UNIVERSITÉ DE PROVENCE
U.F.R. M.I.M.
ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE E.D. 184

THÈSE

présentée pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE PROVENCE

Spécialité : Informatique

par

Lionel PARIS

Titre:

Approches pour les problèmes SAT et CSP : ensembles strong backdoor, voisinage consistant et forme normale généralisée

soutenue publiquement le 09 novembre 2007

JURY

M. Belaïd BENHAMOU	Université de Provence	co-directeur
M. Hachemi BENNACEUR	Université d'Artois	Examinateur
M. Jin-Kao HAO	Université d'Angers	Examinateur
M. Chu Min LI	Université de Picardie Jules Vernes	Rapporteur
M. Pierre SIEGEL	Université de Provence	co-directeur
M. Toby WALSH	University of New South Wales	Rapporteur

Remerciements :

Une page ne suffirait pas pour exprimer ma gratitude à tous ceux qui ont contribué, de près ou de loin, à l'élaboration de cette thèse. Je tiens néanmoins à remercier ici certaines personnes sans qui ce travail n'aurait pas été ce qu'il est, en espérant n'oublier personne.

Je tiens tout d'abord à remercier messieurs Chu Min Li et Toby Walsh pour avoir bien voulu être les rapporteurs de ma thèse, pour le temps qu'ils y ont consacré, pour leurs précieux conseils qui m'ont permis d'améliorer mon manuscrit et pour m'avoir fait l'honneur de leur présence en tant que membre du jury.

Je remercie également messieurs Hachemi Bennaceur et Jin-Kao Hao d'avoir accepté de faire parti du jury en temps qu'examinateur de ma thèse.

J'exprime également ma gratitude envers mes directeurs de thèse Belaïd Benhamou et Pierre Siegel pour avoir accepté de me diriger pendant ces trois années de préparation. Je les remercie de m'avoir initié à la recherche, pour les conseils qu'ils ont su me prodiguer tout au long de mes travaux de recherche, et pour tout ce qu'ils ont fait pour moi.

Je suis également profondément reconnaissant envers Richard Ostrowski, en qui j'ai trouvé bien plus qu'un collaborateur. Merci Richard de m'avoir appris ce qu'était la programmation, de m'avoir aidé et accompagné dans la plupart de mes travaux, et de m'avoir soutenu dans les moments de doutes.

Un grand merci également à toute l'équipe INCA pour la bonne ambiance qui m'a permis de toujours me sentir à l'aise. Merci à Djamal Habet d'avoir accepté de travailler avec moi, merci à Philippe Jégou pour les discussions scientifiques que nous avons eu, merci à Cyril Terrioux de s'être si bien occupé de mon ordinateur, et merci à tous les autres pour leur sympathie.

Je tiens aussi à remercier Lakhdar Saïs pour notre fructueuse collaboration, ainsi que Bruno Zanuttini pour l'intérêt qu'il a montré pour mes travaux et les idées qu'il m'a suggérées.

Je voudrais remercier mes compagnons mathématiciens Clément Marteau et Sébastien Loustau pour l'ambiance à la fois studieuse et décontractée qui régnait chaque jour dans notre bureau, qui m'a donné la motivation nécessaire pour mener à bien cette thèse. Enfin, je ne peux oublier de remercier sincèrement ma famille et mes proches qui ont dû se passer de moi à de si nombreuses reprises quand j'étais trop occupé par ma thèse.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Introduction générale

Le problème de satisfaisabilité de formules propositionnelles sous forme normale conjonctive (SAT) est depuis un grand nombre d'années un problème central dans bon nombre de domaines dont l'intelligence artificielle. Si ce problème a suscité autant d'intérêt, c'est d'abord parce qu'il a été le premier problème démontré comme étant NP-complet, faisant de lui le problème de référence de la théorie de la complexité. De plus, sa simplicité d'appréhension et la robustesse de ses méthodes de résolutions en font un outil de choix pour un grand nombre d'applications pratiques. Parmi ces applications, on peut citer notamment les problèmes liés à l'ordonnancement, la vérification de circuits, l'allocation de ressources, des problèmes de diagnostic etc... Dans la quantité de solveurs SAT existants, on distingue généralement deux types de solveurs : les solveurs complets et les solveurs incomplets.

La plus grande majorité des méthodes complètes sont basées sur la procédure énumérative DPLL. Cette méthode basique est couplée généralement à plusieurs méthodes d'apprentissage et de filtrage importantes comme des formes étendues de propagation de contraintes booléennes, de détection de symétries, etc. Ces dernières années, un nombre important de solveurs capables de traiter des instances de plus en plus grandes ont été proposés. En plus de ces méthodes de simplifications, de plus en plus de pré-traitements y sont intégrés, afin d'identifier et de traiter certaines structures contenues dans les instances. C'est le cas des chaînes d'équivalences ou toutes autres dépendances fonctionnelles qui ont fait et font l'objet de travaux ces dernières années. Plus récemment, une forme particulière de structure a été identifiée et commence à faire l'objet de recherches. Il s'agit de l'identification d'ensembles strong backdoor. Un ensemble strong backdoor est un ensemble de variables tel qu'il résulte de toute interprétation de ces variables une formule simplifiée pouvant être traitée en temps polynomial. La taille des ensembles strong backdoor est directement liée à la complexité intrinsèque de chaque instance. Ceci explique que le calcul d'ensemble strong backdoor de taille minimale est NP-difficile. Les seules méthodes connues à l'heure actuelle souffrent par conséquent d'une très forte et rapide explosion combinatoire rendant leur utilisation impossible dès lors que les problèmes contiennent plus d'une cinquantaine de variables.

Dans cette thèse nous nous sommes intéressés dans un premier temps à la détection et à l'exploitation d'ensembles strong backdoor. Notre but est de rendre praticable le calcul de ces ensembles dans le cas général, y compris pour des instances de grande taille. Pour ce faire, nous proposons une méthode approchée qui calcule des ensembles strong backdoor pour plusieurs classes polynomiales dont la taille est la plus petite possible. Cette méthode passe par le calcul du meilleur renommage d'une formule en fonction d'une classe polynomiale donnée. Ce problème étant NP-difficile, nous fournissons un algorithme d'approximation de ce renommage basé sur l'exploitation de l'algorithme incomplet WalkSat. Les expérimentations que nous avons menées montrent que

notre méthode de calcul est efficace et permet de trouver en général des ensembles strong backdoor de taille intéressante.

Ces ensembles strong backdoor ont ensuite été utilisés lors de la résolution complète d'instances SAT. Le solveur exploitant ces ensembles que nous proposons présente la caractéristique avantageuse d'avoir une complexité exponentielle dans la taille de l'ensemble strong backdoor au lieu du nombre total de variables du problème.

Cependant, certaines instances du problème SAT restent intraitables par la plupart des solveurs complets. Lorsque cette incapacité est due à la taille vraiment trop importante des instances, une alternative consiste à utiliser des méthodes incomplètes. Ces méthodes ont l'avantage de pouvoir traiter des instances de très grande taille, mais ont aussi, comme leur nom l'indique, l'inconvénient de ne pas pouvoir répondre dans tous les cas. La plupart des méthodes incomplètes sont basées sur des méthodes de recherche locale pour lesquelles l'espace de recherche est exploré de manière non systématique. Ces méthodes ont montré des performances surprenantes, notamment sur des instances générées aléatoirement. Généralement, ces méthodes commencent par générer aléatoirement une interprétation complète inconsistante et tentent de la réparer en y effectuant des modifications mineures.

Dans cette thèse, nous apportons également une contribution concernant les méthodes incomplètes. Nous proposons une nouvelle méthode de recherche locale pour le problème SAT dans laquelle nous ne manipulons que des interprétations partielles mais toujours consistantes. Contrairement aux méthodes de recherche locale classiques, notre solveur tente d'interpréter un maximum de variables de manière consistante au lieux de satisfaire un maximum de clauses. Lors du choix de la prochaine variable à instancier, notre méthode est amenée à calculer un ensemble de variables de taille minimale à libérer pour maintenir la consistance. Ce calcul se ramène au calcul d'un transversal de taille minimale dans un hyper-graphe, qui est un problème NP-difficile. Nous proposons deux méthodes pour effectuer ce calcul, une approchée et une exacte, donnant lieu à deux versions de notre solveur.

Le formalisme des problèmes de satisfaction de contraintes (CSP) est un autre formalisme largement utilisé pour représenter et traiter des problèmes en intelligence artificielle. Il est beaucoup plus structuré que le formalisme SAT. Cette structure permet de représenter n'importe quel CSP sous forme de graphe ou d'hyper-graphe de contraintes et donne lieu à la définition de plusieurs propriétés de consistances partielles. Dans le formalisme CSP, il existe une sorte de barrière pratique entre les CSP binaires (toutes les contraintes n'impliquent que deux variables) et les CSP n-aires (contraintes quelconques). En effet, même si dans la théorie presque toutes les propriétés et méthodes des CSP binaires peuvent être généralisées aux CSP n-aires, en pratique on constate que cette généralisation pose quelques problèmes. Cette différence est cependant en train de s'amenuiser ces dernières années, notamment grâce à l'utilisation de contraintes dites globales.

Les formalismes SAT et CSP n'en restent pas moins très proches, puisqu'il existe plusieurs façons de coder une instance CSP en SAT et vice-versa. On est donc en droit de se poser la question de l'existence d'un formalisme général qui permettrait d'exprimer simplement une instance SAT ou CSP et qui pourrait tirer profit des avantages de ces deux formalismes.

Le formalisme CGNF (Cardinality General Normal Form) que nous introduisons dans cette thèse pourrait être une réponse à cette question. En effet, nous montrons qu'il est capable d'expri-

mer et de traiter n'importe quelle instance SAT ou CSP exprimée en extension. S'agissant d'un formalisme logique, nous montrons comment la méthode de calcul des résolvantes peut être étendue et utilisée pour résoudre des instances CGNF. Nous montrons également que dans le cas particulier où des CSP sont codés en CGNF, nous pouvons définir une règle d'inférence qui permet de filtrer les instances de la même manière que le ferait des filtrages par consistances partielles sur leur formulation CSP. Plus précisément, nous montrons que la consistance d'arc et la consistance de chemin (dans le cas de CSP binaires) peuvent être établies dans le codage CGNF avec la même complexité que dans les CSP. Enfin nous verrons que l'adaptation de la procédure DPLL à ce formalisme, combinée à l'exploitation de certaines règles d'inférences, permet de retrouver les algorithmes Forward Checking (FC) et Maintaining Arc Consistency (MAC).

Ce mémoire est composé de trois grandes parties. La première contient un état de l'art que nous dressons pour situer le point de départ de nos travaux. Cette étude bibliographique comprend les définitions des problèmes SAT et CSP, ainsi qu'une présentation des transformations les plus connues entre ces deux formalismes. La seconde partie de ce mémoire est consacrée au problème SAT et présente nos deux premières contributions : nous y présentons tout d'abord notre méthode pour calculer des ensembles strong backdoor ainsi qu'une utilisation de ces ensembles pour la résolution d'instances SAT. Puis nous présentons la nouvelle méthode de recherche locale pour la résolution du problème SAT. Enfin, la troisième partie contient notre dernière contribution : la présentation du formalisme CGNF.

Première partie

La logique propositionnelle, le problème SAT et les problèmes de satisfaction de contraintes

Chapitre 1

La logique propositionnelle et le problème SAT

Sommaire

Ce premier chapitre est consacré au rappel des définitions du formalisme de la logique propositionnelle et du problème SAT. Dans un premier temps, nous introduisons les notions techniques de logique propositionnelle permettant de caractériser le problème SAT. Après une description d'un certain nombre de classes polynomiales, nous présentons les ensembles backdoor et strong backdoor et les méthodes connues pour les calculs. Enfin, nous présentons brièvement les différentes méthodes de résolution du problème SAT.

1.1 La logique propositionnelle

1.1.1 Syntaxe

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

Définition 2 (formule propositionnelle)
Soit un alphabet constitué d'un ensemble de variables propositionnelles et de deux symboles particuliers $\top$ *et* $\bot$. *Soient les connecteurs :*

- *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 parenthèsage « (» et «) ».

Une formule propositionnelle Σ *se construit récursivement en appliquant un nombre fini de fois les règles suivantes :*

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

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. Plus précisément, pour une formule Σ*, nous nous limiterons même à l'ensemble fini des variables qui apparaissent au moins une fois dans* Σ. *Cette restriction sera notamment appliquée lorsque nous parlerons des interprétations.*

Définition 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.

1.1.2 Sémantique

Interprétation d'une formule

Définition 4 (Interprétation)
Une interprétation $\mathcal{I}$ *en calcul propositionnel est une application de l'ensemble des formules propositionnelles dans l'ensemble des valeurs de vérité* $\{$*VRAI ,FAUX* $\}$. *L'interprétation* $\mathcal{I}$*(*Σ*) d'une formule* Σ *est définie par la valeur de vérité donnée à chacune des variables de* Σ. *Cette interprétation se calcule par l'intermédiaire des règles suivantes :*

1. $\mathcal{I}$*(*$\top$*)=VRAI ;*

$\mathcal{I}(\Sigma)$	$\mathcal{I}(\Phi)$	$\mathcal{I}(\neg\Sigma)$	$\mathcal{I}(\Sigma\wedge\Phi)$	$\mathcal{I}(\Sigma\vee\Phi)$	$\mathcal{I}(\Sigma\rightarrow\Phi)$	$\mathcal{I}(\Sigma\leftrightarrow\Phi)$
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

TAB. 1.1 – Table de vérité d'une formule

2. *$\mathcal{I}(\bot)$=FAUX ;*
3. *$\mathcal{I}(\neg\Sigma)$=VRAI si et seulement si $\mathcal{I}(\Sigma)$=FAUX ;*
4. *$\mathcal{I}(\Sigma\wedge\Phi)$=VRAI si et seulement si $\mathcal{I}(\Sigma)$=$\mathcal{I}(\Phi)$=VRAI ;*
5. *$\mathcal{I}(\Sigma\vee\Phi)$=FAUX si et seulement si $\mathcal{I}(\Sigma)$=$\mathcal{I}(\Phi)$=FAUX ;*
6. *$\mathcal{I}(\Sigma\rightarrow\Phi)$=FAUX si et seulement si $\mathcal{I}(\Sigma)$=VRAI et $\mathcal{I}(\Phi)$=FAUX ;*
7. *$\mathcal{I}(\Sigma\leftrightarrow\Phi)$=VRAI si et seulement si $\mathcal{I}(\Sigma)$=$\mathcal{I}(\Phi)$.*

La tableau 1.1 synthétise les différentes valeurs de vérité des formules $(\neg\Sigma)$, $(\Sigma\wedge\Phi)$, $(\Sigma\vee\Phi)$, $(\Sigma\rightarrow\Phi)$ et $(\Sigma\leftrightarrow\Phi)$ en fonction des valeurs de Σ et Φ.

Remarque 1 *De la table de vérité du tableau 1.1, il est possible de déduire que :*
- *la formule $(\Sigma\rightarrow\Phi)$ est équivalente à $(\neg\Sigma\vee\Phi)$;*
- *la formule $(\Sigma\leftrightarrow\Phi)$ s'écrit également $((\Sigma\rightarrow\Phi)\wedge(\Phi\rightarrow\Sigma))$ ou encore $((\neg\Sigma\vee\Phi)\wedge(\neg\Phi\vee\Sigma))$;*
- *le « ou exclusif »(exprimant la « différence »entre deux formules) noté $(\Sigma\oplus\Phi)$ s'écrit également $\neg(\Sigma\leftrightarrow\Phi)$ ou encore $((\neg\Sigma\wedge\Phi)\vee(\neg\Phi\wedge\Sigma))$;*
- *Soit $S_{\mathcal{I}(\Sigma)}$ l'ensemble des interprétations d'une formule Σ. Si Σ possède exactement n variables différentes alors $|S_{\mathcal{I}(\Sigma)}|=2^n$.*

Remarque 2 *En calcul propositionnel, pour décrire une interprétation $\mathcal{I}$ d'une formule, il suffit de connaître la valeur de vérité qu'elle donne à toute variable v de la formule sachant que, soit le littéral v, soit le littéral $\neg v$ est vrai dans $\mathcal{I}$. De manière naturelle on peut caractériser une interprétation $\mathcal{I}$ par l'ensemble des variables qu'elle interprète à vrai. Par exemple pour la formule $(a\wedge b)\rightarrow(c\leftrightarrow d)$ et l'interprétation $\mathcal{I}(a)$=V, $\mathcal{I}(b)$=F, $\mathcal{I}(c)$=V, $\mathcal{I}(d)$=F, $\mathcal{I}$ est notée $\{a,c\}$. Malheureusement cette notation ne permet pas de distinguer les variables n'ayant pas de valeur de vérité « définie »comme dans le cas des interprétations incomplètes. Nous représentons donc une interprétation $\mathcal{I}$ par l'ensemble des littéraux vrais pour l'interprétation. Sur l'exemple précédent, $\mathcal{I}$ sera représenté par l'ensemble $\{a,\neg b,c,\neg d\}$.*

Définition 5 (Interprétation partielle, incomplète et complète)
Soit Σ une formule propositionnelle dont le nombre de variables est égal à n. On dit que :
- *I est une* **interprétation partielle** *de Σ si et seulement si I est une interprétation et $|I|\leqslant n$;*
- *I est une* **interprétation incomplète** *de Σ si et seulement si I est une interprétation et $|I|<n$;*
- *I est une* **interprétation complète** *de Σ si et seulement si I est une interprétation et $|I|=n$.*

Consistance, inconsistance, validité et invalidité d'une formule

Définition 6 (formule satisfaite)
On dit qu'une formule propositionnelle Σ est **satisfaite** *par une interprétation $\mathcal{I}$ si et seulement si $\mathcal{I}(\Sigma)$=V.*

Définition 7 (formule falsifiée)
On dit qu'une formule propositionnelle Σ est **falsifiée** *par une interprétation $\mathcal{I}$ si et seulement si $\mathcal{I}(\Sigma)$=F.*

Définition 8 (modèle)
On appelle **modèle** *d'une formule Σ, une interprétation $\mathcal{I}$ qui satisfait Σ.*

Définition 9 (contre-modèle)
On appelle **contre-modèle** *d'une formule Σ, une interprétation $\mathcal{I}$ qui falsifie Σ.*

Définition 10 (consistance, inconsistance)
Une formule est dite **consistante** *ou satisfiable ou satisfaisable si et seulement si elle admet au moins un modèle. Une formule est dite* **inconsistante** *ou* **insatisfaisable** *si et seulement si elle n'admet pas de modèle, c'est-à-dire : $\forall \mathcal{I} \in S_{\mathcal{I}(\Sigma)}\ \mathcal{I}(\Sigma)$=F.*

Conséquence sémantique

Définition 11 (Conséquence sémantique ou conséquence logique)
Une formule Σ **implique sémantiquement** *une formule Φ notée $\Sigma \models \Phi$, si et seulement si tout modèle de Σ est un modèle de Φ. On dit alors que Σ a pour* **conséquence logique** *Φ ou encore que Φ est une* **conséquence logique de** *Σ.*

Définition 12 (équivalence sémantique)
On dit que deux formules Σ et Φ sont **sémantiquement équivalentes**, *noté $\Sigma \equiv \Phi$ si et seulement si $\Sigma \models \Phi$ et $\Phi \models \Sigma$, c'est à dire si ces deux formules admettent exactement les mêmes modèles.*

Clauses, monômes et formes normales

Clauses et monômes

Définition 13 (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.*

Définition 14 (monôme)
Dualement, on appelle **monôme** *une formule constituée d'une conjonction finie de littéraux. Un monôme est donc satisfait par une interprétation si tous ses littéraux sont vrais dans cette interprétation.*

Dans un souci de synthèse et de clarté, nous considérerons très souvent les clauses (ou les monômes) comme des ensembles de littéraux[1] et en conséquence nous nous autorisons à appliquer aux clauses (ou monômes) tous les opérateurs ensemblistes.

[1] Dans la pratique, ces ensembles seront même représentés par des listes sans répétition.

Définition 15 (clause et monôme fondamentaux)
On appelle **clause fondamentale** *(respectivement* **monôme fondamental***), une clause (respectivement un monôme) qui ne contient pas de littéral complémentaire.*

Définition 16 (clause tautologique)
On appelle clause **clause tautologique** *une clause qui contient au moins deux littéraux complémentaires.*

Définition 17 (clause positive, négative, mixte)
On appelle clause **positive** *(respectivement* **négative***) une clause qui ne contient pas de littéral négatif (respectivement positif). Une clause constituée à la fois de littéraux positifs et négatifs est appelée* **clause mixte**.

Définition 18 (longueur d'une clause ou d'un monôme)
On appelle **longueur de la clause** c *(respectivement du monôme* m*), le nombre de littéraux différents dans* c *(respectivement* m*). Cette longueur sera notée* $lg(c)$ *(respectivement* $lg(m)$*).*

Définition 19 (clause unaire ou unitaire)
Une **clause** c **unaire** *(ou unitaire ou mono-littérale) est une clause telle que* $lg(c) = 1$.

Définition 20 (clause n-aire)
Une clause **n-aire** *est une clause telle que* $lg(c) = n$.

Définition 21 (clause vide)
Une clause est **vide** *si* $lg(c) = 0$*. Elle sera notée* $\perp$ *ou* $\square$.

Définition 22 (clause de Horn)
Une clause de **Horn** *est une clause qui contient au plus un littéral positif.*

Note 1 *Dualement, une clause contenant au plus un littéral négatif sera appelée* **clause reverse-Horn**.

Exemple 1 (Exemples de clauses de Horn) $\{\neg a, \neg b, c, \neg d\}$ *est une clause de Horn.* $\{a, b, c, \neg d\}$ *est une clause reverse-Horn. Les clauses négatives sont des clauses de Horn. Les clauses positives sont des clauses reverse-Horn.*

Définition 23 (sous-sommation ou subsumption)
La clause c_1 **sous-somme** *ou* **subsume** *la clause* c_2 *si et seulement si* $c_1 \subseteq c_2$.

Définition 24 (résolvante)
Deux clauses c_1 *et* c_2 *se résolvent si et seulement si il existe un littéral* l *tel que* $l \in c_1$ *et* $\sim l \in c_2$*. La clause* $((c_1) \setminus \{l\} \cup (c_2) \setminus \{\sim l\})$ *est appelée* **résolvante** *de* c_1 *et* c_2 *en* l*. Il est évident que cette résolvante est logiquement impliquée par les clauses* c_1 *et* c_2.

Impliqués et impliquants

Définition 25 (impliqué)
Soient Σ une formule et c une clause. On dit que c est un impliqué de Σ si et seulement si c est une conséquence logique de Σ. Ceci est équivalent à dire que Σ implique c.

Définition 26 (impliquant)
Soient Σ une formule et m un monôme. On dit que m est un impliquant de Σ si et seulement si Σ est une conséquence logique de m. Ceci est équivalent à dire que Σ est impliqué par m.

Définition 27 (impliqué et impliquant premiers)
Un **impliqué** *(respectivement* **impliquant***)* **premier** *d'une formule est un impliqué (respectivement impliquant) qui est minimal pour l'inclusion.*

1.1.3 Formes normales

Définition 28 (CNF)
Une formule Σ est sous **forme normale conjonctive (CNF)** *si et seulement si Σ est une conjonction de clauses, c'est-à-dire une conjonction de disjonctions de littéraux.*

Définition 29 (DNF)
Une formule Σ est sous **forme normale disjonctive (DNF)** *si et seulement si Σ est une disjonction de monômes, c'est-à-dire une disjonction de conjonctions de littéraux.*

De même que pour les clauses (et les monômes), nous considérons une CNF (respectivement DNF) comme un ensemble de clauses (respectivement monômes).

Définition 30 (littéral pur ou monotone)
Un littéral l est dit **pur** *pour la formule Σ sous forme CNF si et seulement si il apparaît soit uniquement positivement soit uniquement négativement dans Σ.*

Propriété 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 Σ en une formule normale conjonctive équivalente du point de vue de la satisfaisabilité dont la taille est linéaire.

Définition 31 (CNF « simplifiée »)
Soient une CNF Σ et un littéral l, nous notons Σ_l la CNF simplifiée en supprimant de Σ toutes les occurrences de $\sim l$ et toutes les clauses où l apparaît.
Par extension, si c est une clause fondamentale, Σ_c correspond à la simplification de Σ par tous les littéraux de la clause c ($\Sigma_c \equiv \Sigma \wedge l_1 \wedge l_2 \wedge ... \wedge l_q$ où $c = \{l_1, l_2, ..., l_q\}$).
Nous notons Σ^* la CNF simplifiée par la propagation[2] de tous ses littéraux unitaires de Σ.
De même, nous notons $\Sigma^\diamond$ la CNF simplifiée par la propagation de tous les littéraux purs de Σ.
Enfin, Σ^+ correspond à la CNF simplifiée par la propagation de tous les littéraux unitaires et purs de Σ.

[2]La propagation unitaire correspond à une « simplification récursive » des littéraux unitaires.

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 I_p définie uniquement pour la variable l ($I_p(l) = Vrai$ 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 $\sim l$.

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

Exemple 2 (Simplifications d'une CNF) *Soit la CNF* $\Sigma = \{(a \vee \neg c),\ (\neg a \vee b \vee c),\ (\neg d \vee \neg a),\ (\neg c \vee d),\ (\neg b \vee e \vee f),\ (\neg e \vee f \vee \neg g),\ (\neg f \vee \neg e)\}$.
Soit la clause $c_1 = \{a,\ \neg d\}$.
$\Sigma_a = \{(b \vee c),\ (\neg d),\ (\neg c \vee d),\ (\neg b \vee e \vee f),\ (\neg e \vee f \vee \neg g),\ (\neg f \vee \neg e)\}$.
$\Sigma_{c_1} = \{(b \vee c),\ (\neg c),\ (\neg b \vee e \vee f),\ (\neg e \vee f \vee \neg g),\ (\neg f \vee \neg e)\}$.
$\Sigma^*_{c_1} = \{(e \vee f),\ (\neg e \vee f \vee \neg g),\ (\neg f \vee \neg e)\}$.
$\Sigma^+_{c_1} = \{(e \vee f), (\neg f \vee \neg e)\}$.

1.2 Le problème SAT

SAT est la forme réduite de problème de satisfaisabilité de formules propositionnelles sous forme normale conjonctive (CNF). C'est un des problèmes qui a été le plus étudié en informatique et qui continue de l'être par une grande communauté de chercheurs du fait de son importance théorique aussi bien que pratique. En effet, SAT fut le premier problème prouvé comme étant NP-complet par Cook [Cook, 1971], faisant de lui le problème NP-complet de référence. De plus, un grand nombre de problèmes en informatique s'y ramènent ou le contiennent, comme la démonstration automatique, la programmation logique, les bases de données déductives, la vérification de circuits et de microprocesseurs, etc.

Définition 32 (Le problème SAT)
Soit Σ *une CNF construite sur un ensemble* S *de symboles propositionnels.*
Existe-t-il une interprétation sur S *qui satisfasse* Σ *?*

Remarque 3

1.2.1 Notations

Nous allons tout de suite fixer un certain nombre de notations qui nous serviront tout au long de ce mémoire lors des calculs de complexités, de définitions etc...

Généralités : Lors de tout calcul de complexité, le nombre de variables propositionnelles présentes dans une formule CNF Σ sera noté $nbVar(\Sigma)$ ou bien, lorsqu'aucune ambiguïté n'est possible, simplement n. Le nombre de clauses de cette formule sera noté $nbCla(\Sigma)$ ou bien, lorsqu'aucune ambiguïté n'est possible, simplement m. La taille d'une clause correspond au nombre de littéraux qui la composent, la taille d'une clause c sera notée $lg(c)$ et la taille de la plus grande clause sera notée k. La taille totale d'une instance SAT Σ sera

notée $|\Sigma|$ et correspond tout naturellement à la somme des longueurs de chaque clause de Σ :

$$|\Sigma| = \sum_{c_i \in \Sigma} lg(c_i).$$

Variables et littéraux : Les variables propositionnelles, ainsi que les littéraux qui leurs sont associés, s'ils ne sont pas explicitement nommés, seront notés par des lettres minuscules de tout l'alphabet (a, l, v, x) éventuellement indexées ($v_1, \neg l_2, v_i, \neg b_j$). Les indices, s'ils ne sont pas explicitement chiffrés, seront quant à eux notés par des lettres minuscules du milieu de l'alphabet (i, j, k ou des nombres entiers). L'ensemble des variables d'une formule Σ sera noté $\mathcal{V}(\Sigma)$ et l'ensemble de ses littéraux $\mathcal{L}(\Sigma)$.

Clauses : Les clauses seront notées par la lettre minuscule c indexée par des indices identiques à ceux des variables. Comme il a été dit, elles seront représentées indifféremment par des ensembles de littéraux ($\{a, b, c\}$) ou par des disjonctions de littéraux ($\{a \vee b \vee c\}$).

Formule CNF : Les formules propositionnelles, et particulièrement celles sous forme CNF (instance SAT) seront notées par des lettres majuscules de l'alphabet grec, généralement Σ, éventuellement indexées avec encore les même indices. Elles seront représentées par des ensembles de clauses ($\Sigma = \{a \vee b \vee \neg d,\ \neg b \vee \neg c,\ a \vee d \vee \neg c,\ c \vee d\}$ ou $\Sigma = \{c_1, c_2, c_3, c_4\}$).

Divers : pour un littéral l, on note $Occ(l)$ l'ensemble des clauses dans lesquelles le littéral l apparaît.

1.3 Les classes polynomiales de SAT

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 satisfaisabilité 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[3]. Nous appelons *restriction polynomiale* une classe pour laquelle seul un algorithme de décision polynomial existe.

Très peu de classes polynomiales de SAT sont connues à ce jour. Nous en établissons dans ce paragraphe une liste non exhaustive.

1.3.1 Les clauses binaires

La première classe polynomiale à avoir été exhibée est 2-SAT [Cook, 1971], c'est-à-dire les instances de SAT formées de clauses binaires. L'algorithme de reconnaissance pour cette classe est trivial et clairement linéaire. L'algorithme de décision de satisfaisabilité proposé par Cook se base sur le principe de résolution et est assurément polynomial puisque la résolvante de deux clauses binaires est au pire une clause binaire et que le nombre de clauses binaires pouvant être construites pour un nombre donné de variables, est une fonction quadratique de ce nombre de variables.

[3] Il suffit de calculer l'ensemble de ces résolvantes, ce qui est fait en temps et en espace polynomial par définition de la classe, et de répondre « oui l'instance admet une solution »ou « non l'instance n'admet pas de solution », en fonction de la production ou non de la clause vide.

Proposition 1 2-*SAT* $\in$ **P** *[Cook, 1971].*

Par la suite d'autres algorithmes de complexité linéaire furent proposés. Citons celui de Even, Itai et Shamir [Even et al., 1976] se basant sur un algorithme énumératif avec un principe de retour arrière intelligent, ou encore celui de Aspvall, Plass et Tarjan [Aspvall et al., 1979] utilisant un codage des clauses en termes de graphes et l'algorithme de Tarjan [Tarjan, 1972] pour calculer les composantes fortement connexes de ce graphe.

1.3.2 Les clauses de Horn

Une autre classe polynomiale bien connue de SAT est **HORN-SAT**.

Définition 33 (HORN-SAT)
Horn-SAT est la restriction de SAT aux ensembles de clauses de Horn.

L'algorithme de reconnaissance, comme pour les clauses binaires, est trivial et clairement linéaire : il suffit de passer en revue les clauses une à une et de s'assurer qu'il y a bien au plus un littéral positif par clause. En ce qui concerne le test de satisfaisabilité d'un ensemble de clauses de Horn, plusieurs auteurs ont montré que la résolution des clauses positives unitaires, qui s'effectue en temps et en espace linéaire, est suffisante [Minoux, 1988], [Dalal, 1992] et [Rauzy, 1995]. D'autres algorithmes de complexité linéaire, se basant sur la théorie des graphes, ont été proposés [Dowling et Gallier, 1984], [Ghallab et Escalada-Imaz, 1991] et [Scutella, 1990].

Proposition 2 *Horn-SAT* $\in$ **P**.

Définition 34 (Renommages)
On appelle **renommage d'un ensemble de littéraux** $\mathcal{S}_L$*, une application* ρ *telle que :*

$$\rho : \quad \mathcal{S}_L \longrightarrow \mathcal{S}_L \,; \qquad l \longmapsto \rho(l) = \begin{cases} \neg l \\ \text{ou} \\ l \end{cases} \qquad \text{et} \qquad \forall\, l \in \mathcal{S}_L : \rho(\neg l) = \neg\rho(l).$$

Par extension, nous définissons un **renommage** ρ_c **d'une clause** c *comme l'application de* ρ *à tous les littéraux de cette clause :* $\rho_c(c) = \bigvee_{l \in c} \rho(l)$.
De même, un **renommage** ρ_Σ **d'un ensemble de clauses** Σ *se définit comme :* $\rho_\Sigma(\Sigma) = \bigwedge_{c \in \Sigma} \rho_c(c)$.

Définition 35 (formule Horn-renommable)
On dit qu'un ensemble de clauses Σ *est* **Horn-renommable** *si et seulement si il existe un renommage* ρ *des littéraux de* Σ *tel que* $\rho_\Sigma(\Sigma)$ *soit un ensemble de clauses de Horn.*

Exemple 3 (Une instance Horn-renommable) *Soient* $\Sigma = \{a \vee b \vee \neg d,\ \neg b \vee \neg c,\ a \vee d \vee \neg c,\ c \vee d\}$ *et* ρ *le renommage suivant :*

$$\begin{array}{rcl} \rho(a) & = & \neg a \\ \rho(b) & = & \neg b \\ \rho(c) & = & c \\ \rho(d) & = & \neg d \end{array}$$

Nous obtenons : $\rho_\Sigma(\Sigma) = \{ \neg a \vee \neg b \vee d,\ b \vee \neg c,\ \neg a \vee \neg d \vee \neg c,\ c \vee \neg d\}$ *qui est un ensemble de clauses de Horn. Par conséquent* Σ *est bien une formule Horn-renommable.* □

Le problème de reconnaissance de formules Horn-renommables n'est pas aussi trivial que celui des clauses de Horn ou des clauses binaires. En effet, il n'est pas immédiat de déterminer si une instance SAT est de Horn à un renommage près des littéraux. Toutefois, ce problème peut être ramené linéairement à celui de la satisfaisabilité d'un ensemble de clauses binaires, et par conséquent traité de manière linéaire [Lewis, 1978, Aspvall, 1980, Chandru et al., 1990, Hébrard, 1994]. Ces algorithmes fournissant pour la plupart le renommage permettant de transformer l'ensemble de clauses en un ensemble de clauses de Horn, décider de la satisfaisabilité d'une formule Horn-renommable peut être fait en temps linéaire par l'intermédiaire d'un algorithme décisionnel emprunté à la classe *Horn*-SAT .

Proposition 3 *Les instances Horn-renommables de SAT constituent une classe polynomiale de SAT.*

Plusieurs auteurs ont cherché à généraliser les deux classes polynomiales « primaires »de SAT (*Horn*-SAT et 2-SAT). Ces recherches ont donné naissance à plusieurs types de généralisations : les clauses q-Horn, un principe de hiérarchisation des clauses, les formules (presque) de Horn et les formules (presque) ordonnées.

1.3.3 Les clauses q-Horn

La classe des clauses q-Horn a été introduite par Boros *et al.* dans [Boros et al., 1990] et se définit de la manière suivante :

Définition 36 (Ensemble de clauses q-Horn)
Un ensemble de clauses Σ *est dit* **q-Horn** *si et seulement si il existe une partition des variables (modulo un renommage) de* Σ *en deux sous-ensembles disjoints* H *et* Q *tels qu'aucune clause de* Σ *:*

- *ne contient plus de deux variables de* Q *;*
- *ne contient plus d'un littéral positif de* H *;*
- *contenant un littéral positif de* H *ne contient de littéral positif de* Q*.*

La difficulté de cette classe est la reconnaissance des ensembles H et Q. En effet, une fois ces ensembles déterminés, le test de satisfaisabilité revient à affecter à la valeur *FAUX* toutes les variables de H et à résoudre le problème restant qui appartient clairement à *2*-**SAT**. En ce qui concerne l'appartenance d'une formule à la classe des clauses q-Horn, Boros *et al.* dans [Boros et al., 1994] ont proposé un algorithme linéaire pour déterminer les ensembles Q et H.

Exemple 4 *Soit* $\Sigma = \{x_1 \vee \neg x_2 \vee \neg x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg x_3 \vee x_4, \neg x_1 \vee \neg x_2 \vee x_5 \vee x_6, \neg x_1 \vee \neg x_3 \vee \neg x_6 \vee x_7, \neg x_2 \vee \neg x_4 \vee x_6 \vee x_7, \neg x_6 \vee \neg x_7, x_6 \vee \neg x_7, x_5 \vee x_6\}$ *En posant* $Q = \{x_5, x_6, x_7\}$ *et* $H = \{x_1, x_2, x_3, x_4\}$*, on voit que toutes les conditions sont respectées, donc* Σ *est* **q-Horn**.

1.3.4 Les différentes hiérarchies

Une autre manière de généraliser les clauses de Horn passe par l'introduction de différentes formes de hiérarchie d'instances SAT.

Généralisation de Yamasaki et Doshita :

Les premiers à avoir travaillé dans cette voie furent Yamasaki et Doshita qui dans [Yamasaki et Doshita, 1983] autorisent les clauses de Horn à posséder plusieurs littéraux positifs, sous condition que ces littéraux soient en quelque sorte « imbriqués ». Ceci afin de garantir que la résolution unitaire reste suffisante pour répondre à la question de la satisfaisabilité de la formule.

Arvind et Biswas ont proposé un algorithme quadratique pour décider de la satisfaisabilité des instances de cette classe [Arvind et Biswas, 1987].

Hiérarchie de Gallo et Scutellà :

Gallo et Scutellà ont par la suite généralisé les travaux de Yamasaki et Doshita. Dans [Gallo et Scutellá, 1988], une hiérarchie $\Gamma_0, \Gamma_1, ...$ d'instances de SAT est définie. Cette hiérarchie se construit de la façon suivante :

Définition 37 (La hiérarchie de Gallo-Scutellà)

- Γ_0 = *Horn-SAT ;*
- $\Sigma \in \Gamma_k$ *si et seulement si il existe un littéral p tel que* $\Sigma_p \in \Gamma_{k-1}$ *et* $\Sigma_{\sim p} \in \Gamma_k$.

Les instances appartenant à Γ_1 représentent en fait la classe de Yamasaki et Doshita décrite dans [Yamasaki et Doshita, 1983]. Gallo et Scutellà ont proposé un algorithme pour reconnaître et résoudre les instances de la classe Γ_k. Cet algorithme est de complexité $\mathcal{O}(|\Sigma|n^k)$ en temps et en espace. De plus, cette classe vérifie les propriétés suivantes :

Proposition 4

- Γ_0 = *Horn-SAT ;*
- $\Gamma_k \supseteq \Gamma_{k-1}, \quad k = 1,\ 2,\ ...$;
- *SAT* $= \bigcup_{k=0}^{\infty} \Gamma_k$.

Hiérarchie de Dalal et Etherington :

La hiérarchie de Gallo et Scutellà a elle-même été généralisée par Dalal et Etherington dans [Dalal et Etherington, 1992] où deux hiérarchies entrelacées (Δ_i) et (Ω_i) sont définies.

Définition 38 (les hiérarchies de Dalal-Etherington)

Soit Σ *une formule CNF. Nous avons :*

- $\Sigma \in \Delta_0$ *si et seulement si* Σ^+ *contient :*
  - *soit la clause vide ;*
  - *soit uniquement des clauses positives ;*
  - *soit uniquement des clauses négatives ;*
- $\forall\ k,\ \Sigma \in \Omega_k$ *si et seulement si*
  - $\Sigma \in \Delta_k$ *;*
  - *ou pour tout littéral* p *de* Σ^+ *:*
    - *soit* $\Sigma_p^+ \in \Delta_k$ *et* $\Sigma_{\sim p}^+ \in \Omega_k$ *;*
    - *soit* $\Sigma_p^+ \subset \Sigma$ *et* $\Sigma_p^+ \in \Omega_k$ *;*
- $\forall\ k > 0, \Sigma \in \Delta_k$ *si et seulement si il existe un littéral* p *de* Σ^+ *tel que :*
  - *soit* $\Sigma_p^+ \in \Omega_{k-1}$ *et* $\Sigma_{\sim p}^+ \in \Delta_k$ *;*
  - *soit* $\Sigma_p^+ \subset \Sigma$ *et* $\Sigma_p^+ \in \Delta_k$.

Il est à noter que contrairement à la hiérarchie de Gallo et Scutellà, nous avons 2-SAT $\subset \Omega_0$.

Dalal et Etherington ont proposé un algorithme de reconnaissance et de résolution de complexité, spatiale et temporelle $\mathcal{O}(|\Sigma|n^k)$.

1.3.5 Les formules presque Horn

Introduite par Luquet dans sa thèse [Luquet, 2000], cette classe polynomiale est une extension des classes des formules de Horn et des formules Horn-renommables. On pourrait dire que cette classe contient les formules qui sont Horn-renommables « par morceaux ». Pour savoir si une formule Σ est presque Horn, on décompose Σ en $\Sigma = BaseH(\Sigma) \cup Reste(\Sigma)$ où $BaseH(\Sigma)$ est la base de Horn de la formule Σ, c'est-à-dire un sous-ensemble de clauses de Σ qui soient Horn-renommables[4], et $Reste(\Sigma)$ le sous-ensemble des clauses restantes de Σ (non Horn-renommable). Les formules Horn-renommables correspondent aux formules pour lesquelles le $Reste$ est vide.

Si $Reste(\Sigma)$ n'est pas vide, on considère l'ensemble $Reste(Reste(\Sigma))$ et ainsi de suite, que l'on appelle **reste itéré** de Σ. Une formule Σ est dite presque Horn si son reste itéré est vide. Luquet a proposé un algorithme de complexité $\mathcal{O}(n|\Sigma|)$ pour la reconnaissance d'une formule presque Horn et $\mathcal{O}(1)$ pour tester leur satisfaisabilité.

La classe $\mathcal{F}$-Horn*

Toujours dans sa thèse, Luquet a encore élargi cette classe à la classe $\mathcal{F}$-Horn*. Soit $\mathcal{F}$ une classe polynomiale de SAT, une formule Σ est dite $\mathcal{F}$-Horn* si le reste itéré de Σ appartient à la classe $\mathcal{F}$ et que Σ ne contient pas de clause unitaire. On voit donc que la classe des clauses q-Horn coïncide avec la classe (2-SAT)-Horn*.

Soit $\mathcal{F}$ une classe polynomiale de SAT pour laquelle on peut tester l'appartenance d'une formule Σ en temps $\mathcal{O}(g(\Sigma))$, alors le test d'appartenance de cette formule à la classe $\mathcal{F}$-Horn* peut être réalisé en temps $\mathcal{O}(n|\Sigma| + g(\Sigma))$. Soit $\mathcal{F}$ une classe polynomiale de SAT pour laquelle on peut tester la satisfaisabilité d'une formule Σ en temps $\mathcal{O}(f(\Sigma))$. Si une formule Σ appartient à la classe $\mathcal{F}$-Horn*, alors on peut tester la satisfaisabilité de Σ en temps $\mathcal{O}(|\Sigma| + f(\Sigma))$.

1.3.6 Les formules ordonnées

La classe des formules ordonnées, introduite par Benoist et Hébrard [Benoist et Hébrard, 1999], peut être vue naturellement comme une extension de la classe des formules de Horn. L'idée derrière cette classe polynomiale est de considérer des clauses qui ressemblent à des clauses de Horn mais qui contiennent plus d'un littéral positif, à condition que les littéraux positifs soient tous **liés** sauf un.

Définition 39 (Littéraux libres/liés)
Soient $c \in \Sigma$ une clause et $l \in c$ un littéral. On dit que l est lié dans c (par rapport à Σ) si $Occ(\neg l) = \emptyset$ ou s'il existe $t \in c$ $(t \neq l)$ tel que $Occ(\neg l) \subseteq Occ(\neg t)$. Si l n'est pas lié dans c, on dit que l est libre dans c.

Les formules ordonnées sont définies ainsi :

[4] La définition exacte de la base de Horn est un peu plus compliquée et fait appel à la définition de Horn-renommabilité d'un ensemble de clauses par rapport à un sous-ensemble de variables, mais la définition approximative que nous donnons permet de comprendre l'intuition cachée derrière cette classe. Nous invitons le lecteur intéressé par plus de détails à consulter [Luquet, 2000].

Définition 40 (Formules ordonnées)
Une formule CNF Σ est ordonnée si chaque clause $c \in \Sigma$ contient au plus un littéral positif libre dans c.

Exemple 5 (Une formule ordonnée) *Soient la formule CNF $\Sigma_1 = \{c_1, c_2, c_3, c_4\}$ et les clauses $c_1 = \{x_1, x_2, x_3, x_4\}, c_2 = \{\neg x_1, x_2, \neg x_3\}, c_3 = \{\neg x_1, \neg x_2, \neg x_4, x_5\}$ et $c_4 = \{\neg x_1, x_3, x_4, \neg x_5\}$. La formule Σ_1 est ordonnée car x_1 est le seul littéral positif et libre de c_1 et x_3 est le seul littéral positif et libre de c_4.*

Benoist et Hébrard ont proposé un algorithme de complexité en temps $\mathcal{O}(n|\Sigma|)$ pour la reconnaissance d'une formule ordonnée et une complexité $\mathcal{O}(|\Sigma|)$ pour le test de satisfaisabilité d'une telle formule.

Cette classe de formule a aussi été étendue en la classe des **formules ordonné-renommables** suivant le même principe que pour la classe des formules Horn-renommables.

Définition 41 (Formule ordonné-renommable)
On dit qu'un ensemble de clauses Σ est **ordonné-renommable** *si et seulement si il existe un renommage ρ des littéraux de Σ tel que $\rho_\Sigma(\Sigma)$ soit un ensemble de clauses ordonnées.*

Exemple 6 (Une formule ordonné-renommable) *Soient la formule CNF $\Sigma_2 = \{c'_1, c'_2, c'_3, c'_4\}$, les clauses $c'_1 = \{\neg x_1, x_2, \neg x_3, x_4\}, c'_2 = \{x_1, x_2, x_3\}, c'_3 = \{x_1, \neg x_2, \neg x_4, x_5\}$ et $c'_4 = \{x_1, \neg x_3, x_4, \neg x_5\}$ et ρ le renommage suivant :*

$$\begin{array}{rcl} \rho(x_1) & = & \neg x_1 \\ \rho(x_2) & = & x_2 \\ \rho(x_3) & = & \neg x_3 \\ \rho(x_4) & = & x_4 \\ \rho(x_5) & = & x_5 \end{array}$$

La formule Σ_2 est ordonné-renommable car $\rho_{\Sigma_2}(\Sigma_2) = \Sigma_1$ et Σ_1 est ordonnée.

Benoist et Hébrard ont montré qu'il était possible de déterminer si une formule est ordonné-renommable en temps $\mathcal{O}(n|\Sigma|)$ et qu'il est possible de tester la satisfaisabilité d'une formule ordonné-renommable en $\mathcal{O}(|\Sigma|)$.

Enfin, cette classe fut une dernière fois étendue en la classe des formules presque ordonnées à la manière des formules presque Horn [Benoist et Hébrard, 1999]. La définition de la classe des formules presque ordonnées est la même que celle des formules presque Horn, mais en remplaçant la *base de Horn* par la **base ordonnée**. Si le reste itéré d'une formule Σ, calculé en fonction de la base ordonnée au lieu de la base de Horn est vide, et que Σ ne contient pas de clause unitaire, alors Σ est une formule presque ordonnée. Benoit et Hébrard ont montré qu'il est possible de reconnaître une formule presque ordonnée en temps $\mathcal{O}(n^2|\Sigma|)$ et de tester sa satisfaisabilité en temps constant ($\mathcal{O}(1)$).

Remarque 4 *Il existe d'autres généralisations des clauses de Horn, comme celles s'appuyant sur des résultats de programmation linéaire $0/1$. Chandru et Hooker ont défini dans [Chandru et Hooker, 1991] une extension des clauses de Horn pouvant être décidée par résolution unitaire. Ces travaux furent étendus dans [Schlipf et al., 1995] où un algorithme pour un ensemble de formules incluant celle de Chandru et Hooker et celles des formules bien imbriquées définies dans [Conforti et Cornuéjols, 1992]*

fut proposé. Le problème est que nous ne disposons pas d'algorithme de reconnaissance polynomial pour ces restrictions polynomiales, ce qui les rend inutilisables en pratique.

1.3.7 Restriction sur le nombre d'occurrences des variables

Une autre manière de forcer une instance SAT à être polynomiale est de restreindre le nombre d'apparitions des variables dans la formule. En particulier, Tovey a montré dans [Tovey, 1984] que la classe des instances où les variables apparaissent au plus deux fois est une classe polynomiale de SAT.

Dans le même article, Tovey a prouvé que les instances de SAT où chaque variable apparaît au plus r fois (r étant la taille de la plus longue clause de l'instance) étaient satisfaisables. Elles sont donc décidables en temps et en espace polynomial.

Ce résultat fut amélioré en premier lieu par Dubois dans [Dubois, 1990], puis par Kratochvil *et al.* qui montrèrent que les instances de SAT telles que le nombre d'occurrences des variables est inférieur ou égal à une fonction exponentielle de la taille maximale des clauses appartiennent à **P**.

Ces restrictions de SAT sur le nombre d'occurrences des variables en fonction de la taille des clauses appartiennent à un problème plus général, le problème r, s-**SAT**.

1.3.8 Les formules bien imbriquées

Reprenant les travaux de Lichtenstein [Lichtenstein, 1982] sur les formules dont le graphe sous-jacent[5] est planaire, Knuth, dans [Knuth, 1990], a défini la classe des formules bien imbriquées :

Définition 42 (Formules bien imbriquées)
Soit $\prec$ un ordre total sur les variables étendu en un pré-ordre sur les littéraux de telle manière que l et $\sim l$ possèdent le même rang pour ce pré-ordre.
Une clause c_1 chevauche une clause c_2 si : $\exists\, l_1 \in c_1$, $\exists\, l_2 \in c_1$ et $\exists\, l_3 \in c_2$ tels que : $l_1 \prec l_3 \prec l_2$.
Un ensemble de clauses est **bien imbriqué** *si et seulement si il ne contient pas deux clauses se chevauchant.*

Knuth n'ayant pas traité le problème de la reconnaissance de ce type de formules, Rossa dans [Rossa, 1993], a fourni un algorithme linéaire qui reconnaît une formule bien imbriquée, pour un ordre fixé des variables.

Un algorithme de décision linéaire a été proposé par Knuth dans [Knuth, 1990]. Cet algorithme est toutefois, comme pour l'algorithme de reconnaissance, linéaire pour un ordre fixé sur les variables.

Nous ne disposons pas à l'heure actuelle d'algorithme « complet » (c'est-à-dire pour tout ordre sur les variables) qui soit polynomial. L'intérêt de cette classe pour un bon nombre d'applications en est par conséquent très amoindri.

[5] Le graphe sous-jacent d'une formule est le graphe biparti non orienté où les sommets symbolisent d'un coté les clauses et de l'autre les littéraux de la formule, tandis que les arêtes représentent l'appartenance d'un littéral à une clause.

1.3.9 La classe Quad

Dalal dans [Dalal, 1996] a proposé une nouvelle classe polynomiale *Quad* pour SAT, qui peut être vue comme une extension des hiérarchies de Dalal-Etherington. Étant donné un ordre total $\prec$ sur les littéraux ($\prec$ sera étendu aux clauses), *Quad* se définit récursivement de la façon suivante :

Définition 43 (Quad)
Une formule Σ appartient à la classe **Quad** *si et seulement si :*

1. *Σ^*, la formule obtenue à partir de Σ après la propagation des clauses unitaires, respecte une des quatre conditions suivantes :*
   (a) *contient la clause vide ;*
   (b) *ne contient pas de clauses positives ;*
   (c) *ne contient pas de clauses négatives ;*
   (d) *ne contient que des clauses binaires ;*
2. *ou, pour la première sous-clause maximale*[6] *μ de la première clause σ de Σ^* pour lequel $\Sigma^*_{\neg\mu}$ satisfasse la condition 1., on ait :*
   (a) *soit $\Sigma^*_{\neg\mu}$ est satisfaisable ;*
   (b) *soit la formule $\Sigma - \{\sigma\} \cup \{\mu\}$ appartient à Quad.*

Afin de mieux se représenter la classe *Quad*, nous l'illustrons sur le petit exemple proposé par Dalal dans [Dalal, 1996].

Exemple 7 (Une formule appartenant à la classe Quad) *Soient $\Sigma = \{\{p,\ q,\ r\},\ \{p,\ q,\ \neg r\},\ \{\neg p,\ \neg q,\ \neg r\}\}$ et l'ordre $\prec$ induit par $p \prec q \prec r \prec \neg p \prec \neg q \prec \neg r$.*

$\Sigma^* = \Sigma$ ne satisfait pas la condition 1. Soit $\sigma = \{p,\ q,\ r\}$ la première clause de Σ, il en découle que la première sous-clause maximale est égale à : $\mu = \{p,\ q\}$.

$\Sigma^*_{\neg\mu}$ contient la clause vide, par conséquent $\Sigma^*_{\neg\mu}$ satisfait bien la condition 1 mais pas la condition 2a. Il faut donc vérifier si la formule $\Sigma - \{\sigma\} \cup \{\mu\} = \{\{p,\ q\},\ \{p,\ q,\ \neg r\},\ \{\neg p,\ \neg q,\ \neg r\}\}$ appartient à la classe Quad. Nous noterons cette nouvelle formule Σ'.

Nous considérons maintenant la formule $\Sigma' = \{\{p,\ q\},\ \{p,\ q,\ \neg r\},\ \{\neg p,\ \neg q,\ \neg r\}\}$. $(\Sigma')^* = \Sigma'$, Σ' ne satisfaisant toujours pas la première condition, il nous faut vérifier la seconde. A cette fin, la première clause devient $\sigma' = \{p,\ q\}$ et sa première sous-clause maximale $\mu' = \{p\}$. Ainsi $\Sigma'^*_{\neg\mu'} = \{\}$, ce qui satisfait la condition 2a. Il en découle que Σ' appartient à la classe Quad et que par conséquent Σ appartient également à la classe Quad. □

Malgré une définition assez lourde, cette classe est reconnaissable en temps et en espace polynomial. En effet, Dalal a fourni dans [Dalal, 1996] un même algorithme (*QuadSat*[7]) quadratique pour la reconnaissance et la résolution de cette classe.

Cependant cette classe souffre de deux problèmes majeurs qui la rendent quasiment inexploitable. Comme pour la classe des formules bien imbriquées, le premier obstacle est la détermination de l'ordre $\prec$. En effet, une formule appartient à la classe *Quad* pour un certain ordre fixé à

[6] On appelle sous-clause maximale de σ une clause μ telle que μ sous-somme σ et que $|\sigma| - |\mu| = 1$. L'ordre $\prec$ permet ainsi de définir la notion de première sous-clause maximale.

[7] *QuadSat* est en fait en $\mathcal{O}(n^2k)$ où n représente la taille de la formule et k la taille de la plus longue clause de cette formule.

l'avance. Le problème de l'existence d'un tel ordre n'est décidable, dans le pire cas, qu'après l'essai des $n!$[8] ordres possibles. Ce qui rend l'algorithme de reconnaissance de la classe *Quad* pour un ordre non fixé exponentiel. Le second problème est que la classe *Quad* n'est pas stable par adjonction de clauses unitaires, cela rend son utilisation impossible dans de nombreuses applications (*e.g.* l'implication de clauses). Nous proposons un exemple pour illustrer cette non propriété [Marquis, 1997].

Exemple 8 (Quad : une classe polynomiale non stable pour l'adjonction de clauses unitaires) *Soit* $\Sigma = \{\{\neg x, \alpha_1, \beta_1, \gamma_1\}, \{\neg x, \alpha_2, \beta_2, \gamma_2\}, \{\neg\alpha_3, \neg\beta_3, \neg\gamma_3\}, \{\neg\alpha_4, \neg\beta_4, \neg\gamma_4\}, \{x, y\}\}$. *Nous avons :* $\forall \prec, \Sigma \in Quad$. *En effet, quel que soit l'ordre choisi, nous serons amenés à considérer la sous-clause maximale* $\mu = \{x\}$ *car pour tous les autres* μ *possibles,* $\Sigma^*_{\neg\mu}$ *ne satisfait pas la condition 1 (nécessaire à la satisfaction de la condition 2). De plus* $\Sigma^*_{\neg\{x\}} = \{\{\neg\alpha_3, \neg\beta_3, \neg\gamma_3\}, \{\neg\alpha_4, \neg\beta_4, \neg\gamma_4\}\}$ *qui satisfait les conditions 1 et 2a. Par conséquent* $\Sigma \in Quad$.

En revanche, qu'en est-il de $\Sigma' = \Sigma \cup \{x\}$ *?*
Nous avons $\Sigma'^* = \{\{\alpha_1, \beta_1, \gamma_1\}, \{\alpha_2, \beta_2, \gamma_2\}, \{\neg\alpha_3, \neg\beta_3, \neg\gamma_3\}, \{\neg\alpha_4, \neg\beta_4, \neg\gamma_4\}\}$. *Très rapidement, il est facile de voir que, quel que soit l'ordre choisi, il n'existe pas de clause* μ' *tel que* $\Sigma'^*_{\neg\mu'}$ *satisfasse la condition 1 nécessaire à la satisfaction de la condition 2.*

Par conséquent nous avons $\Sigma' \notin Quad$, *ce qui implique que la classe Quad n'est pas stable par adjonction de clauses unitaires.* □

1.3.10 Récapitulatif

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. 1.2 – Tableau comparatif des complexités en temps pour la reconnaissance et le test de satisfaisabilité de quelques classes polynomiales de SAT connues. $g(\Sigma)$ et $f(\Sigma)$ représentent respectivement la complexité en temps de la reconnaissance et du test de satisfaisabilité de la classe $\mathcal{F}$.

[8] n représente ici le nombre de littéraux de la formule.
[9] sans clause unitaire.

Comme nous avons pu le constater, un certain nombre de classes polynomiales ont été étudiées, le tableau 1.2 donne un récapitulatif des différentes complexités des algorithmes de reconnaissance et de test de satisfaisabilité de ces classes polynomiales.

1.4 Les ensembles backdoor et strong backdoor

Les ensembles **backdoor** et **strong backdoor** ont été introduits par Williams, Gomes et Selman [Williams et al., 2003a, Williams et al., 2003b]. Un ensemble B de variables est un ensemble **backdoor**[10] pour une formule CNF Σ s'il existe une interprétation des variables de B pour laquelle la formule simplifiée peut être **satisfaite** en temps polynomial. Cet ensemble est appelé ensemble **strong backdoor** si pour toute interprétation de ses variables, la formule simplifiée appartient à une classe polynomiale de SAT. Plus formellement, ces ensembles sont caractérisés par les définitions suivantes.

Définition 44 (Ensemble (weak) backdoor)
*Soient Σ une formule CNF et B un sous-ensemble de $\mathcal{V}(\Sigma)$. On dit que B est un ensemble **(weak) backdoor** de Σ si et seulement si il existe une interprétation I_B de B telle que Σ_{I_B} peut être **satisfaite** en temps polynomial.*

Définition 45 (Ensemble strong backdoor)
*Soient Σ une formule CNF et B un sous-ensemble de $\mathcal{V}(\Sigma)$. On dit que B est un ensemble **strong backdoor** si et seulement si pour toute interprétation I_B des variables de B, **le test de satisfaisabilité** de la formule simplifiée Σ_{I_B} peut être réalisé en temps polynomial.*

Dans ces deux définitions, la condition *« en temps polynomial »* peut être interprétée de deux manières différentes :

- soit un algorithme de complexité polynomial –comme la propagation unitaire, la résolution bornée, la propagation des littéraux purs, etc– permet de décider de la satisfaisabilité de la formule simplifiée,
- soit la formule simplifiée appartient à une classe polynomiale connue, comme les formules de Horn, les formules 2-SAT etc. Dans ce cas, on parle d'ensemble **(*strong*) $\mathcal{F}$-backdoor**.

Définition 46 (Ensemble $\mathcal{F}$-backdoor)
*Soient Σ une formule CNF, B un sous-ensemble de $\mathcal{V}(\Sigma)$ et $\mathcal{F}$ une classe polynomiale de SAT. On dit que B est un ensemble **$\mathcal{F}$-backdoor** si et seulement si il existe une interprétation I_B des variables de B pour laquelle Σ_{I_B} est satisfaisable et $\Sigma_{I_B} \in \mathcal{F}$.*

Définition 47 (Ensemble strong $\mathcal{F}$-backdoor)
*Soient Σ une formule CNF, B un sous-ensemble de $\mathcal{V}(\Sigma)$ et $\mathcal{F}$ une classe polynomiale de SAT. On dit que B est un ensemble **strong $\mathcal{F}$-backdoor** si et seulement si pour toute interprétation I_B des variables de B, $\Sigma_{I_B} \in \mathcal{F}$.*

Williams *et al.* [Williams et al., 2003a] ont proposé un algorithme de calcul d'ensembles backdoor pour le cas où la satisfaisabilité de la formule simplifiée est prouvée par la propagation des clauses unitaires. Le principe de cet algorithme est de calculer toutes les combinaisons possibles de littéraux formées sur tous les sous-ensembles de variables de la formule ayant une cardinalité

[10] appelé parfois ensemble *weak backdoor*.

croissante de 0 à $n-1$. Nous pouvons constater que le premier ensemble backdoor qui sera trouvé par cet algorithme est le plus petit ensemble backdoor[11] de la formule, mais la complexité d'un tel algorithme est fortement exponentielle. En effet, en admettant que le plus petit ensemble backdoor de la formule soit de taille k, alors l'algorithme devra tester

$$\sum_{i=0}^{k-1} C_n^i \times 2^i$$

combinaisons infructueuses. Bien que ces tests soient réalisés en temps polynomial, le temps de calcul devient rapidement prohibitif. En constatant que le calcul d'ensembles backdoor induit le calcul de satisfaisabilité, ils ont ensuite proposé des raffinements formels de ce calcul d'ensembles backdoor qui, en supposant que la taille du plus petit ensemble backdoor de la formule soit de l'ordre de $\mathcal{O}(\log(n))$, devraient permettre de calculer un ensemble backdoor (et donc prouver la satisfaisabilité) en temps *polynomial*.

Nous avons constaté que l'existence d'un ensemble backdoor est lié à la satisfaisabilité de l'instance, ce qui n'est pas le cas pour les ensembles strong backdoor. Ceci explique les différentes complexités des problèmes de calculs d'ensembles (strong) backdoor. En effet, Nishimura, Ragde et Szeider [Nishimura et al., 2004] ont montré que la complexité du problème paramétré du calcul d'un ensemble $\mathcal{F}$-backdoor, pour $\mathcal{F} \in$ {*Horn*-SAT , 2-SAT }, n'est pas polynomiale, alors qu'elle l'est pour le calcul d'ensembles strong $\mathcal{F}$-backdoor. En d'autres termes, ils ont montré que déterminer s'il existe un ensemble strong Horn-backdoor de taille k pour une formule Σ peut être réalisé en temps $\mathcal{O}(2^k|\Sigma|)$ grâce à l'algorithme 1, et que déterminer s'il existe un ensemble strong 2-SAT -backdoor de taille k pour une formule Σ peut être réalisé en temps $\mathcal{O}(3^k|\Sigma|)$ grâce à l'algorithme 2. Alors que la complexité du problème paramétré du calcul d'un ensemble $\mathcal{F}$-backdoor est $w[2]-$difficile [Creignou et al., 2001], soit exponentielle si on considère que $P \neq NP$.

Ils ont également montré que les versions non paramétrées[12] des problèmes de calculs d'ensembles $\mathcal{F}$-backdoor et strong $\mathcal{F}$-backdoor sont NP-complets dans le cas général.

Reprenant les travaux de [Williams et al., 2003a] sur les ensembles strong backdoor pour lesquels la propagation unitaire suffit à tester la satisfaisabilité des formules simplifiées, Kilby *et al.* [Kilby et al., 2005] ont proposé l'algorithme 3 calculant tous les ensembles backdoor et strong backdoor dont la taille est inférieure à un paramètre fixé. Cet algorithme fait appel au solveur *satz v2.15* qui est un solveur SAT de Chu Min Li [Li, 1999] des plus connus intégrant la propagation unitaire.

Cependant, cette méthode de calcul d'ensembles strong backdoor est confrontée à une forte et rapide explosion combinatoire. C'est pourquoi elle n'a pu être utilisée avec succès seulement sur des instances de petites tailles ne dépassant pas 50 variables et présentant des ensembles strong backdoor de taille 4 au maximum.

1.5 Méthodes de résolution pratiques de SAT

Comme nous l'avons vu, SAT est un problème NP-complet. Tant que la preuve de $P = NP$ n'aura pas été apportée, il y a peu de chance d'arriver à trouver un algorithme qui soit capable de traiter le problème de décision de SAT en temps polynomial dans le cas général. Pour une formule

[11] Plus petit en terme de cardinalité.

[12] C'est-à-dire sans fixer la valeur de k.

Algorithme 1 Procédure de calcul d'ensembles strong Horn-backdoor paramétré.

Procédure sb-Horn
Entrée : une formule CNF Σ et un entier k
Sortie : soit un ensemble strong Horn-backdoor B de Σ de taille au plus k, soit « non » si un tel ensemble B n'existe pas.

1: **si** $\Sigma \in$ Horn **alors**
2: **retourner** $\emptyset$
3: **fin si**
4: **si** $k = 0$ **alors**
5: **retourner** non
6: **fin si**
7: choisir une clause non Horn $c \in \Sigma$ et deux littéraux positifs $l_1, l_2 \in c$
8: **sb-Horn(**$\Sigma - l_1, k - 1$**)**[13]
9: **si** un ensemble B_1 est retourné **alors**
10: **retourner** $B_1 \cup \{l_1\}$
11: **fin si**
12: **sb-Horn(**$\Sigma - l_2, k - 1$**)**[13]
13: **si** un ensemble B_2 est retourné **alors**
14: **retourner** $B_2 \cup \{l_2\}$
15: **fin si**
16: **retourner** non

CNF Σ comptant n variables, la majeure partie des solveurs doit donc en théorie et dans le pire des cas parcourir les 2^n interprétations possibles des variables dans le but de trouver un modèle ou de prouver qu'il n'en existe pas.

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.

1.5.1 Les méthodes complètes

Un grand nombre de méthodes complètes ont été proposées. Voici une petite liste non exhaustive de solveurs :

- la méthode « force brute » (table de vérité) ;
- la procédure DPLL [Davis et al., 1962] et ses dérivées (GRASP [Silva et Sakallah, 1996], satz [Li, 1999], zChaff [Moskewicz et al., 2001], BERKMIN [Goldberg et Novikov, 2002], 2cls+eq [Bacchus, 2002], kcnfs [Dequen et Dubois, 2003]). C'est un algorithme de recherche en profondeur qui énumère de façon implicite toutes les solutions.
- la résolution [Robinson, 1965], qui est similaire à la procédure classique DP.
- les Diagrammes de décision binaires [Bryant, 1987]
- la Méthode de comptage [Iwama, 1989]

[13] $\Sigma - l$ représente la formule Σ dans laquelle toutes les occurences de l et $\neg l$ ont été retirées.

Algorithme 2 Procédure de calcul d'ensembles strong 2-SAT -backdoor paramétré.

Procédure sb-2-SAT
Entrée : une formule CNF Σ et un entier k
Sortie : soit un ensemble strong 2-SAT -backdoor B de Σ de taille au plus k, soit « non » si un tel ensemble B n'existe pas.

si $\Sigma \in$ 2-SAT **alors**
  retourner $\emptyset$
fin si
si $k = 0$ **alors**
  retourner non
fin si
choisir une clause non binaire $c \in \Sigma$ et trois littéraux positifs $l_1, l_2, l_3 \in c$
sb-2-SAT($\Sigma - l_1, k-1$**)**[13]
si un ensemble B_1 est retourné **alors**
  retourner $B_1 \cup \{l_1\}$
fin si
sb-2-SAT($\Sigma - l_2, k-1$**)**[13]
si un ensemble B_2 est retourné **alors**
  retourner $B_2 \cup \{l_2\}$
fin si
sb-2-SAT($\Sigma - l_3, k-1$**)**[13]
si un ensemble B_3 est retourné **alors**
  retourner $B_3 \cup \{l_3\}$
fin si
retourner non

Algorithme 3 Procédure de calcul d'ensembles backdoor et strong backdoor.

Procédure StrongBackdoor
Entrée : une formule CNF Σ et un entier $Max\text{-}card$
Sortie : un ensemble d'ensembles strong backdoor S et un ensemble d'ensembles (weak) backdoor W de Σ dont les tailles n'excèdent pas $Max\text{-}card$

$S \leftarrow W \leftarrow \emptyset$
pour tout sous-ensemble $X \subset \mathcal{V}(\Sigma)$ de taille allant de 0 à $Max\text{-}card$ **faire**
  pour tout ensemble distinct L de littéraux correspondant aux variables de X **faire**
    satz(Σ_L)
    si aucun branchement n'est nécessaire **et** Σ_L est satisfaisable **alors**
      $W \leftarrow W \cup L$
    fin si
  fin pour
  si aucun des ensembles L de littéraux n'a requis de branchement **alors**
    $S \leftarrow S \cup X$
  fin si
fin pour
retourner S, W

Cependant, la plupart des algorithmes de résolution de SAT sont basés sur la procédure DPLL [Davis et al., 1962] et occasionnellement sur le principe de résolution [Robinson, 1965].

La procédure DPLL

Cet algorithme, proposé par Davis, Logeman et Loveland [Davis et al., 1962] est un algorithme de recherche en profondeur d'abord. Il sert de base à la plupart des solveurs de la dernière génération.

Étant donné une formule CNF Σ et l une variable (ou atome) de Σ, la procédure DPLL est basée sur l'idée que Σ est consistante si et seulement si $\Sigma \wedge l$ ou $\Sigma \wedge \neg l$ est satisfaisable. Cette procédure consiste donc à choisir une variable et à tester récursivement les deux sous-formules en interprétant successivement cette variable à *VRAI* puis à *FAUX* si nécessaire. L'algorithme 4 décrit la procédure DPLL.

Algorithme 4 DPLL

Entrée : une CNF Σ.
Sortie : *VRAI* si Σ est consistant, *FAUX* sinon

```
Σ* ← PropagationUnitaire(Σ)
si Σ contient une clause vide alors
   retourner FAUX
fin si
si Σ* == ∅ alors
   retourner VRAI
fin si
l ← choixDeVariable(Σ*)
si DPLL(Σ*∪{l=vrai})=VRAI alors
   retourner VRAI
sinon
   retourner DPLL(Σ*∪{l=faux})
fin si
```

Cette procédure est connue pour être complète et cohérente pour SAT. L'utilisation de la propagation unitaire permet de faire l'économie d'un certain nombre de tests d'interprétations qui ne seraient pas concluant, permettant d'accroître significativement les performances de la recherche de modèle. Les solveurs récents basés sur cette procédure intègrent tous cette propagation, réalisée de manière toujours plus efficace, ainsi que d'autres méthodes de simplification comme la propagation des littéraux purs ou des procédés d'apprentissage.

La propagation unitaire : La propagation unitaire, aussi connue sous les noms de propagation de contraintes booléennes ou résolution unitaire, est un des procédés clé de simplification de formules CNF qui est sans conteste le plus utilisé. Son principe est de supprimer dans la formule CNF toutes les clauses où apparaît un littéral unitaire et de supprimer toutes les occurrences du littéral opposé à ce littéral dans les clauses où il apparaît. Ce processus est répété tant qu'il reste un littéral unitaire où bien jusqu'à ce que la clause vide soit produite.

La propagation unitaire peut être réalisée en temps linéaire par rapport à la taille de la formule, et peut également décider de la satisfaisabilité d'un ensemble de clauses de Horn.

Le principe de résolution

Le principe de résolution, initié par Robinson [Robinson, 1965], consiste à produire des résolvantes à partir des clauses de la formule qui sont conséquences logiques de la formule. Si la formule est inconsistante, alors la clause vide sera produite. Dans le cas contraire, la formule est consistante.

Le principe de résolution, lorsque l'on procède par saturation, est adapté à la preuve de l'inconsistance d'une formule, car on obtient une preuve de l'inconsistance dès la production de la clause vide. Par contre, la preuve de la consistance est plus difficile : il faut s'assurer de ne pas pouvoir produire la clause vide c'est-à-dire de ne plus pouvoir produire de résolvante.

Le principal défaut de cette méthode est la taille de la formule qui devient assez vite prohibitif et son implémentation demande de nombreuses ressources mémoires.

1.5.2 Les méthodes incomplètes

Les algorithmes des méthodes complètes 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.

La plupart des méthodes incomplètes sont basées sur une méthode de 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 pour la recherche de modèle.

Parmi les plus connues de ces méthodes, GSAT [Selman et al., 1992a] est l'une des plus représentatives et des plus anciennes. Son principe est simple et intuitif, il consiste à générer aléatoirement une interprétation complète des variables de l'instance, puis à tenter de réparer cette interprétation en changeant successivement les valeurs de vérité attribuées à un certain nombre de variables[14] fixé au préalable. À chaque étape, la variable qui doit être flippée est choisie en fonction du nombre de clauses falsifiées qui deviennent satisfaites. Cette heuristique de choix est appelée *Min_Conflict*. Le voisinage évalué lors de chaque mouvement est donc composé des interprétations qui diffèrent de l'interprétation courante d'un littéral uniquement (*le voisinage direct*). Si le nombre maximun de flip (*Max_Flips*) est atteint sans qu'un modèle n'ait été trouvé, alors GSAT relance ce processus en partant d'une autre interprétation générée aléatoirement. Le nombre maximal de *restarts* (*Max_Tries*) est lui aussi fixé au départ. Ces restarts successifs permettent une certaine diversification dans la recherche. L'algorithme 1.5.2 décrit le solveur GSAT.

Une des améliorations les plus importantes pour ces méthodes consiste à intégrer la stratégie de « marche aléatoire » pendant la résolution, produisant l'algorithme WalkSat et ses variantes : Novelty, Novelty+, R-Novelty et R-Novelty+ [Selman et al., 1994, McAllester et al., 1997, Hoos, 1999, Hoos et Stützle, 2000].

La marche aléatoire permet d'accélérer le processus de recherche tout en permettant de s'échapper de certains minimum locaux. Avec cette stratégie, le choix de la variable à flipper est restreint aux variables impliquées dans une seule clause falsifiée choisie au hasard. Ainsi, seulement un sous-ensemble du voisinage direct est évalué, permettant d'augmenter significativement le nombre de

[14] Ces opérations sont généralement appelées des *flips*.

Algorithme 5 GSAT

Entrée : Σ un ensemble de clauses, deux entiers : Max_Tries et Max_Flips ;
Sortie : $Vrai$ si un modèle de Σ est trouvé, $Faux$ s'il est impossible de conclure ;

```
pour i allant de 1 à Max_Tries faire
  I=une interprétation complète générée aléatoirement
  pour j allant de 1 à Max_Flips faire
    si I est un modèle alors
      retourner Vrai ;
    sinon
      pour toute variable v de Σ faire
        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}
      fin pour
      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)
    fin si
  fin pour
fin pour
retourner (I est modèle) {retourne la valeur du test : « I est-il un modèle ? »}
```

flips possibles dans un temps fixé. D'autre part, cette stratégie permet aussi, selon une certaine probabilité, de s'affranchir de l'évaluation d'un quelconque voisinage en choisissant aléatoirement une variable impliquée dans une clause falsifiée comme variable à flipper. Ce dernier procédé permet d'introduire une phase de diversification supplémentaire.

Les méthodes de recherche locale sont aussi de plus en plus utilisées pour résoudre des problèmes d'optimisation et notamment le problème Max-SAT, qui consiste non pas à déterminer si une instance est satisfaisable ou non, mais à trouver l'interprétation satisfaisant un maximum de clauses dans une CNF. L'algorithme 6 synthétise la méthode WalkSat avec la marche aléatoire pour le problème Max-SAT.

Algorithme 6 WalkSat

Entrée : Une formule CNF Σ, et deux entiers : Max_Tries et Max_Flips ;
Sortie : Une *interprétation* satisfaisant le maximum de clauses de Σ

Initialiser I_{max} avec toutes les variables de Σ à vrai
pour $i = 1$ à *MAX_TRIES* **faire**
  $I \leftarrow$ interprétation générée aléatoirement
  pour $j = 1$ à *MAX_FLIPS* **faire**
    si I est modèle de Σ **alors**
      retourner I {Σ *est satisfaisable*}
    fin si
    {*Random Walk Strategy*}
    Sélectionner aléatoirement une clause non satisfaite c
    Avec une probabilité p **faire**
    Sélectionner aléatoirement un littéral l de c
    $I \leftarrow \{I - \{l\}\} \cup \{\neg l\}$
    fait
    Avec une probabilité $1 - p$ **faire**
    Soit $l \in I$ tel que $\forall l' \in I$ avec $l \neq l'$, $score(l, I) > score(l', I)$ {Heuristique min-conflict}
    $I \leftarrow \{I - \{l\}\} \cup \{\neg l\}$
    fait
    si nombre de clauses satisfaites par $I >$ nombre de clauses satisfaites par I_{max} **alors**
      $I_{max} \leftarrow I$
    fin si
  fin pour
fin pour
retourner I_{max}

Chapitre 2

Les problèmes de satisfaction de contraintes

Sommaire

Nous allons maintenant définir le formalisme des problèmes de satisfaction de contraintes, communément appelé CSP pour Constraints Satisfaction Problems. Une fois les définitions introduites, nous présentons une petite étude des différentes méthodes de filtrage par consistances locales, processus central dans la résolution des CSP.

2.1 Définition du formalisme

Un problème de satisfaction de contrainte est défini par un ensemble de variables, où chaque variable est associée à un domaine fini discret de valeurs possibles. Certaines de ces variables sont liées par des contraintes définissant les relations existantes entre elles. Ces relations définissent l'ensemble des combinaisons de valeurs autorisées par les contraintes. Nous donnons la définition formelle proposée par Montanari en 1974 [Montanari, 1974] :

Définition 48 (Les problèmes de satisfaction de contraintes)
Un **problème de satisfaction de contraintes** *est défini par un quadruplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{R})$ où :*

- $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ *est un ensemble de* n **variables**.
- $\mathcal{D} = \{D_{X_1}, D_{X_2}, \ldots, D_{X_n}\}$ *est un ensemble de* **domaines** *finis discrets, où D_{X_i} est le domaine associé à la variable X_i représentant l'ensemble des valeurs possibles de X_i.*
- $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ *est un ensemble de* m **contraintes**, *où la contrainte C_i est définie par un ensemble de variables $\{X_{i_1}, X_{i_2}, \ldots, x_{i_{n_i}}\} \subset \mathcal{X}$. L'arité de C_i ($i \in \{i, \ldots, m\}$) vaut n_i.*
- $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ *est un ensemble de* m **relations**, *où R_i est la relation associée à la contrainte C_i. R_i est l'ensemble des combinaisons de valeurs des variables impliquées dans la contrainte qui satisfont C_i. C'est un sous-ensemble du produit cartésien des domaines des variables de C_i, $R_i \subset D_{X_{i_1}} \times D_{X_{i_2}} \times \ldots \times D_{X_{i_{n_i}}}$.*

Généralement, une distinction est faite entre les *CSP binaires* et les *CSP n-aires* (ou non binaires). Un CSP binaire est un CSP qui contient uniquement des contraintes binaires (d'arité deux), c'est-à-dire uniquement des contraintes liant exactement deux variables. Dès lors qu'une contrainte implique plus de deux variables, cette contrainte est dite n-aire et un CSP contenant au moins une contrainte n-aire est un CSP n-aire. Comme nous allons le voir, ces deux classes de CSP ont beaucoup de points communs mais aussi et surtout un certain nombre de différences, tant du point de vue pratique que théorique.

Nous avons opté pour la définition des CSP en extension, c'est à dire que toutes les relations sont décrites par des tables énumérant l'ensemble des combinaisons autorisées par les contraintes, sous forme de n-uplets. Ce choix pourrait être discutable, mais étant donné que nous nous plaçons dans le cadre de domaines finis discrets, il est toujours possible d'exprimer les relations en extension.

Définition 49
Une instanciation *est une affectation des variables d'un sous-ensemble $\mathcal{Y} \subseteq \mathcal{X}$ à une des valeurs de leur domaine de définition. On parle d'instanciation partielle lorsque $\mathcal{Y} \neq \mathcal{X}$ (lorsque toutes les variables de $\mathcal{X}$ ne sont pas affectées), et d'instanciation complète lorsque $\mathcal{Y} = \mathcal{X}$ (toutes les variables de $\mathcal{X}$ sont affectées). Un instanciation, complète ou partielle, est dite consistante si aucune des contraintes impliquant des variables de l'instanciation n'est violée. Par la suite, une instanciation, complète ou pas, pourra être représentée soit par une application qui associe à chaque variable une valeur de son domaine, soit par un ensemble de couples (***variable, valeur***).*

Définition 50
Une solution *d'un CSP est une instanciation* complète *et* consistante *du CSP. Un CSP est dit consistant ou soluble s'il admet au moins une solution. Dans le cas contraire, le CSP est dit insoluble ou inconsistant.*

Le formalisme CSP peut être utilisé de multiple façons. Voici une liste non exhaustive de tâches auxquelles nous pouvons être confrontés lorsque nous traitons un CSP $\mathcal{P}$:

1. Déterminer si $\mathcal{P}$ possède une solution ($\mathcal{P}$ est-il soluble ?)
2. Trouver une solution de $\mathcal{P}$
3. Rechercher l'ensemble des solutions de $\mathcal{P}$
4. Rechercher le nombre de solutions
5. Rechercher la meilleure solution (en fonction d'un critère donné)

Comme pour le problème SAT, le problème de décision (existence de solution) associé au CSP[1] est NP-complet, le problème de recherche du nombre de solutions et de l'ensemble des solutions est #P-complet et le problème de recherche de la meilleure solution est NP-difficile.

Dans cette thèse, nous nous intéresserons en particulier au problème de décision associé au CSP.

2.1.1 Notations

Nous allons d'ores et déjà fixer un certain nombre de notations qui seront utilisées tout au long de ce mémoire.

Généralités : Lors de tout calcul de complexité, le nombre de variables dans un CSP sera noté n (À ne pas confondre avec le n de contraintes n-aires), la taille du plus grand domaine associé à une variable sera noté d, le nombre de contraintes sera noté m et l'arité maximum des contraintes a.

Variables : Les variables d'un CSP, si elles ne sont pas explicitement nommées, seront notées par des lettres majuscules de la fin de l'alphabet (X, Y, Z) éventuellement indexées (X_1, X_i, X_j). Les indices, s'ils ne sont pas explicitement chiffrés, seront quant à eux notés par des lettres minuscules du milieu de l'alphabet (i, j, k ou des nombres entiers).

Domaines et valeurs : Chaque domaine sera noté par la lettre majuscule D indexée par le nom de la variable à laquelle il est associé. Ainsi le domaine associé à une variable appelée X_1 sera noté D_{X_1}, celui associé à une variable appelée *Couleur* sera noté D_{Couleur}. Les valeurs des domaines, quand elles ne sont pas nommées explicitement, seront notées par des lettres minuscules de tout l'alphabet, éventuellement indexées avec les mêmes indices que ceux des variables (x, y, v_1, v_i).

Contraintes et relations : Les contraintes seront notées par la lettre majuscule C indexée par l'ensemble des variables impliquées dans la contrainte. Ainsi, une contrainte d'arité k portant sur les variables $\{X_1, \ldots, X_k\}$ sera notée $C_{X_1 \ldots X_k}$ ou seulement $C_{1 \ldots k}$ si aucune ambiguïté n'est possible. Chaque relation sera notée par la lettre majuscule R indexée par le même indice que la contrainte à laquelle elle est associée.

2.1.2 Représentation des contraintes

Les tables

Les contraintes des problèmes que nous sommes susceptibles de traiter sont de nature aussi diverses que les domaines où les CSP peuvent être employés. En effet, les problèmes d'ordonnancement sont principalement composés de contraintes temporelles et les problèmes de coloriage de graphes sont principalement composés de contraintes de différences. Nous pouvons aussi rencontrer des contraintes algébriques, sous forme de fonctions arithmétiques.

Cependant, comme nous l'avons stipulé, nous nous plaçons dans le cadre de contraintes dont les relations sont exprimées en extension, c'est à dire sous la forme de tables énumérant les tuples autorisés par chaque contrainte. Ainsi, quelque soit la nature de la contrainte traitée, il faudra l'exprimer sous forme de liste de tuples autorisés.

[1] Qu'il s'agisse de CSP binaire ou n-aire.

Avant d'aller plus loin, voici un exemple qui illustre le formalisme des CSP et la représentation des contraintes que nous avons adoptée.

Exemple 9 *Considérons un problème simplifié de production de voitures où l'on doit attribuer des couleurs aux différentes parties des véhicules, en respectant les contraintes suivantes :*

- *La carrosserie doit être plus foncée que les pare-chocs, le toit ouvrant et les enjoliveurs.*
- *Les portières, la carrosserie et le capot doivent être de la même couleur.*

Pour modéliser ce problème sous forme d'un CSP n-aire, nous commençons par définir l'ensemble de variables. Ces variables correspondent dans ce cas aux différents éléments de la carrosserie d'un voiture : le Pare-chocs, *le* Toit ouvrant, *les* Enjoliveurs, *le* Capot, *les* Portières *et enfin la* Carrosserie *(qui correspond à tout le reste du véhicule).*

Il s'agit maintenant de définir les domaines de définition de chacune de ces variables, qui correspondent aux différentes couleurs que peuvent recevoir les éléments de la carrosserie. Nous les énumérons selon le schéma (variable : valeurs) comme suit :

- Pare-chocs *: Blanc*
- Toit ouvrant *: Rouge*
- Enjoliveurs *: Rose ou Rouge*
- Capot *et* Portières *: Rose, Rouge ou Noir*
- Carrosserie *: Blanc, Rose, Rouge ou Noir*

Pour modéliser les contraintes, il nous faut énumérer l'ensemble des combinaisons possibles pour chacune d'elle :

- *La carrosserie doit être plus foncée que les pare-chocs, le toit ouvrant et les enjoliveurs.*
  Cette contrainte donne lieu à la définition des 3 contraintes binaires suivantes : $C_{\text{Pare-chocs Carrosserie}}$, $C_{\text{Toit-ouvrant Carrosserie}}$ *et* $C_{\text{Enjoliveurs Carrosserie}}$ *dont les relations associées sont décrites par les tables de la figure 2.1.*

Pare-chocs	Carrosserie
blanc	rose
blanc	rouge
blanc	noir

(a) $R_{\text{Pare-chocs Carrosserie}}$

Toit ouvrant	Carrosserie
rouge	noir

(b) $R_{\text{Toitouvrant Carrosserie}}$

Enjoliveurs	Carrosserie
rose	rouge
rose	noir
rouge	noir

(c) $R_{\text{Enjoliveurs Carrosserie}}$

FIG. 2.1 – Relations associées aux 3 contraintes binaires.

- *Les portières, la carrosserie et le capot doivent être de la même couleur.*
  Cette contrainte donne lieu à la définition de la contrainte ternaire[2] $C_{\text{Carrosserie Portières Capot}}$ *dont la relation associée est donnée par le tableau 2.1.*

Carrosserie	Portières	Capot
rose	rose	rose
rouge	rouge	rouge
noir	noir	noir

TAB. 2.1 – Relation associée à la contrainte ternaire ($R_{\text{Carrosserie Portières Capot}}$).

[2] D'arité trois.

Les (hyper-)graphes de contraintes

Un des intérêts majeurs de la formalisation CSP réside dans le fait que cette représentation est très structurée. En effet, un autre moyen de représenter un CSP binaire consiste à l'associer à un graphe appelé graphe de contraintes du CSP. Dans cette représentation, l'ensemble des sommets du graphe coïncide avec l'ensemble des variables du CSP, et deux sommets sont reliés par une arête s'il existe une contrainte qui lie les deux variables associées dans le CSP. De plus, chaque arête est étiquetée par la liste des tuples autorisés par la relation de la contrainte correspondante.

Dans le cas de CSP n-aire, ce graphe est remplacé par un hyper-graphe, dans lequel les hyper-arêtes correspondent aux contraintes n-aires du CSP. Pour illustrer cette construction, nous donnons l'hyper-graphe de contraintes associé au CSP de l'exemple 9 dans la figure 2.2.

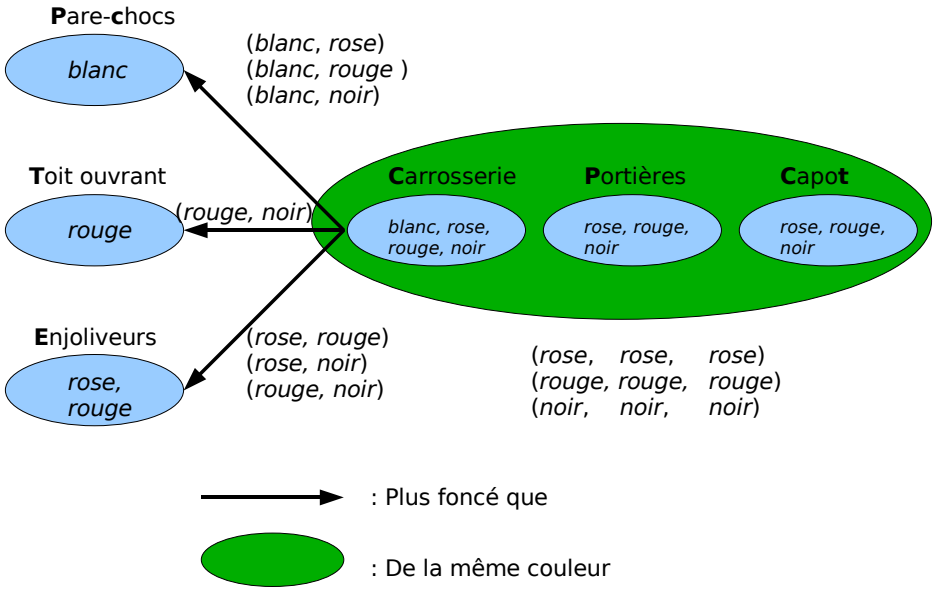


FIG. 2.2 – Le graphe de contraintes pour le problème de production de voitures.

2.2 Algorithmes de recherche complets

Tout comme pour le problème SAT, la majeure partie des solveurs de CSP complets est basée sur un algorithme énumératif de type backtrack. Ces algorithmes parcourent l'espace de toutes les instanciations possibles à la recherche d'une solution de manière systématique. Ils ont la garantie de trouver une solution s'il en existe une, ou de prouver l'inconsistance dans le cas contraire. Il existe également un certain nombre d'algorithmes incomplets utilisant les mêmes techniques que ceux traitant le problème SAT, mais nous ne nous attarderons pas dessus, étant donné que dans cette thèse nous ne nous intéressons qu'à la résolution complète de CSP.

Ici encore, pour palier le phénomène d'explosion combinatoire des méthodes de recherche complètes, un grand nombre de techniques ont vue le jour. C'est le cas notamment des méthodes de retour arrière non-chronologique : *BackJumping* [Gaschnig, 1979], *Conflict-Directed Back-Jumping* [Prosser, 1993, Ginsberg, 1993, Gent et Underwood, 1997], de méthodes d'apprentissage permettant de filtrer l'espace de recherche : *BackMarking* [Gaschnig, 1977], *Learning* [Dechter, 1990, Frost et Dechter, 1994], ou un mélange des deux : *BM-CBJ* [Kondrak et van Beek, 1997]. De plus, et c'est ce qui nous intéresse particulièrement dans cette thèse, un certain nombre de méthodes effectuent un filtrage des domaines des variables tout au long de la recherche, afin de rendre le problème réduit partiellement (ou localement) consistant. C'est le cas notamment des algorithmes comme *Forward Checking* [Haralick et Elliott, 1980] ou *MAC* [Gaschnig, 1979], [Sabin et Freuder, 1997] qui maintiennent à chaque noeud de l'arbre, une forme partielle ou complète de *constance d'arc*. Ces méthodes peuvent appliquer différents filtrages, en fonction du type de consistance considérée. Nous allons décrire les principaux types de consistances partielles.

2.3 Consistances locales et algorithmes de filtrage associés

Plusieurs techniques de filtrage par **consistances locales** ont été proposées pour améliorer l'efficacité des algorithmes de recherches. Ces techniques retirent des valeurs inconsistantes des domaines des variables et/ou induisent certaines contraintes implicites, permettant de réduire l'espace de recherche. Elles sont utilisées comme pré-traitements lorsque qu'elles détectent et retirent des inconsistances locales avant que la recherche ne commence, ou pendant la recherche pour élaguer l'arbre de recherche. Nous allons définir quelques consistances locales issues de la littérature CSP et présenter certains algorithmes de filtrages par consistances locales.

2.3.1 La consistance d'arc pour les CSP binaires

La plus largement utilisée des consistances locales est appelée **consistance d'arc** (AC) [Mackworth, 1977]. Un CSP binaire est arc consistant si, pour tout couple contraint de variables X, Y, pour toute valeur a de D_X il existe au moins une valeur b de D_Y telle que l'affectation (X, a) et (Y, b) satisfait la contrainte entre X et Y. Plus formellement, nous avons la définition suivante :

Définition 51
Un CSP binaire est **arc consistant** *si et seulement si* $\forall X \in \mathcal{X}, D_X \neq \emptyset$ *et* D_X *est arc-consistant. Un domaine* D_X *est arc-consistant si et seulement si* $\forall a \in D_X, \forall Y \in \mathcal{X}, \exists b \in D_Y$ *telle que* b *est un support pour* a *sur* R_{XY}*.*

Toutes les valeurs d'un domaine D_X qui ne sont pas arc consistantes peuvent être retirées, car elles ne peuvent faire partie d'une solution. Cette opération s'appelle le **filtrage par consistance d'arc**. Ce processus est devenu très important pour la résolution des CSP ces dernières années. Il est au cœur de nombreux langages de programmation par contraintes.

Plusieurs algorithmes de filtrage par consistance d'arc ont été proposés dans la littérature car ce filtrage peut être réalisé à moindre coût et peut considérablement simplifier certaines instances. Mackworth a proposé les premiers algorithmes basics AC-1, AC-2 et AC-3 [Mackworth, 1977]. La complexité en temps de AC-3 dans le pire des cas, qui est la meilleure d'entre eux, est en $\mathcal{O}(md^3)$[3] [Mackworth et Freuder, 1985]. La complexité spatiale de AC-3 dans le pire des cas est

[3]Rappelons que m est le nombre de contraintes et d la taille du plus grand domaine.

en $\mathcal{O}(md^2)$. Mohr et Henderson ont introduit AC-4 [Mohr et Henderson, 1986], qui a une complexité en temps dans le pire des cas en $\mathcal{O}(md^2)$ mais qui a un comportement en moyenne moins bon que AC-3. La complexité spatiale de AC-4 est aussi en $\mathcal{O}(md^2)$. Deux algorithmes AC-5 ont été proposés. Ils améliorent la complexité de AC-4 en $\mathcal{O}(md)$ sur des types de contraintes particulières mais ils se réduisent à AC-3 ou AC-4 dans le cas général [Deville et Hentenryck, 1991, Hentenryck et al., 1992, Perlin, 1992]. Bessière et Cordier on développé AC-6 qui améliore le comportement en moyenne de AC-3 mais qui conserve la complexité dans le pire des cas de AC-4 [Bessière et Cordier, 1993, Bessière, 1994]. Bessière et Régin on proposé AC-6++ qui est une amélioration de AC-6. Puis Bessière, Régin et Freuder ont introduit AC-Inference, un algorithme qui utilise des métaconnaisances pour réduire le nombre de tests de compatibilité, et AC-7, un algorithme d'arc consistance général implémentant cette idée [Bessière et al., 1995, Bessière et al., 1999]. Une implémentation soignée de cet algorithme permet d'obtenir une complexité spatiale dans le pire des cas en $\mathcal{O}(md)$. Enfin, Bessière et Régin ont introduit AC-2000 et AC-2001 qui sont tous deux des raffinements de AC-3 [Bessière et Regin, 2001]. AC-2000 possède les même complexité en temps et en espace que AC-3, mais effectue moins de tests de compatibilité en moyenne. AC-2001 a lui une complexité en temps optimale en $\mathcal{O}(md^2)$ grâce à une structure de donnée additionnelle qui ne modifie pas la borne de complexité en espace ($\mathcal{O}(md)$). Ce sont à l'heure actuelle les algorithmes les plus performants pour réaliser un filtrage par consistance d'arc.

2.3.2 Consistance d'arc pour les CSP n-aires

La définition de consistance d'arc a été étendue au CSP n-aires. Cette consistance est communément appelée **consistance d'arc généralisée** (GAC) [Mohr et Masini, 1988, Mohr, 1987]. Un CSP n-aire est arc consistant (généralisé) si pour toute variable X impliquée dans une contrainte d'arité a, toutes les valeurs du domaine D_X ont au moins un tuple support dans la relation associée à cette contrainte.

Quelques travaux ont été réalisés sur des algorithmes de filtrage par consistance d'arc généralisée. Mackworth a proposé l'algorithme CN qui est une généralisation de AC-3 aux contraintes non binaires [Mackworth, 1977]. Dans le pire des cas, sa complexité en temps est en $\mathcal{O}(ma^2d^{a+1})$. Mohr et Massini [Mohr, 1987, Mohr et Masini, 1988] ont proposé l'algorithme GAC-4 qui est une généralisation de AC-4 aux contraintes non binaires. Comme AC-4, il est basé sur l'idée de calculer le nombre de supports de chaque valeur de chaque domaine et de retirer celles qui ont un nombre de support nul. La complexité en temps dans le pire des cas est en $\mathcal{O}(md^a)$, ce qui est meilleur que CN, mais sa complexité en espace est mauvaise, à cause des listes de supports qui doivent être maintenues. Ces deux algorithmes ne sont que très rarement utilisés du fait de leurs mauvaises complexités.

Bessière et Régin ont proposé un algorithme général appelé *GAC-schema* [Bessière et Regin, 1997] qui permet de traiter des contraintes d'arité quelconque exprimées de différentes manières : en extension, sous forme d'expressions arithmétiques ou encore sous forme de prédicats sans sémantique particulière. Cet algorithme est basé sur l'algorithme AC-7, pour lequel un ordre sur les valeurs doit être connu a priori, ce qui permet, à la différence de GAC-4, de ne stocker des informations que sur le plus petit support valide pour une valeur au lieu de tous les supports, permettant un gain en espace considérable par rapport à GAC-4. Sa complexité en temps dans le pire des cas est en $\mathcal{O}(md^a)$, ce qui représente un gain de a^2d par rapport à CN.

Enfin, les mêmes auteurs ont conçu une version améliorée de *GAC-schema* [Bessière et Régin, 1999]. Dans cette version, certaines petites contraintes sont groupées pour former des contraintes plus larges. Ces contraintes sont ensuite considérées comme des sous-problèmes, sur lesquels il existe des moyens d'effectuer un filtrage par consistance d'arc plus rapidement que sur les contraintes initiales.

2.3.3 La consistance de chemin

La consistance de chemin fut introduite par Montanari [Montanari, 1974]. C'est une forme de consistance partielle plus forte que la consistance d'arc. La consistance de chemin impose que pour toute paire de valeurs a et b de deux variables X et Y telle que l'instanciation (X, a) et (Y, b) vérifie la contrainte C_{XY}, il existe une valeur pour chaque variable présente le long de tout chemin entre X et Y telle que toutes les contraintes impliquées dans ce chemin soient satisfaites. Montanary a montré qu'un CSP est chemin consistant si et seulement si tous les chemins de longueur 2 sont chemins consistants. Plus formellement, la consistance de chemin se définit ainsi :

Définition 52
On dit qu'une paire de variables $\{X, Y\}$ est chemin consistante si et seulement si $\forall (a, b) \in R_{XY}, \forall Z \in \mathcal{X}, \exists c \in D_Z$ telle que $(a, c) \in R_{XZ}$ et $(b, c) \in R_{YZ}$. Un CSP est chemin consistant si et seulement si $\forall X, Y \in \mathcal{X}$, la paire $\{X, Y\}$ est chemin consistante.

Alors que les algorithmes de filtrage par consistance d'arc suppriment dans les domaines les valeurs qui ne sont pas arc consistantes, les algorithmes de filtrage par consistance de chemin ont pour effet de supprimer des couples de valeurs dans les relations associées aux contraintes, et éventuellement de rajouter de nouvelles arêtes aux graphes de contraintes. Cette modification du graphe de contraintes est un obstacle à l'utilisation de ce filtrage dans la résolution pratique et efficace de CSP excepté pour ceux ayant des propriétés particulières. On utilise par exemple ce filtrage pour les CSP temporels, car ils sont par construction arc consistants, ce qui fait du filtrage par consistance de chemin le filtrage ayant le coût le plus supportable.

Quelques algorithmes de filtrage par consistance de chemin ont été proposés. Mackworth a conçu l'algorithme PC-2 [Mackworth, 1977], dont la complexité en temps dans le pire des cas est en $\mathcal{O}(n^3d^5)$ et en espace $\mathcal{O}(n^3)$. Puis, se basant sur les principes de l'algorithme AC-4, et en généralisant l'idée des supports pour la consistance de chemin, Han et Lee ont proposé l'algorithme PC-4 [Han et Lee, 1988], dont la complexité en temps dans le pire des cas est en $\mathcal{O}(n^3d^3)$ et en espace $\mathcal{O}(n^3d^3)$. Singh a ensuite proposé l'algorithme PC-5 [Singh, 1995], dont les complexité dans le pire des cas en temps et en espace sont respectivement $\mathcal{O}(n^3d^3)$ et $\mathcal{O}(n^3d^2)$. Enfin, Chmeiss et Jégou ont proposé l'algorithme PC-8 [Chmeiss et Jégou, 1996] pour lequel la complexité en espace est réduite à $\mathcal{O}(n^2d)$ alors que la complexité en temps devient $\mathcal{O}(n^3d^4)$.

2.3.4 Consistances d'ordre supérieur

Dans le cas de CSP binaires, Freuder a généralisé la notion de consistance partielle de telle sorte que les deux consistances définies jusque-là se trouvent être les deux premières consistances partielles de la classification qu'il a proposé [Freuder, 1978]. La consistance d'arc est la consistance de niveau deux, la consistance de chemin, celle de niveau trois. Plus généralement, on dit qu'un CSP est k-consistant si toute instanciation consistante de $k-1$ variables du CSP peut être

étendue de manière consistante à toute $k$ième variable. Plus formellement, la k-consistance est définie comme suit :

Définition 53
Étant donné un CSP P, P est k-consistant si et seulement si $\forall\{X_{i_1}, X_{i_2}, \ldots, \mathcal{X}_{i_{k-1}}\} = Y \subset \mathcal{X}, \forall(d_{i_1}, d_{i_2}, \ldots, d_{i_{k-1}}) \in D_{X_{i_1}} \times D_{X_{i_2}} \times \ldots \times D_{X_{i_{k-1}}}$, une instanciation consistante de Y, $\forall X_{i_k} \in \mathcal{X} - Y, \exists d_{i_k} \in D_{X_{i_k}}$ telle que $(d_{i_1}, d_{i_2}, \ldots, d_{i_{k-1}}, d_{i_k})$ est une instanciation consistante des variables de $Y \cup \{X_{i_k}\}$.

Remarquons tout de même que la n-consistance d'un CSP n'implique pas forcement sa consistance globale. En effet, s'il n'existe aucune instanciation consistante de $n-1$ variables, alors le CSP est n-consistant, or il n'existe pas de solution à un tel CSP pour autant. Ce n'est plus le cas lorsque l'on parle de k-consistance **forte**.

Définition 54
Un CSP P est **fortement** *k-consistant si et seulement si $\forall i, 1 < i < k, P$ est i-consistant.*

Nous constatons donc que si un CSP est fortement n-consistant, alors il est globalement consistant. Cette constatation montre l'intérêt d'algorithmes de filtrage par k-consistance. Cependant, la complexité de tels algorithmes croît exponentiellement en fonction de la taille de k, les rendant totalement inefficaces en pratique. C'est pourquoi les rares algorithmes de filtrage par k-consistance qui existent ne sont jamais utilisés. Rappelons que les algorithmes de chemin consistances ne le sont que très rarement, alors qu'il ne s'agit que de la 3-consistance.

Il existe une notion de consistance partielle encore plus générale que la k-consistance. Elle a été introduite par Freuder également [Freuder, 1985] et est notée (i,j)-consistance. Un CSP est (i,j)-consistant si toute instanciation consistante de i variables peut être étendue de manière consistante sur j variables supplémentaires. Formellement cela donne :

Définition 55
Étant donné un CSP P, P est (i,j)-consistant si et seulement si $\forall\{X_{k_1}, X_{k_2}, \ldots, \mathcal{X}_{k_i}\} = Y \subset \mathcal{X}, \forall(d_{k_1}, d_{k_2}, \ldots, d_{k_i}) \in D_{X_{k_1}} \times D_{X_{k_2}} \times \ldots \times D_{X_{k_i}}$, une instanciation consistante de Y, $\forall X_{l_1}, X_{l_2}, \ldots, X_{l_j} = Y' \in \mathcal{X} - Y, \exists d_{l_1}, d_{l_2}, \ldots, d_{l_j} \in D_{X_{l_1}}, D_{X_{l_2}}, \ldots, D_{X_{l_j}}$ telle que $(d_{k_1}, d_{k_2}, \ldots, d_{k_i}, d_{l_1}, d_{l_2}, \ldots, d_{l_j})$ est une instanciation consistante des variables de $Y \cup Y'$.

La plupart des consistances peuvent s'exprimer avec des valeurs particulières pour i et j :
- k-consistance : $i = k-1$ et $j = 1$, c'est la $(k-1,1)$-consistance.
- consistance de chemin : $i = 2$ et $j = 1$, c'est la $(2,1)$-consistance.
- consistance d'arc : $i = 1$ et $j = 1$, c'est la $(1,1)$-consistance.

Ces définitions n'ont pas d'autre intérêt que la formalisation d'un opérateur de filtrage générique, car en pratique, seule la consistance d'arc ($(1,1)$-consistance) est largement utilisée parmi toutes ces consistances.

2.3.5 La consistance de chemin restreinte

Comme nous venons de le voir, le filtrage par consistance de chemin n'est pas applicable en pratique en général, du fait de sa complexité. C'est pourquoi Berlandier a proposé une forme **restreinte** de consistance de chemin [Berlandier, 1995], toujours pour les CSP binaires. Le filtrage

associé à cette nouvelle consistance supprime plus de valeurs inconsistantes que la consistance d'arc, mais en même temps, il ne souffre pas des inconvénients des filtrages d'ordre supérieur. L'algorithme de filtrage supprime toutes les valeurs qui ne sont pas arc consistantes, et de plus, il vérifie que chaque couple de valeurs a et b pour des variables X et Y tel que b est l'unique support de a dans la relation R_{XY} est chemin consistant. Si un tel couple n'est pas chemin consistant, alors la valeur a est retirée du domaine D_X de X, car la suppression du tuple (a, b) dans la relation R_{XY} entraîne la perte de tous les supports de a dans cette contrainte. Ces suppressions supplémentaires font du filtrage par consistance de chemin restreinte un filtrage plus fort que le filtrage par consistance d'arc, mais qui reste moins coûteux que le filtrage par consistance de chemin. De plus, la structure du graphe de contrainte n'est en rien altérée par ce filtrage. La définition formelle de la consistance de chemin restreinte est la suivante :

Définition 56
Un CSP vérifie la propriété de consistance de chemin restreinte si et seulement si $\forall X \in \mathcal{X}, D_X \neq \emptyset$*,* D_X *est arc consistant et* $\forall X, Y \in \mathcal{X}, \forall (a, b) \in R_{XY}$ *tel que* b *est l'unique support de* a *dans* D_Y *(c'est-à-dire que* $a \notin R_{XY} \setminus (a, b)$*),* $\forall Z \in \mathcal{X}$ *telle que* $C_{XZ}, C_{YZ} \in \mathcal{C}, \exists c \in D_Z$ *tel que* c *est un support de* a *dans* C_{XZ} *et* c *est un support de* b *dans* C_{YZ} *(c'est-à-dire que* $(a, c) \in R_{XZ}$ *et* $(b, c) \in R_{YZ}$*).*

Debruyne et Bessière ont étendu cette notion de consistance de chemin restreinte à la notion de k-**consistance de chemin restreinte** (k-RPC) [Debruyne et Bessière, 1997a]. L'algorithme de filtrage par k-consistance de chemin restreinte effectue le même filtrage que pour la consistance de chemin restreinte, mais teste la consistance de chemin pour les couples de valeurs ayant k supports dans une contrainte, au lieu d'un seul. Ainsi, la consistance de chemin restreinte devient la 1-consistance de chemin restreinte dans cette généralisation, et la consistance d'arc devient la 0-consistance de chemin restreinte.

Dans ce même article, ils décrivent un algorithme permettant d'effectuer un filtrage par (1-)consistance de chemin restreinte : RPC-2, dont la complexité dans le pire des cas en temps est en $\mathcal{O}(mnd^2)$ et celle en espace en $\mathcal{O}(mnd)$.

Toujours dans ce même article, Debruyne et Bessière ont introduit la notion de consistance de chemin restreinte *maximum* (Max-RPC). Un graphe de contrainte vérifie la propriété de consistance de chemin restreinte maximum si chaque valeur de chaque domaine possède au moins un support pour la consistance de chemin par contrainte, quelle que soit le nombre de supports pour la consistance de chemin qu'elle possède. D'après cette définition, un algorithme de filtrage par consistance de chemin restreinte maximum doit supprimer toutes les valeurs qui ne vérifient pas la k-consistance de chemin restreinte, pour toute valeur de k. Ils ont ainsi proposé un algorithme effectuant ce traitement dont la complexité dans le pire des cas en temps est en $\mathcal{O}(mnd^3)$ et celle en espace en $\mathcal{O}(mnd)$.

Grandoni et Italiano [Grandoni et Italiano, 2003] ont proposé deux algorithmes de filtrage par consistance de chemin restreinte maximum. Le premier, Max-RPC-2, a une complexité en espace optimale en $\mathcal{O}(md)$ avec la même complexité en temps. Le second, Max-RPC-2', a une meilleur complexité en temps grâce à l'utilisation de multiplications de matrices rapides ($\mathcal{O}(mnd^{2.575})$), au détriment d'une complexité en espace accrue $\mathcal{O}(mnd^2)$.

2.3.6 Les consistances singletons

Debruyne et Bessière ont introduit d'autres notions de consistance partielle pour les CSP binaires appelées consistances singleton [Debruyne et Bessière, 1997b]. Ces consistances sont basées sur la constatation que si une valeur a pour une variable X est consistante (c'est-à-dire qu'elle appartient à une solution), alors le CSP obtenu en restreignant le domaine D_X de la variable X au singleton $\{a\}$ est consistant. Notons ce CSP $P|_{D_X=\{a\}}$. Si $P|_{D_X=\{a\}}$ n'est pas consistant, alors la valeur a peut être retirée du domaine D_X. Évidemment, déterminer si $P|_{D_X=\{a\}}$ est inconsistant est bien trop coûteux, c'est pourquoi ils se restreignent à déterminer si une consistance locale est vérifiée. Ainsi, l'algorithme de filtrage par consistance singleton supprimera la valeur a du domaine d'une valeur X si $P|_{D_X=\{a\}}$ est localement inconsistant.

L'exemple de consistance singleton traité dans ces travaux est la *consistance d'arc singleton* (SAC), dont la définition formelle est la suivante :

Définition 57
Un CSP P vérifie la consistance d'arc singleton si et seulement si $\forall X \in \mathcal{X}, D_X \neq \emptyset$, D_X est arc consistant et $\forall a \in D_X, P|_{D_X=\{a\}}$ est arc consistant.

Pour tester si une valeur a pour une variable X est singleton arc consistante, l'algorithme qu'ils proposent instancie X à a et applique un algorithme de filtrage par consistance d'arc sur le sous-problème obtenu. Si un domaine se vide entièrement, alors la valeur a peut être retirée du domaine de X. N'importe quel algorithme de filtrage par consistance d'arc peut être utilisé, tant que sa complexité reste polynomiale, leur algorithme de filtrage par consistance d'arc singleton garde une complexité polynomiale. Un premier algorithme a été proposé par Debruyne et Bessière dans [Debruyne et Bessière, 1997b] : SAC-1 dont les complexités en temps et en espace sont respectivement en $\mathcal{O}(mn^2d^4)$ et en $\mathcal{O}(md)$. Puis, l'algorithme SAC-2 a été proposé par Barták et Erben [Barták et Erben, 2004], améliorant la complexité en espace qui devient $\mathcal{O}(mn^2d^2)$. Enfin, Debruyne et Bessière ont proposé deux algorithmes dans [Debruyne et Bessière, 2005] : le premier, SAC-*OPT*, avec une complexité en temps optimale en $\mathcal{O}(mnd^3)$ et une complexité en espace en $\mathcal{O}(mnd^2)$, et le second, SAC-*SDS*, qui admet un compromis sur la complexité en temps ($\mathcal{O}(mnd^4)$) pour réduire la complexité en espace ($\mathcal{O}(mn^2d^2)$)) afin de permettre son utilisation sur des instances de plus grandes tailles.

Tous ces différents algorithmes sont obtenus en changeant l'algorithme de filtrage par consistance d'arc utilisé pour filtrer par consistance d'arc singleton (AC-4, AC-6, AC-2001...). Cependant, toute autre sorte de filtrage supprimant des valeurs dans les domaines des variables pourrait être utilisée comme base de consistance singleton (SPC : chemin consistance singleton, SRPC : chemin consistance restreinte singleton...).

2.3.7 Les consistances inverses

Les consistances inverses ont été introduites par Freuder et Elfe [Freuder et Elfe, 1996] pour les CSP binaires. Un filtrage par consistance inverse supprime, dans les domaines des variables, les valeurs qui ne sont compatibles avec aucune instanciation consistante d'un quelconque ensemble de variables supplémentaires. Les consistances inverses peuvent être définies en terme de (i,j)-consistances. Lorsque i vaut 1 et que j vaut $k-1$, cela donne la k consistance inverse. Les consistances où $i > 1$, comme la consistance de chemin, identifient et enregistrent des combinaisons de valeurs pour des variables pour lesquelles une valeur consistante pour des variables

supplémentaires ne peut être trouvée. Ce qui rajoute des contraintes non unaires au problème, résultant en un accroissement de l'espace mémoire qui devient rapidement prohibitif. À l'inverse, les consistances inverses ne stockent que des contraintes unaires puisqu'elles filtrent les domaines. La complexité spatiale reste donc linéaire.

La consistance inverse la plus simple est la consistance de chemin inverse (PIC), car l'arc consistance ((1,1)-consistance) est équivalente à la consistance d'arc inverse. Cette consistance est équivalente à la (1,2)-consistance. D'après sa définition [Freuder et Elfe, 1996], une valeur a d'une variable X d'un CSP est chemin consistante inverse si elle peut être étendue de manière consistante à tout triplé de variables incluant X. Dans [Debruyne et Bessière, 1997b], Debruyne et Bessière ont montré qu'il n'était pas nécessaire de tester cette condition pour toutes les valeurs pour s'assurer la consistance de chemin inverse. Ils ont montré qu'un CSP vérifie la consistance de chemin inverse si et seulement si il est arc consistant et si pour toute valeur a d'une variable X, pour toute clique de taille 3 (X, Y, Z), l'affectation de X à a peut être étendue en une instanciation consistante de cette 3-clique. Ainsi, sa définition formelle est la suivante :

Définition 58
Un CSP satisfait la propriété de consistance de chemin inverse si et seulement si $\forall X \in \mathcal{X}, \forall a \in D_X, \forall Y, Z \in \mathcal{X}$ *tel que* $Y \neq X \neq Z \neq Y, \exists b \in D_Y$ *et* $c \in D_Z$ *tel que* b *est un support pour* a *sur* C_{XY}*,* c *est un support pour* a *sur* C_{XZ} *et* c *est un support pour* b *sur* C_{YZ}*.*

Un premier algorithme de filtrage par consistance de chemin inverse fut proposé par Freuder et Elfe dans [Freuder et Elfe, 1996] : PIC-1. Sa complexité en temps est en $\mathcal{O}(mn^2d^4)$ et en espace $\mathcal{O}(n)$. Par la suite, Debruyne a proposé un algorithme ayant une complexité en temps optimale : $\mathcal{O}(mnd^3)$ et une complexité en espace accrue : $\mathcal{O}(md + cd)$ [Debruyne, 2000]. Dans le calcul de cette complexité, c représente le nombre de 3-cliques présentes dans le graphe de contraintes.

Freuder et Elfe ont également introduit le concept de **consistance de voisinage inverse** (NIC) [Freuder et Elfe, 1996]. Un CSP satisfait la propriété de consistance de voisinage inverse si pour chaque valeur a de chaque variable X, il est possible de trouver un solution au sous-problème induit par le voisinage de X qui est compatible avec a.

Définition 59
Soit $X \in \mathcal{X}$*. On définit le voisinage de* X *par* $\mathcal{V}(X) = \{Y \in \mathcal{X} | \exists C_{XY} \in \mathcal{C}\}$*. Un CSP satisfait la propriété de consistance de voisinage inverse si et seulement si* $\forall X \in \mathcal{X}, \forall a \in D_X$*, l'instanciation* (X, a) *peut être étendue en une instanciation consistante de* $\mathcal{V}(X)$*.*

L'idée sous-jacente de ce filtrage est d'adapter le niveau de la k-consistance inverse en fonction du nombre de contraintes dans lesquelles les variables sont impliquées. Ils ont proposé un algorithme de filtrage par consistance de voisinage inverse dont la complexité en temps est en $\mathcal{O}(g^2(n + md)d^{g+1})$ et en espace $\mathcal{O}(n)$. Dans ce calcul de complexité, g représente le degré maximum d'une variable (c'est-à-dire le nombre de contraintes maximum dans lesquelles elle est impliquée). Cette complexité dépend donc fortement du degré de contrainte des variables, ce qui explique leur idée de moduler le filtrage en fonction du degré de chaque variable dans le cas de CSP où les variables ne sont pas uniformément contraintes.

Un certain nombre de consistances locales inverses ont été introduites dans [Verfaillie et al., 1999]. Il s'agit précisément d'un cadre théorique générique qui permet de capturer la plupart des consistances inverses existantes, mais aussi de définir un nombre arbitraire de consistances inverses, ainsi

que la généralisation de ces consistances aux contraintes n-aires. Étant donné qu'aucun algorithme de filtrage par ces consistances inverses générales n'a été implémenté, nous n'entrerons pas dans les détails théoriques de ces dernières.

2.3.8 Les consistances relationnelles

Les dernières formes de consistances locales que nous allons décrire portent le nom de **consistances relationnelles**. Elles ont été introduites par van Beek et Dechter dans [van Beek et Dechter, 1995]. Il s'agit de nouvelles définitions plus fortes pour la consistance d'arc et de chemin pour les CSP n-aires. La consistance d'arc **relationnelle** peut être établie en temps polynomial en fonction de l'arité des contraintes, alors qu'établir la consistance de chemin **relationnelle** est un problème NP-Complet. La définition formelle de la consistance d'arc relationnelle est la suivante :

Définition 60
Soit R_S une relation associée à une contrainte C_S portant sur un ensemble $S \subset \mathcal{X}$ de variables d'un CSP n-aire. R_S est relationnellement arc consistance si et seulement si toute instanciation consistante de toutes les variables de S sauf une peut être étendue à cette dernière de telle sorte que C_S soit satisfaite.

Un CSP satisfait est relationnellement arc consistant si toutes les relations associées à ses contraintes sont relationnellement arc consistantes.

Celle de la consistance de chemin relationnelle est la suivante :

Définition 61
Soient R_{S_1} et R_{S_2} deux relations associées à deux contraintes C_{S_1} et C_{S_2} portant respectivement sur deux ensembles $S_1 \subset \mathcal{X}$ et $S_2 \subset \mathcal{X}$ de variables. Le couple de relations (R_{S_1}, R_{S_2}) est relationnellement chemin consistant si et seulement si $\forall X \in S_1 \cap S_2$, toute instanciation consistante des variables de $S_1 \cup S_2) - \{X\}$ peut être étendue pour inclure X de telle sorte que C_{S_1} et C_{S_2} soient simultanément satisfaites.

Un CSP est relationnellement chemin consistant si et seulement si tout couple de relations (R_{S_i}, R_{S_j}) est relationnellement chemin consistant.

Par la suite, Bessière Hebrard et Walsh ont encore généralisé la notion de consistance d'arc relationnelle, donnant lieu à la **k-consistance d'arc relationnelle** [Bessière et al., 2003]. Cette consistance est la restriction de la consistance d'arc relationnelle à des ensembles de variables de taille k.

Définition 62
Soit R_S une relation associée à une contrainte C_S portant sur un ensemble $S \subset \mathcal{X}$ de variables d'un CSP n-aire. R_S satisfait la propriété de k-consistante d'arc relationnelle si et seulement si $\forall A \subset S$ tel que $|A| = k$, toute instanciation consistante des variables de A peut être étendue en une instanciation consistante sur l'ensemble des variables de S.

Un CSP satisfait la propriété de k-consistance d'arc relationnelle si et seulement si toutes les relations associées aux contraintes de ce CSP satisfont la propriété de k-consistance d'arc relationnelle.

Du fait de sa NP-complétude, la consistance de chemin relationnelle n'est rarement, si ce n'est jamais, utilisée. À notre connaissance, il en est de même pour la consistance d'arc relationnelle et la k-consistance d'arc relationnelle pour lesquelles aucun algorithme de filtrage n'a été proposé.

2.3.9 Algorithmes de filtrage spécifiques

Plusieurs algorithmes de filtrage spécifiques ont été proposés dans la littérature. Dans la plupart des cas, ces algorithmes effectuent un filtrage par consistance d'arc généralisée sur des contraintes n-aires présentant des propriétés particulières. L'intérêt étant que ces algorithmes dédiés ont une complexité en temps bien meilleure que celle d'algorithmes génériques comme GAC-schema.

C'est le cas notamment d'un certain nombre de contraintes dites **globales**. Passer en revue les algorithmes spécifiques pour toutes les contraintes globales dépasserait le cadre de cette thèse, mais un lecteur intéressé par ce sujet pourra trouver les bases dans les articles suivants[4] : [Régin, 1994] : la contrainte *all-different*, [Beldiceanu et Contjean, 1994] les contraintes *cumulatives*, [Régin, 1996] : les contraintes globales de *cardinalité*, [Régin et Puget, 1997] : les contraintes globales d'*ordonnancement*, [Régin, 1999] : les contraintes *all-different symétriques*, [Beldiceanu et Contjean, 1994] : les contraintes de *cycle*, [Bessière et al., 2006a] : les contraintes *Nvaluées*, ou encore [Bessière et al., 2006b] : la contrainte *racine*...

Il est à noter que l'algorithme AC-5 [Hentenryck et al., 1992] peut être paramétré pour obtenir une complexité en temps en $\mathcal{O}(md)$ pour une classe importante de contraintes : les contraintes *fonctionnelles*, *anti-fonctionnelles* et *monotones*.

2.3.10 Récapitulatif

Le filtrage par consistance locale est la clé de voûte de la plupart des solveurs de CSP actuels. Le plus utilisé est le filtrage par consistance d'arc, pour lequel un nombre important d'algorithme a été proposé, du fait de son faible coût. Mais la tendance actuelle s'oriente vers des filtrages par consistances un peu plus fortes, toujours avec un coût raisonnable. Le tableau 2.2 propose une comparaison entre les complexités en temps et en espace des différents algorithmes pour les différents filtrages par consistances locales que nous avons décrits.

[4] Liste non exhaustive.

Nom de l'algorithme	**Complexité en temps**	**Complexité en espace**
AC-3 [Mackworth, 1977]	$\mathcal{O}(md^3)$	$\mathcal{O}(m+nd)$
AC-4 [Mohr et Henderson, 1986]	$\mathcal{O}(md^2)$	$\mathcal{O}(md^2)$
AC-5 [Hentenryck et al., 1992]	$\mathcal{O}(md^2)$	$\mathcal{O}(md)$
AC-6 [Bessière et Cordier, 1993]	$\mathcal{O}(md^2)$	$\mathcal{O}(md)$
AC-7 [Bessière et al., 1995]	$\mathcal{O}(md^2)$	$\mathcal{O}(md)$
AC-2000 [Bessière et Regin, 2001]	$\mathcal{O}(md^3)$	$\mathcal{O}(md)$
AC-2001 [Bessière et Regin, 2001]	$\mathcal{O}(md^2)$	$\mathcal{O}(md)$
CN(GAC-3) [Mackworth, 1977]	$\mathcal{O}(ma^2d^{a+1})$	$\mathcal{O}(ma+nd)$
GAC-4 [Mohr et Masini, 1988]	$\mathcal{O}(md^a)$	$\mathcal{O}(md^a+nd)$
GAC-schema [Bessière et Regin, 1997]	$\mathcal{O}(md^a)$	$\mathcal{O}(ma^2d)$
PC-2 [Mackworth, 1977]	$\mathcal{O}(n^3d^5)$	$\mathcal{O}(n^3)$
PC-4 [Han et Lee, 1988]	$\mathcal{O}(n^3d^3)$	$\mathcal{O}(n^3d^3)$
PC-5 [Singh, 1995]	$\mathcal{O}(n^3d^3)$	$\mathcal{O}(n^3d^3)$
PC-8 [Chmeiss et Jégou, 1996]	$\mathcal{O}(n^3d^4)$	$\mathcal{O}(n^2d)$
RPC-1 [Berlandier, 1995]	$\mathcal{O}(mnd^3)$	$\mathcal{O}(mnd)$
RPC-2 [Debruyne et Bessière, 1997a]	$\mathcal{O}(mnd^2)$	$\mathcal{O}(mnd)$
Max-RPC-1 [Debruyne et Bessière, 1997a]	$\mathcal{O}(mnd^3)$	$\mathcal{O}(mnd)$
Max-RPC-2 [Grandoni et Italiano, 2003]	$\mathcal{O}(mnd^3)$	$\mathcal{O}(md)$
Max-RPC-2' [Grandoni et Italiano, 2003]	$\mathcal{O}(mnd^{2.575})$	$\mathcal{O}(mnd^2)$
SAC-1 [Debruyne et Bessière, 1997b]	$\mathcal{O}(mn^2d^4)$	$\mathcal{O}(md)$
SAC-2 [Barták et Erben, 2004]	$\mathcal{O}(mn^2d^4)$	$\mathcal{O}(n^2d^2)$
SAC-OPT [Debruyne et Bessière, 2005]	$\mathcal{O}(mnd^3)$	$\mathcal{O}(mnd^2)$
SAC-SDS [Debruyne et Bessière, 2005]	$\mathcal{O}(mnd^4)$	$\mathcal{O}(n^2d^2)$
PIC-1 [Freuder et Elfe, 1996]	$\mathcal{O}(mn^2d^4)$	$\mathcal{O}(n)$
PIC-2 [Debruyne, 2000]	$\mathcal{O}(mnd^3)$	$\mathcal{O}(md+cd)$
NIC [Freuder et Elfe, 1996]	$\mathcal{O}(g^2(n+md)d^{g+1})$	$\mathcal{O}(n)$

TAB. 2.2 – Tableau comparatif des complexités en temps et en espace des différents algorithmes de filtrage par consistances partielles pour les CSP. m représente le nombre de contraintes du CSP, n le nombre de variables du CSP, d la taille du plus grand domaine, c est le nombre de 3-cliques présentes dans le graphe de contraintes, g est le degré maximum d'une variable et a est la plus grande arité des contraintes.

Chapitre 3

Transformations entre formalismes

Sommaire

Comme nous avons pu le constater, les deux formalismes que nous venons de décrire présentent des similitudes. Les algorithmes de résolution parcourent pour SAT un arbre de recherche binaire, pour les CSP un arbre de recherche quelconque. Dans les deux formalismes, l'accent est mis sur les différentes méthodes de filtrage et de propagation : consistances locales et propagation de contraintes pour les CSP, résolution bornée et propagation unitaire pour SAT. Mais des liens encore plus forts ont été mis en évidence. Plusieurs travaux ont montré qu'il est possible d'exprimer n'importe quel problème SAT sous forme de CSP et vice-versa. Génisson et Jégou ont montré que, pour un problème donné, les algorithmes Forward Checking et Davis et Putam explorent des arbres équivalents selon la représentation choisie pour le problème [Génisson et Jégou, 1996]. Nous allons maintenant détailler les différentes transformations connues entre ces deux formalismes. Ces travaux ont permis de comparer les filtrages par consistances locales des CSP avec la propagation unitaire ou la résolution bornée de SAT.

3.1 SAT → CSP

Il existe quatre transformations connues pour exprimer un problème SAT sous forme de CSP. Les trois premières, *le codage des littéraux*, *le codage dual* et *le codage avec variables cachées* transforme une instance SAT en un CSP binaire, alors que *le codage non-binaire* transforme une instance SAT en un CSP n-aire [Bennaceur, 1996, Walsh, 2000a].

3.1.1 Le codage des littéraux

Ce codage fut introduit par Bennaceur [Bennaceur, 1996, Bennaceur, 2004]. Considérons une formule sous forme CNF ayant n variables et m clauses. Dans ce codage, on associe une variable X_i à chaque clause c_i de la CNF. Le domaine D_{X_i} de la variable X_i contient les littéraux de la clause c_i à laquelle X_i est associée. Par exemple, si la formule CNF contient la clause $c_i = l_1 \vee \neg l_2 \vee l_3 \vee \neg l_4$ alors le domaine D_{X_i} de la variable associée est $D_{X_i} = \{l_1, \neg l_2, l_3, \neg l_4\}$.

Une contrainte binaire $C_{X_iX_j}$ impliquant deux variables X_i et X_j associées à deux clauses c_i et c_j est créée si la clause c_i contient un littéral et la clause c_j contient son opposé. Par exemple, si $c_i = l_1 \vee \neg l_2 \vee l_3$ et $c_j = \neg l_1 \vee l_3 \vee l_4$, alors la contrainte $C_{X_iX_j}$ est créée car c_i contient l_1 et c_j contient $\neg l_1$.

Pour chaque contrainte $C_{X_iX_j}$ on associe la relation $R_{X_iX_j}$ définie comme le produit cartésien $D_{X_i} \times D_{X_j}$ auquel on retire l'ensemble des couples (l_i, l_j) tels que l_i est le littéral opposé à l_j ($l_i = \neg l_j$). Par exemple, la relation $R_{X_iX_j}$ associée à la contrainte ci-dessus est définie par le tableau suivant :

$R_{X_iX_j}$	
X_i	X_j
l_1	l_3
l_1	l_4
$\neg l_2$	$\neg l_1$
$\neg l_2$	l_3
$\neg l_2$	l_4
l_3	$\neg l_1$
l_3	l_3
l_3	l_4

Soit Σ une instance k-SAT et $\mathcal{P}$ le CSP résultant du codage de Σ. Le nombre de variables de $\mathcal{P}$ est égal au nombre de clauses de Σ ($nbCla(\Sigma) = m$), la taille des domaines de $\mathcal{P}$ est bornée par k, et la taille des relations est bornée par $k^2 - 1$. La complexité de la transformation est en $\mathcal{O}(mk^2)$. Ce codage présente quelques propriétés intéressantes. Pour commencer, si chaque clause de la formule Σ de départ contient au moins 2 littéraux, alors le CSP résultant de son codage est arc-consistant. Ainsi, appliquer la propagation unitaire sur la formule de départ est équivalent à effectuer un filtrage par consistance d'arc sur son codage en CSP. De même, si toutes les clauses de Σ contiennent au moins 3 littéraux, alors le CSP résultant de son codage est chemin consistant, et même fortement chemin consistant.

Dans [Walsh, 2000a], Walsh montre qu'un algorithme DP appliqué à la formule de départ visite strictement moins de nœuds qu'un MAC sur le CSP codé, à condition d'utiliser des heuristiques de branchement équivalentes.

Dans [Dimopoulos et Stergiou, 2006], Dimopoulos et Stergiou se sont intéressés à une consistance plus forte que la consistance d'arc : la consistance de chemin inverse (PIC). Ils ont montré que l'application de la règle de *résolution binaire* sur la formule Σ est strictement plus forte que l'application d'un filtrage par consistance de chemin inverse sur le CSP $\mathcal{P}$.

Les trois codages qui suivent sont décris et étudiés de manière systématique par Walsh dans [Walsh, 2000a].

3.1.2 Le codage dual

Dans ce codage, une variable duale X_i est associée à chaque clause c_i. Le domaine D_{X_i} associé à X_i est composé des tuples de valeurs de vérité qui satisfont la clause c_i. Par exemple, la variable X_i associée à la clause $c_i = l_1 \vee l_3$ a pour domaine $D_{X_i} = \{\langle V, F\rangle, \langle F, V\rangle, \langle V, V\rangle\}$. Ce sont les interprétations de l_1 et l_2 qui satisfont la clause c_i. Une contrainte binaire est créée entre deux variables duales X_i et X_j qui sont associées à deux clauses partageant une ou plusieurs variables propositionnelles. Par exemple, entre les variables duales X_i et X_j associées aux clauses $c_i = l_1 \vee l_3$ et $c_j = l_2 \vee \neg l_3$, la contrainte $C_{X_iX_j}$ est ajoutée. Toute relation $R_{X_iX_j}$ associée à une contrainte $C_{X_iX_j}$ assure que les variables propositionnelles partagées par les deux clauses prennent les même valeurs dans les tuples des domaines des deux variables CSP X_i et X_j si les variables propositionnelles apparaissent avec la même polarité dans les clauses c_i et c_j, ou bien des valeurs complémentaires si elles apparaissent avec des polarités opposées. Dans notre exemple, nous avons $D_{X_j} = \{\langle V, V\rangle, \langle F, F\rangle, \langle V, F\rangle\}$. Étant donné que la variable propositionnelle l_3 apparaît avec des polarités opposées dans c_i et c_j, la relation $R_{X_iX_j}$ assure que les couples de valeurs compatibles pour X_i et X_j ont les valeurs de vérité de leurs secondes composantes[1] opposées, ce qui donne :

$R_{X_iX_j}$	
X_i	X_j
$\langle V, F\rangle$	$\langle V, V\rangle$
$\langle F, V\rangle$	$\langle F, F\rangle$
$\langle F, V\rangle$	$\langle F, V\rangle$
$\langle V, V\rangle$	$\langle F, F\rangle$
$\langle V, V\rangle$	$\langle F, V\rangle$

Soit Σ une instance k-SAT et $\mathcal{P}$ le CSP résultant du codage dual de Σ. Walsh a montré que filtrer par consistance d'arc le CSP $\mathcal{P}$ est plus fort que réaliser la propagation unitaire sur Σ. C'est à dire que si la propagation unitaire produit une clause vide, le filtrage par consistance d'arc fera apparaître un domaine vide, alors que l'inverse n'est pas toujours vrai. De plus, si la propagation unitaire sur Σ aboutit à l'interprétation de certaines variables, alors un filtrage par consistance d'arc supprimera toutes les valeurs contradictoires avec cette interprétation.

Il montre aussi que, moyennant des heuristiques de branchement équivalentes, appliquer un DP sur Σ parcourt strictement moins de nœuds que FC appliqué sur $\mathcal{P}$.

Dans [Dimopoulos et Stergiou, 2006], Dimopoulos et Stergiou ont montré que les filtrages par consistance d'arc (AC) et par consistance de chemin inverse (PIC) sur $\mathcal{P}$ peuvent être retrouvés dans Σ par des procédés de *résolution d'ensemble* et de *résolution d'ensemble étendue* respectivement.

3.1.3 Le codage avec variables cachées

Dans ce codage, les mêmes variables CSP *duales* sont associées à chaque clause de la formule CNF, auxquelles sont associés les mêmes domaines composés des tuples de valeurs de vérité satisfaisant la clause de départ. En plus de ces variables (duales), on rajoute autant de variables

[1] Celles correspondantes à l_3.

CSP (propositionnelles) qu'il y a de variables booléennes, avec pour domaine les valeurs *vrai* et *faux*. Ainsi, pour chaque variable booléenne l_j, on associe X_{l_j} telle que $D_{X_{l_j}} = \{V, F\}$. Une contrainte binaire est posée entre une variable CSP duale et une variable CSP propositionnelle si la clause à laquelle est associée la variable duale contient une occurrence (positive ou négative) de la variable booléenne à laquelle est associée la variable CSP propositionnelle. Dans le cas où la variable booléenne apparaît positivement dans la clause, la relation associée à cette contrainte assure que la valeur de la composante des tuples des variables CSP duales correspondant à la variable propositionnelle est identique à celle de la valeur de la variable CSP propositionnelle. Dans le cas contraire, la relation assure que ces valeurs sont complémentaires. Par exemple, en considérant la clause $c_i = l_2 \vee \neg l_3$, on pose les contraintes $C_{X_i X_{l_2}}$ et $C_{X_i X_{l_3}}$ et la relation associée $R_{X_i X_{l_2}}$ et $R_{X_i X_{l_3}}$ suivante :

$R_{X_i X_{l_2}}$	
X_i	X_{l_2}
$\langle V, V \rangle$	V
$\langle F, F \rangle$	F
$\langle V, F \rangle$	F

$R_{X_i X_{l_3}}$	
X_i	X_{l_3}
$\langle V, V \rangle$	F
$\langle F, F \rangle$	V
$\langle V, F \rangle$	V

Il n'y a pas de contrainte entre deux variables duales. La complexité des codages dual et avec variables cachées est plus importante que celle du codage des littéraux. Soient Σ une instance k-SAT et $\mathcal{P}_{Dual}$ le CSP résultant du codage dual de Σ, $\mathcal{P}_{Var}$ le CSP résultant du codage avec variables cachées de Σ et $\mathcal{P}_{Lit}$ le CSP résultant du codage des littéraux de Σ. Alors que la taille des domaines de $\mathcal{P}_{Lit}$ est en $\mathcal{O}(k)$, la taille des domaines de $\mathcal{P}_{Dual}$ et de $\mathcal{P}_{Var}$ est en $\mathcal{O}(2^k)$. De même, la complexité en espace des relations pour $\mathcal{P}_{Lit}$ est en $\mathcal{O}(k^2)$, alors que celle des relations de $\mathcal{P}_{Dual}$ et de $\mathcal{P}_{Var}$ est en $\mathcal{O}(2^k)$.

Walsh [Walsh, 2000a] a montré qu'un filtrage par consistance d'arc sur $\mathcal{P}_{Var}$ est équivalent à la propagation unitaire sur Σ et qu'un DP appliqué sur Σ visite les même nœuds que FC ou MAC appliqué sur $\mathcal{P}_{Var}$.

3.1.4 Le codage non-binaire

Dans ce codage, on retrouve les variables CSP propositionnelles du codage précédent. C'est-à-dire qu'une variable CSP est associée à chaque variable booléenne, et n'a que deux valeurs dans son domaine : *Vrai* et *Faux*. Une contrainte non binaire est posée entre toutes les variables CSP pour lesquelles les variables booléennes associées apparaissent en même temps dans une clause. La relation associée à chaque contrainte est constituée de tuples *interdits*, ces tuples représentent les combinaisons de valeurs de vérité qui falsifient la clause concernée. Par exemple, en considérant la clause $C_i = l_1 \vee l_2 \vee \neg l_3$, nous obtenons la contrainte $C_{X_{l_1} X_{l_2} X_{l_3}}$ entre les variables CSP associées aux variables booléennes l_1, l_2 et l_3, dont la relation associée $R_{X_{l_1} X_{l_2} X_{l_3}}$ est donnée par :

$R_{X_{l_1} X_{l_2} X_{l_3}}$		
X_{l_1}	X_{l_2}	X_{l_3}
F	F	V

Soit Σ une formule k-SAT et $\mathcal{P}$ le CSP résultant du codage non binaire de Σ. La complexité de ce codage est identique à celle du problème de départ. Nous retrouvons le même nombre de variables CSP que de variables booléennes, avec des domaines de taille deux. Nous avons autant de contraintes que de clauses, et la taille des relations est égale à k.

Walsh [Walsh, 2000a] a montré qu'un filtrage par consistance d'arc sur $\mathcal{P}$ est plus fort que la propagation unitaire sur Σ, et que moyennant des heuristiques de branchement équivalentes, DP explore un arbre de recherche de même taille que nFC0 et plus grand que tout autre algorithme énumératif complet maintenant la consistance d'arc à chaque nœud de l'arbre (nFC1, ..., nFC5, MAC).

3.2 CSP → SAT

Il existe quatre codages permettant d'exprimer un CSP sous la forme d'une formule CNF. Deux d'entre eux (*le codage des supports* [Kasif, 1990] et *le codage logarithmique* [Walsh, 2000b, Gelder, 2007, Iwama et Miyazaki, 1994]) ne s'appliquent que sur des CSP binaires, et les deux autres (*le codage direct* [Kleer, 1989] et *le codage k-AC* [Bessière et al., 2003]) peuvent s'appliquer sur les CSP n-aires.

3.2.1 Le codage direct

C'est le plus utilisé et le plus ancien des codages. Il fut introduit par De Kleer [Kleer, 1989]. Dans ce codage, on associe une variable propositionnelle à chaque valeur du domaine de chaque variable du CSP. Par exemple, une variable CSP X avec pour domaine $D_X = \{v_1, v_2, \ldots, v_k\}$ définit k variables booléennes : $x_{v_1}, x_{v_2}, \ldots, x_{v_k}$. Interpréter x_{v_i} à *vrai* signifie que la variable CSP X est affectée à la valeur v_i. Ces variables propositionnelles apparaissent dans trois types de clauses :

Les clauses de type *au-moins-un* : Il y a une clause de type *au-moins-un*, par variable. Elles codent les domaines du CSP, exprimant le fait que chaque variable doit recevoir une valeur de son domaine. Considérons l'exemple d'une variable CSP X avec un domaine $D_X = \{v_1, v_2, v_3\}$. La clause $c_X = x_{v_1} \vee x_{v_2} \vee x_{v_3}$ est ajoutée pour coder le domaine D_X. Ces clauses seront notées clauses *a.m.u.* par la suite.

Les clauses de type *au-plus-un* : Il y a une clause de type *au-plus-un* pour chaque couple de valeurs du domaine de chaque variable. Ce sont des clauses binaires codant le fait que chaque variable du CSP ne peut recevoir plus d'une valeur de son domaine. On les appelle aussi *clause d'exclusion mutuelle*. Reprenons l'exemple précédent. Pour le domaine $D_X = \{v_1, v_2, v_3\}$, il faut ajouter les 3 clauses binaires suivantes : $\neg x_{v_1} \vee \neg x_{v_2}$, $\neg x_{v_2} \vee \neg x_{v_3}$ et $\neg x_{v_1} \vee \neg x_{v_3}$. Ces clauses seront notées clauses *a.p.u.* par la suite. Il est en général possible de s'affranchir de cet ensemble de clauses. En effet, si un modèle affectant plusieurs valeurs à une même variable CSP est trouvé, il est possible d'en choisir une seule aléatoirement et d'ignorer les autres.

Les clauses de conflits : Il y a une clause de conflit pour chaque *tuple interdit* de chaque contrainte du CSP. Elles expriment les contraintes en codant toutes les combinaisons interdites par celles-ci. Par exemple, considérons une contrainte entre trois variables CSP X, Y et Z et $[u \in D_X, v \in D_Y, w \in D_Z]$ un tuple interdit par la contrainte C_{XYZ} (c'est-à-dire que $[u, v, w] \notin R_{XYZ}$). La clause de conflit $c_{XYZ} = \neg x_u \vee \neg y_v \neg z_w$ est ajoutée pour interdire l'affectation simultanée de X à u,Y à v et Z à w.

Soit $\mathcal{P}$ un CSP et Σ_{Dir} la formule CNF résultant du codage direct de $\mathcal{P}$. Walsh a prouvé qu'effectuer un filtrage par consistance d'arc sur $\mathcal{P}$ est plus fort que d'effectuer la propagation unitaire sur Σ_{Dir}. Il a également démontré, comme cela l'avait été par Génisson et Jégou [Génisson et Jégou, 1996] que moyennant des heuristiques de branchement équivalentes, FC appliqué à $\mathcal{P}$ et DP appliqué à Σ_{Dir} étaient équivalents.

La complexité du codage direct d'un CSP est en $\mathcal{O}(md^a)$, en rappelant que m représente le nombre de contraintes du CSP, a la plus grande arité des contraintes du CSP et d la taille du plus grand domaine du CSP. Plus précisément, dans le codage direct d'un CSP en CNF, on retrouve n clauses *a.m.u.* de taille d. Chaque variable CSP ajoute $\frac{d(d-1)}{2}$ clauses *a.p.u.* de taille 2. Enfin chaque contrainte peut contenir au plus d^a tuples interdits de longueur a. Ce qui donne une complexité totale de $(nd + 2n\frac{d(d-1)}{2} + mad^a) \equiv \mathcal{O}(md^a)$.

3.2.2 Le codage des supports

Le codage des supports fut introduit par Kasif [Kasif, 1990] et spécialement conçu pour les CSP binaires. Dans ce codage, on retrouve les même variables propositionnelles que pour le codage direct. On y retrouve également les même clauses *a.m.u.* et *a.p.u.*. La seule différence réside dans les clauses de conflits qui sont remplacées par des clauses appelées clauses de supports.

Les clauses de supports : Une clause de support est ajoutée pour chaque couple (*valeur, liste de supports*) de chaque contrainte binaire. Elle code le fait que tant qu'une *valeur* reste dans le domaine d'une des variables impliquées dans la contrainte, alors au moins un de ces supports doit rester dans le domaine de la seconde variable impliquée. Par exemple, considérons X et Y deux variables CSP et le couple $(v \in D_X, \{s_1, s_2, \ldots, s_k\} \in D_Y)$ où v est une valeur du domaine de X et $\{s_1, s_2, \ldots, s_k\}$ sont les supports de $X = v$ dans le domaine de Y pour la contrainte C_{XY}. La clause $\neg x_v \vee y_{s_1} \vee y_{s_2} \vee \ldots \vee y_{s_k}$ est rajoutée pour exprimer le fait que tant que v n'est pas retiré (filtré) du domaine de X, au moins un de ses supports dans D_Y ne doit pas être supprimé. Cette clause est équivalente à $x_v \rightarrow (y_{s_1} \vee y_{s_2} \vee \ldots \vee y_{s_k})$. On voit donc que si tous les supports sont retirés (falsifiés), $\neg x_v$ est impliqué, c'est-à-dire que v est retirée du domaine de X lors d'un filtrage par consistance d'arc.

On voit donc l'intérêt de ce codage pour les CSP binaires par rapport au codage direct. Ce codage permet de récupérer la consistance d'arc dans la CNF. Soit $\mathcal{P}$ un CSP et Σ_{Sup} la formule CNF résultant du codage des supports de $\mathcal{P}$. Kasif a montré que la propagation unitaire appliquée à Σ_{Sup} est équivalente à un filtrage par consistance d'arc sur $\mathcal{P}$. Cela a permis à Gent [Gent, 2002] de montrer que moyennant des heuristiques de branchement équivalentes, DP appliqué à Σ_{Sup} est équivalent à MAC appliqué à $\mathcal{P}$.

La complexité du codage des supports d'un CSP est en $\mathcal{O}(nd^2 + md^2)$. Précisément, on retrouve les même clauses *a.m.u.* et *a.p.u.* (nd^2), et pour chaque clause, il y a $2d$ couples (*valeur, liste de supports*) possibles, chaque liste pouvant contenir au plus d valeurs ($2md^2$). La complexité totale est donc de $(nd^2 + 2md^2) \equiv \mathcal{O}((n + m)d^2)$.

3.2.3 Le codage k-AC

Ce codage a été proposé par Bessière, Hebrard et Walsh [Bessière et al., 2003] et se trouve être la généralisation du codage des supports aux CSP n-aires. L'idée sous-jacente est de coder des supports sur des ensembles de variables de taille quelconque pour des ensembles de variables de taille quelconque au lieu de support sur une unique variable pour une unique variable.

Dans ce codage, on retrouve encore les même variables propositionnelles et les même clauses *a.m.u.* et *a.p.u.*, et l'ensemble des clauses de supports est remplacé par un ensemble de clauses k-AC.

Les clauses k-AC : Pour chaque contrainte d'arité a impliquant un ensemble S de variables, une clause k-AC est ajoutée pour tout couple (I, L), où I est une instanciation des variables d'un sous-ensemble de S de taille k autorisée par R_S et L est la liste des supports de I sur les variables de $S - S_1$. La clause k-AC associée à une instanciation code le fait que tant que cette instanciation est possible, c'est-à-dire tant qu'une des variables n'a pas été affectée à une autre valeur, l'instanciation d'au moins un des supports de cette instanciation doit être possible sur les variables de $S - S_1$. Par exemple, admettons que la relation associée à une contrainte portant sur un ensemble S de variables autorise l'instanciation partielle $I = \{(X_1, v_1), (X_2, v_2), \ldots, (X_k, v_k)\}$, et que cette instanciation possède m supports sur les variables de $S - \{X_1, X_2, \ldots, X_k\}$. Alors la clause k-AC $\neg X_{1_{v_1}} \vee \neg X_{2_{v_2}} \vee \ldots \vee \neg X_{k_{v_k}} \vee s_1 \vee s_2 \vee \ldots \vee s_m$ est ajoutée. Étant donné que dans ce codage, une instanciation coïncide avec une conjonction, l'instanciation I correspond à la conjonction $(X_{1_{v_1}} \wedge X_{2_{v_2}} \wedge \ldots \wedge X_{k_{v_k}})$. La clause k-AC est donc équivalente à $I \rightarrow (s_1 \vee s_2 \vee \ldots \vee s_m)$. On voit donc que si tous les supports sont falsifiés, alors la clause k-AC est réduite à une clause de longueur k interdisant l'instanciation I.

Cependant, un support s_i pour une instanciation I des variables d'un sous-ensemble $S_1 \subset S$ tel que $|S| = a$ et $|S_1| = k$ n'est autre qu'une instanciation des variables de $S - S_1$ compatible avec I. Donc comme nous venons de le voir, s_i est équivalent à une conjonction de plusieurs littéraux, excepté lorsque $a - k = 1$ ou $a - k = 0$. Dans le cas où $a - k = 1$, s_i est une conjonction réduite à un seul littéral. Dans le cas où $a - k = 0$, s_i est réduit à la valeur de vérité *vrai* ou *faux*. Les clauses k-AC ne sont donc pas des clauses à proprement parler. Pour conserver la forme clausale, les auteurs ont introduit des variables additionnelles qu'ils appellent variables de supports, pour chaque support équivalent à une conjonction d'au moins deux littéraux. Par exemple, si le support s_i est l'instanciation $\{(X_1, v_1), (X_2, v_2), \ldots, (X_p, v_p)\}$, il faut rajouter la formule $s_i \leftrightarrow X_{1_{v_1}} \wedge X_{2_{v_2}} \wedge \ldots \wedge X_{p_{v_p}}$, qui peut être écrite sous forme clausale par : $(\neg s_i \vee X_{1_{v_1}}), (\neg s_i \vee X_{2_{v_2}}), \ldots, (\neg s_i \vee X_{p_{v_p}})$ et $s_i \vee \neg X_{1_{v_1}} \vee \neg X_{2_{v_2}} \vee \ldots \vee \neg X_{p_{v_p}}$. Le tableau 3.1 montre les codages k-AC possibles pour une contrainte d'arité 3 avec plusieurs valeurs de k.

Soient $\mathcal{P}$ un CSP et Σ_{k-AC} la formule CNF résultant du codage k-AC de $\mathcal{P}$. Malgré l'ajout des variables de support, les auteurs ont montré que la complexité de ce codage restait en $\mathcal{O}(md^a)$, quelque soit la valeur de k choisie, puisque les nd^2 clauses *a.m.u.* peuvent être négligées. De plus, ils ont montré qu'effectuer la propagation unitaire sur Σ_{k-AC} était équivalente à filtrer $\mathcal{P}$ par k-consistance d'arc relationnelle. Il en découle donc que, moyennant des heuristiques de branchement équivalentes, DP appliqué à Σ_{k-AC} est équivalent à un algorithme du style MAC qui maintient la k-consistance d'arc relationnelle à chaque étape.

De plus, dans le cas de CSP binaires, les auteurs ont également montré qu'en rajoutant la jointure de certaines contraintes avant d'appliquer le codage k-AC du CSP, l'application de la propagation unitaire sur la formule obtenue est équivalente à un filtrage par (i, j)-consistance sur

R_{XYZ}		
X	Y	Z
a	a	b
a	b	b
b	a	a
b	a	b

$\Longrightarrow_{\text{Codage}}$

Codage 0-AC	Codage 3-AC
$V \rightarrow (s_1 \vee s_2 \vee s_3 \vee s_4) \wedge$ $(X_a \wedge Y_a \wedge Z_b) \leftrightarrow s_1 \wedge$ $(X_a \wedge Y_b \wedge Z_a) \leftrightarrow s_2 \wedge$ $(X_b \wedge Y_a \wedge Z_a) \leftrightarrow s_3 \wedge$ $(X_b \wedge Y_a \wedge Z_b) \leftrightarrow s_4 \wedge$	$((X_a \wedge Y_a \wedge Z_a) \rightarrow F) \wedge$ $((X_a \wedge Y_b \wedge Z_a) \rightarrow F) \wedge$ $((X_b \wedge Y_b \wedge Z_a) \rightarrow F) \wedge$ $((X_b \wedge Y_b \wedge Z_b) \rightarrow F)$
Codage 2-AC	**Codage 1-AC**
$((X_a \wedge Y_a) \rightarrow Z_b) \wedge$ $((X_a \wedge Y_b) \rightarrow Z_b) \wedge$ $((X_b \wedge Y_a) \rightarrow (Z_b \vee Z_a)) \wedge$ $((X_b \wedge Y_b) \rightarrow F) \wedge$ $((X_a \wedge Z_a) \rightarrow F) \wedge$ $((X_a \wedge Z_b) \rightarrow (Y_a \vee Y_b)) \wedge$ $((X_b \wedge Z_a) \rightarrow Y_a) \wedge$ $((X_b \wedge Z_b) \rightarrow Y_a)$ $((Y_a \wedge Z_a) \rightarrow X_b)$ $((Y_a \wedge Z_b) \rightarrow (X_a \vee W_b)) \wedge$ $((Y_b \wedge Z_a) \rightarrow F) \wedge$ $((Y_b \wedge Z_b) \rightarrow X_a)$	$(X_a \rightarrow (s_1 \vee s_2)) \wedge$ $(X_b \rightarrow (s_3 \vee s_1)) \wedge$ $(Y_a \rightarrow (s_4 \vee s_5 \vee s_6)) \wedge$ $(Y_b \rightarrow s_4) \wedge$ $(Z_a \rightarrow s_7) \wedge$ $(Z_b \rightarrow (s_7 \vee s_8 \vee s_9)) \wedge$ $((Y_a \wedge Z_b) \leftrightarrow s_1) \wedge$ $((Y_b \wedge Z_b) \leftrightarrow s_2) \wedge$ $((Y_a \wedge Z_a) \leftrightarrow s_3) \wedge$ $((X_a \wedge Z_b) \leftrightarrow s_4) \wedge$ $((X_b \wedge Z_a) \leftrightarrow s_5) \wedge$ $((X_b \wedge Z_b) \leftrightarrow s_6) \wedge$ $((X_b \wedge Y_a) \leftrightarrow s_7) \wedge$ $((X_a \wedge Y_a) \leftrightarrow s_8) \wedge$ $((X_a \wedge Y_b) \leftrightarrow s_9) \wedge$

TAB. 3.1 – Exemple de codage k-AC d'une contrainte ternaire impliquant les variables X, Y et Z pour 4 valeurs de k. (V = Vrai et F = Faux).

le CSP de départ. La complexité de la formule ainsi obtenue étant cette fois en $\mathcal{O}(n^{i+j}d^{i+j})$ et la propagation unitaire pouvant être effectuée en temps linéaire, on peut effectuer un filtrage par (i,j)-consistance sur un CSP binaire en temps $\mathcal{O}(n^{i+j}d^{i+j})$ en passant par un codage k-AC. Cette complexité est optimale pour un tel filtrage.

3.2.4 Le codage logarithmique

Ce codage pour CSP binaire fut tout d'abord introduit par Iwama et Miyazaki [Iwama et Miyazaki, 1994] pour exprimer un problème *k-clique* sous forme de problème SAT. Il fut repris par Walsh dans [Walsh, 2000b] dans le cadre des CSP et par Gelder [Gelder, 2007] dans le cadre du problème de coloration de graphe.

Dans ce codage, seulement $n\lceil \log_2(d) \rceil$[2] variables propositionnelles sont utilisées. Les valeurs des domaines sont représentées sous la forme de bits, sachant que $n\lceil \log_2(d) \rceil$ représente le nombre de bits nécessaires pour coder en base 2 un domaine de taille d. Ainsi, un domaine D_{X_i} de taille d donnera lieu aux variables $X_{i_1}, X_{i_2}, \ldots, X_{i_{\lceil \log_2(d) \rceil}}$, où l'interprétation X_{i_j} = *vrai* signifie que la variable CSP X_i est affectée à une valeur pour laquelle le bit j de son codage en base 2 vaut 1.

Les clauses *a.m.u.* et *a.p.u.* ne sont pas nécessaires. Seulement deux types de clauses sont utilisées : *les clauses de conflits* et *les clauses de valeurs prohibées.*

[2] $\lceil \log_2(d) \rceil$ représente le premier nombre entier supérieur ou égal à $\log_2(d)$.

Les clauses de conflits : Pour une contrainte binaire $C_{X_iX_j}$, une clause de conflit est ajoutée pour tout couple de valeurs interdit par la relation $R_{X_iX_j}$. Ces clauses sont de longueur $2\lceil\log_2(d)\rceil$. Si $\{(X_i, v), (X_j, w)\}$ est interdit par $R_{X_iX_j}$, alors on rajoute la clause de conflit

$$\left(\bigvee_{b=0}^{\lceil\log_2(d)\rceil-1} v_b \oplus X_{i_b}\right) \vee \left(\bigvee_{b=0}^{\lceil\log_2(d)\rceil-1} w_b \oplus X_{j_b}\right)$$

où v_b et w_b sont les $b^{ème}$ bits des représentations binaires de v et w et $\oplus$ est le *ou exclusif* logique. Intuitivement, $v_b \oplus X_{i_b}$ correspond au littéral X_{i_b} si le bit v_b vaut 0 (v_b est *faux*), ou $\neg X_{i_b}$ dans le cas contraire.

Par exemple, considérons deux variables CSP X_1 et X_2 ayant pour domaine $D_{X_1} = D_{X_2} = \{0, 1, 2\}$, reliées par une contrainte imposant $X_1 \leq X_2$. Il faut $\lceil\log_2(3)\rceil = 2$ bits pour coder les domaines. La contrainte interdit 3 combinaisons :

- $\{(X_1, 1), (X_2, 0)\}$: ajout de la clause de conflit $\neg X_{1_0} \vee X_{1_1} \vee X_{2_0} \vee X_{2_1}$;
- $\{(X_1, 2), (X_2, 0)\}$: ajout de la clause de conflit $X_{1_0} \vee \neg X_{1_1} \vee X_{2_0} \vee X_{2_1}$;
- $\{(X_1, 2), (X_2, 1)\}$: ajout de la clause de conflit $X_{1_0} \vee \neg X_{1_1} \vee \neg X_{2_0} \vee X_{2_1}$.

Les clauses de valeurs prohibées : Dans le cas où la taille des domaines de CSP n'est pas une puissance de deux, c'est-à-dire que $\log_2(d)$ n'est pas un entier, il faut exclure certaines valeurs qui seraient hors domaine. Pour ce faire, on rajoute une clause de valeurs prohibées pour chaque valeur excédante de chaque domaine. Cette clause est de longueur $\lceil\log_2(d)\rceil$. Si une valeur v n'appartient pas à un domaine D_{X_i} et s'écrit $v = \Sigma_{b=0}^{\lceil\log_2(d)\rceil-1} 2^b v_b$, la clause de valeurs prohibées rajoutée est :

$$\neg\left(\bigwedge_{b=0}^{\lceil\log_2(d)\rceil-1} \neg(v_b \oplus X_{i_b})\right) \equiv \left(\bigvee_{b=0}^{\lceil\log_2(d)\rceil-1} v_b \oplus X_{i_b}\right).$$

Dans notre exemple, la valeur 3 peut être codée avec 2 bits, mais elle est excédante. Il faut donc rajouter deux clauses de valeurs prohibées : $\neg X_{1_0} \vee \neg X_{1_1}$ et $\neg X_{2_0} \vee \neg X_{2_1}$.

Soient $\mathcal{P}$ un CSP et $\Sigma_{\log}$ la formule CNF résultant du codage logarithmique de $\mathcal{P}$. La complexité de ce codage est en $\mathcal{O}(\lceil\log_2(d)\rceil md^2)$, bien que le nombre de variables propositionnelles utilisées soit uniquement de $n\lceil\log_2(d)\rceil$ au lieu de nd dans les autres codages. Ce gain est malheureusement compensé par la taille des clauses de conflits qui rendent les solveurs SAT inefficaces sur ce type de codage en comparaison avec les autres codages, bien que la taille totale des instances soit plus petite.

Walsh [Walsh, 2000b] a montré que l'application d'un filtrage par consistance d'arc sur $\mathcal{P}$ était plus fort que l'application de la propagation unitaire sur $\Sigma_{\log}$. Par conséquent, moyennant des heuristiques de branchement équivalentes, FC appliqué à $\mathcal{P}$ est plus efficace[3] que DP appliqué à $\Sigma_{\log}$.

Ce codage a été amélioré par Gavanelli [Gavanelli, 2007] pour donner lieu au codage *log-support*. Dans ce codage, qui utilise les même $n\lceil\log_2(d)\rceil$ variables propositionnelles, on retrouve la même idée que dans le codage des supports. Elle consiste à coder la liste des supports d'un couple (variable, valeur) d'une variable sur la seconde variable dans un contrainte binaire. Cependant, les supports de l'affectation d'une variable CSP à une valeur dans ce codage sont représentés

[3] Visite moins de nœuds pendant la recherche.

par des conjonctions de littéraux de taille $\lceil \log_2(d) \rceil$. Une clause de support qui pourrait s'écrire $X_{i_v} \rightarrow X_{j_{v_1}} \vee X_{j_{v_2}} \vee \ldots \vee X_{j_{v_k}}$ pour le codage des supports prendrait la forme :

$$\left(\bigvee_b \neg(v_b \oplus X_{i_b}) \right) \rightarrow \left(\bigvee_b \neg(v_{1_b} \oplus X_{j_b}) \right) \vee \left(\bigvee_b \neg(v_{2_b} \oplus X_{j_b}) \right) \vee \ldots \vee \left(\bigvee_b \neg(v_{k_b} \oplus X_{j_b}) \right).$$

Exprimer cette formule sous forme clausale générerait un nombre exponentiel de clauses. C'est pourquoi Gavanelli se restreint aux implications qui n'ont qu'un seul littéral en conclusion, et plus précisément celles ayant un littéral associé à un bit de poids maximal ou minimal uniquement.

Sur l'exemple précédent, on remarque que l'affectation $(X_1, 2)$ n'admet que le support $(X_2, 2)$ dans la contrainte. Il se trouve que le bit de poids fort est *vrai* pour 2. Cela permet d'écrire la clause de support $\neg X_{1_0} \wedge X_{1_1} \rightarrow X_{2_1}$. L'ajout de cette clause permet de supprimer deux clauses de conflits du codage logarithmique. Ainsi, le codage log-support du CSP devient :

- clauses de valeurs prohibées : $\neg X_{1_0} \vee \neg X_{1_1}$ et $\neg X_{2_0} \vee \neg X_{2_1}$
- clauses de supports : $X_{1_0} \vee \neg X_{1_1} \vee X_{2_1}$
- clauses de conflits : $\neg X_{1_0} \vee X_{1_1} \vee X_{2_0} \vee X_{2_1}$

Ce procédé permet de remplacer plusieurs clauses de conflits de tailles $2\lceil \log_2(d) \rceil$ par des clauses de taille $\lceil \log_2(d) \rceil + 1$. Il peut être étendu aux autres bits (pas uniquement les plus significatifs) et réduire encore la taille du codage. Ce codage est plus compétitif que le codage logarithmique simple et requiert en pratique moins de mémoire pour coder les instances. Mais tous les supports n'étant pas encodés, la propagation unitaire n'est toujours pas en mesure d'égaler le filtrage par consistance d'arc.

Conclusion

Le problème SAT, malgré sa nature très élémentaire, est un problème complexe à résoudre. Étant le problème NP-complet de référence, les résultats théoriques, mais également pratiques, obtenus quant à la résolution de ce problème permettent indubitablement une avancée dans la résolution des nombreux problèmes qui s'y ramènent naturellement.

Nous avons constaté qu'un grand nombre de classes polynomiales de SAT existaient. Cependant la plupart des résultats obtenus sur ces classes polynomiales s'appuient très souvent sur une restriction du langage. Cette perte d'expressivité est souvent trop contraignante pour permettre une exploitation pratique effective de ces classes polynomiales. Les ensembles backdoor et strong backdoor représentent a priori un moyen détourné de les exploiter en pratique. Reste qu'à l'heure actuelle, le calcul de ces ensembles est trop coûteux pour être utilisé de manière générale.

Dans le chapitre 4 de ce manuscrit, nous proposons une méthode de recherche locale qui, étant certaines classes polynomiales, calcule des ensembles strong backdoor de taille minimale. Cette méthode est capable de traiter des instances contenant plusieurs milliers de variables et de clauses. Nous proposons ensuite une méthode complète de résolution du problème SAT exploitant les ensembles strong backdoor calculés.

Parfois, lorsque la taille des instances est trop importante et qu'elle n'appartient à aucune classe polynomiale, il n'est d'autre choix que d'utiliser une méthode de recherche locale. Nous avons présenté brièvement le principe des méthodes de résolution du problème SAT, complètes comme incomplètes, qui ont été suggérées et implémentées par le passé, chacune ayant ses avantages et ses inconvénients.

Nous présentons dans le chapitre 5 notre contribution aux méthodes de recherche incomplètes pour SAT. Nous proposons une méthode de recherche locale qui est inspirée des méthodes complètes et qui a la particularité de n'explorer que des interprétations partielles consistantes.

Le formalisme des CSP, bien que plus expressif à la base, est fortement lié au formalisme SAT. Nous avons vu que les problèmes exprimés sous forme de CSP pouvaient être codés de plusieurs manières en instances SAT et vice-versa, et que certaines propriétés peuvent être conservées. En particulier, nous avons pu constater que pour certains codages il y a une correspondance entre la propagation unitaire dans une instance SAT et un un filtrage par consistance locale dans le CSP correspondant.

Concernant les filtrages par consistances partielles, nous avons constaté que la tendance actuelle consiste à chercher le meilleur compromis entre effort de calcul (complexité de l'algorithme de filtrage associé à la consistance locale choisie) et pouvoir filtrant (de l'algorithme de filtrage as-

socié à la consistance locale choisie). On imagine de plus en plus de solveurs intégrant un filtrage par consistance locale assez forte en guise de pré-traitement, puis un filtrage plus faible lors de la recherche, mais généralement au moins aussi fort que le filtrage par consistance d'arc. Cependant la question du meilleur compromis reste un problème ouvert qui semble à première vue dépendre du type d'instance à résoudre.

Dans le chapitre 6, nous proposons un formalisme propositionnel qui est une généralisation de la forme CNF dans lequel il est possible d'exprimer n'importe quel CSP défini en extension. Nous mettons en évidence quelques règles d'inférences, dont certaines se trouvent être équivalentes à des consistances locales des CSP. De plus, dans le formalisme CSP, une grande partie des méthodes de filtrage ou de résolution ont été conçues pour des CSP binaires, et lorsque nous voulons traiter des CSP n-aires, nous devons faire face à des problèmes de représentations et certaines méthodes deviennent inutilisables. Dans le formalisme que nous proposons, les règles d'inférence et de résolution ne sont pas dépendantes de l'arité des contraintes du CSP codé.

Deuxième partie

Ensembles strong backdoor et voisinage consistant pour la résolution du problème SAT

Chapitre 4

Calcul et exploitation des ensembles strong backdoor

Sommaire

4.1 Introduction

La résolution du problème SAT a connu un essor particulier ces dernières années, s'accompagnant d'un certains nombre de solveurs efficaces capables de traiter des instances de grande taille. Un des facteurs important justifiant l'efficacité des solveurs SAT concerne l'exploitation de la structure des problèmes. Sur plusieurs instances, particulièrement sur celles codant des problèmes réels, l'efficacité d'un solveur SAT est fortement liée à sa capacité à exploiter la structure cachée de ces instances.

Récemment, une forme de structure intéressante, appelée ensembles (strong) backdoor, a été proposée dans [Williams et al., 2003b]. Calculer un tel ensemble est un sujet de recherche très étudié, à cause de sa connexion avec la complexité du problème [Kilby et al., 2005]. Nous rappelons qu'un ensemble de variables forme un ensemble backdoor pour une formule donnée si il existe une interprétation de ces variables telle que la formule simplifiée peut être satisfaite en temps polynomial. Un tel ensemble est appelé ensemble strong backdoor si toute affectation de ces variables mène à une sous-formule de classe polynomiale.

L'intérêt de calculer un ensemble strong backdoor est très simple à appréhender lorsque l'on connaît le fonctionnement des algorithmes énumératifs complets qui résolvent le problème SAT. Pour une instance SAT contenant n variables, n'importe quel solveur complet de type DPLL a une complexité théorique en temps de $\mathcal{O}(2^n)$. Si on connaît un ensemble strong backdoor B pour cette instance, on peut réduire cette complexité à $\mathcal{O}(2^{|B|}p(n))$, où $p(n)$ est la complexité (polynomiale) de résolution d'instances de la classe polynomiale considérée. En effet, il suffit de n'énumérer que sur les variables de l'ensemble strong backdoor car le test de satisfaisabilité de toutes les formules simplifiées par n'importe quelle interprétation des variables de B peut être réalisé en temps polynomial.

Cependant, nous rappelons que calculer le plus petit ensemble strong backdoor est un problème NP-difficile [Nishimura et al., 2004]. En pratique, approximer (en temps polynomial) un ensemble strong backdoor de taille « raisonnable » est donc un challenge intéressant et important. Plusieurs travaux précédents ont abordé cette question. L'approche proposée dans [Grégoire et al., 2005] essaie tout d'abord d'identifier un ensemble de portes logiques (fonctions booléennes) à partir d'une CNF donnée. Puis, elle applique des heuristiques pour déterminer un ensemble coupe-cycle pour le graphe représentant la formule hybride. On obtient un ensemble strong backdoor composé à la fois de l'ensemble des variables indépendantes et de celles de l'ensemble coupe-cycle. Cependant, sur certaines instances où aucune porte logique n'est identifiée, l'ensemble strong backdoor coïncide avec l'ensemble des variables du problème. D'autres approches ont été proposées qui utilisent différentes techniques, telles que les algorithmes de recherche systématique adaptée [Williams et al., 2003a, Kilby et al., 2005]), mais la forte combinatoire de ces méthodes ne leur permet pas d'être utilisé de manière générale.

Dans ce chapitre, nous proposons une approche polynomiale capable de fournir un ensemble strong $\mathcal{F}$-backdoor, de taille raisonnable pour des instances de grandes tailles, et ce pour plusieurs classes polynomiales $\mathcal{F}$. À cet effet, l'approche que nous avons conçu commence par considérer une formule CNF donnée comme la conjonction de deux sous-formules différentes, où la première appartient à une classe polynomiale et la seconde est composée du reste des clauses. Nous montrons qu'un ensemble strong backdoor peut être obtenue à partir des variables de la seconde sous-formule. Cependant, un tel ensemble strong backdoor peut être très grand dans certains cas (la plupart des variables de la seconde sous-formule apparaissent dans la première). Pour pallier ce

phénomène, notre approche tente de réduire le nombre de variables communes à ces deux sous-formules.

Nous nous intéressons dans un premier temps à deux classes polynomiales particulières que sont les clauses de Horn et les clauses ordonnées, ainsi que leurs extensions respectives : les clauses Horn-renommables et clauses ordonné-renommables.

Pour réduire la taille de l'ensemble strong backdoor, nous décomposons notre approche en deux étapes :

1. trouver un renommage des variables qui maximise (resp. minimise) la taille de la partie (resp. non) Horn/ordonnée de la formule en terme de nombre de clauses.
2. calculer, à partir de la sous-formule non polynomiale réduite (obtenue en 1.) un ensemble strong backdoor dont la taille est la plus petite possible.

Malheureusement, ces deux problèmes s'avèrent NP-difficiles (trouver le meilleur renommage et calculer le plus petit ensemble strong backdoor). En fait, calculer la sous-formule Horn-renommable maximale pour une CNF donnée est équivalent au problème MAX2SAT [Hirsch, 2000, Papadimitriou, 1994]) *i.e.* pour un ensemble de clauses binaires donné, trouver un modèle qui satisfait le plus grand nombre de clauses. Le problème de décision du second problème est défini dans [Nishimura et al., 2004] par Nishimura *et al.* et il y est démontré qu'il est NP-complet. Tout d'abord, il appartient à la classe NP car l'auteur montre que l'on peut vérifier en temps polynomial qu'un ensemble B est un strong backdoor. Ensuite, la NP-complétude y est démontrée en utilisant une réduction polynomiale du problème Vertex-Cover au problème du calcul d'un ensemble strong backdoor. Par conséquent, trouver un ensemble strong backdoor de taille minimale est aussi NP-difficile.

Pour contourner la difficulté de ces deux problèmes, notre approche utilise une méthode de recherche locale pour approximer le renommage qui maximisera la sous-formule polynomiale (première étape). Puis nous décrivons une approche heuristique efficace pour calculer un ensemble strong backdoor (seconde étape). Cette méthode heuristique ne prend en compte, pour le calcul de l'ensemble strong backdoor, que les variables apparaissant dans les clauses de la partie non polynomiale de la formule renommée.

Nous montrons par la suite pourquoi ce procédé n'a aucune chance d'être exploité efficacement pour la classe polynomiale des formules bien imbriquées.

Ces travaux ont donné lieux aux publications suivantes : [Paris, 2006], [Paris et al., 2006b], [Paris et al., 2007b], [Paris et al., 2006c] et [Paris et al., 2007c].

4.2 Calcul d'ensembles strong Horn-backdoor

4.2.1 Rappel des définitions préliminaires et notations

Nous rappelons ici la définition des ensembles strong $\mathcal{F}$-backdoor et donnons les notations qui seront utilisées dans ce chapitre.

Définition 47 (Ensemble strong $\mathcal{F}$-backdoor)
Soient Σ une formule CNF, B un sous-ensemble de $\mathcal{V}(\Sigma)$ et $\mathcal{F}$ une classe polynomiale de SAT. On dit que B est un ensemble ***strong $\mathcal{F}$-backdoor*** *si et seulement si pour toute interprétation I_B des variables de B, $\Sigma_{I_B} \in \mathcal{F}$.*

Notations Nous rappelons que l'ensemble des littéraux d'une formule Σ est noté $\mathcal{L}(\Sigma)$ et que l'ensemble de ces variables est noté $\mathcal{V}(\Sigma)$. Nous notons $\mathcal{L}^-(\Sigma)$ l'ensemble des littéraux négatifs présents dans la formule Σ et $\mathcal{L}^+(\Sigma)$ l'ensemble des littéraux positifs.

4.2.2 Calcul d'ensembles strong Horn-backdoor

Dans cette section, nous décrivons comment calculer un ensemble strong Horn-backdoor à partir d'une formule CNF Σ.

Commençons par donner un exemple simple d'ensemble *Strong Horn-backdoor* sur une formule de petite taille.

Exemple 10 *Considérons l'ensemble de clauses $\{a \vee b \vee \neg c, a \vee c \vee \neg d, a \vee e \vee f \vee \neg g\}$ et l'ensemble B=$\{a, e\}$. Pour toute interprétation des variables de B, l'ensemble de clauses devient vide, ou se transforme en un ensemble de clauses de Horn. L'ensemble B est donc un ensemble strong Horn-backdoor pour cet ensemble de clauses.*

Soit Σ une formule CNF. Nous montrons qu'un ensemble strong Horn-backdoor est forcément composé de variables apparaissant positivement dans la partie non Horn de Σ.

Proposition 5 *Soit $\Sigma = \phi \wedge \psi$ une formule CNF telle que ϕ (resp. ψ) soit (resp. ne soit pas) une formule de Horn et $B \subset \mathcal{L}^+(\psi)$. B est un ensemble strong Horn-backdoor de Σ si et seulement si $\forall c \in \Sigma$, $|\mathcal{L}^+(c) \backslash B| \leq 1$.*

Preuve *Comme B est un ensemble strong Horn-backdoor, pour toute interprétation I de B, $I(\Sigma)$ est de Horn. Supposons que $\exists c \in \psi$ telle que $|\mathcal{L}^+(c) \backslash B| > 1$. Alors, il existe au moins deux littéraux positifs dans c qui n'appartiennent pas à B. On peut construire une interprétation I de B telle que $I(c)$ contienne au moins deux littéraux positifs non affectés. Il ne reste plus qu'à interpréter les autres littéraux de B apparaissant dans $\mathcal{L}^+(c)$ à $faux$ dans I (et les autres littéraux de B à une valeur quelconque), ainsi on en déduit qu'il existe une interprétation I telle que $I(\Sigma)$ contienne une clause qui n'est pas de Horn, ce qui contredit l'hypothèse.*

Pour la réciproque, étant donné que pour chaque clause $c \in \psi$, $|\mathcal{L}^+(c) \backslash B| \leq 1$, au plus un littéral positif non affecté de c n'appartient pas à B. Le reste des littéraux positifs de c est dans B. Par conséquent, pour toute interprétation I de B, c est soit satisfaite par I, soit elle contient au plus un littéral positif i.e. $I(c)$ est une clause de Horn. $I(\psi)$ est donc une formule de Horn. En conséquence, B est un ensemble strong Horn-backdoor.

Le problème qui consiste à trouver le plus petit ensemble strong Horn-backdoor revient donc à trouver un ensemble de variables apparaissant positivement telles qu'une fois retirées de la sous-formule non polynomiale, elle devient une formule de Horn.

Pour calculer un ensemble strong Horn-backdoor, nous ne considérons donc que la sous-formule non Horn de Σ. La méthode que nous proposons pour accomplir cette tâche est décrite comme suit :

Tout d'abords, nous considérons une formule Σ' telle que $\forall c' \in \Sigma'$, $\exists c \in \Sigma$, avec $c' \subseteq c$ et $\forall l \in c'$, l est un littéral positif *i.e.* Σ' est la formule monotone composée de la partie positive non Horn de Σ.

Nous transformons ensuite chaque clause de la formule Σ' en une formule de cardinalité où chaque formule de cardinalité obtenue à partir d'une clause c' de taille k est dotée de sémantique

« *au moins* $k-1$ *littéraux sont vrais parmi l'ensemble des littéraux de* c' ». Pour conserver la forme clausale, chacune de ces formules est remplacée par un ensemble de clauses binaires.

La formule ainsi obtenue est une formule monotone 2-SAT. Trouver un ensemble strong Horn-backdoor de taille minimale de Σ se ramène au problème de trouver le modèle de taille minimale de formules monotones 2-SAT, connu pour être NP-difficile. L'exemple 11 illustre cette méthode sur la formule de l'exemple 10.

Exemple 11 *Considérons à nouveau l'ensemble de clauses de l'exemple 10 :* $\Sigma = \{a \vee b \vee \neg c, a \vee c \vee \neg d, a \vee e \vee f \vee \neg g\}$.

La formule Σ' *telle qu'elle vient d'être définie vaut :* $\Sigma' = \{a \vee b, a \vee c\ , a \vee e \vee f\}$.

Lors du passage aux formules de cardinalité exprimé par des clauses binaires, seule la dernière clause a besoin d'être remplacée par 3 clauses binaires, ce qui nous donne la formule monotone 2-SAT suivante : $\{a \vee b, a \vee c, a \vee e, a \vee f, e \vee f\}$.

$\{a,e\}$ *représente une interprétation minimale qui satisfait toutes les clauses. Il représente un strong Horn-backdoor.*

Évidemment, une telle méthode souffre des même maux que les méthodes proposées par le passé par Nishimura *et al.* [Nishimura et al., 2004] ou Kilby *et al.* [Kilby et al., 2005] et n'est pas utilisable en pratique du fait de sa forte combinatoire.

Pour circonscrire cette explosion combinatoire, nous proposons un algorithme glouton pour calculer un sous-ensemble de variables (littéraux positifs) de taille raisonnable. Cet algorithme consiste à choisir la variable qui apparaît le plus dans la sous-formule et d'en retirer toutes ses occurrences positives jusqu'à ce que la sous-formule devienne de Horn.

L'algorithme 7 montre comment calculer un ensemble strong Horn-backdoor en utilisant cette méthode gloutonne.

Algorithme 7 Strong Horn-Backdoor

Fonction Strong backdoor
Entrée : Σ : une formule CNF
Sortie : B : Un ensemble strong Horn-backdoor
1: init $B = \emptyset$
2: $\psi = \{c | c \in \Sigma, |\mathcal{L}^+(c)| > 1\}$
3: **répéter**
4: choisir un littéral positif l de $\mathcal{L}^+(\psi)$
5: $\psi = \{c \backslash \{l\} | c \in \psi, l \in \mathcal{L}^+(c)\} \cup \{c | c \in \Sigma, l \notin \mathcal{L}^+(c)\}$
6: $B = B \cup \{v\}$ {v est la variable associée au littéral l}
7: **jusqu'à ce que** ψ soit de Horn
8: **retourner** B

4.2.3 Ensembles strong Horn-backdoor et hiérarchie de Gallo et Scutellà

Nous rappelons tout d'abords la définition de la hiérarchie de Gallo et Scutellà [Gallo et Scutellá, 1988] :

Définition 37 (La hiérarchie de Gallo-Scutellà)
- Γ_0 = *Horn-SAT ;*
- $\Sigma \in \Gamma_k$ *si et seulement si il existe un littéral p tel que* $\Sigma_p \in \Gamma_{k-1}$ *et* $\Sigma_{\sim p} \in \Gamma_k$.

Avec cette définition, une formule Σ appartenant à la classe Γ_k peut être caractérisée de la façon suivante : il existe un ensemble de k variables de Σ tel qu'il existe une interprétation I_k des ces variables telle que Σ_{I_k} est une formule de Horn. Cette définition ressemble à de la définition d'un ensemble Horn-backdoor, excepté qu'elle n'impose pas que Σ soit satisfaisable.

Elle se différencie également de la définition d'un ensemble strong Horn-backdoor par le fait que pour un ensemble strong Horn-backdoor, toute interprétation des variables mène à une formule de Horn. La définition des ensembles strong backdoor est donc plus forte en un sens, ce qui nous permet d'avancer que si notre méthode de calcul d'ensembles strong Horn-backdoor retourne un ensemble de taille k, alors nous pouvons affirmer que la formule Σ appartient dans le pire des cas dans la classe Γ_k.

Ceci nous permettrait d'utiliser l'algorithme proposé par Gallo et Scutellà de complexité $\mathcal{O}(|\Sigma|n^k)$ pour la résolution, ce qui serait mieux que la complexité des solveurs classiques en $\mathcal{O}(2^n)$. Cependant, comme il a été dit, en limitant l'énumération des solveurs sur les k variables de l'ensemble strong backdoor il est possible d'atteindre une complexité en $\mathcal{O}(2^k p(n))$ qui est encore meilleure.

4.2.4 Expérimentations

Problèmes aléatoires

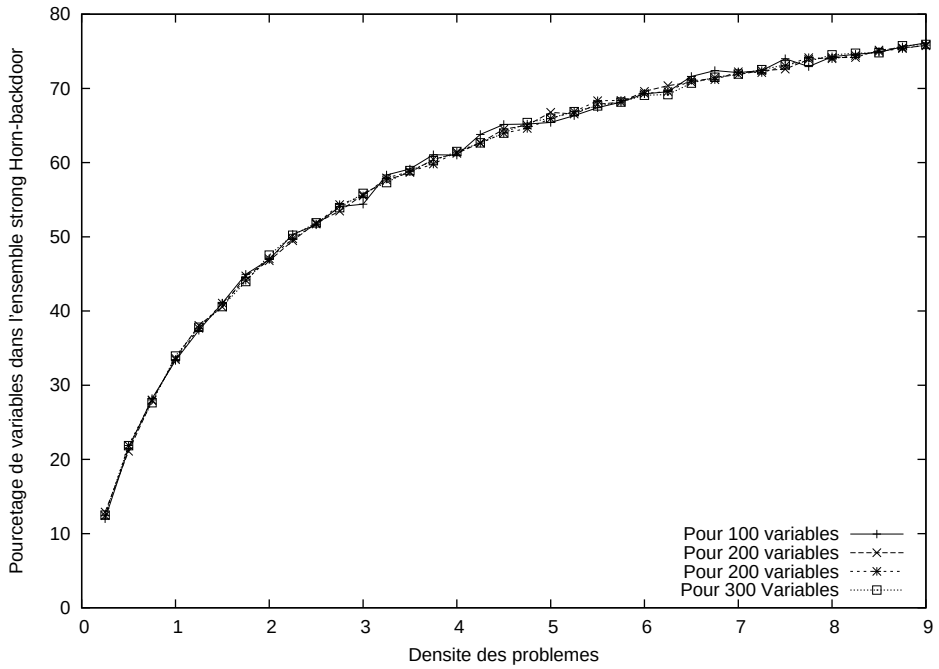


FIG. 4.1 – Tailles des ensembles strong Horn-backdoor pour des instances 3-SAT aléatoires

Les premières expérimentations ont été effectuées sur plusieurs classes d'instances 3-SAT générées aléatoirement. Chaque classe est définie par le nombre de variables présentes dans les problèmes. Ce nombre varie de 100 à 400 variables par pas de 100. Pour chaque classe, nous avons fait varier le rapport $\frac{nbCla}{nbVar}$ (la densité du problème) de 0.25 jusque 9. Pour chacun de ces

ratio, nous avons générés 50 instances et calculé la moyenne du nombre de variables apparaissant dans l'ensemble strong Horn-backdoor.

Les courbes de la figure 4.1 montrent le pourcentage de variables contenu dans les ensembles strong Horn-backdoor pour chacune des classes décrites en fonction de la variation de la densité du problème. Nous constatons que quelle que soit la taille de l'instance, la proportion de la taille des ensembles strong backdoor est la même pour une densité donnée.

Problèmes réels et industriels

Instance	$nbVar$	$nbCla$	$\|SB\|$	NB_NH
9symml_gr_rcs_w5	1295	24309	791	6085
9symml_gr_rcs_w6	1554	29119	1010	7645
alu2_gr_rcs_w7	3570	73478	2425	17715
alu2_gr_rcs_w8	4080	83902	2842	22301
apex7_gr_rcs_w5	1500	11695	831	3192
balancedhidden-k3-n918-pos5-neg5-01-as.sat03-910	918	3046	601	1518
balancedhidden-k3-n918-pos5-neg5-02-as.sat03-911	918	3049	594	1509
bf0432-007	1040	3668	540	920
bf1355-075	2180	6778	1137	1885
bf1355-638	2177	6768	1135	1873
bf2670-001	1393	3434	588	964
c499_gr_rcs_w5	1560	15777	934	4190
c880_gr_rcs_w5	3280	44291	1985	11334
c880_gr_rcs_w6	3936	53018	2509	13981
c880_gr_rcs_w7	4592	61745	3023	15598
ca032	558	1606	305	613
ca064	1132	3264	599	1263
ca128	2282	6586	1217	2571
clus-1200-4800-20-20-003-as.sat03-1087	1200	4800	738	2386
clus-1200-4919-10-10-001-as.sat03-1090	1200	4919	736	2432
clus-1200-4919-10-10-003-as.sat03-1092	1200	4919	737	2461
cnt07	1786	5856	1123	2570
cnt08	4089	13531	2590	5943
dp03s03	637	1468	218	478
dp03u02	478	1007	156	340
dp04s04	1271	3152	461	1090
dp04u03	1017	2411	363	826
dp05s05	1885	4818	710	1641
dp05u04	1571	3902	568	1336
dp06s06	2752	7202	1035	2436
dp06u05	2359	6053	890	2087
dp07s07	3649	9642	1416	3240
dp07u06	3193	8308	1232	2906

Instance	$nbVar$	$nbCla$	$\|SB\|$	NB_NH
dp08s08	5374	14508	2098	4908
dp09s09	6661	18160	2636	6136
dp10s10	8372	23004	3261	7769
dp11s11	10033	27728	3979	9342
dp12s12	12065	33520	4779	11359
example2_gr_rcs_w5	2220	23144	1256	6068
ezfact256_1	49153	324873	32927	211014
f2clk_50	34678	101319	13161	32392
fifo8_100	64762	176313	25046	54994
glassyb-v648-s1381725490	648	3024	431	1519
hanoi4	718	4934	400	1860
hanoi5	1931	14468	1063	5364
hanoi6_fast	7086	78492	3022	24187
hanoi6_on	7080	78066	3047	23822
hgen2-v700-s504028057	700	2450	461	1239
hidden-k3-s2-r4-n700-03-S1945356584-as.sat03-1047	700	2800	430	1395
ip38	49967	162142	20702	52611
ip50	66131	214786	27257	70293
k2fix_gr_2pinvar_w9	5028	307674	4607	294543
okgen-c1300-v650-s1167163901-1167163901	650	1300	312	666
okgen-c1400-v700-s211648331-211648331	700	1400	329	647
okgen-c1750-v500-s748188212-748188212	500	1750	295	904
okgen-c2100-v600-s1967026721-1967026721	600	2100	358	1112
okgen-c2100-v600-s536911537-536911537	600	2100	349	1019
rand_net40-25-1	2000	5921	996	2024
rand_net40-25-10	2000	5921	972	2002
rand_net40-25-5	2000	5921	1002	2020
rand_net40-30-1	2400	7121	1184	2398
rand_net40-30-10	2400	7121	1186	2372
rand_net40-30-5	2400	7121	1167	2379
rand_net40-40-10	3200	9521	1562	3221
rand_net40-40-5	3200	9521	1572	3241
rand_net40-60-10	4800	14321	2364	4832
rand_net50-25-1	2500	7401	1247	2462
rand_net50-25-10	2500	7401	1241	2541
rand_net50-25-5	2500	7401	1238	2506
rand_net50-30-1	3000	8901	1497	3019
rand_net50-30-5	3000	8901	1496	3071
rand_net50-40-5	4000	11901	1973	3981
rand_net50-60-1	6000	17901	2971	5984
rand_net50-60-5	6000	17901	2982	6045

Instance	$nbVar$	$nbCla$	$\|SB\|$	NB_NH
rand_net60-25-1	3000	8881	1482	2972
rand_net60-25-10	3000	8881	1480	3039
rand_net60-30-1	3600	10681	1830	3614
rand_net60-30-10	3600	10681	1762	3586
rand_net60-30-5	3600	10681	1798	3622
rand_net60-40-1	4800	14281	2408	4842
rand_net60-40-5	4800	14281	2372	4788
rand_net60-60-1	7200	21481	3530	7185
rand_net70-25-1	3500	10361	1762	3565
rand_net70-25-5	3500	10361	1753	3510
rand_net70-30-1	4200	12461	2095	4189
rand_net70-30-5	4200	12461	2073	4198
rand_net70-40-1	5600	16661	2767	5651
rand_net70-40-5	5600	16661	2767	5535
rand_net70-60-1	8400	25061	4158	8293
rand_net70-60-10	8400	25061	4145	8415
sha1	61377	255417	31494	121599
too_large_gr_rcs_w5	2595	36129	1635	8097
too_large_gr_rcs_w6	3114	43251	2032	11500
too_large_gr_rcs_w7	3633	50373	2452	12649
too_large_gr_rcs_w8	4152	57495	2877	15024
too_large_gr_rcs_w9	4671	64617	3310	16541
unif-c1000-v500-s1356655268	500	1000	248	522
unif-c1300-v650-s425854131	650	1300	307	638
unif-c1400-v700-s1822569467	700	1400	332	692
unif-c2450-v700-s1373176568	700	2450	429	1261
unif-c2450-v700-s1465124739	700	2450	410	1227
unif-c2600-v650-s2035184328	650	2600	394	1327
unif-r3-v700-c2100-02-S1776031682-as.sat03-1106	700	2100	377	1030
unif-r4-v700-c2800-03-S869054807-as.sat03-1152	700	2800	433	1417
vda_gr_rcs_w7	5054	102047	3507	25078
vda_gr_rcs_w8	5776	116522	4069	29453
vda_gr_rcs_w9	6498	130997	4695	33705
w10_45	16931	51803	6908	16633
w10_60	26611	83538	10843	26909
w10_70	32745	103556	13305	33736
w10_75	36291	115273	14850	37421

TAB. 4.1 – Strong backdoor sur les instances industrielles

Nous avons ensuite calculé des ensembles strong Horn-backdoor pour un certain nombre d'instances réelles et industrielles issues des précédentes compétitions SAT. Le tableau 4.1 récapitule les résultats obtenus. Pour chaque instance, les colonnes $nbVar$ et $nbCla$ donnent le nombre de variables et de clauses de l'instance, la colonne $|SB|$ donne la taille de l'ensemble strong Horn-backdoor et la colonne *NB_NH* le nombre de clauses non Horn de l'instance.

4.3 Calcul d'ensembles strong *Horn-renommable*-backdoor

Il arrive très fréquemment que l'ensemble des variables apparaissant dans la partie non Horn d'une instance SAT coïncide avec l'ensemble des variables de l'instance. Ceci a pour conséquence que la taille des ensembles strong Horn-backdoor calculés à partir de cet ensemble, est importante.

L'idée que nous avons mis en œuvre pour réduire la taille de ces ensembles consiste à effectuer un pré-traitement sur la formule de départ avant de lancer le calcul effectif de l'ensemble strong Horn-backdoor. Ce pré-traitement consiste à tenter de renommer la formule de départ pour essayer de se rapprocher au maximum d'une formule de Horn, puis de lancer le calcul de l'ensemble strong Horn-backdoor sur la formule renommée. L'ensemble strong Horn-backdoor de la formule renommée constitue un ensemble strong *Horn-renommable*-backdoor de la formule de départ.

Cependant, comme il a été dit, calculer le meilleur Horn-renommage est un problème NP-difficile, nous allons donc calculer un Horn renommage maximal de manière approchée.

4.3.1 Approximation du meilleur Horn-renommage

Notre méthode pour approximer le meilleur Horn-renommage est basée sur l'exploitation de l'algorithme bien connu *WalkSat* (voir algorithme 6 section 1.5.2). Cet algorithme permet de trouver une interprétation satisfaisant un maximum de clauses pour une CNF donnée. Il commence par générer aléatoirement une interprétation complète, puis à chaque étape une variable est choisie et sa valeur de vérité est inversée (la variable est « flippée »). Différentes stratégies ont été proposées pour choisir cette variable (*e.g. Random Walk Strategy* [Selman et Kautz, 1993]). Plusieurs améliorations ont été proposées. Toutes ces stratégies essaient de choisir la « meilleure » variable, c'est-à-dire celle dont le fait de changer sa valeur de vérité fera décroître au maximum le nombre de clauses falsifiées du problème. Cette étape est répétée jusqu'à ce que (i) un nombre maximum (fixé à l'avance) de flips (*MAX_FLIPS*) est atteint ou (ii) un modèle est trouvé *i.e.* la formule est satisfaisable. Dans le premier cas, le processus est répété avec une autre interprétation générée aléatoirement jusqu'à ce qu'un nombre maximum de tentatives (fixé à l'avance) est atteint (*MAX_TRIES*). Si aucun modèle n'est trouvé, l'algorithme répond qu'il est incapable de statuer sur la satisfaisabilité de l'instance et fournit la meilleur interprétation qu'il ait trouvé.

Nous allons détailler quelque peu les spécificités de l'algorithme Walksat afin de faire le lien entre le problème SAT qu'il traite et le problème qui nous intéresse, à savoir le calcul du meilleur Horn renommage.

Fonction objectif de Walksat : min-conflict

La fonction objectif de WalkSat est basée sur deux compteurs qu'il maintient à jour en permanence : le compteur de *Break* qui comptabilise pour chaque variable x le nombre de clauses de Σ qui sont satisfaites par une interprétation I et qui deviennent falsifiées si on change la valeur de vérité de x dans I, et le compteur de *Make* qui comptabilise pour chaque variable x le nombre de clauses de Σ qui sont falsifiées par une interprétation I et qui deviennent satisfaites si on change la valeur de vérité de x dans I. Leur définition formelle est la suivante :

Définition 63 (BreakCount et MakeCount)
Soient Σ une formule CNF, $x \in \mathcal{V}(\Sigma)$, I une interprétation et I' l'interprétation obtenue à partir de I en inversant la valeur de vérité de x. On définit $breakCount(x, I) = |\{c \in \Sigma | I[c] =$

$vrai, I'[c] = faux\}|$ *et* $makeCount(x, I) = |\{c \in \Sigma | I[c] = faux, I'[c] = vrai\}|$*. On définit* $score(x, I) = makeCount(x, I) - breakCount(x, I)$ *comme le score de* x *pour* I

À chaque étape, Walksat choisit la variable à flipper dans une clause falsifiée. Il s'agit de celle qui possède le meilleur $score$ ou n'importe laquelle choisie au hasard en fonction d'une certaine probabilité. Après que la variable ait été flippée, les compteurs $MakeCount$ et $BreakCount$ sont mis à jour.

Considérer un renommage comme une interprétation

La définition d'un renommage d'un ensemble de variables comme elle a été énoncée[1], permet de l'identifier à une interprétation de ce même ensemble de variables. On peut donc voir un renommage ainsi :

Définition 64 (Renommage bis)
À la manière des interprétations, on peut définir un renommage ρ *d'un ensemble de variables* V *comme une application de* V *dans* $\{vrai, faux\}$ *ou bien de l'ensemble les littéraux issus de* V *dans ce même ensemble comme dans la définition initiale. Soit* l *le littéral positif issu d'une variable* $v \in V$*, alors* $\rho(l) = \neg l$ *si* $\rho[v] = vrai$ *et* $\rho(l) = l$ *si* $\rho[v] = faux$*. Par convention, les variables qui sont renommées (c'est-à-dire dont la polarité est inversée) par un renommage sont les variables* v *qui ont la valeur* $vrai$ *dans* ρ *(*$\rho[v] = vrai$*).*

Par extension, nous pouvons redéfinir de même les formules renommées :

Définition 65 (Formule renommée)
Soient Σ *une formule CNF,* ρ *un renommage de* $\mathcal{V}(\Sigma)$*. On définit la formule renommée* $\rho_\Sigma(\Sigma)$ *comme la formule obtenue en substituant, pour toutes les variables* x *telles que* $\rho[x] = vrai$*, toutes les occurrences de* x *(resp.* $\neg x$*) par* $\neg x$ *(resp.* x*).* x *est alors renommée dans* Σ*. Quand* $\rho_\Sigma(\Sigma)$ *est une formule de Horn,* ρ *est appelé un Horn-renommage de* Σ*.*

Notre méthode s'appuie sur le concept de littéraux *renommé-positifs* et *renommé-négatifs* suivant :

Définition 66 (Littéraux renommé-positifs et renommé-négatifs)
On dit qu'un littéral l *est positif dans* c *pour* ρ *si* $l \in c$ *et* $\neg l \in \rho$*, ou bien si* $\neg l \in c$ *et* $l \in \rho$*. On note ceci* $estPos(l, c, \rho)$*. Ce même littéral est renommé-négatif dans* c *pour* ρ *si* $l \in c$ *et* $l \in \rho$*, ou bien* $\neg l \in c$ *et* $\neg l \in \rho$*. On note ceci* $estNeg(l, c, \rho)$*.*

[1] Rappel :

Définition 34 (Renommages)
On appelle **renommage d'un ensemble de littéraux** $\mathbb{S}_L$*, une application* ρ *telle que :*

$$\rho : \quad \mathbb{S}_L \longrightarrow \mathbb{S}_L ; \qquad l \longmapsto \rho(l) = \begin{cases} \neg l \\ \text{ou} \\ l \end{cases} \qquad \text{et} \qquad \forall\, l \in \mathbb{S}_L : \rho(\neg l) = \neg\rho(l).$$

Par extension, nous définissons un **renommage** ρ_c **d'une clause** c *comme l'application de* ρ *à tous les littéraux de cette clause :* $\rho_c(c) = \bigvee_{l \in c} \rho(l)$*.*
De même, un **renommage** ρ_Σ **d'un ensemble de clauses** Σ *se définit comme :* $\rho_\Sigma(\Sigma) = \bigwedge_{c \in \Sigma} \rho_c(c)$*.*

On note le nombre de littéraux renommé-positifs (resp. renommé-négatif) dans c pour ρ par $nbPos(c, \rho)$ (resp. $nbNeg(c, \rho)$).

On note $nbPosTot(\Sigma, \rho)$ le nombre total de littéraux renommé-positifs présents dans la partie non Horn de Σ pour le renommage ρ.

Ce qui permet de définir la notion de *Horn-renommabilité* :

Définition 67 (Clause Horn-renommée)
Soient Σ une formule CNF et ρ un renommage. Une clause $c \in \Sigma$ est dite Horn-renommée par ρ (notée $h\_ren(c, \rho)$) si $nbPos(c, \rho) \leq 1$ i.e. c contient au plus un littéral positif pour ρ ; sinon elle est dite Horn-falsifiée par ρ (notée $h\_fal(c, \rho)$).

On note $nbHorn(\Sigma, \rho)$ le nombre de clauses de Σ Horn-renommées par ρ.

L'exemple 12 illustre ces définitions.

Exemple 12 (Illustration des notations) *Considérons la formule CNF*

$$\Sigma = \{c_1 = (\neg a \vee b \vee c),\ c_2 = (a \vee b),\ c_3 = (a \vee c),\ c_4 = (\neg b \vee \neg c)\}$$

et le renommage ρ telle que :

$$\rho[a] = vrai,\ \rho[b] = vrai \text{ et } \rho[c] = \text{vrai}$$

qui sera noté par la suite $\rho = \{a, b, c\}$.

La formule renommée associée à ce renommage est :

$$\rho_\Sigma(\Sigma) = \{\rho_\Sigma(c_1) = (a \vee \neg b \vee \neg c),\ \rho_\Sigma(c_2) = (\neg a \vee \neg b),\ \rho_\Sigma(c_3) = (\neg a \vee \neg c),\ \rho_\Sigma(c_4) = (b \vee c)\}$$

Les différents compteurs valent :
$nbPos(c_1, \rho) = 1,\ nbPos(c_2, \rho) = 0,\ nbPos(c_3, \rho) = 0,\ nbPos(c_4, \rho) = 2$
$nbHorn(\Sigma, \rho) = 3$

L'adaptation de l'algorithme WalkSat au calcul du meilleur Horn-renommage se fait en substituant une interprétation pour le problème SAT par un renommage et en substituant la notion de clause satisfaite par clause Horn-renommée. Nous obtenons ainsi l'algorithme WalkHorn décrit par l'algorithme 8.

Il est à noter que les conditions d'arrêt de notre algorithme sont légèrement différentes de celles de l'algorithme original WalkSat. *WalkHorn* termine dans trois cas différents :

1. le nombre maximum de tentatives (*MAX_TRIES*) est atteint. Dans ce cas, le meilleur renommage trouvé est retourné (ligne 26) ou,
2. un Horn-renommage est trouvé (ligne 6). La formule Σ est Horn-renommable ou,
3. un renommage ρ tel que pour toute clause c de Σ, $nbNeg(c, \rho) > 0$, ou tel que pour toute clause c de Σ, $nbPos(c, \rho) > 0$. En conséquence, toutes les clauses de $\rho_\Sigma(\Sigma)$ contiennent au moins un littéral renommé-positif, ou toutes les clauses de $\rho_\Sigma(\Sigma)$ contiennent au moins un littéral renommé-négatif (ligne 9).

Algorithme 8 Algorithme WalkHorn

Fonction WalkHorn
Entrée : Une formule CNF Σ et deux entiers *MAX_FLIPS* et *MAX_TRIES*
Sortie : Un *renommage* de Σ

Initialiser ρ_{max} avec toutes les variables de Σ à vrai
pour $i = 1$ à *MAX_TRIES* **faire**
  $\rho =$ renommage généré aléatoirement
  pour $j = 1$ à *MAX_FLIPS* **faire**
    si $nbHorn(\Sigma, \rho) = nbCla(\Sigma)$ **alors**
      retourner ρ {Σ *est Horn renommable*}
    fin si
    si ($\forall c \in \Sigma, nbPos(c, \rho) > 0$) **ou** ($\forall c \in \Sigma, nbNeg(c, \rho) > 0$) **alors**
      retourner ρ {Σ *est trivialement satisfaisable*}
    fin si
    {*Horn Random Walk Strategy*}
    Sélectionner aléatoirement une clause non Horn c
    Avec une probabilité p **faire**{p est un paramètre fixé arbitrairement}
    Sélectionner aléatoirement un littéral l de c
    $\rho = \rho - \{l\} \cup \{\neg l\}$
    fait
    Avec une probabilité $1 - p$ **faire**
    Soit $l \in c$ tel que $\forall l' \in c$ avec $l \neq l'$, $score(l, \rho) > score(l', \rho)$
    $\rho = \rho - \{l\} \cup \{\neg l\}$
    fait
    si $nbHorn(\Sigma, \rho) > nbHorn(\Sigma, \rho_{max})$ **alors**
      $\rho_{max} = \rho$
    fin si
  fin pour
fin pour
retourner ρ_{max}

Dans le troisième cas, la formule n'est pas forcément Horn-renommable, mais un modèle trivial existe pour Σ. Si toute les clauses de $\rho_\Sigma(\Sigma)$ contiennent au moins un littéral positif, alors $\rho_\Sigma(\Sigma)$ est satisfaisable. Étant donné que Σ et $\rho_\Sigma(\Sigma)$ sont équivalents pour le problème SAT, Σ est aussi satisfaisable. Dans ce cas, l'ensemble *strong backdoor* de Σ est vide. L'exemple 13 illustre un déroulement possible de cet algorithme sur l'exemple précédent.

Exemple 13 *Considérons à nouveau la formule CNF*

$$\Sigma = \{c_1 = (\neg a \vee b \vee c),\ c_2 = (a \vee b),\ c_3 = (a \vee c),\ c_4 = (\neg b \vee \neg c)\}$$

Considérons les renommages successifs suivants :

1. $\rho = \{a, b, c\}$:

$$\rho_\Sigma(\Sigma) = \{\rho_\Sigma(c_1) = (a \vee \neg b \vee \neg c),\ \rho_\Sigma(c_2) = (\neg a \vee \neg b),\ \rho_\Sigma(c_3) = (\neg a \vee \neg c),\ \rho_\Sigma(c_4) = (b \vee c)\}$$

nous avons $nbPos(c_1, \rho) = 1,\ nbPos(c_2, \rho) = 0,\ nbPos(c_3, \rho) = 0,\ nbPos(c_4, \rho) = 2$*, et* $nbHorn(\Sigma, \rho) = 3$ *(les clauses* c_1*,* c_2 *et* c_3 *sont Horn-renommées par* ρ*).*

2. *Si on inverse la valeur de la variable* a*, on obtient :* $\rho' = \{\neg a, b, c\}$ *:*

$$\rho'_\Sigma(\Sigma) = \{\rho'_\Sigma(c_1) = (\neg a \vee \neg b \vee \neg c),\ \rho'_\Sigma(c_2) = (a \vee \neg b),\ \rho'_\Sigma(c_3) = (a \vee \neg c),\ \rho'_\Sigma(c_4) = (b \vee c)\}$$

nous avons $nbPos(c_1, \rho') = 0,\ nbPos(c_2, \rho') = 1,\ nbPos(c_3, \rho') = 1,\ nbPos(c_4, \rho') = 2$*, et les clauses* c_1*,* c_2 *et* c_3 *sont toujours Horn-renommées par* ρ' *(*$nbHorn(\Sigma, \rho') = 3$*).*

3. *Si on inverse la valeur de la variable* b *on obtient :* $\rho'' = \{\neg a, \neg b, c\}$ *:*

$$\rho''_\Sigma(\Sigma) = \{\rho''_\Sigma(c_1) = (\neg a \vee b \vee \neg c),\ \rho''_\Sigma(c_2) = (a \vee b),\ \rho''_\Sigma(c_3) = (a \vee \neg c),\ \rho''_\Sigma(c_4) = (\neg b \vee c)\}$$

nous avons $nbPos(c_1, \rho'') = 1,\ nbPos(c_2, \rho'') = 2,\ nbPos(c_3, \rho'') = 1,\ nbPos(c_4, \rho'') = 1$*, cette fois-ci, les clauses* c_1*,* c_3 *et* c_4 *sont Horn-renommées par* ρ'' *(*$nbHorn(\Sigma, \rho') = 3$*). De plus, chaque clause de* $\rho''_\Sigma(\Sigma)$ *contient au moins un littéral renommé-positif, donc* $\rho''_\Sigma(\Sigma)$ *est trivialement satisfaisable, il suffit de mettre tous les littéraux à* $vrai$*. En appliquant le renommage à ce modèle, on obtient l'interprétation* $\{a, b, \neg c\}$ *qui est modèle de* Σ*. Dans ce cas, nous considérons que l'ensemble strong backdoor est vide.*

Il ne reste plus qu'à définir le critère de sélection de la variable à flipper, la fonction objectif ($score$).

4.3.2 Le renommage Horn_min_clauses : fonction objectif min-conflict

Les définitions précédentes nous permettent de définir des compteurs de *Horn-Break* et de *Horn-Make* pour le problème de Horn-renommage analogues à ceux du problème de satisfaisabilité :

Définition 68 (Horn-BreakCount et Horn-MakeCount)
Soient Σ *une formule CNF,* $x \in \mathcal{V}(\Sigma)$*,* ρ *un renommage et* ρ_x *le renommage obtenu à partir de* ρ *en inversant la valeur de vérité de* x*. On définit* $h\_breakCount(x, \rho) = |\{c | h\_ren(c, \rho), h\_fal(c, \rho_x)|$ *et* $h\_makeCount(x, \rho) = |\{c | h\_fal(c, \rho), h\_ren(c, \rho_x)\}|$*.*

On définit $h\_score(x, \rho) = h\_makeCount(x, \rho) - h\_breakCount(x, \rho)$*.*

Exemple 14 (Horn-BreakCount et Horn-MakeCount) *Reprenons la formule CNF de l'exemple 12 :*

$$\Sigma = \{c_1 = (\neg a \vee b \vee c),\ c_2 = (a \vee b),\ c_3 = (a \vee c),\ c_4 = (\neg b \vee \neg c)\}$$

et le renommage $\rho = \{\neg a, \neg b, \neg c\}$ *(aucune variable renommée).*

Les différents compteurs valent :
$h\_BreakCount(a, \rho) = 0$*,* $h\_MakeCount(a, \rho) = 2$ *et* $h\_score(a, \rho) = 2$
$h\_BreakCount(b, \rho) = 0$*,* $h\_MakeCount(b, \rho) = 2$ *et* $h\_score(b, \rho) = 2$
$h\_BreakCount(c, \rho) = 0$*,* $h\_MakeCount(c, \rho) = 2$ *et* $h\_score(c, \rho) = 2$

Ainsi, en considérant la fonction h_score dans l'algorithme WalkHorn (algorithme 8) au lieu de $score$, nous obtenons un algorithme calculant le meilleur Horn renommage en terme de nombre

de clauses qui sont Horn-renommées. Ce renommage sera appelé renommage *Horn_Min_Clause* (HMC).

Le fait d'utiliser les même concepts que pour le problème SAT dans la version Horn de l'algorithme WalkSat nous permet également d'obtenir de manière simple des variantes de techniques de recherche (*e.g.* tabu, novelty, rnovelty [McAllester et al., 1997], rnovelty+ [Hoos, 1999] ...) pour le calcul de la sous-formule de Horn maximale.

4.3.3 Le renommage Horn_min_littéraux : fonction objectif min-size

Nous présentons maintenant une autre fonction objectif pour le calcul du meilleur Horn renommage. Contrairement à la fonction objectif min-conflict, cette nouvelle fonction objectif prend en compte la taille de la sous-formule non Horn, et plus précisément le nombre de littéraux positifs apparaissant dans la partie non Horn de la formule. Cette fonction objectif va tenter de trouver un renommage qui minimise cette taille. En effet, il semble raisonnable de penser que si nous arrivons à diminuer le nombre total de littéraux présents dans la partie non Horn (quitte à avoir plus de clauses non Horn), nous devrions aussi diminuer le nombre de variables présentes dans l'ensemble strong Horn-backdoor qui sera calculé à partir de cette partie non Horn.

La seule chose que nous ayons à faire pour mettre la nouvelle fonction objectif en œuvre, est de redéfinir les compteurs de *Horn-Break* et de *Horn-Make*, permettant à la fonction h_score de l'algorithme 8 d'être en adéquation avec cette fonction objectif.

Définition 69 (Horn-Min-BreakCount et Horn-Min-MakeCount)
Soient Σ une formule CNF, $x \in \mathcal{V}(\Sigma)$, ρ un renommage et ρ_x le renommage obtenu à partir de ρ en inversant la valeur de vérité de x.
Soient $h\_Break_1(x,\rho) = \{c \in \Sigma | nbPos(c,\rho) = 1$ *et* $estNeg(x,c,\rho)\}$,
$h\_Break_2(x,\rho) = \{c \in \Sigma | nbPos(c,\rho) > 1$ *et* $estNeg(x,c,\rho)\}$,
$h\_Make_1(x,\rho) = \{c \in \Sigma | nbPos(c,\rho) = 2$ *et* $estPos(x,c,\rho)\}$
et $h\_Make_2(x,\rho) = \{c \in \Sigma | nbPos(c,\rho) > 2$ *et* $estPos(x,c,\rho)\}$.
Les compteurs sont définis comme suit :
$hM\_breakCount(x,\rho) = 2 \times |h\_Break_1(x,\rho)| + |h\_Break_2(x,\rho)|$ *et*
$hM\_makeCount(x,\rho) = 2 \times |h\_Make_1(x,\rho)| + |h\_Make_2(x,\rho)|$.
Enfin on définit : $hM\_score(x,\rho) = hM\_makeCount(x,\rho) - hM\_breakCount(x,\rho)$

L'ensemble h_Break_1 correspond à l'ensemble des clauses qui sont Horn-renommées par ρ et qui ne le sont pas par ρ_x. Cela signifie que chacune de ces clauses contient *deux* littéraux renommé-positifs pour ρ_x, et qu'il faut rajouter *deux* littéraux pour chaque clause de $Break_1$ à $nbPosTot(\Sigma,\rho)$ si l'on passe de ρ à ρ_x (*i.e.* $nbPosTot(\Sigma,\rho_x) = 2 + nbPosTot(\Sigma,\rho)$ pour chacune de ces clauses). L'ensemble h_Break_2 correspond à l'ensemble des clauses qui ne sont pas Horn-renommées par ρ et dans lesquelles x est renommé-négatif pour ρ. Ces clauses ne sont pas Horn-renommées par ρ_x non plus et contiennent toutes un littéral renommé-positif de plus par rapport à ρ_x (x en l'occurrence). Il faut donc rajouter *un* littéral à $nbPosTot(\Sigma,\rho)$ pour chacune de ces clauses dans l'hypothèse où l'on passe de ρ à ρ_x.

D'un autre côté, l'ensemble h_Make_1 correspond à l'ensemble des clauses qui ne sont pas Horn-renommées par ρ et qui le sont pour ρ_x. Ce qui signifie que chacune de ces clauses contient *deux* littéraux renommé-positifs pour ρ qu'il faut soustraire à $nbPosTot(\Sigma,\rho)$ pour chacune de ces clauses lors du passage de ρ à ρ_x. Et enfin, l'ensemble h_Make_2 correspond à l'ensemble des

clauses qui ne sont Horn-renommées ni par ρ, ni par ρ_x et dans lesquelles x est renommé-positif pour ρ. Il faut donc retirer *un* littéral à $nbPosTot(\Sigma, \rho)$ pour chacune des clauses de cet ensemble.

Exemple 15 (Horn-BreakCount et Horn-MakeCount) *Reprenons la formule CNF :*

$$\Sigma = \{c_1 = (\neg a \vee b \vee c),\ c_2 = (a \vee b),\ c_3 = (a \vee c),\ c_4 = (\neg b \vee \neg c)\}$$

et le renommage $\rho = \{\neg a, \neg b, \neg c\}$ (aucune variable renommée).

Les différents compteurs valent :

- $h\_Break_1(a, \rho) = 0$, $h\_Break_2(a, \rho) = 1$, $hM\_BreakCount(a, \rho) = 1$
  $h\_Make_1(a, \rho) = 2$, $h\_Make_2(a, \rho) = 0$, $hM\_MakeCount(a, \rho) = 4$
  $hM\_score(a, \rho) = 3$
- $h\_Break_1(b, \rho) = 0$, $h\_Break_2(b, \rho) = 0$, $hM\_BreakCount(b, \rho) = 0$
  $h\_Make_1(b, \rho) = 2$, $h_Make_2(b, \rho) = 0$, $hM\_MakeCount(b, \rho) = 4$
  $hM\_score(b, \rho) = 4$
- $h\_Break_1(c, \rho) = 0$, $h\_Break_2(c, \rho) = 0$, $hM\_BreakCount(c, \rho) = 0$
  $h\_Make_1(c, \rho) = 2$, $h\_Make_2(c, \rho) = 0$, $hM\_MakeCount(c, \rho) = 4$
  $hM\_score(c, \rho) = 4$

Il ne reste plus qu'à remplacer l'appel à la fonction $score$ de l'algorithme 8 par la fonction hM_score, et nous pouvons calculer le meilleur renommage minimisant le nombre de littéraux positifs des clauses qui ne sont pas de Horn. Ce renommage sera appelé renommage *Horn_Min_Littéraux* (HML).

4.3.4 Expérimentations

Dans toutes les expérimentations qui ont été menées pour tout le chapitre, les paramètres passés à l'algorithme walkHorn avaient pour valeur : MAX_TRIES = 20 et MAX_FLIPS = 50000 ou bien un temps limite de 200 secondes qui n'est atteint qu'exceptionnellement pour les très grosses instances. Le temps moyens passé à calculer le renommage est de l'ordre de quelques secondes seulement.

Problèmes aléatoires

Nous avons testé cette méthode de calcul d'ensemble strong backdoor dans un premier temps sur les instances générées aléatoirement. Comme précédemment, les instances testées comprennent 100, 200, 300 et 400 variables. Pour chaque classe nous avons fait varier le rapport $\frac{nbCla}{nbVar}$ de 0.25 jusque 9. Pour chaque ratio, nous avons générés 50 instances et calculé la moyenne du nombre de variables apparaissant dans l'ensemble strong Horn-backdoor suivant trois approches :

1. nous avons considéré la formule originale (sans renommage) et calculé un strong backdoor à l'aide de l'algorithme 7.
2. nous avons considéré la formule renommée par le meilleur renommage fourni par l'algorithme 8 selon le critère du nombre de clauses Horn-renommées et calculé un ensemble strong backdoor à l'aide de l'algorithme 7 (Le renommage utilisé est le renommage *Horn_Min_Clause* (HMC)).

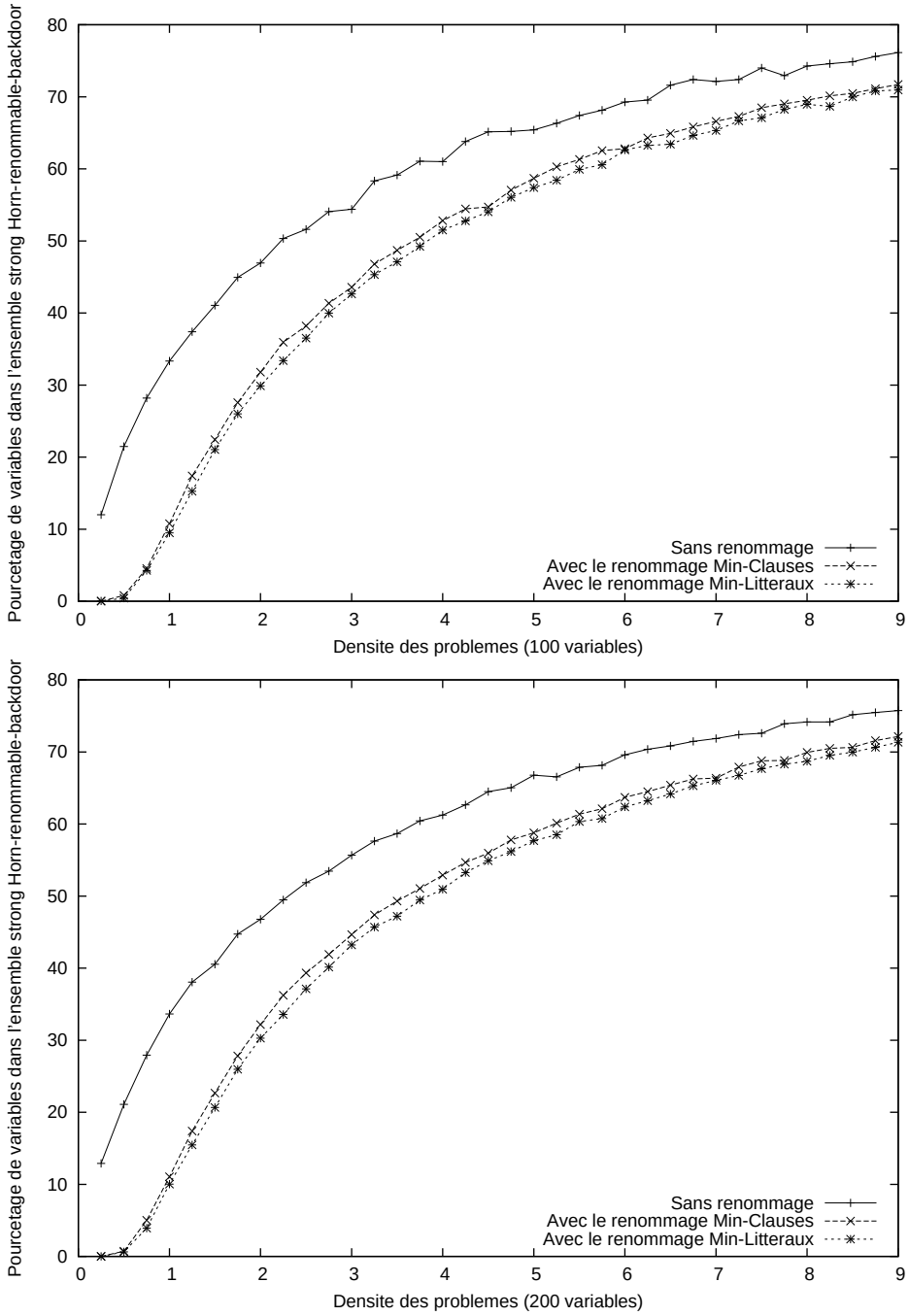


FIG. 4.2 – Tailles des ensembles strong Horn-renommable-backdoor pour des instances 3-SAT aléatoires à 100 et 200 variables.

3. nous avons considéré la formule renommée par le meilleur renommage fourni par l'algorithme 8 selon le critère du nombre de littéraux positifs figurant dans la partie non Horn de la formule renommée et calculé un strong backdoor à l'aide de l'algorithme 7 (Le renommage utilisé est le renommage *Horn_Min_Littéraux* (HML)).

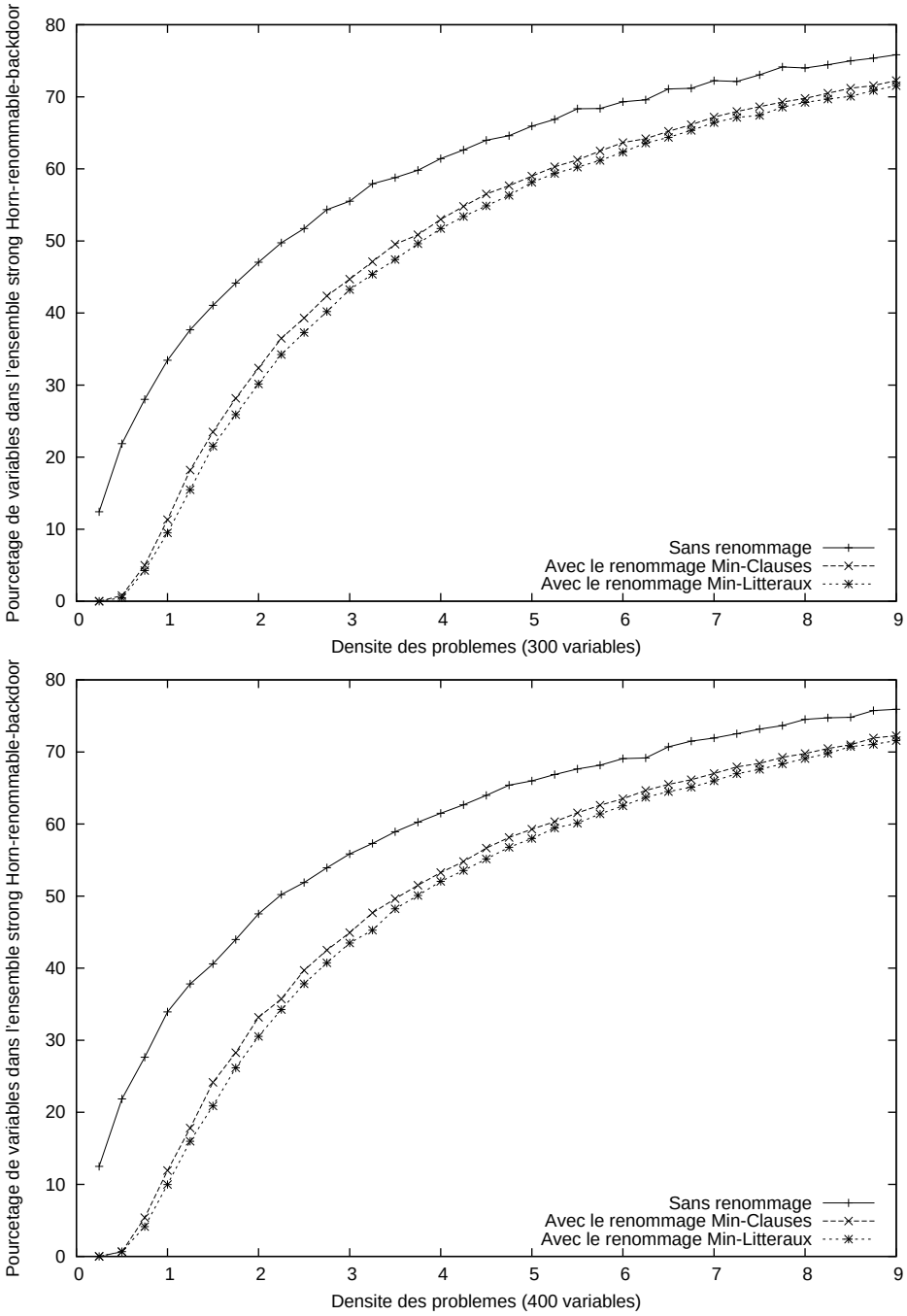


FIG. 4.3 – Tailles des ensembles strong Horn-renommable-backdoor pour des instances 3-SAT aléatoires à 300 et 400 variables.

Les figures 4.2 et 4.3 récapitulent les résultats pour chaque classe d'instances. Nous remarquons que comparativement à la formule originale, nous obtenons de meilleurs résultats en terme de taille des ensembles strong Horn-backdoor quelque soit le renommage choisi. De plus la nouvelle heuristique du renommage minimisant le nombre de littéraux (HML) se trouve être

légèrement meilleure que celle minimisant le nombre de clauses (HMC) proposée en terme de taille des ensembles strong Horn-backdoor.

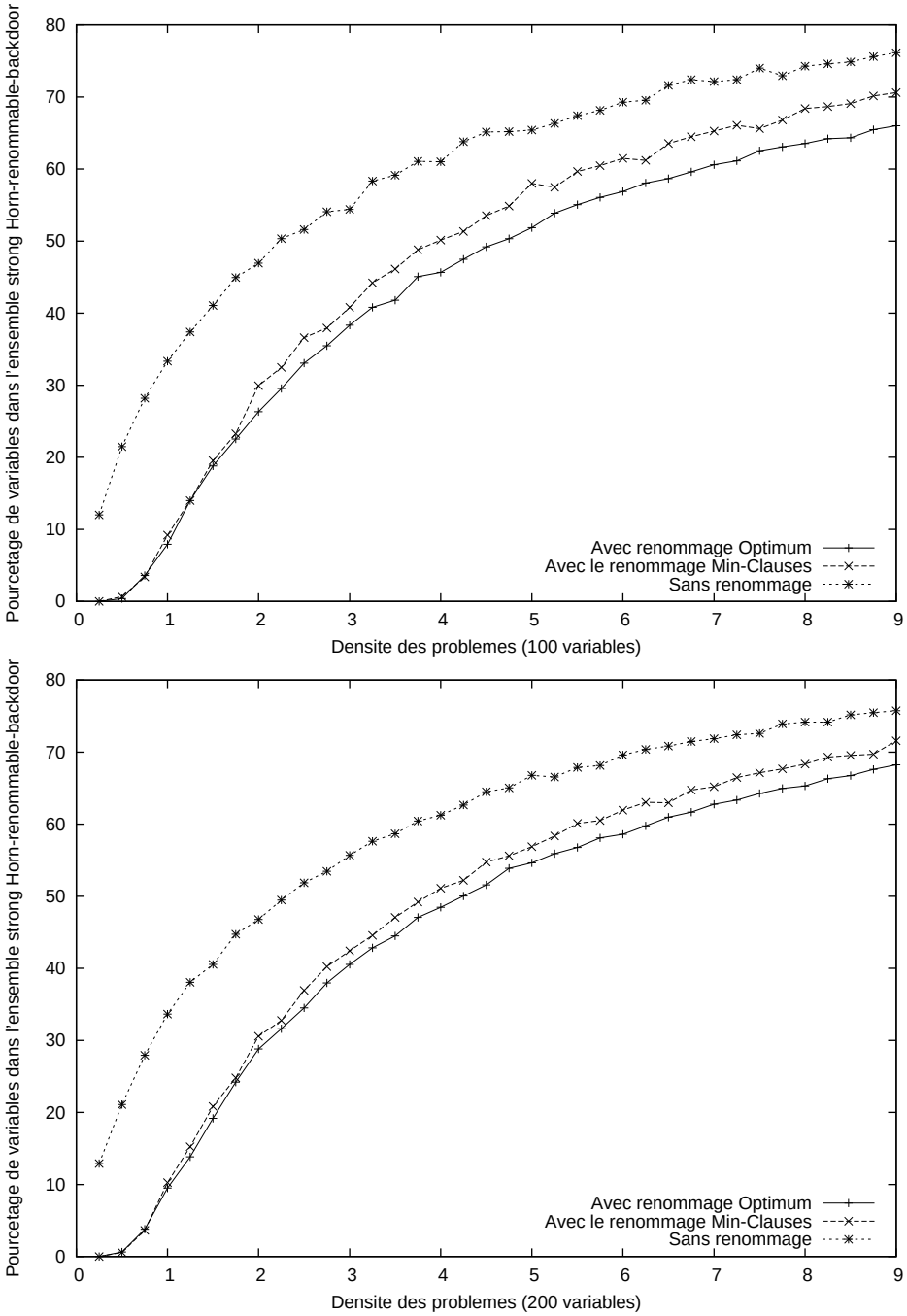


FIG. 4.4 – Tailles « optimales » des ensembles strong Horn-renommable-backdoor pour des instances 3-SAT aléatoires à 100 et 200 variables.

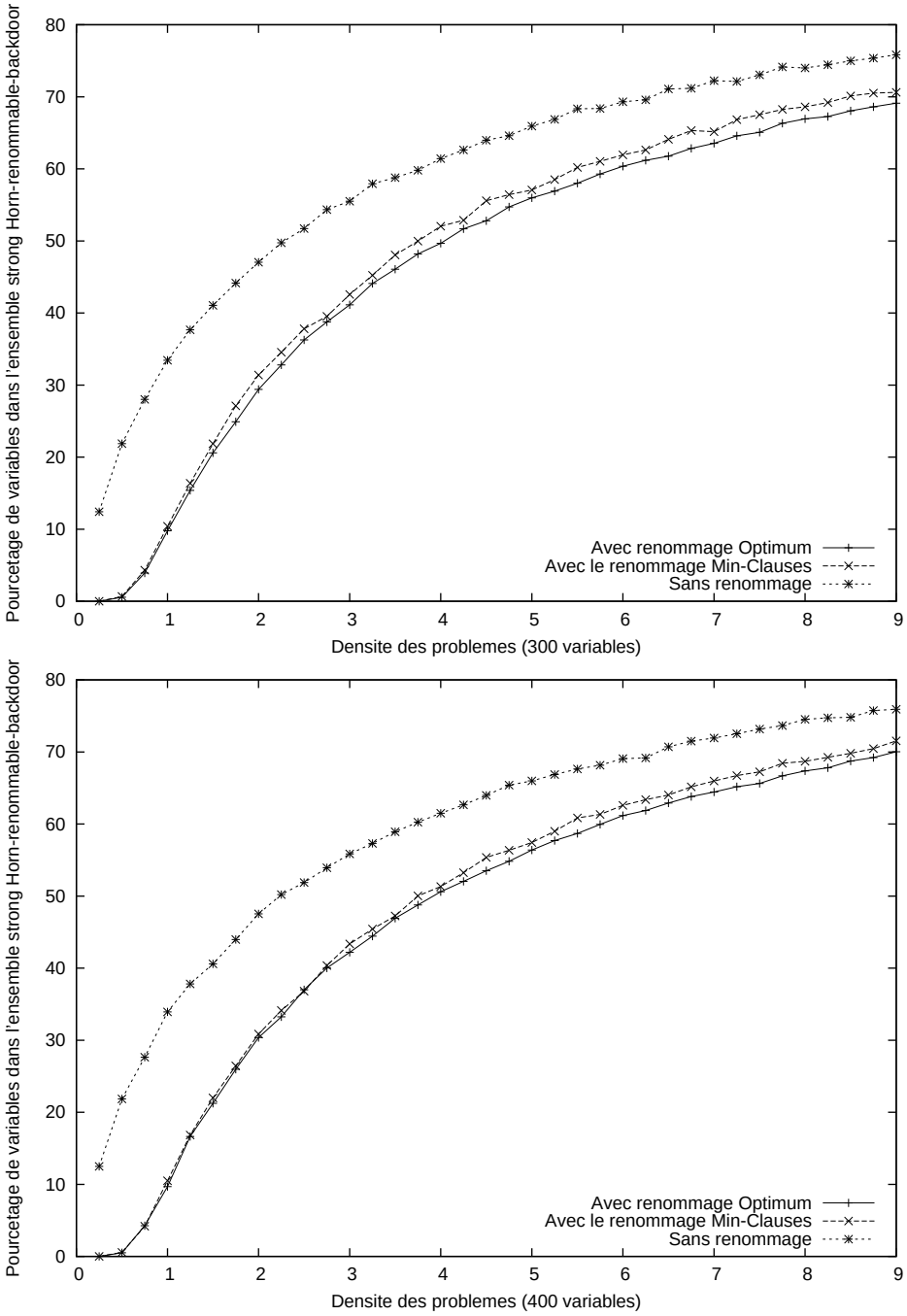


FIG. 4.5 – Tailles « optimales » des ensembles strong Horn-renommable-backdoor pour des instances 3-SAT aléatoires à 300 et 400 variables.

Méthode de calcul « optimal » Afin d'essayer d'évaluer la pertinence de la fonction objectif minimisant le nombre de clause pour le renommage HMC, nous avons expérimenté une autre méthode de calcul d'ensembles strong Horn-renommable-backdoor en utilisant un renommage que nous appellerons renommage *optimum*. Le principe de cette méthode est de calculer un ensemble

strong Horn-backdoor **pour chaque renommage** exploré par l'algorithme walkHorn et de retenir naturellement celui ayant donné le plus petit ensemble strong backdoor.

Les courbes des figures 4.4 et 4.5 établissent une comparaison entre cette méthode et la méthode utilisant le renommage HMC sur les même instances aléatoires que précédemment. Pour compléter encore cette comparaison, la taille des ensembles strong Horn-backdoor est également fournie. Cette méthode ayant un coût beaucoup plus important que les autres, nous lui avons laissé 4 fois plus de temps que les autres. Ces résultats montrent qu'un gain est réalisé mais que ce dernier tend à s'annuler lorsque la taille des instances augmente.

Nous avons réalisé la même expérience pour le renommage HML et les même constatations ont été faites. Nous n'avons donc pas poursuivi les expérimentations avec cette méthode du fait de son trop fort coût et nous considérons que les méthodes de calcul utilisant les renommages HMC et HML sont les meilleurs compromis.

Problèmes réels et industriels

Le tableau 4.2 montre les résultats obtenu en calculant des ensembles strong backdoor sur les mêmes instances que dans le tableau 4.1 (issues des précédentes compétitions SAT). Pour chaque instance, nous comparons la taille des ensembles strong Horn-backdoor calculés sans renommage, en utilisant le renommage HMC et en utilisant le renommage HML (colonnes $|SB|$). Pour chaque calcul, nous donnons également le nombre de clauses non Horn contenues soit dans la formule originale, soit dans la formule renommée ayant servi au calcul de l'ensemble strong backdoor (colonnes *NB_NH*).

Les résultats expérimentaux montrent clairement que minimiser le nombre de variables apparaissant dans la partie non Horn (renommage HML) permet de calculer des ensembles strong backdoor de taille inférieure en générale par rapport au renommage minimisant le nombre de clauses non Horn (HMC). D'un autre côté, le renommage HML produit une formule contenant plus de clauses non Horn que le renommage HMC.

Instance	Sans Renommage		Renommage HMC		Rennomage HML	
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH
9symml_gr_rcs_w5	791	6085	780	244	716	230
9symml_gr_rcs_w6	1010	7645	1003	243	897	244
alu2_gr_rcs_w7	2425	17715	2495	517	2220	491
alu2_gr_rcs_w8	2842	22301	2978	516	2602	495
apex7_gr_rcs_w5	831	3192	721	233	630	199
balancedhidden-...-01	601	1518	583	1268	564	1264
balancedhidden-...-02	594	1509	591	1264	563	1269
bf0432-007	540	920	360	431	348	416
bf1355-075	1137	1885	805	1152	729	1085
bf1355-638	1135	1873	786	1155	730	1084
bf2670-001	588	964	377	402	337	368
c499_gr_rcs_w5	934	4190	885	279	795	247
c880_gr_rcs_w5	1985	11334	1950	628	1765	586

Instance	Sans Renommage		Renommage HMC		Rennomage HML	
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH
c880_gr_rcs_w6	2509	13981	2549	651	2221	593
c880_gr_rcs_w7	3023	15598	3061	658	2676	604
ca032	305	613	242	381	227	368
ca064	599	1263	516	801	483	763
ca128	1217	2571	1035	1671	970	1587
clus-1200-4800-20-20-003	738	2386	649	1523	631	1484
clus-1200-4919-10-10-001	736	2432	643	1573	623	1527
clus-1200-4919-10-10-003	737	2461	648	1614	642	1569
cnt07	1123	2570	1062	1705	1046	1655
cnt08	2590	5943	2484	4030	2388	3918
dp03s03	218	478	206	328	171	313
dp03u02	156	340	137	219	114	211
dp04s04	461	1090	387	674	356	638
dp04u03	363	826	292	507	265	485
dp05s05	710	1641	592	1053	538	982
dp05u04	568	1336	478	847	435	793
dp06s06	1035	2436	882	1599	841	1509
dp06u05	890	2087	750	1344	684	1263
dp07s07	1416	3240	1179	2178	1102	2046
dp07u06	1232	2906	1038	1872	938	1769
dp08u07	2098	4908	1459	2623	1315	2498
dp09s09	2636	6136	2061	3811	1928	3586
dp10s10	3261	7769	2632	4889	2438	4638
dp11s11	3979	9342	3149	5870	2944	5518
dp12s12	4779	11359	3944	7248	3578	6788
example2_gr_rcs_w5	1256	6068	1179	382	1022	338
ezfact256_1	32927	211014	32929	188731	32918	182989
f2clk_50	13161	32392	13879	19503	13065	18957
fifo8_100	25046	54994	22841	33799	21012	32925
glassyb-v648-s1381725490	431	1519	398	987	383	980
hanoi4	400	1860	431	854	409	849
hanoi5	1063	5364	1115	2585	1066	2626
hanoi6_fast	3022	24187	2303	9453	2207	9509
hanoi6_on	3047	23822	2319	9474	2209	9529
hgen2-v700-s504028057	461	1239	449	999	431	995
hidden-k3-s2-r4-n700-03	430	1395	378	892	365	878
ip38	20702	52611	22972	29067	21888	26303
ip50	27257	70293	30219	41619	29396	37263
k2fix_gr_2pinvar_w9	4607	294543	4569	251956	4567	259258
okgen-c1300-v650-1167163901	312	666	213	299	190	280
okgen-c1400-v700-211648331	329	647	226	328	207	307

Instance	Sans Renommage		Renommage HMC		Rennomage HML	
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH
okgen-c1750-v500-748188212	295	904	247	509	239	500
okgen-c2100-v600-1967026721	358	1112	305	637	297	619
okgen-c2100-v600-536911537	349	1019	323	670	313	653
rand_net40-25-1	996	2024	813	1228	754	1156
rand_net40-25-10	972	2002	814	1231	725	1154
rand_net40-25-5	1002	2020	810	1234	723	1136
rand_net40-30-1	1184	2398	963	1435	881	1332
rand_net40-30-10	1186	2372	934	1431	863	1318
rand_net40-30-5	1167	2379	926	1395	867	1315
rand_net40-40-10	1562	3221	1276	1969	1176	1836
rand_net40-40-5	1572	3241	1250	1954	1156	1822
rand_net40-60-10	2364	4832	1857	2945	1697	2705
rand_net50-25-1	1247	2462	973	1504	914	1400
rand_net50-25-10	1241	2541	1007	1544	920	1423
rand_net50-25-5	1238	2506	1013	1552	935	1457
rand_net50-30-1	1497	3019	1194	1813	1085	1655
rand_net50-30-5	1496	3071	1149	1814	1058	1702
rand_net50-40-5	1973	3981	1579	2471	1439	2265
rand_net50-60-1	2971	5984	2344	3668	2141	3347
rand_net50-60-5	2982	6045	2364	3669	2176	3362
rand_net60-25-1	1482	2972	1200	1816	1103	1682
rand_net60-25-10	1480	3039	1184	1812	1089	1686
rand_net60-30-1	1830	3614	1445	2219	1292	2046
rand_net60-30-10	1762	3586	1420	2205	1309	2043
rand_net60-30-5	1798	3622	1413	2230	1331	2065
rand_net60-40-1	2408	4842	1917	2972	1744	2712
rand_net60-40-5	2372	4788	1828	2897	1709	2654
rand_net60-60-1	3530	7185	2880	4512	2640	4164
rand_net70-25-1	1762	3565	1414	2148	1312	1994
rand_net70-25-5	1753	3510	1420	2199	1310	2025
rand_net70-30-1	2095	4189	1701	2610	1536	2378
rand_net70-30-5	2073	4198	1661	2606	1520	2373
rand_net70-40-1	2767	5651	2193	3472	2020	3123
rand_net70-40-5	2767	5535	2197	3471	1983	3147
rand_net70-60-1	4158	8293	3415	5385	3179	4966
rand_net70-60-10	4145	8415	3412	5452	3138	5109
sha1	31494	121599	33956	101021	33317	99769
too_large_gr_rcs_w5	1635	8097	1620	519	1480	502
too_large_gr_rcs_w6	2032	11500	2127	538	1876	492
too_large_gr_rcs_w7	2452	12649	2561	534	2244	507
too_large_gr_rcs_w8	2877	15024	3000	532	2616	518

Instance	Sans Renommage		Renommage HMC		Rennomage HML	
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH	$\|SB\|$	NB_NH
too_large_gr_rcs_w9	3310	16541	3395	535	2987	521
unif-c1000-v500-s1356655268	248	522	169	224	157	216
unif-c1300-v650-s425854131	307	638	212	304	204	280
unif-c1400-v700-s1822569467	332	692	232	314	212	297
unif-c2450-v700-s1373176568	429	1261	356	746	345	729
unif-c2450-v700-s1465124739	410	1227	356	731	348	728
unif-c2600-v650-s2035184328	394	1327	345	789	334	765
unif-r3-v700-c2100-02	377	1030	319	591	297	567
unif-r4-v700-c2800-03	433	1417	378	877	353	861
vda_gr_rcs_w7	3507	25078	3629	755	3264	742
vda_gr_rcs_w8	4069	29453	4243	756	3834	719
vda_gr_rcs_w9	4695	33705	4853	753	4356	734
w10_45	6908	16633	7002	9127	6797	8751
w10_60	10843	26909	11307	14725	10998	14211
w10_70	13305	33736	14019	18404	13698	17895
w10_75	14850	37421	15676	20542	15345	20053

TAB. 4.2 – Tailles des ensembles Horn strong backdoor pour des instances réelles.

4.4 Calcul d'ensembles strong ordonné-backdoor

Dans cette section, nous montrons comment il est possible de calculer des ensembles strong backdoor pour une autre classe polynomiale du problème SAT : la classe des formules ordonnées introduite par Benoist et Hébrard en 1999 [Benoist et Hébrard, 1999].

4.4.1 Rappels sur les formules ordonnées

La classe des formules ordonnées peut être vue naturellement comme une extension de la classe des formules de Horn. Elles est basée sur la notion de littéraux libres et de littéraux liés [Benoist et Hébrard, 1999].

Définition 39 (Littéraux libres/liés)
Soient $c \in \Sigma$ une clause et $l \in c$ un littéral. On dit que l est lié dans c (par rapport à Σ) si $Occ(\neg l) = \emptyset$; ou s'il existe $t \in c$ $(t \neq l)$ tel que $Occ(\neg l) \subseteq Occ(\neg t)$. Si l n'est pas lié dans c, on dit que l est libre dans c.

Les formules ordonnées sont définies ainsi :

Définition 40 (Formules Ordonnées)
Une formule CNF Σ est ordonnée si chaque clause $c \in \Sigma$ contient au plus un littéral positif libre dans c.

En rappelant qu'une formule CNF est une formule de Horn si chacune de ses clauses ne contient pas plus d'un littéral positif, on voit bien que toute formule de Horn est forcément or-

donnée, ce qui étaye l'idée que la classe des formules ordonnées est une extension de la classe des formules de Horn.

On sait que si Σ est une formule de Horn et ne contient aucune clause unitaire positive, alors Σ est satisfaisable. Cette propriété reste vraie si Σ est ordonnée. Pour les formules ordonnées, nous avons également la proposition suivante (prouvée dans [Benoist et Hébrard, 1999]) :

Proposition 6 *Si Σ est ordonnée et ne contient aucune clause unitaire positive $c = \{x\}$ telle que x est libre dans c, alors Σ est satisfaisable.*

De plus, il est montré dans [Benoist et Hébrard, 1999] qu'une formule ordonnée est stable par propagation unitaire *i.e.* une formule ordonnée reste ordonnée quelque soient les propagations unitaire que l'on y réalise. De ce fait, avec la proposition 6, on voit que le problème de satisfaisabilité d'une formule ordonnée peut être résolu en temps linéaire.

Enfin, comme pour les formules de Horn, la classe des formules ordonnées peut être étendue par la classe des formules ordonnées renommables. Il existe un algorithme polynomial pour la reconnaissance de ces formules et un autre, polynomial également, pour décider de la satisfaisabilité d'une formule ordonnée renommable [Benoist et Hébrard, 1999].

Ainsi, nous pouvons envisager de calculer des ensembles strong ordonné-backdoor en utilisant la même approche que pour les ensembles strong Horn-backdoor *i.e.* pour une formule CNF originale, trouver un renommage maximisant la sous-formule ordonnée et extraire un ensemble strong ordonné-backdoor de la sous-formule non ordonnée de la formule renommée.

4.4.2 Calcul d'ensembles strong ordonné-backdoor

Cette fois encore, le problème du calcul d'un ensemble strong ordonné-backdoor de taille minimale est NP-difficile. Nous allons donc utiliser le même procédé que pour les ensembles strong Horn-backdoor *i.e.* pour une formule CNF, calculer un sous-ensemble dont la taille est la plus petite possible, contenant des littéraux *libres* positifs apparaissant dans la sous-formule non ordonnée tels qu'une fois retirés les littéraux de cet ensemble, le reste de la formule est ordonné. Cependant, ceci n'est valable que si le retrait de ces littéraux, par leur affectation à *vrai* ou à *faux*, ne rend pas libres d'autres littéraux de la formule qui étaient liés. La proposition suivante nous assure que c'est le cas :

Proposition 7 *Soit Σ une formule CNF. Soit $\mathcal{U}$ un ensemble de clauses unitaires sur $\mathcal{V}(\Sigma)$. Soit I une interprétation telle que $I = \mathcal{V}(\mathcal{U})$. Soit $c \in \Sigma$, $l \in c$ tel que l est lié dans c par rapport à Σ, alors l est aussi lié dans c' par rapport à $\Sigma' = \Sigma^*_I$ ($c' = c^*_I$), ou bien c est retirée (satisfaite) de Σ'.*

Preuve *Du fait que l est lié, deux cas sont possibles :*

- *$Occ_\Sigma(\neg l) = \emptyset$: dans ce cas, il est trivial que $Occ_{\Sigma'}(\neg l) = \emptyset$, et l est toujours lié dans Σ' dans toutes les clauses où il apparaît.*
- *$Occ_\Sigma(\neg l) \neq \emptyset$: $\exists t \in c$ tel que $Occ_\Sigma(\neg l) \subseteq Occ_\Sigma(\neg t)$. Ici trois cas sont possibles :*
  - *$t \in \mathcal{U}$: dans ce cas, la clause c est satisfaite dans Σ'.*
  - *$\neg t \in \mathcal{U}$: dans ce cas, chaque clause contenant $\neg t$ sont satisfaites (retirées) in Σ'. Du fait que $Occ_\Sigma(\neg l) \subseteq Occ_\Sigma(\neg t)$, toutes les clauses contenant $\neg l$ sont aussi satisfaites dans Σ'. Ainsi $Occ_{\Sigma'}(\neg l) = \emptyset$ et l est toujours liée dans c' par rapport à Σ'.*

– *$t \notin \mathcal{U}$ et $\neg t \notin \mathcal{U}$: soit S l'ensemble des clauses de Σ satisfaites (retirées) dans Σ'. On a $Occ_{\Sigma'}(\neg t) = Occ_{\Sigma \setminus S}(\neg t)$ et $Occ_{\Sigma'}(\neg l) = Occ_{\Sigma \setminus S}(\neg l)$. Du fait que $Occ_{\Sigma}(\neg l) \subseteq Occ_{\Sigma}(\neg t)$, on a $Occ_{\Sigma \setminus S}(\neg l) \subseteq Occ_{\Sigma \setminus S}(\neg t)$. Donc $Occ_{\Sigma'}(\neg l) \subseteq Occ_{\Sigma'}(\neg t)$, et l est toujours lié dans c' par rapport à Σ'.*

Avec cette proposition, nous sommes assurés de la stabilité de la liaison pour l'affection *i.e.* dans une formule CNF, toute variable liée dans une clause reste liée dans une clause de la formule simplifiée par n'importe quelle affectation. Ainsi, nous pouvons calculer un ensemble strong ordonné-backdoor S de taille la plus petite possible comme suit : pour une formule CNF Σ quelconque, nous ne considèrons que la partie non ordonnée ψ de Σ. Nous construisons S en y ajoutant des variables dont les occurrences positives libres figurent dans $\mathcal{L}^+(\psi)$ que nous retirons de ψ itérativement jusqu'à l'obtention d'une sous-formule de ψ qui soit ordonnée. À chaque étape, la variable apparaissant le plus souvent sous forme positive et libre dans ψ est choisie. Nous utilisons l'algorithme 7 en rajoutant simplement la condition que les littéraux choisis soient libres.

Nous voyons donc que par rapport à la classe des formules de Horn, les ensembles strong backdoor calculés pour les formules ordonnées sont de taille au pire égale. En effet, dans le cas où aucun littéral n'est lié, les ensembles strong ordonné-backdoor et strong Horn-backdoor seront identiques. Mais dès lors qu'il y a des littéraux liés, ils peuvent être ignorés et donc réduire la taille des ensembles strong backdoor.

4.5 Calcul d'ensembles strong *ordonné-renommable*-backdoor

Comme dans le cas des formules de Horn, la taille des ensembles strong ordonné-backdoor que nous calculons avec cette méthode heuristique est souvent trop importante pour être intéressante. Nous proposons donc de calculer des ensembles strong *ordonné-renommable*-backdoor en suivant le même schéma que pour les formules de Horn, en calculant le meilleur ordonné-renommage possible.

4.5.1 Approximation de la sous-formule ordonné-renommable maximale

Comme dans le cas des formules de Horn, il existe un algorithme de complexité polynomial pour savoir si une formule est ordonné-renommable ([Benoist et Hébrard, 1999]), et comme pour les formule de Horn, si la formule n'est pas ordonné-renommable, calculer la sous-formule ordonné-renommable maximale est un problème NP-difficile. En effet, dans le pire des cas, si aucun littéral n'est lié, la classe des formules ordonnées est équivalente à la classe des formules de Horn.

La proposition suivante nous permet d'exploiter l'algorithme 8 pour approximer la sous-formule ordonné-renommable maximale :

Proposition 8 *Soient Σ une formule CNF et $c \in \Sigma$ une clause de Σ. Soient l et t deux littéraux de c. Soit ρ un renommage des variables de Σ ($\mathcal{V}(\Sigma)$). Si l est lié à t dans c par rapport à Σ, alors $\rho(l)$ est toujours lié à $\rho(t)$ dans $\rho_c(c)$ par rapport à $\rho_\Sigma(\Sigma)$.*

Preuve *Il suffit de constater que si une variable $v \in \mathcal{V}(\Sigma)$ est renommée dans ρ, alors, pour les littéraux qui lui sont associés v et $\neg v$, on a $Occ_\Sigma(v) = Occ_{\rho_\Sigma(\Sigma)}(\rho(v)) = Occ_{\rho_\Sigma(\Sigma)}(\neg v)$ et $Occ_\Sigma(\neg v) = Occ_{\rho_\Sigma(\Sigma)}(\neg\rho(v)) = Occ_{\rho_\Sigma(\Sigma)}(v)$.*

Cette proposition nous assure que la liaison entre deux littéraux est stable par renommage. Ainsi, pour une formule CNF donnée, il suffit de calculer une seule fois au préalable une table des liaisons pour chaque littéral de la formule, et à tout moment, il suffira de regarder dans cette table si un littéral est lié ou non (en fonction du renommage).

Ici encore, deux critères pour calculer le meilleur renommage sont possibles : minimiser le nombre de clauses non ordonnées, ou bien minimiser le nombre de littéraux positifs libres dans l'ensemble des clauses non ordonnées.

Nous avons besoin des définitions suivantes avant de pouvoir décrire les deux fonctions objectifs que nous proposons.

Définition 70 (Littéral renommé libre)
Soient Σ une formule CNF, ρ un renommage de $\mathcal{V}(\Sigma)$. On dit qu'un littéral l est positif et libre dans c pour ρ si $estPos(l, c, \rho)$ et que l n'est pas lié dans c par rapport à Σ. On note cela $estPos\&Libre(l, c, \rho)$. Ce même littéral est négatif et libre dans c pour ρ si $estNeg(l, c, \rho)$ et que l n'est pas lié dans c par rapport à Σ. On note cela $estNeg\&Libre(l, c, \rho)$.

On note le nombre de littéraux positifs (resp. négatif) et libres dans c pour ρ par $nbPos\&Libre(c, \rho)$ (resp. $nbNeg\&Libre(c, \rho)$).

On note $nbPos\&LibreTot(\Sigma, \rho)$ le nombre total de littéraux positifs et libres présents dans la partie non ordonnée de Σ pour le renommage ρ.

Cette définition nous permet de définir la notion d'*ordonné-renommabilité* :

Définition 71 (Clause ordonné-renommée)
Soient Σ une formule CNF et ρ un renommage. Une clause $c \in \Sigma$ est dite ordonné-renommée par ρ (notée $o\_ren(c, \rho)$) si $nbPos\&Libre(c, \rho) \leq 1$ i.e. c contient au plus un littéral positif libre pour ρ ; sinon elle est dite ordonné-falsifiée par ρ (notée $o\_fal(c, \rho)$).

On note $nbOrd(\Sigma, \rho)$ le nombre de clauses de Σ ordonné-renommées par ρ.

Il ne nous reste plus qu'à définir les compteurs de gain (*MakeCount*) et de perte (*BreakCount*), ainsi que les fonctions $score$ associées pour chacune des deux fonctions objectifs.

4.5.2 Le renommage ordonné_min_clauses : fonction objectif min-conflict

Les compteurs de gain et de perte permettant de définir la fonction objectif calculant le meilleur ordonné-renommage sont définis comme suit :

Définition 72 (Ordonné-BreakCount et ordonné-MakeCount)
Soient Σ une formule CNF, $x \in \mathcal{V}(\Sigma)$, ρ un renommage et ρ_x le renommage obtenu à partir de ρ en inversant la valeur de vérité de x.
On définit $o\_BreakCount(x, \rho) = |\{c | o\_ren(c, \rho), o\_fal(c, \rho_x)|$ et
$o\_MakeCount(x, \rho) = |\{c | o\_fal(c, \rho), o\_ren(c, \rho_x)\}|$.
On définit $o\_score(x, \rho) = o\_makeCount(x, \rho) - o\_breakCount(x, \rho)$

Exemple 16 (Ordonné-BreakCount et ordonné-MakeCount) *Reprenons la formule CNF :*

$$\Sigma = \{c_1 = (\neg a \vee b \vee c),\ c_2 = (a \vee b),\ c_3 = (a \vee c),\ c_4 = (\neg b \vee \neg c)\}$$

et le renommage $\rho = \{\neg a, \neg b, \neg c\}$ (aucune variable renommée).

Les seuls littéraux liés sont dans la clauses c_1, où le littéral b est lié au littéral c et le littéral c est lié au littéral b. Ceci fait que la clause c_1 est ordonnée.

Les différents compteurs valent :

$o\_BreakCount(a, \rho) = 0$, $o\_MakeCount(a, \rho) = 2$ *et* $o\_score(a, \rho) = 2$

$o\_BreakCount(b, \rho) = 0$, $o\_MakeCount(b, \rho) = 1$ *et* $o\_score(b, \rho) = 1$

$o\_BreakCount(c, \rho) = 0$, $o\_MakeCount(c, \rho) = 1$ *et* $o\_score(c, \rho) = 1$

Pour une formule CNF donnée, en remplaçant $nbHorn(\Sigma)$ par $nbOrd(\Sigma)$ et $score$ par o_score dans l'algorithme 8, on obtient un algorithme pour calculer un renommage de Σ maximisant le nombre de clauses ordonné-renommées. Ce renommage sera appelé renommage *Ordonné_Min_Clause* (OMC).

4.5.3 Le renommage ordonné_min_littéraux : fonction objectif min-size

Pour la minimisation du nombre de littéraux positifs libres dans les clauses non ordonnées, la fonction objectif est définie par :

Définition 73 (Ordonné-Min-breakCount et ordonné-Min-makeCount)
Soient Σ une formule CNF, $x \in \mathcal{V}(\Sigma)$, ρ un renommage et ρ_x le renommage obtenu à partir de ρ en inversant la valeur de vérité de x.
Soient $o\_Break_1(x, \rho) = \{c \in \Sigma | nbPos\&Libre(c, \rho) = 1$ *et* $estNeg\&Libre(x, c, \rho)\}$,
$o\_Break_2(x, \rho) = \{c \in \Sigma | nbPos\&Libre(c, \rho) > 1$ *et* $estNeg\&Libre(x, c, \rho)\}$,
$o\_Make_1(x, \rho) = \{c \in \Sigma | nbPos\&Libre(c, \rho) = 2$ *et* $estPos\&Libre(x, c, \rho)\}$
et $o\_Make_2(x, \rho) = \{c \in \Sigma | nbPos\&Libre(c, \rho) > 2$ *et* $estPos\&Libre(x, c, \rho)\}$.
Les compteurs sont définis comme suit :
$oM\_breakCount(x, \rho) = 2 \times |o\_Break_1(x, \rho)| + |o\_Break_2(x, \rho)|$ *et*
$oM\_makeCount(x, \rho) = 2 \times |o\_Make_1(x, \rho)| + |o\_Make_2(x, \rho)|$.
Enfin on définit : $oM\_score(x, \rho) = oM\_makeCount(x, \rho) - oM\_breakCount(x, \rho)$

Les ensembles o_Break et o_make sont analogues aux ensembles h_Break et h_make (définition 69) respectivement, pour la classe des formules ordonnées avec la contrainte supplémentaire que x doit être libre à chaque fois, car tous les littéraux liés sont ignorés.

Exemple 17 (Ordonné-BreakCount et ordonné-MakeCount) *Reprenons la formule CNF :*

$$\Sigma = \{c_1 = (\neg a \vee b \vee c),\ c_2 = (a \vee b),\ c_3 = (a \vee c),\ c_4 = (\neg b \vee \neg c)\}$$

et le renommage $\rho = \{\neg a, \neg b, \neg c\}$ (aucune variable renommée).

Les différents compteurs valent :

Les seuls littéraux liés sont dans la clauses c_1, où le littéral b est lié au littéral c et le littéral c est lié au littéral b. Ceci fait que la clause c_1 est ordonnée.

Les différents compteurs valent :

- $o\_Break_1(a, \rho) = 0$, $o\_Break_2(a, \rho) = 0$, $oM\_BreakCount(a, \rho) = 0$
  $o\_Make_1(a, \rho) = 2$, $o\_Make_2(a, \rho) = 0$, $oM\_MakeCount(a, \rho) = 4$
  $oM\_score(a, \rho) = 4$
- $o\_Break_1(b, \rho) = 0$, $o\_Break_2(b, \rho) = 0$, $oM\_BreakCount(b, \rho) = 0$
  $o\_Make_1(b, \rho) = 1$, $o_Make_2(b, \rho) = 0$, $oM\_MakeCount(b, \rho) = 2$
  $oM\_score(b, \rho) = 2$

– $o\_Break_1(c,\rho) = 0$, $o\_Break_2(c,\rho) = 0$, $oM\_BreakCount(c,\rho) = 0$
$o\_Make_1(c,\rho) = 1$, $o\_Make_2(c,\rho) = 0$, $oM\_MakeCount(c,\rho) = 2$
$oM\_score(c,\rho) = 2$

En remplaçant $score$ par oM_score dans l'algorithme 8, on obtient un algorithme pour calculer un renommage minimisant le nombre de littéraux positifs et libres dans les clauses non ordonné-renommées. Ce renommage sera appelé renommage *Ordonné_Min_Littéraux* (OML).

4.5.4 Expérimentations

Problèmes aléatoires

Tout comme dans le cas des formules de Horn, nous avons testé cette approche sur les instances aléatoires pour un nombre de variables allant de 100 à 400 et avons fait varier le rapport $\frac{nbCla}{nbVar}$. Nous obtenons exactement les même courbes que dans la figure 4.2 excepté lorsque le ratio est inférieur à 2 où la taille des ensembles strong ordonné-backdoor est très légèrement inférieure à celle des ensembles strong Horn-backdoor (environ 1% de différence). En effet, dès lors que le ratio dépasse 2, le nombre de littéraux liés devient quasi-nul et tend très rapidement vers 0 lorsque le ratio augmente. Dans ce cas, les formules ordonnées deviennent strictement équivalentes au formules de Horn et il est donc normal que l'on obtienne les même résultats que pour les formules de Horn en terme de taille des ensembles strong backdoor.

Problèmes réels et industriels

Instance	Horn		Ordonné			
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NO	$\|SB\|$	NB_NO
9symml_gr_rcs_w5	716	230	796	244	725	231
9symml_gr_rcs_w6	897	244	1016	249	900	245
alu2_gr_rcs_w7	2220	491	2495	517	2220	491
alu2_gr_rcs_w8	2602	495	2978	516	2602	495
apex7_gr_rcs_w5	630	199	721	233	630	199
balancedhidden-...-01	564	1264	583	1268	564	1264
balancedhidden-...-02	563	1269	591	1264	563	1269
bf0432-007	348	416	378	432	343	415
bf1355-075	729	1085	784	1133	724	1096
bf1355-638	730	1084	794	1137	722	1091
bf2670-001	337	368	359	395	319	357
c499_gr_rcs_w5	795	247	885	279	795	247
c880_gr_rcs_w5	1765	586	1980	637	1758	596
c880_gr_rcs_w6	2221	593	2504	649	2225	603
c880_gr_rcs_w7	2676	604	3143	657	2644	605
ca032	227	368	234	363	218	345
ca064	483	763	479	762	438	729
ca128	970	1587	982	1578	920	1505
clus-1200-4800-20-20-003	631	1484	650	1511	634	1483
clus-1200-4919-10-10-001	623	1527	643	1570	642	1536
clus-1200-4919-10-10-003	642	1569	659	1604	642	1581

Instance	Horn		Ordonné			
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NO	$\|SB\|$	NB_NO
cnt07	1046	1655	1062	1705	1046	1655
cnt08	2388	3918	2484	4030	2388	3918
dp03s03	171	313	195	314	173	297
dp03u02	114	211	122	206	112	201
dp04s04	356	638	378	667	354	633
dp04u03	265	485	291	496	262	476
dp05s05	538	982	598	1040	552	981
dp05u04	435	793	469	825	432	788
dp06s06	841	1509	864	1590	827	1517
dp06u05	684	1263	745	1330	673	1261
dp07s07	1102	2046	1173	2159	1110	2042
dp07u06	938	1769	1002	1834	949	1751
dp08u07	1315	2498	1478	2618	1340	2467
dp09s09	1928	3586	2067	3827	1933	3567
dp10s10	2438	4638	2645	4881	2399	4625
dp11s11	2944	5518	3155	5866	2930	5528
dp12s12	3578	6788	3865	7179	3596	6776
example2_gr_rcs_w5	1022	338	1163	384	1004	334
ezfact256_1	32918	182989	32929	188731	32918	182989
f2clk_50	13065	18957	13439	19115	12621	18567
fifo8_100	21012	32925	21372	32654	19558	31678
glassyb-v648-s1381725490	383	980	398	987	383	980
hanoi4	409	849	419	851	392	866
hanoi5	1066	2626	1124	2584	1040	2633
hanoi6_fast	2207	9509	2329	9401	2232	9436
hanoi6_on	2209	9529	2312	9366	2209	9406
hgen2-v700-s504028057	431	995	449	999	431	995
hidden-k3-s2-r4-n700-03	365	878	374	890	374	874
ip38	21888	26303	22488	28579	21535	25955
ip50	29396	37263	29835	41294	29038	36826
k2fix_gr_2pinvar_w9	4567	259258	4563	251799	4547	259307
okgen-c1300-v650-1167163901	190	280	204	294	196	272
okgen-c1400-v700-211648331	207	307	235	324	211	301
okgen-c1750-v500-748188212	239	500	250	511	240	506
okgen-c2100-v600-1967026721	297	619	304	634	292	628
okgen-c2100-v600-536911537	313	653	316	673	305	650
rand_net40-25-1	754	1156	813	1228	754	1156
rand_net40-25-10	725	1154	814	1231	725	1154
rand_net40-25-5	723	1136	810	1234	723	1136
rand_net40-30-1	881	1332	963	1435	881	1332
rand_net40-30-10	863	1318	934	1431	863	1318
rand_net40-30-5	867	1315	926	1395	867	1315
rand_net40-40-10	1176	1836	1276	1969	1176	1836
rand_net40-40-5	1156	1822	1250	1954	1156	1822
rand_net40-60-10	1697	2705	1857	2945	1697	2705
rand_net50-25-1	914	1400	973	1504	914	1400

Instance	Horn		Ordonné			
	$\|SB\|$	NB_NH	$\|SB\|$	NB_NO	$\|SB\|$	NB_NO
rand_net50-25-10	920	1423	1007	1544	920	1423
rand_net50-25-5	935	1457	1013	1552	935	1457
rand_net50-30-1	1085	1655	1194	1813	1085	1655
rand_net50-30-5	1058	1702	1149	1814	1058	1702
rand_net50-40-5	1439	2265	1579	2471	1439	2265
rand_net50-60-1	2141	3347	2344	3668	2141	3347
rand_net50-60-5	2176	3362	2364	3669	2176	3362
rand_net60-25-1	1103	1682	1200	1816	1103	1682
rand_net60-25-10	1089	1686	1184	1812	1089	1686
rand_net60-30-1	1292	2046	1445	2219	1292	2046
rand_net60-30-10	1309	2043	1420	2205	1309	2043
rand_net60-30-5	1331	2065	1413	2230	1331	2065
rand_net60-40-1	1744	2712	1917	2972	1744	2712
rand_net60-40-5	1709	2654	1828	2897	1709	2654
rand_net60-60-1	2640	4164	2880	4512	2640	4164
rand_net70-25-1	1312	1994	1414	2148	1312	1994
rand_net70-25-5	1310	2025	1420	2199	1310	2025
rand_net70-30-1	1536	2378	1701	2610	1536	2378
rand_net70-30-5	1520	2373	1661	2606	1520	2373
rand_net70-40-1	2020	3123	2193	3472	2020	3123
rand_net70-40-5	1983	3147	2197	3471	1983	3147
rand_net70-60-1	3179	4966	3415	5385	3179	4966
rand_net70-60-10	3138	5109	3412	5452	3138	5109
sha1	33317	99769	33918	101089	33364	99856
too_large_gr_rcs_w5	1480	502	1611	521	1485	490
too_large_gr_rcs_w6	1876	492	2078	530	1863	505
too_large_gr_rcs_w7	2244	507	2509	531	2240	500
too_large_gr_rcs_w8	2616	518	3005	535	2630	503
too_large_gr_rcs_w9	2987	521	3376	530	3002	521
unif-c1000-v500-s1356655268	157	216	169	221	155	209
unif-c1300-v650-s425854131	204	280	213	298	202	285
unif-c1400-v700-s1822569467	212	297	230	312	217	302
unif-c2450-v700-s1373176568	345	729	358	749	346	724
unif-c2450-v700-s1465124739	348	728	351	739	349	726
unif-c2600-v650-s2035184328	334	765	333	789	332	782
unif-r3-v700-c2100-02	297	567	317	590	292	568
unif-r4-v700-c2800-03	353	861	378	877	353	861
vda_gr_rcs_w7	3264	742	3636	758	3272	731
vda_gr_rcs_w8	3834	719	4210	756	3820	735
vda_gr_rcs_w9	4356	734	4822	756	4356	729
w10_45	6797	8751	6864	9023	6596	8633
w10_60	10998	14211	11175	14619	10901	14108
w10_70	13698	17895	13777	18293	13619	17686
w10_75	15345	20053	15479	20428	15279	19789

TAB. 4.3 – Tailles des ensembles strong backdoor pour des instances réelles.

En ce qui concerne des problèmes réels issus des précédentes compétitions SAT, le tableau 4.3 montre que la généralisation du calcul d'un ensemble strong backdoor, en considérant cette fois ci la classe des formules ordonnées, nous permet d'obtenir de meilleurs résultats sur de nombreuses

$\#V$	$\#C$	**Zchaff**			**Zchaff+**SB_{MC}			**Zchaff+**SB_{ML}		
		Temps(s)	Nœuds	MxD	Temps(s)	Nœuds	MxD	Temps(s)	Nœuds	MxD
200	850	5,15	55610,76	28,26	4,18	47071,1	25,52	4,07	46215,26	25,88
250	1062	192,18	388970,5	34,18	133,21	328879,58	30,46	138,4	332056,52	30,62
300	1275	4499,91	2287975,5	39	3601,22	2016263,68	34,9	3302,74	1978685,62	34,7
350	1487	29509,85	7088901,68	45,16	27057,2	6850739,48	40,41	26752,72	6801607,05	40,32

TAB. 4.4 – ZChaff sur les instances aléatoires

instances. Tout comme le tableau 4.2, nous donnons pour chaque instance, la taille des ensembles strong backdoor calculés en utilisant le renommage HML, celle des ensembles strong backdoor calculés en utilisant le renommage OMC et celle utilisant OML (colonnes $|SB|$) ainsi que le nombre de clauses non Horn (colonne *NB_NH*) ou non ordonnées (colonnes *NB_NO*) de la formule renommée qui a permis le calcul de l'ensemble strong backdoor. Comparativement à l'approche utilisant HML, nous remarquons que sur de nombreux problèmes, la généralisation aux formules ordonnés permet d'obtenir de meilleurs ensembles strong backdoor.

Ceci met en évidence le phénomène surprenant suivant : bien que l'on ne trouve jamais d'instances ordonnées (renommables) en pratique, un certain nombre d'instances codant des applications réelles contiennent un nombre plutôt important de littéraux liés, comme en atteste la taille des ensembles strong ordonné-backdoor par rapport à celle des ensembles strong Horn-backdoor.

4.6 Exploitation des ensembles strong backdoor pour la résolution pratique

Dans cette section, nous présentons les résultats expérimentaux que nous avons obtenus en exploitant les ensembles strong backdoor calculés précédemment lors de la résolution d'instances SAT. Nous avons utilisé le solveur *Zchaff*[2] [Moskewicz et al., 2001] que nous avons modifié pour exploiter les ensembles strong backdoor. Les modifications apportées au solveur consiste à restreindre les choix de l'heuristique de choix de variables aux variables présentes dans les ensembles strong backdoor. Une fois que toutes les variables contenues ces ensembles sont instanciées, étant donné que *Zchaff* intègre la propagation unitaire suffisante pour décider de la satisfaisabilité d'un ensemble de clauses de Horn ou de clauses ordonnées, nous pouvons stopper l'exploration et décider de la satisfaisabilité s'il n'y a pas de clause falsifiée. Si une clause est falsifiée, *Zchaff* déclenche de lui-même un retour-arrière.

Ainsi, la complexité dans le pire des cas de notre approches se trouve être en $\mathcal{O}(2^{|B|})$ ou B est l'ensemble strong backdoor que nous avons calculé.

Zchaff est écrit en C++, compilé sous linux et testé sur un ordinateur P_{IV} cadencé à 3.0 Ghz avec 1 Go de RAM.

4.6.1 Instances aléatoires

Les premières expérimentations ont été réalisées sur des instances aléatoires toutes non satisfaisables, avec un ratio $\frac{\#C}{\#V} = 4.25$. Nous avons choisi des instances non satisfaisables afin d'être sûrs que le temps de réponse du solveur n'est pas la conséquence du choix chanceux de l'heuristique qui aurait mené directement à une solution.

[2] Version 2003.

Comme nous l'avons signalé dans la section précédente, étant donné que le ratio est supérieur à 2, les ensembles strong backdoor pour les deux classes des formules de Horn et des formules ordonnées sont identiques. C'est pourquoi dans le tableau 4.4, nous ne précisons pas pour quelle classe l'ensemble strong backdoor a été calculé, mais seulement le renommage utilisé [3] : MC pour minimisation du nombre de clauses et ML pour minimisation du nombre de littéraux.

Dans ce tableau, chaque ligne fournit des données moyennes sur 50 instances. On peut y trouver le temps nécessaire en seconde pour la résolution (Temps), le nombre de nœuds explorés pendant la recherche (Nœuds), ainsi que la profondeur maximum de l'arbre de recherche atteint par le solveur pendant la recherche (MxD), et ceci pour les trois méthodes expérimentées. Nous pouvons constater qu'à tout point de vue, l'exploitation des ensembles strong backdoor améliore significativement les performances de Zchaff, et il semblerait que plus la taille des problèmes augmente, plus l'approche ML se différencie en mieux de l'approche MC.

Nous voyons que les ensembles strong backdoor contiennent en moyenne 50% des variables des instances aléatoires, ce qui nous permet d'avancer une complexité de $2^{n/2}$ dans le pire des cas. Or, comme il a été remarqué dans [Audemard et al., 2000], la profondeur de l'arbre de recherche développé par un solveur sur une instance 3-SAT aléatoire au seuil est bien inférieure à $n/2$, ce qui pourrait laisser imaginer qu'il existe des ensembles strong backdoor bien inférieur en taille. Cependant, du fait des heuristiques de choix dynamiques, il se peut que 100% des variables de l'instance soient instanciées au moins une fois pendant la recherche. Notre approche nous permet de décider de la satisfaisabilité en ne s'intéressant qu'à 50% des variables.

4.6.2 Instances réelles et industrielles

[3] En sachant que les deux classes ont produit les même résultats

Instance	S/U	zchaff		zchaff+SB_{HMC}		zchaff+SB_{HML}		zchaff+SB_{OMC}		zchaff+SB_{OML}	
		tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds
9symml_gr_rcs_w5	U	0,06	1418	0,1	1405	0,1	1371	0,09	1176	0,09	1093
9symml_gr_rcs_w6	S	0,07	1303	0,1	1132	0,1	1092	0,11	1244	0,1	1218
alu2_gr_rcs_w7	U	11,54	70064	9,1	49411	15,89	62096	12,75	55957	16,97	67533
alu2_gr_rcs_w8	S	0,19	684	0,23	643	0,22	492	0,22	591	0,22	516
apex7_gr_rcs_w5	S	0,03	701	0,04	499	0,04	426	0,04	509	0,04	548
balancedhidden-...-01	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
balancedhidden-...-02	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
bf0432-007	U	0,02	841	0,09	1594	0,07	1331	0,08	1567	0,1	1785
bf1355-075	U	0,02	239	0,03	204	0,02	125	0,03	204	0,02	160
bf1355-638	U	0,02	206	0,02	167	0,02	102	0,03	300	0,03	274
bf2670-001	U	0,01	88	0,01	95	0,01	90	0,01	108	0,01	69
c499_gr_rcs_w5	U	0,06	1738	0,12	1883	0,1	1722	0,1	1590	0,08	1124
c880_gr_rcs_w5	U	0,13	1256	0,21	1203	0,2	1232	0,22	1361	0,18	1076
c880_gr_rcs_w6	U	2,02	30562	55,41	128987	77,55	145810	65,46	133296	69,5	143863
c880_gr_rcs_w7	S	0,21	3121	0,26	1113	0,24	950	0,3	1484	0,24	964
ca032	U	0,02	4628	0,12	6694	0,11	6350	0,11	5756	0,14	7119
ca064	U	0,09	20611	0,81	26568	0,79	26953	0,73	24779	0,71	25130
ca128	U	0,42	91141	7,41	144840	5,77	111993	5,81	113453	7,4	138236
clus-1200-4800-...-003	S	380,17	601197	392,59	603506	532,56	763287	- -	- -	386,9	633071
clus-1200-4919-...-001	S	13	127945	34,52	231548	40,04	244409	530,93	787796	88,96	349893
clus-1200-4919-...-003	S	209,75	504095	26,75	209017	251,86	566146	170,62	441430	204,44	470170
cnt07	S	0,3	25103	1,43	34590	1,35	32123	1,5	32551	1,76	34032
cnt08	S	9,69	139736	28,98	215696	21,88	191570	24,42	203024	18,55	185133
dp03s03	S	0	59	0	87	0	44	0	41	0	35
dp03u02	U	0	25	0	24	0	18	0	20	0	13
dp04s04	S	0,01	198	0,02	189	0,01	127	0,01	118	0,01	133

Instance	S/U	zchaff		zchaff+SB_{HMC}		zchaff+SB_{HML}		zchaff+SB_{OMC}		zchaff+SB_{OML}	
		tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds
dp04u03	U	0,01	126	0,01	121	0,01	123	0,01	123	0,01	154
dp05s05	S	0,02	238	0,03	216	0,02	229	0,04	348	0,02	187
dp05u04	U	0,04	407	0,05	369	0,05	472	0,05	457	0,05	414
dp06s06	S	0,05	715	0,25	1870	0,18	1237	0,11	883	0,49	3081
dp06u05	U	0,19	1751	0,25	1655	0,29	1763	0,25	1562	0,21	1213
dp07s07	S	0,13	1052	0,19	1129	5,41	16571	0,14	930	0,84	4549
dp07u06	U	1,11	6990	1,6	6636	1,38	5532	1,29	5481	1,44	6113
dp08u07	U	33,16	80134	30,32	62577	46,53	79428	28,52	58346	25,91	57366
dp09s09	S	0,64	4670	2,21	5427	19,59	40603	72,41	113819	0,71	3024
dp10s10	S	377,69	449994	175,49	211362	206	225368	171,37	190503	32,92	58738
dp11s11	S	0,46	3943	243,42	215193	313,36	262402	39,13	52065	570,3	467945
dp12s12	S	- -	- -	- -	- -	- -	- -	333,46	274302	2416,93	1291718
example2_gr_rcs_w5	U	0,09	1866	0,3	2918	0,28	2546	0,13	1520	0,14	1507
ezfact256_1	-	476,99	4769049	- -	- -	- -	- -	- -	- -	- -	- -
f2clk_50	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
fifo8_100	U	74,12	151084	264,06	105492	258,47	108329	355,34	141962	344,99	136466
hanoi4	S	1,35	15708	3,36	26585	5,54	37972	5,99	37772	1,06	11486
hanoi5	S	- -	- -	913,19	859670	784,78	769299	626,06	757422	2915,62	1755395
hanoi6_fast	S	181,4	248892	1020,28	766762	110,5	182502	1462,21	825750	184,21	248646
hanoi6_on	S	360,95	456578	- -	- -	1589,42	969210	- -	- -	- -	- -
hgen2-v700-s504028057	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
hidden-k3-s2-r4-n700-03	S	- -	- -	- -	- -	- -	- -	- -	- -	1615,69	1383956
ip38	U	3010,57	1082808	2233,59	650405	3077,5	808930	- -	- -	- -	- -
ip50	U	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
k2fix_gr_2pinvar_w9	U	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
okgen-c1300-v650-11…	S	0	431	0	194	0	166	0	178	0	165
okgen-c1400-v700-21…	S	0	442	0	206	0	202	0	208	0,01	187
okgen-c1750-v500-74…	S	0	299	0,14	2544	0,01	231	0,01	297	0,01	345

Instance	S/U	zchaff		zchaff+SB_{HMC}		zchaff+SB_{HML}		zchaff+SB_{OMC}		zchaff+SB_{OML}	
		tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds
okgen-c2100-v600-19…	S	0,01	333	0	127	0,01	379	0,01	293	0,01	177
okgen-c2100-v600-53…	S	0	209	0,01	291	0,06	1220	0,02	412	0,02	328
rand_net40-25-1	U	0,25	2533	0,3	2720	0,38	2704	0,28	1985	0,26	2122
rand_net40-25-10	U	234,93	442508	389,84	617079	198,15	420511	349,92	539019	277,88	456810
rand_net40-25-5	U	198,08	344425	200,5	329849	40,99	132043	164,02	296932	62	169776
rand_net40-30-1	U	1,06	8973	1,7	8716	1,95	10464	1,68	8281	1,85	9821
rand_net40-30-10	U	58,9	177671	86,72	216328	104,62	267689	85,79	230904	87,27	222487
rand_net40-30-5	U	29,86	111978	41,26	119423	41,95	121841	60,97	152623	50,25	133206
rand_net40-40-10	U	137,57	311358	152,01	314143	151,21	297820	167,47	320576	137,8	285891
rand_net40-40-5	U	53,46	133375	93,78	175079	56,07	127578	78,85	158466	82,52	156776
rand_net40-60-10	U	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net50-25-1	U	1,83	16206	2,25	11957	2,56	14847	2,09	13840	2,54	13945
rand_net50-25-10	U	949,39	1114736	- -	- -	1190,92	1473959	1337,55	1685493	1585,44	1616574
rand_net50-25-5	U	1023,63	1003649	1217,02	1144136	1380,47	1135874	1832,24	1401118	1140,09	1123426
rand_net50-30-1	U	2,02	11297	3,1	13513	2,45	11336	2,44	9620	3,06	12775
rand_net50-30-5	U	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net50-40-5	U	925,42	971865	904,65	873508	1757,23	1262020	1931,45	1361625	1622,49	1165377
rand_net50-60-1	U	0,91	4306	0,85	2508	1,71	4865	1,33	3727	0,78	2451
rand_net50-60-5	U	3232,64	1561844	- -	- -	- -	- -	- -	- -	3099,29	1349859
rand_net60-25-1	U	9,37	53590	17,84	63317	27,44	81105	13,47	49608	12,54	49883
rand_net60-25-10	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net60-30-1	U	0,75	6194	0,87	4790	1,15	6219	1,24	6501	0,84	4196
rand_net60-30-10	U	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net60-30-5	U	1295,94	1249286	727,51	833383	1034,97	1085851	633,2	749447	1173,38	1124490
rand_net60-40-1	U	14,12	45403	7,54	23076	9,2	28213	8,31	24433	7,02	21481
rand_net60-40-5	U	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net60-60-1	U	10,72	38310	8,42	21223	5,36	13569	11,58	25282	12,82	27438
rand_net70-25-1	U	113,36	228410	77,98	163133	49,6	126285	72,73	151790	164,66	267168

Instance	S/U	zchaff		zchaff+SB_{HMC}		zchaff+SB_{HML}		zchaff+SB_{OMC}		zchaff+SB_{OML}	
		tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds	tps(s)	nœuds
rand_net70-25-5	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net70-30-1	U	4,34	22635	11,06	31227	8,01	25793	8,01	24073	13,55	35279
rand_net70-30-5	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net70-40-1	U	21,17	59838	32,91	62125	31,47	61951	24,97	51430	15,11	38029
rand_net70-40-5	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
rand_net70-60-1	U	6,32	18136	10,48	19723	12,93	23860	16,38	29544	9,85	19539
rand_net70-60-10	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
too_large_gr_rcs_w5	U	0,15	2467	0,23	2308	0,19	1596	0,2	1735	0,23	1937
too_large_gr_rcs_w6	U	0,18	3641	0,61	5821	0,62	6267	2,87	21577	0,92	9703
too_large_gr_rcs_w7	S	0,15	2019	0,25	1765	0,24	1478	0,24	1603	0,22	1371
too_large_gr_rcs_w8	S	0,14	790	0,18	640	0,22	1016	0,23	1222	0,21	894
too_large_gr_rcs_w9	S	0,15	994	0,23	919	0,24	883	0,22	854	0,22	782
unif-c1000-v500-s13…	S	0	328	0	151	0	142	0	154	0	144
unif-c1300-v650-s42…	S	0	431	0	204	0	177	0	190	0	179
unif-c1400-v700-s18…	S	0	459	0	208	0	195	0	203	0,01	196
unif-c2450-v700-s13…	S	0,02	755	0,01	302	0,03	649	0,05	895	0,05	995
unif-c2450-v700-s14…	S	0,01	219	0,1	2101	0,03	634	1,01	12742	0,05	1243
unif-c2600-v650-s20…	S	10,76	83141	274,48	468181	0,43	7053	250,03	450142	11,48	76866
unif-r3-v700-c2100-02	S	0	343	0,01	181	0,01	183	0,01	185	0,01	191
unif-r4-v700-c2800-03	-	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -
vda_gr_rcs_w7	U	24,04	101515	71,94	143310	50,62	120371	60,7	115218	49,14	111084
vda_gr_rcs_w8	S	204,18	384921	684,72	703527	745,87		764,88	753326	681,23	
vda_gr_rcs_w9	S	0,35	3316	0,8	3142	0,72	2755	0,78	3170	0,65	2235
w10_45	U	5,47	18765	17,13	21489	15,19	18780	17,37	21417	15,95	19408
w10_60	U	27,46	51535	122,52	82879	129,99	85195	135,06	86095	124,31	73015
w10_70	U	178,49	137310	282,19	145002	427,46	167610	277	126026	511,79	201524
w10_75	S	90,55	126213	323,09	174081	154,88	101746	324,86	181654	214,23	137091

TAB. 4.5 – ZChaff sur les instances issues des compétitions SAT précédentes

Nous avons ensuite expérimenté la résolution d'instances issues des précédentes compétitions SAT en exploitant les ensembles strong backdoor. Le tableau 4.5 indique les résultats que nous avons obtenus sur une partie de ces instances, en utilisant les ensembles strong backdoor calculés pour les deux classes de formules de Horn et de formules ordonnées à l'aide des deux méthodes utilisant les renommages orientés nombre de clauses ($*MC$) et nombres de littéraux ($*ML$). Enfin, la colonne S/U nous renseigne sur la satisfaisabilité éventuelle de l'instance (S pour satisfaisable et U pour non satisfaisable).

Ce que l'on peut en conclure, c'est que l'exploitation des ensembles strong backdoor en général semble améliorer soit le temps de résolution, soit la taille de l'espace de recherche (nombre de nœuds). Il semble néanmoins difficile de conclure que les ensembles d'une classe sont plus pertinents que ceux d'une autre. Nous constatons que cela dépend beaucoup de l'instance considérée, comme le montre les instances *rand_net*.

4.7 Ensembles strong backdoor pour les formules bien imbriquées

Nous montrons dans cette section pourquoi la taille des ensembles strong backdoor pour les formules bien imbriquées, notés ensembles strong *imbriqué*-backdoor par la suite, pour des problèmes 3-SAT générés aléatoirement au seuil de difficulté 4.25 sera au minimum égale à 60% de l'ensemble des variables du problème.

Il est précisé dans [Knuth, 1990] que le nombre maximum d'éléments (littéraux) présents dans un ensemble de formules bien imbriquées ne peut dépasser $2m+n$, avec n le nombre de variables booléennes et m le nombre de clauses. Ce qui signifie que dans un problème k-SAT, pour qu'un ensemble de formules soient potentiellement bien imbriquées selon un ordre donné, il faut s'assurer que : $km < 2m + n$, soit : $m < \frac{n}{k-2}$.

Pour un problème 3-SAT, cela signifie que $m < n$. Dans le cas de problèmes générés aléatoirement ceux-ci représentent des problèmes triviaux en général car les problèmes 3-SAT aléatoires difficiles ont un rapport entre m et n de l'ordre de 4.25. En d'autres termes, un problème 3-SAT aléatoire au seuil ne peut en aucun cas être un ensemble de clauses bien imbriquées.

Quel serait la taille d'un ensemble strong backdoor pour de tels problèmes ?

Le nombre total de littéraux d'une instance 3-SAT au seuil de difficulté 4.25 est de l'ordre de $4.25 \times 3 \times n$, soit $12.75n$. En admettant qu'elles soient équitablement répartis, il y aurait environ 6.35 occurrences positives et autant de négatives de chaque variable. En interprétant une variable (quelle que soit la valeur choisie), nous diminuerions de 6.35 le nombre de clauses. Ainsi, en supposant que cette quantité ne change pas au cours d'interprétations successives (nous considérons que nous supprimons toujours 6.35 clauses à chaque interprétation), nous pouvons dire qu'après x interprétations, il reste $n - x$ variables à instancier et $m - 6.35x$ clauses (avec $m = 4.25n$). L'ensemble des clauses ne pourront être bien imbriquées que lorsque nous aurons : $4.25n - 6.35x < n - x$, soit $x > \frac{3.25}{5.36} \equiv x > 0.6n$. Il faudra donc au minimum interpréter 60% des variables pour que la formule simplifiée ait une chance d'être une formule bien imbriquée, ce qui signifie qu'un ensemble strong *imbriqué*-backdoor contiendra au minimum 60% des variables du problème, d'autant que rien ne dit qu'une fois qu'il restera moins de clauses que de variables, la formule soit bien imbriquée pour un quelconque ordre.

Nous en déduisons donc que la classe des formules bien imbriquées n'est pas une bonne candidate pour le calcul d'ensemble strong backdoor sur des instances aléatoires. De plus, étant donné qu'il n'existe aucun moyen de trouver un ordre pour lequel une formule soit bien imbriquée, cette

classe n'est malheureusement pas exploitable en pratique.

4.8 Conclusions et perspectives

4.8.1 Conclusions

L'identification et l'exploitation des structures cachées dans un problème SAT sont des clés permettant de contrecarrer l'explosion combinatoire de sa résolution. Il arrive néanmoins que cette identification soit elle même difficile à réaliser de manière exacte. C'est le cas des ensembles strong backdoor, pour lesquels les méthodes exactes peinent à traiter des instances contenant plus de 50 variables.

Nous avons proposé dans ce chapitre une méthode pour calculer des ensembles strong $\mathcal{F}$-backdoor pour $\mathcal{F}$ dans $\{$ Horn, Horn-renommable, ordonnée et ordonné-renommable$\}$ basé sur une méthode de recherche locale permettant de traiter des instances contenant plusieurs milliers de variables. Cette méthode se décompose en deux étapes : le calcul d'un renommage permettant de minimiser soit le nombre de clauses non Horn (resp. non ordonnées) ou bien le nombre de littéraux positifs présents dans la partie non Horn (resp. non ordonnée), suivi du calcul de l'ensemble strong backdoor à partir de la formule renommée.

Nous avons constaté que le calcul du renommage en pré-traitement permettait de réduire significativement la taille des ensembles strong backdoor. De même, la minimisation du nombre de littéraux dans la partie non Horn de la formule renommée a permis également de réduire encore la taille des ensembles strong backdoor par rapport à la minimisation du nombre de clauses de la partie non Horn. Enfin, le passage aux formules ordonnées, extension naturelle des formules de Horn, nous a permis de réduire encore la taille des ensembles strong backdoor sur certaines instances réelles ou industrielles, tout en garantissant que dans le pire des cas, cette taille ne serait pas augmentée.

L'exploitation des ensembles strong backdoor que nous avons réalisé pour la résolution pratique des instances SAT nous a mené aux constatations suivantes. Tout d'abords d'un point de vue général, il paraît difficile de dire quel renommage, entre celui minimisant le nombre de clauses et celui minimisant le nombre de littéraux, mène à de meilleurs résultats. En effet, sur les instances réelles comme aléatoires, les performances des solveurs exploitant les ensembles strong backdoor sont équivalentes, bien que les ensembles strong backdoor soient plus petite avec le renommage minimisant le nombre de littéraux. Peut-être est-ce due au nombre de clauses non Horn (ou non ordonnées) qui est plus important pour ce renommage.

Les performances pour la résolution d'instances réelles ou industrielles sont assez mitigées. Nous avons constaté que sur certaines classes les gains en temps pouvaient être importants alors que sur d'autres classes on note même des pertes. Néanmoins de manière générale, le nombre de nœuds parcourus est inférieur lorsque les ensembles strong backdoor sont utilisés.

Concernant les instances aléatoires, la taille moyenne des ensembles strong backdoor est légèrement inférieure à 50% de l'ensemble des variables des instances. Les performances en terme de temps, aussi bien en temps qu'en nombre de nœuds, sont nettement meilleures sur ces instances. En effet, nous avons pu constater un gain de plus de 20% pour la résolution des instances à 300 variables.

La méthode de calcul d'ensembles strong backdoor que nous avons proposé s'avère donc efficace et capable de traiter des instances de grandes tailles. L'exploitation que nous avons fait des ensembles strong backdoor lors de la résolution s'avère quant à elle intéressante surtout sur

les instances aléatoires où des gains substantiels ont été réalisés.

4.8.2 Perspectives

À l'heure actuelle, nous envisageons deux suites possibles à donner à ces travaux.

Extension à d'autres classes polynomiales

Le problème de calcul d'ensembles strong backdoor de taille minimale est un problème difficile. L'utilisation de méthodes de recherche locale pour ce calcul nous a permis de le rendre praticable pour un certain nombre de classes polynomiales.

Le première perspective que nous envisageons pour ces travaux concerne le calcul d'ensembles strong backdoor pour d'autres classes polynomiales. Nous pensons en particulier aux formules presque Horn ou presque ordonnées. L'idée consiste à calculer le meilleur Horn renommage d'une formule à l'aide de WalkHorn, puis de relancer le processus sur la partie non Horn et ainsi de suite jusqu'à l'obtention d'un point fixe. Ceci revient à calculer le reste itéré de la formule [Luquet, 2000]. Il resterait ensuite à calculer un ensemble strong backdoor sur ce reste itéré et à trouver un moyen d'obtenir un ensemble strong backdoor de la formule d'origine en partant de celui-ci.

Les ensembles strong backdoor et le problème Max-SAT

La seconde perspectives que nous envisageons consiste à étendre le concept d'ensembles strong backdoor au problème d'optimisation Max-SAT. Cependant, elle se heurte à un problème de taille : il n'existe pas de classe polynomiale connue pour le problème Max-SAT à l'heure actuelle.

En effet, toutes les classes polynomiales connues pour le problème de décision de SAT ne le sont plus lorsque l'on s'intéresse au problème d'optimisation Max-SAT. C'est le cas notamment pour les clauses binaires et les clauses de Horn qui sont la base de la quasi-totalité des classes référencées [Creignou et al., 2001]. Cependant, bien que non polynomiale dans le pire des cas, le problème Max-SAT pour un ensemble de clauses de Horn devrait pouvoir être traité en exploitant les propriétés de ces formules, en l'occurrence la constatation qu'un ensemble de clauses de Horn qui ne contient pas de clause unitaire positive est satisfaisable.

Voici comment exploiter cette propriété : comme pour le problème SAT où nous utilisons un algorithme de type DPLL en le forçant à choisir ses variables de branchements dans un ensemble strong Horn-backdoor B, nous utiliserions un algorithme classique de type *Branch & Bound* que nous forcerions à brancher sur les variables de l'ensemble strong backdoor. Une fois toutes les variables de B interprétées, si il n'y a pas de clause unitaire positive, alors la borne supérieure comptant le nombre minimum de clauses falsifiées peut être abaisser à la valeur courante du nombre de clauses falsifiées car le reste de la formule peut être satisfaite. Cela éviterait d'énumérer sur les variables restantes. Dans le cas où il y aurait des clauses unitaires positives, le choix des variables de branchements seraient orienté vers les variables apparaissant dans ces clauses unitaires positives jusqu'à ce qu'il n'y en ai plus. Quand il n'y aura plus de clauses unitaires positives, le même traitement sera appliqué sur le borne supérieure.

Force est de constater que dans le pire des cas, il y aura des clauses unitaires positives jusqu'au bout de l'énumération, ce qui nous empêche d'avancer une complexité en $\mathcal{O}(2^{|B|})$ pour cette méthode qui reste en $\mathcal{O}(2^n)$. L'intégration de ce traitement particulier dans un solveur Max-SAT

performant devrait permettre de déclencher des retour-arrières plus tôt et à moindre coût. Un tel solveur est en cours de développement, il est basé sur le solveur *maxsatz* [Li et al.,].

Cette méthode pourrait également exploiter les ensembles strong ordonné-backdoor.

Chapitre 5

Voisinage consistent pour le problème SAT

Sommaire

5.1 Introduction

La recherche locale est un des paradigmes fondamentaux pour la résolution de problème à forte combinatoire comme le problème SAT. Elle constitue la base de certaines méthodes de résolution les plus efficaces pour des problèmes de grande taille et difficile rencontrés dans beaucoup d'applications réelles. En dépit des avancées considérables des méthodes complètes, les méthodes de recherche locales représentent la seule issue praticable pour résoudre ces instances de taille importante et complexes.

La plupart des méthodes de recherche locale sont basées sur une exploration partielle de l'espace de recherche [Selman et al., 1992b, Hoos et Stützle, 2004]. Plus précisément, elles partent

d'une interprétation complète (de toutes les variables du problème) inconsistante générée aléatoirement ou à l'aide d'une heuristique, et tentent de la réparer en effectuant des changements mineurs sur cette interprétation candidate. Typiquement, ces changements sont opérés en inversant la valeur de vérité de certaines variables jusqu'à ce que l'interprétation devienne un modèle de la formule CNF. Le choix de la variable dont la valeur de vérité est changée peut être réalisé selon plusieurs heuristiques, généralement dépendantes du nombre de clauses satisfaites.

Une des améliorations les plus importantes pour ces méthodes consiste à intégrer une stratégie de « marche aléatoire » pendant la résolution, produisant l'algorithme *WalkSat* et ses variantes : Novelty, Novelty+, R-Novelty and R-Novelty+ [Selman et al., 1994] [McAllester et al., 1997]

[Hoos, 1999] [Hoos et Stützle, 2000]. Ces méthodes contrôlent la marche aléatoire en modifiant une valeur p (paramètre de bruit), permettant d'introduire une phase de diversification à une phase d'amélioration stricte de la valeur de la fonction objectif (méthode de descente). Plus récemment, dans [Li et Huang, 2005], un nouveau mécanisme de diversification a été proposé, permettant de faire des choix déterministes aux lieux d'aléatoires pour élargir l'espace de recherche parcouru, assurant systématiquement une amélioration de la solution courante. Les méthodes de recherche locale intégrant une phase de diversification aléatoire sont appelées *méthodes de recherche locale stochastiques*.

Les dernières méthodes de résolution du problème SAT offrent un large panel de méthodes de recherche locale. Par exemple, Mazure *et al.* ont introduit un algorithme basic de recherche taboue pour SAT, TSAT [Mazure et al., 1997], où une longueur optimale de la liste taboue est mise en évidence pour les instances K-SAT aléatoires difficiles.

Hirsch et Kojevnikov [Hirsch et Kojevnikov, 2005] ont développé un solveur SAT efficace, UnitWalk, en combinant une recherche locale et un procédé d'élimination des clauses unitaires.

Lardeux *et al.* [Lardeux et al., 2005] ont introduit une logique tri-valuée dans laquelle la valeur *indéterminée* est ajoutée. Ce formalisme a permis de concevoir un algorithme de recherche locale manipulant à la fois des interprétations partielles et complètes, et d'introduire de nouveaux mécanismes de diversification et d'intensification.

Dans [Smyth et al., 2003], Smyth *et al.* ont introduit un nouvel algorithme de recherche locale stochastique, Iterated Robust Tabu Search (IRoTS) pour le problème MAX-SAT (la version d'optimisation du problème SAT), qui combine deux méthodes de recherche locale , Iterated Local Search (ILS) [Ramalhinho-Lourenço et al., 2000] et Robust Tabu Search (RoTS) [Taillard, 1991].

Enfin, dans [Pham et al., 2007], un moyen original d'exploiter la structure des problèmes SAT, en particulier les chaînes d'équivalences, généralement intégré au solveurs complets, a été introduit dans les solveurs incomplets. Cette nouvelle méthode s'avère très efficace sur les instances structurées.

Tous ces algorithmes, excepté celui utilisant la logique tri-valuée [Lardeux et al., 2005], autorisent la falsification de certaines clauses en visitant des interprétations inconsistantes et complètes.

Dans ce chapitre, nous proposons un nouvel algorithme de recherche locale pour SAT appelé $CN\text{-}SAT$ qui ne visite que des interprétations partielles consistantes. Au lieu de générer une interprétation complète et inconsistante, il essaie de construire un modèle en affectant les variables une à une à la manière d'une méthode complète. Chaque fois qu'un conflit survient, alors qu'une méthode complète aurait déclenché un retour-arrière (backtrack), notre méthode libère un ensemble minimum de variables impliquées dans les clauses falsifiées pour restaurer la consistance. Ainsi, pour une interprétation partielle donnée, ce mécanisme ne visite que le *voisinage consistant* de l'interprétation courante. Cette idée a été initiée dans le cadre des CPS binaires [Vasquez et al., 2005].

Pour s'assurer que notre méthode n'instancie et ne libère pas toujours les même variables, nous couplons notre algorithme avec une liste taboue [Glover et Laguna, 1997] pour laquelle nous verrons expérimentalement que la durée du statut tabou (*tabu tenure* en anglais) peut être empiriquement fixée. Ces expérimentations montreront aussi que cette méthode peut être efficace sur plusieurs classes d'instances, et résout même des instance que R-Novelty+ ne résout pas.

Ces travaux ont donné lieux aux publications suivantes : [Paris et al., 2007a], [Habet et al., 2007].

5.2 Préliminaires

Consistance locale

Pour une CNF Σ, une interprétation partielle d'un ensemble de variables $\mathcal{S} \subset \mathcal{V}(\Sigma)$ est localement consistante si et seulement si aucune clause impliquant les variables de $\mathcal{S}$ n'est falsifiée. De plus, une interprétation partielle qui ne peut pas appartenir à un modèle est un *nogood*.

Recherche taboue

Une recherche taboue (RT) est une méta-heuristique conçue pour aborder les problèmes d'optimisation combinatoires [Glover et Laguna, 1997]. Contrairement aux approches purement aléatoires, RT est basée sur la croyance qu'une recherche intelligente devrait inclure une forme plus systématique d'exploration, basée sur une mémoire et un apprentissage adaptatifs. RT peut être décrite comme une forme de recherche dans un voisinage avec un ensemble de composants critiques et complémentaires.

Pour un problème d'optimisation (S, f), caractérisé par un espace de recherche S (composé des interprétations possibles) et une fonction objectif f, un voisinage $\mathcal{N}$ est introduit, qui associe à chaque interprétation s de S un sous-ensemble non vide $\mathcal{N}(s)$ de S. Un algorithme RT typique commence avec une interprétation initiale s dans S, puis visite successivement une série de « meilleures » interprétations locales en suivant la fonction de voisinage. À chaque itération, un des meilleurs voisins $s' \in S$ est sélectionné pour devenir l'interprétation courante, même si s' n'améliore pas l'interprétation courante du point de vue de la fonction objectif.

Pour éviter l'apparition de cycle lors du parcours de l'espace de recherche, et permettre à la recherche de quitter des optimums locaux, une liste taboue est introduite. Ceci ajoute une composante mémoire à la méthode. En effet, une liste taboue maintient un historique sélectionné H, composé des interprétations visitées précédemment, ou plus généralement, certains attributs pertinents de ces interprétations. Une stratégie simple consiste à empêcher les interprétations contenues dans H d'être considérées à nouveau pour les k prochaines itérations (k est appelé la *durée taboue*). Cette durée peut varier en fonction des différents critères et dépend généralement du problème considéré. À chaque itération, RT cherche le meilleur voisin de s dans $\mathcal{N}(H, s)$ maintenu à jour dynamiquement, au lieu de $\mathcal{N}(s)$ lui-même.

Quand les attributs des interprétations sont enregistrés dans la liste taboue au lieu des interprétations elles-mêmes, la liste taboue peut empêcher certaines interprétations non visitées, mais néanmoins intéressantes, d'être considérées. Des critères d'aspiration peuvent être utilisés pour palier ce problème. Le critère d'aspiration le plus utilisé consiste à annuler le statut tabou d'un mouvement qui aboutit à une interprétation meilleure que toutes celles obtenues jusqu'à lors.

5.3 Composants de CN-SAT

Nous allons décrire dans cette section la méthode que nous proposons (CN-SAT) pas à pas. Nous commençons par les définitions d'interprétations partielles et de la notion de voisinage consistant, puis nous détaillons les différents composants de l'algorithme que nous proposons.

5.3.1 Interprétation partielles

Dans la plupart des algorithmes de recherche locale (RL) pour SAT, une interprétation est une interprétation de toutes les variables du problème. L'espace de recherche correspond à l'ensemble de toutes les interprétations qui sont soit des modèles, soit des interprétations qui mènent à une contradiction. Dans le cas d'une instance SAT à n variables, une interprétation peut être représentée par un vecteur de dimension n ; $s = (v_1,\ v_2,\ ...,\ v_n)$ tel que $\forall i = 1 \cdots n$, $v_i \in \{0,\ 1\}$ est la valeur de vérité de la variable x_i.

Comme nous l'avons annoncé dans la section 1, dans notre approche l'espace de recherche est constitué d'*interprétations partielles* notées $s_i = (v_{i_1}, v_{i_2}, ..., v_{i_{n_i}})$ où n_i variables sont instanciées ($n_i = |s_i| \leq n$). Dans ces interprétations, les variables non encore instanciées sont affectées à la valeur *indéterminée* u.

5.3.2 Voisinage consistant

Les algorithmes de recherche locale classiques remplacent une interprétation s par une nouvelle interprétation obtenue en opérant un mouvement, noté $mv(x_i,\ v_i)$, qui modifie la valeur courante de la variable x_i de $s[x_i]$ à v_i. Dans notre approche, nous définissions un mouvement comme suit : tout d'abord, les mouvements ne s'appliquent qu'aux variables libres (instanciées à la valeur u). Ensuite, ils sont suivis par une restauration de la consistance qui libère au moins une variable par clause falsifiée. L'ensemble des interprétations ainsi atteintes représente le voisinage consistant de s.

De cette manière, seules les interprétations localement consistantes sont évaluées dans $\mathcal{N}$, interprétations qui peuvent être distantes les unes des autres par plus d'une affectation. Pour rester concis, nous conserverons la notation $mv(x_i,\ v_i)$ pour la nouvelle définition d'un mouvement, mais un tel mouvement consiste en un ancien mouvement, dit élémentaire, $mv(x_i, v_i)$ suivi éventuellement d'un certain nombre de mouvements de la forme $mv(x_j, u)$ nécessaires à la restauration de la consistance.

5.3.3 Évaluation du voisinage

Les méthodes de recherche locale comme WalkSat partent d'une interprétation générée aléatoirement, qui est complète et inconsistante, et tentent de la réparer jusqu'à satisfaire toutes les clauses. En conséquence, la fonction objectif est celle qui maximise le nombre de clauses satisfaites, et le passage d'une interprétation à une autre est guidé par la valeur de cette fonction.

Dans l'approche CN-SAT, le critère d'évaluation d'une interprétation est *le nombre de variables instanciées* dans s. Ainsi, la fonction objectif à maximiser est $f = |s|$. Cependant, plutôt que d'évaluer un mouvement en comptant le nombre de variables instanciées après l'avoir effectué, nous calculons le nombre de variables libérées si ce mouvement est effectué. En effet, la difficulté réside dans le fait de libérer le moins de variables possible à chaque mouvement, ainsi, une fois les variables à libérer connues, avec leur nombre, nous pouvons immédiatement savoir

combien de variables seront instanciées au total après ce mouvement. Pour ce faire, une fonction δ est calculée.

Algorithme 9 $evaluate\_\mathcal{N}$

Fonction $evaluate\_\mathcal{N}$
Entrée : Une interprétation s et une formule CNF Σ
Sortie : La liste des meilleurs mouvement possible pour améliorer s

```
1: δ_min^¬tabu ← nbVar(Σ) ;
2: L_cand ← ∅ ;
3: pour tout x_i ∈ s, tel que x_i = u faire
4:    pour chaque k ∈ {0, 1} faire
5:       si (x_i, k) n'est pas tabou alors
6:          si δ(x_i, k) < δ_min^¬tabu alors
7:             δ_min^¬tabu ← δ(x_i, k) ;
8:             L_cand ← (x_i, k) ;
9:          sinon
10:            si δ(x_i, k) = δ_min^¬tabu alors
11:               L_cand ← L_cand ∪ {(x_i, k)} ;
12:            fin si
13:         fin si
14:      fin si
15:   fin pour
16: fin pour
17: retourner L_cand
```

Plus précisément, si s' est l'interprétation obtenue à partir de s en appliquant le mouvement $mv(x_i, k)$ alors $\delta(x_i, k)$ retourne le nombre de variables déjà instanciées qui seront libérées (*i.e.* affectées à u) si x_i est affectée à k, $k \in \{0,\ 1\}$. L'évaluation du voisinage d'une interprétation est réalisée par l'algorithme 9, où $\delta_{min}^{\neg tabu}$ est la valeur minimale de δ biaisée par le statut tabou de certaines variables *i.e.* seuls les mouvements non tabous sont considérés.

5.3.4 Calcul de δ

Exemple

Rappelons que la valeur δ correspond au nombre de variables qui doivent être libérées (affectées à u) pour un mouvement donné.

Par exemple, considérons l'ensemble des clauses suivant :

$(c_1) : \neg x_1 \vee x_2 \vee \neg x_4 \vee \neg x_6$ $\quad (c_2) : \neg x_1 \vee \neg x_5$
$(c_3) : x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_7$ $\quad (c_4) : x_2 \vee \neg x_3 \vee \neg x_5$
$(c_5) : \neg x_6 \vee \neg x_7 \vee x_4$ $\quad (c_6) : \neg x_8 \vee x_4$
$(c_7) : \neg x_6 \vee \neg x_7 \vee x_5$ $\quad (c_8) : \neg x_8 \vee x_5$

et l'interprétation s suivante : $s = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (1, 0, 1, u, u, 1, 1, 1)$, où les variables x_4 et x_5 sont libres.

Avant d'évaluer les différents mouvements, donnons la table de vérité du $\vee$ pour la logique tri-valuée $\{0, 1, u\}$:

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1
0	u	u
u	0	u
1	u	1
u	1	1
u	u	u

TAB. 5.1 – Table de vérité du $\vee$ pour la logique tri-valuée.

Au regard de ce tableau, toutes les clauses de notre exemple prennent la valeur u pour l'interprétation s, nous devons donc calculer δ pour les affectations des variables x_4 et x_5 à 1 et 0 :

- $\delta(x_4, 1) = 1$ cela signifie que si l'on affectait la valeur $vrai$ à x_4 nous devrions libérer *une* variable pour restaurer la consistance : ici x_2. En fait, affecter x_4 à $vrai$ falsifierait les clauses (c_1) et (c_3). La libération de x_2 supprime cette falsification.
- $\delta(x_4, 0) = 2$: si x_4 était affectée à $faux$ nous devrions libérer *deux* variables : x_6 et x_8 sinon les clauses (c_5) et (c_6) seraient falsifiées.
- $\delta(x_5, 1) = 2$, x_1 et x_2 devraient être libérées sinon les clauses (c_2) et (c_4) seraient falsifiées.
- $\delta(x_5, 0) = 2$, x_6 et x_8 devraient être libérées pour éviter de falsifier les clauses (c_7) et (c_8).

Après cette évaluation, le mouvement sélectionné pour l'itération courante devrait minimiser la valeur de δ. Ainsi, $mv(x_4, 1)$ serait sélectionné (où x_4 est affecté à $vrai$ et x_2 à u). Notons que cet exemple ne prend pas en compte le statut tabou des différents mouvements.

Algorithme d'évaluation approchée de δ

Calculer l'ensemble minimal des variables à libérer après une affectation est équivalent au problème du calcul d'un transversal minimal (minimal Hitting-Set) d'un hypergraphe, qui est NP-difficile [Kavvadias et Stavropoulos, 2005].

Reprenons l'exemple précédent, affecter x_4 à $vrai$ falsifie les clauses (c_1) et (c_3) et satisfait (c_5) et (c_6). Pour maintenir la consistance, nous devons sélectionner un nombre de variables apparaissant dans (c_1) et (c_3) (excepté x_4 pour ne pas boucler).

Si on réduit (c_1) et (c_3) en supprimant x_4, on obtient les sous-clauses $(c'_1) : \neg x_1 \vee x_2 \vee \neg x_6$ et $(c'_3) : x_2 \vee \neg x_3 \vee \neg x_7$ qui sont falsifiées. On peut les représenter par un hypergraphe $G = (V, E)$ où $V = \{x_1, x_2, x_3, x_6, x_7\}$ est l'ensemble des sommets et $E = \{(x_1, x_2, x_6), (x_2, x_3, x_7)\}$ est l'ensemble des hyper-arètes *i.e.* chaque hyper-arète représente une clause. Le transversal minimal de l'hypergraphe G est l'ensemble $\{x_2\}$. Pour approximer cet ensemble minimal, nous utilisons un algorithme glouton simple basé sur le degré des sommets. Cet algorithme choisit successivement le sommet ayant le degré le plus fort dans les hyper-arètes non traitées, puis le retire et marque les hyper-arètes le contenant comme étant traitées. Ce procédé est répété jusqu'à ce que toutes les hyper-arètes sont diminuées d'au moins un sommet. L'ensemble des variables associées aux sommets qui ont été retirés constitue notre ensemble minimal de variables à libérer. L'algorithme 10 décris cette méthode.

Algorithme 10 Approx Min Hitting-Set

Fonction ApproxMinHittingSet

Entrée : x_i : une variable, k : une valeur booléenne, Σ : une formule CNF, s : une interprétation partielle (modèle en cours de construction)

Sortie : un ensemble de variables dont la taille est la plus petite possible à libérer pour restaurer la satisfaisabilité de Σ, si on étend le modèle partiel s avec $x_i \leftarrow k$.

1: $MinTrans \leftarrow \emptyset$; {le futur transversal minimal}
2: $SC \leftarrow$ toutes les clauses falsifiées par l'affectation de x_i à k ;
3: retirer x_i (ou $\neg x_i$ en fonction de la valeur de k) de chaque clauses de SC ;
4: entier $fixed[|SC|]$; {un tableau de $|SC|$ entiers}
5: **pour** chaque clause $c \in SC$ **faire**
6: $\quad fixed[c] \leftarrow 0$;
7: **fin pour**
8: **pour** chaque clause $c \in SC$ telle que pour chaque clause $c' \in SC$ ($c \neq c'$), $c \cap c' = \emptyset$ **faire**
9: $\quad MinTrans \leftarrow MinTrans \cup choose\_var\_in\_clause(c)$; {Sélection de la variable à libérer pour éviter l'inconsistance}
10: $\quad fixed[c] \leftarrow 1$; {la clause c n'est plus falsifiée}
11: **fin pour**
12: **tant que** $\exists c \in$ tel que $fixed[c] = 0$ **faire**
13: $\quad$ Soit $l \in SC$ tel que $\forall l' \in SC, l \neq l', Occ(l) \geq Occ(l')$ {$Occ(l)$ désigne le nombre de clauses de SC dans lesquelles l apparaît}
14: $\quad$ **pour** chaque clause $c \in SC$ telle que $l \in c$ **faire**
15: $\quad\quad fixed[c] \leftarrow 1$;
16: $\quad$ **fin pour**
17: $\quad MinTrans \leftarrow MinTrans \cup l$ {Plus précisément la variable associée à l}
18: **fin tant que**
19: **retourner** $MinTrans$

Algorithme d'évaluation exacte de δ

Parallèlement à l'algorithme glouton, nous avons développé un algorithme de calcul exact pour δ décrit par l'algorithme 11. Il commence par initialiser un certain nombre de structures (lignes 1 à 9), puis traite les clauses indépendantes[1] en choisissant arbitrairement, pour chacune d'elles, une de leur variable à libérer (fonction $choose\_var\_in\_clause(c)$). Ensuite un algorithme simple de Branch & Bound est appelé (cf. algorithme 12) pour finir de produire l'ensemble minimal de variables à libérer (δ). Pendant sa construction, cette ensemble est appelé HS dans les algorithmes 11 et 12.

5.3.5 Gestion de la liste taboue

Nous avons décrit une méthode utilisant des mouvements multi-attributs (impliquant plusieurs variables), pouvant ainsi produire des cycles dans l'espace de recherche. Pour éviter de tels phénomènes nous introduisons une liste taboue de mouvements interdits. Avant de détailler

[1] Nous considérons indépendantes les clauses dont les littéraux qui les composent n'apparaissent dans aucune autre clause.

Algorithme 11 Exact Min Hitting-Set

Fonction ExactMinHittingSet

Entrée : x_i : une variable, k : une valeur booléenne, Σ : une formule CNF, s : une interprétation partielle (modèle en cours de construction)

Sortie : un ensemble de taille minimale de variables à libérer pour restaurer la satisfaisabilité de Σ, si on étend le modèle partiel s avec $x_i \leftarrow k$.

1: $SC \leftarrow$ toutes les clauses falsifiées par l'affectation de x_i à k ;
2: retirer x_i (ou $\neg x_i$ en fonction de la valeur de k) de chaque clauses de SC ;
3: $SL \leftarrow \emptyset$; {un ensemble vide de littéraux}
4: $MinCover \leftarrow \emptyset$; {le futur transversal minimal}
5: $B \leftarrow |SC| + 1$; {la taille du transversal minimal dans le pire des cas}
6: entier $fixed[|SC|]$; {un tableau de $|SC|$ entiers}
7: **pour** chaque clause $c \in SC$ **faire**
8: $\quad fixed[c] \leftarrow B$;
9: **fin pour**
10: **pour** chaque clause $c \in SC$ telle que pour chaque clause $c' \in SC$ ($c \neq c'$), $c \cap c' = \emptyset$ **faire**
11: $\quad SL \leftarrow SL \cup choose\_var\_in\_clause(c)$; {Sélection de la variable à libérer pour éviter l'inconsistance}
12: $\quad fixed[c] \leftarrow -1$; {la clause c n'est plus falsifiée}
13: **fin pour**
14: $c \leftarrow$ première clause de SC telle que $fixed[c] \neq -1$;
15: **pour** chaque littéral l de c **faire**
16: $\quad$ MinHittingSet($SL \cup l, 0, B, l, SC, fixed, HS$) ;
17: **fin pour**
18: $MinCover \leftarrow HS$;
19: **retourner** $MinCover$

les spécificités de cette liste, rappelons d'abord qu'un mouvement $mv(x_i, v_i)$ consiste en l'affectation de v_i à x_i, puis la libération de r variables $x_{j_1}, ..., x_{j_r}$ pour pallier la falsification de certaines clauses. Si un tel mouvement est appliqué alors les mouvements $mv(x_{j_1}, v_{j_1}), ..., mv(x_{j_r}, v_{j_r})$ seront considérés tabous durant un certain nombre d'itérations pour éviter de libérer x_i trop tôt.

Par ce procédé, nous rendons tabou toutes les valeurs de variables libérées qui sont en conflit avec la nouvelle valeur de x_i. Nous définissons dynamiquement la durée de l'interdiction d'une affectation de la variable x_k à la valeur v_k en fonction du nombre de fois où x_k a été affectée à v_k depuis le début de la recherche. Ce nombre que nous notons $freq(x_k, v_k)$ correspond à la fréquence du mouvement $mv(x_k, v_k)$. À chaque itération $iter$, nous fixons le statut tabou pour chaque couple (variable,valeur) des variables libérées à : $tabou(x_j, k) = iter + \alpha \times freq(x_j, k)$, où k est la valeur à laquelle la variable x_j était affectée avant de la libérer et α est un coefficient utilisé pour paramétrer la longueur de la liste taboue. Ainsi, le statut tabou du couple (x_j, v_j) est proportionnel à la fréquence à laquelle x_j a été affectée à v_j depuis le début de la recherche et l'empêche donc d'y être affecté trop souvent. Par ailleurs, le statut tabou d'un mouvement $mv(x_i, v_i)$ tel que s' = s+$mv(x_i, v_i)$ est annulé si $|s'| > |s^*|$ (où s^* est l'interprétation la plus complète obtenue depuis le début de la recherche). Ceci correspond au critère *d'aspiration*. Dans la fonction $evaluate\_\mathcal{N}$, la condition *n'est pas tabou* correspond à $(tabu(x_i, k) \leq iter \vee |s'| > |s^*|)$.

Algorithme 12 Min Hitting-Set

Procédure MinHittingSet

Entrée : SL : un ensemble de littéraux, $depth$, B : entier, lit : un littéral, SC : un ensemble de clauses, $fixed[|SC|]$: entier, HS : un ensemble de littéraux.

Sortie : HS : un des plus petits transversaux minimaux de SC, et B : sa taille.

1: **si** $|SL| <$ B **alors**
2: **pour** chaque clause c de CS telle que $lit \in c$ et $fixed[c] \geq depth$ **faire**
3: $fixed[c] \leftarrow depth$;
4: **fin pour**
5: **si** il n'y a plus de clause falsifiée **alors**
6: $B \leftarrow |SL|$;
7: $HS \leftarrow SL$;
8: **sinon**
9: $c \leftarrow$ première clause de SC telle que $fixed[c] \geq depth$;
10: **pour** chaque littéral l de c **faire**
11: HittingSet$(SL \cup l,\ depth + 1,\ B,\ l,\ SC, fixed,\ HS)$;
12: **fin pour**
13: **fin si**
14: **fin si**

5.3.6 Diversification et intensification

Le but de la diversification est d'aider la recherche à quitter les minima locaux de la fonction objectif. Dans ce but, nous introduisons un procédé de diversification simple, mais souvent efficace, qui consiste à redémarrer la recherche avec une nouvelle interprétation partielle et consistante différente de la précédente. Cette nouvelle tentative s'accompagne toujours d'une initialisation de la liste taboue à vide. Plus précisément, nous autorisons un nombre fixé de tentatives, et à chacune d'elles, la recherche taboue explore l'espace de recherche d'un point initial (interprétation) différent des précédents essais.

Par opposition à la diversification, l'intensification tente de concentrer la recherche dans des zones prometteuses de l'espace de recherche. Pour atteindre ce but, nous vidons la liste taboue, augmentons le coefficient α (pour augmenter la durée de restriction des mouvements tabous), et nous relançons la recherche à partir de la meilleur solution trouvée s^*. La phase d'intensification est appelée sous deux conditions : tous les mouvements candidats sont tabous ($\mathcal{L}_{cand} = \emptyset$), ou un certain nombre d'itérations préalablement fixé est atteint.

5.3.7 Voisinage consistant pour SAT

La combinaison de tous les points, décrits dans les précédentes sections, nous amène à un algorithme de recherche taboue sur un voisinage consistant pour SAT (CN-SAT) :

L'algorithme $greedy()$ tente d'affecter une variable libre tant qu'aucune clause n'est falsifiée. Sa caractéristique principale est de produire une interprétation partielle différente à chaque appel. Cela permet une certaine diversification de la recherche comme expliqué précédemment. À chaque tentative, nous générons ainsi la première interprétation partielle consistante, et effectuons un certain nombre de mouvements jusqu'à ce qu'une interprétation complète consistante soit trouvée, ou bien jusqu'à atteindre un nombre fixé de mouvements, ou encore jusqu'à ce qu'aucun mouvement

Algorithme 13 $CN\text{-}SAT$

Procédure $CN\text{-}SAT$

Entrée : Un formule CNF Σ et deux entier max_tries et max_moves

Sortie : l'interprétation maximisant le nombre de variables interprétées de manière consistante

```
1: date ← 0 ;
2: pour try = 1 à max_tries faire
3:    s ← greedy(Σ) ; s* ← s ;
4:    pour move = 1 à max_moves faire
5:       L_cand ← evaluate_N(s, Σ) ;
6:       si L_cand ≠ ∅ alors
7:          date + + ;
8:          (x_i, k) ← Select(L_cand) ;
9:          propagate_move(x_i, k, Σ) ;
10:         freq(x_i, k) + + ;
11:         si |s| > |s*| alors
12:            s* ← s ;
13:            si |s*| = nbVar(Σ) alors
14:               retourner s*
15:            fin si
16:         fin si
17:      sinon
18:         Intensify(s*, Σ) ;
19:         si |s*| = nbVar(Σ) alors
20:            retourner s*
21:         fin si
22:         move ← max_moves
23:      fin si
24:   fin pour
25: fin pour
26: retourner s*
```

ne soit candidat. Dans ce dernier cas, la phase d'intensification ($Intensify(s^*)$) est lancée. La fonction $Select(\mathcal{L}_{cand})$ choisit aléatoirement un mouvement candidat dans $\mathcal{L}_{cand}$ comme décrit dans les sections 5.3.3 et 5.3.4. La fonction $propagate\_move(x_i, v_i)$ affecte x_i à v_i et libère les variables pour réparer les conflits éventuels, tout en réalisant la propagation unitaire.

5.4 Expérimentation

Pour évaluer les performances de $CN\text{-}SAT$, nous avons réalisé des expérimentations sur des instances 3-SAT aléatoires difficiles[2] et sur des instances structurées. Nous le comparons avec l'algorithme de recherche locale R-Novelty+, considéré comme l'un des solveurs incomplets les plus efficaces pour SAT. $CN\text{-}SAT$ est écrit en C, compilé sous linux et testé sur un ordinateur P_{IV} cadencé à 3.0 Ghz avec 1 Go de RAM. Les paramètres utilisés pour $CN\text{-}SAT$ et R-Novelty+ sont : *max_mouvements* = 10^6 par tentative et le temps limite d'exécution est fixé à 300 secondes pour les deux méthodes. Le paramètre de bruit de R-Novelty+ est fixé à sa valeur par défaut. Le paramètre α utilisé dans le calcul de la durée taboue d'un mouvement est fixé à 1.9 [3] pendant la phase de RT, puis à $2,8$ durant les phases d'intensification. Nous avons codé les deux méthodes de calcul de δ (approchée et exacte), ce qui nous amène à $CN\text{-}SAT$ Approx et $CN\text{-}SAT$ Exact dans les tableaux suivants.

5.4.1 Instances structurées

Nous avons choisi des instances SAT structurées et satisfaisables disponibles sur le site `http://www.satlib.org`. Dans le tableau 5.2, les caractères « - - » signifient que l'instance n'a pas été résolue.

Sur les instances *latin square* (notées qg*), $CN\text{-}SAT$ Exact est plus efficace que R-Novelty+ et $CN\text{-}SAT$ Approx. Sur les instances de planification (notées bw_large*), R-Novelty+ est capable de résoudre une instance de plus, mais les performances des trois algorithmes sont comparables. Sur les instances des séries « All intervall » (notées ais*), R-Novelty+ semble être meilleur. Finalement, sur les instances parity (notées parity*), on peut remarquer que sur les petites instances (parity8*), $CN\text{-}SAT$ Exact semble le plus efficace, ce qui se confirme sur les instances de plus grande taille (parity16*), car seul la version $CN\text{-}SAT$ Exact arrive à en résoudre certaines. Pour conclure, on peut dire que $CN\text{-}SAT$ Exact est plus performant que $CN\text{-}SAT$ Approx sur les instances structurées et que sur certaines de ces instances, $CN\text{-}SAT$ Exact domine aussi R-Novelty+.

5.4.2 Instances 3-SAT aléatoires

Nous avons généré aléatoirement 8 classes d'instances aléatoires et chacune d'elles est composée de 50 instances difficiles et satisfaisables. Le nombre des variables (qui définit les 8 classes) varie de 100 à 450 avec un pas de 50.

Dans le tableau 5.3, les colonnes « % résolues », « Temps » et « #Flips » correspondent à la moyenne du nombre d'instances résolues, la moyenne du temps requis (en seconde) pour la résolution et le nombre total de mouvements réalisés. Pour les instances à 100, 150, 200, 250 et 300 variables, $CN\text{-}SAT$* et R-Novelty+ résolvent sensiblement la même quantité d'instances, même si R-Novelty+ est plus rapide. Pour le reste des instances, R-Novelty+ dépasse les deux version

[2] Difficiles dans le sens où le ratio $\frac{Nb\ clauses}{Nb\ variables}$ est au seuil de 4.25

[3] Cette valeur a été choisie empiriquement car c'est celle qui a fournie les meilleurs résultats.

Instances	R-Novelty+		CN-SAT Approx		CN-SAT Exact	
	Temps	# Flips	Temps	# Flips	Temps	# Flips
qg1-07	15,21	1580369	13,63	112616	0,85	4432
qg1-08	- -	- -	- -	- -	144,11	310072
qg2-07	18,61	2007560	184,54	1281703	45,08	277824
qg2-08	- -	- -	- -	- -	18,8	35191
qg3-08	60,67	35467402	- -	- -	29,75	1107726
qg4-09	46,8	24648700	61,33	1989338	41,83	1151820
qg5-11	- -	- -	- -	- -	17,3	32108
qg6-09	- -	- -	129,09	1089167	121,19	1073697
qg7-09	- -	- -	0,29	1659	6,61	40430
qg7-13	- -	- -	- -	- -	- -	- -
bw_large.a	0	1688	0,07	5575	0,18	19046
bw_large.b	0,14	137167	20,22	1248845	1,61	90192
bw_large.c	66,57	36721914	- -	- -	- -	- -
bw_large.d	- -	- -	- -	- -	- -	- -
ais10	3,22	1960070	- -	- -	12,84	1849783
ais12	44,98	21858330	- -	- -	- -	- -
ais6	0,03	37574	10,44	5000029	0	1794
ais8	0,39	342178	44,12	15000127	1,3	288649
par8-1-c	0	4245	0,06	17641	0,02	4962
par8-1	5,07	11513624	0,01	2648	0,01	1152
par8-2-c	0,12	282750	0,02	6792	0,01	3602
par8-2	4,09	9308173	36,56	8001315	10,16	2003006
par8-3-c	5	11786722	48,27	13000763	3,41	789625
par8-3	7,25	16001600	36,6	8001176	6,43	1001334
par8-4-c	0,4	905109	3,55	1000756	0,53	133122
par8-4	54,92	122546333	0,01	1573	0,01	1547
par8-5-c	1,22	3000128	4,49	1005056	2,86	567755
par8-5	22,07	49074973	237,6	43377770	30,6	5359820
par16-1-c	- -	- -	- -	- -	199,45	13246954
Par16-2-c	- -	- -	- -	- -	- -	- -
Par16-3-c	- -	- -	- -	- -	- -	- -
par16-4-c	- -	- -	- -	- -	58,06	3917540
par16-5-c	- -	- -	- -	- -	149,99	8495650

TAB. 5.2 – CN-SAT vs. R-Novelty+ sur des instances structurées.

Taille		R-Novelty+			CN-SAT Approx			CN-SAT Exact		
$nbVar$	$nbClauses$	% Résolues	Temps	# Flips	% Résolues	Temps	# Flips	% Résolues	Temps	# Flips
100	425	100,00%	0,02	46198,1	100%	0,32	237870	100%	0,07	29685,26
150	637	100,00%	0,08	157475,51	100%	2,09	1261123	100%	0,93	298455,54
200	850	100,00%	0,04	64346,41	100%	4,62	1335377	100%	3,02	809924,87
250	1062	100,00%	1,22	2062181,27	100%	11,92	4703387	100%	20,98	5006912,67
300	1275	100,00%	0,68	1154850,27	100%	21,36	6670677	93%	33,67	7140076,12
350	1487	100,00%	0,87	1500417,86	97%	39,85	11338048	85,00%	50,99	9920538,94
400	1700	100,00%	7,84	13106152,22	87%	41,02	9980921	68,00%	70,96	12686260,23
450	1912	100,00%	3,39	5177363,17	85%	52,47	11519389	45,00%	133,04	22190221,05

TAB. 5.3 – CN-SAT vs. R-Novelty+ sur des instances 3-SAT aléatoires (seuil 4,25)

de CN-SAT aussi bien en terme du temps de calcul que du pourcentage d'instance résolues. On peut également noter que CN-SAT Approx est plus efficace que CN-SAT Exact sur les plus grandes instances aléatoires.

5.5 Travaux de référence, discussion et comparaison

– Comme nous l'avons dit, cette méthode s'inspire des travaux précédents de [Vasquez et al., 2005] pour les CSP binaires. La difficulté supplémentaire pour exploiter cette méthode pour la résolution du problème SAT, et apport majeur de notre travail, se situe au niveau du choix des variables à libérer. En effet, pour un CSP binaire, lorsque une contrainte entre deux variables X et Y est violée par l'affectation de la variable X, il n'y pas de choix quant à la variable à libérer pour restaurer la consistance. Seule la variable Y est candidate, si on ne veut pas libérer la variable que l'ont vient juste d'instancier.
Pour un problème SAT, lorsqu'une clause est falsifiée, il faut choisir parmi toutes les variables de la clause. C'est pour cela que nous avons proposé deux algorithmes calculant le meilleur choix qu'il est possible de faire. Il en est de même pour les CSP non binaires. En effet, lorsqu'une contrainte n-aire est violée, il faut choisir, parmi toutes les variables impliquées dans la clause, celle à libérer. Nos travaux peuvent donc très facilement s'étendre au problèmes des CSP n-aires.
– Jussien et Lhomme [Jussien et Lhomme, 2002] ont proposé un algorithme nommé *Path-repair* conçue pour les CSP et appliquée aux problèmes d'*open-shop scheduling*. Cette méthode incomplète utilise une liste taboue dans laquelle les *explications* entières des conflits sont stockées sous forme de nogood. Elle diffère donc de notre méthode car nous ne stockons dans la liste taboue que certains mouvements qui ne sont pas forcément connectés aux conflits, ce qui nous évite de rentrer dans des cycles infinis. La liste taboue est utilisée comme moyen de diversification dans notre méthode.

5.6 Conclusions et perspectives

5.6.1 Conclusions

Nous avons proposé une nouvelle méthode de recherche locale pour résoudre le problème SAT. Sa principale caractéristique est que son espace de recherche est composé d'interprétations partielles consistantes. À chaque tentative, une interprétation partielle différente est générée, s'en suit une construction de voisinage assurant la non falsification de toutes les clauses contenant des variables déjà instanciées. Ainsi, au lieu de réparer une interprétation courante, comme dans les méthodes de recherche locale classiques, nous essayons à chaque étape d'instancier une variable tout en préservant la consistance. Derrière ce schéma simple, un problème de taille a dû être traité. Lors de l'instanciation d'une variable, il arrive qu'une ou plusieurs clauses se retrouvent falsifiées. Afin de maintenir la consistance, pour chaque interprétation visitée, nous devons évaluer son voisinage, ce qui revient à trouver un nombre minimal de variables à libérer pour réparer les clauses falsifiées par chaque interprétation voisine de l'interprétation courante.

Le problème du meilleur choix de ces variables est équivalent au problème de calcul de transversaux minimaux. Étant un problème NP-difficile, nous avons initialement proposé un algorithme approché pour réaliser ce calcul. Puis constatant que la taille des transversaux calculés était de l'ordre de quelques variables seulement, nous avons décidé d'implémenter un algorithme exacte pour l'évaluation du voisinage. Ces deux modes d'évaluation ont donné lieu à deux versions de notre méthode que nous avons expérimentée.

Les expérimentations présentées sont prometteuses, même si certaines améliorations devraient être effectuées. En fait, sur les instances difficiles testées, il ne reste vraiment que peu de va-

riables que notre algorithme ne parvient pas à affecter. Cette observation peut être expliquée par la présence de nogoods qui empêchent d'atteindre une interprétation complète. Il est tout de même à noter que sur quelques instances particulièrement difficiles que peu de solveurs incomplets sont capables de résoudre, nous voulons parler des instances *parity*, notre méthode arrive à résoudre quelques instances.

5.6.2 Perspectives

Les suites possibles à donner à ces travaux sont multiples. Tout d'abords, comme nous l'avons déjà évoqué, dans la plupart des instances (particulièrement les instances aléatoires) que notre solveur n'arrive pas à résoudre, seules quelques variables ne sont pas instanciées. Cela nous fait penser à la présence de nogood qu'il faudrait identifier et exploiter dans la mesure du possible pour introduire un autre mécanisme de diversification.

Une autre amélioration qui est envisagée s'appuie sur les travaux de Dubois et Dequen en ce qui concerne le calcul d'ensemble backbone [Dubois et Dequen, 2001]. Nous voudrions guider l'algorithme *greedy()* en le faisant interpréter en priorité, et dans la mesure du possible, les variables présentes dans un backbone calculé au préalable.

Cette méthode pourrait davantage tirer parti des méthodes complètes dont le mode de construction de modèle est semblable. En effet, une des clés de l'efficacité de la plupart de ces méthodes repose sur leur aptitude et leur efficacité à propager les clauses unitaires. Il est regrettable qu'à l'heure actuelle, notre solveur ne puisse l'intégrer. Il y a cependant une raison à cela. Si la propagation unitaire n'a pas été implémentée, c'est à cause du problème majeur suivant : lorsque l'on effectue un mouvement et que certaines clauses sont falsifiées, alors il faut libérer un certain nombre de variables pour supprimer les conflits. Or à chaque fois que l'on libère une variable pour réparer une clause falsifiée, cette même clause devient une clause unitaire. La propagation de cette clause peut poser un problème si le mouvement qu'elle implique apparaît dans la liste taboue, ce qui s'avère malheureusement souvent le cas. De plus il s'avère que ces propagations unitaires fassent apparaître de nouvelles clauses falsifiées, entraînant la libération d'autres variables, créant d'autres clauses unitaires et ainsi de suite. Il faudrait donc trouver un moyen de couper ces cycles infinis pour pouvoir exploiter la propagation unitaire.

Une dernière amélioration pourrait consister à intégrer de la marche aléatoire analogue à celle de l'algorithme WalkSat pour augmenter la diversification lors de l'exploration.

Enfin, cette approche peut également être adaptée pour traiter la variante MAX-SAT du problème SAT en changeant la fonction objectif pour maximiser le nombre de clauses satisfaites au lieu du nombre de variables instanciées.

Troisième partie

Une forme normale généralisée en logique propositionnelle pour les problèmes SAT et CSP

Chapitre 6

Un formalisme exprimant les problèmes SAT et CSP n-aires

Sommaire

6.1 Introduction

La résolution de contraintes est un des moyens les plus utilisés pour résoudre des problèmes en intelligence artificielle. Principalement, deux formalismes sont utilisés pour la représentation

et l'expression des contraintes : le formalisme propositionnel (SAT) généralement exprimé sous forme CNF et le formalisme des problèmes de satisfaction de contraintes (CSP).

Chacun de ces deux formalismes possède des avantages et des défauts, tant au niveau représentation qu'au niveau résolution. Pour le problème SAT, plus ancien que les CSP, le principal avantage réside dans ses méthodes de résolutions particulièrement robustes qui sont capables de résoudre n'importe quel type de contraintes exprimées sous forme CNF. En plus de leur robustesse, leur variété est également un atout : on trouve des méthodes basées sur le calcul de résolvantes, d'autres basées sur une énumération. D'un autre côté, dans ce formalisme, la structure des problèmes est souvent perdue ou cachée la rendant difficilement exploitable. L'avantage majeur du formalisme CSP en revanche réside dans la formulation des problèmes, dans laquelle la structure est conservée en générale et beaucoup plus facile à exploiter pour propager des contraintes. Malheureusement, il existe un distinction notable entre les méthodes de résolution de CSP binaires et celles de CSP n-aires. En effet, la plupart des méthodes de résolution efficaces ne fonctionnent que sur des CSP binaires, bien que depuis quelques années cette différence tend à s'amoindrir. Ces progrès ont été réalisés notamment grâces à l'exploitation de contraintes dites globales où à la généralisation aux contraintes n-aires de méthodes conçues pour les CSP binaires comme l'algorithme *Forward Checking*.

Les formalismes SAT et CSP n'en restent pas moins étroitement liés. Il existe plusieurs travaux sur les transformations de CSP en instances SAT, dont certaines ne fonctionnent que sur des CSP binaires, et plusieurs transformations de SAT vers les CSP. Cependant, la plupart des transformations de CSP vers SAT souffrent d'un léger accroissement de la taille du problème et de la perte de sa structure.

Fort de ces constatations, la question de l'existence d'un formalisme plus général qui permettrait de tirer profit de tous les avantages de chacun de ces deux formalismes se pose naturellement.

C'est pour tenter de répondre à cette question que nous présentons dans ce chapitre une représentation logique appelée GNF (General Normal Form) qui est une généralisation de la forme CNF classique. De plus, nous montrons qu'en rajoutant un opérateur de cardinalité à ce formalisme, nous obtenons un nouveau foamlisme mixte : le formalisme CGNF (Cardinality General Normal Form). Ce dernier permet d'exprimer des instances SAT, ainsi que des instances CSP, et possède certains avantages des formalismes SAT et CSP. En l'occurrence, lorsque l'on représente un CSP exprimé en extension sous forme CGNF, la représentation n'augmente pas la taille du problème et surtout conserve sa structure. Nous présentons deux méthodes pour traiter un problème exprimé sous forme CGNF qui sont des généralisations des méthodes de calcul de résolvantes et d'énumérations connues pour le formalisme SAT. Comme dans le cas de SAT, ces méthodes s'appliquent sur tous types de contraintes. La méthode généralisant le calcul de résolvantes permet de concevoir une méthode de preuve de l'inconsistance, alors que la seconde, généralisation de la procédure DPLL permet la conception d'une méthode de recherche de solution.

Nous montrons également que dans le cas particulier d'instances CGNF codant des CSP, une règle d'inférence peut être définie. L'application de cette règle permet de récupérer des filtrages par consistances partielles comme la consistance d'arc ou la consistance de chemin. Ainsi, nous montrons que la méthode énumérative généralisant la procédure DPLL s'avère équivalente à l'algorithme *Forward Checking* (FC) ou *Maintaining Arc Consistency* (MAC) selon l'utilisation faite d'une règle d'inférence.

Ces travaux ont donné lieux aux publications suivantes : [Paris et al., 2005], [Paris et al., 2006a].

6.2 Un codage commun pour SAT et CSP

L'idée d'encoder un CSP sous la forme SAT fut introduite par De Kleer dans [Kleer, 1989]. Il proposa le *codage direct*. Ensuite, Kasif [Kasif, 1990] a proposé le *codage AC* pour les CSP binaires. Plus récemment, Bessière et al. [Bessière et al., 2003] ont généralisé le *codage AC* aux CSP n-aires. Notre approche est différente. Elle consiste à fournir une forme logique généralisée qui peut prendre en compte les formes CNF (SAT) et CSP, au lieu de transformer des CSP sous la forme SAT. Nous décrivons dans cette section ce nouveau codage logique et nous montrons comment les CSP n-aires sont naturellement représentés de manière optimale (sans accroissement de la taille de la représentation par rapport à la formulation CSP) dans ce codage.

6.2.1 La forme normale généralisée (GNF)

Une clause généralisée C est une disjonction de formules booléennes $f_1 \vee \ldots \vee f_m$ où chaque f_i est une conjonction de littéraux, *i.e.* $f_i = l_1 \wedge l_2 \wedge \ldots \wedge l_n$. Une formule est sous forme normale généralisée (GNF) si et seulement si c'est une conjonction de clauses généralisée. La sémantique des clauses généralisée est triviale : la clause généralisée C est satisfaite par une interprétation I si au moins une de ses conjonctions f_i ($i \in [1, m]$) est *vrai* dans I, sinon elle est falsifiée par I. Une clause classique est une clause généralisée simplifiée dans laquelle toutes les conjonctions f_i sont réduites à de simples littéraux. Ceci prouve que GNF est une généralisation de CNF. Nous montrons dans la suite que chaque contrainte C_i d'un CSP donné est représentable par une clause généralisée. Nous obtenons la taille de la représentation optimale en utilisant les formules de cardinalité $(\pm 1, L)$ qui signifie « exactement 1 littéral parmi ceux de la liste L doit être affecté à *vrai* dans tout modèle », pour exprimer efficacement que chaque variable d'un CSP ne doit être affectée qu'à une unique valeur de son domaine. On note *CGNF* la forme *GNF* combinée aux formules de cardinalité.

6.2.2 Le codage CGNF et les CSP n-aires

Pour coder un CSP n-aire $P = (X, D, C, R)$ sous la forme CGNF, nous définissons d'abord l'ensemble des variables propositionnelles utilisées pour la représentation, puis deux types de clauses : les *clauses de domaine* et les *clauses de contrainte*, nécessaires pour coder les domaines et les contraintes du CSP.

- L'ensemble des variables booléennes : comme dans les autres codages on associe une variable booléenne Y_v à chaque valeur possible v du domaine de chaque variable Y du CSP. Ainsi, Y_v =*vrai* signifie que la valeur v est affectée à la variable Y du CSP. Nous avons besoin de $\sum_{i=1}^{n} |D_i|$ variables booléennes. Ce nombre est majoré par nd.
- Les clauses de domaine : soient Y une variable CSP et $D_Y = \{v_0, v_1, \ldots, v_k\}$ son domaine. La formule de cardinalité $(\pm 1, Y_{v_0} \ldots Y_{v_k})$ oblige la variable Y à être affecté à exactement une valeur de D_Y. Nous avons besoin de n formules de cardinalité pour représenter les n clauses de domaine.
- Les clauses de contrainte : chaque contrainte C_i du CSP P est représentée par la clause généralisée définie comme suit : Soit R_i la relation correspondant à C_i impliquant l'ensemble de variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_a}\}$. Chaque tuple $t_j = (v_{j_1}, v_{j_2}, \ldots, v_{j_a})$ de R_i est exprimé par la conjonction $f_j = X_{v_{j_1}} \wedge X_{v_{j_2}} \wedge \ldots \wedge X_{v_{j_a}}$. Si R_i contient k tuples $t_1, t_2, \ldots, t_k$, on introduit la clause généralisée $C_{C_i} = f_1 \vee f_2 \vee \ldots \vee f_k$, pour exprimer la contrainte C_i. Nous avons besoin de m clauses généralisées pour exprimer les m contraintes du CSP.

Remarque 5 (Notations) ***Variables :*** *Afin de bien différencier les variables propositionnelles codant des valeurs de variables CSP, nous les noterons par une lettre majuscule portant le nom de la variables CSP, indexée par la valeur du domaine de la variables qu'elle représente.*

Clauses de domaine : *Une clause de domaine (qui est une formule de cardinalité) sera notée par la lettre majuscule C indexée par la variable CSP à laquelle est elle est associée.*

Clauses de contrainte : *Une clause de contrainte (qui est une clause généralisée) sera notée par la lettre majuscule C suivi de la contrainte CSP à laquelle elle est associée (généralement la lettre C indexée par l'ensemble des variables qu'elle implique).*

Comme les clauses de domaines sont encodées en $o(nd)$ et les clauses de contraintes en $\mathcal{O}(mad^a)$, la taille du codage *CGNF* pour un CSP est de $\mathcal{O}(mad^a + nd)$ dans le pire des cas. En fait, on peut même considérer que c'est en $\mathcal{O}(mad^a)$ étant donné que nd est souvent négligeable. Cette complexité spatiale est identique à celle du codage du CSP original, ce qui justifie l'optimalité de la complexité spatiale du codage *CGNF*. Les auteurs de [Bessière et al., 2003] annoncent une complexité spatiale dans le pire des cas en $\mathcal{O}(mad^a)$ pour le codage k-AC. D'après ce que nous avons compris de ce calcul, cette complexité est arrondie et ne tient compte ni des clauses "at-least-one", ni des clauses "at-most-one" de ce codage. Ce qui aurait augmenté sa complexité à $\mathcal{O}(mad^a + nd^2)$.

Exemple 18 *Nous reprenons l'exemple 2.1 du chapitre 2 traitant d'un problème de production de voitures :*

- *On définit une variable propositionnelle pour chaque valeur du domaine de chaque variable du CSP. Ainsi chaque variable CSP donne :*
  - Pare-chocs *:* Pc_{blanc}.
  - Toit ouvrant *:* T_{rouge}.
  - Enjoliveurs *:* E_{rose} *et* E_{rouge}.
  - Carrosserie *:* $C_{\text{blanc}},\ C_{\text{rose}},\ C_{\text{rouge}}$ *et* C_{noir}.
  - Portière *:* $P_{\text{rose}},\ P_{\text{rouge}}$ *et* P_{noir}.
  - Capot *:* $Ct_{\text{rose}},\ Ct_{\text{rouge}}$ *et* Ct_{noir}.
- *On définit deux types de clauses :*
  - *les clauses de domaine (une pour chaque variable du CSP) :*
    $C_{\text{Pare-chocs}} = (\pm 1, Pc_{\text{blanc}})$
    $C_{\text{Toit ouvrant}} = (\pm 1, T_{\text{rouge}})$
    $C_{\text{Enjoliveurs}} = (\pm 1, E_{\text{rose}}\ E_{\text{rouge}})$
    $C_{\text{Carrosserie}} = (\pm 1, C_{\text{blanc}}\ C_{\text{rose}}\ C_{\text{rouge}}\ C_{\text{noir}})$,
    $C_{\text{Portières}} = (\pm 1, P_{\text{rose}}\ P_{\text{rouge}}\ P_{\text{noir}})$
    $C_{\text{Capot}} = (\pm 1, Ct_{\text{rose}}\ Ct_{\text{rouge}}\ Ct_{\text{noir}})$
  - *les clauses de contrainte (une pour chaque contrainte du CSP) :*
    $CC_{Pc-C} = (Pc_{\text{blanc}} \wedge C_{\text{rose}}) \vee (Pc_{\text{blanc}} \wedge C_{\text{rouge}}) \vee (Pc_{\text{blanc}} \wedge C_{\text{noir}})$
    $CC_{T-C} = (T_{\text{rouge}} \wedge C_{\text{noir}})$
    $CC_{E-C} = (E_{\text{rose}} \wedge C_{\text{rouge}}) \vee (E_{\text{rose}} \wedge C_{\text{noir}}) \vee (E_{\text{rouge}} \wedge C_{\text{noir}})$
    $CC_{C-P-Ct} = (C_{\text{rose}} \wedge P_{\text{rose}} \wedge Ct_{\text{rose}}) \vee (C_{\text{rouge}} \wedge P_{\text{rouge}} \wedge Ct_{\text{rouge}}) \vee (C_{\text{noir}} \wedge P_{\text{noir}} \wedge Ct_{\text{noir}})$

Nous revenons maintenant sur la correction et la complétude du codage *CGNF*.

Définition 74
Soit P un CSP et C son codage CGNF. Soit I une interprétation de C. On définit l'instanciation

équivalente à I pour P dans le formalisme CSP comme étant l'instanciation I_p qui vérifie la condition suivante : pour toute variable CSP X et pour toute valeur v de son domaine, $X = v$ dans I_p si et seulement si $I[X_v]$ =vrai.

Théorème 1 *Soient P un CSP, C son codage CGNF, I une interprétation du codage CGNF et I_p l'instanciation équivalente à I pour P dans le formalisme CSP. I est un modèle de C si et seulement si I_p est une solution de P.*

Preuve *Soit I un modèle de C et I_p son instanciation équivalente dans P. I satisfait chaque clause de C, car c'est un de ses modèle. Nous devons prouver que chaque variable du CSP est assignée a une unique valeur de son domaine dans I_p et que I_p satisfait toutes les contraintes de P. Si une clause de domaine $C_X = (\pm 1, X_{v_1}\ X_{v_2} \ldots X_{v_d})$ de C est satisfaite, cela signifie qu'une et une seule variable booléenne parmi $\{X_{v_1}\ X_{v_2} \ldots X_{v_d}\}$ est vraie dans I. Comme toutes les clauses de domaine sont satisfaites, chaque variable du CSP est affectée à une seule variable de son domaine. De plus, chaque clause de contrainte $CC_i = f_1 \vee f_2 \vee \ldots \vee f_m$ a au moins une de ses f_i satisfaite dans I. Ce qui signifie que les valeurs affectées au variables CSP impliquées dans la contrainte C_i associée à CC_i forme un tuple de la relation R_i. Ainsi, C_i est satisfaite par I_p et toutes les contraintes de P sont satisfaites par I_p. Donc I_p est une solution de P.La réciproque se démontre de manière analogue.*

6.3 Généralisation de la règle de résolution propositionnelle

6.3.1 la règle de résolution généralisée

La méthode de résolution [Robinson, 1965] est bien connue pour être correcte et complète pour tester l'insatisfaisabilité de clauses propositionnelles. Cette méthode est basée sur la règle de résolution de Robinson. Nous allons généraliser cette règle pour des formules exprimées sous forme normale généralisée (GNF) et prouver que cette règle généralisée est correct et complète pour tester l'insatisfaisabilité de formules GNF.

Nous rappelons ci-dessous la règle de résolution entre deux clauses.

Définition 75 (Le principe de résolution propositionnel : la règle de résolution)
Soit $C_1 = (x_1 \vee x_2 \vee \ldots \vee x_n)$ et $C_2 = (\neg x_1 \vee y_2 \vee \ldots \vee y_m)$ deux clauses que nous notons $C_1 = (x_1 \vee X)$ et $C_2 = (\neg x_1 \vee Y)$. La résolvante de C_1 et C_2 est la clause $R = X \vee Y$ obtenue en appliquant la résolution sur x_1 et $\neg x_1$.

La résolvante $X \vee Y$ est une conséquence logique de ses parents C_1 et C_2 (c'est à dire $C_1 \wedge C_2 \models X \vee Y$). Maintenant, nous étendons cette règle aux formules GNF.

Définition 76 (La règle de résolution généralisée)
Soit $C_1 = f_1 \vee f_2 \vee \ldots \vee f_i \vee \ldots \vee f_n$ et $C_2 = g_1 \vee g_2 \ldots \vee g_j \vee \ldots \vee g_m$ deux clauses généralisées. Si il existe $i \in [1, n]$ et $j \in [1, m]$ tels que $f_i \wedge g_j \vdash \square$, alors la clause généralisée $C_{1,2} = f_1 \vee f_2 \vee \ldots \vee f_{i-1} \vee f_{i+1} \vee \ldots \vee f_n \vee g_1 \vee g_2 \vee \ldots \vee g_{j-1} \vee g_{j+1} \vee \ldots \vee g_m$ est la résolvante généralisée de C_1 et C_2, lorsque la résolution est appliquée sur les littéraux généralisés f_i et g_j.

Tester si deux littéraux généralisés sont « opposés » ($f_i \wedge g_j \vdash \square$) est très simple de part la nature des littéraux généralisés. f_i et g_j étant des conjonction de littéraux, il suffit de tester si f_i

contient un littéral opposé à un littéral de g_j. Si $l \in f_i$ et $\neg l \in g_j$, alors $(l \wedge \neg l) \in \{f_i \wedge g_j\}$ donc $f_i \wedge g_j$ est trivialement inconsistant.

Remarque 6 *Remarquons que la règle de fusion entre deux clauses s'étend tout à fait naturellement au clauses généralisées. Si une clause généralisée contient plusieurs fois un même littéral généralisé, alors elle peut être remplacée par la même clause ne contenant qu'une seule occurrence de ce littéral. Par exemple :* $(a \wedge \neg b) \vee (a \wedge \neg b) \vdash (a \wedge \neg b)$

Une méthode de résolution généralisée pour des formules exprimées sous forme GNF peut être définie par une application récursive de la règle de résolution généralisée jusqu'à l'obtention d'une clause vide ou jusqu'à obtenir un point fixe, en appliquant toutes les fusions. Avant de prouver la correction et la complétude de la résolution généralisée, nous définissons la notion d'arbres de résolution et d'arbres de réfutation.

Définition 77 (Arbres de résolution et arbres de réfutation)
Un arbre de résolution pour une formule Γ exprimée sous forme GNF est un arbre enregistrant l'historique de la preuve dans lequel les feuilles sont étiquetées par les clauses généralisées exprimant Γ et où chaque nœud intérieur est étiqueté par une clause généralisée qui est la résolvante généralisée de ces deux parents. Un arbre de réfutation est un arbre de résolution qui possède une racine étiquetée par la clause vide ($\square$).

Il est à noter que dans certains cas, toutes les clauses généralisées ne participent pas à la dérivation de la clause vide. Ceci se traduit au niveau des arbres de réfutation par des feuilles non connectées qui sont étiquetées par ces clauses généralisées. La notion d'arbre est donc quelque peu détériorée, mais les raisonnements que nous ferons en utilisant les arbres ne seront pas altérés par ces cas particuliers.

Proposition 9 (Correction) *La méthode de résolution généralisée est correcte.*

Preuve *Il suffit de prouver que chaque résolvante généralisée est une conséquence logique des deux parents dont elle est inférée. Cela revient à prouver que $C_1 \wedge C_2 \models C_{1,2}$, ce qui est trivial.*

Maintenant, nous prouvons la complétude de la méthode de résolution généralisée.

Proposition 10 (Complétude) *La méthode de résolution généralisée est complète, c'est-à-dire qu'une formule Γ exprimée sous forme GNF est insatisfaisable si et seulement si la clause vide ($\square$) peut être produite en appliquant la règle de résolution généralisée récursivement et en réalisant toutes les fusions possibles.*

Preuve *Soit Γ une formule insatisfaisable exprimée sous forme GNF. Nous nous appuyons sur le fait que produire la clause vide à partir d'une formule GNF Γ revient à trouver un arbre de réfutation pour Γ. Soit $D(\Gamma)$ le nombre de disjonctions ($\vee$) de Γ. Par convention $D(\square) = 0$. La preuve se fait par induction sur $D(\Gamma)$:*

Le cas de base : si $D(\Gamma) = 0$, alors, soit $\Gamma = \square$, et dans ce cas la clause vide est produite directement, soit il existe deux clause généralisées unitaires f_i et g_j telles que $f_i \equiv \neg g_j$ (sinon Γ serait satisfaisable). Dans ce cas la clause vide est déduite en appliquant la règle de résolution généralisée sur f_i et g_j.

Phase d'induction : supposons que la proposition 10 soit vraie pour toute Γ *telle que* $D(\Gamma) < n$*, nous devons prouver que cette proposition est toujours valide pour toute formule* Γ *telle que* $D(\Gamma) = n$*.*

Dans le cas où $0 < D(\Gamma) = n$*, on peut décomposer* Γ *en* $\Gamma = \Gamma' \wedge C$ *où* Γ' *est un sous-ensemble de clauses de* Γ *non vide, et* C *est une clause généralisée contenant au moins une disjonction. La clause généralisée* C *peut être écrite à son tour* $C = C' \vee f_i$*, où* f_i *est un littéral généralisé et* $C' \neq \Box$ *contient le reste de la clause* C*. Ainsi,* $\Gamma = \Gamma' \wedge (C' \vee f_i) = (\Gamma' \wedge C') \vee (\Gamma' \wedge f_i)$*.* Γ *est insatisfaisable par hypothèse, donc à la fois* $(\Gamma' \wedge C')$ *et* $(\Gamma' \wedge f_i)$ *sont insatisfaisables. De plus, on peut remarquer que* $D(\Gamma' \wedge C') < n$ *et que* $D(\Gamma' \wedge f_i) < n$*. Par hypothèse d'induction, on peut donc construire un arbre de réfutation pour* $(\Gamma' \wedge C')$ *et pour* $(\Gamma' \wedge f_i)$ *que nous notons respectivement* D_1 *et* D_2*. Considérons l'arbre* D_1*, si nous ajoutons* f_i *aux feuilles étiquetées par* C' *et à chacun des nœuds intérieurs correspondant à une résolvante ayant* C' *comme ancêtre , nous obtenons un arbre de résolution noté* D'_1 *pour* $(\Gamma' \wedge (C' \vee f_i)) = \Gamma$*. Il y a deux cas possibles : soit* D'_1 *a une racine étiquetée par* $\Box$ *du fait de l'existence d'un nœud dans* D'_1 *étiqueté par* g_j *tel que* $f_i \equiv \neg g_j$*, ainsi* D'_1 *est un arbre de réfutation pour* Γ *; ou alors,* D'_1 *a une racine étiquetée par* f_i*. Dans ce cas, nous pouvons greffer l'arbre* D_2*, correspondant à* $\Gamma' \wedge f_i$*, à* D'_1 *et nous obtenons un arbre* $D'_1 \cup D_2$ *dont la racine est étiquetée par* $\Box$*, qui est donc un arbre de réfutation pour* Γ*.*

La preuve de la réciproque est triviale, du fait de la correction de la règle de résolution généralisée, et du fait que la fermeture transitive Γ^*[1] *et* Γ *sont logiquement équivalent.*

Remarque 7 *La méthode de résolution généralisée est également correcte et complète dans le cas de formules CGNF (le formalisme CGNF étant la combinaison de clauses généralisées et de formules de cardinalité). En particulier, cette méthode l'est pour des instances de CSP codées en CGNF.*

Il est important de constater que dans le cas de formules CGNF, la règle de résolution généralisée peut être appliquée entre deux clauses généralisées, disons C_i et C_j, lorsqu'il existe une formule de cardinalité, disons C_X, qui contient deux littéraux différents X_i et X_j qui apparaissent respectivement dans CC_i et CC_j. En effet, si X_i apparaît dans le littéral généralisé f_i de CC_i et X_j apparaît dans le littéral généralisé g_j de CC_j, alors $f_i \wedge \neg g_j \vdash \Box$. Ainsi, la règle peut être appliquée entre CC_i et CC_j. Un bon exemple de formule CGNF est une instance codant un CSP, nous décrivons donc un exemple de CSP pour illustrer le raisonnement.

Exemple 19 *Soit* Γ *une formule CGNF codant une instance CSP où les clauses sont les suivantes :*

$C_X = (\pm 1, X_1 X_2)$
$C_Y = (\pm 1, Y_1 Y_2)$
$C_Z = (\pm 1, Z_1 Z_2)$
$CC_{XY} = (X_1 \wedge Y_1) \vee (X_2 \wedge Y_2)$
$CC_{XZ} = (X_1 \wedge Z_2) \vee (X_2 \wedge Z_1)$
$CC_{YZ} = (Y_1 \wedge Z_1) \vee (Y_2 \wedge Z_2)$

En suivant le raisonnement donné dans la preuve de la proposition 10, nous pouvons réécrire Γ comme $\Gamma = \Gamma' \wedge (C' \vee f_i)$ où $\Gamma' = \Gamma - \{C_{YZ}\}$, $f_i = (Y_2 \wedge Z_2)$ et $C' = (Y_1 \wedge Z_1)$. Nous obtenons l'arbre de réfutation D_1 de la figure 6.1 pour $(\Gamma' \wedge C')$ et l'arbre de réfutation D_2 de la figure 6.2 pour $(\Gamma' \wedge f_i)$.

[1] Γ^* est l'union de Γ et de toutes les résolvantes généralisées que l'on peut en déduire récursivement.

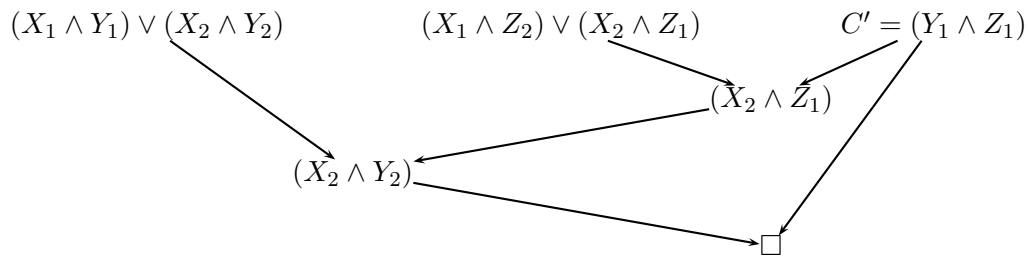


FIG. 6.1 – D_1 : l'arbre de réfutation de $(\Gamma' \wedge C')$

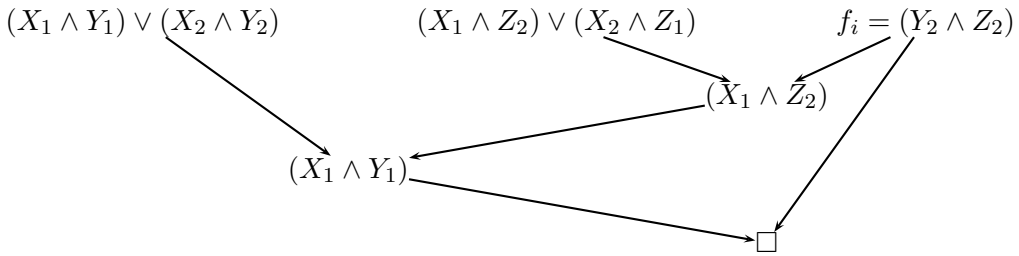


FIG. 6.2 – D_2 : l'arbre de réfutation de $(\Gamma' \wedge f_i)$

si nous rajoutons f_i à C' dans D_1 ainsi qu'à chacune des résolvantes généralisées dérivées de C', nous obtenons l'arbre D'_1 de la figure 6.3, qui n'est autre qu'un arbre de résolution pour Γ.

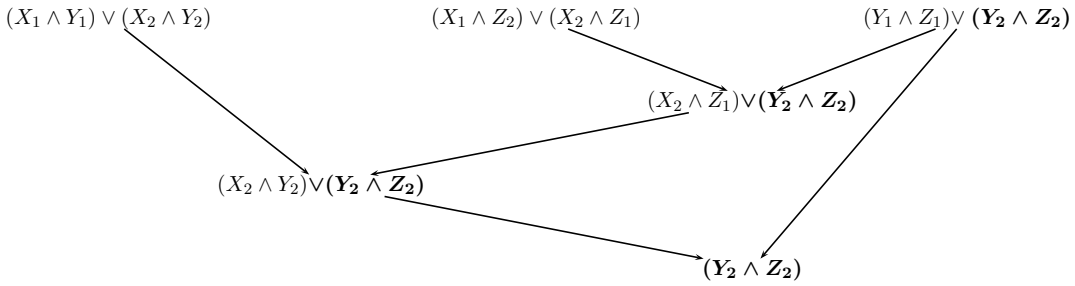


FIG. 6.3 – D'_1 : un arbre de résolution pour (Γ)

Et enfin, en greffant D_2 sous D'_1 comme montré en gras dans la figure 6.4, nous obtenons un arbre de réfutation pour Γ, permettant d'exhiber une preuve de l'inconsistance de Γ.

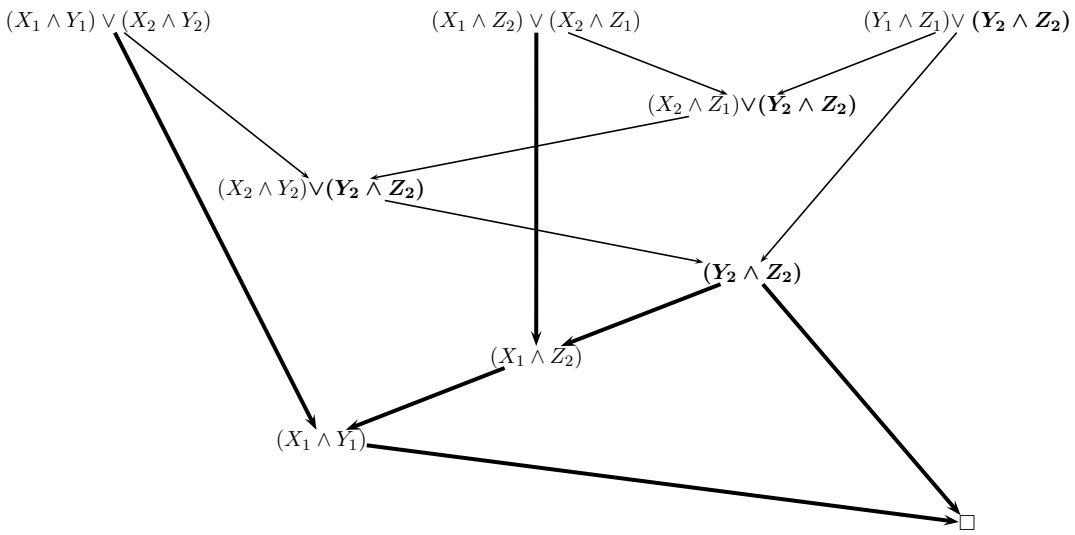


FIG. 6.4 – Un arbre de réfutation pour Γ (les arêtes en gras représentent l'arbre D_2, les autres représentent D'_1).

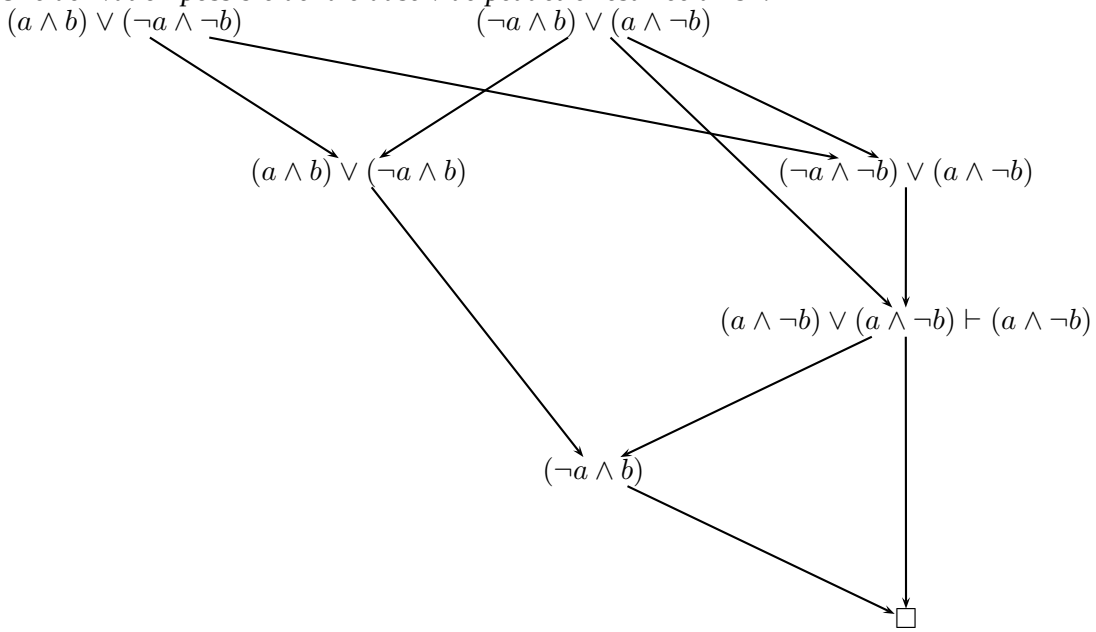
Voici un autre exemple montrant la dérivation de la clause vide pour une instance GNF :

Exemple 20 *Soit la formule GNF codant :* $(a \Leftrightarrow b) \wedge (a \Leftrightarrow \neg b)$ *:*

C_1 *:* $(a \wedge b) \vee (\neg a \wedge \neg b)$

C_2 *:* $(\neg a \wedge b) \vee (a \wedge \neg b)$

Une dérivation possible de la clause vide peut être résumée ainsi :



6.3.2 Un algorithme pour la méthode de résolution généralisée

La correction, et surtout la complétude de la méthode de résolution généralisé nous permet d'écrire un premier algorithme basé sur la règle de résolution généralisée pour résoudre le problème de satisfaisabilité pour une instance CGNF codant un CSP. Cet algorithme est décrit par l'algorithme 14.

Algorithme 14 Méthode de résolution généralisée CGNF

Fonction MetResGen
Entrée : une formule CGNF Γ (codant un CSP)
Sortie : la réponse au test de satisfaisabilité pour Γ

1: CC_1, CC_2 : deux clauses
2: L_1, L_2 : deux littéraux généralisés
3: **tant que** $\square \notin \Gamma$ **faire**
4: $(CC_1, CC_2, L_1, L_2) \leftarrow \mathrm{SelClaRes}(\Gamma)$
5: **si** $(CC_1, CC_2) = (\emptyset, \emptyset)$ **alors**
6: **retourner** *Insatisfaisable*
7: **fin si**
8: $\Gamma \leftarrow \Gamma \cup \mathrm{RegResGen}(CC_1, CC_2, L_1, L_2)$
9: **fin tant que**
10: **retourner** *Satisfaisable*

Cet algorithme fait appel à deux fonctions. La première est la fonction *RegResGen* qui prend en paramètre deux clauses généralisées ainsi que deux littéraux généralisés leur appartenant, retourne la clause généralisée correspondant à la résolvante de ces deux clauses appliquée selon les deux littéraux. Son code est donné dans l'algorithme 15.

Algorithme 15 Règle de résolution généralisée

Fonction RegResGen
Entrée : deux clauses généralisées CC_1 et CC_2, ainsi que deux littéraux généralisés L_1 et L_2 appartenant respectivement à C_1 et C_2
Sortie : la résolvante généralisée issue de CC_1 et CC_2 portant sur les littéraux généralisé L_1 et L_2

1: **retourner** $\{CC_1 \backslash L_1\} \vee \{CC_2 \backslash L_2\}$

La seconde est la fonction de *sélection des clauses* sur lesquelles appliquer la règle de résolvante généralisée. La description que nous donnons dans l'algorithme 16 nous permet d'être complet dans ce choix. Cependant, cette complétude s'obtient au prix d'une complexité en temps et en espace exponentielle dans le pire des cas, car le nombre de résolvantes possibles est potentiellement exponentiel.

Ces complexités constituent un frein quant à une éventuelle programmation efficace de la méthode de résolution généralisée, comme ce fut le cas pour le calcul propositionnel. C'est pourquoi nous ne voyons que deux moyens d'exploiter la règle de résolution généralisée : le premier consisterait en une phase de pré-traitement pendant laquelle une stratégie de résolution bornée serait appliquée afin de déduire des résolvantes susceptibles de simplifier le problème. Nous pourrions par exemple nous inspirer des travaux de [Galil, 1975, Li et Anbulagan, 1997] pour le calcul

propositionnel et voir s'il est possible de les étendre au cadre CGNF.

Le second moyen consisterait à utiliser la règle de résolution généralisée de façon incomplète pour prouver l'inconsistance d'instances CGNF. Cette seconde approche pourrait s'inscrire dans le cadre du cinquième challenge proposé par Bart Selman en 1997 lors de la conférence IJCAI [Selman et al., 1997]. Ce challenge portait sur la preuve d'inconsistance à l'aide de méthodes incomplètes.

Algorithme 16 Sélection des clauses pour l'application de la règle de résolution généralisée

Fonction SelClaRes

Entrée : une formule CGNF Γ

Sortie : deux clauses généralisées CC_i et CC_j de Γ, ainsi que deux littéraux généralisés L_i et L_j leur appartenant tels que l'application de la règle de résolution généralisée sur CC_i et CC_j portant sur L_i et L_j ne produit pas de clauses généralisée redondante

1: **pour tout** $CC_{i_1} \in \Gamma$ **faire**
2: **pour tout** $L_{i_2} \in CC_{i_1}$ **faire**
3: **pour tout** $CC_{j_1} \in \Gamma (CC_{i_1} \neq CC_{j_2})$ **faire**
4: **pour tout** $L_{j_2} \in CC_{j_1}$ **faire**
5: **si** $\exists l_i \in L_{i_2}$ tel que $\exists l_j \in L_{j_2} (l_i \neq l_j)$ tel que $\exists C_{\text{card}} (\text{une formule de cardinalité})$ tel que $l_i \in C_{\text{card}}$ et $l_j \in C_{\text{card}}$ et que $\text{RegResGen}(CC_{i_1}, CC_{j_1}, L_{i_2}, L_{j_2}) \notin \Gamma$ **alors**
6: **retourner** $(CC_{i_1}, CC_{j_1}, L_{i_2}, L_{j_2})$
7: **fin si**
8: **fin pour**
9: **fin pour**
10: **fin pour**
11: **fin pour**
12: **retourner** $(\emptyset, \emptyset, \emptyset, \emptyset)$

6.4 Une nouvelle règle d'inférence pour le codage CGNF

Le codage *CGNF* permet une représentation optimale des CSP, cependant il ne capture pas la propriété de consistance d'arc qui est la clé de presque tous les algorithmes énumératifs des CSP. Nous introduisons dans cette section une règle d'inférence simple qui s'applique sur le codage *CGNF* et prouvons que la consistance d'arc peut être établie en temps linéaire en appliquant la règle jusqu'à saturation. Cette règle exploite la structure de l'instance en utilisant une clause de domaine et une clause de contrainte.

6.4.1 Définition de la règle d'inférence IR

Définition 78

Soient P un CSP, C son codage CGNF, CC_i une clause de contrainte, C_X une clause de domaine, L_{C_i} l'ensemble des littéraux figurant dans CC_i et L_X l'ensemble des littéraux de C_X. Si $L_{C_i} \cap L_X \neq \emptyset$ alors on infère chaque négation $\neg X_v$ d'un littéral positif[2] qui figure dans L_X mais pas dans L_{C_i}. Nous obtenons la règle suivante :

[2]Dans le codage CGNF d'un CSP, il n'apparaît que des littéraux positifs.

IR : si $L_{C_i} \cap L_X \neq \emptyset$, $X_v \in L_X$ et $X_v \notin L_{C_i}$ alors $C_X \wedge C_{C_i} \vdash \neg X_v$[3].

Exemple 21 *Soit C_B=$(\pm 1, B_{v_0}\ B_{v_1})$ la clause de domaine correspondant à la variable B et CC_i=$(A_{v_1} \wedge B_{v_0} \wedge C_{v_0}) \vee (A_{v_0} \wedge B_{v_0} \wedge C_{v_1})$ une clause de contrainte correspondant à la contrainte du CSP C_i impliquant les variables $\{A, B, C\}$. L'application de la règle IR sur ces deux clauses donne : $C_B \wedge CC_i \vdash \neg B_{v_1}$.*

Proposition 11 *La règle d'inférence IR est correcte.*

Preuve *Soient X_v un littéral figurant dans L_X mais pas dans L_{C_i} et I un modèle de $CC_i \wedge C_X$. La clause de contrainte CC_i est une disjonction de conjonctions f_i et chaque conjonction f_i contient un littéral de L_X. Au moins une des conjonctions f_i, disons f_j, est satisfaite par I car I est un modèle de CC_i. Ainsi, il y a un littéral $X_{v'}$ ($X_{v'} \neq X_v$ car $X_v \notin L_{C_i}$) de f_j qui figure dans L_X, et tel que $I[X_{v'}]$ =vrai. À cause de l'exclusion mutuelle des littéraux de C_X, $X_{v'}$ est le seul littéral de L_X satisfait par I. Ainsi, $I[\neg X_v]$ =vrai et I est un modèle de $\neg X_v$.*

6.4.2 La règle d'inférence et la consistance d'arc

Définition 79
Un CSP P est arc consistant si et seulement si tous ses domaines sont arc consistants. Un domaine $D_{X_{i_1}}$ est arc consistant si et seulement si pour chaque valeur v_{i_1} de $D_{X_{i_1}}$ et pour chaque contrainte C_j d'aride k impliquant les variables $\{X_{i_1}, \ldots, X_{i_k}\}$ il existe un tuple
$(v_{i_2}, \ldots, v_{i_k}) \in D_{X_{i_2}} \times \ldots \times D_{X_{i_k}}$ *tel que* $(v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \in R_j$.

Nous utilisons la règle d'inférence IR sur le codage CGNF C d'un CSP P, pour établir la consistance d'arc. Nous montrons qu'en appliquant IR sur C jusqu'à saturation (un point fixeest atteint) et en propageant les mono-littéraux négatifs inférés nous maintenons la consistance d'arc sur C avec une complexité linéaire.

Proposition 12 *Soient P un CSP et C son codage CGNF. Une valeur $v \in D_Y$ est filtrée par consistance d'arc dans P si et seulement si la négation $\neg Y_v$ de la variable booléenne correspondante est inférée par l'application de IR sur C.*

Preuve *Soit v une valeur du domaine D_Y qui ne vérifie pas la propriété de consistance d'arc. Il y a au moins une contrainte C_j impliquant Y telle que v n'apparaisse dans aucun des tuples autorisés de la relation correspondante R_j. La variable booléenne Y_v n'apparaît pas dans la clause de contrainte associée CC_j, mais elle apparaît dans la clause de domaine C_Y associée à D_Y. Si on applique la règle IR sur C_Y et CC_j on infère $\neg Y_v$.*

La réciproque se démontre de la même manière.

Théorème 2 *Soient P un CSP et C son codage CGNF. La saturation de IR sur C et la propagation de tous les littéraux inférés est équivalent au filtrage par consistance d'arc dans le CSP P original.*

Preuve *C'est une conséquence de la proposition 12.*

[3] $\vdash$ représente l'inférence logique.

6.4.3 La consistance d'arc par application de IR

Pour programmer un filtrage par consistance d'arc sur C en appliquant IR, nous avons besoin de définir quelques structures de données. On suppose que les nd variables booléennes de C sont encodées par les premiers entiers $[1..nd]$. On définit un tableau OCC_j de taille nd pour chaque clause de contrainte CC_j de telle sorte que $OCC_j[i]$ contient le nombre d'occurrences de la variable i dans CC_j. Il y a un total de m tableaux de ce type qui correspondent aux m clauses de contrainte. Si $OCC_j[k] = 0$ pour un $k \in \{1 \dots nd\}$ et un $j \in \{1 \dots m\}$, la négation $\neg i$ de la variable booléenne i est inférée par IR. Cette structure de donnée ajoute un facteur $\mathcal{O}(mnd)$ en complexité spatiale. La complexité spatiale totale de C est $\mathcal{O}(mad^a + nd + mnd)$, mais les facteurs nd et mnd sont toujours inférieurs au facteur mad^a, donc la complexité spatiale de C reste en $\mathcal{O}(mad^a)$.

Le principe du filtrage par consistance d'arc consiste tout d'abord à lire les m tableaux OCC_j pour détecter les variables $i \in [1 \dots nd]$ qui ont un nombre d'occurrences nul ($OCC_j[i] = 0$). Ceci est fait par les lignes 3 à 7 de l'algorithme 17 en $\mathcal{O}(mnd)$, car il y a m tableaux de taille nd. Ensuite, il faut appliquer la propagation unitaire sur les variables détectées, et propager leur effet jusqu'à saturation (i.e. plus de nouvelle variable i ayant $OCC_j[i] = 0$). La procédure de consistance d'arc est décrite dans l'algorithme 17. Elle fait appel à la procédure ***Propager*** décrite dans l'algorithme 18.

Algorithme 17 Consistance d'arc

Procédure Consistance d'arc
Entrée : une instance CGNF C
Sortie : la même instance C rendue arc-consistante

```
1:  var L : liste de littéraux {à propager}
2:  L ← ∅
3:  pour chaque clause de contrainte CC_j faire
4:     pour chaque littéral i de OCC_j faire
5:        si OCC_j[i] = 0 alors
6:           ajouter ¬i dans L
7:        fin si
8:     fin pour
9:  fin pour
10: tant que L ≠ ∅ et □ ∉ C faire
11:    extraire ¬l de L
12:    (C, L) ← propager(C, ¬l, L, vrai)
13: fin tant que
14: retourner C
```

La complexité de la procédure de consistance d'arc est principalement donnée par l'effet de la propagation des littéraux de la liste L (ligne 10 à 13 de l'algorithme 17). Il est aisé de voir que dans le pire des cas il y aura nd appels à la procédure *propager*. Toutes les propagations dues à ces appels sont effectuées en $\mathcal{O}(mad^a)$ dans le pire des cas. En fait, il y a au plus d^a conjonctions de a littéraux pour chaque clause de contrainte de C. Le nombre total de conjonctions traitées à la ligne 3 de l'algorithme 18 ne peut pas excéder md^a car chaque conjonction f considérée à la ligne 3 est retirée à la ligne 5 (f est interprétée à faux). Comme il y a a littéraux par conjonction f, la propagation totale s'effectue en $\mathcal{O}(mad^a)$. Ainsi, la complexité de la procédure de consistance

Algorithme 18 Propager

Procédure Propager
Entrée : une instance CGNF C, un littéral $\neg i$, une liste L de littéraux à propager, un booléen val
Sortie : la même instance C dans laquelle $\neg i$ a été propagé et la liste L mise-à-jour

```
1: si i n'est pas encore affecté alors
2:    affecter i à la valeur faux
3:    pour chaque conjonction non affectée f de C contenant i faire
4:       affecter f à la valeur faux
5:       pour chaque littéral j de f tel que i ≠ j faire
6:          soustraire 1 à OCC_k[j] {k étant l'indice de la clause de contrainte contenant f}
7:          si OCC_k[j] = 0 et val =vrai alors
8:             ajouter j dans L
9:          fin si
10:      fin pour
11:   fin pour
12: fin si
13: retourner (C, L)
```

d'arc est $\mathcal{O}(mad^a + mnd)$. Mais une fois de plus, le facteur mnd est souvent négligeable devant mad^a. La complexité est donc réduite à $\mathcal{O}(mad^a)$. Elle est linéaire dans la taille de C.

6.5 Deux méthodes énumératives pour le codage CGNF

Nous étudions dans cette partie deux méthodes énumératives : la première (MAC) maintient la consistance d'arc pendant la recherche tandis que la seconde (FC) ne maintient qu'une forme partielle de consistance d'arc, comme le fait la méthode du *forward checking* classique dans les CSP [Haralick et Elliott, 1980]. Ces deux méthodes effectuent une énumération booléenne et des simplifications. Elle sont basées sur une adaptation de la procédure DPLL au codage *CGNF*.

6.5.1 La méthode MAC

MAC commence par un premier appel à l'algorithme 17 pour vérifier la consistance d'arc à la racine de l'arbre de recherche. Ensuite, il se contentera de différents appels à la procédure *propager* décrite par l'algorithme 18 à chaque nœuds de l'arbre pour maintenir la consistance d'arc pendant le recherche. C'est à dire, les mono-littéraux de chaque nœuds sont inférés et leurs effets sont propagés. Le code de *MAC* est décrit dans l'algorithme 19.

6.5.2 La méthode FC

Il est aisé d'obtenir un algorithme équivalent à *Forward Checking (FC)* à partir de celui de MAC pour le codage CGNF. Le principe est le même que pour *MAC*, excepté qu'au lieu de maintenir la consistance d'arc sur C à chaque nœud de l'arbre de recherche, (FC) ne la maintient que sur le voisinage proche de la variable qui vient d'être instanciée. Ceci est fait en restreignant l'effet de la propagation de l'affectation du littéral aux seuls littéraux de la clause de contrainte dans laquelle il apparaît. En remplaçant l'algorithme Satisfaisable par Satisfaisable_FC (algorithme 21)

Algorithme 19 *MAC* pour le codage CGNF.

Procédure MAC
Entrée : une instance CGNF C
Sortie : la réponse au test de satisfaisabilité de C (*satisfaisable ou insatisfaisable*)

```
1: C ← Consistanced'arc(C)
2: si □ ∈ C alors
3:    retourner (insatisfaisable)
4: sinon
5:    choisir un littéral l ∈ C
6:    si satisfaisable(C, l,vrai) alors
7:       retourner (satisfaisable)
8:    sinon si satisfaisable(C, l,faux) alors
9:       retourner (satisfaisable)
10:   sinon
11:      retourner (insatisfaisable)
12:   fin si
13: fin si
```

dans l'algorithme MAC, on obtient l'algorithme Forward Checking. Il est important de noter qu'il est trivial de retrouver l'algorithme FC pour notre codage, alors que ce n'est pas le cas pour les CSP n-aires, où on trouve pas moins de six versions différentes de *FC* [Bessière et al., 2002].

6.5.3 Heuristiques de choix de variable

Étant donné que le codage CGNF conserve la structure du CSP, on peut trouver aisément des heuristiques de choix de variables ou de valeurs équivalentes à celles utilisées dans les CSP. Par exemple, l'heuristique du domaine minimal (*MD*) qui consiste à choisir, pendant la recherche, la variable CSP dont le domaine est le plus petit, est équivalent dans le codage CGNF, à choisir un littéral qui apparaît dans la clause de domaine la plus courte.

Un autre exemple, l'heuristique *DomDeg* qui consiste à choisir la variable qui minimise le ratio $\frac{|D_{X_i}|}{degree(X_i)}$ telle que $i \in [1..n]$, où $degree(X_i)$ représente le nombre d'occurrences de la variable X_i dans toutes les contraintes du CSP. Dans le codage CGNF cette heuristique revient à choisir la variable qui minimise la valeur de $\frac{|C_{D_i}|}{occ(C_{D_i})}$ telle que $i \in [1..n]$ et où $occ(C_{D_i})$ donne le nombre de clauses de contrainte où au moins un littéral X_k de C_{D_i} apparaît.

L'heuristique *MD* a été implémentée dans les méthodes MAC et FC et les résultats obtenus sont présentés à la section suivante.

6.6 Expérimentations

Nous avons expérimenté les deux méthodes *MAC* et *FC* et comparé leurs performances sur des instances du problème de Ramsey et sur des CSP à contraintes d'arités 3 et 4 générés aléatoirement encodés sous formes CGNF. Ces programmes sont écrits en C de manière itérative, compilés et exécutés sous Linux avec un processeur 2.8 Gh et 1Go de RAM.

Algorithme 20 Fonction Satisfaisable.

Fonction Satisfaisable
Entrée : une instance CGNF C, une variable l, un booléen val
Sortie : un booléen

```
1: var L : liste de littéraux
2: L ← ∅
3: si val = vrai alors
4:    affecter l à la valeur vrai
5:    ajouter chaque littéral i ≠ l de la clause de domaine contenant l dans L
6:    tant que L ≠ ∅ et □ ∉ C faire
7:       extraire i de L
8:       (C, L) ← propager(C, i, L,vrai)
9:    fin tant que
10: sinon
11:    (C, L) ← propager(C, l, L,vrai)
12:    si C = ∅ alors
13:       retourner (vrai)
14:    sinon si □ ∈ C alors
15:       retourner (faux)
16:    sinon
17:       choisir un littéral p de C
18:       si satisfaisable(C, p,vrai) alors
19:          retourner (vrai)
20:       sinon si (satisfaisable(C, p,faux)) alors
21:          retourner (vrai)
22:       sinon
23:          retourner (faux)
24:       fin si
25:    fin si
26: fin si
```

Algorithme 21 Fonction Satisfaisable_FC.

Fonction Satisfaisable_FC
Entrée : une instance CGNF C, une variable l, un booléen val
Sortie : un booléen

1: **var** L, L' : listes de littéraux
2: $L \leftarrow \emptyset$
3: $L' \leftarrow \emptyset$
4: **si** $val = vrai$ **alors**
5: affecter l à la valeur *vrai*
6: ajouter chaque littéral $i \neq l$ de la clause de domaine contenant l dans L
7: **tant que** $L \neq \emptyset$ et $\square \notin C$ **faire**
8: extraire i de L
9: (C, L') $\leftarrow \text{propager}(C, i, L'$,*vrai*$)$
10: **fin tant que**
11: **tant que** $L' \neq \emptyset$ et $\square \notin C$ **faire**
12: extraire i de L'
13: (C, L) $\leftarrow \text{propager}(C, i, \emptyset$,*faux*$)$
14: **fin tant que**
15: **sinon**
16: (C, L) $\leftarrow \text{propager}(C, l, \emptyset$,*faux*$)$
17: **si** $C = \emptyset$ **alors**
18: **retourner** (*vrai*)
19: **sinon si** $\square \in C$ **alors**
20: **retourner** (*faux*)
21: **sinon**
22: choisir un littéral p de C
23: **si** $\text{satisfaisable}(C, p$,*vrai*$)$ **alors**
24: **retourner** (*vrai*)
25: **sinon si** ($\text{satisfaisable}(C, p$,*faux*$)$) **alors**
26: **retourner** (*vrai*)
27: **sinon**
28: **retourner** (*faux*)
29: **fin si**
30: **fin si**
31: **fin si**

6.6.1 Le problème de Ramsey

Le problème de Ramsey ([CSPlib,]) est un problème combinatoire bien connu. Il consiste à colorer les arêtes d'un graphe complet à n sommets à l'aide de k couleurs de telle sorte qu'il n'y ai pas de triangle monochromatique. Son codage CGNF nécessite $k \times \frac{(n^2-n)}{2}$ variables booléennes, $\frac{(n^2-n)}{2}$ clauses de domaine de taille k et $C^3_{\frac{(n^2-n)}{2}}$ clauses de contrainte de tailles $3 \times (k^3 - k)$.

Le tableau 6.5 montre les résultats de *MAC* et de *FC* avec l'heuristique MD sur quelques instances du problème de Ramsey où $k = 3$ et n varie entre 5 et 14. Ces problèmes ont des solutions. La première colonne définit le problème : Rn_k dénote l'instance du problème de Ramsey avec n sommets et k couleurs, la seconde et la troisième colonnes montrent les nombres de nœuds et les performances en temps CPU de *FC* et *MAC* respectivement augmentés par l'heuristique (MD) décrite précédemment.

Problème	FC + MD		MAC + MD	
	Noeuds	Temps	Noeuds	Temps
R5_3	*12*	*0 s 48 μs*	*12*	*0 s 85 μs*
R6_3	*27*	*0 s 231 μs*	*21*	*0 s 297 μs*
R11_3	*237*	*0 s 5039 μs*	*103*	*0 s 4422 μs*
R12_3	*538*	*0 s 21558 μs*	*130*	*0 s 11564 μs*
R13_3	*1210*	*0 s 28623 μs*	*164*	*0 s 9698 μs*
R14_3	*216491*	*9 s 872612 μs*	*6735*	*1 s 752239 μs*

FIG. 6.5 – Résultats de *MAC* et *FC* pour quelques problèmes de Ramsey.

On peut voir que *FC* est meilleur que *MAC* pour les instances de petite taille ($n \leq 6$) mais visite plus de noeuds. Cependant, *MAC* est meilleur que *FC* en temps et en nombre de noeuds visités dès lors que la taille du problème augmente ($n \geq 7$). *MAC* surpasse *FC* dans la plupart des cas pour ce problème.

6.6.2 Les problèmes aléatoires

La seconde classe de problème étudiée correspond aux CSP générés aléatoirement. Nous avons mené des expériences sur des CSP à contraintes d'arité 3 (3-CSP) et 4 (4-CSP) générés aléatoirement. Dans les deux cas, le nombre de variables est 10. Le nombre de valeurs par domaine est de 10 pour les 3-CSP et de 5 pour les 4-CSP. Notre générateur utilise 5 paramètres : n le nombre de variables, d la taille des domaines, a l'arité des contraintes, $dens$ la densité des contraintes qui est le rapport entre le nombre de contraintes et le nombre maximum de contraintes possibles, et t la dureté qui est la proportion de tuples interdits de chaque contraintes. Les CSP obtenus ainsi sont exprimés sous forme CGNF.

La Figure 6.6 (respectivement Figure 6.7) montre les courbes moyennes représentant les temps CPU pour MAC et FC augmentés par l'heuristique MD par rapport à une variation de la dureté des contraintes sur les 3-CSP (respectivement 4-CSP). Les deux courbes de droite de la Figure 6.6 (respectivement Figure 6.7) sont celles de MAC et FC pour une faible densité : *dens*=0,20 (courbes A) (respectivement *dens*=0,096 (courbe D)), les deux au milieu correspondent à une densité moyenne : *dens*=0,50 (courbes B) (respectivement *dens*=0,50 (courbes E)) et les deux de gauches correspondent à une densité haute : *dens*=0,83 (courbes C) (respectivement *dens*=0,96 (courbes F)). On peut voir que le pic de difficulté de la région difficile pour chaque classe de

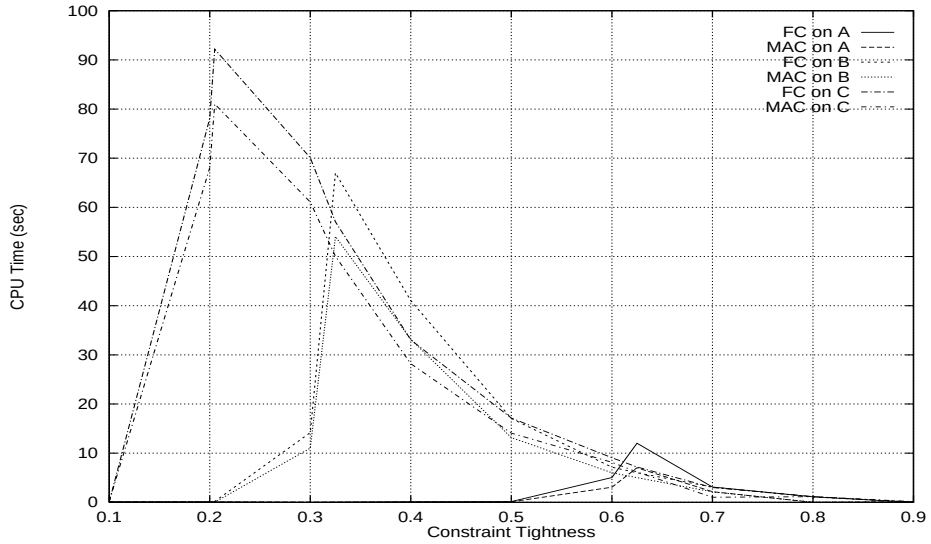


FIG. 6.6 – Résultats de MAC et FC sur des 3-CSP ayant pour densités : dens=0,20, dens=0,50 et dens=0,83.

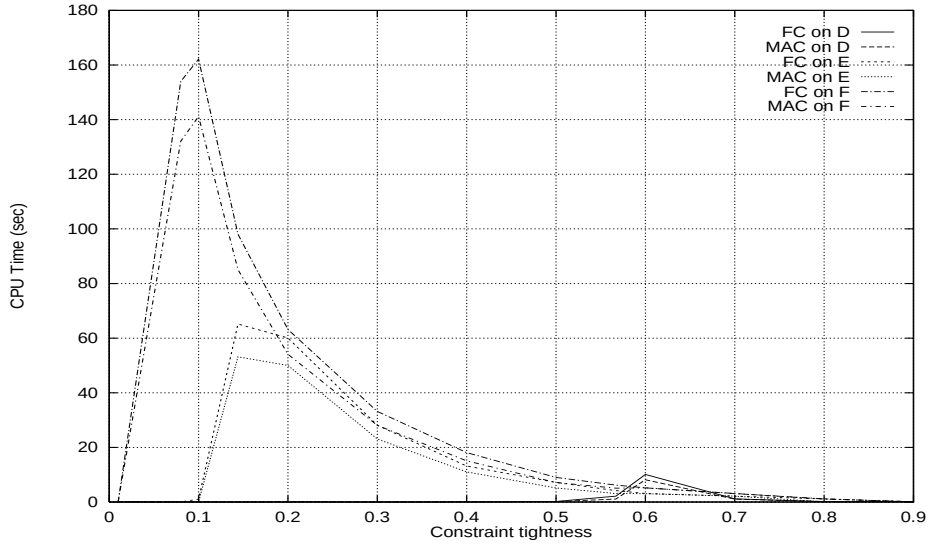


FIG. 6.7 – Résultats de MAC et FC sur des 4-CSP ayant pour densités : dens=0,096, dens=0,50 et dens=0,95.

problème correspond à une valeur critique de dureté et que ce pic de difficulté augmente si la densité augmente. Plus la densité est faible, plus la dureté critique correspondante est grande. On peut également noter que *MAC* surpasse définitivement *FC* sur les six classes de problèmes.

6.7 Établir la consistance de chemin avec IR

Nous décrivons dans cette section un moyen de réaliser la consistance de chemin [Mohr, 1987] pour les CSP **binaires** codés en CGNF en utilisant la règle IR décrite précédemment. Rappelons

que la consistance de chemin est une autre forme de consistance locale qui est plus forte que la consistance d'arc.

Nous ne considérerons dans cette section que le cas de CSP contenant au plus une contrainte par couple de variables. Ceci sans perte de généralité car plusieurs contraintes entre deux variables X et Y peuvent aisément être fusionnées en une seule.

Définition 80 (Chemin consistance)
Dans une CSP binaire, un couple de valeurs $(i \in D_X, j \in D_Y)$, autorisé par une contrainte entre les variables X et Y est chemin consistante si et seulement si pour toute variable Z telle que $Z \neq X$ et $Z \neq Y$, il existe une valeur $k \in D_Z$ telle que l'affectation partielle $(X = i, Y = j, Z = k)$ satisfait toutes les contraintes (binaires) définies sur tout couples de variables formés à partir de $\{X, Y, Z\}$[2].

Proposition 13 *Soit P un CSP binaire exprimé en CGNF. Supposons que P est arc consistant. Si un couple de valeurs (i, j) pour les variables X et Y n'est pas chemin consistant dans P, alors $(Y = j)$ est retirée (c'est à dire que $\neg Y_j$ est inférée par IR) lors de l'établissement de la consistance d'arc avec IR sur $P_{D_X=\{i\}}$, défini comme étant le CSP P dans lequel le domaine de la variable X est réduit à l'unique valeur $i \in D_X$[4]. Toutes les valeurs $j \in D_Y$ telles que (i, j) n'est pas chemin consistant pour certains $i \in D_X$ peuvent être retirées (tous les $\neg Y_j$ peuvent être inférés) avec une complexité en temps de $\mathcal{O}(2.md^2) \approx \mathcal{O}(md^2)$.*

Preuve *Si $(X = i, Y = j)$ n'est pas chemin consistant, alors il existe une variable Z dans P telle que pour toutes les valeurs $k \in D_Z$, l'affectation $(X = i, Y = j, Z = k)$ viole une contrainte de P. Du fait que $(X = i, Y = j)$ est autorisé par la contrainte entre X et Y, cela signifie que chaque valeur k affectée à Z viole soit la contrainte entre X et Z, soit celle entre Y et Z. En d'autres termes, les valeurs du domaine de Z qui sont consistantes avec soit $X = i$ ou $Y = j$ forment une partition $Z_i \cup Z_j$ telles que $Z_i \cap Z_j = \emptyset$, dont les valeurs dans Z_i sont compatibles avec $X = i$ mais pas avec $Y = j$, et celles dans Z_j réciproquement compatibles avec $Y = j$ mais pas avec $X = i$. Ainsi, toutes les valeur de Z_j sont arc inconsistantes dans $P_{D_X=\{i\}}$ (du fait de la contrainte entre X et Z) ; et sont retirées du domaine de Z lors de l'établissement de la consistance d'arc sur $P_{D_X=\{i\}}$ en appliquant IR. Par conséquent, l'affectation $Y = j$ devient à son tour arc inconsistant, car Z_j est le support de $Y = j$ dans $P_{D_X=\{i\}}$, et la valeur j est retirée du domaine D_Y. La complexité en temps pour retirer toutes les valeurs $j \in D_Y$ formant des couples (i, j), $i \in D_X$, qui ne sont pas chemin consistants est identique à la complexité en temps de l'algorithme 17 appliqué sur le CSP binaire $P_{D_X=\{i\}}$ codé en CGNF. Ainsi, la complexité totale en temps de cette opération est en $\mathcal{O}(2md^2) \approx \mathcal{O}(md^2)$ dans le pire des cas.*

Complexité de l'algorithme basique de chemin consistance : La méthode induite par la proposition précédente pour effectuer la chemin consistance sur une instance CGNF codant un CSP binaire est basé sur l'algorithme 17. Elle consiste à lancer l'algorithme Consistance d'arc (algorithme 17) sur le CSP $P_{D_X=\{i\}}$, codé en CGNF, pour chaque variable X et chaque valeur $i \in D_X$. Chaque fois qu'une valeur j pour une variable Y est supprimée par consistance d'arc dans $P_{D_X=\{i\}}$, on peut inférer la clause binaire négative $(\neg X_i \vee \neg Y_j)$ dans P pour signifier que le couple $(X = i, Y = j)$ n'est pas chemin inconsistant. Cette opération est répétée tant ques

[4]Réduire la domaine D_X d'une variable CSP X à une unique valeur $\{i\}$ revient à ajouter la clause unitaire X_i au codage CGNF du CSP.

des couples n'étant pas chemins consistants sont détectés. La complexité en temps de cet algorithme est en $\mathcal{O}(mn^2d^4) \subseteq \mathcal{O}(n^4d^4)$ dans le pire des cas. Le pire des cas étant atteint lorsque le filtrage par chemin consistance retire toutes les couples sauf un, mais seulement un par un ; auquel cas, l'algorithme 17 doit être exécuté une fois pour chaque couple supprimé, c'est-à-dire $\mathcal{O}(n^2d^2)$ fois. Ainsi, la complexité totale est en $\mathcal{O}(mn^2d^4)$. Cette approche pour effectuer un filtrage par chemin consistance n'étant pas spécifique au codage CGNF, cet algorithme basique pourrait être amélioré en considérant des techniques optimisées de filtrage par consistance d'arc [Bessière et Regin, 2001].

6.7.1 Établir la consistance de chemin forte en combinant IR et la règle de résolution généralisée

L'algorithme décrit précédemment permet de générer toutes les paires $(X = i, Y = j)$ qui ne satisfont pas la propriété de consistance de chemin dans un CSP binaire. Ceci se traduit par la production de toutes les clauses binaires correspondantes $(\neg X_i \vee \neg Y_j)$ dans le codage CGNF du CSP. Nous allons maintenant décrire comment ces clauses binaires peuvent être exploitées pour filtrer la formule dans le but de rendre le problème fortement chemin consistant.

Nous rappelons la définition de la chemin consistance forte :

Définition 81 (Chemin consistance forte)
Dans une CSP binaire, un couple de valeurs $(i \in D_X, j \in D_Y)$, autorisé par une contrainte entre les variables X et Y est fortement chemin consistant si et seulement si à la fois $i \in D_X$ et $j \in D_Y$ sont arc-consistants et pour toute variable Z telle que $Z \neq X$ et $Z \neq Y$, il existe une valeur $k \in D_Z$ telle que l'affectation partielle $(X = i, Y = j, Z = k)$ satisfait toutes les contraintes (binaires) définies sur tout couples de variables de $\{X, Y, Z\}^2$.

Supposons qu'une clause binaire $(\neg X_i \vee \neg Y_j)$ ait été inférée car le couple $(X = i, Y = j)$ n'est pas chemin consistant. Cette clause peut être réécrite sous la forme $\neg(X_i \wedge Y_j)$. Ainsi si une clause de contrainte CC_k contient le littéral généralisé $(X_i \wedge Y_j)$ *i.e.* $CC_k = (X_i \wedge Y_j) \vee \ldots$, alors la règle de résolution généralisée peut être appliquée entre CC_k et cette clause binaire et produire la résolvante généralisée CC'_k définie comme étant la clause généralisée CC_k dans laquelle le littéral généralisé $(X_i \wedge Y_j)$ a été retiré. Ceci pourrait être effectué avec une complexité linéaire à chaque fois qu'une clause binaire est générée (il suffit pour ce faire de regarder toutes les clauses généralisées contenant soit X_i, soit Y_j). Filtrer par consistance de chemin forte est sensé réaliser un filtrage de domaines, il ne reste plus qu'à appliquer une nouvelle fois l'algorithme de consistance d'arc à l'aide de IR chaque fois qu'un nouveau couple est detecté comme n'étant pas chemin consistant. Ainsi, la complexité en temps totale pour réaliser un filtrage par consistance de chemin est en $\mathcal{O}(mn^2d^4 + mn^2d^4)$ qui reste en $\mathcal{O}(mn^2d^4)$.

Nous constatons donc que la règle d'inférence IR peut être utilisée non seulement pour effectuer un filtrage par consistance d'arc, mais aussi pour effectuer un filtrage par consistance de chemin sur des CSP binaires codés en CGNF. De plus, lorsque que IR est combinée avec la règle de résolution généralisée, elle permet de réaliser un filtrage par consistance de chemin forte qui est plus fort que d'effectuer un filtrage par consistance d'arc singleton (SAC en anglais pour Singleton Arc Consistency) [Debruyne et Bessière, 1997b], [Prosser et al., 2000] et [Debruyne et Bessière, 2005].

6.8 Travaux de référence, discussion et comparaison

Bien que notre approche ne puisse pas vraiment être qualifiée de transformation ou de codage de CSP vers SAT, puisque nous introduisons un formalisme logique exprimant naturellement un CSP, nous dressons dans cette section un petit récapitulatif des transformations les plus connues de CSP vers SAT tout en comparant leurs propriétés.

- *Le codage direct* [Kleer, 1989] est la transformation de CSP vers SAT la plus utilisée. Pour un CSP P, le codage direct a une complexité en espace dans le pire des cas en $\mathcal{O}(nd + \frac{nd(d-1)}{2} + mad^a)$ et ne capture pas la consistance d'arc.
- *Le codage AC* (ou *Support Encoding*) [Kasif, 1990] est défini pour les CSP binaires. Sa particularité est de représenter la propriété de consistance d'arc dans le codage. Il permet à une procédure simple SAT (la propagation unitaire) d'obtenir la consistance d'arc. Sa complexité en espace dans le pire des cas est la même que celle du codage direct.
- *Le codage k-AC* [Bessière et al., 2003] est une généralisation du codage AC au CSP n-aires. Sa complexité spatiale est identique à celle des deux autres, et légèrement supérieure à celle du codage CGNF qui est en $\mathcal{O}(nd + mad^a)$. Ce codage capture la *k-consistance d'arc relationnelle* [Bessière et al., 2003] et la (i, j)*-consistance* [Freuder, 1985]. L'inconvenient majeur de ce codage réside dans le fait qu'il faut choisir a priori la valeur k, qui est un paramètre de la transformation, sans avoir de connaissance sur la valeur la plus appropriée.
- *Comparaison et discussion* : Notre approche est différente. Elle consiste en une généralisation de la forme CNF qui permet un codage naturel et optimal des CSP n-aires exprimés en extension, tout en conservant leur structure, ainsi que des instances SAT. Dans ce nouveau formalisme, une règle d'inférence simple permet de récupérer la consistance d'arc, la consistance de chemin (dans le cas de CSP binaires) et même la consistance de chemin forte (toujours pour des CSP binaires) en la combinant à la règle de résolution généralisée. Les filtrages associés à ces différentes consistances locales peuvent y être réalisés avec une complexité identique à celle des algorithmes les réalisant dans les CSP.

6.9 Conclusions et perspectives

Nous avons conçu le formalisme GNF (General Normal Form) en généralisant la forme CNF qui nous permet d'exprimer les CSP n-aires de manière compacte lorsque l'on utilise un opérateur de cardinalité (CGNF). Nous avons montré que la taille de la représentation CGNF d'un CSP est identique à la taille de sa représentation originale.

Dans ce nouveau formalisme logique, nous avons montré qu'il est possible d'étendre des techniques de résolution généralement exploitées pour résoudre le problème SAT. En l'occurrence, nous avons généralisé la règle de résolvante de Robinson au formalisme GNF et nous avons montré que la méthode de résolution qui en découle est complète pour les deux formalismes GNF et CGNF. Ceci nous a permis de concevoir une méthode de preuve de consistance ou d'inconsistance pour toute instance exprimée dans un de ces formalisme, notamment lorsqu'il s'agit de CSP exprimés sous forme CGNF. Du moins en théorie, car la complexité de cette méthode ne nous permet pas d'envisager son implémentation en tant que méthode complète dans le cas général.

Cependant, nous avons également montré qu'il est possible d'utiliser une autre méthode de résolution, basée cette fois sur une énumération du genre DPLL pour traiter des instances CGNF. De plus, dans le cas particulier de CSP, nous avons démontré qu'il existe une règle d'inférence

spécifique au codage CGNF, dont l'application jusqu'à saturation permet de filtrer le problème par consistance d'arc. Étant donné que ce procédé peut être réalisé en temps linéaire par rapport à la taille de l'instance, nous avons proposé deux algorithmes de résolution d'instances CGNF en combinant l'adaptation la procédure DPLL et l'utilisation de cette règle. Ces deux algorithmes sont équivalents à deux algorithmes bien connus pour la résolution des CSP : l'algorithme *Forward Checking* et l'algorithme *MAC*.

Afin de s'assurer de la praticabilité de ces deux méthodes, elles ont été expérimentées sur des instances du problème de Ramsey ainsi que sur des CSP à contraintes d'arité 3 et 4 générés aléatoirement. Nous avons obtenu des résultats qui montrent que le maintient de la consistance d'arc complète dans le codage CGNF semble être plus efficace.

Nous avons également montré que cette règle d'inférence pouvait être utilisée de manière différente afin d'effectuer un filtrage par consistance de chemin dans le cas de CSP binaires exprimés sous forme CGNF. Enfin nous avons montré qu'en combinant cette règle d'inférence et la règle de résolution généralisée, nous pouvons réaliser un filtrage équivalent à celui réalisé en rendant un CSP fortement chemin consistant.

Nous avons donc constaté qu'un certain nombre de techniques de résolution et de filtrage par consistances locales généralement utilisées pour traiter des CSP pouvaient être exploitées également dans le formalisme CGNF que nous avons proposés. Il y a fort à parier que tout ce qui concerne le traitement de CSP en extension peut être récupéré par ce formalisme. En ce sens, rien d'étonnant au fait que les expérimentations préliminaires que nous avons menées ne montrent pas de meilleures performances que les méthodes de résolutions spécifiques de CSP (voir même au contraire). Pour pouvoir rivaliser avec les solveurs dernière génération, il faudrait intégrer à nos solveurs des méthodes de restarts, backjumping, nogood recording etc... Nous ne sommes pas à ce stade intéressés par tout ce que nous pourrions récupérer comme techniques d'optimisations existantes pour les CSP mais plutôt par ce que nous apporte le formalisme propositionnel qui n'a pas d'équivalence en CSP. La contribution majeure de ce chapitre est donc a nos yeux la généralisation de la règle de résolution qui permet de déduire de la connaissance qu'il est impossible de déduire dans le formalisme CSP. En effet, la résolvante produite entre deux clauses de contrainte est une clause de contrainte ne correspondant à aucune contrainte dans le CSP initial.

La perspective principale que nous envisageons pour la continuité de ce travail consiste à trouver un moyen d'exploiter cette règle de résolution généralisée, soit en guise de pré-traitement, soit de manière incomplète, en choisissant les clauses généralisées sur lesquelles appliquer la règle de façon heuristique. Cette règle permettant de prouver l'inconsistance lorsque la clause vide est produite, si nous trouvons un moyen de l'exploiter de manière incomplète, nous pourrions ouvrir une voie de recherche pour répondre au cinquième challenge lancé par Bart Selman [Selman et al., 1997].

Conclusion générale

Nous nous sommes intéressés dans cette thèse à trois problèmes majeurs pour lesquels nous avons tenté d'apporter une contribution.

Le premier de ces problèmes concerne le calcul et l'exploitation d'ensembles strong backdoor pour le problème SAT. Le calcul de tels ensembles est un problème à très forte combinatoire qui n'avait jusqu'alors était traité que pour des problèmes de petite taille. Nous avons proposé une méthode de recherche locale permettant de calculer des ensembles strong backdoor dont la taille est la plus petite possible. Cette méthode est capable de traiter des instances contenant plusieurs milliers de variables en un temps paramétrable. Elle est basée sur une procédure calculant le meilleur renommage possible pour une classe polynomiale. Ce problème étant lui aussi NP-difficile, nous avons conçu une méthode incomplète pour chercher des solutions approchées. Nous avons ensuite décrit une méthode complète basée sur l'algorithme DPLL pour résoudre des instances SAT en exploitant les ensembles strong backdoor calculés. L'avantage de cette méthode est de pouvoir raffiner la borne de complexité dans le pire des cas de la résolution.

Nous nous sommes ensuite intéressés aux méthodes incomplètes pour traiter le problème SAT directement. Nous avons proposé une méthode de recherche locale qui a la particularité de n'explorer que des interprétations partielles mais toujours consistantes en lieu et place d'interprétations complètes et inconsistantes comme c'est le cas dans la plupart des méthodes de recherche locale. Nous avons appelé ce solveur CN-SAT pour Consistent Neighborhood for SAT. Dans ce solveur, le choix des mouvements à effectuer à chaque étape est guidé par le nombre de variables interprétées de manière consistante au lieu du nombre de clauses satisfaites. Lors de l'évaluation des voisinages que nous devons effectuer avant de choisir quelle variable interpréter, nous avons été confrontés au problème du calcul d'un transversal minimal pour lequel nous avons proposé deux méthodes de calcul. La première est une méthode exacte de type Branch and Bound et la seconde est de nouveau une méthode approchée. L'implémentation de ces deux méthodes d'évaluation a donné lieu à deux versions différentes de notre solveur.

Enfin, dans un troisième temps, nous nous sommes intéressés à un nouveau formalisme propositionnel qui pourrait tirer avantage de la structure des problèmes. Cette structure étant aisément reconnaissable lorsque les problèmes sont exprimés sous forme CSP , nous avons mis au point un formalisme qui soit capable d'exprimer simplement tout CSP. Le formalisme que nous avons introduit est une généralisation de la forme CNF et porte le nom de formalisme GNF (General Normal Form). Dans un but de concision, nous avons ajouté à ce formalisme un opérateur de cardinalité permettant d'atteindre une représentation des CSP de taille optimale. Nous avons baptisé le formalisme GNF combiné aux formules de cardinalité CGNF pour Cardinality General Normal Form. Nous avons montré que les méthodes de résolutions traditionnelles de SAT pouvaient s'étendre à ce nouveau formalisme plus général. En l'occurrence nous avons montré qu'il est possible de prouver l'inconsistance d'une instance en utilisant une règle de résolution généralisée. Cette

méthode permet la conception d'une méthode originale pour la preuve d'inconsistance lorsque l'on considère des CSP codés sous forme CGNF. De plus, nous avons également montré qu'il est possible de résoudre une instance CGNF en utilisant une méthode énumérative du type DPLL. Enfin, dans le cas de CSP exprimés en CGNF, nous avons montré qu'une règle d'inférence, lorsqu'elle est appliquée jusqu'à saturation, permet de récupérer la propriété de consistance d'arc avec une complexité linéaire dans la taille de la donnée. Une utilisation particulière de cette règle permet également de réaliser un filtrage par consistance de chemin dans le cas de CSP binaires exprimés sous forme CGNF. Ce filtrage est réalisé avec une complexité polynomiale identique à celle des algorithmes connus pour les CSP. Enfin, lorsque cette règle est combinée avec la règle de résolution généralisée, nous avons montré que l'on pouvait effectuer un filtrage par consistance de chemin forte, toujours sur des CSP binaires, avec la même complexité polynomiale.

Ces contributions aux différents problèmes considérés nous permettent de dégager un certain nombre de perspectives à donner à ces travaux.

En ce qui concerne les ensembles strong backdoor, nous avons proposé de calculer des ensembles strong backdoor pour certaines classes polynomiales, à savoir Horn, ordonnée, Horn-renommable et ordonné-renommable. La première intuition que nous avons serait de voir s'il est possible de calculer des ensembles strong backdoor pour d'autres classes polynomiales. Un autre axe de recherche nous semble également pertinent pour ce travail, il s'agit d'étendre ses travaux aux problèmes Max-SAT. Bien qu'aucune classe polynomiale n'ait été identifiée à ce jour, nous pensons qu'il est possible de traiter un ensemble de clauses de Horn plus efficacement qu'un ensemble de clauses non Horn. Il nous semble qu'il est possible d'utiliser les ensembles strong backdoor pour réduire la complexité de la résolution du problème Max-SAT pour un ensemble de clauses de Horn sans pour autant la rendre polynomiale.

Concernant le solveur CN-SAT, outre les optimisations de programmation comme l'intégration de la propagation unitaire, nous envisageons de combiner notre approche avec l'exploitation d'ensembles backbone pour guider la génération des instanciations de départ. Nous envisageons d'autres pistes comme la détection et l'exploitation de nogood et le passage du problème SAT au problème Max-SAT.

Enfin, le dernier axe sur lequel nous pensons qu'il est intéressant de se pencher concerne la méthode de preuve d'inconsistance pour les CSP exprimés en CGNF. La méthode de résolution complète que nous avons présentée n'étant pas exploitable à cause de sa trop forte complexité, nous cherchons un moyen d'intégrer une heuristique de choix des clauses sur lesquelles appliquer le calcul des résolvantes. Si nous parvenons à trouver une telle heuristique, nous pourrions concevoir une méthode incomplète de preuve d'inconsistance, qui est un des challenges actuel en intelligence artificielle.

Une autre voie que nous voulons explorer concerne l'étude des classes polynomiales de ce formalisme. Il serait intéressant de savoir si les classes polynomiales de SAT ou des CSP peuvent être retrouvées dans ce formalisme, et de voir si il n'existe pas de classe polynomiale propre à ce formalisme.

Une amélioration possible de l'efficacité des méthodes énumératives que nous avons proposées pourrait passer par l'intégration de techniques de détection et d'élimination des symétries dans les instances codées en CGNF.

Bibliographie

[Arvind et Biswas, 1987] Arvind, V. et Biswas, S. (1987). An $\mathcal{O}(n^2)$ algorithm for the satisfiability problem of a subset of propositional sentences in CNF that includes all Horn sentences. In *Information Processing Letters*, volume 24, pages 67–69.

[Aspvall, 1980] Aspvall, B. (1980). Recognizing disguised NR(1) instances of the satisfiability problem. *Journal of Algorithms*, 1(1) :97–103.

[Aspvall et al., 1979] Aspvall, B., Plass, M., et Tarjan, R. (1979). A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8 :121–123.

[Audemard et al., 2000] Audemard, G., Benhamou, B., et Siegel, P. (2000). AVAL : An enumerative method for SAT. In *Proceedings of first international conference on computational logic CL00, London*, volume 1861 of *LNCS*, pages 373–383. Springer Verlag.

[Bacchus, 2002] Bacchus, F. (2002). Exploring the computational tradeoff of more reasoning and less searching. In *Proceedings of the Fifth Int. Symp. Theory and Applications of Satisfiability Testing*, pages 7–16.

[Barták et Erben, 2004] Barták, R. et Erben, R. (2004). A new algorithm for singleton arc consistency. In *FLAIRS'04 : Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference*, Miami Beach, Florida. AAAI Press.

[Beldiceanu et Contjean, 1994] Beldiceanu, N. et Contjean, E. (1994). Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12 :97–123.

[Bennaceur, 1996] Bennaceur, H. (1996). The satisfiability problem regarded as constraint satisfaction problem. In *Proceedings of ECAI'96*, pages 155–159.

[Bennaceur, 2004] Bennaceur, H. (2004). A comparison between SAT and CSP techniques. *Constraints*, 9(2) :123–138.

[Benoist et Hébrard, 1999] Benoist, E. et Hébrard, J.-J. (1999). Ordered formulas. In *Les cahiers du GREYC, CNRS - UPRES-A 6072 (search report)*, number 14, Université de Caen - Basse-Normandie.

[Berlandier, 1995] Berlandier, P. (1995). Improving domain filtering using restricted path consistency. In *CAIA '95 : Proceedings of the 11th Conference on Artificial Intelligence for Applications*, page 32, Washington, DC, USA. IEEE Computer Society.

[Bessière, 1994] Bessière, C. (1994). Arc-consistency and arc-consistency again. *Artif. Intell.*, 65(1) :179–190.

[Bessière et Cordier, 1993] Bessière, C. et Cordier, M.-O. (1993). Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, Washington, DC.

[Bessière et al., 1995] Bessière, C., Freuder, E. C., et Régin, J.-C. (1995). Using inference to reduce arc consistency computation. In *Proceedings of IJCAI'95, Montréal, Canada*, pages 592–598.

[Bessière et al., 1999] Bessière, C., Freuder, E. C., et Régin, J.-C. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, pages 125–148.

[Bessière et al., 2006a] Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., et Walsh, T. (2006a). Filtering algorithms for the nvalue constraint. *Constraints*, 11(4) :271–293.

[Bessière et al., 2006b] Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., et Walsh, T. (2006b). The roots constraint. In *CP'06 : 12th International Conference on Principles and Practice of Constraint Programming*, pages 75–90.

[Bessière et al., 2003] Bessière, C., Hebrard, E., et Walsh, T. (2003). Local consistency in SAT. In *International Conference on Theory and Application of satisfiability testing*, pages 400–407.

[Bessière et al., 2002] Bessière, C., Meseguer, P., Freuder, E. C., et J.Larrosa (2002). On forward-checking for non-binary constraint satisfaction. *Journal of Artificial Intelligence*, 141 :205–224.

[Bessière et Regin, 1997] Bessière, C. et Regin, J. (1997). Arc consistency for general constraint networks : preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya, Japan.

[Bessière et Regin, 2001] Bessière, C. et Regin, J. (2001). Refining the basic constraint propagation algorithm. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 309–315.

[Bessière et Régin, 1999] Bessière, C. et Régin, J.-C. (1999). Enforcing arc consistency on global constraints by solving subproblems on the fly. In *CP '99 : Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 103–117, London, UK. Springer-Verlag.

[Boros et al., 1990] Boros, E., Crama, Y., Hammer, P., et Saks, M. (1990). A complexity index of satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1 :21–32.

[Boros et al., 1994] Boros, E., Hammer, P. L., et Sun, X. (1994). Recognition of q-Horn formulae in linear time. *Discrete Applied Mathematics*, 55 :1–13.

[Bryant, 1987] Bryant, R. (1987). Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, pages 677–691.

[Chandru et al., 1990] Chandru, V., Coullard, C., Hammer, P., Montanez, M., et Sun, X. (1990). On renamable Horn and generalized Horn functions. *Annals of Mathematics and Artificial Intelligence*, 1 :33–47.

[Chandru et Hooker, 1991] Chandru, V. et Hooker, J. (1991). Extended Horn sets in propositional logic. *Journal of the ACM*, 38 :205–221.

[Chmeiss et Jégou, 1996] Chmeiss, A. et Jégou, P. (1996). Two new constraint propagation algorithms requiring small space complexity. In *ICTAI '96 : Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI '96)*, page 286, Washington, DC, USA. IEEE Computer Society.

[Conforti et Cornuéjols, 1992] Conforti, M. et Cornuéjols, G. (1992). A class of logical inference problems solvable by linear programming. In *Proceedings of FOCS'92*, volume 33, pages 670–675.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA. ACM Press.

[Creignou et al., 2001] Creignou, N., Khanna, S., et Sudan, M. (2001). *Complexity classifications of boolean constraint satisfaction problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[CSPlib,] CSPlib. CSPLib : Constraint Programming Benchmark Library. http ://www.csplib.org/.

[Dalal, 1992] Dalal, M. (1992). Efficient propositional constraint propagation. In *AAAI'92 : Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 409–414, San-Jose, California (USA).

[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.

[Dalal et Etherington, 1992] Dalal, M. et Etherington, D. W. (1992). A hierarchy of tractable satisfiability problems. *Information Processing Letters.*, 44(4) :173–180.

[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.

[Debruyne, 2000] Debruyne, R. (2000). A property of path inverse consistncy leading to an optimal algorithm. In *ECAI'00 : proceedings of the 14th European Conference on Artificial Intelligence*, pages 88–92, Berlin, Germany.

[Debruyne et Bessière, 1997a] Debruyne, R. et Bessière, C. (1997a). From restricted path consistency to max-restricted path consistency. In *CP'07 : Proceedings of Principles and Practice of Constraint Programming*, pages 312–326, Linz, Austria.

[Debruyne et Bessière, 1997b] Debruyne, R. et Bessière, C. (1997b). Some practicable filtering techniques for the constraint satisfaction problem. *Proceedings of the fifteenth International Joint Conference on Artificial Intelligence IJCAI 1997*, pages 412–417.

[Debruyne et Bessière, 2005] Debruyne, R. et Bessière, C. (2005). Optimal and suboptimal singleton arc consistency algorithms. *Proceedings of the nineteenth International Joint Conference on Artificial Intelligence IJCAI 2005*, pages 54–59.

[Dechter, 1990] Dechter, R. (1990). Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3) :273–312.

[Dequen et Dubois, 2003] Dequen, G. et Dubois, O. (2003). kcnfs : An efficient solver for random K-Sat formulae. In in Computer Sciences 2919, L. N., editor, *SAT'03 : Sixth International Symposium on Theory and Applications of Satisfiability Testing*, pages 486–501.

[Deville et Hentenryck, 1991] Deville, Y. et Hentenryck, P. V. (1991). An efficient arc consistency algorithm for a class of CSP problems. In *Proceedings of the international joint conference on artificial intelligence (IJCAI'91)*, pages 325–330.

[Dimopoulos et Stergiou, 2006] Dimopoulos, Y. et Stergiou, K. (2006). Propagation in CSP and SAT. In *CP'06 : 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 137–151. Springer.

[Dowling et Gallier, 1984] Dowling, W. et Gallier, J. (1984). Linear time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3 :267–284.

[Dubois, 1990] Dubois, O. (1990). On the r,s-SAT satisfiability problem and a conjecture of Tovey. *Discrete Applied Mathematics*, 26 :51–60.

[Dubois et Dequen, 2001] Dubois, O. et Dequen, G. (2001). A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 248–253, Seattle, Washington,.

[Even et al., 1976] Even, S., Itai, A., et Shamir, A. (1976). On the complexity of timetable and integral multi-commodity flow problems. *SIAM J. on Comp*, 5(4) :691–703.

[Freuder, 1978] Freuder, E. C. (1978). Synthesizing constraint expressions. *Commun. ACM*, 21(11) :958–966.

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

[Freuder et Elfe, 1996] Freuder, E. C. et Elfe, C. D. (1996). Neighborhood inverse consistency preprocessing. In *AAAI'96 : Proceedings of the Thirteenth National Conference on Artificial Intelligence*, volume 1, pages 202–208.

[Frost et Dechter, 1994] Frost, D. et Dechter, R. (1994). Dead-end driven learning. In *Proceedings of the Twelfth National Conference of Artificial Intelligence (AAAI'94)*, volume 1, pages 294–300, Seattle, Washington, USA. AAAI Press.

[Galil, 1975] Galil, Z. (1975). On the validity and complexity of bounded resolution. *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 72–82.

[Gallo et Scutellá, 1988] Gallo, G. et Scutellá, M. G. (1988). Polynomially solvable satisfiability problems. In *Information Processing Letters*, volume 29, pages 221–227.

[Gaschnig, 1977] Gaschnig, J. G. (1977). A general backtrack algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 457.

[Gaschnig, 1979] Gaschnig, J. G. (1979). *Performance measurement and analysis of certain search algorithms*. PhD thesis.

[Gavanelli, 2007] Gavanelli, M. (2007). The log-support encoding of CSP into SAT. In Bessiere, C., editor, *CP'07 : The 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 815–822, Berlin Heidelberg. Springer-Verlag. To appear.

[Gelder, 2007] Gelder, A. V. (2007). Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, (to appear). (Preliminary version appeared in COLOR02, held in conjunction with CP02, Ithaca, NY).

[Génisson et Jégou, 1996] Génisson, R. et Jégou, P. (1996). Davis and Putnam were already forward checking. In *ECAI'96 : Proceedings of the 12th European Conference on Artificial Intelligence*, pages 180–184.

[Gent, 2002] Gent, I. P. (2002). Arc consistency in SAT. *ECAI'02 : Proceedings of the 15th European Conference on Artificial Intelligence*, pages 121–125.

[Gent et Underwood, 1997] Gent, I. P. et Underwood, J. L. (1997). The logic of search algorithms : Theory and applications. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 77–91.

[Ghallab et Escalada-Imaz, 1991] Ghallab, M. et Escalada-Imaz, G. (1991). A linear control algorithm for a class of rule-based systems. *Journal of Logic Programming*, 11(2) :117–132.

[Ginsberg, 1993] Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46.

[Glover et Laguna, 1997] Glover, F. et Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers.

[Goldberg et Novikov, 2002] Goldberg, E. et Novikov, Y. (2002). BerkMin : A fast and robust SAT-solver. In *Proceedings of International Conference on Design, Automation, and Test in Europe (DATE '02)*, pages 142–149.

[Grandoni et Italiano, 2003] Grandoni, F. et Italiano, G. F. (2003). Improved algorithms for max-restricted path consistency. In *CP'03 : proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 858–862.

[Grégoire et al., 2005] Grégoire, E., Mazure, B., Ostrowski, R., et Saïs, L. (2005). Automatic extraction of functional dependencies. In *proc. of SAT*, volume 3542 of *LNCS*, pages 122–132.

[Habet et al., 2007] Habet, D., Paris, L., et Benhamou, B. (2007). Consistent Neighborhood for the Satisfiability Problem. In *Proceedings of the 19th International Conference on Tools with Artificial Intelligence (ICTAI'2007)*, pages 497–501, Patras, Greece.

[Han et Lee, 1988] Han, C.-C. et Lee, C.-H. (1988). Comments on mohr and henderson's path consistency algorithm. *Artif. Intell.*, 36(1) :125–130.

[Haralick et Elliott, 1980] Haralick, R. et Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313.

[Hébrard, 1994] Hébrard, J.-J. (1994). A linear algorithm for renaming a set of clauses as a Horn set. In *TCS*, volume 124, pages 343–350.

[Hentenryck et al., 1992] Hentenryck, P. V., Deville, Y., et Teng, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3) :291–321.

[Hirsch, 2000] Hirsch, E. A. (2000). A new algorithm for MAX-2-SAT. *Lecture Notes in Computer Science*, 1770 :65–73.

[Hirsch et Kojevnikov, 2005] Hirsch, E. A. et Kojevnikov, A. (2005). UnitWalk : A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1-4) :91–111.

[Hoos, 1999] Hoos, H. H. (1999). The Run-Time Behaviour of Stochastic Local Search Algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666.

[Hoos et Stützle, 2000] Hoos, H. H. et Stützle, T. (2000). Local Search Algorithms for SAT : An Empirical Evaluation. *J. Autom. Reason.*, 24(4) :421–481.

[Hoos et Stützle, 2004] Hoos, H. H. et Stützle, T. (2004). *Stochastic Local Search : Foundations and applications*. Elsevier / Morgan Kaufmann, San Francisco, CA.

[Iwama, 1989] Iwama, K. (1989). CNF-satisfiability test by counting and polynomial average time. *SIAM Journal on Computing*, 18(2) :385–391.

[Iwama et Miyazaki, 1994] Iwama, K. et Miyazaki, S. (1994). SAT-variable complexity of hard combinatorial problems. In *IFIP World computer congress*, pages 253–258.

[Jussien et Lhomme, 2002] Jussien, N. et Lhomme, O. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1) :21–45.

[Kasif, 1990] Kasif, S. (1990). On the Parallel Complexity of Discrete Relaxation in Constraint Satisafaction Networks. *Journal of Artificial Intelligence*, 45 :275–286.

[Kavvadias et Stavropoulos, 2005] Kavvadias, D. J. et Stavropoulos, E. C. (2005). An efficient algorithm for the transversal hypergraph generation. *J. Graph Algorithms Appl.*, 9(2) :239–264.

[Kilby et al., 2005] Kilby, P., Slaney, J., Thiebaux, S., et Walsh, T. (2005). Backbones and backdoors in satisfiability. In *AAAI'05 : Proceedings of the twenty-second national conference on Artificial intelligence*, pages 1368–1373, Pittsburgh, Pennsylvania.

[Kleer, 1989] Kleer, J. D. (1989). A comparison of ATMS and CSP techniques. In *Proceedings of IJCAI'89*, pages 290–296, DETROIT.

[Knuth, 1990] Knuth, D. E. (1990). Nested satisfiability. In *Acta Informatica*, volume 28, pages 1–6.

[Kondrak et van Beek, 1997] Kondrak, G. et van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *Artif. Intell.*, 89(1-2) :365–387.

[Lardeux et al., 2005] Lardeux, F., Saubion, F., et Hao, J.-K. (2005). Three Truth Values for the SAT and MAX-SAT Problems. In *Proc. of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, Lecture Notes in Computer Science. Springer.

[Lewis, 1978] Lewis, H. (1978). Renaming a set of clauses as a Horn set. *Journal of the Association for Computing Machinery*, 25 :134–135.

[Li, 1999] Li, C. M. (1999). A Constraint-Based Approach Search Trees for Satisfiability. *Information on Processing Letters*, 71(2) :75–80.

[Li et Anbulagan, 1997] Li, C. M. et Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. In *Proceedings of 15th International Joint Conference on Artificial Interlligence (IJCAI'97)*, pages 366–371.

[Li et Huang, 2005] Li, C. M. et Huang, W. Q. (2005). Diversification and determinism in local search for satisfiability. In Bacchus, F. et Walsh, T., editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 158–172. Springer.

[Li et al.,] Li, C. M., Manya, F., et Planes, J. New Inference Rules for Max-SAT. *Journal of Artificial Intelligence Research*, (to appear).

[Lichtenstein, 1982] Lichtenstein, D. (1982). Planar formulae and their uses. *SIAM Journal of Computing*, 11(2) :329–343.

[Luquet, 2000] Luquet, P. (2000). *Horn renommage partiel et littéraux purs dans les formules CNF*. PhD thesis, Université de Caen.

[Mackworth, 1977] Mackworth, A. (1977). Consistency in a network of relations. *AI*, 8(1) :99–118.

[Mackworth et Freuder, 1985] Mackworth, A. K. et Freuder, E. C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artif. Intell.*, 25(1) :65–74.

[Marquis, 1997] Marquis, P. (1997). Communication personnelle.

[Mazure et al., 1997] Mazure, B., Saïs, L., et Grégoire, É. (1997). Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 281–285. AAAI Press.

[McAllester et al., 1997] McAllester, D., Selman, B., et Kautz, H. (1997). Evidence for Invariants in Local Search. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97*, pages 321–326, Providence, Rhode Island. MIT Press.

[Minoux, 1988] Minoux, M. (1988). LTUR : A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29(1) :1–12.

[Mohr, 1987] Mohr, R. (1987). A correct path consistency algorithm and an optimal generalized arc consistency algorithm. *Rapport CRIN*, 87-R-030.

[Mohr et Henderson, 1986] Mohr, R. et Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233.

[Mohr et Masini, 1988] Mohr, R. et Masini, G. (1988). Good old discrete relaxation. In *European Conference on Artificial Intelligence (ECAI'88)*, pages 651–656.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints : Fundamental properties and application to picture processing. *Journal of Information Science*, 9(2) :95–132.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., et Malik, S. (2001). Chaff : Engineering an efficient SAT solver. In *Proceedings of 38th Design Automation Conference (DAC01)*.

[Nishimura et al., 2004] Nishimura, N., Ragde, P., et Szieder, S. (2004). Detecting backdoor sets with respect to horn and binary clauses. In *7th International Conference on theory and Application of Satisfiability Testing*.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational complexity*. Addison–Wesley.

[Paris, 2006] Paris, L. (2006). Calcul et exploitation d'ensembles Horn Strong Backdoor. In *Actes de la quatrième Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (MajecStic'06)*, page 8, Lorient, France.

[Paris et al., 2005] Paris, L., Benhamou, B., et Siegel, P. (2005). Un cadre théorique et pratique commun aux formalismes SAT et CSP n-aires. In *7e Rencontres Jeunes Chercheurs en Intelligence Artificielle (RJCIA'05)*, pages 169–182.

[Paris et al., 2006a] Paris, L., Benhamou, B., et Siegel, P. (2006a). A Boolean Encoding encluding SAT and n-ary CSPs. In Euzenat, J. et (Eds), J. D., editors, *Proceedings of The Twelfth International Conference on Artificial Intelligence : Methodology, Systems, Applications (AIMSA 2006)*, Springer LNAI 4183, pages 33–43, Varna, Bulgaria.

[Paris et al., 2007a] Paris, L., Habet, D., et Benhamou, B. (2007a). Voisinage consistant pour le problème de satisfaisabilité. In *Actes des troisièmes Journées Francophones de Programmation par Contraintes (JFPC'2007)*, pages 59–67, Rocquencourt, France.

[Paris et al., 2006b] Paris, L., Ostrowski, R., Saïs, L., et Siegel, P. (2006b). Approximation d'ensembles Horn strong backdoor par recherche locale. In *Actes des deuxièmes Journées Francophones de Programmation par Contraintes (JFPC'2006)*, pages 277–284, Nîmes, France.

[Paris et al., 2007b] Paris, L., Ostrowski, R., et Siegel, P. (2007b). Des ensembles Horn strong backdoor aux ensembles ordonné strong backdoor. In *Actes des troisièmes Journées Francophones de Programmation par Contraintes (JFPC'2007)*, pages 48–57, Rocquencourt, France.

[Paris et al., 2006c] Paris, L., Ostrowski, R., Siegel, P., et Saïs, L. (2006c). Computing and exploiting Horn strong backdoor sets thanks to local search. In *Proceedings of the 18th International Conference on Tools with Artificial Intelligence (ICTAI'2006)*, pages 139–143, Washington DC, United States.

[Paris et al., 2007c] Paris, L., Ostrowski, R., Siegel, P., et Saïs, L. (2007c). From Horn Strong Backdoor Sets to Ordered Strong Backdoor Sets. In Gelbukh, A. et Morales, A. K., editors, *Proceedings of the 6th Mexican International Conference on Artificial Intelligence (MICAI'07)*, volume 4827 of *LNAI*, pages 105–117, Aguascalientes, Mexico. Springer-Verlag.

[Perlin, 1992] Perlin, M. (1992). Arc consistency for factorable relations. *Artificial Intelligence*, 53(2-3) :329–342.

[Pham et al., 2007] Pham, D. N., Thornton, J., et Sattar, A. (2007). Building Structure into Local Search for SAT. In *Proceedings of IJCAI'07*, pages 2359–2364, Hyderabad, India.

[Prosser, 1993] Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3) :268–299.

[Prosser et al., 2000] Prosser, P., Stergiou, K., et Walsh, T. (2000). Singleton consistencies. *Proceedings of the sixth International Conference on Principles and Practice of Constraint Programming CP'00*, pages 353–368.

[Ramalhinho-Lourenço et al., 2000] Ramalhinho-Lourenço, H., Martin, O. C., et Stützle, T. (2000). Iterated Local Search. *International Series in Operations Research & Management Science*, 57 :321–353.

[Rauzy, 1995] Rauzy, A. (1995). Polynomial restrictions of SAT : What can be done with an efficient implementation of the Davis and Putnam's procedure. In *International Conference on Principle of Constraint Programming, CP'95*, volume 976, pages 515–532, Springer Verlag.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In *AAAI '94 : Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA. American Association for Artificial Intelligence.

[Régin, 1996] Régin, J.-C. (1996). Generalized arc consistency for global cardinality constraint. In *AAAI '96 : Proceedings of the thirteenth national conference on Artificial intelligence*, pages 209–215.

[Régin, 1999] Régin, J.-C. (1999). The symmetric alldiff constraint. In *IJCAI '99 : Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 420–425, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Régin et Puget, 1997] Régin, J.-C. et Puget, J.-F. (1997). A filtering algorithm for global sequencing constraints. In *CP'97 : Third International Conference on Principles and Practice of Constraint Programming*, pages 32–46.

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1) :23–41.

[Rossa, 1993] Rossa, P. (1993). Formules bien imbriquées : reconnaissance et satisfaisabilité. Mémoire de DEA, Université de Caen, Laboratoire d'Informatique.

[Sabin et Freuder, 1997] Sabin, D. et Freuder, E. (1997). Understanding and improving the mac algorithm. In *Proceeding of CP'97*, pages 167–181.

[Schlipf et al., 1995] Schlipf, J., Annexstein, F., Franco, J., et Swaminathan, R. (1995). On finding solutions to extended Horn formulas. *Information Processing Letters*, 54 :133–137.

[Scutella, 1990] Scutella, M. G. (1990). A note on dowling and gallier's topdown algorithm for propositional Horn satisfiability. *Journal of Logic Programming*, 8 :265–273.

[Selman et Kautz, 1993] Selman, B. et Kautz, H. A. (1993). An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence(AAAI-93)*, Washington DC.

[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., 1997] Selman, B., Kautz, H. A., et McAllester, D. A. (1997). Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54.

[Selman et al., 1992a] Selman, B., Levesque, H., et Mitchell, D. (1992a). A new method for solving hard satisfiability problems. In *AAAI'92 : Proceedings of the Tenth National Conference on Artificial Intelligence)*, pages 459–465.

[Selman et al., 1992b] Selman, B., Levesque, H. J., et Mitchell, D. (1992b). A New Method for Solving Hard Satisfiability Problems. In Rosenbloom, P. et Szolovits, P., editors, *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92*, pages 440–446, Menlo Park, California.

[Siegel, 1987] Siegel, P. (1987). *Représentation et Utilisation de la connaissance en calcul propositionnel*. Thèse d'état, Université de Provence, GIA–Luminy, Marseille (France).

[Silva et Sakallah, 1996] Silva, J. P. M. et Sakallah, K. A. (1996). GRASP : A new search algorithm for satisfiability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, Washington. IEEE Computer Society Press.

[Singh, 1995] Singh, M. (1995). Path consistency revisited. In *ICTAI '95 : Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, page 318, Washington, DC, USA. IEEE Computer Society.

[Smyth et al., 2003] Smyth, K., Hoos, H. H., et Stützle, T. (2003). Iterated Robust Tabu Search for MAX-SAT. In *Canadian Conference on AI*, pages 129–144.

[Taillard, 1991] Taillard, E. (1991). Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17(4-5) :443–455.

[Tarjan, 1972] Tarjan, R. E. (1972). Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1 :146–160.

[Tovey, 1984] Tovey, C. A. (1984). A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8 :85–89.

[van Beek et Dechter, 1995] van Beek, P. et Dechter, R. (1995). On the minimality and global consistency of row-convex constraint networks. *J. ACM*, 42(3) :543–561.

[Vasquez et al., 2005] Vasquez, M., Dupont, A., et Habet, D. (2005). Consistent Neighborhood in a Tabu Search. In *Metaheuristics : Progress as real Problem Sovers, MIC 2003 Post-conference volume*. Kluwer Academic Publishers.

[Verfaillie et al., 1999] Verfaillie, G., Martinez, D., et Bessière, C. (1999). A generic customizable framework for inverse local consistency. In *AAAI '99/IAAI '99 : Proceedings of the sixteenth*

national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence, pages 169–174, Menlo Park, CA, USA. American Association for Artificial Intelligence.

[Walsh, 2000a] Walsh, T. (2000a). Reformulating propositional satisfiability as constraint satisfaction. In *SARA '00 : Proceedings of the 4th International Symposium on Abstraction, Reformulation, and Approximation*, pages 233–246, London, UK. Springer-Verlag.

[Walsh, 2000b] Walsh, T. (2000b). SAT v CSP. In *Proceedings of CP'00*, pages 441–456.

[Williams et al., 2003a] Williams, R., Gomes, C., et Selman, B. (2003a). Backdoors to typical case complexity. In *Procceding of Intenrational Joint Conference on Artificial Intelligence (IJCAI'2003)*.

[Williams et al., 2003b] Williams, R., Gomes, C., et Selman, B. (2003b). On the connections between heavy-tails, backdoors, and restarts in combinatorial search. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'2003)*.

[Yamasaki et Doshita, 1983] Yamasaki, S. et Doshita, S. (1983). The satisfiability problem for a class consisting of Horn sentences and some non-Horn sentences in proportional logic. In *Information and Control*, volume 59, pages 1–12.

Abstract

This thesis is divided in three parts, for which the common goal is the resolution of NP-complete problems.

The first part concerns the identification and the exploitation of hidden structures in SAT problems. In particular, we focus on a structure called strong backdoor sets which are strongly linked to the intrinsic difficulty of SAT problems. A strong backdoor set is a set of variables such that for any assignment of its variables, the simplified formula belongs to a tractable class. We propose a method which allows to compute strong backdoor sets for many well known tractable classes (Horn, renameable-Horn, Ordered, renameable-Ordered) for which the size is as small as possible. This approximation is realized in two steps. First we compute the best renaming of the formula according to a tractable class from the point of view of the size of the non-tractable part of the renamed formula (either in terms of number of clauses, or in terms of number of literals). Then a strong backdoor set is extracted from the non-tractable part of the renamed formula.

The second part concerns our contribution to local search methods for SAT. We propose a new local search method which deals with incomplete but always consistent assignments, instead of complete and inconsistent ones. This method tries to extend the current partial assignment in the same way as a complete method would. But instead of backtracking when a conflict arises, it frees at least one variable per falsified clause to restore consistency. Thus the explored neighborhood is always consistent, whereas it is not the case for other classical local search algorithms. The experimental results obtained show the competitiveness of our method compared to other local search methods.

Lastly, the third part concerns a theoretical and practical framework common to SAT and n-ary CSPs formalisms. We propose and study a generalization of the CNF form called GNF. In order to represent n-ary CSPs in a concise way in this logical formalism, we add a cardinality operator to the GNF formalism, and get the CGNF formalism. We show that the size of the encodings of any instance in both CSP and CGNF are identical. We introduce an inference rule, as well as a generalized resolution rule for this formalism. The inference rule allows to enforce some kind local consistencies when treating CSPs expressed in this new formalism, whereas the generalized resolution rule allows to design a method to prove inconsistency of CGNF instances.

Key words: SAT, Strong Backdoor Sets, Consistent Neighborhood, Local Search, CSP, Local Consistencies.

Résumé

Cette thèse comporte trois grandes parties, dont le point commun est la résolution de problèmes NP-complets.

La première de ces parties concerne l'identification et l'exploitation de structures cachées dans un problème SAT. En particulier, nous nous intéressons à une structure appelée ensembles strong backdoor qui sont fortement liés à la difficulté intrinsèque des problèmes SAT. Un ensemble strong backdoor est un ensemble de variables tel que pour toute interprétation de ces variables la formule simplifiée appartient à une classe polynomiale. Nous proposons une méthode permettant de calculer des ensembles strong backdoor pour plusieurs classes polynomiales connues (Horn, ordonnée, Horn-renommable et ordonnée-renommable) dont la taille est la plus petite possible. Cette approximation est réalisée en deux étapes. Dans un premier temps, nous calculons le meilleur renommage de la formule pour une classe polynomiale du point de vue de la taille de la partie non polynomiale (soit en terme de nombre de clauses, soit en terme de nombre de littéraux). Ensuite nous extrayons un ensemble strong backdoor de la partie non polynomiale de la formule renommée.

La seconde partie est une contribution aux méthodes de recherche locale pour le problème de SAT. Nous proposons une nouvelle méthode de recherche locale qui gère des interprétations incomplètes, mais toujours consistantes, au lieu de complètes et inconsistantes. Cette méthode tente de prolonger l'interprétation partielle courante comme le ferait une méthode complète. Cependant, au lieu de déclencher un retour arrière (*backtrack*) lorsqu'un conflit survient, elle libère au moins une variable impliquée dans chaque clause falsifiée afin de restaurer la consistance. Ainsi, le voisinage exploré est toujours consistant alors que ce n'est pas le cas pour les algorithmes de recherche locale classiques. Les résultats expérimentaux montrent la compétitivité de notre méthode par rapport à d'autres méthodes de recherche locale.

Enfin, la troisième partie concerne un cadre théorique et pratique commun au formalisme SAT et CSP n-aires. Nous proposons et étudions une généralisation de la forme *CNF*, appelée GNF. Afin de pouvoir représenter de CSP n-aires de manière concise dans ce formalisme logique, nous rajoutons un opérateur de cardinalité au formalisme GNF, et obtenons le formalisme CGNF. Nous montrons que la taille des représentations CSP et CGNF d'un problème sont identiques. Nous introduisons une règle d'inférence, ainsi qu'une règle de résolution généralisée pour ce formalisme. La règle d'inférence permet de récupérer un certain nombre de consistances locales lorsque nous traitons des CSP exprimés dans ce nouveau formalisme. La règle de résolution généralisée quant à elle nous permet de concevoir une méthode de preuve de l'inconsistance d'une instance CGNF.

Mots clés : SAT, ensembles strong backdoor, voisinage consistant, recherche locale, CSP, consistances partielles.

Laboratoire des Sciences de l'Information et des Systèmes (LSIS) - UMR CNRS 6168