REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE ABDERAHMANE MIRA DE BEJAIA
FACULTE DES SCIENCES EXACTES

DEPARTEMENT D'INFORMATIQUE
ECOLE DOCTORALE RESEAUX ET SYSTEMES DISTRIBUES

Mémoire de Magistère

En vue de l'obtention du diplôme de magistère en informatique

Option : Réseaux et Systèmes Distribués

Thème

Représentations de formules SAT pour la Résolution Séquentielle

Présenté par :

Fatima BOUDJERIDA

Devant le jury composé de :

Président :	Kerkar	Moussa	Professeur.	Université de Bejaïa, Algérie.
Rapporteur :	HABBAS	Zineb	M. C- HDR.	Université de Metz, France.
Examinateur :	MELIT	Ali	M. C.	Université de Jijel, Algérie.
Examinateur :	BOUKERRAM	Abdellah	M. C.	Université de Sétif, Algérie.

Promotion 2006-2007

Résumé

La simplicité du formalisme de la logique propositionnelle a considérablement contribué aux succès courants de la résolution des problèmes SAT. D'une part, la simplicité du formalisme facilite l'implémentation de solveurs efficaces par des structures de données simples. D'autre part, plusieurs problèmes NP-complet ont des encodages linéaires de petite taille comme la forme normale de négation décomposable (DNNF), qui est fortement approprié pour des objectifs pratiques.

Dans ce présent travail, nous avons analysé les formalismes les plus connu dans le cadre de la résolution séquentielle. Dans ces formalismes, les solveurs a pour but de résoudre les différents types de problèmes théoriques et industriels réputés dans la communauté SAT dans un temps polynomial. Nous avons étudié les trois formalismes: CNF, OBDD et DNNF. La première est le plus ancien et le plus utilisé dans la littérature comme Rel-SAT, Grasp, SATO, Chaff, BerkMin, Siege, Minisat,….mais la dernière DNNF est plus efficace. Pour faire face à ce formalisme nous avons proposé une résolution basée sur un formalisme hybride (CNF et DNNF). Les résultats obtenus montrent l'efficacité de la solution proposée à compiler les instances des problèmes SAT, encodés en CNF, au d-DNNF et à calculer les solutions de ces instances.

Mots clé: SAT, satisfiabilité propositionnelle, CNF,OBDD, DNNF, résolution séquentielle, DPLL.

Abstract

The simplicity of the formalism of propositional logic contributed considerably to current successes of the resolution of problems SAT. On the one hand, the simplicity of the formalism facilitates the implementation of effective solvers by structures of data simple. In addition, several problems NP-complete have linear encodings of small size as the decomposable negation normal form (DNNF), which is strongly adapted for practical objectives.
In this present work, we analyzed the formalisms most known in the framework of the sequential resolution. In these formalisms, the purpose of the solvers is to solve the various types of theoretical problems and industrialists considered in community SAT in a polynomial time. We studied the three formalisms: CNF, OBDD and DNNF. The first is oldest and more used in the literature like Rel-SAT, Grasp, SATO, Chaff, BerkMin, Siege, Minisat… but the last DNNF is more effective. To prevent from this formalism we proposed a resolution based on a hybrid formalism (CNF and DNNF).The results obtained show the effectiveness of the solution suggested to compile the instants of problems SAT, encodes in CNF, to the d-DNNF and to calculate the solutions of these instants

Keywords: SAT, propositional satisfiability, CNF, OBDD, DNNF, sequential resolution, DPLL.

Dédicaces

Je dédie ce travail à la mémoire de mon père.

Remerciements

Je remercie mon Dieu le tous puissant et miséricordieux qui m'a aidé dans toute ma vie ;

Je tiens aussi à remercie mes encadreurs Pr/ HABBAS Zineb et SINGER Daniel, professeurs à l'université de Metz, de la France, *pour ses précieux conseils et leur soutien;*

Je n'oublie pas de remercier le président du jury, Monsieur le Professeur KERKAR, et les examinateurs : le Pr. BOUKERRAM et le Pr. MELLIT d'avoir accepté d'évaluer et de juger mon travail ;

Mes remerciements vont également à tous les enseignants et enseignantes du département de l'informatique qui nous ont formé et guidé tout au longue de notre cursus, surtout Mr TARI Abdelkamel, Docteur et chef de département de l'informatique de m'avoir aidé, orienté et encouragé.

Ma reconnaissance va particulièrement à tous mes enseignants de l'école doctorale d'Informatique et du département d'Informatique de l'université A/Mira de Béjaia.

Et un merci tout particulier à tous ceux qui m'ont apporté leur soutien.

Tables des matières

Chapitre IV :Compile & SAT : une nouvelle méthode pour la résolution séquentielle du SAT

Liste des figures

Liste des tableaux

Introduction

Le problème SAT, qui consiste à vérifier si un ensemble de clauses est satisfaisable ou pas, est central en informatique et dans le domaine de l'intelligence artificielle. Il est notamment utilisé pour la preuve de théorème, la planification, le raisonnement et la vérification de circuits.

Durant ces dernières décennies, plusieurs approches ont été proposées pour résoudre des instances difficiles du SAT en utilisant des algorithmes complets ou incomplets. Les algorithmes incomplets sont basés la recherche locale (voir [Selman et al, 1992]). La plupart des solveurs complets sont basés sur la technique du backtrack initiée par Davis Putnam Logemann Loveland (DPLL). Ces algorithmes de base intègrent des techniques de plus en plus efficaces comme l'apprentissage, la propagation de contraintes, le pré-traitement, l'exploitation des symétries, etc. L'impact de ces différentes améliorations dépend du type des instances à résoudre. Par exemple, l'apprentissage est plus efficace sur les instances issues du monde réel que sur les instances aléatoires.

Par ailleurs, malgré l'évolution de la puissance des machines, il apparaît que les besoins industriels (VLSI, Planification, Model Checking, ...), et fondamentaux deviennent croissants tant en nombre de problèmes à résoudre qu'en taille des formules à traiter. Les problèmes provenant de l'industrie ont généralement une description booléenne très volumineuse pouvant aller jusqu'à plusieurs milliers de variables. Leur difficulté de résolution réside, outre dans leur nature difficile propre, mais aussi dans ce volume de données à traiter. Ainsi, une limite matérielle et logicielle des traitements commence à se faire sentir avec les solveurs SAT basés sur le codage traditionnel sous forme normale conjonctive (CNF). Le codage sous forme CNF induit une perte des connaissances structurelles qui sont mieux exprimées dans d'autres formalismes et qui peuvent permettre une résolution plus efficace [A.Darwiche ,2001]. Pour exploiter ces connaissances structurelles, des travaux récents ont été proposés. Quelques uns utilisent la forme étendue des formules booléennes (nonCNF [K. L. McMillan,2002]) pour le codage des formules. Tandis que d'autres essayent de récupérer

et/ou déduire des propriétés structurelles à partir des formules CNF (comme les Diagrammes de Décision Binaires Ordonnés (OBDD) et les Formes Decomposables Negations **(**DNNF)). C'est dans ce contexte que se situe le travail que nous allons présenter dans ce mémoire. Le problème que nous avons traité est la comparaison de ces trois propositions dans le cadre d'une Résolution Séquentielle. Différents solveurs *en freeware* devront être expérimentés sur différents types de problèmes théoriques et industriels réputés dans la communauté SAT. Il s'agit donc de faire une présentation théorique de ces trois formalismes et aussi de les expérimenter en pratique sur un certain nombre de benchmarks bien connus.

Notre mémoire est organisé en cinq chapitres. Le premier chapitre traite d'une manière générale la satisfiabilité propositionnelle (SAT) ainsi que les différentes méthodes utilisées pour leur résolution.

Dans le deuxième et le troisième chapitre , nous donnons un état de l'art sur les trois formalismes étudiés qui sont le formalisme CNF, OBDD et DNNF dans le cadre du SAT, ainsi qu'une comparaison entre eux. Suivi par les différentes méthodes de résolution et les travaux importants qui s'inscrivent dans ce contexte .

Nous présentons la méthode hybride proposé dans le quatrième chapitre où nous définissons clairement notre contribution pour l'amélioration de la résolution séquentielle du SAT.

Nous terminons ce mémoire par une conclusion et des perspectives.

Chapitre I

La satisfiabilité propositionnelle (SAT)

Introduction

Un problème de satisfaction de contraintes booléennes, encore appelé problème de satisfiabilité, consiste à décider, étant donné un ensemble de contraintes définies sur des variables booléennes, s'il existe une assignation de valeurs aux variables satisfaisant toutes les contraintes (et éventuellement à déterminer une telle assignation).

Ce chapitre est une introduction générale au domaine de satisfaction des contraintes et plus précisément aux terminologies du SAT.

Définition : Le problème de satisfaction de contraintes booléennes est le problème connu sous le nom de SAT (SATisfaction des problèmes booléennes), consistant à décider si une formule propositionnelle (exprimée comme une conjonction de disjonctions) est satisfaisable ou non.

SAT a été le premier problème montré *NP*-complet par Cook [Cook, 71]. La *NP*-complétude du SAT assure qu'aucun algorithme pour ce problème ne peut être efficace au pire cas, sous l'hypothèse *P≠NP*. Néanmoins, il existe, en pratique, de nombreux algorithmes efficaces pour résoudre le problème

Exemple : Donnée : Une formule booléenne mise sous forme CNF :

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

Question : Est-ce que la formule F admet au moins un modèle?

Réponse: Pour cet exemple, la réponse est oui : l'affectation a$= vrai, b = vrai$ et $c = vrai$ satisfait la formule F. Comme F admet une seule solution, on a $F \wedge (\neg a \vee \neg b \vee \neg c)$ est insatisfiable.

La première méthode qui vient à l'esprit (sans être un spécialiste de SAT) pour décider si une formule F est satisfiable ou non est celle qui consiste à passer en revue les 2^n affectations possibles. Ce qui donne une complexité en temps dans le pire cas de l'ordre de 2^n.

A l'heure actuelle, il n'existe pas de méthode permettant de décider la satisfiabilité de toute formule de type F en temps polynomial en fonction de n. L'existence d'un tel algorithme est peu vraisemblable, même si sa non-existence n'a pas été prouvée. En effet, la découverte fondamentale de Cook au début des années 70[Cook,1971] est que de nombreux problèmes (e.g. En recherche opérationnelle et en théorie des graphes) peuvent être réduis au problème SAT.

Pour « résoudre SAT », on ne peut donc que chercher des heuristiques et de nouveaux raffinements permettant d'accélérer en pratique les traitements ou exhiber des restrictions pour lesquelles on peut garantir une résolution en temps polynomial [SAÏS,2000].

Les méthodes de résolution

Dans les solveurs SAT existants, on distingue généralement deux types de solveurs: les solveurs complets et les solveurs incomplets. Les algorithmes complets répondent à la fois à la satisfiabilité et à l'insatisfiabilité. On les appelle « complets » car si on leur laisse un temps et un espace illimité, ils parcourent complètement l'espace de recherche.

La plus grande majorité des méthodes complètes sont basées sur la procédure énumérative DPLL (Davis, Patnam, Logemann et Loveland) [Davis et Putnam, 1960][Davis et al, 1962] à la quelle ils apportent plusieurs optimisations et simplifications à la procédure SAT comme le backtracking non chronologique, l'apprentissage par l'analyse des conflits, la détection efficace des clauses unitaires etc.

Les méthodes incomplètes sont basées généralement sur la recherche locale (LR) pour SAT (voir [Selman et al, 1992] comme exemple, et [Selman et al. 1994] pour une vue générale). Une méthode de recherche locale n'effectue pas un parcours systématique de l'espace de recherche. Le plus souvent elle choisit une affectation des variables du problème au hasard et ensuite, si la solution n'est pas trouvée, on flippe (c'est à dire qu'on inverse sa valeur) les variables une à une en suivant une stratégie de réparation. L'algorithme s'arrête lorsqu'une solution est trouvée ou lorsque le temps ou l'espace dédié à l'algorithme est dépassé [Steven et al,2006].

La structuration des problèmes SAT

La résolution de problèmes par la satisfaction de contrainte est de nature déclarative, composé de deux parties: la modélisation et la résolution. Le rôle de la modélisation est de décrire les problèmes à résoudre dans un formalisme mathématique (une contrainte ou langage de modélisation), comme CNF (forme normale conjonctive) en décrivant les relations entre les variables du problème afin de poser des contraintes sur leurs affectations des valeurs possibles. Le résultat de la modélisation est une traduction du problème spécifique dans un langage de modélisation.

La logique du premier ordre fournit des formalismes de modélisation pour des problèmes de complexité diverse.

Le formalisme simple de la logique propositionnelle a considérablement contribué aux succès courants de la résolution des problèmes SAT. D'une part, la simplicité du formalisme facilite l'implémentation de solveurs efficaces par des structures de données simples. D'autre part, plusieurs problèmes NP-complet ont des codages linéaires de petite taille comme la forme normale de négation décomposable (DNNF) [A.Darwiche ,2001]), qui est fortement approprié pour des objectifs pratiques.

En résolvant des problèmes avec les solveurs SAT, la formule propositionnelle codant l'exemple original du problème est typiquement traduite en forme normale conjonctive (CNF) [R.COWEN et K.WYATT, 1993] c-à-d. en conjonction des disjonctions des variables booléennes et de leurs négations. Dans ce sens, SAT peuvent être vus comme des cas particulières de problèmes de satisfaction de contrainte (CSPs) [L.PARIS, 2007] [Bennaceur et Li, 2002]. D'une manière générale, CSPs nous permettent d'exprimer des conjonctions des contraintes arbitraires portant sur des variables avec des domaines donnés. Par conséquent, CNF SAT est un cas spécial de CSPs dans lequel on permet seulement des conjonctions des contraintes clausales des variables avec des domaines binaires. Ceci a comme conséquence le fait que, dans beaucoup de cas, les avancées dans des techniques de résolution des problèmes SAT peuvent être appliquées aux CSPs [Toby Walsh,2000].

Alors on peut dire que les techniques appliquées avec succès pour résoudre SAT peuvent être divisées en deux approches idéologiquement différentes: celles basées sur la compilation de la connaissance (comme OBDD et DNNF [Selman & Kautz, 199 ; A.Darwiche ,2001 ; et Bryant 1986]) et les approches basées sur la résolution.

Notre travail est basé sur la compilation des différentes formalismes des formules SAT. En se concentrant sur la satisfiabilité, les problèmes sous ces nouveaux formalismes peuvent être résolus en un temps linéaire, alors qu'ils sont à l'origine exponentiels, ceci est un facteur principal dans le succès des solveurs SAT basé sur la compilation.

Ces dernières années, un nombre important de solveurs capables de traiter des instances de plus en plus grandes ont été proposés [Darwiche et al, 2005 ; L. VonNorden, 2006 ; Singer, 2006]. En plus des méthodes de simplifications liées à la résolution comme l'apprentissage, la propagation de contraintes , de plus en plus de prétraitements y sont intégrés, afin d'identifier et de traiter certaines structures contenues dans les instances. Pour ces instances, il est clair que l'analyse de conflits (apprentissage dans les domaines SAT et CSP) joue un rôle fondamental dans l'implémentation de solveurs SAT efficaces. De plus, d'autres techniques comme la technique de « backtracking » et l'utilisation de structures paresseuses, comme l'implémentation des « watched literals » [Moskewicz et al ,2001] permettent d'améliorer considérablement les performances des solveurs. L'analyse de conflits permet d'une part d'effectuer un backtracking non chronologique et d'autre part d'ajouter des informations dans la base de connaissances dans le but de couper des branches dans l'arbre de recherche.

De nombreuses approches d'analyse de conflits ont été développées dans différents domaines de l'Intelligence Artificielle (IA), notamment dans les systèmes de maintien de cohérence, des problèmes de satisfaction de contraintes (CSP) et en déduction automatique. Dans le cadre SAT, les premiers résultats autour de l'intégration de ces techniques d'apprentissage dans les algorithmes de résolution sont apparus dans les travaux de Silva par la résolution GRASP [J. P. M. Silva and K. A, 96]. Ces techniques se sont révélées efficaces dans la résolution d'instances difficiles issues d'applications réelles encodant d'une certaine manière des connaissances structurelles comme CNF, OBDD et DNNF.

Il sera efficace de tenir compte des propriétés structurelles pour résoudre le problème SAT. En compilation de la connaissance, l'idée fondamentale est de compiler d'abord la formule propositionnelle dans la phase offline. La représentation résultante est utilisée pour résoudre le problème dans la phase Online. Dans le meilleur des cas, les opérations appliquées aux problème original incluent l'énumération des modèles, l'équivalence en plus de la satisfiabilité. Beaucoup de nouvelles représentations ont été proposées, comme la forme

normale de négation décomposable (DNNF) [A.Darwiche ,2001] et les diagrammes de décision binaires ordonnés (OBDDs) [Bryant 1986].

Conclusion

Ce chapitre a servi de prélude à notre travail en nous permet d'aborder les aspects formels liés au concept du problème SAT et sur les approches utilisées pour les résoudre.

Pour plus de détails sur les représentations existantes et les méthodes de résolutions utilisées sont donnés dans les chapitres 3 et 4.

Chapitre II

Les représentations de formules SAT

Introduction

Les recherches basées SAT sont devenues un outil standard pour résoudre divers problèmes réels de taille croissante et difficile, il y a une demande de plus en plus de solveurs plus robustes et plus efficaces. Pour comprendre ce succès des solveurs SAT pour les problèmes structurés, il est important d'étudier comment différents types de propriétés structurales des instances SAT sont liés à l'efficacité de résolution des problèmes en utilisant des techniques de satisfaction de contrainte. C'est le but fondamental de ce mémoire.

Ce chapitre est consacré à la présentation des différentes représentations du problème SAT considérées dans notre travail. Nous allons donner quelques définitions de base relatives aux formules propositionnelles.

1. Définition de base [SAÏS,2000] :

Avant de donner la définition du problème SAT, et de présenter ses variantes, nous allons présenter au début quelques définitions concernant la logique mathématique qui sont nécessaires pour la compréhension et l'utilisation ultérieure :

Définition 1.1 (variable propositionnelle) : Une **variable propositionnelle**, parfois appelée **proposition atomique** ou **atome**, est une variable booléenne prenant ses valeurs dans l'ensemble {FAUX,VRAI}, {0, 1} ou encore {F, V }.

Définition 1.2 (formule propositionnelle) : Soit un alphabet constitué d'un ensemble de variables propositionnelles et de deux symboles particuliers $\top$ et $\bot$. Soient les connecteurs suivants: de négation: $\neg$ (non), de conjonction: $\wedge$ (et), de disjonction: $\vee$ (ou), d'implication: $\rightarrow$(si . . .alors . . .), d'équivalence: $\leftrightarrow$(. . .si et seulement si . . .), et les opérateurs auxiliaires de parenthésage « (» et «) ».

Une formule propositionnelle f se construit récursivement en appliquant les règles suivantes :

1. une variable, $\top$ et $\bot$ sont des formules ;
2. si f est une formule alors (f) est une formule ;
3. si f est une formule alors $\neg f$ est une formule ;
4. si f et f' sont des formules alors : ($f \vee f'$) ; ($f \wedge f'$); ($f \rightarrow f'$); et ($f \leftrightarrow f'$) sont des formules.

Dans le cas où aucune ambiguïté n'est possible, les parenthèses peuvent être omises. Dans la pratique, nous ne considérerons que des alphabets contenant un nombre fini de variables propositionnelles.

Définition 1.3 (littéral) : On désigne par **littéral** une variable l ou sa négation $\neg l$: l est appelé **littéral positif** et $\neg l$ **littéral négatif**. On note $\sim l$ le littéral complémentaire à l.

Définition 1.4 (clause) : On appelle **clause** une formule constituée d'une disjonction finie de littéraux. Une clause est donc satisfaite par une interprétation si au moins un de ses littéraux est vrai dans cette interprétation.

2. Formes normales

Définition 2.1 (DAG, NNF) : Nous considérons des graphiques orientés acycliques (DAG) où chaque nœud interne est étiqueté avec une conjonction (*et*, ∧) ou disjonction (*ou*, ∨), et chaque feuille est étiquetée avec un symbole littéral propositionnel ou constant (*true/false*, ou 1/0). Ce devrait être clair qu'un tel DAG est une propriété qui caractérise la Négation Normal Form (NNF) [Barwise, 1977].

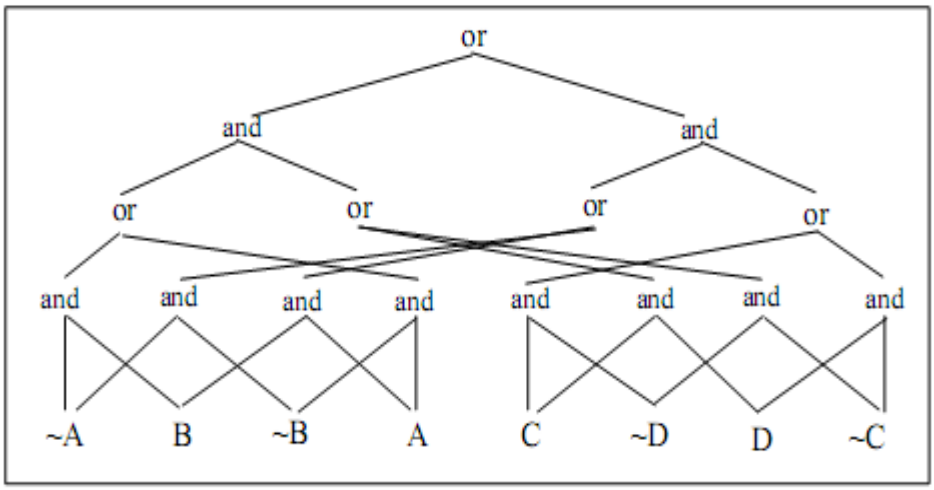

Fig 2.1 : exemple du formalisme NNF

Définition 2.2 (CNF) : Le formalisme populaire CNF (forme normale conjonctive) peut maintenant être définie comme le sous-ensemble de NNF (Négation Normal Form) [Barwise, 1977] qui satisfait :

(i) égalité: la hauteur du DAG (graphiques orientés acycliques) est au plus deux;

(ii) disjonction simple: toute disjonction est seulement sur nœuds feuille (c.-à-d., est une clause).

Définition 2.3 (DNF) : De la même façon, DNF (forme normale disjonctive) est le sous-ensemble de NNF qui satisfait l'égalité et la conjonction simple: toute conjonction est seulement sur nœuds feuille (c.-à-d., est un terme).

Définition 2.4 (littéral pur ou monotone) : Un littéral l est dit **pur** pour la formule f sous forme CNF si et seulement s'il apparaît soit uniquement positivement soit uniquement négativement dans f.

Propriété 2.1 *Toute formule propositionnelle peut être réécrite sous forme normale conjonctive.*

Néanmoins cette transformation peut nécessiter une croissance exponentielle de la taille de l'ensemble obtenu. On peut trouver dans la thèse de Siegel [Siegel, 1987], un algorithme permettant de transformer toute formule f en une formule normale conjonctive équivalente du point de vue de la satisfiabilité dont la taille est linéaire.

Définition 2.5 (CNF « simplifiée ») : Soient une CNF f et un littéral l, nous notons fl la CNF simplifiée en supprimant de f toutes les occurrences de ¬l et toutes les clauses où l apparaît.

Par extension, si c est une clause fondamentale, fc correspond à la simplification de f par tous les littéraux de la clause c , Nous notons $f*$ la CNF simplifiée par la propagation de tous ses littéraux unitaires de f. De même, nous notons $f\diamond$ la CNF simplifiée par la propagation de tous les littéraux purs de f.

Enfin, $f+$ correspond à la CNF simplifiée par la propagation de tous les littéraux unitaires et purs de f. Intuitivement, la simplification d'une formule par un littéral l est une opération élémentaire consistant à exercer les règles d'évaluation d'une formule sur l'interprétation partielle Ip définie uniquement pour la variable l ($Ip(l) = Vrai$ si l est positif, *Faux* sinon). Dans le cas d'une CNF, cette simplification s'effectue en deux étapes:

1. suppression des clauses satisfaites par l, c'est-à-dire les clauses c telles que $l \in c$;
2. suppression des occurrences falsifiées de$l$, c'est-à-dire des occurrences de¬l.

Cette simplification est couramment appelée propagation de l'effet de l'affectation (ou de l'assignation) d'un littéral.

Exemple : *Soit la CNF* Σ = {(a ∨ ¬c), (¬a ∨ b ∨ c), (¬d ∨ ¬a), (¬c ∨ d), (¬b ∨ e ∨ f), (¬e ∨ f ∨ ¬g), (¬f ∨ ¬e)}.

Soit la clause c1 = {a, ¬d}.

fa = {(b ∨ c), (¬d), (¬c ∨ d), (¬b ∨ e ∨ f), (¬e ∨ f ∨ ¬g), (¬f ∨ ¬e)}.

Σc1 = {(b ∨ c), (¬c), (¬b ∨ e ∨ f), (¬e ∨ f ∨ ¬g), (¬f ∨ ¬e)}.

Σ∗c1 = {(e ∨ f), (¬e ∨ f ∨ ¬g), (¬f ∨ ¬e)}.

Σ+c1 = {(e ∨ f), (¬f ∨ ¬e)}.

2.1 Procédure de base pour SAT :

Définition : [Matti,2008]

- Pour une formule CNF F , le graphe d'implication G=(V,E), dans un moment d'exécution de DPLL avec l'ensemble de décision des littéraux D, est un graphe orienté. L'ensemble des nœuds est

$$V = \{\Lambda\} \cup D \cup \{l \mid F,D \vdash_{UP} l\},$$

  Où Λ est un nœud spécial de conflit, $E = \{(\neg l_i, l) \mid \{l_1, \ldots, l_k, l\} \in F \text{ et } \neg l_1, \ldots, \neg l_k \in V\} \cup \{(x,\Lambda), (\neg x,\Lambda) \mid x, \neg x \in V\}$.
- Pour un graphe d'implication donné, une variable $x \in V$ s'appelle une variable de conflit, et x, ¬ x est des littéraux de conflit.
- Un graphe d'implication contient un conflit s'il contient une variable de conflit; DPLL a un conflit à une étape donnée si le graphe d'implication à cette étape contient un conflit.

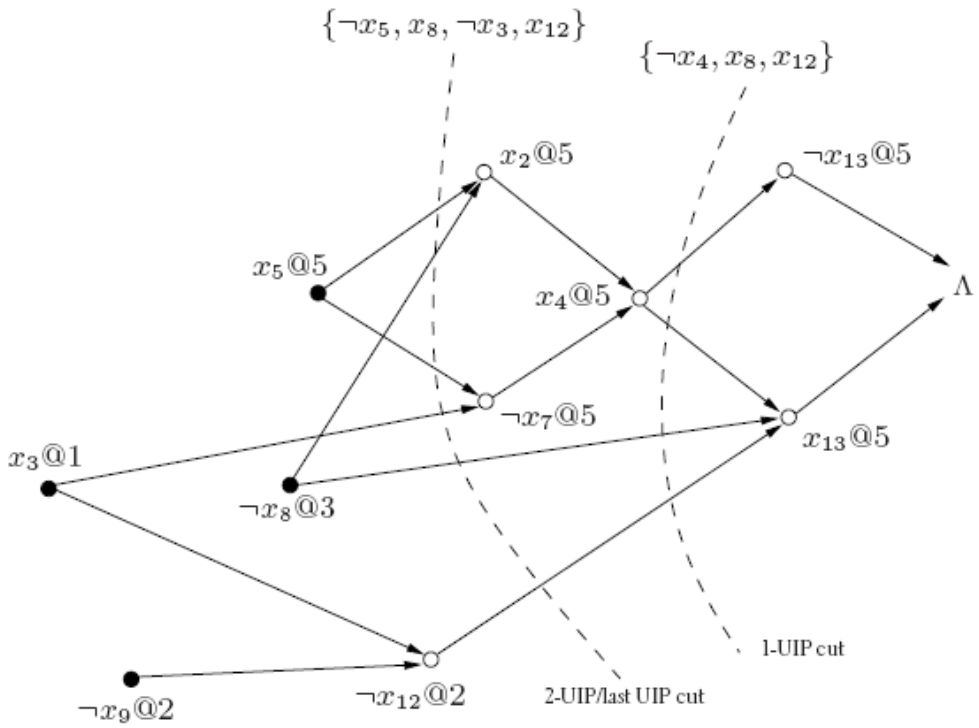

Fig 2.2: un exemple d'un graphe de conflit.

Nous présentons une procédure générique pour déterminer la satisfiabilité d'une formule f en CNF:

```
procedure SAT(CNF f ) : Booléen
    soit A = ∅
    repeat
        if f contient une clause vide then
            return FAUX
        else if quelques clauses c en conflit then
            ajouter clause deduce(c, A, f) à f
            supprimer quelque littéraux de A
            else if Af est total then return vrais
                else choisir un littéral l où l ∉ A et ¬l ∉ A
                    ajouter l à A

                end if
            end if
        end if
end
```

La procédure se termine quand l'affectation implicite Af est totale (dans ce cas nous avons une affectation satisfaisante) ou la clause vide 0 est déduite (dans ce cas f est

insatisfiable). Nous supposons que le graphe d'implication $IG(A, f)$ est mis à jour incrémentalement pour indiquer tous les changements dans A.

Les graphes d'implication capturent les chemins de dérivation des variables en appliquant la règle de clause unitaires sur les affectations branchées. Nous appliquerons ce concept dans le chapitre suivant pour définir la notion de clause learning. Cependant, nous avons d'abord besoin d'une certaine terminologie supplémentaire.

La procédure DPLL d'une formule CNF F est caractérisée par les littéraux de décision. Considérant pour une branchement aléatoire, les variables assignées en étant branché s'appellent les variables de décision et ces valeurs assignées par la propagation unitaire sont des variables implicites.

Le niveau de décision d'une variable de décision x est supérieure au nombre de variables de décision avant le branchement de x .Le niveau de décision d'une variable implicite x est le nombre des variables de décision quand x est assigné par une valeur. Le niveau de décision de DPLL à n'importe quelle étape est le nombre de variables de décision dans le branchement.

Pour une formule donnée de CNF F et un ensemble L de littéraux, nous dénotons par F, $L|-_{UP}$ l le fait que l peut être déduit à partir de F et L en appliquant itérativement la règle de clause unitaire (UP).

Noter qu'il y a beaucoup de choix d'heuristiques à faire. Les plus importantes sont le choix des littéraux à ajouter à A (l'heuristique de décision), le choix des littéraux à éliminer par la résolution dans conflict-based learning, et l'ordre d'établissement du graphe d'implication. Ces choix des heuristiques varient selon les solveurs et ils génèrent une forte efficacité de la procédure SAT. Pour plus de détails sur ses heuristique, voire le chapitre III.

K. L. McMillan prolonge cet algorithme de base pour convertir une formule booléenne arbitraire en CNF, plutôt que vérifier simplement sa validité [K. L. McMillan,2002]. L'idée est, étant donné une formule arbitraire p (pas dans CNF), plutôt que de vérifier la validité de p, nous souhaitons construire une formule équivalente à p sous forme normale conjonctive. Cela peut être fait par une modification légère de l'élément essentiel d'algorithme SAT.

```
procedure toCNF (p) : CNF
    soit f = CNF(p)∧¬lp ,x = 1 et A = ∅
    repeat
        if f contient une clause vide then return x
        else if quelques clauses c en conflit then
        ajouter clause deduce(c, A, f) à f
        supprimer quelque littéraux de A
            else if Af est total then
                choisir un blocking clause c'
                ajouter c' à f et x
                else choisir un littéral l où l ∉ A et ¬l ∉ A
                ajouter l à A

                end if
            end if
        end if
end
```

Chaque fois qu'une affectation satisfaisante est obtenue, la procédure produit une nouvelle clause dont le complément caractérise un ensemble d'affectations satisfaisantes (i.e., il élimine un ensemble de cas où p est faux).Quand la formule de CNF devient insatisfiable, ces clauses caractérisent avec précision p. Nous pouvons discuter l'exactitude partielle de cette procédure comme suit. La procédure maintient l'invariable que p implique χ (puisque seulement des clauses implicites par p sont ajoutées à χ). De plus, f est à tout moment équivalent à CNF(p) $\wedge \neg l_p \wedge \chi$. Ainsi, sur l'arrêt, quand f =0, il n'y a aucune affectation qui rend p faux et χ vraie, en d'autres termes, χ implique p. Si la procédure se termine, donc,χ une formule de CNF équivalente à p. la clause du blocage, doit avoir les propriétés suivantes:

- Il doit contenir seulement variables dans $V_I \subset V$,
- Il doit être faux dans l'affectation courante A_f, et
- Il doit être impliqué par $l_p \wedge$ CNF (p).

2.2 Les classes polynomiales de CNF

Nous rappelons que définir une classe polynomiale nécessite deux algorithmes de complexité polynomiale :

- un algorithme de reconnaissance, qui décide si l'instance du problème appartient à cette classe ou pas ;
- un algorithme de décision de la satisfiabilité des instances de cette classe.

Il est clair qu'il existe des classes d'instances ne répondant qu'à l'un ou l'autre des critères.

Ces classes sont quasiment inexploitables en pratique. Nous pouvons citer, par exemple, la classe constituée de l'ensemble des instances de SAT ayant un nombre polynomial de résolvantes. Clairement cette classe admet un algorithme de décision polynomial. Nous appelons restriction polynomiale une classe pour laquelle seul un algorithme de décision polynomial existe.

Plusieurs classes polynomiales de SAT sont connues à ce jour [L.PARIS, 2007]. Le tableau 2.1 donne un récapitulatif des différentes complexités des algorithmes de reconnaissance et de test de satisfiabilité de ces classes polynomiales.

Dans ce tableau, g(f) et f(f) représentent respectivement la complexité en temps de la reconnaissance et du test de satisfiabilité de la classe F.

Classe polynomiale	**Reconnaissance**	**Satisfaisabilité**
Horn	$\mathcal{O}(\|\Sigma\|)$	$\mathcal{O}(\|\Sigma\|)$
Horn renommable	$\mathcal{O}(\|\Sigma\|)$	$\mathcal{O}(\|\Sigma\|)$
Binaire	$\mathcal{O}(\|\Sigma\|)$	$\mathcal{O}(\|\Sigma\|)$
Γ_k (Gallo-Scutellà)	$\mathcal{O}(n^k\|\Sigma\|)$	$\mathcal{O}(n^k\|\Sigma\|)$
Quad	$\mathcal{O}(\|\Sigma\|^2)$	$\mathcal{O}(\|\Sigma\|^2)$
Δ_k et Ω_k(Dalal-Etherington)	$\mathcal{O}(n^{k+1})$	$\mathcal{O}(n^{k+1})$
Presque Horn[9]	$\mathcal{O}(n\|\Sigma\|)$	$\mathcal{O}(1)$
$\mathcal{F}$-Horn*	$\mathcal{O}(n\|\Sigma\| + g(\Sigma))$	$\mathcal{O}(\|\Sigma\| + f(\Sigma))$
q-Horn	$\mathcal{O}(\|\Sigma\|)$	$\mathcal{O}(\|\Sigma\|)$
Ordonnée	$\mathcal{O}(n\|\Sigma\|)$	$\mathcal{O}(n)$
Ordonnée renommable	$\mathcal{O}(n\|\Sigma\|)$	$\mathcal{O}(n)$
Presque ordonnée[9]	$\mathcal{O}(n^2\|\Sigma\|)$	$\mathcal{O}(1)$

TAB. 2.1 – les différentes classes polynomiales de SAT.

2.3. Le codage de CNF

Pour la représentation CNF, puisqu'elle est classique et c'est la première représentation de problèmes SAT, il y a une base DIMACS où tous les exemples fournis sont des formules CNF codées dans le format de CNF de DIMACS, ce format est soutenu par la plupart des solveurs fournis dans la collection de solveurs de SATLIB. Pour plus de détails du format de CNF de DIMACS, voir la définition de DIMACS sur le site http://www.satlib.org/Benchmarks/SAT/satformat.ps et ftp://dimacs.rutgers.edu/pub/challenge/satisfiability de DIMACS/

Les fichiers DIMACS sont composés de deux sections : un préambule et des clauses. Le préambule contient les informations sur l'instance dans des lignes, chaque ligne commence par un caractère pour déterminer le type de ligne, elle peut être :

- Un commentaire : donner des informations sur ce fichier et ignorer par le programme de résolution, elle doit toujours commencer par le caractère c,

- Ligne problème : apparaît dans la dernière ligne du préambule, suivie le format suivant : *p FORMAT VARIABLES CLAUSES*
  p signifier ligne de problème,
  FORMAT doit contenir « cnf »
  VARIABLES contient un entier n spécifiant le nombre de variables dans l'instance,
  CLAUSES contient un entier m nombre des clauses dans l'instance.

Les clauses figurent juste après la ligne problème et les variables sont supposés de 1 à n mais toutes les variables n'apparaissent pas nécessairement dans l'instance. Chaque clause sera représentée par une séquence des nombres séparé par des blanc, un tab ou un nouveau caractère ligne, la négation d'une variable *i* est représenté par *-i*. Chaque clause se termine par 0.

Exemple : soit une CNF d'un problème $(x1 \vee x3 \vee \neg x4) \wedge (x4) \wedge (x2 \vee \neg x3)$, le fichier input possible doit être :

```
c Example CNF format file
c
P cnf 4 3
1  3  -4  0
2 -3 0
4  0
```

Un autre exemple réel est aim-100-1_6-no-1.cnf

```
c FILE: aim-100-1_6-no-1.cnf
c
c SOURCE: Kazuo Iwama, Eiji Miyano (miyano@cscu.kyushu-u.ac.jp),
c         and Yuichi Asahiro
c
c DESCRIPTION: Artifical instances from generator by source.  Generators
c              and more information in sat/contributed/iwama.
c
c NOTE: Not Satisfiable
c
p cnf 100 160
16 30 95 0
```

-16 30 95 0
-30 35 78 0
-30 -78 85 0
-78 -85 95 0
8 55 100 0
8 55 -95 0
9 52 100 0
9 73 -100 0
…..

Les instances DIMACS Benchmark : on peut les regrouper par :

- AIM: Artificially generated Random-3-SAT - 48 instances satisfiables, 24 insatisfiables.
- LRAN: Large Random-3-SAT instances - 3 instances, tous satisfiables.
- JNH: Random SAT instances avec des clauses de longueur variable - 16 instances satisfiable, 34 instances insatisfiables.
- DUBOIS: Random generated SAT instances - 13 instances, tous insatisfiables.
- GCP: Large SAT-encoded Graph Colouring problems - 4 instances, tous satisfiable.
- PARITY: Instances pour le problème dans learning the parity function - 20 instances, tous satisfiable.
- II: instances d'un problème dans l'inférence inductive - 41 instances, tous satisfiable.
- HANOI: SAT-encoding de Towers of Hanoi - 2 instances, tous satisfiable.
- BF: Circuit fault analysis: bridge fault - 4 instances, tous insatisfiables.
- SSA: Circuit fault analysis: single-stuck-at fault - 4 instances satisfiable, 4 instances insatisfiables.
- PHOLE: Pigeon hole problem - 5 instances, tous insatisfiables.
- PRET: Encoded 2-colouring forcé être insatisfiables - 8 instances, tous insatisfiables.

3. Diagrammes de Décision Binaires

Les Diagrammes de Décision Binaires Ordonnés (OBDDs) [Bryant 1986] fournissent une telle représentation. Cette représentation est défini par des restrictions imposantes sue le Diagramme de Décision Binaire (BDD) présentée par Lee1 [lie 1959] et Akers [Akers 1978].

Définition 3.1 : Un diagramme de décision binaire est un arbre. A l'exception des feuilles, il sort de chaque nœud deux arcs et chaque nœud est étiqueté par une variable booléenne. Tous les nœuds qui sont à la même distance de la racine sont étiquetés par la même variable booléenne. Les feuilles sont étiquetées par 0 ou 1.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

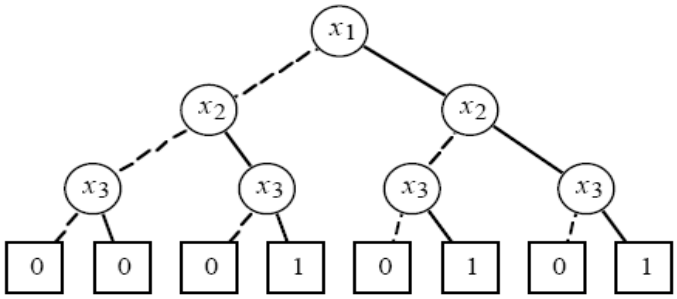

Fig 2.3: **Table de vérité et représentations de l'Arbre de décision d'une Formule booléenne.**

3.1. Ordonnancement et réduction

Il est possible de transformer un diagramme de décision binaire en un graphe qui représente la même formule booléenne mais qui est de taille moindre. De plus, cette représentation est unique (nous verrons que l'unicité dépend de l'ordre des variables). Cette structure est appelée un diagramme de décision binaire ordonné (OBDD).

Définition 3.2 (OBDD) [Bryant 1986]**:** Soient S un ensemble d'états, V un ensemble non-vide de variables booléennes, < un ordre total sur les variables de V et var : $S \rightarrow V \cup \{0, 1\}$ une formule qui associe au nœud $s \in S$ une variable. Un OBDD sur {V,<} est un graphe orienté acyclique (DAG) avec un ensemble non-vide S de nœuds qui sont de deux types :

- Des nœuds terminaux s étiquetés par une valeur booléenne, var(s) = 0 ou var(s) = 1 ;
- Des nœuds non-terminaux s étiquetés par une variable booléenne, $var(s) \in V$, qui ont deux enfants left(s) et right(s) tels que pour tout nœud $t : t \in \{left(s), right(s)\} \Rightarrow$ ((var(s) < var(t)) ou (t est terminal)).

La deuxième condition exige que chaque variable soit rencontrée au plus une seule fois et dans le même ordre pour tous les chemins qui commencent à la racine et qui se terminent à une feuille. C'est pour cette raison qu'on dit que le BDD est ordonné (OBDD).

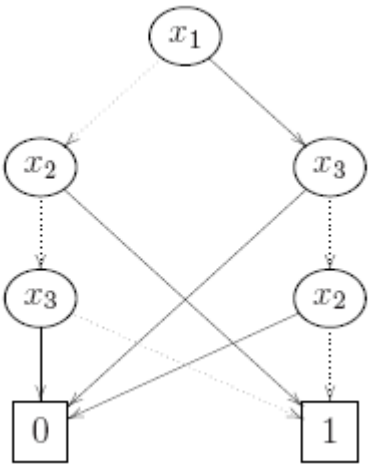

Fig 2.4 : Exemple d'un graphe orienté sans cycle non-ordonné.

Dans l'arbre de décision du figure 2.3, par exemple, les variables sont ordonnées $x_1 < x_2 < x_3$. En principe, l'ordre de variable peut être choisie arbitrairement Dans la pratique, le choix d'un ordre satisfaisant est critique pour la manipulation symbolique efficace. Cette issue est discutée dans la prochaine section.

3.1.1. Les règles de réduction des OBDD

Pour réduire la taille d'un BDD, Bryant [Bryant 1986] a ajouté trois contraintes à la définition des OBDD. Voici ces trois contraintes :

1. pour tous nœuds terminaux s et t, var(s) = var(t) $\Rightarrow$ s = t
2. pour tout nœud s, left(s) $\neq$ right(s)
3. pour tous nœuds s et t, ((var(s) = var(t)) $\wedge$ (left(s) = left(t)) $\wedge$ (right(s) = right(t))) $\Rightarrow$ s = t

Ces trois contraintes sont appelées les règles de réduction parce qu'à chacune correspond une transformation du OBDD pour que celui-ci respecte la contrainte. Un OBDD est réduit lorsque toutes les règles de réduction ont été appliquées et qu'aucune d'elles ne peut encore changer le graphe (nous verrons plus loin que l'ordre d'application n'est pas important). Les OBDD réduits sont souvent notés ROBDD de l'anglais Reduced OBDD. Dans ce texte, lorsque nous parlerons d'un OBDD, il sera réduit.

Exemple : Voyons à l'aide d'exemples ce que les règles de réduction impliquent.

- Pour appliquer la règle 1, Nous voyons sur la figure 3 une utilisation de la règle 1. Il ne doit y avoir que des sommets terminaux ayant les valeurs 0 et 1.

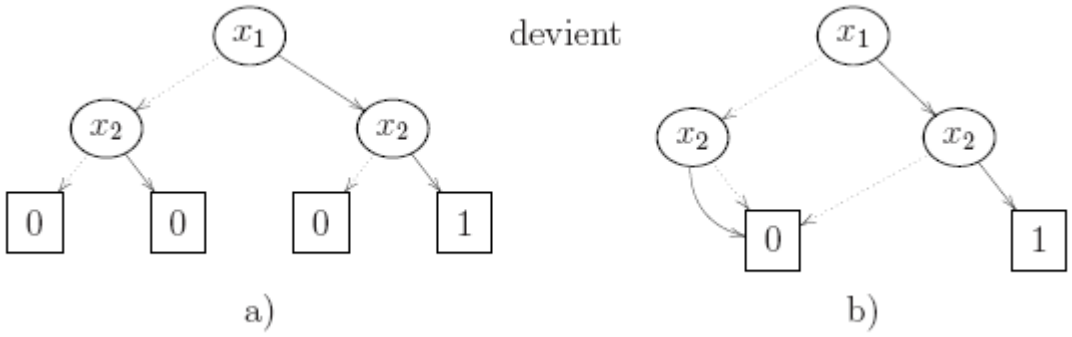

Fig 2.5 : Exemple de réduction par la règle 1.

- Pour appliquer la règle 2, il faut éliminer les nœuds dont les deux arcs sortants ont la même destination. La figure 2.6 nous montre une utilisation de la règle 2 sur le graphe b) de la figure 2.5.

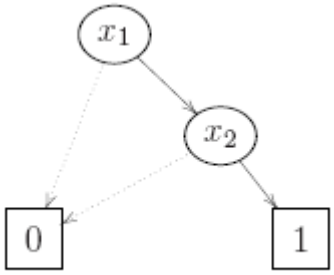

Fig 2.6 : Exemple de réduction par la règle 2.

- Pour appliquer la règle 3, nous devons conserver une seule copie des nœuds qui ont des enfants étiquetés par les mêmes variables. Cette règle sert donc à éliminer les sous-graphes isomorphes. L'OBDD de la figure 2.6 est réduit car les trois contraintes sont satisfaites. Réduisons maintenant l'OBDD de la figure 2.7.

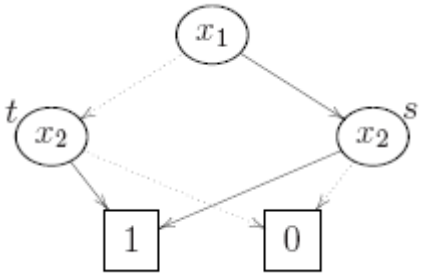

Fig 2.7 : Exemple de graphe contenant deux sous-graphes isomorphes.

Pour mieux les distinguer, nous nommerons t le nœud de gauche étiqueté x2 et s celui de droite. Remarquons que les sous-graphes de la figure 2.7 qui ont comme racine t et s respectivement, sont isomorphes. En effet, l'arc pointillé du nœud t pointe vers 0 et son arc plein vers 1. De même pour le nœud s. Par la règle 3, il faut garder une seule copie de ces sous-graphes. Nous obtenons alors l'OBDD de la figure 2.8.

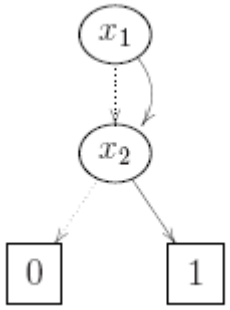

Fig 2.8 : Exemple de réduction par la règle 3.

IL reste à appliquer la règle 2 sur ce graphe pour obtenir un OBDD complètement réduit, ce qui donne le graphe de la figure 2.9.

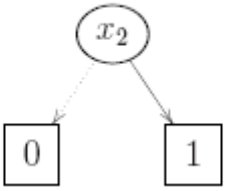

Fig. 2.9 : OBDD réduit.

L'algorithme suivant sert à réduire un BDD. La réduction se fait en traversant le BDD de bas en haut en donnant des étiquettes à chaque nœud. L'étiquette du nœud v est notée id(v) et E est l'ensemble des étiquettes.

```
Algorithme Réduction(BDD B )
    for toute feuille v
       if var(v) = 0 then id(v) = 0
       if var(v) = 1then id(v) = 1
    end for
    for toute hauteur à partir de la hauteur 1
       for tout sommet v
             if id(left(v)) = id(right(v)) then id(v) = id(left(v))
             else
             for tout v′
                   if var(v) = var(v′), id(left(v′)) = id(left(v)) et id(right(v′)) = id(right(v))
                   then id(v) = id(v′)
                   else id(v) = x o`u x ∈ E
                                E := E \ {x}
            End for
       End for
    End for
end
```

La réduction se fait en O(|B| · log |B|) où |B| représente la taille du BDD B [Bryant 1986]. Cette façon de faire est simple mais elle nécessite d'avoir le BDD qui représente la formule booléenne pour pouvoir le réduire et ce BDD peut être très gros. Il est possible aussi de construire et de réduire l'OBDD en même temps. La description de cette technique se fera plus en détail lorsque nous discuterons de l'implémentation des outils de manipulation des OBDD.

3.1.2. L'ordre des variables et la taille des OBDD

La taille de l'OBDD pour une formule booléenne donnée dépend essentiellement de l'ordre des variables dans celui-ci. Par exemple, la formule $f = (x_0 \vee x_1) \wedge (x_2 \vee x_3)$ avec l'ordre de variable $x_0 < x_1 < x_2 < x_3$ est représentée par l'OBDD de la figure 2.10. Par contre, si pour la même formule nous choisissons l'ordre $x_0 < x_2 < x_1 < x_3$, nous obtenons un OBDD de taille plus grande, celui de la figure 2.11.

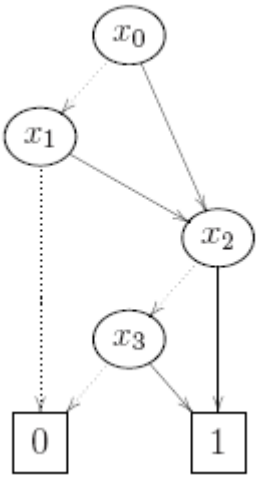

Fig 2.10 : OBDD de la formule f avec $x_0 < x_1 < x_2 < x_3$.

Plus généralement, si la formule est $g(x_0, ..., x_{2n-1}) = (x_0 \vee x_1) \wedge ... \wedge (x_{2n-2} \vee x_{2n-1})$ et que nous choisissons l'ordre $x_0 < x_2 < x_4 < ... < x_1 < x_3 < ... < x_{2n-1}$ nous aurons un OBDD de taille $2n+1 - 2$, c'est-à-dire toujours exponentiel (exemple tiré de Huth et Ryan [30]).

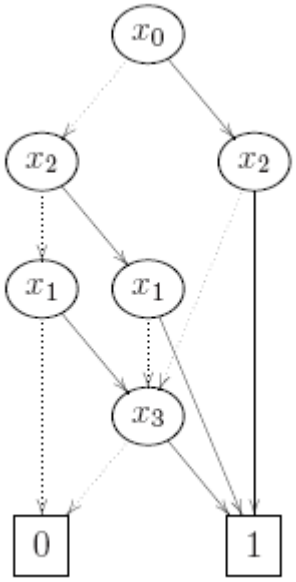

Fig 2.11 : OBDD de la formule f avec l'ordre x0 < x2 < x1 < x3.

Tandis qu'avec l'ordre x0 < x1 < ... < x2n−1, nous aurons 2n nœuds dans l'OBDD (attention, ici n n'est pas le nombre de variables). L'ordre des variables joue donc un rôle très important dans la taille d'un OBDD. Malheureusement, déterminer l'ordre qui minimise la taille de l'OBDD est un problème NP-Difficile [Bryant 1986]. Le meilleur algorithme connu pour résoudre ce problème s'exécute en O(3n · n2) [Bryant 1986], n étant le nombre de variables de l'OBDD. Il est généralement inutilisable sur des formules qui ont plus de 25

variables. Il faut alors se rabattre sur des heuristiques pour déterminer un ordre qui rendra l'OBDD plus petit.

Remarquons que puisque nous n'obtenons pas le même OBDD selon l'ordre des variables, il peut y avoir deux OBDD différents qui représentent la même fonction. Le théorème suivant nous assure cependant que pour un ordre de variable donné, la représentation par OBDD est canonique [12]. Dans ce qui suit, f_B est la formule booléenne que l'OBDD B représente.

Théorème 3.1 Soit V = {x0, x1, ..., xn} un ensemble de variables et < un ordre total sur V. Pour toute paire de OBDD B et B′ sur {V,< } on a $f_B = f'_B \Leftrightarrow$ B et B′ sont isomorphes.

Une conséquence du théorème est que l'ordre d'application des règles de déduction n'a pas d'importance puisqu'une fois réduit, l'OBDD obtenu sera toujours le même.

La taille des tables de vérité et des BDD ne dépend pas de la formule qu'ils représentent mais plutôt du nombre de variables dans celle-ci. Ce n'est cependant pas le cas avec les OBDD; ceux-ci n'ont pas toujours une taille exponentielle par rapport au nombre de variables. Ainsi, l'ordre de grandeur de la taille d'un OBDD peut être moindre que celui des tables de vérité. Malheureusement, la représentation reste exponentielle en pire cas.

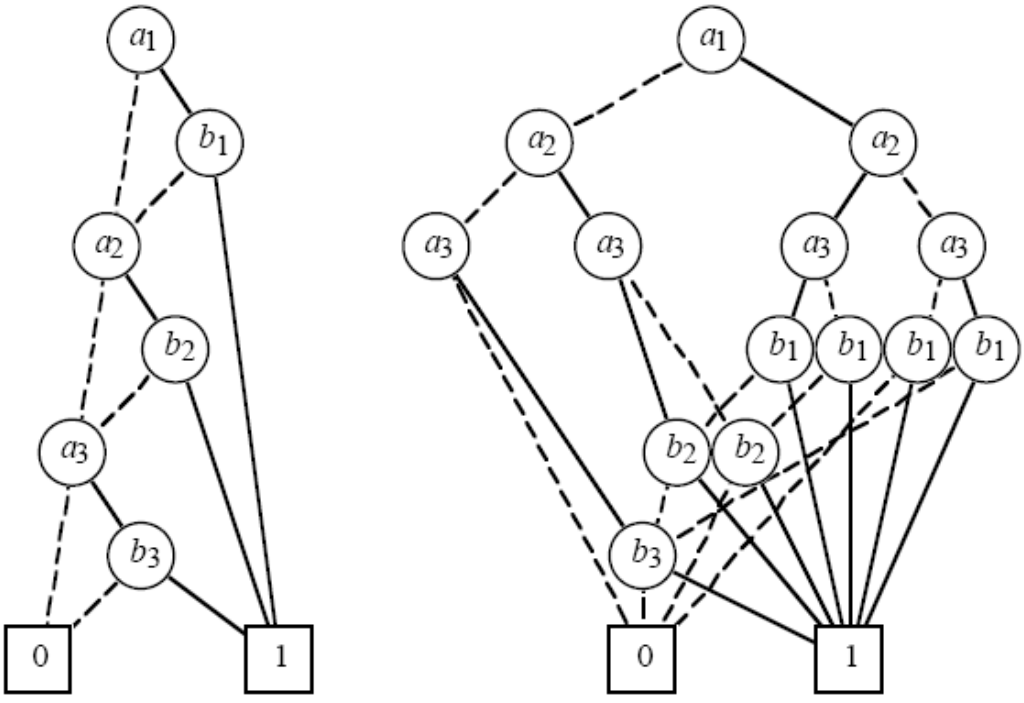

Fig 2.12: **Représentations d'OBDD d'une formule simple pour deux ordres de variable différent**.

Nous avons donc une représentation canonique des formules booléennes dont la taille peut être polynomiale par rapport au nombre de variables d'entrées.

La plupart des applications employant OBDDs choisissent l'ordre des variables au départ et construisent tous les graphiques selon cet ordre. Cet ordre est choisi manuellement, ou par une analyse heuristique du système particulier à représenter. Par exemple, on a conçu plusieurs méthodes heuristiques qui, donné un réseau de porte de logique, dérivent généralement un bon ordonnancement pour des variables représentant les entrées primaires.[Fujita et autres 1988 ;Malik et autres 1988].D'autres ont été développés pour l'analyse séquentielle de système [Jeong et autres 1991]. Noter que ces l'heuristique n'a pas besoin de trouver le meilleur ordonnancement, l'ordre choisie n'a aucun effet sur la convenance des résultats. Aussi longtemps qu'on peut trouver un ordre qui évite la croissance exponentielle, les opérations sur OBDDs restent raisonnablement efficaces.

3.2. Manipulations des OBDD

L'algorithme suivant [A.Darwiche et al,2004] décrit une procédure naïve de DPLL-modèle qui convertit un CNF ¢ en OBDD en convertissant périodiquement ses deux restrictions, $¢|_{vi=0}$ et $¢|_{vi=1}$, et en combinant les résultats en appelant get node (ligne 5), où $¢|_{x=0}$ et $¢|_{x=1}$ sont obtenus en plaçant x à 0 et à 1, respectivement, dans CNF ¢.

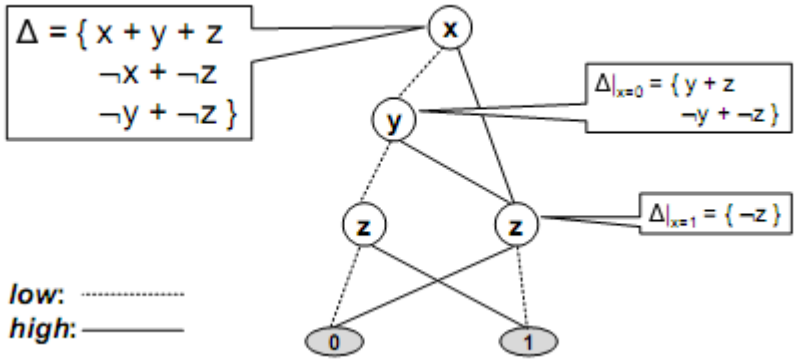

Fig 2.13: CNF et son OBDD correspondante

Noter qu'une technique commune connue sous le nom de nœuds uniques n'est employée que si le résultat final est un OBDD réduit par DAG. Spécifiquement on peut obtenir un nœud, sans construire de nouveau nœud, dans ces deux cas :

1. si ses deux derniers arguments sont identiques, l'un d'entre eux est retourné immédiatement ;
2. si là existe déjà un nœud qui est marqué avec le premier argument et a les deux derniers arguments comme enfants (dans le bon ordre), ce nœud est retourné.

```
Algorithme OBDD(CNF ¢, int i)// initialement i=1, value(S) retourne un vecteur
de bit représentant les valeurs des variables S dans un certain ordre fixe.
        if (il y a clause vide dans ¢) then
            return 0-sink
        if (il n'y a pas variable dans ¢) then
            return 1-sink
        if ((lookup = cache i-1[value(separator i-1)]) != nil ) then
            return lookup
        result = get node(i; obdd(¢|v̄i; i + 1); obdd(¢|vi ; i + 1))
        cache i-1[value(separator i-1)] = result
        return result

get node(int i, BDD low, BDD high)
        if low == high then
            return low
        if (lookup = unique[(i; low; high)]) != nil then
            return lookup
        result = new BDD(i; low; high)
        unique[(i; low; high)] = result
        return result
```

Pour plus de détails sur cet algorithme et le calcul de complexité voir l'article de [A.Darwiche et al,2004]. On peut dire que Pour CNF ¢ et un ordre de variable$\prod$, la complexité de l'algorithme OBDD est $O(sn2^w)$ en temps et espace, où s est la taille de ¢, n est le nombre de variables, et de w = de $pw_{¢}(\prod)$.

3.3. Les opérations

Les formules de logique sont souvent composées de sous-formules liées par des opérateurs. Divers algorithmes ont été développés pour effectuer ces opérations sur les OBDD. Les algorithmes présentés ici (voir le tableau ci-dessous) sont ceux de Bryant [12] et de Huth et Ryan [30].

Procédure	***Temps De Résultat***	***Complexité***
Reduce	G réduire au forme canonique	O(|G|·log|G|)
Apply f	1 <op> f	2 O(|G1|·|G2|)
Restrict	$f\|_{xi=b}$	O(|G|·log|G|)
Compose	$f1\|_{xi=f2}$	O(|G1|2·|G2|)
Satisfy-one	quelque élément de S_f	O(n)
Satisfy-all	S_f	O(n·|S_f|)
Satisfy-count	|S_f|	O(|G|)

Tab 2.2 : résumé des opérations de bases.

Dans ce qui suit, Chaque nœud dans la formule OBDD est représenté par un record déclaré comme suit :

```
type vertex = record
    low, high: vertex;
    index: 1..n+1;
    val: (0,1,X);
    id: integer;
    mark: boolean;
end;
```

Les nœuds non terminaux et terminaux sont représentés par le même type de record, mais les valeurs de champ pour un sommet v dépendent du type de sommet comme indiqué dans la table suivante.

champ	***Terminal***	***Non terminal***
low	null	low(v)
high	null	high(v)
index	n+1	index(v)
Val	value(v)	X

3.3.1 Procédure Reduction : transforme un graphe arbitraire de formule en graphe réduit dénotant la même formule. Pour examiner si deux arbres sont isomorphes. Procédant à partir des nœuds terminaux jusqu'à la racine, une marque unique de nombre entier est assignée à chaque racine unique de sous-graphe. C'est-à-dire, pour chaque nœud v elle assigne à un id(v) d'étiquette tels que pour deux nœuds quelconques u et v, id(u) = id (v) si et seulement si $fu = f\ v$. Donné ce marquage l'algorithme construit un graphe avec un nœud pour chaque étiquette unique.

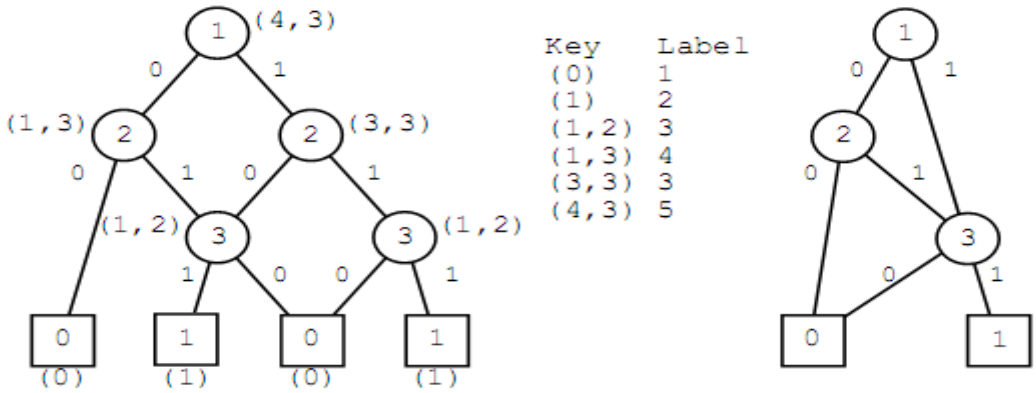

Fig 2.14: exemple d'exécution de Reduce

```
function Reduce(v: vertex): vertex;
    var subgraph: array[1..|G|] of vertex;
    var vlist: array[1..n+1] of list;
begin
    Put each vertex u on list  vlist[u.index]
    nextid := 0;
    for i := n+1 downto 1 do
    begin
        Q := empty set;
        for each u in vlist[i] do
            if u.index = n+1 then add <key,u> to Q where key = (u.value) {terminal
            vertex}
            else if u.low.id = u.high.id then u.id := u.low.id {redundant vertex}
                 else add <key,u> to Q where key = (u.low.id, u.high.id);
        Sort elements of Q by keys;
        oldkey := (−1;−1); {unmatchable key}
        for each <key,u> in Q removed in order do
            if key = oldkey then u.id:= nextid; {matches existing vertex}
            else
             begin  {unique vertex}
                 nextid := nextid + 1; u.id := nextid; subgraph[nextid] := u;
                 u.low := subgraph[u.low.id]; u.high := subgraph[u.high.id];
                 oldkey := key;
            end;
    end;
    return(subgraph[v.id]);
end;
```

3.3.2. Procédure Apply :

Est utilisée pour calculer des opérations binaires, comme le et et le ou logique, entre deux OBDD. Etant donnés deux OBDD B_f et B_g, représentant les formules f et g, l'appel de la formule apply(op,B_f ,B_g) avec op $\in \{\wedge,\vee,\oplus, ...\}$, retourne l'OBDD réduit représentant la formule booléenne f op g. L'algorithme s'exécute de façon récursive sur les deux OBDD de la façon suivante:

```
function Apply(v1, v2: vertex; <op>: operator): vertex
    var T: array[1..|G1|, 1..|G2|] of vertex;

    {Recursive routine to implement Apply}
    function Apply-step(v1, v2: vertex): vertex;
    begin
        u := T[v1.id, v2.id];
        if u ≠ null then return (u); {have already evaluated}
        u := new vertex record; u.mark := false;
        T[v1.id, v2.id] := u; {add vertex to table}
        u.value := v1.value <op> v2.value;
        if u.value ≠ X then
        begin {create terminal vertex}
            u.index := n+1; u.low := null; u.high := null;
        end
        else begin {create nonterminal and evaluate further down}
            u.index := Min(v1.index, v2.index);
            if v1.index = u.index
                then begin vlow1 := v1.low; vhigh1 := v1.high end
                else begin vlow1 := v1; vhigh1 := v1 end;
            if v2.index = u.index
                then begin vlow2 := v2.low; vhigh2 := v2.high end
                else begin vlow2 := v2; vhigh2 := v2 end;
            u.low := Apply-step(vlow1, vlow2);
            u.high := Apply-step(vhigh1, vhigh2);
        end;
        return (u);
    end;
begin {Main routine}
    Initialize all elements of T to null;
    u := Apply-step(v1, v2);
    return (Reduce(u));
end;
```

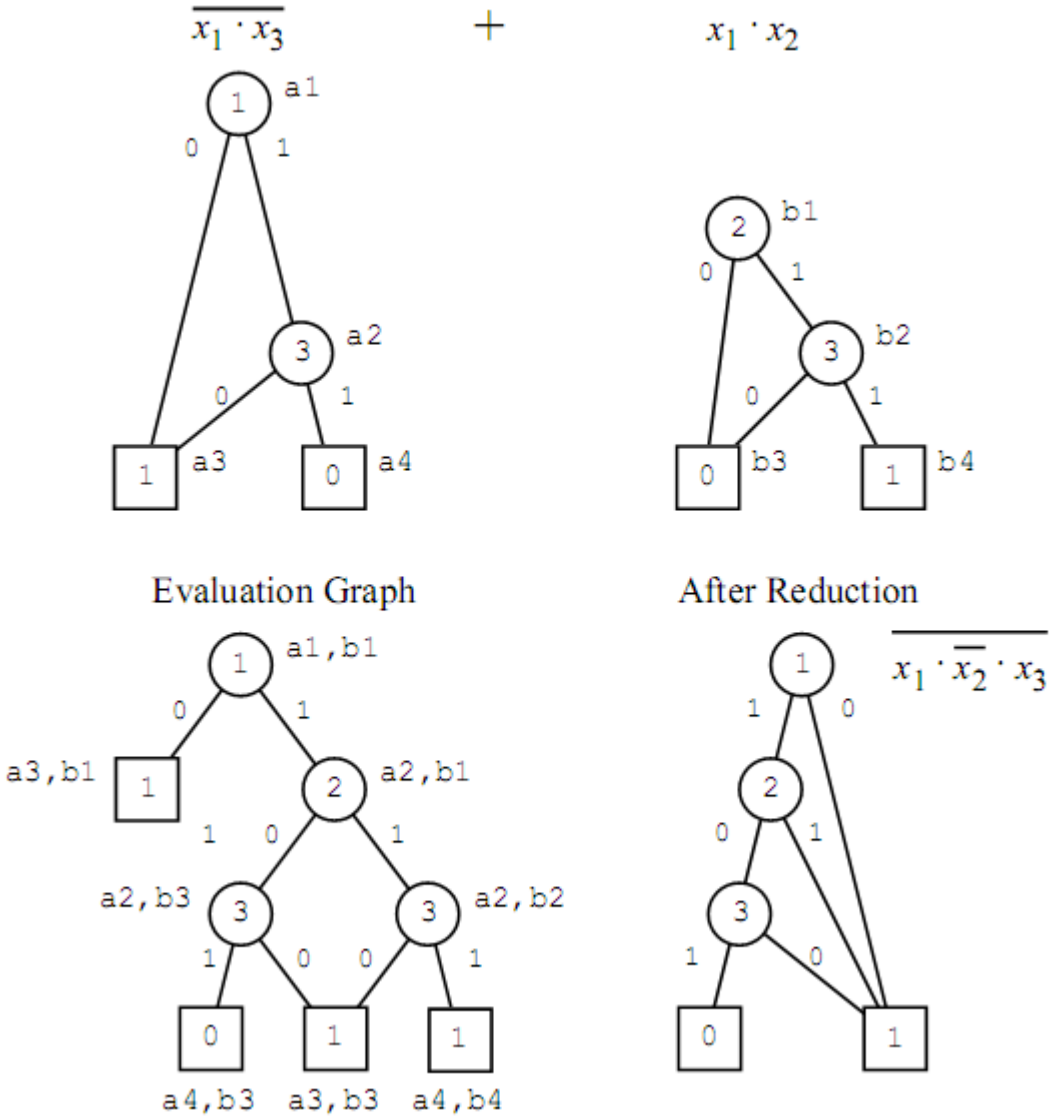

Fig 2.15: exemple de « or » Appy pour deux graphe.

L'algorithme implémenté de façon récursive a un temps exponentiel. Cependant, lorsque la programmation dynamique est utilisée, le temps d'exécution de la formule apply appliquée à deux OBDD est O(|Bf | ・ |Bg|), donc linéaire par rapport à la taille des OBDD.

3.3.3. Restriction de variables

L'algorithme restrict sert à fixer la valeur d'une variable dans l'OBDD. Ainsi, restrict(B, xi, b) retourne l'OBDD B dans lequel la valeur de la variable xi a été fixée à la valeur booléenne b. Cette opération se fait facilement en dirigeant tous les arcs pointant sur chaque noeud n étiqueté par xi vers le fils gauche (arc pointillé) de n si b = 0 et vers le fils

droit (arc plein) de n si b = 1. Avec les méthodes utilisées à l'implémentation, présentées à la fin du chapitre, l'OBDD obtenu est réduit. Cette méthode prend un temps en O(|B|).

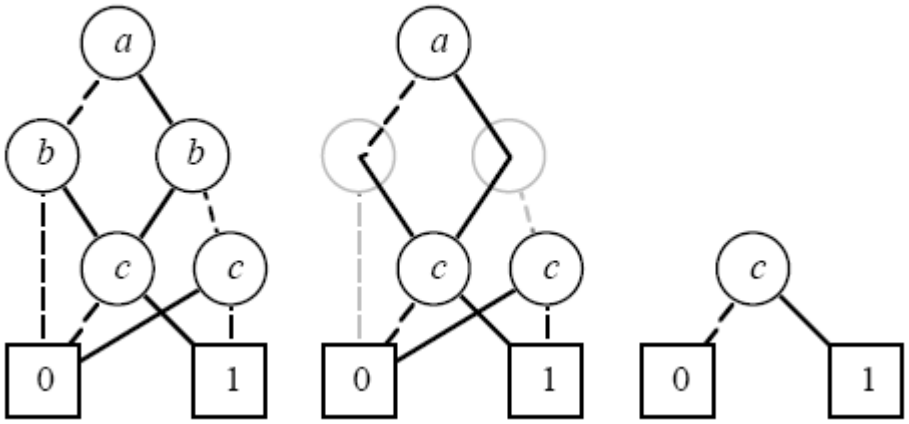

Fig 2.16 : Exemple d'opération Restrict.

Le variable *b* réduit de l'argument (gauche) à la valeur 1 comporte des sommets déviants marqués par *b* (centre), et réduire le graphique (droit).

En plus des articles de Bryant, plusieurs documents offrent une bonne documentation sur les OBDD.

Les algorithmes apply et restrict fournissent donc une manière de mettre en application ces autres opérations. En outre, pour chacune de ces opérations, la complexité et la taille du graphique produit sont liées par une certaine fonction polynôme des formules d'argument. Pour la formule *f,* soit m_f dénote la taille de sa représentation d'OBDD. Donné deux formules *f* et *g,* et "ne pas s'inquiéter" les conditions exprimées par une formule *d,* nous pouvons calculer l'équivalence *f* et *g* pour des conditions d en temps $O(m_f . m_g . m_d)$. Nous pouvons calculer la composition des formules *f* et *g* avec deux restrictions et trois appels à apply. Cette approche aurait la complexité $O(m_f^2 \; m_g \; m_d)$ .En mettant en application le calcul entier avec un traversal, cette complexité peut être réduite $O(m_f \; m_g^2)$ [Bryant 1986]. En conclusion, nous pouvons calculer la quantification d'une variable dans une formule *f* à temps $O(m_f^2)$

3.3.4. Satisfaction

Il y a beaucoup de questions qu'on pourrait se demander sur l'ensemble satisfaisant S_f d'une formule, y compris le nombre d'éléments, une liste des éléments, ou peut-être juste un élément simple. Comme peut être vu du tableau 2.2, ces opérations sont effectuées par des algorithmes de complexité considérablement variable. Si nous souhaitons trouver un élément

de l'ensemble satisfaisant certaine propriété, il peut être très inefficace d'énumérer tous les éléments de l'ensemble satisfaisant et puis de sélectionner un élément avec les caractéristiques désirées. Au lieu de cela, nous devrions indiquer cette propriété en termes de formule booléenne, calculons le produit booléen de cette formule et de la formule originale, et puis employons la procédure *Satisfy-one* pour choisir un élément. En conclusion, nous pouvons calculer la taille de l'ensemble satisfaisant par un algorithme de temps proportionnel à la taille du graphe (des opérations de nombre entier de la précision suffisante peuvent être effectuées dans le temps constant.) En général, il est beaucoup plus rapide pour appliquer cet algorithme que pour énumérer tous les éléments de l'ensemble satisfaisant (procédure *Satisfy-all)* et les compter (procédure *Satisfy-count*).

```
function Satisfy-one(v: vertex; var x: array[1..n] of integer): boolean
begin
    if v.value = 0 then return(false); {failure}
    if v.value = 1 then return(true); {success}
    x[i] := 0;
    if Satisfy-one(v.low, x) then return(true);
    x[i] := 1;
    return(Satisfy-one(v.high, x))
end;
```

La procédure fera backtracking seulement à un certain sommet quand le premier enfant qu'il juge est un sommet terminal avec la valeur 0, et dans ce cas-ci on le garantit de réussir pour le deuxième enfant. Ainsi, la complexité de l'algorithme est O(n).

Nous présentons ici un exemple des algorithmes SAT qui utilisent les OBDD au lieu de CNF pour la représentation du problème à résoudre ; cet algorithme comprend une procédure $sat\ (i)$ récursive qui est au commencement appelé $sat(n)$. Il y a $l(v)$ une variable globale pour chaque nœud v de G et une variable ai pour le $l \leq i \leq n$.

```
procedure Satisfy-all(i: integer; v: vertex; x: array[1..n] of integer):
begin
    if v.value = 0 then return; {failure}
    if i = n+1 and v.value = 1
    then begin {success}
          Print element x[1],...,x[n];
          return;
    end;
    if v.index > i
    then begin {function independent of xi}
           x[i] := 0; Satisfy-all(i+1, v, x);
           x[i] := 1; Satisfy-all(i+1, v, x);
    end
    else begin {function depends on xi}
           x[i] := 0; Satisfy-all(i+1, v.low, x);
           x[i] := 1; Satisfy-all(i+1, v.high, x);
    end;
end;
```

Pour énumérer tous les éléments de l'ensemble satisfaisant, nous pouvons exécuter une recherche approfondie du graphe, imprimer l'élément correspondant au chemin courant chaque fois que nous atteignons un nœud terminal avec la valeur 1. Le *Satisfy-all* applique cette méthode.

Comme l'algorithme précédent, cette procédure fonctionnera pour n'importe quel graphe de formule, mais il pourrait avoir besoin du temps exponentielle dans n pour un graphe non réduit indépendamment de la taille de l'ensemble satisfaisant (considérer un arbre binaire complet avec tous les nœuds terminaux ayant la valeur 0.) Pour un graphe réduit, cependant, nous sommes garantis que la recherche échouera seulement quand la procédure est invitée un nœud terminal avec la valeur 0, et dans ce cas-ci l'appel récursif à l'autre enfant réussira. Par conséquent au moins la moitié des appels récursifs au *Satisfy-all* produisent au moins d'une nouvelle valeur d'argument à un certain élément dans l'ensemble satisfaisant, et la complexité globale est $O(n.|S_f|)$.

4. la représentation sous Forme Decomposable Negation (Darwich) :

La représentation DNNF (Decomposable NNF) est conçu par A.Darwich en 1999 [A.Darwiche ,2001], il a récemment proposé une propriété de décomposabilité, qui transforme NNF en forme fortement tractable, connue sous le nom de Decomposable Negation Normal Form(DNNF).

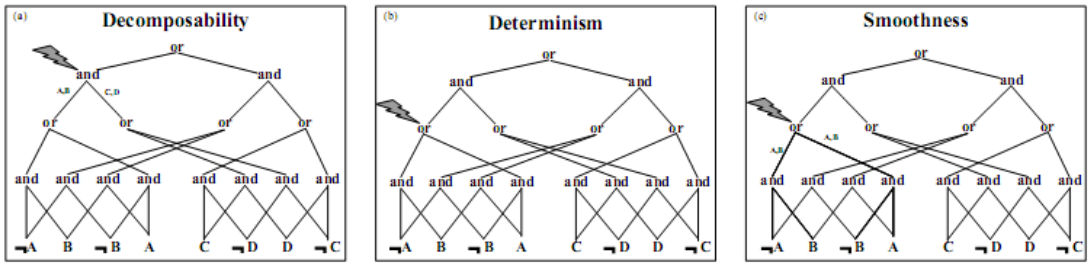

fig 2.17 : exemple de formule NNF.

Définition 4.1 : [A.Darwiche, 2001] le Decomposable Negation Normal Form de A (DNNF) est une forme NNF satisfaisant la propriété de décomposabilité.

Définition 4.2 : un nœud «∧» α dans une NNF est **décomposable** si et seulement si les conjoints de α ne partagent pas de variables. C'est-à-dire que si $\alpha 1, \ldots, \alpha n$ sont les fils du nœud«∧» α, alors $\mathrm{Var}(\alpha i) \cap \mathrm{Var}(\alpha j) = \varnothing$ pour tout i,j. Une formule NNF satisfait la propriété de décomposition si et seulement si tout nœud «∧» de cette formule est décomposable.

Le NNF dans la figure 1-a est décomposable, il y a dix conjonctions et les propositions conjointes de chacune ne partage aucune variable.

Par exemple le NNF suivante: $(A \vee B) \wedge (\neg A \vee C)$ n'est pas décomposable grâce au variable A qui est partagé entre les deux conjointes. Mais on peut les rendre décomposable par l'étude des cas des variables qui violent la propriété, notre cas est seulement la variable A. Supposant A est vraie donc NNF réduit au valeur de C et si A est faux, NNF réduit au B. le résultat finale est le NNF $(A \wedge C) \vee (\neg A \wedge B)$.qui est décomposable d'où DNNF. Cela est réalisé par l'opération de *conditionnement* qui réduit un NNF étant donné un littéral α .

Définition 4.3 : [A.Darwiche, 2001] soit Δ une formule propositionnel et soit α un O-instanciation. Le **conditionnement** de Δ par α, dénoté Δ | α, est une formule propositionnel obtenue en remplaçant chaque O-littéral dans Δ avec vrai (faux) s'il est conformé (contradictoire) à l'instanciation

Le *conditionnement* d'une NNF Δ par le littéral α est obtenu par le remplacement de chaque nœud feuille α dans le DAG par true et chaque nœud feuille $\neg\alpha$ par false. si le NNF Δ:(A∨ B) ∧(¬A∨ C), alors Δ | A=(true ∨ B) ∧ (false ∨ C) qui est simplifié au B.

D'où : $\Delta \equiv (\Delta|A \wedge A) \vee (\Delta|\neg A \wedge \neg A)$

Cela mène à une deuxième propriété utile appelée le déterminisme, provoquant la classe spéciale de DNNF Déterminé :

Définition 4.4 : [A.Darwiche, 2001].Un NNF **déterministe**, écrit le d-NNF, est un NNF satisfaisant les propriétés suivantes :

- pour chaque disjonction $\alpha = \vee i\, \alpha i$ apparaissant dans Δ, chaque paire de disjonction dans α est logiquement contradictoire, c.-à-d. $\alpha i \wedge \alpha j \models$ false for i≠j.

Déterminisme est la propriété qui fait l'opération d'Enumération des *Modèles* traitable.

Définition 4.5 :[A. Darwiche, 2001].Une forme NNF est **uniforme** ssi pour chaque disjonction $\vee i\, \alpha i$ sous la forme :atoms(α) = atoms(α i) pour chaque i.

Cette propriété simplifié les résultats de plusieurs opérations de DNNF comme ***Enumération des Modèles*** qui obtient dans un temps linéaire de O(nm) où n=|Models(Δ)|^2, et m: la taille de DNNF.

4.1. Les opérations tractables par DNNF :

4.1.1. Satisfiabilité

Décomposabilité est la propriété qui rend DNNF traitable: une décomposable formule NNF $\bigwedge_i \alpha i$ est satisfiable ssi chaque αi est satisfiable, et pour les $\bigvee_i \alpha i$ il suffit de satisfaite une des αi. La satisfiabilité d'un DNNF peut ainsi être évaluée dans le temps linéaire [A.Darwiche, 2001] au moyen d'un simple parcoure ascendant de son DAG.

Définition 4.6 [A. Darwiche, 2001].**:** pour une NNF F, SAT(F) est défini comme suit :

1) $SAT(F) = \begin{cases} true, & si\ F = true; \\ false, & si\ F = false; \end{cases}$

2) $SAT(F = \bigvee_i \alpha i) = true \quad ssi\ SAT(\alpha i)\ est\ true\ pour\ certain\ i,$

3) $Models(F = \bigwedge_i \alpha i) = true\ \ ssi\ SAT(\alpha i)\ est\ true\ pour\ tout\ i.$

Il devrait être clair que le SAT puisse être évalué dans un temps linéaire avec la taille de NNF. SAT est en effet valide et complet pour DNNF [A. Darwiche, 2001].

4.1.2. Enumération des Modèles

Définition 4.7 :[A. Darwiche, 2001] pour une NNF F, Model(F) est défini comme suit :

4) $Models(F) = \begin{cases} \{\{p = true\}\}, & si\ F = p; \\ \{\{p = false\}\}, & si\ F = \neg p; \\ \{\{\}\}, & si\ F = true; \\ \{\}, & si\ F = false. \end{cases}$

5) $Models(F = \bigvee_i \alpha i) = \bigcup_i Models(\alpha i).$

6) $Models(F = \bigwedge_i \alpha i) = \{\bigcup_i \beta i : \beta i \in Models(\alpha i).$

Définition 4.8 [Darwiche et al., 2005] le nombre des modèles pour le DNNF $\wedge i\, \alpha\, i$ est le produit des nombres des modèles pour chacune des conjonctions α i. et pour le DNNF $\vee i\, \alpha\, i$, le nombres des modèles sera la somme des nombres de modèles pour chacune des disjonction α i.

Une autre opération clé sur DNNFs est la projection.

4.1.3. Projection

Définition 4.9: [Darwiche, 2001]. Pour DNNF Δ et le variable A, projet (Δ, A) est défini comme suit: La projection d'une formule Δ sur un ensemble de variables V est la formule la plus forte impliquée par Δ sur ces variables. Nous le dénoterons par le Projet (Δ, V).

La projection est duale à l'élimination ou l'oubli : c'est-à-dire projetant Δ sur V est équivalent à l'élimination toutes les variables qui ne sont pas dans V.

Comme les autres opérations sur DNNF, la projection est linéaire [Darwiche, 2001]. Pour projeter un ensemble de variable V sur une DNNF Δ est de remplacer chaque littéral dans Δ par true (false) si ce littéral est mentionné ou non dans V. Par exemple, la projection de DNNF: (A ¬B) ∨ C sur les variables {B,C} est le DNNF (true∧¬B) ∨C ≡ ¬B ∨C.

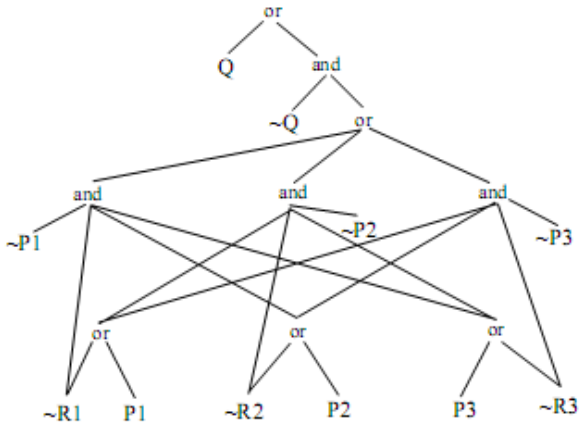

Fig 2.18: exemple d'une formule DNNF.

Concéderont le DNNF dans le figure ci-dessus, qui est équivalant à la formule : $\{p1 \wedge p2 \wedge p3 \rightarrow q, r1 \wedge \neg p1 \rightarrow q, r2 \wedge \neg p2 \rightarrow q, r3 \neg p2 \rightarrow q\}$, la projection de la formule sur r1, r2, r3 et q donne la formule DNNF $r1 \wedge r2 \wedge r3 \rightarrow q$, donc c'est simplement remplacer $p1, \neg p1, p2, \neg p2, p3, \neg p3,$ par vraie.

5. Comparaison

Une comparaison des différentes représentations de formules propositionnelles a été réalisée par DArwiche et al dans [Darwiche and Marquis, 2002] , une comparaison qu'ils ont mise à jour pour prendre en considération de nouvelles représentations telle que la d-DNNF.

Pour les besoins de la comparaison, des métriques ont été introduites. Nous sommes intéressés par conséquent à analyser la *succinct* et *tractabilité* de formalismes de représentation, afin que donné une tâche du raisonnement exigée, nous pouvons choisir le formalisme la plus succincte qui supporte l'ensemble d'opérations nécessaires dans un temps polynomial. Ce qui sui sont les définitions classiques de succinct et tractabilité:

Définition 5.1 Soient L1 et L2 deux sous-ensembles de NNF. L1 est au moins succinct(réduit) que L2, dénoté L1$\leq$ L2, ssi il existe un p polynôme tels que pour chaque $\alpha \in$ L2, existe un logiquement équivalent $\beta \in$ L1 où $|\beta| \leq\leq p\,|\alpha|$. Ici, le $|\alpha|$ et le $|\beta|$ sont les tailles du α et du β, respectivement.

Les formalismes que nous avons décrit dans ce chapitre satisfont les relations de succinct suivantes: NNF <DNNF <d-DNNF <FBDD <OBDD, NNF <CNF, et DNNF <DNF [Darwiche et Marquis, 2002].

Définition 5.2 : Un formalisme de SAT est dite tractable s'il existe un algorithme polynomial pour les résoudre.

En évaluant la convenance d'une représentation à une application particulière, le succincte du formalisme doit être équilibré contre l'ensemble de questions et de transformations qu'il soutient dans le temps polynomial.

Language	CO	VA	CE	IM	EQ	SE	CT	ME
NNF	○	○	○	○	○	○	○	○
DNNF	√	○	√	○	○	○	○	√
d-DNNF	√	√	√	√	?	○	√	√
BDD	○	○	○	○	○	○	○	○
FBDD	√	√	√	√	?	○	√	√
OBDD	√	√	√	√	√	○	√	√
$OBDD_<$	√	√	√	√	√	√	√	√
DNF	√	○	√	○	○	○	○	√
CNF	○	√	○	√	○	○	○	○

Tab 2.3 : questions Polynômiales pour un formalisme

NOTE : ° vous dire "ne satisfait pas à moins que P=NP ".

? vous dire "que nous ne connaissons pas".

Certains des résultats connus de Darwiche et de marquis [Darwiche et Marquis, 2002] concernant la tractabilité des formalismes sont récapitulés dans le tableau 2.3 (questions) et le tableau 2.4 (transformations).Les abréviations dans la première rangée du tableau 1 pour les huit questions suivantes, respectivement :**Uniformité (Co)** (est ce que la formule est satisfiable ?), **validité (VA)** (est ce que la formule évaluent à 1 pour toutes les affectations des variables), **Entailment Clausale (CE)** (est ce que la formule impliquent une clause donnée),

Implicant (IM) (est ce que la formule est implicite par un variable donné), **équivalence (EQ)** (est ce que les deux formules sont logiquement équivalentes), **Sentential Entailment (SE)** (est ce que une formule implique l'autre), **calcul de modèles(CT)** (combien d'affectation satisfaisant par la formule), **énumération modèle (ME)** (ce qui sont les affectations satisfaisantes de la formule).Les abréviations dans la première rangée du tableau 2.4 pour les huit transformations suivantes, respectivement : **Conditionement (CD)** (plaçant un ensemble de variables aux constantes), **oubliant(FO)** (mesurant de façon existentielle un ensemble de variables), **oublier Simple-Variable (SFO)** (mesurant de façon existentielle une variable simple), **conjonction** (conjoindre un ensemble de formules), **bounded conjunction** (conjoindre un nombre lié de formules), **disjonction** (disjoindre un ensemble de formules), **disjonction liée** (disjoindre un nombre lié de formules), **négation** (faire la négation d'une formule).

Language	CD	FO	SFO	∧C	∧BC	∨C	∨BC	¬C
NNF	√	○	√	√	√	√	√	√
DNNF	√	√	√	○	○	√	√	○
d-DNNF	√	○	○	○	○	○	○	?
BDD	√	○	√	√	√	√	√	√
FBDD	√	•	○	•	○	•	○	√
OBDD	√	•	√	•	○	•	○	√
$OBDD_<$	√	•	√	•	√	•	√	√
DNF	√	√	√	•	√	√	√	•
CNF	√	○	√	√	√	•	√	•

Tab 2.4 :Les transformations Polynômiales supportées par un formalisme.

Note : • vous dire "en satisfait pas,"

° *vous dire* "ne satisfait pas à moins que P=NP,"

? vous dire " ne connait pas".

Définition 5.3 (Co, VA) L satisfait Co (VA) ssi il existe un algorithme polynomial qui trace chaque formule f de L à 1 si f est consistant (valide), et à 0 sinon

Définition 5.4 (CE) L satisfait CE ssi il existe un algorithme polynomial qui trace chaque formule f de L et de chaque clause c de NNF à 1 si prises, et à 0 sinon.

Définition 5.5 (EQ, Se) L satisfait EQ (Se) ssi il existe un algorithme polynomial qui trace chaque paire de formules f , g de L à 1 si $f \equiv g$ (f |= g), et à 0 sinon.

Définition 5.6 (IM) L satisfait IM ssi il existe un algorithme polynomial qui trace chaque formule f de L et chaque littéral l de NNF à 1 si l |= f se tient, et à 0 sinon.

Définition 5.7 (CT) L satisfait CT ssi il existe un algorithme polynomial qui trace chaque formule f de L à un nombre entier non négatif qui représente le nombre de modèles de f (dans la numération binaire).

Définition 5.8 (ME) L satisfait ME ssi il existe un polynomial p(. ; .) et un algorithme qui produit tous les modèles d'une formule arbitraire f de L dans le p(n ,m) de temps, où n est la taille du f et m le nombre de ses modèles (variables appartient dans le f).

La figure suivante montre les résultats de cette comparaison, ainsi une flèche relie une représentation L1 avec une autre représentation L2 si L1 est une sous ensemble de L2

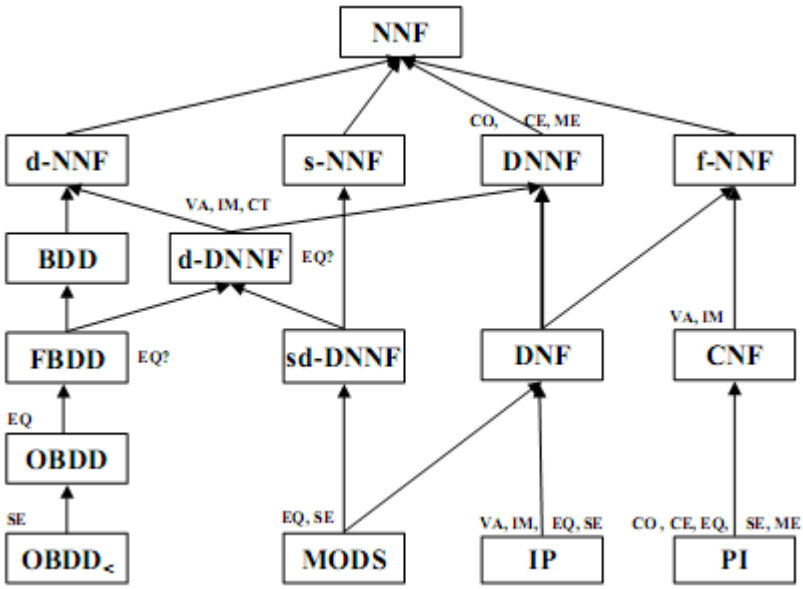

Fig 2.19: comparaison des représentations basées sur DAG.

Conclusion

Les deux versions, déterministe et généralisée de DNNF sont intéressantes. Elles sont les plus utilisées même si elles sont concurrencées par d'autres méthodes équivalentes telles que FBDD. Le succès est dû en partie à sa définition à la fois simple et générale; ce qui permet comme nous allons le voir dans le chapitre suivant, la mise en œuvre d'algorithmes pour la calculer.

Chapitre III

Les Méthodes de Résolution

Introduction

La résolution d'un SAT consiste à parcourir toutes les combinaisons possibles ou alors d'explorer un arbre de recherche (arbre de résolution) afin de trouver une ou toutes les solutions. De nombreux algorithmes ont été proposés pour résoudre SAT. Les principes de base de méthodes de résolution sont très variés: résolution, réécriture, énumération, comptage du nombre de solutions, recherche locale, programmation linéaire en nombre entiers, diagrammes binaires de décision, algorithmes génétiques et évolutionnistes, etc. [Singer, 2006]

Nous distinguons deux classes de solveurs : les solveurs complets et les solveurs incomplets. Les premiers sont les plus anciens, et répondent au problème donné que l'instance soit satisfaisable ou non, avec une complexité exponentielle dans le cas général. Les seconds ont l'avantage d'avoir une complexité paramétrable mais ne peuvent fournir de réponse que dans le cas d'instances satisfaisables, leur incapacité à fournir une réponse ne signifiant malheureusement pas que l'instance n'est pas satisfaisable.

Dans ce qui suit, nous allons présenter de façon générale les différents algorithmes proposés dans la littérature pour résoudre SAT.

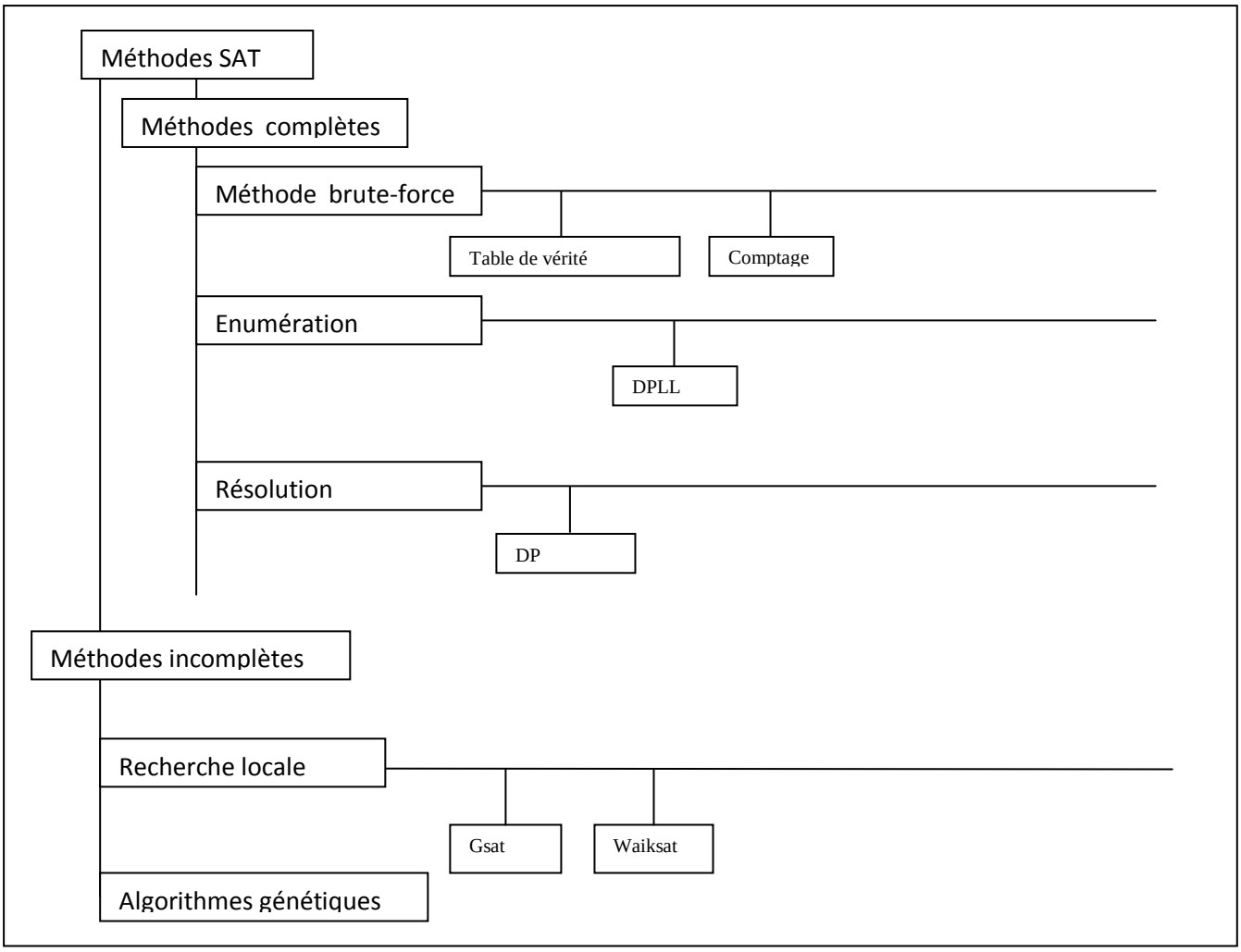

Fig 3.1. : Les méthodes de résolution SAT

1. Les méthodes complètes

On les dit « complets » car si on leur laisse un temps et un espace illimité, ils parcourent *complètement* l'espace de recherche. Il existe deux principaux types d'algorithmes complets. Les algorithmes syntaxiques et les algorithmes sémantiques. Les premiers s'intéressent à la structure du problème en ne tenant pas compte des valeurs des variables, on prendra ici pour exemple le principe de résolution. Les seconds s'attachent au sens des variables pour tenter de trouver une solution.

1.1. Résolution

Le système bien connu de preuve de résolution [197] (RES) est basé sur la règle de résolution. Soit C,D des clauses, et x une variable booléenne. La règle de résolution est

$$\frac{\{x\} \cup C \quad \{\neg x\} \cup D}{C \cup D}$$

ou, en d'autres termes, nous pouvons directement dériver $C \cup D$ à partir de $\{x\} \cup C$ et $\{\neg x\} \cup D$ par résolution de x .Pour une formule donnée de CNF F , la RES d'une clause C de F est une séquence des clauses $\pi = (C_1, C_2, \ldots, C_m = C)$, où chacun C_i $1 \leq i \leq m$, est:

(i) une clause de F (une clause initiale), ou

(ii) directement dérivé avec la règle de résolution de deux clauses C_j, C_k où $1 \leq j, k < i$ (une clause dérivée).

La longueur de π est m , le nombre de clauses dans π. RES (pour l'insatisfiabilité) d'une formule de CNF F est la RES de n'importe quelle dérivation de la clause vide ? de F .

N'importe quelle RES dérivation $\pi = (C_1, C_2, \ldots, C_m)$ peut être présentée comme un DAG dans lequel les feuilles sont les clauses initiales et les autres nœuds représentent la clause dérivée. La relation de bord est définie de sorte qu'il y ait des bords de C_i et C_j à C_k , si et seulement si C_k a été directement dérivé de C_i et C_j en utilisant la règle de résolution.

La résolution à l'origine remonte à Blake [SAÏS,2000] (voir aussi une variante de la résolution, la version originale de la procédure de Davis et Putnam [Davis & Putnam 1960]) est souvent attribuée à Robinson [Robinson 1965] qui a proposé une procédure de preuve en calcul des prédicats. Cette procédure permet de combiner le principe de résolution et d'unification [Robinson 1965].

Le principe de résolution est donc un « algorithme syntaxique ». C'est un principe élémentaire mais fondamental en logique. On applique successivement les trois règles suivantes jusqu'à l'arrêt de la procédure :

- règle de résolution
- règle de fusion
- règle de sous-sommation

Si on arrive sur la clause vide, c'est à dire que la clause générée ne contient plus de littéraux on peut conclure qu'il n'y a pas de solution. Par contre si l'on ne parvient plus à générer de nouvelles clauses et que l'on n'a pas généré de clause vide, c'est qu'il existe au

moins une solution. Le principe de résolution n'est pas l'algorithme complet le plus utilisé du fait que le nombre de clauses à ajouter dans le pire des cas est exponentiel par rapport à la taille initiale du problème.

La méthode propose de choisir une à une les variables et à chaque fois de générer toutes les résolvantes liées à cette variable. Ensuite on peut supprimer toutes les clauses contenant cette variable. Au final, soit on trouve la clause vide et on conclut à l'insatisfiabilité, soit on ne parvient plus à générer de résolvantes et on conclut à la satisfaisabilité de l'instance. Cependant, la génération d'un modèle est quelque peu fastidieuse et cet algorithme a tendance à saturer la mémoire. En effet, en cours d'exécution, le nombre de clauses générées peut vite s'avérer très important.

On appelle réfutation, toute suite de clauses générées par ce processus, la dernière étant une clause vide. Une réfutation peut être représentée par un *arbre binaire de résolution* dont la racine est la clause vide, tout nœud interne est étiqueté par la résolvante produite à partir de ses deux fils, les feuilles sont étiquetées par des clauses de la formule.

Cette méthode est relativement inefficace car le nombre de résolvantes à engendrer est souvent important. De nombreuses stratégies ont été proposées afin de limiter le nombre de clauses produites.

1.2. Énumération

L'idée de base des algorithmes énumératifs est la construction d'un arbre binaire de recherche où chaque nœud représente une instanciation partielle des variables. À chaque nœud de l'arbre, on procède généralement à un filtrage (simplification) de la formule courante. Le choix de la prochaine variable à instancier fait l'objet en général d'une heuristique. Un exemple typique est la procédure de Davis, Logemann et Loveland (communément appelée DPL) [Davis *et al.* 1962]: la simplification s'appuie sur la propagation des littéraux purs et des mono-littéraux; le choix de la prochaine variable à affecter est fait suivant une heuristique (e.g. choix de la variable figurant le plus souvent dans les clauses les plus courtes). La différence entre les deux procédures DP et DPL réside dans le fait que la première procède par élimination de variables en remplaçant le problème initial par un sous-problème plus large, alors que la seconde procède par séparation, en remplaçant le problème original par deux sous-problèmes de taille plus réduite. Ces deux procédures admettent des comportements différents et sont incomparables du point de vue de l'analyse de la complexité. La procédure DPL est plus souvent implantée et utilisée pour résoudre SAT

que la version originale DP; car la règle d'élimination de variables admet plusieurs inconvénients: elle est plus difficile à programmer que la règle de séparation; elle tend très rapidement à augmenter le nombre de clauses; elle génère de nombreuses clauses redondantes ou sous-sommées.

La procédure de Davis–Putnam–Logemann–Loveland

L'algorithme DPLL [Davis et Putnam 1960, Davis et al. 1962], nommé d'après ses inventeurs Davis, Patnam, Logemann et Loveland, est la procédure la plus classique pour décider si une formule propositionnelle en CNF est satisfiable ou non. Pour cela l'algorithme essaye de construire une instanciation des variables propositionnelles qui rende la formule vraie. En pratique, l'algorithme vérifie toutes les 2^n possibilités d'instancier les variables, mais il utilise deux heuristiques intelligentes qui lui permettent d'aller plus vite :

- **La propagation des contraintes booléennes :** à chaque fois qu'une valeur de vérité est choisie pour une variable, la formule est simplifiée en conséquence : les littéraux devenus faux sont supprimés des clauses, et si une clause contient un littéral devenu vrai, toute la clause est supprimée ;

- **La règle de la clause unitaire : si** la formule contient une clause qui ne contient qu'un seul littéral, la valeur de vérité de la variable du littéral est choisie de sorte que le littéral soit vrai.

De cette manière, l'algorithme procède en attribuant une valeur de vérité à chaque variable propositionnelle appartenant à la formule, jusqu'à ce que l'un des deux événements suivants arrive :

- La formule simplifiée est la conjonction vide $\varnothing$; dans ce cas, une instanciation des variables a été trouvée qui rende vraie la formule de départ : elle est donc satisfaisable et l'algorithme s'arrête ;

- L'algorithme a atteint un état dans lequel la formule simplifiée contient une clause vide (un conflit) : dans ce cas, l'algorithme effectue du backtracking à l'endroit le plus récent où il peut affecter une autre valeur de vérité à une variable.

D'un point de vue algorithmique, le choie de l'ordre de branchement des variables fait partie de la stratégie de solveur, et basé sur l'heuristique de branchement (ou décision). Les autres branches résultent de l'application particulière de la règle de branchement à

partir de backtracking. Avec cette intuition, DPLL se rapporte à une telle construction en utilisant les règles de branchement et des clauses unitaires.

Les évaluations de DPLL sont faites sur :

- les heuristiques de la décision des variables.
- La résolution Propagation Unitaire des clauses pures et les implications.
- La résolution des conflits et le backtracking.

```
Algorithme DPLL(CNF f ) : Booléen
    f* ← PropagationUnitaire(f)
    if f contient une clause vide then
        return FAUX
    end if
    if f* == ∅ then
        return VRAI
    end if
    l ← choixDeVariable(f*)
    if DPLL(f*∪{l=vrai})=VRAI then
        return VRAI
    else
        return DPLL(f*∪{l=faux})
    end if
end

Algorithme PropagationUnitaire(CNF f) :CNF
    if ∃ un littéral unitaire l then
        return PropagationUnitaire (f (l))
    end if
    if ∃ un littéral pur l then
        return PropagationUnitaire(f (l))
    end if
    return f
end
```

les heuristiques de décision des variables

Le choie de la variable de branchement est un facteur primordial de la procédure DPLL. Plusieurs heuristiques ont été conçues, l'heuristique Maximum Occurrences on Minimum sized clauses (MOM) [Marques-Silva, 1999], d'une efficacité et d'une simplicité étonnante, consiste à se brancher à la variable ayant le maximum d'occurrences dans les

clauses les plus courtes. Néanmoins, les travaux réalisés dans Posit et Satz [J. W. Freeman, 1995] suggèrent d'intégrer la propagation des clauses unitaires à cette heuristique.

Par conséquent, des heuristiques de branchement plus efficaces sont nécessaires. Dans Chaff [Moskewicz et al ,2001], les auteurs ont proposé une heuristique appelée Variable State Independent Decaying Sum (VSIDS) [Moskewicz et al ,2001]. Pour chaque littéral, on associe un compteur initialisé à 0. Puisque les solveurs SAT modernes ont un mécanisme d'apprentissage, des clauses sont ajoutées à la base de données des clauses pendant que la recherche progresse, à l'ajout d'une nouvelle clause, les compteurs des littéraux associés sont incrémentés de 1. Pour instancier une variable, on choisit celle qui n'est pas encore instanciée dont le compteur est maximal. Périodiquement, tous les compteurs sont divisés par une constante (= 2).

Les expériences prouvent que VSIDS est tout à fait concurrentiel comparé à l'autre heuristique de branchement sur le nombre de branches requises pour résoudre un problème. Puisque VSIDS est indépendant d'état (c.-à-d. les points ne dépendent pas des affectations des variables). Les expériences prouvent que la procédure de décision employant VSIDS prend un pourcentage très petit du temps d'exécution même pour des problèmes avec des millions de variables.

Dans d'autres efforts, le satz [Li, Anbulagan, 1997] a proposé l'utilisation de l'heuristique look-ahead pour s'embrancher; et les cnfs [O. Dubois, 2001] ont proposé l'utilisation de l'heuristique backbone-directed pour s'embrancher. Ils partagent le dispositif commun ce que tous les deux semblent être tout à fait efficaces sur des problèmes aléatoires difficiles. Cependant, ils sont également tout à fait chers comparés à VSIDS, alors qu'ils peuvent être pratiques pour appliquer ces heuristique chère aux problèmes aléatoires plus petits, leurs frais généraux tendent à être inacceptables pour les problèmes bien-structurés plus grands.

Plus récemment, BerkMin [Goldberg et Novikov, 2002] proposé un autre schéma de décision qui pousse l'idée de VSIDS plus loin. Comme VSIDS, la stratégie de décision essaye de décider des variables qui sont "en activité récemment".

La résolution Propagation Unitaire des clauses pures et les implications

Cependant, il semble que la règle de clause unitaire (pure) est la plus efficace parce qu'elle exige relativement peu de puissance informatique mais peut diviser les grands espaces de recherche. La règle de clause pure déclare que pour une certaine clause, si tout sauf une de ses littéraux ont été assignés la valeur 0, alors la variable (non affectée) restante doit être

assignée la valeur 1 pour satisfaire cette clause, qui est essentielle pour que la formule soit satisfaite. De telles clauses s'appellent les clauses unitaires, Le processus d'assigner la valeur 1 à tous les littéraux unitaires s'appelle Propagation Unitaire, ou la propagation de contrainte parfois appelée de Boolean (BCP). Presque tous les solveurs SAT modernes introduisent cette règle dans le processus de déduction. Dans un solveur SAT, BCP prend habituellement la partie la plus significative du temps d'exécution puisqu'il détecte la clause unitaire et celle de conflit. Par conséquent, son efficacité est directement liée à l'exécution du BCP.

Une exécution simple et intuitive pour BCP doit garder des compteurs pour chaque clause. Des schémas semblables sont plus tard utilisés dans beaucoup de solveurs tels que Grasp [J. P. M. Silva and K. A, 96], le satz [J. W. Freeman, 1995] etc…Par exemple, dans Grasp, chaque clause maintient deux compteurs, un pour le nombre de littéraux de la valeur 1 dans la clause et l'autre pour le nombre de littéraux de la valeur 0 dans la clause. Chaque variable a deux listes qui contiennent toutes les clauses où cette variable apparaît comme littérale positive et négative, respectivement. Quand une variable est assignée une valeur, toutes les clauses qui contiennent cette littéral auront leurs compteurs mis à jour comme le montre le schéma suivant.

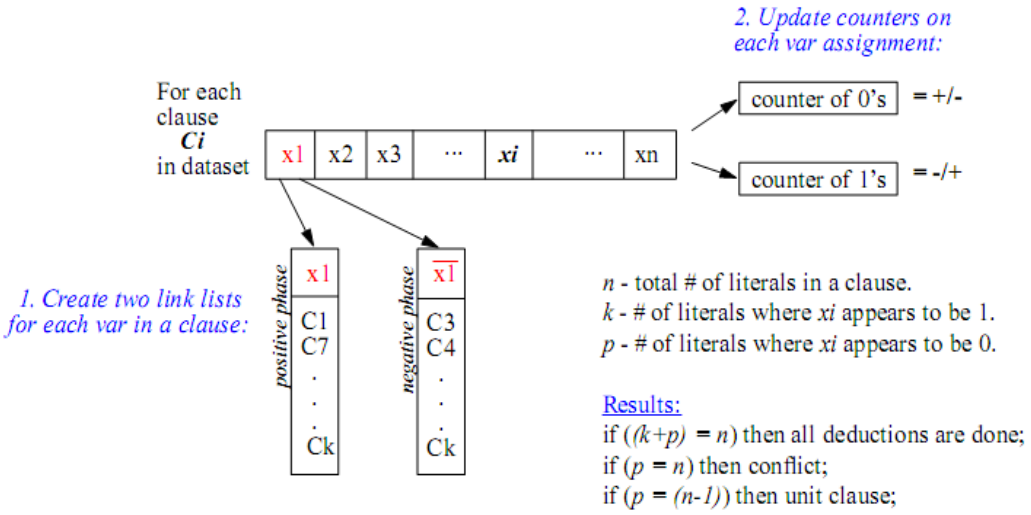

Fig 3.2 : le mécanisme de solveur Grasp

Il est facile de comprendre et implémenter un compteur-basé de BCP, mais ce schéma n'est pas le plus efficace. Si l'exemple a des clauses de m et des variables de n, et en moyenne chaque clause a l littéraux, alors toutes les fois qu'une variable obtenu est affectée, les

compteurs du lm/n ont besoin d'être mis à jour. Au backtracking d'un conflit, nous devons défaire les contre- affectations pour les variables non affectées pendant le backtracking.

Un autre mécanisme pour BCP utilisé dans SATO [H. Zhang et al, 1996] est head/tail liste. Dans ce mécanisme, chaque clause a deux indicateurs liés à elle, appelé l'indicateur de tête et de queue respectivement. Une clause stocke tous ses littéraux dans une rangée. Au commencement, l'indicateur de tête se dirige à la première littérale de la clause (c.-à-d. commencement de la rangée), et les points d'indicateur de queue à la dernière littérale de la clause (c.-à-d. fin de la rangée). Chaque variable garde quatre listes liées qui contiennent l'indicateur aux clauses.

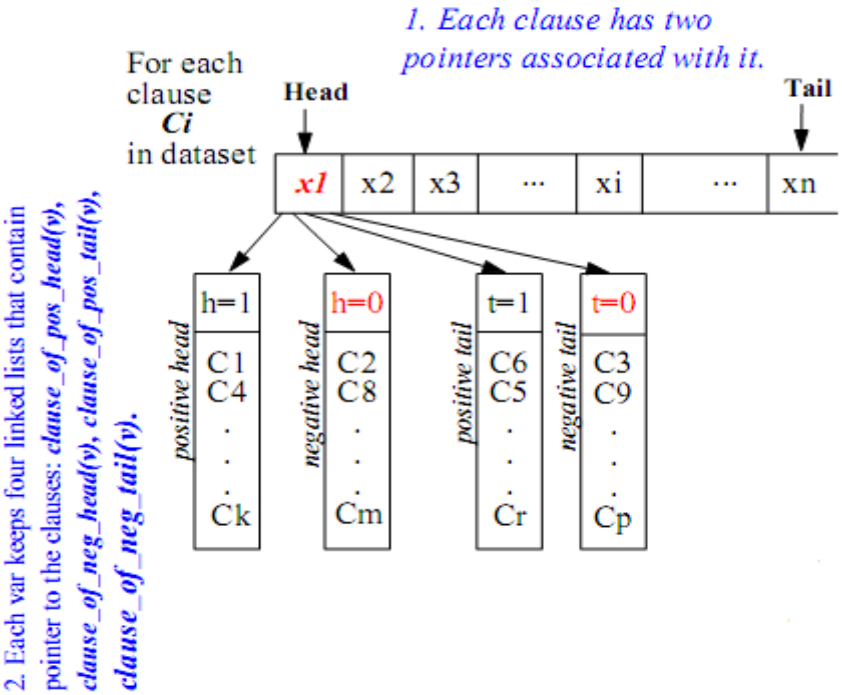

Fig 3.3 : le mécanisme de solveur SATO

2-literal watching utilisé par Chaff [Moskewicz et al ,2001] et ses dérivés comme zChaff [Zhaohui et al,2006] a le même avantage que le mécanisme de liste de head/tail. D'ailleurs, à la différence des deux autres mécanismes, défaire une affectation de variable pendant un backtracking est effectué dans un temps constant. C'est parce que les deux littéraux observées sont le dernier à assigner à 0, pour un résultat, n'importe quel backtracking s'assurera que les littéraux étant observées sont l'un ou l'autre non affectées, ou assigné à une. Ainsi, aucune action n'est exigée pour mettre à jour les indicateurs pour les littéraux observées. Par

conséquent, elle est sensiblement plus rapide que les mécanismes compteur-basé et de head/tail pour BCP.

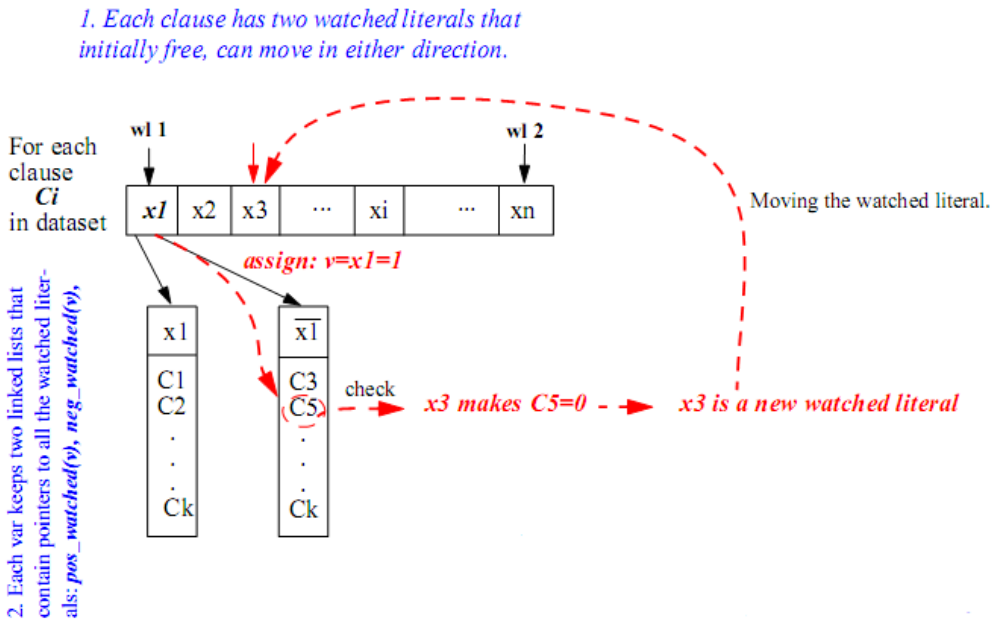

Fig 3.4: le mécanisme de solveur Chaff

Un exemple d'exécution de ce mécanisme est illustré dans la figure suivante pour la clause ¬v1 ∨ v4 ∨ ¬v7 ∨ v11 ∨ v12 ∨ v15:

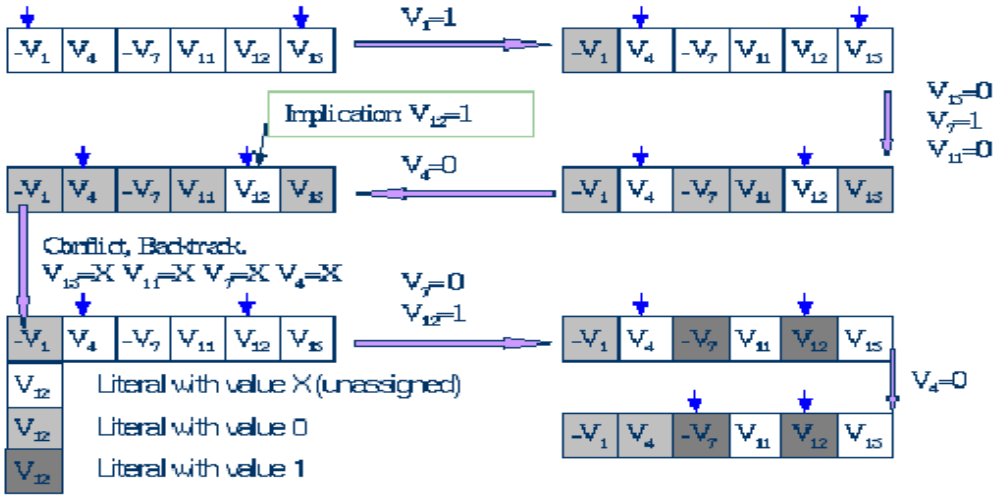

Fig 3.5 : Un exemple d'exécution du BCP avec 2-literal watching.

Dans [Daniel et al, 2003], les auteurs ont examiné les mécanismes mentionnés ci-dessus et ont présenté quelques nouvelles structures et mécanismes de données de déduction. En particulier, les expériences suggèrent que le mécanisme appelé la liste de Head/Tail

surpasse réellement le mécanisme 2-literal watching pour BCP. Cependant, les expériences sont effectuées dans un cadre mis en application dans Java comme OpenSat. Les auteurs admettent qu'il peut ne pas représenter l'exécution réelle si mis en application dans C/C++.

La résolution des conflits et le backtracking

Quand une clause contradictoire est produite, le solveur doit faire backtracking et défaire les décisions. L'analyse de conflit est la procédure qui trouve la raison d'un conflit et essaye de le résoudre. Elle indique au solveur SAT qu'il n'existe aucune solution pour le problème dans un certain espace de recherche, et indique un nouvel espace de recherche pour continuer la recherche. L'algorithme original de DPLL a proposé la méthode d'analyse de conflit la plus simple. Pour chaque variable de décision, le solveur continue s'il a été essayé en deux valeurs (c.-à-d. renversé) ou pas. Quand un conflit se produit, la procédure d'analyse de conflit recherche la variable de décision avec le niveau le plus élevé de décision qui n'a pas été renversé, et marquer la renverse, défait toutes les affectations entre ce niveau de décision et niveau courant, et puis essaye l'autre valeur pour la variable de décision. Cette méthode s'appelle le backtracking chronologique parce qu'elle essaye toujours de défaire la dernière décision qui n'est pas renversée. Le backtracking chronologique fonctionne bien pour des exemples aléatoire et sont utilisés dans certains solveurs SAT (par exemple satz [J. W. Freeman, 1995]).

Pour des problèmes structurés (qui est habituellement pour des problèmes produits de vraies applications du monde), le backtracking chronologique généralement n'est pas efficace en divisant l'espace de recherche. Des moteurs plus avancés d'analyse de conflit analyseront les clauses contradictoires produites et figureront hors de la raison directe du conflit. Cette méthode fera backtracking habituellement à un niveau la décision précédent que la dernière décision non changée. Par conséquent, cela s'appelle le backtracking non-chronologique. Pendant le processus d'analyse de conflit, des informations sur le conflit courant peuvent être enregistrées comme clauses et être ajoutées à la base de données originale. Les clauses supplémentaires, cependant superflues dans le sens qu'elles ne changeront pas la satisfiabilité du problème original, peuvent souvent aider à diviser l'espace de recherche à l'avenir. Ce mécanisme s'appelle conflict-directed learning. De telles clauses instruites s'appellent les clauses de conflit par opposition aux clauses contradictoires, qui se rapportent aux clauses qui produisent des conflits.

Le backtracking Non-chronologique, parfois désigné sous le nom du conflit a dirigé backjumping, d'abord dans le domaine du problème de satisfaction de contrainte (CSP). Ceci, ainsi que conflict-directed learning, ont été incorporés la première fois à un solveur SAT par Silva et à Sakallah dans Grasp [J. P. M. Silva and K. A, 96].Ces techniques sont essentielles pour la solution efficace des problèmes structurés. Beaucoup de solveurs tels que SATO [H. Zhang et al, 1996] et Chaff [Moskewicz et al ,2001] ont incorporé la technique semblable dans le processus de solution.

Les dernières années ont constamment vu évoluer les solveurs SAT, ceux-ci sont maintenant capables de résoudre de grandes instances et des problèmes difficiles du "monde réel" (instances de model-checking, de planification.). De nouvelles idées (d'implantation, d'algorithmes, de structures de données) sont constamment proposées.

Ce scénario pose quelques problèmes lorsque l'on veut évaluer les performances et les mérites d'un nouveau solveur ou d'un nouvel algorithme. Derrière différentes structures de données, de façons de coder, de langages, chaque solveur SAT implante ses propres techniques de recherches, stratégies et heuristiques. La taille de certaines instances (plusieurs dizaines de milliers de variables, plusieurs centaines de milliers de clauses) rend désormais une mauvaise implantation d'un algorithme dramatiquement lente. Idéalement, tous les algorithmes devraient être évalués avec le même degré d'optimisation. Une plate-forme de solveurs SAT permet de tendre vers cet objectif comme la plate-forme OpenSAT, Mini Sat etc.

2. Les méthodes incomplètes

Les algorithmes des méthodes incomplètes ont un temps d'exécution fixé à l'avance, et ne fournissent en général que deux réponses possibles : « *l'instance est consistante* » ou bien « *Impossible de statuer sur la consistance de l'instance* ». Lorsque la seconde réponse est donnée, cela signifie que l'instance est peut-être satisfaisable mais que l'algorithme n'a pu fournir une solution dans le temps qui lui était imparti.

Les méthodes considérées ici, contrairement aux algorithmes énumératifs, considèrent des « configurations », c'est-à-dire des instanciations complètes de toutes les variables de la formule. Au départ, ce sont des méthodes d'optimisation (on peut par exemple chercher à maximiser le nombre de clauses satisfaites). Sauf adaptation, ces méthodes sont incomplètes,

et ne garantissent donc pas l'obtention d'une solution. En particulier, elles ne permettent pas de prouver qu'une instance est insatisfiable.

La plupart des méthodes incomplètes sont basées sur : les algorithmes génétiques, recherche locale qui parcourt l'espace des interprétations, appelé aussi espace de recherche, de manière non systématique. Ainsi, contrairement aux méthodes complètes, ces méthodes disposent d'une plus grande flexibilité dans le choix des interprétations à visiter, car leur nombre est très réduit. Ces méthodes ont fait preuve d'une certaine efficacité dans le cas des problèmes qui ont un espace de recherche très grand.

Recherche locale

Une méthode de recherche locale n'effectue pas un parcours systématique de l'espace de recherche. Le plus souvent elle choisit une affectation des variables du problème au hasard et ensuite, si la solution n'est pas trouvée, on flippe (c'est à dire qu'on inverse sa valeur) les variables une à une en suivant une stratégie de réparation. L'algorithme s'arrête lorsqu'une solution est trouvée ou lorsque le temps ou l'espace dédié à l'algorithme est dépassé.

La méthode de choix de la variable à *flipper* caractérise la méthode de recherche locale. L'objectif de ces méthodes est de s'extraire des *minima locaux*. Il s'agit de situations où quelque soit la variable que l'on flippe, le nombre de clauses insatisfaites ne diminue pas. Il est nécessaire de trouver des heuristiques (dites d'échappement) qui permettent de sortir de ce genre de piège.

Pour cet algorithme, il faut dégager différents points stratégiques : tout d'abord dans les entrées, le nombre de réparations maximal n'est pas anodin. Il est parfois préférable de recommencer plusieurs fois l'algorithme à zéro plutôt que de s'enliser dans un nombre trop grand de réparations.

Ensuite la génération de la configuration initiale peut se faire de différentes manières. Il est par exemple possible de choisir une répartition 50/50 des littéraux vrais et des littéraux faux, ou encore mettre tout à vrai ou tout à faux. Ces choix peuvent s'avérer important selon la nature du problème.

Enfin, vient le choix de la variable à « flipper » qui s'effectue dans la boucle *while*. Certains algorithmes de recherche locale prônent une analyse totale du problème pour choisir celle qui aura le meilleur rendement. D'autres, pour gagner du temps, préfèrent prendre une

variable au hasard. Les meilleures solutions s'appuient sur un compromis entre ces deux extrêmes.[Selman *et al.*; 98]

```
fonction RL return boolean
Data: un ensemble F de clauses et max_reparations le nombre maximum de
réparations autorisées
Result: true si F admet un modèle, false si on ne peut conclure
 begin
        Générer une configuration initiale I;
        nb_reparations = 0;
        while (nb_reparations < max_reparations) and (I n'est pas un modèle)
do
            if une descente est possible then
               Remplacer I par une interprétation voisine permettant la descente;
           else
              Remplacer I par une interprétation voisine selon le critère
              d'échappement ;
           nb_reparations++;
        return (I est modèle) ;
        %% retourner la valeur du test : "I est un modèle ?"
 end
```

a) GSAT

L'algorithme GSAT est une heuristique de recherche locale qui a été introduite par Selman, Levesque et Mitchell [Selman et al, 1992]. GSAT se base sur l'idée de minimiser la valeur de la fonction objective qui est le nombre de clauses insatisfaites, en balayant le voisinage de la solution courante par le basculement de la valeur d'une seule variable de la solution.

Plusieurs variantes de GSAT ont été développées dans le but d'améliorer l'algorithme d'origine tel que GWSAT et HSAT [Hoos,2000].

a.1 GSAT de base

Dans l'algorithme GSAT simple, on commence par la génération d'une solution aléatoire à chaque itération de la recherche locale puis on choisit la variable dont le basculement de valeur (de vrai à faux ou de faux à vrai) donne le plus grand gain de nombre des clauses satisfaites. Si on a plusieurs cas avec le même gain, alors le choix sera aléatoire. On inverse la valeur de la variable choisie et on considère la nouvelle solution générée. On répète l'opération de sélection d'une variable et le basculement jusqu'à un nombre fixe d'itérations *MAX-FLIPS*. On recommence ce processus par la génération d'une nouvelle

solution aléatoire et on le répète jusqu'à la rencontre d'un critère d'arrêt. Ce critère peut être le nombre maximum d'essais *MAX-TRIES* ou de trouver la solution optimale.

```
Algorithme  GSAT (CNF f ; entier MaxTries, MaxFlips): Booléen

for i allant de 1 à Max Tries do
      I=une interprétation complète générée aléatoirement
      for j allant de 1 `a Max Flips do
            if I est un modèle then
                retourner V rai ;
           else
                for tout variable v de Σ do
                fals_to_sat[v]=le nombre de clauses falsifiées par I qui deviendraient
                satisfaites si v était flippée ;
                sat_to_fals[v]=le nombre de clauses satisfaites qui deviendraient falsifiées si
                v était
                flippée ;
                score[v]=fals_to_sat[v]-sat_to_fals[v] ; {min-conflict}
                end for
                list of max diff=liste des variables ayant le meilleur score ;
                x=une variable au hasard parmi list of max diff ;
                I=(I avec la valeur d'affectation de x inversée)
           End if
      End for
End for
retourner (I est modèle) {retourne la valeur du test : « I est-il un modèle ? »}
```

a.2 GSAT avec marche aléatoire (GWSAT : SAT with random walk)

GWSAT [Selman et al, 1992] [Hoos et al,1999] est une extension de la méthode GSAT de base, qui introduit un deuxième type de déplacement de la recherche locale, ainsi nommé le pas de la marche aléatoire. La méthode est de choisir aléatoirement une clause non satisfaite, puis l'une de ses variables sera basculée (changer la valeur). Donc, forcement la clause devient satisfaite. L'idée de base est de décider à chaque étape avec une probabilité fixée *P* (appelée probabilité de marche ou bruit de réglage) le pas à choisir (le pas de GSAT standard ou le pas de marche aléatoire).

b) WALKSAT

WALKSAT a été proposée par Selman, Kantz et Cohen en 1994 [Selman et al, 1994], mais elle a été définie formellement comme un algorithme par McAllester, Selman et Kantz en 1997 [K. L. McMillan,2002] [K. L. McMillan,2002]. Son principe est le même que GSAT mais se diffère de la fonction de score, qui est dans WALKSAT on prend en compte le

nombre des clauses qui sont endommagées (les clauses qui sont satisfaites mais après un basculement d'une variable, elles deviennent insatisfaites). Il y a quelques variantes de WALKSAT, tel que WTSAT [K. L. McMillan,2002]

b.1 WALKSAT de base

La méthode de fonctionnement de WALKSAT se base sur l'opération de sélection de la variable à basculer, en utilisant la fonction de score définie précédemment. La méthode de fonctionnement de cette opération est comme suit:

1. S'il existe une variable *x* avec *scoreb(x)=0*, alors cette variable est sélectionnée et sa valeur est basculée.

2. Si non, alors une variable avec le score minimal est sélectionnée avec une probabilité *P* (*noise setting*).

3. Dans les autres cas (si le choix n'est pas effectué dans 1 et 2) on sélectionne une variable de la clause aléatoirement (*random walk flip*).

b.2 WALKSAT/TABOU

WALKSAT/TABOU [K. L. McMillan,2002] est une variante de WALKSAT qui utilise une liste taboue. Cette dernière sert à empêcher le basculement d'une variable sélectionnée qui appartient à la liste taboue qu'après un nombre *tl* prédéfini d'itérations.

b.3 Innovation Novelty

Innovation Novelty [Matti,2008] est une autre variante de WALKSAT. Conceptuellement, elle est la combinaison de deux algorithmes : l'architecture de WALKSAT et le mécanisme de sélection de variables qui est inspiré de HSAT, De plus, elle utilise aussi une liste taboue, et la fonction de score de GSAT.

Comme les autres méthodes qui sont présentées précédemment, Novelty a plusieurs variantes, comme : Novelty+ [Hoos,2000], R-Novelty [Matti,2008] et R-Novelty+ [Hoos,2000].

Conclusion

Nous avons détaillé quelques uns des algorithmes de résolution qui se basent sur la recherche énumérative en mettant l'accent sur les différences qui existent entre les algorithmes en modifiant les heuristiques de la décision des variables, la résolution Propagation Unitaire des clauses pures, les implications, et la résolution des conflits. Avec cet outil nous avons pu confirmer l'intérêt pratique des algorithmes de résolution qui se base sur la recherche. Néanmoins, leurs complexités théoriques, étant exponentielles en la taille de problème, laissent présager l'impossibilité de leur utilisation pour des classes de SAT de grande taille. A cet effet nous allons explorer, dans le chapitre suivant, une autre approche de résolution de SAT qui se base sur les propriétés structurelles des SAT et permettant de réduire la complexité théorique de leur résolution.

Chapitre IV

Compile & SAT : une méthode pour la résolution séquentielle du SAT

Introduction

Deterministic Decomposable Negation Normal Form, d-DNNF, est une représentation tractable, qui permet à quelques questions logiques généralement intractables d'être calculées dans un temps polynômial. Ces questions incluent la satisfiabilité la minimisation et l'énumération des modèles. Plus précisément, le d-DNNF est meilleur que OBDDs, qui est populaire en soutenant diverses applications AI, y compris le diagnostic et la planification.

1. Processus de compilation :

La forme normale disjonctive (DNF), sans le partage des littéraux, est un sous-ensemble de DNNF, alors toutes les propriétés que nous prouvons pour DNNF sont également valides pour DNF. La question qui peut surgir alors est pourquoi ne pas compiler des formules propositionnelles dans DNF ? Car il s'avère qu'il y a des formules propositionnelles qui, malgré des représentations exponentielles sous DNF, ont des représentations linéaires de DNNF.

La question est la même pour les autres représentations des formules propositionnelles, la preuve de Darwiche *[Darwiche, 2001] est* illustrée dans le théorème suivant:

Théorème : soient $f1\ et\ f2$ deux DNNF et soit X= $var(f1) \cap var\ (f2)$. Soit f la formule $\vee_{\beta}(f1\ |\beta) \wedge\ (f2\ |\ \beta) \wedge \beta$, où β est X-instanciation.

Alors f est une DNNF et équivalente à $f1\ \wedge f2$.

Cet algorithme convertit n'importe quelle formule en forme clausale en DNNF équivalente, mais aux dépens de l'augmentation de la taille de formule. L'augmentation de la taille vient principalement de l'étude de cas exécutés sur les variables partagées par les sous formules f1 et f2.

D'abord, la taille de DNNF résultant est sensible à la méthode que nous utilisons pour couper la formule f en deux sous formules f1 et f2. En second lieu, la procédure ci-dessus n'est pas déterministe puisqu'il n'indique pas comment couper la formule f en deux sous formules. Pour rendre la procédure déterministe, Darwiche a utilisé un arbre de décomposition qui représente une division récursive des clauses dans f.

Définition 1 : Un dtree (decomposition tree) [Darwiche et Hopkins, 2001] est un arbre binaire complet qui induit une décomposition récursive sur un DAG. Un dtree est utilisé pour conduire des algorithmes de division.

Ce qui suit est la définition formelle d'un dtree.

Définition 2 [Darwiche Hopkins, 2001].: Un dtree t pour une CNF C est un arbre binaire complet dont les feuilles correspondent aux clauses de C. Chaque nœud interne représente un sous-ensemble de C correspondant à toutes les feuilles sous lui ; la racine représente en particulier la formule originale C. Sachant qu'un arbre induit naturellement un schéma récursif de décomposition, partitionnant la formule en deux parties représentées par les deux nœuds fils tl et tr.

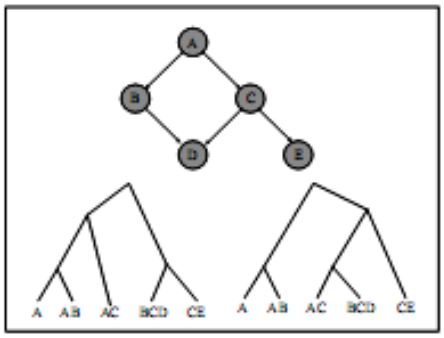

Fig 4.1 : un DAG et ses deux dtree correspondant

La figure 4.1 illustre deux dtrees pour le DAG représenté sur la même figure. Examinant le premier dtree t, Le niveau supérieur indique une partition des formules de DAG en deux ensembles :{A, AB, AC} et {BCD, CE} le sous-arbre gauche indique une partition

des formules noté t_l:{A, AB, AC}, alors que le sous-arbre droit indique une partition des formules noté t_r {BCD, CE}.

Un certain nombre de propriétés peuvent être défini pour les nœuds de dtree [Darwiche et Hopkins, 2001].

Définition 3 : le **cutset** d'un nœud interne t de dtree, noté cutset(t), est le suivant :

$$cutset(t) = var(tl) \cap var(tr) - acutset(t)$$

Où acutset(t) est l'union de cutsets de tous les ancêtres de t.

C'est-à-dire, le cutset du nœud t de dtree contient les nœuds de DAG qui doivent être placés afin d'enlever toutes les variables partagées entre les formules sous le t_l et celles sous le t_r.

Définition 4 : le **contexte** d'un nœud t dans un dtree, noter context(t), est défini comme suit : $var(t) \cap acutset(t)$.

Définition 5: - le **cluster** d'un nœud t dans un dtree est défini comme suit :

$$cluster(t) = \begin{cases} var(t), & si\ t\ est\ un\ noeud\ feuille; \\ cutset(t) \cup context(t), & sinon. \end{cases}$$

- Le **width** d'un dtree est défini par la taille de ses clusters les plus larges moins 1

Définition 6 [Darwiche et Huang, 2004**]: l'ordre du groupe de variable** (v.g.o.) induit par un dtree est obtenu périodiquement comme suit :

- produire le cutset de la racine ;

- produire le v.g.o.de son nœud feuille gauche ;

- produire le v.g.o.de son nœud feuille droit ;

- jeter les ensembles vides.

1.1 Compilation de CNF au d-DNNF :

Une formule propositionnelle f peut être compilé au d-DNNF en ordonnant simplement les variables qui apparaissent dans f dans un ordre $x1.. .,xn$, et puis partager f sur x1 suivant $(f|x1 \wedge x1) \vee (f|\neg x1 \wedge \neg x1)$, et puis compilant périodiquement chacune des formules conditionnées $f|x1$ *et* $f|\neg x1$ en utilisant le sub-ordre $x2\ldots,xn$. Pour éviter de compiler deux fois les mêmes formules, la compilation est couplée avec un cache.

Pour contrôler ce processus de décomposition, nous employons un arbre de décomposition (dtree), qui est un arbre binaire dont les feuilles sont étiquetées avec les clauses de CNF.

L'algorithme ci-dessous présenté par [Darwiche et Hopkins, 2001] décrit la méthode de construction de dtree à partir d'une CNF :

Algorithme cl2dt
/* f est le CNF*/
/* π est l'ordre $\pi(1), \pi(2), \ldots$ des variables de f*/
/* compose(Γ) combine les arbres dans τ arbitrairement à une seule arbre binaire*/
/* compose({T}) =T.*/
/* compose({T_1,T_2}) est un arbre binaire avec un nœud racine ayant T_1et T_2 comme nœuds fils*/
/* compose({T_1,…,T_n})= compose({T_1 compose({T_2,…,T_n})}), où n>2.*/

cl2dt(f,π)
$\Sigma \leftarrow \{T\alpha: \alpha \epsilon f\}$, //Où Tα est dtree contient un seul nœud t et $f(t) = \{\alpha\}$
pour i=1 à length(π) **faire**

$\Gamma \leftarrow \{T : T \epsilon \Sigma, \text{var}(\pi(i)) \epsilon \ l'arbre T\}$
$\Sigma \leftarrow (\Sigma \backslash \Gamma) \cup \{compose(\Gamma)\}$
Retourner $compose(\Sigma)$
Fin

La figure 2 décrit un dtree pour un CNF qui contient 4 clauses. Chaque nœud interne n de dtree a un ensemble de variables liées à lui, appelé le séparateur S, qui est simplement les variables qui sont partagées par les sous-arbres gauches et droits du nœud n.

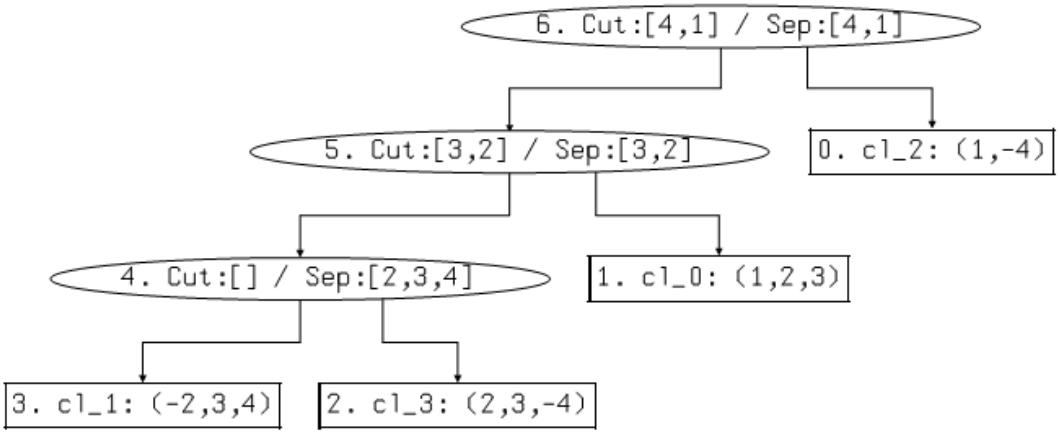

Fig 4.2 :Un dtree pour une CNF avec quatre clauses.

Chaque nœud de dtree de la figure 4.2 est marqué avec un index commençant à 0 (s'étendant de 0 à 6). Des nœuds de feuille sont étiquetés avec les clauses de CNF, marqué également avec un index commençant à 0 (s'étendant de 0 à 3).Chaque nœud interne est étiqueté par un cutset et un séparateur (sep).

Les utilisations de la procédure de compilation en d-DNNF est :

- une version approfondie de la procédure DPLL [Darwiche, 2004],

- décomposer chaque nœud de dtree, en assurant que les variables de séparateur pour ce nœud sont l'une ou l'autre instancié ou n'ont plus partagé entre les sous-arbres gauches et droits.

Quand cela se produit, la procédure de compilation est appliquée récursivement sur les sous-arbres gauches et droits, compilant chacun indépendamment et combinant les résultats. Elle emploie également des techniques sophistiquées pour éviter de compiler en mêmes temps de sub-CNF, autant que possible.

```
Algorithme  cnf2ddnnf(dtree t):
    term = all newly implied (not decided) literals of dtree t
    compute current separator s of dtree t
    if s empty then
        return conjoin(term,cnf2-aux(t:left),cnf2-aux(t:right))
   else
       select a variable v from separator s
      p = false                              /* positive case */
     if decide(v) then
         p =cnf2ddnnf(t)
     undo-decide(v)
     if p = false then
         if at-assertion-level() and assert-cd-literal() then
             return cnf2ddnnf(t)          /* try again */
         else
             return false                  /* backtracking */
      n = false                            /* negative case */
      if decide(:v) then
          n =cnf2ddnnf(t)
      undo-decide(:v)
      if n = false then
          if at-assertion-level() and assert-cd-literal() then
              return cnf2ddnnf(t)   /* try again */
          else
              return false /* backtracking */
      return conjoin(term,disjoin(conjoin(v,p),conjoin(:v,n)))

Algorithm  cnf2-aux(dtree t)
    if t is a leaf dtree then
        return a d-DNNF corresponding to current clause at t
   else
       compute key key of CNF corresponding to dtree t
       if cache(key) != null then
           return cache(key)
       else
           r = cnf2ddnnf(t)
           if r! = false then
               cache(key)=r
           return r
```

Noter qu'un séparateur v pour le nœud n de dtree peut être instancié avant que nous atteignions le nœud n dans le processus récursif de décomposition, ceci pour deux raisons :

- v peut appartenir au séparateur d'un ancêtre m de n et a pu donc avoir été instancié en décomposant m ;
- v a pu avoir été placé par résolution d'unité, qui est une partie standard de DPLL.

Considérons le dtree sur la figure 4.2 comme exemple. Supposons que nous commençons au nœud racine par placer la variable 4. En utilisant la résolution unitaire et la clause cl_2, nous pouvons conclure que la variable 1 doit être vraie. Les sous-arbres gauches

et droits de la racine sont maintenant décomposés puisque toutes ses variables de séparateur sont instanciées, ainsi la procédure de compilation s'exécute récursivement sur chacun indépendamment.

Un dtree peut être produit en utilisant une de deux méthodes discutées dans les articles de [Darwiche et Hopkins, 2001] et [Zabiyaka et Darwiche, 2007]:

Les résultats expérimentaux :

La première méthode transforme le CNF en hypergraphe et puis emploie la décomposition d'hypergraphe pour décomposer périodiquement la CNF.

La deuxième méthode pour produire un dtree est en utilisant trois types d'ordre variable d'élimination [Darwiche et Hopkins, 2001]:

- l'ordre d'élimination (1,2,…,n-1,n).
- l'ordre d'élimination (n, n-1,…,2,1).
- l'ordre d'élimination minfill.

Nous considérons maintenant les résultats expérimentaux qui illustrent l'efficacité de notre approche. Notre exécution est en C++ sur un processeur de 1.6GHz Intel, avec 2GB de RAM. Les résultats des tests sont résumés dans les tableaux 4.1 et 4.2.

Tab 4.1 : les résultats expérimentaux de transformation des CNF au d-DNNF avec la méthode d'hypergraphe.

Benchmarks	satisfiabilité	CNFs		dtree	d-DNNF		Temps (s)
		#Vars	#Clauses	#nœuds	#nœuds	#niveaux	
uf20	sat	20	91	181	44.8	72.6	0.131
Dubois	All insat	89.25	238	475	1	0	0.457
Aim-100-1_6-no	insat	100	160	319	1	0	0.828
Aim-100-1_6-yes	sat	100	160	319	101	100	0.813
uf250-041	sat	250	1065	2129	1526	3478	127.921
uf250-058	sat	250	1065	2129	1034	3849	249.484
sat-grid-pbl-0010	sat	110	191	381	1301	2504	1,092
sat-grid-pbl-0015	sat	240	436	871	9646	20804	2,281
sat-grid-pbl-0020	sat	420	781	1591	2432972	5363430	668,203
sat-grid-pbl-0025	sat	650	1226	2451	17093470	37905634	382,656
prob001.pddl	sat	939	3785	7569	2454	8016	12.781
prob002.pddl	sat	1337	24777	63713	15590	549558	59.593
c432	sat	196	514	1027	8615	25535	1.515

c7552	sat	3185	8588	17175	3621506	28114310	232.906
Ra	sat	1236	11416	22831	162603	1363064	41.375
Rb	sat	1854	11324	22647	603548	2556335	384.703
Bmc	sat	6247.5	33776.5	11873.5	418073	1451638.5	522.8195
Bf	All insat	1894.25	5998	11996	1	0	22.56225

Tab 4.2 : les résultats expérimentaux de transformation des CNF au d-DNNF avec la méthode d'ordre d'élimination.

Benchmarks	**satisfiabilité**	**CNFs**		**dtree**	**d-DNNF**		**Temps (s)**
		#Vars	#Clauses	#nœuds	#nœuds	#niveaux	
uf20	All sat	20	91	181	52.6	90.4	0.015
Dubois	All insat	89.25	238	475	1	0	0.015
Aim-100-1_6-no	Insat	100	160	319	1	0	0.015
Aim-100-1_6-yes	Sat	100	160	319	101	100	0.015
uf250-041	Sat	250	1065	2129	-	-	-
uf250-058	Sat	250	1065	2129	-	-	-
sat-grid-pbl-0010	Sat	110	191	381	1716	3223	0.015
sat-grid-pbl-0015	Sat	240	436	871	6039	11884	0.062
sat-grid-pbl-0020	Sat	420	781	1591	92093	188617	0.625
sat-grid-pbl-0025	Sat	650	1226	2451	1246181	2566931	12.109
prob001.pddl	Sat	939	3785	7569	3848	17599	0.281
prob002.pddl	Sat	1337	24777	63713	10828	402901	56.453
Ra	Sat	1236	11416	22831	124573	1296739	5.484
Rb	Sat	1854	11324	22647	272039	2410583	31.109
c432	Sat	196	514	1027	470151	2244487	5.765
c7552	Sat	3185	8588	17175	-	-	-
Bmc	Sat	6247.5	33776.5	11873.5	-	-	-
Bf	All insat	1894.25	5998	11996	1	0	0,12875

A partir des tableaux précédents, nous pouvons constater un certain nombre de points.

- Pour les deux méthodes et dés l'étape de compilation en d-DNNF, les tableaux permettent de constater que nous pouvons savoir la satisfiabilité. Lorsque le nombre des nœuds du d-DNNF des problèmes compilés est nul on peut dire que ses problèmes sont insatisfiables (comme les instances dubois, uf20 et bf).
- On constate aussi, quand la taille de CNF utilisé est faible les résultats des deux méthodes sont très proches de 0s.
- Mais pour des tailles grandes de CNF, la méthode de décomposition d'hypergraphe est plus performante que celle d'ordre d'élimination surtout avec les instances de bmc et uf250. L'inverse se produit lorsqu'on se trouve face à des instances sat-grid-pbl.

Ceci peut être expliqué par la présence des clauses unaires et binaires dans ces problèmes.

- On peut voir aussi que le nombre des nœuds de dtree est égal toujours à 2n-1 où n est nombre des clauses dans CNF. Cela se produit grâce à la définition de dtree. Nous rappelons que les nœuds de feuille dans un dtree doivent être dans la correspondance linéaire avec les clauses de CNF. Par conséquence, le dtree doit avoir n feuilles. D'ailleurs, un arbre binaire avec n feuilles doit avoir (n - 1) nœuds internes.

En général, l'étape de compilation ne prend pas du temps et elle est très performante [Darwiche et Hopkins, 2001].

Lorsque le d-DNNF f se produit, nous avons deux avantages :

- la 1^{ere} concernant la satisfiabilité ou pas du problème,
- la 2emes nous exécutons simplement un parcoure ascendant de f pour calculer les solutions (modèles).

Exemple : soit le d-DNNF spécifié au dtree de la figure 4.2 :

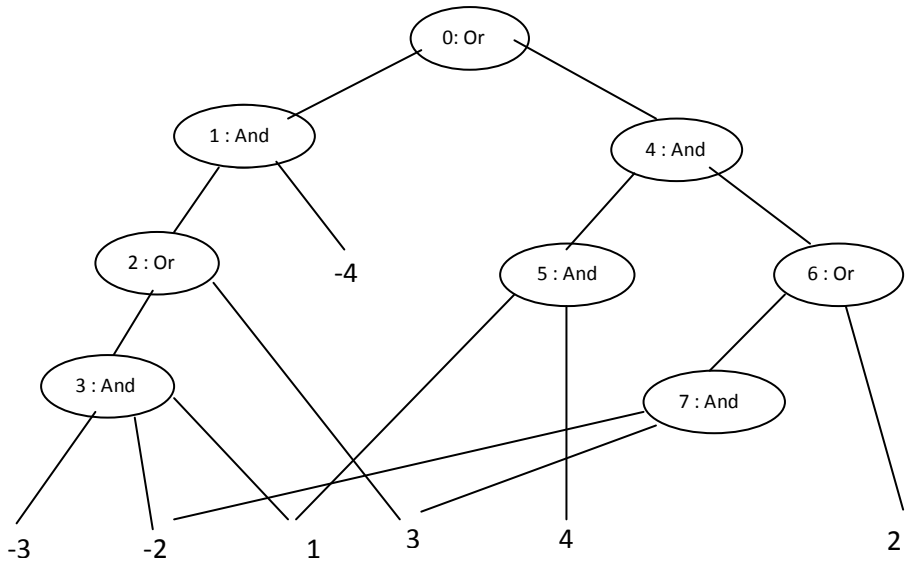

Fig 4.3 : d-DNNF correspond au CNF de la figure 4.2.

Le d-DNNF vérifie les propriétés de :

- Décomposition : l'intersection des variables de chaque nœud And indexé 1, 3, 4, 5 et 7 est vide.

- Déterministe : pour chaque nœud Or apparaissant dans le d-DNNF indexé 0, 2 et 6, les variables de chacune sont logiquement contradictoire, par exemple pour le nœud Or indexé 2 : -3 ∧ -2 ∧ 1 ∧ 3 $\vdash false$.

2. Calcul des modèles

L'algorithme ci-dessous décrit la méthode du calcule des modèles à partir d'une d-DNNF :

```
Algorithme calcule (d-DNNF f):


for i=1 à nbr_neouds do
   if (noeud_courant est une feuille ) then
      sauvegarder ce nœud dans la liste des feuilles
  else
      if (noeud_courant = « and » ) then
          incrémenter le nbr_des modèle
          for j=1 à nbr_lit associé à ce noeud do
                if(noeud_courant est une feuille) then
                      sauvegarder cette solution dans un tampon
                else
                      calcule (nœud_courant)
          end
         fusionner ces solutions comme une seul solution
      else
        if (noeud_courant = « or » ) then
            for j=1 à nbr_lit associé à ce noeud do
                if(noeud_courant est une feuille) then
                      sauvegarder cette solution
                      incrémenter le nbr_des modèle
                else
                      calcule (nœud_courant)
          end
end.
```

La complexité du calcul des modèles d'une DNNF F d'après [A.Darwiche ,2001] est O(mn), où m est la taille de F et $n=|Models(F)|^2$. Si le nombre des modèles est très petit, il tent vers une constante. Donc le processus d'énumération est devenu linéaire avec la taille de DNNF.

Maintenant, nous calculons les modèles des d-DNNF des tableaux précédents.

Benchmarks	satisfiabilité	d-DNNF			Nombres des modèles	Temps (s)
		#Vars	#nœuds	#niveaux		
uf20	sat	20	44.8	72.6	8	0.000
Dubois	All insat	89.25	1	0	-	-
Aim-100-1_6-no	insat	100	1	0	-	-
Aim-100-1_6-yes	sat	100	101	100	1	0,001
uf250-041	sat	250	1526	3478	112880008	0.015
uf250-058	sat	250	1034	3849	127350656	0.016
sat-grid-pbl-0010	sat	110	1301	2504	593962746002226256572855	0.000
sat-grid-pbl-0015	sat	240	6039	11884	3012964503482414330783936367006634039953704207876657607	0,03
sat-grid-pbl-0020	sat	420	92093	188617	505529009203800755681036 389586231130590495862311305904953769300615987441885918031770076271649889444375607267 19	0.573
prob001.pddl	sat	939	2288	5630	564153552511417968750	0.015
prob002.pddl	sat	1337	15590	549558	32334741710	0.003
c432	sat	196	3508	13408	68719476736	0.016
c7552	sat	3185	3621506	28114310	205688069665150755269371147819668813122841983204197482918576128	81.932
Ra	sat	1236	162603	1363064	Nombre important	2.433
Rb	sat	1854	603548	2556335	Nombre important	6.146
Bf	All insat	1894.25	1	0	-	-

Tab 4.3 : les résultats expérimentaux du calcule des modèles des CNF qui sont compilés au d-DNNF.

D'après ce tableau, on peut citer les remarques suivantes :

- Les benchmarks non satisfiables ne contiennent aucun modèle comme le cas de aim-100-1_6-no et toutes les instances de Dubois et Bf ;
- Il y a des benchmarks qui possèdent une seule solution ou un nombre petit comme aim-100-1_6-yes qui possède un seul modèle et aussi le cas des instances de uf-20 qui possèdent huit modèles.
- Et il y a des benchmarks qui possèdent un nombre important des modèles comme le cas des sat-grid, uf-250, c432, c7552 et prob.
- Les instances comme Ra et Rb possèdent un grands nombre de modèles(milliards de milliards).

- Et dans tout les cas, le calcule est fait dans un temps petit ne dépassant pas quelques minutes.

Conclusion

Dans ce chapitre nous avons présenté notre principale contribution dans ce travail. Nous avons présenté de façon formelle la méthode de résolution en utilisant la compilation en d-DNNF. Nous avons prouvé sa faculté d'être appliquée à des problèmes de grande taille car elle se base sur des algorithmes polynomiaux.

La compilation au d-DNNF est utilisé pour avoir une structure très simple et des algorithmes de manipulation simple et linéaire avec la représentation compilée.

Conclusion et perspectives

L'utilisation de la recherche basé sur DPLL pour la résolution du SAT, combinée à des techniques rétrospectives tel que le backtracking intelligent, des techniques prospectives tel que l'exploitation d'heuristiques sur le choix des variables, obtient en général des résultats pratiques satisfaisants sur une large palette de problèmes. Néanmoins, la complexité théorique d'une telle approche est exponentielle en taille de problème et ceci qu'il s'agisse de la complexité temporelle ou de la complexité spatiale. Ce constat a motivé le développement des méthodes de résolution basée sur la structure du problème, qui permettent de borner la complexité théorique.

Donc, l'étude des représentations de formules SAT a été un sujet central pour la résolution SAT surtout s'il a combinée à des techniques de transformation, il obtient en général des résultats pratiques satisfaisants sur une large palette de problèmes. la transformation consiste à convertir une formule donnée d'une représentation en des autres de tel sorte que certaines tâches de raisonnement comme la satisfiabilité deviennent traitable sur la nouvelle représentation.

La représentation d-DNNF est l'une de ces représentations les plus intéressantes, notamment parce qu'elle généralise une bonne partie des représentations existantes en plus de l'existence de méthodes tractables qui peuvent calculer les toutes solutions dans un temps optimale, ce qui a orienté les recherches ces dernières années vers le développement de méthodes basées sur cette représentation.

Pour contribuer à la résolution des problèmes SAT en utilisant d-DNNF, nous avons proposé une méthode dite compile & SAT. Notre méthode est composée de 2 phases :

· L'extraction des problèmes sous forme d-DNNF.

· La résolution des problèmes SAT sous cette nouvelle forme avec le calcul des solutions.

Les expérimentations de compile & SAT ont montré l'intérêt pratique de notre méthode qui donne des résultats meilleurs que ceux des méthodes basées sur CNF ou OBDD.

Les résultats de l'utilisation de l'approche de résolution séquentielle par d-DNNF sont à la fois encourageants et non satisfaisants. Encourageants car ils montrent bien qu'après la phase de compilation, constituée des décompositions et de filtrage, la recherche d'une solution se fait extrêmement rapidement. Mais non satisfaisants car le coût de compilation est encore important quand ils s'agit de problèmes dont la taille des relations est importante même avec l'utilisation d'une décomposition par dtree ou hypergraphe, ce qui nous amène aux perspectives à court terme suivantes :

- La parallèlisation de la résolution.
- La prise en compte d'autres paramètres dans construction de d-DNNF tel que la propriété d'uniformité de la représentation DNNF.
- La recherche de nouvelles représentations pour la représentation des problèmes SAT plus générale et plus efficace que d-DNNF.

Bibliographie

[A.Dar wiche et al,2004] Jinbo Huang and Adnan Darwiche. Using DPLL for Efficient OBDD Construction, Computer Science Department University of California, Los Angeles, 2004.

[A.Darwiche ,2001] A. Darwiche.Decomposable Negation Normal Form. Journal of the ACM, 2001.

[Darwiche et al., 2005] Hector Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS), 2005.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. Journal of Artificial Intelligence Research, 17:229–264, 2002.

[BAZGAN,2008] Cristina BAZGAN. Complexité des Problèmes de Satisfaction de Contraintes Booléennes, *Random Structures and Algorithms*, vol. 23(1), p. 1-31,2008.

[Cook,1971] A. COOK. « The complexity of theorem-proving procedures ». Dans proceedings of the Third Annual ACM Symposium on Theory of Computing, pages 151–158, New York (USA), 1971.

[Dalal, 1996] Dalal, M. (1996). An almost quadratic class of satisfiability problems. In ECAI'96: Proceedings of the Twelfth European Conference on Artificial Intelligence, pages 355–359.

[Daniel et al, 2003] Gilles Audemard Daniel Le Berre Olivier Roussel. OpenSAT : Une plate-forme SAT Open Source, 2003.

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

[Davis & Putnam 1960] Martin DAVIS et Hilary PUTNAM. « A Computing Procedure for Quantification Theory ». Journal of the Association for Computing Machinery, 7:201–215, 1960.

[Goldberg et Novikov, 2002] Evgueni Goldberg Yakov Novikov. BerkMin: a Fast and Robust Sat-Solver, IEEE Transactions of Computers (DATE.02).

[Garey et al,79] M.R. Garey and D.S. Johnson. Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, 1979.

[Jeroslow, 90] R. E. Jeroslow and J. Wang. Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence, 1:167{187, 1990.

[Marques-Silva, 1999] Marques-Silva, J.P., "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), September 1999.

[J. P. M. Silva and K. A, 96] J. P. M. Silva and K. A. Sakallah. Grasp : a new search algorithm for satisfiability. In Proceedings of the International Conference on Computer-Aided Design, November 1996, 1996

[J. W. Freeman, 1995] J. W. Freeman, "Improvements to Propositional Satisfiability Search Algorithms," in Ph.D. Thesis, Department of Computer and Information Science: University of Pennsylvania, 1995.

[Hooker et al; 95] J.N. Hooker and V. Vinay. Branching Rules for Satisfiability. Journal of Automated Reasoning, 15:359{383, 1995.

[H. Zhang et al, 1996] H. Zhang and M. Stickel, "An efficient algorithm for unit-propagation," presented at International Symposium on Artificial Intelligence and Mathematics, Ft. Lauderdale, Florida, 1996.

[Hoos et al,1999] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for sat. In Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99), pages 661–666, 1999.

[Hoos,2000] Holger H.Hoos and Thomas Stützle. Local search algorithms for SAT: An empirical evaluation. Journal of Automated Reasoning, 24(4):421– 481, 2000.

[Bennaceur et Li, 2002] Hachemi Bennaceur, Chu Min Li. Caractérisation de problèmes SAT à laide de la propriété de la convexité par ligne. *Actes JNPC'02*.

[Kowalski & Kuehner 1971] R.A. KOWALSKI et D. KUEHNER. « Linear Resolution with Selection Function ». Artificial Intelligence, 2:227–260, 1971.

[Kautz et al, 92] H. Kautz and B. Selman. Planning as Satisfiability. In Proceedings of the10th ECAI, pages 359{363. European Conference on Artificial Intelligence,1992.

[K. L. McMillan,2002] K. L. McMillan. Applying SAT methods in Unbounded Symbolic Model Checking. Cadence Berkeley Labs, 2002.

[L. VonNorden, 2006] VAN NORDEN, Operations Research based approaches for the (maximum) satisfiability problem,2006

[Li, Anbulagan,1997] C. M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems. In 15th Int. Joint Conference on AI, IJCAI'97, Morgan Kaufmann Pub., Nagoya Japon, pp.366-371, 1997.

[Li et al , 1999] Djamal Habet,Chu Min Li, Laure Devendeville, Michel Vasquez et Jean-Lus Guérin. Approche hybride pour SAT, Actes JNPC'02, pages 115-126.

[L.PARIS, 2007] Lionel PARIS. Approches pour les problèmes SAT et CSP : ensembles strong backdoor, voisinage consistant et forme normale généralisée, THESE présentée pour obtenir le grade de DOCTEUR DE L'UNIVERSIT´E DE PROVENCE Spécialiée : Informatique,2007

[Leslie G. Valiant,1979] Leslie G. Valiant. The complexity of computing the permanent. Theoretical Computer Science, 8:189–201, 1979.

[Marquis1995] Pierre MARQUIS. « Knowledge compilation using theory prime implicates ». Dans Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95), pages 837–843, 1995.

[Matti,2008] Matti Jarvisalo. Structure-Based Satisfiability checking, Analyzing and Harnessing the Potential. TKK Dissertations in Information and Computer Science, 2008.

[Moskewicz et al ,2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, Proc. of the 39th. DAC, Las Vegas USA, 2001.

[M. Böehm, E. Speckenmeyer, 1994] M. B¨oehm, E. Speckenmeyer, A fast Parallel Sat Solver - efficient work load balancing, In Third Int. Symp. on AI and Maths. AIMSA, Fort Lauderdale, Florida USA, 1994.

[Olivier, 2005] Olivier Fourdrinoy. Hybridation des méthodes de résolution pour SAT, Centre de Recherche en Informatique de Lens, CNRS FRE 2499 Université d'Artois. 2005

[O. Dubois, 2001] O. Dubois and G. Dequen, "A backbone-search heuristic for efficient solving of hard 3-SAT formulae," presented at International Joint Conference on Artificial Intelligence (IJCAI), 2001.

[Robinson 1965] J.A. ROBINSON. « A machine-oriented logic based on the resolution principle ». Journal of the Association for Computing Machinery, 12:23–41, 1965.

[R.COWEN&K.WYATT, 1993] ROBERT COWEN and KATHERINE WYATT.BREAKUP: A reprocessing Algorithm for Satisfiability Testing of CNF Formulas, Notre Dame Journal of Formal Logic Volume 34, Number 4, Fall 1993, pages 602-606, 1993.

[R. Werchner et al, 96] Ralph Werchner, Thilo Harich , Rolf Drechsler and Bernd Becker. Satisfiability Problems for Ordered Functional Decision Diagrams, 1996.

[Reiter et al,89] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. Artificial Intelligence, 42(2):125{155, 1989.

[Selman & Kautz, 1991] Bart SELMAN et Henry A. KAUTZ. « Knowledge compilation using Horn approximations ». Dans Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91), pages 904–909, 1991.

[SAÏS,2000] Lakhdar SAÏS, De la résolution du problème SAT a la résolution du problème autour de SAT, mémoire d'habilitation à diriger des recherches, université d'Artois, 2000.

[Selman et al. 1994] Selman, B., Kautz, H. A., et Cohen, B. (1994). Noise Strategies for Improving Local Search. In press, M., editor, Proceedings of the 12th National Conference on Artificial Intelligence AAAI'94, volume 1, pages 337–343.

[Selman et al. 1992] Bart SELMAN, Hector J. LEVESQUE, et David MITCHELL. « GSAT: A New Method for Solving Hard Satisfiability Problems ». Dans Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92), pages 440–446, 1992.

[P.SIMON, 2005] SIMON PAQUETTE. Evaluation symbolique de systèmes probabilistes à espace d'états continu. Mémoire présenté à la Faculté des études supérieures de l'Université Laval dans le cadre du programme de maitrise en informatique pour l'obtention du grade de maitre ès sciences (M.Sc.),2005.

[Steven et al,2006] Steven D. Prestwich and Inês Lynce. Local search for unsatisfiability. In Armin Biere and Carla P. Gomes, editors, Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06),volume 4121 of Lecture Notes in Computer Science, pages 283–296. Springer, 2006

[Singer, 2006] Daniel SINGER. Parallel Resolution of the Satisfiability Problem: A Survey, Publications du LITA, N◦ 2007-101 Service de reprographie de l'U.F.R. M.I.M. Université Paul Verlaine – Metz, 2007.

[Toby Walsh,2000] Toby Walsh. SAT v CSP. In Rina Dechter, editor, Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00), volume 1894 of Lecture Notes in Computer Science, pages 441–456. Springer, 2000.

[Zhaohui et al,2006] Zhaohui Fu, Yogesh Marhajan, Sharad Malik. zChaff SAT Solver. Princeton University.Princeton, NJ 08544, USA.2006

[Zabiyaka et Darwiche, 2007] Yuliya Zabiyaka and Adnan Darwiche. On Tractability and Hypertree Width. University of California, Los Angeles Los Angeles, CA 90095-1596, USA,2007.