GENT et Peter NIGHTINGALE. « A new encoding of AllDifferent into SAT ». *Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110, 2004.

[GENT & PROSSER 2002] I.P. GENT et B.M. PROSSER, P. andSmith. « A 0/1 encoding of the GACLex constraint for pairs of vectors ». Dans *ECAI 2002 workshop W9 : Modelling and Solving Problems with Constraints*. University of Glasgow, 2002.

[GENT 2002] Ian P. GENT. « Arc Consistency in SAT ». Dans van Harmelen [VAN HARMELEN 2002], pages 121–125.

[GERGOV & MEINEL 1993] Jordan GERGOV et Christoph MEINEL. « Combinational Logic Verification with FBDDs ». *Universität Trier, Mathematik/Informatik, Forschungsbericht*, 93-08, 1993.

[GIUNCHIGLIA & TACCHELLA 2004] Enrico GIUNCHIGLIA et Armando TACCHELLA, éditeurs. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 de *Lecture Notes in Computer Science*. Springer, 2004.

[GLOVER 1989] Fred GLOVER. « Tabu Search - Part I ». *INFORMS Journal on Computing*, 1(3) :190–206, 1989.

[GLOVER 1990] Fred GLOVER. « Tabu Search - Part II ». *INFORMS Journal on Computing*, 2(1) :4–32, 1990.

[GOEL *et al.* 2010] Gagan GOEL, Pushkar TRIPATHI, et Lei WANG. « Combinatorial Problems with Discounted Price Functions in Multi-agent Systems. ». Dans *FSTTCS*, pages 436–446, 2010.

[GOLDBERG & NOVIKOV 2002] Evguenii I. GOLDBERG et Yakov NOVIKOV. « BerkMin : A Fast and Robust Sat-Solver ». Dans *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pages 142–149, 2002.

[GOLDBERG 1979] Allen T GOLDBERG. *On the complexity of the satisfiability problem*. New York University, 1979.

[GOLDEN & PERNY 2010] Boris GOLDEN et Patrice PERNY. « Infinite order Lorenz dominance for fair multiagent optimization ». Dans *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : volume 1-Volume 1*, pages 383–390. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[GOMES *et al.* 2000] Carla P. GOMES, Bart SELMAN, Nuno CRATO, et Henry A. KAUTZ. « Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems ». *J. Autom. Reasoning*, 24(1/2) :67–100, 2000.

[GOMORY 1958] Ralph E GOMORY. « Outline of an algorithm for integer solutions to linear programs ». *Bulletin of the American Mathematical society*, 64(5) :275–278, 1958.

[GONZALES *et al.* 2008] Christophe GONZALES, Patrice PERNY, et Sergio QUEIROZ. « Preference Aggregation with Graphical Utility Models ». Dans Dieter FOX et Carla P. GOMES, éditeurs, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1037–1042. AAAI Press, 2008.

[GRABISCH & LABREUCHE 2008] Michel GRABISCH et Christophe LABREUCHE. « A decade of application of the Choquet and Sugeno integrals in multi-criteria decision aid ». *4OR*, 6(1) :1–44, 2008.

[HAKEN 1985] Armin HAKEN. « The intractability of resolution ». *Theoretical Computer Science*, 39(0) :297 – 308, 1985.

[HAMADI *et al.* 2009] Youssef HAMADI, Said JABBOUR, et Lakhdar SAIS. « LySAT : solver description ». *SAT Competition*, pages 23–24, 2009.

[HAO & DORNE 1994] Jin-Kao HAO et Raphaël DORNE. « An Empirical Comparison of Two Evolutionary Methods for Satisfiability Problems ». Dans *First International Conference on Evolutionary Computation*, pages 451–455, 1994.

[HASSOUN 2006] Soha HASSOUN, éditeur. *2006 International Conference on Computer-Aided Design (ICCAD'06), November 5-9, 2006, San Jose, CA, USA*. ACM, 2006.

[HENTENRYCK 2002] Pascal Van HENTENRYCK, éditeur. *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 de *Lecture Notes in Computer Science*. Springer, 2002.

[HOLLAND 1975] John H HOLLAND. *Adaptation in natural and artificial systems : An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[HÖLLDOBLER & NGUYEN 2013] Steffen HÖLLDOBLER et Van Hau NGUYEN. « On SAT-Encodings of the At-Most-One Constraint ». Dans George KATSIRELOS et Claude-Guy QUIMPER, éditeurs, *Proc. The Twelfth International Workshop on Constraint Modelling and Reformulation, Uppsala, Sweden, September 16-20*, pages 1–17, 2013.

[HOOKER & VINAY 1995] John N. HOOKER et V. VINAY. « Branching Rules for Satisfiability ». *J. Autom. Reasoning*, 15(3) :359–383, 1995.

[HOOKER 1988] J.N. HOOKER. « Generalized resolution and cutting planes ». *Annals of Operations Research*, 12(1) :217–239, 1988.

[HOOKER 1993] John N. HOOKER. « Solving the incremental satisfiability problem ». *J. Log. Program.*, 15(1&2) :177–186, 1993.

[HOOS & MITCHELL 2005] Holger H. HOOS et David G. MITCHELL, éditeurs. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 de *Lecture Notes in Computer Science*. Springer, 2005.

[HOOS & STÜTZLE 1999] Holger H. HOOS et Thomas STÜTZLE. « Towards a Characterisation of the Behaviour of Stochastic Local Search Algorithms for SAT ». *Artif. Intell.*, 112(1-2) :213–232, 1999.

[HUANG 2007] Jinbo HUANG. « The Effect of Restarts on the Efficiency of Clause Learning ». Dans Veloso [VELOSO 2007], pages 2318–2323.

[HUTTER *et al.* 2002] Frank HUTTER, Dave A. D. TOMPKINS, et Holger H. HOOS. « Scaling and Probabilistic Smoothing : Efficient Dynamic Local Search for SAT ». Dans Hentenryck [HENTENRYCK 2002], pages 233–248.

[JEROSLOW & WANG 1990] Robert G. JEROSLOW et Jinchang WANG. « Solving Propositional Satisfiability Problems ». *Ann. Math. Artif. Intell.*, 1 :167–187, 1990.

[JIN & HAO 2015] Yan JIN et Jin-Kao HAO. « General swap-based multiple neighborhood tabu search for the maximum independent set problem ». *Eng. Appl. of AI*, 37 :20–33, 2015.

[JONG & SPEARS 1989] Kenneth A. De JONG et William M. SPEARS. « Using Genetic Algorithms to Solve NP-Complete Problems ». Dans Schaffer [SCHAFFER 1989], pages 124–132.

[KADIOGLU & SELLMANN 2008] Serdar KADIOGLU et Meinolf SELLMANN. « Efficient Context-Free Grammar Constraints ». Dans Fox et Gomes [FOX & GOMES 2008], pages 310–316.

[KARP 1972] Richard M. KARP. « Reducibility Among Combinatorial Problems ». Dans Miller et Thatcher [MILLER & THATCHER 1972], pages 85–103.

[KATSIRELOS *et al.* 2011] George KATSIRELOS, Nina NARODYTSKA, et Toby WALSH. « The weighted Grammar constraint ». *Annals of OR*, 184(1) :179–207, 2011.

[KAUTZ *et al.* 1992] Henry A KAUTZ, Bart SELMAN, et OTHERS. « Planning as Satisfiability. ». Dans *ECAI*, volume 92, pages 359–363, 1992.

[KAUTZ *et al.* 2002] Henry A. KAUTZ, Eric HORVITZ, Yongshao RUAN, Carla P. GOMES, et Bart SELMAN. « Dynamic Restart Policies ». Dans Dechter et Sutton [DECHTER & SUTTON 2002], pages 674–681.

[KIMMIG *et al.* 2012] Angelika KIMMIG, Guy Van den BROECK, et Luc De RAEDT. « Algebraic Model Counting ». *CoRR*, abs/1211.4475, 2012.

[KLIEBER & KWON 2007] W. KLIEBER et G. KWON. « Efficient CNF Encoding for Selecting 1 from N Objects ». Dans *Fourth International Workshop on Constraints in Formal Verification*, 2007.

[KOHLI *et al.* 1994] R. KOHLI, R. KRISHNAMURTI, et P. MIRCHANDANI. « The Minimum Satisfiability Problem ». *SIAM J. Discrete Math.*, 7(2) :275–283, 1994.

[KORICHE *et al.* 2013] Frédéric KORICHE, Jean-Marie LAGNIEZ, Pierre MARQUIS, et Samuel THOMAS. « Knowledge Compilation for Model Counting : Affine Decision Trees ». Dans Rossi [ROSSI 2013], page 947–953.

[KOSHIMURA *et al.* 2012] Miyuki KOSHIMURA, Tong ZHANG, Hiroshi FUJITA, et Ryuzo HASEGAWA. « QMaxSAT : A Partial Max-SAT Solver ». *JSAT*, 8(1/2) :95–100, 2012.

[KOSSEIM & INKPEN 2012] Leila KOSSEIM et Diana INKPEN, éditeurs. *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, volume 7310 de *Lecture Notes in Computer Science*. Springer, 2012.

[KOSTREVA *et al.* 2004] Michael M KOSTREVA, Włodzimierz OGRYCZAK, et Adam WIERZBICKI. « Equitable aggregations and multiple criteria analysis ». *European Journal of Operational Research*, 158(2) :362–377, 2004.

[KROENING 2009] Daniel KROENING. Software Verification. Dans Biere et al. [BIERE *et al.* 2009], pages 505–532.

[KUEGEL 2012] Adrian KUEGEL. « Improved Exact Solver for the Weighted MAX-SAT Problem ». Dans Daniel LE BERRE, éditeur, *POS-10*, volume 8 de *EPiC Series*, pages 15–27. EasyChair, 2012.

[KUIPERS & WEBBER 1997] Benjamin KUIPERS et Bonnie L. WEBBER, éditeurs. *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island*. AAAI Press / The MIT Press, 1997.

[KULLMANN 2009] Oliver KULLMANN, éditeur. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 de *Lecture Notes in Computer Science*. Springer, 2009.

[LANG & MARQUIS 2010] Jérôme LANG et Pierre MARQUIS. « Reasoning under inconsistency : A forgetting-based approach ». *Artif. Intell.*, 174(12-13) :799–823, 2010.

[LAWRENCE 2004] RYAN LAWRENCE. « Efficient Algorithms for Clause-Learning SAT Solvers ». *Simon Fraser University Master's Thesis*, 2004.

[LE BERRE & LONCA 2014] Daniel LE BERRE et Emmanuel LONCA. « Réutiliser ou adapter les prouveurs SAT pour l'optimisation booléenne ». *Revue d'Intelligence Artificielle*, 28(5) :615–636, 2014.

[LE BERRE & PARRAIN 2010] Daniel LE BERRE et Anne PARRAIN. « The Sat4j library, release 2.2 ». *JSAT*, 7(2-3) :59–6, 2010.

[LE BERRE *et al.* 2012] Daniel LE BERRE, Emmanuel LONCA, Pierre MARQUIS, et Anne PARRAIN. « Optimisation multicritère pour la gestion de dépendances logicielles : utilisation de la norme de tchebycheff ». Dans *RFIA 2012 (Reconnaissance des Formes et Intelligence Artificielle)*, pages 978–2, 2012.

[LE BERRE 2001] Daniel LE BERRE. « Exploiting the real power of unit propagation lookahead ». *Electronic Notes in Discrete Mathematics*, 9 :59–80, 2001.

[LE BERRE 2014] Daniel LE BERRE, éditeur. *POS-13. Fourth Pragmatics of SAT workshop, a workshop of the SAT 2013 conference, July 7, 2013, Helsinki, Finland*, volume 29 de *EPiC Series*. EasyChair, 2014.

[LEE 1959] Chang-Yeong LEE. « Representation of Switching Circuits by Binary-Decision Programs ». *Bell System Technical Journal*, 38(4) :985–999, 1959.

[LEWIS & PAPADIMITRIOU 1998] Harry R. LEWIS et Christos H. PAPADIMITRIOU. *Elements of the theory of computation (2. ed.)*. Prentice Hall, 1998.

[LI & ANBULAGAN 1997a] Chu Min LI et ANBULAGAN. « Heuristics Based on Unit Propagation for Satisfiability Problems ». Dans Polack [POLACK 1997], pages 366–371.

[LI & ANBULAGAN 1997b] Chu Min LI et ANBULAGAN. « Look-Ahead Versus Look-Back for Satisfiability Problems ». Dans Smolka [SMOLKA 1997], pages 341–355.

[LI & LI 2012] Chu Min LI et Yu LI. Satisfying versus falsifying in local search for satisfiability. Dans *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 477–478. Springer, 2012.

[LI & MANYÀ 2009] Chu Min LI et Felip MANYÀ. MaxSAT, Hard and Soft Constraints. Dans Biere et al. [BIERE *et al.* 2009], pages 613–631.

[LI *et al.* 2012] Chumin LI, Chong HUANG, et Ruchu XU. « Balance between intensification and diversification : two sides of the same coin ». *SAT Competition 2013*, page 10, 2012.

[LIBERATORE 2000] Paolo LIBERATORE. « On the complexity of choosing the branching literal in DPLL ». *Artif. Intell.*, 116(1-2) :315–326, 2000.

[LITTMAN *et al.* 2001] Michael L LITTMAN, Stephen M MAJERCIK, et Toniann PITASSI. « Stochastic boolean satisfiability ». *Journal of Automated Reasoning*, 27(3) :251–296, 2001.

[LOOMES & SUGDEN 1982] Graham LOOMES et Robert SUGDEN. « Regret theory : An alternative theory of rational choice under uncertainty ». *The Economic Journal*, pages 805–824, 1982.

[LUBY *et al.* 1993] Michael LUBY, Alistair SINCLAIR, et David ZUCKERMAN. « Optimal Speedup of Las Vegas Algorithms ». *Inf. Process. Lett.*, 47(4) :173–180, 1993.

[LUCE 1956] R Duncan LUCE. « Semiorders and a theory of utility discrimination ». *Econometrica, Journal of the Econometric Society*, pages 178–191, 1956.

[LUSS 1999] Hanan LUSS. « On equitable resource allocation problems : A lexicographic minimax approach ». *Operations Research*, 47(3) :361–378, 1999.

[LYNCE & TREINEN 2010] Inês LYNCE et Ralf TREINEN, éditeurs. *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010*, volume 29 de *EPTCS*, 2010.

[MANCINELLI *et al.* 2006] Fabio MANCINELLI, Jaap BOENDER, Roberto Di COSMO, Jerome VOUILLON, Berke DURAK, Xavier LEROY, et Ralf TREINEN. « Managing the Complexity of Large Free and Open Source Package-Based Software Distributions ». Dans *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 199–208, 2006.

[MANQUINHO & SILVA 2000] Vasco M. MANQUINHO et João P. Marques SILVA. « On Solving Boolean Optimization with Satisfiability-Based Algorithms ». Dans *AMAI*, 2000.

[MANQUINHO *et al.* 1997] Vasco M. MANQUINHO, Paulo F. FLORES, João P. Marques SILVA, et Arlindo L. OLIVEIRA. « Prime Implicant Computation Using Satisfiability Algorithms ». Dans *9th International Conference on Tools with Artificial Intelligence, ICTAI '97, Newport Beach, CA, USA, November 3-8, 1997*, pages 232–239, 1997.

[MANTHEY *et al.* 2012] Norbert MANTHEY, Marijn HEULE, et Armin BIERE. « Automated Reencoding of Boolean Formulas ». Dans Biere et al. [BIERE *et al.* 2013], pages 102–117.

[MANTHEY 2012] Norbert MANTHEY. « Solver Description of RISS 2.0 and PRISS 2.0 ». *SAT Challenge 2012*, page 48, 2012.

[MARLER & ARORA 2004] R Timothy MARLER et Jasbir S ARORA. « Survey of multi-objective optimization methods for engineering ». *Structural and multidisciplinary optimization*, 26(6) :369–395, 2004.

[MARQUES-SILVA & GLASS 1999] João MARQUES-SILVA et Thomas GLASS. « Combinational equivalence checking using satisfiability and recursive learning ». Dans *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 145–149. IEEE, 1999.

[MARQUES-SILVA & SAKALLAH 2007] João MARQUES-SILVA et Karem A. SAKALLAH, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 de *Lecture Notes in Computer Science*. Springer, 2007.

[MARQUIS 2000] Pierre MARQUIS. Consequence finding algorithms. Dans *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 5, pages 41–145. Springer, 2000.

[MARTINS *et al.* 2012] Ruben MARTINS, Vasco M. MANQUINHO, et Inês LYNCE. « Parallel search for maximum satisfiability ». *AI Commun.*, 25(2) :75–95, 2012.

[MARTINS *et al.* 2014] Ruben MARTINS, Saurabh JOSHI, Vasco M. MANQUINHO, et Inês LYNCE. « Incremental Cardinality Constraints for MaxSAT ». Dans O'Sullivan [O'SULLIVAN 2014], pages 531–548.

[MAZURE *et al.* 1997] Bertrand MAZURE, Lakhdar SAIS, et Éric GRÉGOIRE. « Tabu Search for SAT ». Dans Kuipers et Webber [KUIPERS & WEBBER 1997], pages 281–285.

[MCALLESTER *et al.* 1997] David A. MCALLESTER, Bart SELMAN, et Henry A. KAUTZ. « Evidence for Invariants in Local Search ». Dans Kuipers et Webber [KUIPERS & WEBBER 1997], pages 321–326.

[MCALLESTER 2000] David A. MCALLESTER, éditeur. *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 de *Lecture Notes in Computer Science*. Springer, 2000.

[MCGUINNESS & FERGUSON 2004] Deborah L. MCGUINNESS et George FERGUSON, éditeurs. *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*. AAAI Press / The MIT Press, 2004.

[MERCHEZ *et al.* 2001] Sylvain MERCHEZ, Christophe LECOUTRE, et Frédéric BOUSSEMART. « AbsCon : A Prototype to Solve CSPs with Abstraction ». Dans Walsh [WALSH 2001], pages 730–744.

[MIKSA & NORDSTRÖM 2014] Mladen MIKSA et Jakob NORDSTRÖM. « Long Proofs of (Seemingly) Simple Formulas ». Dans Sinz et Egly [SINZ & EGLY 2014], pages 121–137.

[MILANO 2012] Michela MILANO, éditeur. *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 de *Lecture Notes in Computer Science*. Springer, 2012.

[MILLER & THATCHER 1972] Raymond E. MILLER et James W. THATCHER, éditeurs. *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York*, The IBM Research Symposia Series. Plenum Press, New York, 1972.

[MINOUX 1988] Michel MINOUX. « LTUR : A simplified linear-time unit resolution algorithm for horn formulae and computer implementation ». *Information Processing Letters*, 29(1) :1–12, 1988.

[MIZUMOTO 1989] Masaharu MIZUMOTO. « Pictorial representations of fuzzy connectives, part I : cases of t-norms, t-conorms and averaging operators ». *Fuzzy Sets and Systems*, 31(2) :217–242, 1989.

[MORGADO & SILVA 2005] António MORGADO et João P. Marques SILVA. « Good Learning and Implicit Model Enumeration ». Dans *17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2005), 14-16 November 2005, Hong Kong, China*, pages 131–136, 2005.

[MORGADO *et al.* 2012] António MORGADO, Federico HERAS, et João MARQUES-SILVA. « Improvements to Core-Guided Binary Search for MaxSAT ». Dans Cimatti et Sebastiani [CIMATTI & SEBASTIANI 2012], pages 284–297.

[MORGADO *et al.* 2013] António MORGADO, Federico HERAS, Mark H. LIFFITON, Jordi PLANES, et Joao MARQUES-SILVA. « Iterative and core-guided MaxSAT solving : A survey and assessment ». *Constraints*, 18(4) :478–534, 2013.

[MORRIS 1993] Paul MORRIS. « The Breakout Method for Escaping from Local Minima ». Dans Fikes et Lehnert [FIKES & LEHNERT 1993], pages 40–45.

[MOSKEWICZ *et al.* 2001] Matthew W MOSKEWICZ, Conor F MADIGAN, Ying ZHAO, Lintao ZHANG, et Sharad MALIK. « Chaff : Engineering an efficient SAT solver ». Dans *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[MUISE *et al.* 2012] Christian J. MUISE, Sheila A. MCILRAITH, J. Christopher BECK, et Eric I. HSU. « Dsharp : Fast d-DNNF Compilation with sharpSAT ». Dans Kosseim et Inkpen [KOSSEIM & INKPEN 2012], pages 356–361.

[MUROFUSHI & SUGENO 1991] Toshiaki MUROFUSHI et Michio SUGENO. « A theory of fuzzy measures : representations, the Choquet integral, and null sets ». *Journal of Mathematical Analysis and Applications*, 159(2) :532–549, 1991.

[OHRIMENKO *et al.* 2009] Olga OHRIMENKO, Peter J. STUCKEY, et Michael CODISH. « Propagation via lazy clause generation ». *Constraints*, 14(3) :357–391, 2009.

[ORLIN 2009] James B ORLIN. « A faster strongly polynomial time algorithm for submodular function minimization ». *Mathematical Programming*, 118(2) :237–251, 2009.

[O'SULLIVAN 2014] Barry O'SULLIVAN, éditeur. *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 de *Lecture Notes in Computer Science*. Springer, 2014.

[PAPADIMITRIOU & YANNAKAKIS 1984] Christos H PAPADIMITRIOU et Mihalis YANNAKAKIS. « The complexity of facets (and some facets of complexity) ». *Journal of Computer and System Sciences*, 28(2) :244–259, 1984.

[PAPADIMITRIOU 1994] Christos H. PAPADIMITRIOU. *Computational complexity*. Addison-Wesley, 1994.

[PERNY 1992] P PERNY. « Sur le non respect de l'axiome d'indépendance dans les méthodes de type Electre ». *Cahiers du Centre d'études de recherche opérationnelle*, 34(2-4) :211–232, 1992.

[PHILLIPS 1987] Nancy V PHILLIPS. « A weighting function for pre-emptive multicriteria assignment problems ». *Journal of the Operational Research Society*, pages 797–802, 1987.

[PILEGGI & KUEHLMANN 2002] Lawrence T. PILEGGI et Andreas KUEHLMANN, éditeurs. *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, 2002, San Jose, California, USA, November 10-14, 2002*. ACM, 2002.

[PIPATSRISAWAT & DARWICHE 2007] Knot PIPATSRISAWAT et Adnan DARWICHE. « A Lightweight Component Caching Scheme for Satisfiability Solvers ». Dans Marques-Silva et Sakallah [MARQUES-SILVA & SAKALLAH 2007], pages 294–299.

[PIPATSRISAWAT & DARWICHE 2008] Knot PIPATSRISAWAT et Adnan DARWICHE. « New Compilation Languages Based on Structured Decomposability ». Dans Fox et Gomes [FOX & GOMES 2008], pages 517–522.

[PIPATSRISAWAT & DARWICHE 2011] Knot PIPATSRISAWAT et Adnan DARWICHE. « On the power of clause-learning SAT solvers as resolution engines ». *Artif. Intell.*, 175(2) :512–525, 2011.

[PIRLOT & VINCKE 1997] Marc PIRLOT et Philippe VINCKE. *Semiorders - Properties, Representations, Applications*, volume 36 de *Theory and decision library : series B*. Springer, 1997.

[POLACK 1997] Martha POLACK, éditeur. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, 1997.

[PRESTWICH & LYNCE 2006] Steven David PRESTWICH et Inês LYNCE. « Local Search for Unsatisfiability ». Dans Biere et Gomes [BIERE & GOMES 2006], pages 283–296.

[QUINE 1950] WV QUINE. *Methods of logic*. Harvard University Press, 1950.

[RAS & SKOWRON 1999] Zbigniew W. RAS et Andrzej SKOWRON, éditeurs. *Foundations of Intelligent Systems, 11th International Symposium, ISMIS '99, Warsaw, Poland, June 8-11, 1999, Proceedings*, volume 1609 de *Lecture Notes in Computer Science*. Springer, 1999.

[RAUZY 1995] Antoine RAUZY. « Polynomial restrictions of SAT : What can be done with an efficient implementation of the Davis and Putnam's procedure ? ». Dans *Principles and Practice of Constraint Programming—CP'95*, pages 515–532. Springer, 1995.

[RINTANEN 2009] Jussi RINTANEN. Planning and SAT. Dans Biere et al. [BIERE *et al.* 2009], pages 483–504.

[RIVEST *et al.* 1983] R.L. RIVEST, A. SHAMIR, et L.M. ADLEMAN. « Cryptographic communications system and method », septembre 20 1983. US Patent 4405829 A.

[ROBINSON 1965] John Alan ROBINSON. « A Machine-Oriented Logic Based on the Resolution Principle ». *J. ACM*, 12(1) :23–41, 1965.

[ROSENBERG 1975] IG ROSENBERG. « Reduction of bivalent maximization to the quadratic case ». *Cahiers du Centre d'études de Recherche Operationnelle*, 17 :71–74, 1975.

[ROSSI 2003] Francesca ROSSI, éditeur. *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 de *Lecture Notes in Computer Science*. Springer, 2003.

[ROSSI 2013] Francesca ROSSI, éditeur. *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

[ROUSSEL & MANQUINHO 2009] Olivier ROUSSEL et Vasco M. MANQUINHO. Pseudo-Boolean and Cardinality Constraints. Dans Biere et al. [BIERE *et al.* 2009], pages 695–733.

[RYAN 2004] Lawrence RYAN. « *Efficient algorithms for clause-learning SAT solvers* ». PhD thesis, Simon Fraser University, 2004.

[SAKALLAH & SIMON 2011] Karem A. SAKALLAH et Laurent SIMON, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 de *Lecture Notes in Computer Science*. Springer, 2011.

[SANG *et al.* 2005] Tian SANG, Paul BEAME, et Henry A. KAUTZ. « Performing Bayesian Inference by Weighted Model Counting ». Dans Veloso et Kambhampati [VELOSO & KAMBHAMPATI 2005], pages 475–482.

[SANNER & MCALLESTER 2005] Scott SANNER et David A. MCALLESTER. « Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference ». Dans *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1384–1390, 2005.

[SAVAGE 1951] Leonard J SAVAGE. « The theory of statistical decision ». *Journal of the American Statistical association*, 46(253) :55–67, 1951.

[SCHAFFER 1989] J. David SCHAFFER, éditeur. *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*. Morgan Kaufmann, 1989.

[SCHULTE 2013] Christian SCHULTE, éditeur. *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 de *Lecture Notes in Computer Science*. Springer, 2013.

[SEGURA *et al.* 2010] Sergio SEGURA, Robert M. HIERONS, David BENAVIDES, et Antonio Ruiz CORTÉS. « Automated Test Data Generation on the Analyses of Feature Models : A Metamorphic Testing Approach ». Dans *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 35–44. IEEE Computer Society, 2010.

[SEGURA *et al.* 2011] Sergio SEGURA, Robert M. HIERONS, David BENAVIDES, et Antonio Ruiz CORTÉS. « Automated metamorphic testing on the analyses of feature models ». *Information & Software Technology*, 53(3) :245–258, 2011.

[SEGURA *et al.* 2015] Sergio SEGURA, Amador DURÁN, Ana B. SÁNCHEZ, Daniel Le BERRE, Emmanuel LONCA, et Antonio Ruiz CORTÉS. « Automated metamorphic testing of variability analysis tools ». *Softw. Test., Verif. Reliab.*, 25(2) :138–163, 2015.

[SELMAN & KAUTZ 1993] Bart SELMAN et Henry A. KAUTZ. « Domain-Independent Extensions to GSAT : Solving Large Structured Satisfiability Problems ». Dans Bajcsy [BAJCSY 1993], pages 290–295.

[SHANNON 1949] Claude SHANNON. « The Synthesis of Two-Terminal Switching Circuits ». *Bell System Technical Journal*, 28(1) :59–98, 1949.

[SHEINI & SAKALLAH 2006] Hossein M SHEINI et Karem A SAKALLAH. « Pueblo : A hybrid pseudo-boolean SAT solver ». *Journal on Satisfiability, Boolean Modeling and Computation*, 2 :165–189, 2006.

[SILVA & SAKALLAH 1996] João P. Marques SILVA et Karem A. SAKALLAH. « GRASP - a new search algorithm for satisfiability ». Dans *ICCAD*, pages 220–227, 1996.

[SILVA & SAKALLAH 1999] João P. Marques SILVA et Karem A. SAKALLAH. « GRASP : A Search Algorithm for Propositional Satisfiability ». *IEEE Trans. Computers*, 48(5) :506–521, 1999.

[SILVA *et al.* 2009] João P. Marques SILVA, Inês LYNCE, et Sharad MALIK. Conflict-Driven Clause Learning SAT Solvers. Dans Biere et al. [BIERE *et al.* 2009], pages 131–153.

[SIMONIS 2014] Helmut SIMONIS, éditeur. *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 de *Lecture Notes in Computer Science*. Springer, 2014.

[SINGER *et al.* 2000] Josh SINGER, Ian P. GENT, et Alan SMAILL. « Backbone Fragility and the Local Search Cost Peak ». *J. Artif. Intell. Res. (JAIR)*, 12 :235–270, 2000.

[SINZ & EGLY 2014] Carsten SINZ et Uwe EGLY, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 de *Lecture Notes in Computer Science*. Springer, 2014.

[SINZ 2005] Carsten SINZ. « Towards an Optimal CNF Encoding of Boolean Cardinality Constraints ». Dans van Beek [VAN BEEK 2005], pages 827–831.

[SMOLKA 1997] Gert SMOLKA, éditeur. *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, volume 1330 de *Lecture Notes in Computer Science*. Springer, 1997.

[SÖRENSSON & BIERE 2009] Niklas SÖRENSSON et Armin BIERE. « Minimizing Learned Clauses ». Dans Kullmann [KULLMANN 2009], pages 237–243.

[SÖRENSSON & EÉN 2009] Niklas SÖRENSSON et Niklas EÉN. « MINISAT 2.1 and MINISAT++ 1.0—SAT race 2008 editions ». *SAT-Race 2010 : Solver Descriptions*, page 31, 2009.

[SPENCE 2010] Ivor SPENCE. « sgen1 : A generator of small but difficult satisfiability benchmarks ». *ACM Journal of Experimental Algorithmics*, 15, 2010.

[STEUER & CHOO 1983] Ralph E STEUER et Eng-Ung CHOO. « An interactive weighted Tchebycheff procedure for multiple objective programming ». *Mathematical Programming*, 26(3) :326–344, 1983.

[STOCKMEYER 1976] Larry J STOCKMEYER. « The polynomial-time hierarchy ». *Theoretical Computer Science*, 3(1) :1–22, 1976.

[STRICHMAN & SZEIDER 2010] Ofer STRICHMAN et Stefan SZEIDER, éditeurs. *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 de *Lecture Notes in Computer Science*. Springer, 2010.

[SUGENO 1974] Michio SUGENO. *Theory of fuzzy integrals and its applications*. Tokyo Institute of Technology, 1974.

[SWARTOUT 1992] William R. SWARTOUT, éditeur. *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992*. AAAI Press / The MIT Press, 1992.

[SYRJÄNEN 1999] Tommi SYRJÄNEN. « A rule-based formal model for software configuration ». Rapport technique, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, 1999.

[TREINEN & ZACCHIROLI 2009] Ralf TREINEN et Stefano ZACCHIROLI. « Common upgradeability description format (CUDF) 2.0 ». *The Mancoosi project (FP7)*, 3, 2009.

[TSEITIN 1968] G.S. TSEITIN. « On the complexity of derivations in the propositional calculus ». Dans H.A.O. SLESENKO, éditeur, *Structures in Constructives Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

[TURING 1936] Alan Mathison TURING. « On computable numbers, with an application to the Entscheidungsproblem ». *J. of Math*, 58 :345–363, 1936.

[TURING 1939] Alan Mathison TURING. « Systems of logic based on ordinals ». *Proceedings of the London Mathematical Society*, 2(1) :161–228, 1939.

[UCKELMAN *et al.* 2009] Joel UCKELMAN, Yann CHEVALEYRE, Ulle ENDRISS, et Jérôme LANG. « Representing Utility Functions via Weighted Goals ». *Math. Log. Q.*, 55(4) :341–361, 2009.

[VAN BEEK 2005] Peter van BEEK, éditeur. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 de *Lecture Notes in Computer Science*. Springer, 2005.

[VAN DER TAK *et al.* 2011] Peter van der TAK, Antonio RAMOS, et Marijn HEULE. « Reusing the Assignment Trail in CDCL Solvers ». *JSAT*, 7(4) :133–138, 2011.

[VAN GELDER & SPENCE 2010] Allen VAN GELDER et Ivor SPENCE. « Zero-One Designs Produce Small Hard SAT Instances ». Dans Strichman et Szeider [STRICHMAN & SZEIDER 2010], pages 388–397.

[VAN GELDER 2002] Allen VAN GELDER. « Generalizations of Watched Literals for Backtracking Search ». Dans *AMAI*, 2002.

[VAN HARMELEN 2002] Frank van HARMELEN, éditeur. *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*. IOS Press, 2002.

[VAN LAMBALGEN 2006] Martijn van LAMBALGEN. « 3MCard 3MCard A Lookahead Cardinality Solver ». Master's thesis, Delft University of Technology, 2006.

[VEANES & VIGANÒ 2013] Margus VEANES et Luca VIGANÒ, éditeurs. *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 de *Lecture Notes in Computer Science*. Springer, 2013.

[VELOSO & KAMBHAMPATI 2005] Manuela M. VELOSO et Subbarao KAMBHAMPATI, éditeurs. *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. AAAI Press / The MIT Press, 2005.

[VELOSO 2007] Manuela M. VELOSO, éditeur. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.

[VORONKOV 1993] Andrei VORONKOV, éditeur. *Logic Programming and Automated Reasoning,4th International Conference, LPAR'93, St. Petersburg, Russia, July 13-20, 1993, Proceedings*, volume 698 de *Lecture Notes in Computer Science*. Springer, 1993.

[VORONKOV 2002] Andrei VORONKOV, éditeur. *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 de *Lecture Notes in Computer Science*. Springer, 2002.

[WALSH 1999] Toby WALSH. « Search in a Small World ». Dans Dean [DEAN 1999], pages 1172–1177.

[WALSH 2000] Toby WALSH. « SAT v CSP ». Dans Dechter [DECHTER 2000], pages 441–456.

[WALSH 2001] Toby WALSH, éditeur. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 de *Lecture Notes in Computer Science*. Springer, 2001.

[WANG *et al.* 2014] Yang WANG, Jin-Kao HAO, Fred GLOVER, et Zhipeng LÜ. « A tabu search based memetic algorithm for the maximum diversity problem ». *Eng. Appl. of AI*, 27 :103–114, 2014.

[WEAVER 2012] Sean WEAVER. « *Satisfiability Advancements Enabled by State Machines* ». PhD thesis, University of Cincinnati, 2012.

[WEYUKER 1982] Elaine J. WEYUKER. « On Testing Non-Testable Programs ». *Comput. J.*, 25(4) :465–470, 1982.

[WHITTEMORE *et al.* 2001] Jesse WHITTEMORE, Joonyoung KIM, et Karem A. SAKALLAH. « SATIRE : A New Incremental Satisfiability Engine ». Dans *Proceedings of the 38th annual Design Automation Conference*, pages 542–545, 2001.

[WIERZBICKI 1986] Andrzej P WIERZBICKI. « On the completeness and constructiveness of parametric characterizations to vector optimization problems ». *Operations-Research-Spektrum*, 8(2) :73–87, 1986.

[WILSON 2005] Nic WILSON. « Decision Diagrams for the Computation of Semiring Valuations ». Dans *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 331–336, 2005.

[YAGER 1988] Ronald R YAGER. « On ordered weighted averaging aggregation operators in multicriteria decisionmaking ». *Systems, Man and Cybernetics, IEEE Transactions on*, 18(1) :183–190, 1988.

[ZHANG & MALIK 2002] Lintao ZHANG et Sharad MALIK. « The Quest for Efficient Boolean Satisfiability Solvers ». Dans Voronkov [VORONKOV 2002], pages 295–313.

[ZHANG & STICKEL 2000] Hantao ZHANG et Mark E. STICKEL. « Implementing the Davis-Putnam Method ». *J. Autom. Reasoning*, 24(1/2) :277–296, 2000.

[ZHANG 1997] Hantao ZHANG. SATO : An efficient prepositional prover. Dans *Automated Deduction—CADE-14*, pages 272–275. Springer, 1997.

Résumé

L'aide à la décision a pour but d'assister un opérateur humain dans ses choix. La nécessité d'employer de telles techniques s'est imposée avec la volonté de traiter des problèmes dépendant d'une quantité de données toujours plus importante.

L'intérêt de l'aide à la décision est encore plus manifeste lorsqu'on souhaite obtenir non pas une solution quelconque, mais une des *meilleures* solutions d'un problème combinatoire selon un *critère* donné. On passe alors d'un *problème de décision* (déterminer l'existence d'une solution) à un *problème d'optimisation monocritère* (déterminer une des meilleures solutions possibles selon un critère). Un décideur peut aussi souhaiter considérer plusieurs critères, et ainsi faire passer le problème de décision initial à un problème d'*optimisation multicritère*. La difficulté de ce type de problèmes provient du fait que les critères considérés sont généralement antagonistes, et qu'il n'existe donc pas de solution meilleure que les autres pour l'ensemble des critères. Dans ce cas, il s'agit plutôt de déterminer une solution qui offre un bon *compromis* entre les critères.

Dans cette thèse, nous étudions dans un premier temps la complexité théorique de tâches d'optimisation complexes sur les langages de la logique propositionnelle, puis nous étudions leur résolution pratique *via* le recours à des logiciels bâtis pour résoudre de manière efficace en pratique des problèmes de décision combinatoires, les prouveurs SAT.

Mots-clés: aide à la décision, optimisation multicritère, SAT

Abstract

Decision aiding aims at helping a decision-maker to pick up a solution among several others. The usefulness of such approaches is as prominent as the size of the problems under consideration increases.

The need of decision aiding techniques is salient when the problem does not just consist in deciding whether a solutions exists, but to find one of the *best* solutions according to a given *criterion*. In this case, the problem goes from a *decision problem* (decide whether a solution exists) to a *single criterion optimization problem* (find one of the best solutions according to a criterion). A decision-maker may even want to consider multiple criteria, which turns the initial decision problem into a *multicriteria optimization problem*. The main issue arising in such cases lies in the fact that the criteria under consideration are often antagonistic, which implies that there is no solution which is the best for each of the objectives. In this case, a good *compromise* solution is looked for.

In this thesis, we first focus on the complexity of optimization requests based on a set of propositional languages. We then study the practical aspects of the resolution of such problems, using pieces of software designed for dealing with combinatorial decision problems, namely SAT solvers.

Keywords: Decision aid, MCDA, SAT