Détection de fonctions booléennes pour la preuve d'inconsistance

Richard Ostrowski[1] **Lionel Paris**[2]

[1] Université de Provence (Aix-Marseille 1)
[2] Université Paul Cézanne (Aix-Marseille 3)
LSIS UMR 6168
{richard.ostrowski,lionel.paris}@lsis.org

Résumé

En 1997, B. Selmam et H. Kautz ont proposé une série de 10 challenges. L'un d'entre eux était l'élaboration d'une méthode de recherche locale pour la preuve d'inconsistance d'une formule (challenge 5). Dix ans plus tard, seules quelques pistes ont été explorées. Dans cet article, nous proposons un algorithme en deux phases pour prouver l'insatisfaisabilité de formules sous forme CNF. La première étape consiste à détecter différentes fonctions booléennes (équivalences, portes «et») de manière incrémentale. Ensuite nous utilisons les propriétés de ces différentes portes afin de prouver l'inconsistance des formules (un algorithme polynomial pour les équivalences, des règles de production pour les portes «et»). Nous montrons que cette technique offre de bons résultats par rapport aux approches existantes.

Abstract

In 1997, B. Selman and H. Kautz proposed a series of 10 challenges. One of them concerned the design of a practical stochastic local search procedure for proving unsatisfiability (Challenge 5). Today, more than 10 years later, only few attempts were led to address this challenge, in spite of the great number of incomplete methods for proving satisfiability. In this paper, we propose a two steps algorithm for proving unsatisfiability of CNF formulas. The first step consists in detecting $\Leftrightarrow$-gates greedily. At the same time, a polynomial algorithm is used to eventually prove the unsatisfiability of the extracted set of $\Leftrightarrow$-gates. We show that this method outperforms the existing ones on some classes of instances. This method is then extended to others logical functions ($\wedge$-gates & $\vee$-gates).

1 Introduction

Ces dernières décennies, de nombreuses méthodes incomplètes variées ont été conçues dans le domaine de la programmation par contraintes. Certains d'entre eux traitent des problèmes de décision (problèmes NP-complets), d'autres de problèmes d'optimisation (problèmes NP-difficiles). On peut citer GSAT, Novelty, Novelty+, R-Novelty, R-Novelty+, Adapt-Novelty... (voir [10] pour un état de l'art détaillé). Elles peuvent aussi être utilisées pour extraire des structures particulières des instances, comme les ensembles Strong Backdoor [13] ou les MUS [8] (tout deux des problèmes NP-difficiles). Toutes ces méthodes ont l'avantage d'être efficace en pratique et sont, dans certains cas, le seul moyen de traiter les problèmes les plus gros et les plus difficiles.

Cependant, une infime minorité d'entre elles sont conçues pour prouver l'insatisfaisabilité (problème Co-NP), en dépit du challenge essentiel que cela représente. Un challenge allant dans ce sens fut proposé en 1997 par Selman et Kauz [17] : challenge 5 : "Design a practical stochastic local search procedure for proving unsatisfiability". Dans ce challenge il est question de concevoir une méthode de recherche locale stochastique pour prouver l'inconsistance. Il fut soumis à la communauté et était supposé durer 5 - 10 ans. Mais 12 ans plus tard, aucune réponse satisfaisante à ce challenge n'a encore été proposée, qu'il s'agisse de méthode de recherche locale ou de méthode incomplète en général.

Dans la communauté CSP, quelques tentatives ont été proposées. Elles sont pour la plupart basées sur des propriétés de coloration de la micro-structure des problèmes [4], sur l'utilisation de filtrage par consistances partielles

diverses (consistance d'arc, de chemin, consistance d'arc singleton, k-consistance...) ou basées sur l'exploitation de symétries et de propriété de dominance [3]. D'un autre côté, dans la communauté SAT, les seuls tentatives sont basées sur l'utilisation restreinte de la règle des résolvantes ou des règles de productions permettant de déduire la clause vide (GUNSAT [1], RANGER [15]). Malheureusement, toutes ces méthodes souffrent du passage à l'échelle. Même si certaine d'entre elle se comporte relativement bien sur de petites instances, leurs performances se détériorent de manière dramatique lorsque la taille des instances croît.

Dans ce travail, nous présentons une nouvelle méthode incomplète pour prouver l'inconsistance d'une formule SAT sous forme normal conjonctive (CNF). Alors que la plupart des méthodes déjà proposées utilisent largement la règle de résolution de Robinson [16] ou la règle d'hyper résolution binaire [2], notre méthode comporte deux étapes. La première consiste à détecter puis extraire des fonctions booléennes dans la formule CNF des instances SAT. Ces fonctions (ou portes) sont de la forme $y = f(x_1, x_2, \ldots, x_{n-1})$ (où f = $\Leftrightarrow$ dans un premier temps). Ce processus d'extraction est très coûteux en temps ($O(2^{n-1}$ pour une fonction $y = f(x_1, x_2, \ldots, x_{n-1}))$, c'est pourquoi seulement une sélection de portes sont recherchées. Dans la seconde étape, lorsque des fonctions ont été identifiées, un processus de méta-raisonnement sur l'ensemble de ces fonctions est utilisé pour détecter une éventuelle contradiction. Ce processus est réalisé en temps polynômial par rapport à la taille de l'ensemble des fonctions et est complet sur ce dernier. Cette nouvelle méthode diffère complètement des approches précédemment proposées dans le sens où tout le raisonnement fait sur les portes-$\Leftrightarrow$ est fait en espace polynômial. Il n'y a aucun risque d'explosion combinatoire du nombre de portes, contrairement à l'utilisation de la règle des résolvantes.

Nous avons ensuite étendu ce mode d'extraction et d'exploitation à deux autres fonctions booléennes, les portes-$\wedge$ et les portes-$\vee$. Dans ce cas, le processus de raisonnement sur un ensemble de ces fonctions n'est pas complet. Mais nous avons exhibé quelques propriétés qui de temps en temps (relativement souvent) permettent de détecter l'insatisfaisabilité ou de déduire des littéraux à propager avec une complexité en temps et en espace linéaire.

Nous montrons que les méthodes que nous proposons se comportent très bien par rapport aux méthodes existantes pour prouver l'insatisfaisabilité.

Le papier est organisé comme suit : tout d'abord, nous rappelons les définitions basiques et les notations du problème SAT et des méthodes de déduction. Ensuite nous présentons les deux étapes de notre méthode avec les portes-$\Leftrightarrow$. D'un côté, nous présentons la politique de choix des portes à chercher que nous avons utilisé, ainsi que leur extraction. D'un autre côté, nous exposons l'étape de raisonnement sur l'ensemble des portes extraites. Nous montrons dans la section suivante l'extension de cette méthode aux portes-$\wedge$ et portes-$\vee$. Dans la section d'après, nous décrivons les résultats expérimentaux que nous avons obtenus en utilisant notre approche face à la meilleure des approches existante (GUNSAT). Finalement, une conclusion et des extensions possibles de ce travail sont présentés.

2 Définitions préliminaires

Dans cette section nous rappelons les définitions basiques du problème SAT et les notations que nous utiliserons. Nous donnons aussi des notions de déduction logique (règles d'inférence) qui sont généralement utilisées pour tester l'insatisfaisabilité. Ceci nous permettra de mettre en évidence les différences entre notre méthode et les méthodes existantes.

2.1 Le problème du test de satisfaisabilité (SAT)

Soit $\mathcal{B}$ un langage propositionnel (*i.e.* booléen) composé de formules construites de façon standard, en utilisant les connecteurs usuels ($\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) et un ensemble de variables propositionnelles. Une *formule CNF* Σ est un ensemble, (interprété comme une conjonction) de *clauses*, où une clause est un ensemble (interprété comme une disjonction) de *littéraux*. Un littéral est une occurrence positive ou négative d'une variable propositionnelle. On représente par $\sim l$ le littéral complémentaire d'un littéral l (si l est un littéral positif d'une variable v, $\sim l = \neg v$, sinon $\sim l = v$).

Rappelons que toute formule booléenne peut être réécrite sous la forme d'une CNF en utilisant le codage de Tseitin [18]. La taille d'une formule CNF Σ est définie par $\sum_{c \in \Sigma} |c|$ où $|c|$ est le nombre de littéraux de la clause c. Une clause *unitaire* (resp. *binaire*) est une clause de taille 1 (resp. 2). Un *monolittéral* est l'unique littéral d'une clause unitaire. On note $nbVar(\Sigma)$ (resp. $nbCla(\Sigma)$) le nombre de variables (resp. clauses) de Σ. $\mathcal{V}(\Sigma)$ (resp. $\mathcal{L}(\Sigma)$) est l'ensemble de variables (resp. littéraux) apparaissant dans Σ. L'ensemble $\mathcal{L}(\Sigma)$ est l'union des littéraux positifs $\mathcal{L}^+(\Sigma)$ et des littéraux négatifs $\mathcal{L}^-(\Sigma)$. Un ensemble de littéraux $S \subset \mathcal{L}(\Sigma)$ est consistant si et seulement si $\forall l \in S, \neg l \notin S$.

L'*interprétation* d'une formule booléenne est l'affectation de valeur de vérité $\{vrai, faux\}$ à ses variables. Un littéral l est satisfait (resp. falsifié) par I si l est positif et $I[l] = vrai$ ou l est négatif et $I[l] = faux$ (resp. l est négatif et $I[l] = vrai$ ou l est positif et $I[l] = faux$). Un *modèle* d'une formule est une interprétation qui satisfait la formule. Usuellement, le problème de décision associé au problème SAT consiste à déterminer si une formule

sous forme CNF admet une solution. Ce problème est NP-complet. Dans ce travail, nous étudions le problème complémentaire du problème SAT, qui consiste à déterminer si une formule CNF n'admet aucun modèle. Ce problème est Co-NP.

2.2 Notion de déduction

Lorsque l'on traite l'insatisfaisabilité en logique propositionnelle, l'utilisation de règles d'inférence est presque obligatoire. L'application d'une règle d'inférence sur un ensemble S de clauses produit une nouvelle clause qui ne change pas la satisfaisabilité de l'ensemble original. Si la clause produite est vide ($S \vdash \bot$), alors l'ensemble de clauses est insatisfaisable.

Une des règles d'inférence la plus connue est la règle des résolvantes de Robinson, appelée aussi règle de résolution [16]. Cette règle permet d'inférer une nouvelle clause, appelée résolvante, à partir de deux clauses initiales sous certaines conditions.

Définition 1 (Règle de résolution de Robinson) *Soit $c_1 = (x_1 \vee x_2 \vee \ldots \vee x_n)$ et $c_2 = (\neg x_1 \vee y_2 \vee \ldots \vee y_m)$ 2 clauses. La clause $R = (x_2 \vee \ldots \vee x_n \vee y_2 \vee \ldots \vee y_m)$ est la résolvante obtenue par application de la règle de résolution sur x_1 et $\neg x_1$ (entre c_1 et c_2) : $c_1, c_2 \vdash_{\text{Rob}} R$.*

Si un ensemble de clauses est insatisfaisable, l'application de cette règle un nombre illimité de fois sur cet ensemble de clauses (incluant les clauses produites) jusqu'à saturation est une méthode complète pour prouver l'insatisfaisabilité. Cela signifie que la clause vide sera toujours produite.

L'inconvénient majeur du procédé associé à l'utilisation répétée de la règle de résolution de Robinson est sa complexité spatiale. En fait, la taille des clauses produites ne peut être bornée (sauf par le nombre total de variables de la formule), et leur nombre peut être exponentiel avant de produire la clause vide. C'est la raison pour laquelle plusieurs travaux ont été entrepris dans le but d'affaiblir cette règle pour avoir quelques garanties sur les complexités en temps et en espace. Il en résulte la perte de la complétude. On peut citer la résolution étendue [18], la résolution directionnelle [5] ou l'hyper-résolution binaire [2] par exemple. Les deux solveurs GUNSAT [1] et RANGER [15] sont basés sur différentes combinaisons de ces règles.

Dans notre méthode, nous n'utilisons aucune de ces règles d'inférence. Nous utilisons une seule règle, très simple, basée sur la propagation unitaire pour déduire des clauses. En fait, quand nous voulons tester la présence d'une clause, nous exploitons la propriété suivante :

Propriété 1 *Soit S un ensemble de clauses et $c = l_1 \vee l_2 \vee \ldots \vee l_n$ une clause. $S \vdash c$ si et seulement si $S \wedge \neg c \vdash \bot$. i.e. $S \vdash l_1 \vee l_2 \vee \ldots \vee l_n$ si et seulement si $S \wedge \neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n \vdash \bot$.*

Mais nous n'utilisons que la propagation unitaire (PU) pour tester la satisfaisabilité de l'ensemble $S \wedge \neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n$. ($S \wedge \neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n \vdash_{\text{PU}} \bot$) qui peut être réalisée en temps linéaire et espace constant.

2.3 Fonctions booléennes

Dans ce papier, nous nous intéressons principalement à 3 fonctions booléennes : portes-$\Leftrightarrow$, portes-$\wedge$ et portes-$\vee$.

Définition 2 (Fonctions booléennes (ou portes)) *On appelle fonction booléenne (ou porte) une formule propositionnelle notée $y = f(x_1, x_2, \ldots, x_{n-1})$, où f est un des connecteurs appartenant à $\{\Leftrightarrow, \oplus, \wedge, \vee\}$ et y et x_i sont des littéraux.*

Définition 3 (Littéraux d'entrées et de sortie) *Dans une porte écrite sous la forme $y = f(x_1, x_2, \ldots, x_{n-1})$, on dit que y est le littéral de sortie de la porte et que tous les x_i sont les littéraux d'entrées de la porte.*

3 Prouver l'insatisfaisabilité avec les portes-$\Leftrightarrow$

Nous décrivons dans cette section un algorithme en deux phases. Étant donné une formule Σ, la première étape consiste à identifier les portes-$\Leftrightarrow$. La seconde consiste à raisonner sur l'ensemble des portes extraites lors de la première phase. Mais commençons par la définition précise des portes-$\Leftrightarrow$.

3.1 Définition

Une porte-$\Leftrightarrow$ $l_1 =\Leftrightarrow (l_2, l_3, \ldots, l_n)$ peut être réécrite comme $(l_1 \Leftrightarrow l_2 \Leftrightarrow \ldots \Leftrightarrow l_n)$. Cette porte, appelée aussi chaîne d'équivalences, furent introduites et étudiées dans [6]. Une porte-$\Leftrightarrow$ de longueur n peut être codée de manière unique par un ensemble de clauses contenant 2^{n-1} clauses de longueur n.

Exemple 1 *Représentation clausale d'une porte-$\Leftrightarrow$ de longueur 3 :*
$l_1 =\Leftrightarrow (l_2, l_3)$ ($l_1 \Leftrightarrow l_2 \Leftrightarrow l_3$) est codée par :
- $l_1 \vee l_2 \vee l_3$
- $l_1 \vee \neg l_2 \vee \neg l_3$
- $\neg l_1 \vee l_2 \vee \neg l_3$
- $\neg l_1 \vee \neg l_2 \vee l_3$

Ces fonctions booléennes possèdent des propriétés très intéressantes :

Propriété 2

$$l_1 =\Leftrightarrow (l_2, l_3) \text{ est équivalent à } l_2 =\Leftrightarrow (l_3, l_1) \quad (1)$$

$$\neg l_1 =\Leftrightarrow (\neg l_2, \neg l_3) \text{ est équivalent à } \neg l_1 =\Leftrightarrow (l_2, l_3) \quad (2)$$

$$\neg l_1 =\Leftrightarrow (l_2, l_3) \text{ est équivalent à } l_1 =\Leftrightarrow (l_2, \neg l_3) \quad (3)$$

$$l_1 =\Leftrightarrow (l_2), l_3 =\Leftrightarrow (l_1, l_4) \text{ est remplacé par } l_3 =\Leftrightarrow (l_2, l_4) \quad (4)$$

$$l_1 =\Leftrightarrow (l_2, l_2, l_3) \text{ est équivalent à } l_1 =\Leftrightarrow (l_3) \quad (5)$$

$$l_1 =\Leftrightarrow (\neg l_1) \vdash \bot \quad (6)$$

Description :

1. *L'opérateur $\Leftrightarrow$ est commutatif et associatif.*
2. *Les couples de négations peuvent être supprimés, ainsi, toute chaîne est équivalente à une chaîne contenant au plus une négation.*
3. *La négation éventuelle peut être placée sur n'importe quel littéral.*
4. *On peut remplacer un littéral d'entrée d'une fonction par tous les littéraux d'entrées d'une autre fonction dont ce littéral est littéral de sortie.*
5. *Les paires d'occurrences d'un même littéral peuvent être supprimées.*
6. *Une chaîne d'équivalences binaire contenant un littéral et son opposé est contradictoire.*

Si une instance ne contient que des portes-$\Leftrightarrow$, elle peut être résolue en temps polynomiale. Certains solveurs savent exploiter ces portes-$\Leftrightarrow$ pour effectuer des simplifications comme pré-traitements.(lsat [12],march_dl [9],eqsatz [11]).

Remarque 1 *Les portes-$\Leftrightarrow$ sont similaires aux fonctions XOR (portes-$\oplus$) du fait de ces propriétés :*

- $l_1 \Leftrightarrow l_2 \Leftrightarrow ... \Leftrightarrow l_n = \neg l_1 \oplus l_2 \oplus ... \oplus l_n$ *ssi* $n = 2*k$, $k \in \mathbb{N}$.
  En d'autres termes, $l_1 =\Leftrightarrow (l_2, l_3, \ldots, l_n)$ *est équivalent à* $\neg l_1 = \oplus(l_2, l_3, \ldots, l_n)$ *ssi* $n = 2*k, k \in \mathbb{N}$
- $l_1 \Leftrightarrow l_2 \Leftrightarrow ... \Leftrightarrow l_n = l_1 \oplus l_2 \oplus ... \oplus l_n$
  ssi $n = 2*k+1, k \in \mathbb{N}$.
  En d'autres termes, $l_1 =\Leftrightarrow (l_2, l_3, \ldots, l_n)$ *est équivalent à* $l_1 = \oplus(l_2, l_3, \ldots, l_n)$ *ssi* $n = 2*k+1, k \in \mathbb{N}$

3.2 Extraction des portes-$\Leftrightarrow$

Il y a plusieurs moyens de détecter les portes-$\Leftrightarrow$. Celui que nous avons retenu commence par choisir un ensemble de variables S de longueur arbitraire suivant une heuristique donnée. Ensuite il énumère toutes les interprétations possibles sur cet ensemble de variables. Pour chaque interprétation, il tente de produire une clause par propagation unitaire en utilisant la propriété 1. *i.e.* $\Sigma, \neg S \vdash_{\text{PU}} \bot$? À chaque étape, un compteur pour chaque porte possible est maintenu (il n'y en a que 2, la chaîne positive($y =\Leftrightarrow (x_1..x_n)$) et celle négative ($\neg y =\Leftrightarrow (x_1..x_n)$)). Selon la clause qui est produite, l'un ou l'autre de ces compteurs est incrémenté (en fonction du nombre de littéraux négatifs) dans le but de compter le nombre de clauses appartenant aux définitions possibles qui peuvent être produites. Si un de ces compteurs est égal à la moitié du nombre d'interprétations possibles, alors la porte-$\Leftrightarrow$ est détectée. Sinon, si une clause ne peut être dérivée par propagation unitaire, toutes les autres clauses appartenant à la même définition de la porte ne seront pas testées.

Cette procédure de détection est répétée jusqu'à ce qu'un nombre de tentatives (nbTentatives) est atteint. Notre algorithme est incrémental. Il commence à chercher des portes de longueur 2. S'il n'en trouve aucune alors il en cherche avec des longueurs de plus en plus grandes jusqu'à atteindre une longueur maximale (longueurMax). Si pour une longueur n donnée, des portes sont identifiées, la seconde phase est effectuée avant de continuer avec une longueur supérieure. La seconde phase consiste à déterminer si l'ensemble des portes détectées est satisfaisable ou pas.

L'algorithme 1 montre le procédé global.

Algorithm 1: Detection-portes-$\Leftrightarrow$(entrée : une CNF Σ, `nbTentatives`,V,`longueurMax`, sortie : $vrai$ si des portes contradictoires ont pu être produites, $faux$ sinon)

```
1  n ← 2
2  portes-∧ ← {∅} ; /* initialize an empty set of gates */
3  tant que (n ≠ longueurMax) faire
4      essai ← 0
5      tant que (essai ≠ nbTentatives) faire
6          S ← sousEnsemble(V,n)
7          ClausesAvecNbLittNegImpairs ← 0 ; /* Nombre de
           clauses contenant un nombre impair de
           littéraux négatifs                          */
8          ClausesAvecNbLittNegPairs ← 0 ; /* Nombre de clauses
           contenant un nombre pair de littéraux
           négatifs                                    */
9          pour tous les (interprétations I possibles sur S) faire
10             si (Σ ∧ I ⊢^BCP ⊥) alors
11                 si (NbLittNeg(I)% 2) = 0 ) alors
12                     ClausesAvecNbLittNegPairs ←
                       ClausesAvecNbLittNegPairs +1
13                 sinon
14                     ClausesAvecNbLittNegImpairs ←
                       ClausesAvecNbLittNegImpairs +1
15             si (ClausesAvecNbLittNegPairs=n × 2^(n-1)) alors
16                 ajouter(portes-⇔,O,n)
17             si (ClausesAvecNbLittNegImpairs=n × 2^(n-1)) alors
18                 ajouter(portes-⇔,1,n)
19         essai ← essai+1
20     si testSatisfaisabilité(portes-⇔) = vrai alors retourner vrai sinon
       n ← n + 1
21 retourner Faux ;
```

la fonction *testSatisfaisabilité(portes-$\Leftrightarrow$)* est décrite dans la section suivante.

3.3 Raisonner avec les portes-$\Leftrightarrow$

Comme il a été dit précédemment, il existe un algorithme de complexité polynomiale pour tester la satisfaisabilité d'un ensemble de portes-$\Leftrightarrow$ [6]. Le processus, qui correspond à la fonction *testSatisfaisabilité(portes-$\Leftrightarrow$)* de l'algorithme 1 est très simple. Tout d'abords, il faut déplacer l'éventuelle négation de chaque chaîne sur le littéral de sortie, de telle sorte qu'il ne reste plus que des portes-$\Leftrightarrow$ avec des littéraux d'entrée positifs (en utilisant les propriétés 2.2 et 2.3. Ensuite, en utilisant la propriété 2.4, on choisit une porte ayant un littéral positif en entrée puis on injecte sa définition (les littéraux d'entrées) dans toutes les autres portes contenant ce littéral en entrée. On supprime la porte choisie. Puis on simplifie toutes les nouvelles portes obtenues en utilisant les propriétés 2.5 et 2.6. Si $\perp$ est produite, alors l'ensemble de portes-$\Leftrightarrow$ est insatisfaisable (et donc toute la formule de départ), sinon on recommence avec une autre porte et un autre littéral jusqu'à ce que chaque littéral ai été traité. Si $\perp$ n'est jamais produite, alors l'ensemble est satisfaisable.

4 Prouver l'insatisfaisabilité avec les portes-$\wedge$

Nous allons voir dans cette partie comment implémenter le même type d'algorithme afin de détecter des portes $\wedge$ et $\vee$ (portes-$\wedge$ et portes-$\vee$).

4.1 Définitions

Tout comme les portes-$\Leftrightarrow$, les portes-$\wedge$ ont une représentation clausale minimale. Pour une porte de taille n, le nombre minimum de clauses pour représenter une telle fonction est composée d'une clause de taille n contenant le littéral de sortie et les littéraux complémentaires des littéraux d'entrées et $n-1$ clauses binaires contenant le littéral complémentaire du littéral de sortie et d'un des $n-1$ littéraux d'entrés.

Exemple 2 *Représentation clausale minimale d'une porte-$\wedge$ de taille 3 :*
$l_1 = \wedge(l_2, l_3)$ *représenté par :*
- $l_1 \vee \sim l_2 \vee \sim l_3$
- $\sim l_1 \vee l_2$
- $\sim l_1 \vee l_3$

Il y a une dualité entre les portes-$\wedge$ et les portes-$\vee$:

Proposition 1

$$l_s = \wedge(l_{l1}, ..., l_n) = \neg\neg(l_s = \wedge(l_1, ..., l_n)) = \neg(\neg l_s \neq \vee(\neg l_1, ..., \neg l_n))$$

Dans ce cas, si nous traitons les portes-$\wedge$, il n'est pas nécessaire de traiter les portes-$\vee$ car nous ne pourrons pas inférer plus de choses.

4.2 Extraction des portes-$\wedge$

La première étape, tout comme pour les portes-$\Leftrightarrow$, consiste à identifier les clauses appartenant à la définition de la porte considérée. Malheureusement, l'utilisation de la propagation unitaire, nous permettra d'identifier ces clauses que si elles apparaissent sous la forme minimale (une clause de taille n et $n-1$ clauses binaires). Ce que nous voulons ici, c'est de pouvoir retrouver des portes-$\wedge$ qui ne sont pas explicitement présentes dans la formule comme celles détectées dans [7]. Nous devons, pour cela, connaitre le nombre exact de clauses appartenant à la définition. Pour une porte-$\wedge$ comportant $n-1$ littéraux d'entrées, 2^{n-1} clauses sont nécessaires. (Ceci s'explique par la dualité entre les portes-$\wedge$ et portes-$\vee$). Toutes ces clauses sont de taille n (le littéral de sortie et les $n-1$ littéraux d'entrés). Il est possible de retrouver la forme minimale en utilisant la règle de Résolution 1 sur les $2^{n-1}-1$ clauses.

Exemple 3 *Forme clausale étendue d'une porte-$\wedge$ de taille 3 :*
$l_1 = \wedge(l_2, l_3)$ *est représenté par :*
- $l_1 \vee \sim l_2 \vee \sim l_3$
- $\sim l_1 \vee l_2 \vee \sim l_3$
- $\sim l_1 \vee l_2 \vee l_3$
- $\sim l_1 \vee \sim l_2 \vee l_3$

Etant donné un ensemble de variables, combien de définitions différentes de portes-$\wedge$ sont possibles ?

Proposition 2 *Pour n variables, il y a exactement $n.2^n$ définitions différentes de portes-$\wedge$.*

Preuve 1 *Pour chaque variable $V_i, i \in \{1, \ldots, n\}$ nous associons son littéral positif (l_i) et son littéral négatif ($\neg l_i$). Si nous considérons le littéral de sortie (positif), il est possible d'avoir toutes les combinaisons différentes de définitions construites sur les $n-1$ littéraux d'entrés. Nous en avons exactement le même nombre si l'on considère le littéral de sortie (négatif). Donc pour une variable de sortie $V_i, i \in \{1, \ldots, n\}$, nous avons exactement 2^n définitions. Enfin, si nous considérons toutes les variables de sortie (n), nous obtenons $n * 2^n$ définitions possibles.*

Proposition 3 *Etant donné une clause $c = l_1 \vee \cdots \vee l_n$, le nombre de portes-$\wedge$ contenant la clause c dans leur définition est égal à $n * 2^{n-1}$.*

Preuve 2 *Soit $nb(i)$ le nombre de portes-$\wedge$ contenant la clause numéro i dans leur définition. On remarque que la somme de tous ces nombres pour chaque clause ($\sum_{i=1}^{2^n} nb(i)$) est égal au nombre possible de portes-$\wedge$ sur n littéraux ($n.2^{n-1}$), multiplié par le nombre de clauses de leur définition (2^n). Nous avons donc $\sum_{i=1}^{2^n} nb(i) = n.2^{n-1}.2^n$. Maintenant, nous remarquons que chaque clause apparait dans le même nombre de définitions de*

portes-$\wedge$, donc $nb(i)$ est le même pour chaque clause et donc $\sum_{i=1}^{2^n} nb(i) = 2^n.nb(i)$. On en conclut que, $nb(i) = n.2^{n-1}$.

Le processus d'extraction des portes-$\wedge$ est similaire à celui des portes-$\Leftrightarrow$, mais en considérant la représentation sous forme clausale étendue des définitions des portes-$\wedge$. Il n'y a que deux différences :

1. L'énumération de toutes les affectations possibles, sur les n variables, ne peut pas être interrompue si une clause ne peut pas être inférée par propagation unitaire (car nous devons tester l'existence des $n.2^n$ portes possibles).
2. Lorsque nous avons testé toutes les possibilités, nous devons extraire toutes les portes-$\wedge$ possibles ($n.2^n$), contrairement aux portes-$\Leftrightarrow$ (deux définitions possibles).

L'algorithme 2 montre le processus d'extraction des portes-$\wedge$. Dans cet algorithme, le tableau `nbClauses` va comptabiliser le nombre de clauses identifiées pour chaque définition possible de portes-$\wedge$ sur l'ensemble $\mathcal{S}$ de variables. A chaque fois qu'une clause peut être inférée par PU dans Σ ($\Sigma \wedge \mathcal{I} \vdash^{\text{UP}} \bot$), la fonction `mettreAJour` incrémente de un les définitions de portes-$\wedge$ concernées.

Algorithm 2: Detection-portes-$\wedge$(entrée : Σ, `nbtentatives`,V,`longueurMax`, sortie : $vrai$ si des portes contradictoires ont pu être produites, $faux$ sinon)

```
1  n ← 2
2  portes-∧ ← {∅} ; /* initialisation de l'ensemble des
   portes à vide                                          */
3  tant que (n ≠ longueurMax) faire
4      essai ← 0
5      tant que (essai ≠ nbtentatives) faire
6          S ← sous-ensemble(V,n)
7          nbClauses[n × 2^n] ← [0,...,0]; /* nbClauses est un
           tableau de taille (n × 2^n) rempli de 0        */
8          pour tous les (interprétations I possibles sur S) faire
9              si (Σ ∧ I ⊢^UP ⊥) alors mettreAJour(I,nbClauses)
10         pour tous les (i dans 1,...,n × 2^n) faire
11             if (nbClauses[i]=n × 2^(n-1)) then add(∧-gate,i)
12         essai ← essai+1
13     ensembleMonoLit ← SimplifierPortes(portes-∧)
14     si (ensembleMonoLit ≠ ∅) alors
15         Σ ← UP(Σ ∪ ensembleMonoLit)
16         si (Σ = ∅) alors
17             retourner faux ; /* Σ est satisfaisable    */
18         sinon si (□ ∈ Σ) alors
19             retourner vrai ; /* Σ est insatisfaisable  */
20         sinon
21             n ← 2
22     sinon n ← n + 1
23 retourner faux
```

4.3 Raisonner avec les portes-$\wedge$

La seconde étape consiste à simplifier ces portes afin de trouver une contradiction. Malheureusement, il n'existe pas d'algorithme polynomial afin de tester la satisfaisabilité d'un ensemble de portes-$\wedge$. Cependant, certaines propriétés peuvent être utilisées afin de simplifier cet ensemble.

Propriété 3

$$l_s = \wedge(l_1, \ldots, l_k), l_s = \wedge(l_{k+1}, \ldots, l_n) \vdash l_s = \wedge(l_1, \ldots, l_k, l_{k+1}, \ldots, l_n) \quad (7)$$

$$l_s = \wedge(l_1, \ldots, l_k, \neg l_k) \vdash \neg l_s \quad (8)$$

$$l_s = \wedge(\neg l_s, l_1, \ldots, l_k) \vdash \neg l_s \quad (9)$$

$$l_s = \wedge(l_1, \ldots, l_k), l_s \vdash l_1, \ldots, l_k \quad (10)$$

$$l_s = \wedge(l_1, \ldots, l_k, l_{k+1}, \ldots, l_n), \neg l_s = \wedge(l_1, \ldots, l_k, l'_{k+1}, \ldots, l'_n) \vdash l_1, \ldots, l_k \quad (11)$$

$$l_s = \wedge(l_1, \ldots, l_k), \neg l_s = \wedge(l_1, \ldots, l_k) \vdash \bot \quad (12)$$

$$l_s = \wedge(l_1, \ldots, l_k), \neg l_s = \wedge(l_1, \ldots, l_k, l_{k+1}) \vdash l_s, l_1, \ldots, l_k, \neg l_{k+1} \quad (13)$$

Preuve 3
- *(7) : propriété de fusion. C'est une règle d'inférence. Tout modèle de la partie gauche est un modèle de la partie droite.*
- *(8) : En considérant la représentation clausale minimale, nous avons les deux clauses $\neg l_s \vee \neg l_k$ et $\neg l_s \vee l_k$. Par résolution, nous obtenons $\neg l_s$.*
- *(9) : En considérant la représentation clausale minimale, nous avons la clause $\neg l_s \vee \neg l_s$.*
- *(10) : Comme l_s est à vrai, nous avons $l_1, \ldots, l_k$ qui sont à vrai.*
- *(11) : Si l_s est à vrai alors $l_1, \ldots, l_k, l_{k+1}, \ldots, l_n$ sont déduits, si l_s est à faux, alors $l_1, \ldots, l_k, l'_{k+1}, \ldots, l'_n$ sont aussi déduits.*
- *(12) : Avec (10), on infère $l_1, \ldots, l_k$ qui falsifie la seconde porte.*
- *(13) : Avec (10), on infère $l_1, \ldots, l_k$ et ensuite l_s. Pour satisfaire la seconde porte, nous devons avoir $\neg l_{k+1}$.*

5 Expérimentations

Les expérimentations ont été réalisées sur des Pentium IV 3.2GHz avec 1 GB de RAM sous linux. Pour chaque problème nous avons comparé notre méthode avec GUNSAT[1] [1].

Nous avons mené deux types d'expérimentations. Nous avons d'abord fait des tests sur des problèmes insatisfaisables générés aléatoirement et ensuite sur des problèmes insatisfaisables structurés issus de la satlib [2].

[1] http ://www.lri.fr/~simon/research/gunsat/gunsat-V1.jar
[2] *http ://www.cs.ubc.ca/~hoos/SATLIB/benchm.html*

Instance	$\Leftrightarrow$			$\wedge$			les deux				GUNSAT	
	%S	# $\Leftrightarrow$	T(s)	%S	#$\wedge$	T(s)	%S	# $\Leftrightarrow$	#$\wedge$	T(s)	%S	T(s)
ssa	25	498	36	25	4415	0.3	50	593	268	0.1	–	–
pret	100	281	0	0	0	0	95	1372	441	2	0	0
jnh	100	70	3.1	100	1913	2.7	89.5	719	273	0.35	57	8.48
dubois	92	217	0	0	0	0	74.6	1051	330	0.69	0	0
aim-50	37.5	3.6	0.07	88.24	11098	9.2	79	741	252	0.35	100	1.41
aim-100	21.4	2	10.81	14.29	172	9.74	80	1089	302	1	55	11.93
aim-200	0	0	0	0	0	0	0	0	0	0	60	63

TAB. 1 – Résultats de notre méthode sur des instances structurées. Chaque ligne du tableau correspond à une classe d'instance dont le nom est donné dans la colonne Instance. La colonne %S donne le % d'instances résolues. La colonne $\Leftrightarrow$ (resp. $\wedge$) donne le nombre moyen de portes-$\Leftrightarrow$ (resp. portes-$\wedge$) détectées pendant la recherche. Enfin, la colonne T(s) donne le temps moyen de résolution en secondes. Chaque problème a été testé 10 fois.

Nous avons évalué trois versions différentes de notre méthode. La première, correspondant à la colonne $\Leftrightarrow$ dans le tableau 1 et 2, est l'implantation de la méthode décrite dans 3 (portes-$\Leftrightarrow$). La seconde, correspondant à la colonne $\wedge$ des tableaux 1 et 2, est l'implantation de la méthode décrite dans 4 (portes-$\wedge$). La dernière, correspondant à la colonne *les deux* des deux tableaux, est l'implantation des deux méthodes.

L'heuristique utilisée pour choisir les n variables, pour chacune des méthodes, afin de détecter des portes, fonctionne comme suit : si l'on recherche une porte de longueur n, et qu'il existe des clauses de taille n alors l'ensemble des variables de cette clause est considéré. S'il n'en existe pas ou si toutes les clauses de la taille considérée ont été testées, alors l'heuristique génère aléatoirement un ensemble de n variables parmi $10.n$ variables ayant le plus grand nombre d'occurrences dans la formule. Ceci est effectué 1000 fois.

Nous remarquons que sur les instances aléatoires ainsi que les structurés, notre méthode est meilleure que GUNSAT et que la combinaison de détection des portes-$\Leftrightarrow$ et portes-$\wedge$ donne de bons résultats.

6 Conclusion et perspectives

Dans cet article, nous avons vu comment utiliser la propagation unitaire afin de déduire des clauses d'une formule Σ. La détection de ces clauses permet d'identifier des fonctions booléennes. Après avoir identifié un certain nombre de portes (portes-$\Leftrightarrow$ et portes-$\wedge$), nous avons vu comment les utiliser afin de développer une méthode incomplète en deux phases pour la preuve de l'inconsistance. Les résultats obtenus sont très encourageants. Notre méthode se montre plus performante que GUNSAT sur de nombreuses classes d'instances. Ceci montre encore l'importance d'exploiter la structure cachée des différents problèmes. Ces bons résultats mettent en valeur les travaux de Pham *et al.* [14]. Dans ces travaux, les auteurs exploitent la structure des problèmes dans une méthode de recherche locale. Cette méthode était la première capable de résoudre les problèmes parity-32.

Ces travaux ouvrent de nombreuses perspectives. D'abord, il serait intéressant de voir s'il est possible de trouver d'autres règles d'inférences concernant les différentes portes afin de simplifier la partie fonctionnelle. Les propriétés proposées, concernant les portes-$\wedge$ permettent de produire de nombreux mono-littéraux. Il serait intéressant de voir si une telle détection et simplification peut être utilisée comme pré-traitement. Enfin, sur les portes-$\Leftrightarrow$, il existe aussi des propriétés qui ne sont pas utilisés afin de produire des mono-littéraux.

Références

[1] Gilles Audemard and Laurent Simon. Gunsat : a greedy local search algorithm for unsatisfiability. In *International Joint Conference on Artificial Intelligence(IJCAI'07)*, pages 2256–2261, jan 2007.

[2] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[3] Belaïd Benhamou and Mohamed Réda Saïdi. A new incomplete method for csp inconsistency checking. In *Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*, Chicago, Illinois, USA, juillet 2008. AAAI Press. à paraître.

[4] J.N. Bès and P. Jégou. Proving graph un-colorability with a consistency check of csp. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 693–694, Hong Kong, China, novembre 2005. IEEE. Poster.

Aléatoire		$\Leftrightarrow$		$\wedge$		les deux		GUNSAT	
$nbVar$	$nbCla/nbVar$	%S	T(s)	%S	T(s)	%S	T(s)	%S	T(s)
50	4.25	100	0.25	99	0.26	99	0.26	58	60
50	5	100	0.12	100	0.11	100	0.11	86	18
50	6	100	0.05	100	0.04	100	0.04	97	5
60	4.25	100	0.95	100	1.46	100	1.52	35	126
60	5	100	0.23	100	0.28	99	0.28	68	67
60	6	100	0.1	100	0.1	100	0.1	92	16
70	4.25	100	4.77	99	16	99	15.4	23	189
70	5	100	0.54	99	0.9	99	0.88	51	187
70	6	100	0.16	100	0.22	100	0.21	87	59
80	4.25	100	13.6	52.5	58.4	88.5	70	–	–
80	5	100	1.49	100	3.56	100	3.48	–	–
80	6	100	0.32	100	0.57	99	0.56	79	146

TAB. 2 – Résultats de notre méthode sur les instances aléatoires (clauses de longueur 3). Chaque ligne représente une catégorie d'instance caractérisée par son nombre de variables et de clauses. Chaque catégorie contient 20 problèmes et chaque problème est testé 10 fois. La colonne %S est le pourcentage d'instances résolues et la colonne T(s) donne le temps moyen de résolution en seconde.

[5] Rina Dechter and Irina Rish. Directional resolution : The davis-putnam procedure, revisited. In *In Proceedings of KR-94*, pages 134–145. Morgan Kaufmann, 1994.

[6] B. Dunham and H. Wang. Towards feasible solutions of the tautology problem. *Annals of Math. Log.*, 10 :117–154, 1976.

[7] É. Grégoire, B. Mazure, R. Ostrowski, and L. Saïs. Automatic extraction of functional dependencies. In *proc. of SAT*, volume 3542 of *LNCS*, pages 122–132, 2005.

[8] Eric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of muses. *Constraints Journal*, 12(3) :325–344, 2007.

[9] Marijn Heule, Joris van Zwieten, Mark Dufour, and Hans van Maaren. March_eq : Implementing additional reasoning into an efficient lookahead sat solver. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, pages 345–359, 2004.

[10] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search : Foundations and applications*. Elsevier / Morgan Kaufmann, San Francisco, CA, 2004.

[11] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *proceedings of Conference on Artificial Intelligence AAAI*, pages 291–296, Austin, USA, 2000. AAAI Press.

[12] R. Ostrowski, E. Grégoire, B. Mazure, and L. Saïs. Recovering and exploiting structural knowledge from CNF formulas. In *CP'02*, pages 185–199, 2002.

[13] Lionel Paris, Richard Ostrowski, Pierre Siegel, and Lakhdar Saïs. 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, 2006.

[14] Duc Nghia Pham, John Thornton, and Abdul Sattar. Building Structure into Local Search for SAT. In *Proceedings of IJCAI'07*, pages 2359–2364, Hyderabad, India, January 2007.

[15] Steven Prestwich and Inês Lynce. Local search for unsatisfiability. In *In Proceedings of SAT*, pages 283–296. Springer, 2006.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1) :23–41, January 1965.

[17] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54, 1997.

[18] G.S. Tseitin. On the complexity of derivations in the propositional calculus. In H.A.O. Slesenko, editor, *Structures in Constructives Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.