

Extraction automatique de dépendances fonctionnelles

Éric Grégoire Richard Ostrowski Bertrand Mazure Lakhdar Saïs

CRIL CNRS & IRCICA – Université d'Artois

rue Jean Souvraz SP-18

F-62307 Lens Cedex France

email: {gregoire,ostrowski,mazure,sais}@cril.univ-artois.fr

Résumé

Dans ce papier, nous proposons une nouvelle technique d'extraction de dépendances fonctionnelles dans les formules booléennes. Nous utilisons de manière originale la technique de propagation de contraintes en prétraitement, qui nous permet d'extraire plus de fonctions booléennes et de variables dépendantes pouvant se cacher dans la CNF que les approches précédente sur de nombreuses classes d'instances.

Mots clés : SAT, fonction booléenne, raisonnement propositionnel et recherche.

1 Introduction

Les progrès effectués ces dernières années dans la résolution pratique du problème SAT permettent d'envisager aujourd'hui la résolution d'instances issues d'applications réelles (voir par ex. [Ole00, GMTZ01, ZMMM01]). Alors qu'il reste une forte compétition dans le but d'implémenter des solveurs efficaces pour résoudre les instances k -SAT aléatoires [DD01], il y a aussi un réel intérêt pour l'implémentation de systèmes performants capables de résoudre des instances difficiles de grandes tailles issues du monde réel. De nombreuses instances sont proposées et des compétitions internationales (e.g. la compétition DIMACS en 1993, la compétition Beijing en 1996, les compétitions SAT 2001-2003) sont organisées autour de ces instances structurées.

Il reste que la représentation sous forme normale conjonctive (CNF) de problèmes issus d'applications réelles ne permettent pas une représentation simple et compréhensible des connaissances, alors qu'à l'inverse la logique propositionnelle lorsqu'elle est utilisée dans sa totalité permet de représenter simplement une grande variété de connaissances. Cette structure peut s'avérer utile pour la résolution [RSB99, KMS97].

Dans ce papier, nous proposons un nouveau prétraitement pour la résolution d'instances SAT, qui extrait et exploite certaines structures cachées dans la CNF. Cette technique se base sur une utilisation originale de la procédure de propagation de contraintes

(BCP). Alors que BCP est souvent utilisée pour produire des littéraux impliqués ou équivalents, nous montrons qu'il est possible d'étendre son utilité dans le but de nous donner une formule hybride avec une partie clausale et une partie constitué de formules de la forme $y = f(x_1, \dots, x_n)$ où f est un opérateur parmi $\{\vee, \wedge\}$ avec y et x_i représentant les variables booléennes de la formule initiale. Ces fonctions nous permettent de détecter un sous ensemble de variables indépendantes qui peuvent être exploités par des solveurs SAT.

Ce papier étend de manière significative des résultats préliminaires de [ORL02] dans le sens où plus de fonctions et plus de variables dépendantes sont détectées dans de nombreuses instances. Malheureusement dans cet ensemble de dépendances fonctionnelles, des cycles peuvent apparaître, et la détection d'un ensemble coupe cycle minimal est un problème NP-dur. Néanmoins, nous utilisons des heuristiques efficaces pour les supprimer. Cela nous permet de séparer les variables entre variables dépendantes et indépendantes.

Le papier est organisé de la manière suivante. Après quelques définitions préliminaires, nous présentons les portes booléennes et leur propriétés. Nous montrons ensuite comment il es possible de détecter plus de fonctions booléennes que dans [ORL02] en utilisant la procédure de propagation de contraintes. Puis, nous présentons une méthode permettant de nous fournir un ensemble de variables dépendantes, dans le but de réduire l'espace de recherche. Nous présentons quelques résultats expérimentaux montrant l'intérêt de notre approche. Finalement quelques pistes de recherche sont avancées en conclusion.

2 Définitions préliminaires

Soit \mathcal{B} un langage booléen (i.e. propositionnel) de formules, 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 variable propositionnelle positive ou négative. On note $\mathcal{V}(\Sigma)$ (resp. $\mathcal{L}(\Sigma)$) l'ensemble des variables (resp. littéraux) apparaissant dans Σ . Une *clause unitaire* est une clause formée d'un unique littéral. Un *littéral unitaire* est l'unique littéral d'une clause unitaire.

En plus de ces notations, nous définissons la négation d'un ensemble de littéraux ($\neg\{l_1, \dots, l_n\}$) comme l'ensemble correspondant aux littéraux opposés ($\{\neg l_1, \dots, \neg l_n\}$).

Une *interprétation* d'une formule booléenne est une affectation des valeurs $\{vrai, faux\}$ de ces variables. Un *modèle* d'une formule CNF est une interprétation qui satisfait la formule. Le problème SAT revient à trouver un modèle d'une formule CNF s'il en existe un ou a prouver qu'il n'en existe pas.

Soit c_1 une clause contenant un littéral a et c_2 une clause contenant le littéral opposé $\neg a$, la *résolvante* de c_1 avec c_2 est la disjonction de tous les littéraux de c_1 et c_2 privé de a et $\neg a$. Une résolvante est dites *tautologique* si elle contient des littéraux opposés.

Rappelons qu'une formule booléenne peut être transformé de manière équivalente en formule CNF en temps linéaire (par addition de variables propositionnelles). La plupart des algorithmes de résolution travaillent sur la forme clausale où la structure du problème se retrouve caché. Par la suite une formule CNF sera représenté par un ensemble de portes booléennes.

3 Portes booléennes

Une *porte (booléenne)* est une expression de la forme $y = f(x_1, \dots, x_n)$, où f est un opérateur parmi $\{\vee, \wedge, \Leftrightarrow\}$ et où y et x_i sont des littéraux propositionnels, défini de la manière suivante :

- $y = \wedge(x_1, \dots, x_n)$ représente l'ensemble de clauses $\{y \vee \neg x_1 \vee \dots \vee \neg x_n, \neg y \vee x_1, \dots, \neg y \vee x_n\}$, traduisant le fait que la valeur de y est entièrement déterminée par les valeurs de x_i s.t. $i \in [1..n]$;
- $y = \vee(x_1, \dots, x_n)$ représente l'ensemble de clauses $\{\neg y \vee x_1 \vee \dots \vee x_n, y \vee \neg x_1, \dots, y \vee \neg x_n\}$;
- $y = \Leftrightarrow(x_1, \dots, x_n)$ représente la *chaîne d'équivalences* (aussi appelé *formule biconditionnelle*) $y \Leftrightarrow x_1 \Leftrightarrow \dots \Leftrightarrow x_n$, qui est équivalent à 2^n clauses.

Par la suite, nous considérons les portes de la forme $y = f(x_1, \dots, x_n)$ où y est une variable ou la constante booléenne *vrai*.

En effet, chaque clause peut être représentée par une porte de la forme *vrai* = $\vee(x_1, \dots, x_n)$. De plus, une porte $\neg y = \wedge(x_1, \dots, x_n)$ (resp. $\neg y = \vee(x_1, \dots, x_n)$) est équivalente à $y = \vee(\neg x_1, \dots, \neg x_n)$ (resp. $y = \wedge(\neg x_1, \dots, \neg x_n)$). De part la propriété sur les chaînes d'équivalences, disant que chaque chaîne d'équivalences avec un nombre impair de négations (resp. pair) est équivalent à la chaîne formé des même littéraux mais tous positifs (resp. excepté un), chaque porte de la forme $y = \Leftrightarrow(x_1, \dots, x_n)$ peut toujours être réécrite en une porte où y est un littéral positif. Par exemple, $\neg y = \Leftrightarrow(x_1, x_2, x_3)$ est équivalent à $y = \Leftrightarrow(x_1, x_2, x_3)$ et $\neg y = \Leftrightarrow(\neg x_1, x_2, \neg x_3)$ est équivalent par exemple à $y = \Leftrightarrow(x_1, x_2, \neg x_3)$.

Une variable propositionnelle y (resp. x_1, \dots, x_n) est une *variable de sortie* (resp. *des variables d'entrée*) d'une porte de la forme $y = f(x'_1, \dots, x'_n)$, où $x'_i \in \{x_i, \neg x_i\}$.

Une variable propositionnelle z est une *variable de sortie (dépendante) d'un ensemble de portes* ssi z est une variable de sortie d'au moins une porte de cet ensemble. Une *variable d'entrée (indépendante) d'un ensemble de portes* est une variable qui n'est variable de sortie d'aucune autre porte de cet ensemble.

Une porte est satisfaite sous une interprétation ssi la partie gauche et droite de la porte sont toutes les deux à *vrai* ou à *faux* sous cette interprétation. Une interprétation satisfait un ensemble de portes ssi chaque porte de cet ensemble est satisfaite sous cette interprétation. Une telle interprétation est appelée modèle de cet ensemble de portes.

4 De la CNF aux portes

En pratique, nous voulons trouver une représentation d'une formule CNF Σ utilisant les portes nous permettant de *maximiser* le nombre de variables dépendantes, dans le but d'en réduire la complexité pour décider de la satisfiabilité de Σ . En réalité, nous décrivons une technique qui extrait ces portes pouvant être déduites à partir de Σ , et qui *couvre* un sous ensemble de clauses de Σ . Le reste des clauses de Σ est représenté par une porte «ou» de la forme *vrai* = $\vee(x_1, \dots, x_n)$, dans le but d'avoir une représentation uniforme.

Plus formellement, supposons que l'ensemble G de portes correspondant aux clauses $Cl(G)$ sont des conséquences logiques d'une CNF Σ , l'ensemble $\Sigma_{non-couvert(G)}$ des clauses non couvertes de Σ par rapport à G est l'ensemble de clauses de $\Sigma \setminus Cl(G)$.

Par conséquent, $\Sigma \equiv \Sigma_{non-couvert(G)} \cup Cl(G)$.

Non trivialement, on peut voir que les clauses additionnelles $Cl(G)\setminus\Sigma$ peuvent jouer un rôle important pour la déduction ou la recherche de satisfiabilité.

Connaître les variables de sortie peut jouer un rôle important pour la décidabilité d'une formule CNF.

En effet, la valeur de vérité d'une variable y d'une porte dépend de la valeur de vérité de ses variables d'entrées x_i . La valeur de vérité de cette variable de sortie peut être obtenu par propagation et peut ne pas être pris en compte dans les heuristiques de branchement des algorithmes DPLL [DLL62]. Dans le cas général, connaître n' variables de sorties dans une représentation d'une formule CNF sous forme de portes utilisant n variables permet de réduire le nombre d'interprétations possibles de 2^n à $2^{n-n'}$. Évidemment, la réduction de l'arbre de recherche augmente avec le nombre de variables dépendantes détectées.

Malheureusement, pour obtenir une telle réduction de l'arbre de recherche, on pourrait se poser les problèmes suivants :

- Extraire des portes d'une formule CNF, dans le cas général, est trop coûteux, à moins de fournir quelques critères simples d'en extraire en effet, montrer que $y = f(x_1, \dots, x_i)$ (où y, x_1, \dots, x_i appartiennent à Σ) est une porte de Σ , est un problème coNP-complet.
- Lorsque l'ensemble des portes détectées contient des définitions «récurives» (comme $y = f(x, t)$ et $x = g(y, z)$), brancher sur les variables d'entrées (indépendantes) n'est pas suffisant pour déterminer la valeur de vérité des variables dépendantes. Traiter ces définitions récursives coïncide avec le problème NP-dur de détection d'ensemble coupe cycle minimal dans un graphe.

Dans ce papier, nous traitons ces deux aspects. Le premier en restreignant la déduction par propagation seulement. Le second en utilisant des heuristiques sur les graphes orientés.

Rappelons tout d'abord quelques définitions sur la propagation de contraintes.

5 Propagation de contraintes booléennes (BCP)

La propagation des contraintes booléennes ou résolution unitaire est la plus utilisée et la plus utile des techniques pour SAT.

Soit Σ une formule CNF, $BCP(\Sigma)$ est la formule CNF obtenu en *propageant* tous les littéraux unitaires de Σ . Propager un littéral unitaire l de Σ consiste à supprimer d'une part toutes les clauses c de Σ tel que $l \in c$ et d'autre part de remplacer toutes les clauses c' de Σ tel que $\neg l \in c'$ par $c' \setminus \{\neg l\}$. La formule CNF obtenu de cette manière est équivalente à Σ du point de vue de la satisfiabilité.

L'ensemble des littéraux unitaires propagés de Σ utilisant BCP est noté $UP(\Sigma)$. Évidemment, nous avons $\Sigma \models UP(\Sigma)$. BCP est une forme de résolution et applicable en temps linéaire. Cette méthode est complète pour les formules de Horn. De plus, de son utilisation dans les procédures DPLL, BCP est utilisée dans de nombreux solveurs SAT comme prétraitement afin d'en déduire des informations intéressantes comme des littéraux impliqués [DABC96] et des littéraux équivalents [BSG00][LB01]. Des traitements locaux basés sur cette technique permet de fournir des choix de variables intéressants (comme l'heuristique UP [LA97]).

Dans la suite, nous montrons que cette technique peut être étendu, nous permettant d'extraire plus de dépendances fonctionnelles.

6 BCP et dépendances fonctionnelles

En réalité, BCP peut être utilisée afin de détecter des dépendances fonctionnelles. Le résultat principal de ce papier consiste à exploiter de manière pratique la propriété suivante : les portes peuvent être détectées en utilisant simplement BCP, alors que dans le cas général déterminer si une porte est une conséquence logique d'une formule CNF est un problème coNP-complet

Propriété 1

Soit Σ une formule CNF, $l \in \mathcal{L}(\Sigma)$, et $c \in \Sigma$ tel que $l \in c$. Si $c \setminus \{l\} \subset \neg UP(\Sigma \wedge l)$ alors $\Sigma \models l = \wedge(\neg\{c \setminus \{l\}\})$.

Preuve 1

Soit $c = \{l, \neg l_1, \neg l_2, \dots, \neg l_m\} \in \Sigma$ tel que $c \setminus \{l\} = \{\neg l_1, \neg l_2, \dots, \neg l_m\} \subset \neg UP(\Sigma \wedge l)$. La fonction booléenne $l = \wedge(\neg\{c \setminus \{l\}\})$ peut être réécrite comme $l = \wedge(l_1, l_2, \dots, l_m)$. Afin de prouver que $\Sigma \models l = \wedge(l_1, l_2, \dots, l_m)$, nous devons montrer que chaque modèle de Σ , est aussi un modèle de $l = \wedge(l_1, l_2, \dots, l_m)$. Soit I un modèle de Σ , alors

1. soit l est à vrai dans I : I est aussi un modèle de $\Sigma \wedge l$. Comme $\{\neg l_1, \neg l_2, \dots, \neg l_m\} \subset \neg UP(\Sigma \wedge l)$, nous avons $\{l_1, l_2, \dots, l_m\} \subset UP(\Sigma \wedge l)$, alors $\{l_1, l_2, \dots, l_m\}$ sont à vrai dans I . Donc, I est aussi un modèle de $l = \wedge(l_1, l_2, \dots, l_m)$;
2. ou l est à faux dans I : comme $c = \{l, \neg l_1, \neg l_2, \dots, \neg l_m\} \in \Sigma$ alors I satisfait $c = \{\neg l_1, \neg l_2, \dots, \neg l_m\} \in \Sigma$. Donc, au moins un littéral $l_i, i \in \{1, \dots, m\}$ est à vrai dans I . Donc, I est aussi un modèle de $l = \wedge(l_1, l_2, \dots, l_m)$;

Clairement, en fonction du signe du littéral l , les portes «et» et «ou» peuvent être détectés. Par exemple, la porte «et» $\neg l = \wedge(l_1, l_2, \dots, l_n)$ est équivalente à la porte «ou» $l = \vee(\neg l_1, \neg l_2, \dots, \neg l_n)$. Cette propriété couvre aussi les équivalences binaires car $a = \wedge(b)$ est équivalent à $a \Leftrightarrow b$.

En réalité, cette propriété nous permet de détecter de nouvelles portes par rapport à la méthode syntaxique utilisé dans [ORL02]. Illustrons le par un exemple.

Exemple 1

Soit $\Sigma_1 \supseteq \{y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg y \vee x_1, \neg y \vee x_2, \neg y \vee x_3\}$.

Dans [ORL02], Σ_1 peut être représenté par un graphe où chaque sommet est une clause et chaque arête indique s'il existe une résolvente tautologique entre deux clauses du graphe. Chaque composante connexe du graphe peut représenter une porte. Comme on peut le voir ces quatre clauses appartiennent à une même composante connexe. C'est une condition nécessaire pour qu'un tel sous ensemble de clauses encode une porte. En ayant cet ensemble de clauses (celles apparaissant dans une composante connexe), on doit ensuite déterminer de manière syntaxique si l'on est en présence d'une porte «et» ou «ou». Cette détermination se fait en temps polynômial. Dans l'exemple précédent, nous avons $y = \wedge(x_1, x_2, x_3)$.

Considérons maintenant l'exemple suivant,

Exemple 2

$\Sigma_2 \supseteq \{y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg y \vee x_1, \neg x_1 \vee x_4, \neg x_4 \vee x_2, \neg x_2 \vee x_5, \neg x_4 \vee \neg x_5 \vee x_3\}$.

Clairement la représentation sous forme de graphe de cet ensemble de clauses est différente et la technique utilisé précédemment ne nous permet pas de découvrir la porte $y = \wedge(x_1, x_2, x_3)$. La propriété nécessaire mais non suffisante n'est plus satisfaite.

Maintenant, selon la propriété 1, les deux portes «et» des exemples 1 et 2 sont détectées. En effet, dans l'exemple 1, $UP(\Sigma_1 \wedge y) = \{x_1, x_2, x_3\}$ et $\exists c \in \Sigma_1$, $c = (y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3)$ tel que $c \setminus \{y\} \subset \neg UP(\Sigma_1 \wedge y)$. De plus, dans l'exemple 2, $UP(\Sigma_2 \wedge y) = \{x_1, x_4, x_2, x_5, x_3\}$ et $\exists c' \in \Sigma_2$, $c' = (y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3)$ tel que $c' \setminus \{y\} \subset \neg UP(\Sigma_2 \wedge y)$.

En conséquence, une technique de recherche de portes consiste à vérifier la propriété 1 pour chaque littéral apparaissant dans Σ . Une autre étape, consiste à trouver les variables dépendantes de la formule originale, ayant maintenant l'ensemble de portes. Une porte nous permet de déterminer la variable dépendante par rapport aux variables d'entrées qui sont des variables indépendantes. Maintenant, si plusieurs portes partagent un même littéral, la caractérisation des variables dépendantes n'est plus la même. En effet, on peut voir apparaître des cycles comme le montre l'exemple suivant.

Exemple 3

$$\Sigma_3 \supseteq \{x = \wedge(y, z), y = \vee(x, \neg t)\}.$$

Clairement, Σ_3 contient un cycle. En effet, x dépend des variables y et z , alors que y dépend des variables x et t . Si l'on ne considère qu'une seule porte, affecter des valeurs aux variables d'entrées permet d'en déterminer la valeur de la variable de sortie. Mais dans l'exemple 3, affecter une valeur aux variables qui ne sont pas considérées comme dépendantes n'est pas suffisant dans tous les cas pour déterminer la valeur des variables de sorties. Dans l'exemple, affecter une valeur à z et t n'est pas suffisant pour déterminer la valeur de x et de y . Néanmoins, dans l'exemple, si l'on affecte une valeur à une variable (x , qui est appelé *variable coupe cycle*) du cycle, la valeur de y est déterminé dans tous les cas. Nous devons donc supprimer ces formes de cycles afin de déterminer un sous ensemble de variable qui une fois affecté déterminera toutes les autres. Un tel ensemble est appelé *backdoor* dans [WGS03]. Dans l'exemple 3, le backdoor correspond à l'ensemble $\{x\} \cup \{z, t\}$. Dans ce contexte, le backdoor est constitué des variables indépendantes et des variables du coupe cycle. Trouver l'ensemble minimal de variables coupe cycle dans un graphe est un problème NP-dur. Nous en parlons dans la section qui suit.

7 Recherche de variables dépendantes

Par la suite, nous considérons une représentation sous forme de graphe des portes détectées. Plus formellement, un ensemble de portes peut être représenté par un graphe biparti $G = (O \cup I, E)$ comme suit :

- Pour chaque porte nous lui associons deux sommets. Le premier $o \in O$ représente la variable de sortie de la porte, et le second $i \in I$ représente l'ensembles des variables d'entrées. Le nombre total de sommets est donc inférieur à $2 \times \#gates$, où $\#gates$ est le nombre de portes.
- Pour chaque porte, une arête (o, i) entre les deux sommets o et i représentant la partie gauche et droite de la porte est créée. Des arêtes supplémentaires sont créées entre $o \in O$ et $i \in I$ si une variable de sortie associée au sommet o appartient à l'ensemble des variables d'entrées du sommet i .

Trouver le *plus petit* sous ensemble V' de O tel que le sous graphe $G' = (V' \cup I, E')$ est acyclique est un problème NP-dur.

En réalité, tout sous ensemble V' rendant le graphe acyclique associé à l'ensemble des variables indépendantes, permet de déterminer toutes les autres. Lorsque V' est de taille c , et l'ensemble des variables dépendantes est de taille d , alors l'espace de recherche est réduit de 2^n à $2^{n-(d-c)}$, où n est le nombre total de variables de la formule CNF.

Nous devons trouver un compromis entre la taille de V' , et le gain en temps pour la résolution ensuite.

Dans la suite, nous proposons deux heuristiques pour trouver un ensemble coupe cycle V' . La première s'appelle *Maxdegré*. Cette heuristique consiste à construire V' de manière incrémentale en sélectionnant le sommet de plus haut degré jusqu'à ce que le graphe soit acyclique.

La seconde est appelée *MaxdegréCycle*. Elle consiste à construire de manière incrémentale V' en sélectionnant le sommet ayant le plus haut degré mais cette fois ci appartenant à un cycle. Cette heuristique garanti, qu'à chaque fois qu'une variables est choisie, la suppression d'un cycle.

Dans la section qui suit, nous présentons et discutons des résultats expérimentaux de notre approche. Ces résultats regroupent à la fois les portes détectées et les variables du coupe cycle dans le but de nous fournir un partitionnement des variables (indépendantes et dépendantes). Deux stratégies sont explorées : dans la première, à chaque fois qu'une porte est trouvée les clauses la constituant sont supprimées de Σ ; dans la seconde, les clauses participant au codage d'une porte ne sont supprimées qu'à la fin de la détection des portes. Bien que la première approche dépend fortement de l'ordre dans lequel on va propager les littéraux, la seconde approche est indépendante ce celui ci. Ces deux stratégies sont comparées en terme de portes détectées, de la taille de l'ensemble coupe cycle, du nombre de variables dépendantes et du nombre de clauses non couvertes.

8 Résultats expérimentaux

Nous avons implémenté ce prétraitement. Il est écrit en C sous Linux Redhat 7.1 (<http://www.cril.univ-artois.fr/~ostrowski/Binaries/11satpreproc>). Toutes les expérimentations ont été effectuées sur un Pentium IV, 2.4 Ghz. La description des instances testées peuvent être trouvé sur SATLib (<http://www.satlib.org>).

Nous avons testé à la fois la technique décrite dans [ORL02] et celle décrite précédemment sur les instances issues de la dernière compétition SAT [SAT02, SAT03], dont par exemple les instances de model-checking, de VLSI et de planification.

Des résultats plus complet sont disponibles à l'adresse :

<http://www.cril.univ-artois.fr/~ostrowski/result-11satpreproc.ps>.

Par la suite, nous montrons quelques exemples. Sur chaque classe d'instances, nous calculons la moyenne et l'écart-type par rapport au nombre d'instances de la classe.

Dans la table 1, pour chaque classe considérée, les résultats de la technique décrite dans [ORL02] et de celle décrite dans ce papier sont présentés, en terme de nombre de portes identifiées ($\#G$). Deux approches : d'un côté on ne supprime pas les clauses et de l'autre on les enlève aussitôt qu'une porte est détectée. Les résultats montrent clairement que cette approche nous permet d'identifier plus de portes. De plus, il n'est pas surprenant de détecter moins de portes lorsque l'on retire les clauses au fur et à mesure.

Dans la table 2, on a pris l'option de ne pas supprimer les clauses. On compare les deux heuristiques (*Maxdegré* et *MaxdegréCycle*) afin de couper les cycles. Pour chaque classe d'instances, on fournit en moyenne le nombre de variables dépendantes ($\#D$), la taille

Classes d'Instances		technique dans [ORL02]	Notre approche		
Nom	(#Inst.,#V[<i>min</i> -Max],#C[<i>min</i> -Max])		Nb cl. supp. #G	Cl. supp. #G #C supp.	
Blocks	(3,484[283-758],27423[9690-47820])	14[6]	236[134]	18[5]	271[142]
Logistics	(8,994[116-3016],12706[953-50457])	380[265]	437[417]	169[213]	630[585]
Pipe	(6,1642[834-2577],18624[6695-33270])	1312[679]	1407[697]	1240[639]	13898[9083]
Facts	(13,3178[2218-4315],48737[22539-90646])	713[147]	1601[541]	497[170]	1731[510]
Parity	(30,1044[64-3176],3614[254-10325])	568[828]	510[594]	328[455]	663[870]
Qg	(10,969[512-1331],33747[9685-64054])	310[91]	1828[652]	298[80]	1708[601]
Ca	(7,637[26-2282],1835[70-6586])	419[547]	459[592]	414[542]	1233[1615]
Dp	(11,1427[213-3193],3580[376-8308])	1117[856]	1468[1211]	915[812]	2534[2298]
Bmc2	(5,1952[316-4089],6908[1002-13531])	895[714]	1025[850]	744[623]	2082[1824]
Rand	(6,2217[2000-2500],6568[5921-7401])	2133[236]	2444[381]	2103[252]	6212[692]
Ezfact	(40,1441[193-3073],9169[1113-19785])	40[18]	268[127]	68[33]	68[33]
Med	(3,761[341-1159],20154[5556-36291])	66[32]	316[162]	14[5]	319[164]
Avg-checker	(4,917[648-1188],28661[17087-40441])	324[105]	1098[375]	304[101]	1092[373]
nw/nc/fw	(13,3997[2756-5074],15829[10886-20123])	89[40]	468[136]	125[38]	125[38]
Am	(4,2011[433-4264],6925[1458-14751])	989[835]	772[585]	393[276]	927[625]
Cnf	(2,2424[2424-2424],14812[14812-14812])	2336[0]	3280[0]	2301[6]	13703[149]

TAB. 1 – #G : Nombre de portes détectées(moyenne[écart-type])

Classe d'Instance	(#V[<i>min</i> -Max])	<i>Maxdegré</i>			<i>MaxdegréCycle</i>		
		#D	#CS	#B	#D	#CS	#B
Blocks	(484[283-758])	38[13]	198[123]	353[215]	39[9]	197[124]	352[216]
Logistics	(994[116-3016])	113[158]	245[218]	441[532]	143[164]	214[194]	410[522]
Pipe	(1642[834-2577])	980[768]	265[219]	582[201]	764[449]	481[192]	798[348]
Facts	(3178[2218-4315])	738[237]	813[256]	1964[604]	487[124]	1064[362]	2216[623]
Parity	(1044[64-3176])	243[388]	84[46]	573[528]	287[410]	40[21]	528[505]
Qg	(969[512-1331])	303[202]	228[236]	228[236]	11[6]	521[194]	521[194]
Ca	(637[26-2282])	290[434]	130[142]	344[403]	265[341]	155[206]	369[481]
Dp	(1427[213-3193])	513[463]	451[485]	725[625]	551[496]	412[343]	686[498]
Bmc2	(1952[316-4089])	662[716]	27[22]	886[874]	660[696]	30[10]	888[893]
Rand	(2217[2000-2500])	1777[301]	357[339]	440[343]	1152[134]	981[111]	1064[115]
Ezfact	(1441[193-3073])	28[35]	66[45]	1370[1073]	55[27]	39[18]	1343[1060]
Med	(761[341-1159])	205[102]	110[72]	110[72]	14[4]	302[157]	302[157]
Avg-checker	(917[648-1188])	209[357]	606[283]	606[283]	276[94]	539[187]	539[187]
nw/nc/fw	(3997[2756-5074])	39[48]	151[47]	3899[854]	94[24]	96[23]	3844[855]
Am	(2011[433-4264])	327[263]	97[68]	413[241]	298[206]	126[99]	441[287]
Cnf	(2424[2424-2424])	472[564]	1801[564]	1953[564]	1170[2]	1103[2]	1255[2]

TAB. 2 – Taille du backdoor sans suppression de clauses

de l'ensemble coupe cycle (*#CS*), la taille du backdoor (*#B*) et le temps CPU cumulé pour déterminer les portes et trouver cet ensemble. Sur certaines classes d'instances, le backdoor peut ne représenter que 10% du nombre de variables seulement.

Dans la table 3, nous avons pris l'option de supprimer les clauses. Le nombre de portes détectées est bien inférieur que lorsqu'on les laisse. D'un autre coté, la taille de l'ensemble coupe cycle est généralement plus petit.

En conséquence, aucune option n'est préférable dans le cas général. En effet, trouver un petit backdoor, dépend à la fois de la classe d'instance considérée et des options.

Cependant, dans la plupart des cas, l'option de supprimer les clauses et l'heuristique *MaxdegréCycle* engendre un backdoor plus réduit.

Nous essayons actuellement d'intégrer de manière efficace ce prétraitement dans un solveur SAT efficace, qui lui permettra de brancher directement sur ces variables (i.e. le backdoor). Pour l'instant ce prétraitement n'as pas du tout été optimisé. Néanmoins, il est applicable en pratique (**moins de 1 seconde dans la plupart des cas**). c'est pour

classe d'Instances	(#V[<i>min</i> - <i>Max</i>])	<i>Maxdegré</i>			<i>MaxdegréCycle</i>		
		# <i>D</i>	# <i>CS</i>	# <i>B</i>	# <i>D</i>	# <i>CS</i>	# <i>B</i>
Blocks	(484[283-758])	18[4]	0[0]	373[219]	18[4]	0[0]	373[219]
Logistics	(994[116-3016])	135[147]	25[48]	419[539]	152[178]	7[13]	401[509]
Pipe	(1642[834-2577])	1020[735]	219[215]	543[223]	956[513]	282[124]	606[283]
Facts	(3178[2218-4315])	488[127]	0[0]	2214[621]	488[127]	0[0]	2214[621]
Parity	(1044[64-3176])	318[426]	0[0]	497[480]	318[426]	0[0]	497[480]
Qg	(969[512-1331])	122[99]	138[87]	410[189]	181[60]	80[25]	351[140]
Ca	(637[26-2282])	317[433]	94[113]	317[392]	302[388]	109[151]	332[434]
Dp	(1427[213-3193])	724[643]	149[151]	513[357]	728[641]	145[143]	509[353]
Bmc2	(1952[316-4089])	680[706]	1[1]	868[883]	680[705]	1[1]	868[884]
Rand	(2217[2000-2500])	1591[418]	495[396]	625[401]	1200[129]	886[102]	1016[111]
Ezfact	(1441[193-3073])	48[23]	10[5]	1350[1064]	49[23]	9[5]	1349[1064]
Med	(761[341-1159])	14[4]	0[0]	302[157]	14[4]	0[0]	302[157]
Avg-checker	(917[648-1188])	302[100]	0[0]	512[181]	302[100]	0[0]	512[181]
nw/nc/fw	(3997[2756-5074])	73[14]	40[22]	3864[857]	95[24]	18[10]	3842[856]
Am	(2011[433-4264])	367[254]	0[0]	373[239]	367[254]	0[0]	373[239]
Cnf	(2424[2424-2424])	1988[12]	285[12]	437[12]	2210[6]	63[6]	215[6]

TAB. 3 – Taille du backdoor avec option de suppression

cela que le temps n'est pas mentionné dans les différentes tables.

9 Perspectives

Une première piste possible concerne l'utilisation des clauses découvertes. Notre représentation de formules sous la forme de portes peut se révéler efficace pour la déduction ou la recherche de satisfiabilité. Pour illustrer cette idée, considérons de nouveau l'exemple 2. De la formule Σ , on en déduit la porte $y = \wedge(x_1, x_2, x_3)$. La forme clausale de cette porte est donné par $\{y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg y \vee x_1, \neg y \vee x_2, \neg y \vee x_3\}$.

Clairement, les clauses additionnelles $\{\neg y \vee x_2, \neg y \vee x_3\}$ sont des résolvantes de Σ , qui ne peuvent être obtenu qu'au bout de deux et six étapes de résolution respectivement. Donc, la représentation des portes de Σ induit des résolvantes binaires non triviales qui peuvent rendre la déduction ou la résolution plus facile. Prendre en compte cette caractéristique dans une base de clauses ou de portes serait une piste intéressante pour la déduction ou la recherche de satisfiabilité. Certaines portes découvertes représentent aussi des équivalences ($x \Leftrightarrow y$) et les remplacer permettrait d'obtenir une réduction importante de l'espace de recherche.

Une autre piste possible concerne l'analyse du graphe obtenu. Comment par exemple utiliser des techniques de décomposition sur celui ci. Afin de réduire encore la taille du backdoor, nous regarderons comment exploiter les parties traitables de la formule (par exemple les formules de horn ou horn-renommable).

10 Conclusions

Clairement, nos expérimentations sont assez encourageantes. Détecter un ensemble de variables dépendantes peut se faire à moindre cout. Des cycles dans le graphe apparaissent mais peuvent être supprimé efficacement. Nous implémentons actuellement ce prétraitement dans un solveur SAT efficace. De plus, nous pensons que l'étude des cycles et des variables dépendantes est essentiel pour la compréhension des problèmes SAT difficiles.

11 Remerciements

Ce travail à été soutenu en partie par le CNRS, le FEDER, l' *IUT de Lens* et le *Conseil Régional du Nord/Pas-de-Calais*. Nous remercions les relecteurs pour leurs précieux commentaires de la version précédente de ce papier.

Références

- [BSG00] L. Brisoux, L. Sais, and E. Grégoire. Recherche locale : vers une exploitation des propriétés structurelles. In *Actes des Sixièmes Journées Nationales sur la Résolution Pratique des Problèmes NP-Complets(JNPC'00)*, pages 243–244, Marseille, 2000.
- [BW03] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Sixth International Symposium on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.
- [DABC96] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier. Sat versus unsat. In D.S. Johnson and M.A. Trick, editors, *Second DIMACS Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pages 415–436, 1996.
- [DD01] Olivier Dubois and Gilles Dequen. 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)*, volume 1, pages 248–253, Seattle, Washington (USA), August 4–10 2001.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5 :394–397, 1962.
- [GMTZ01] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR '01)*, Siena, June 2001.
- [JJN04] Matti Järvisalo, Tommi Junttila, and Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for Boolean circuits. In *AI&M 2004, 8th International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, USA, January 4–6 2004.
- [KMS97] Henry A. Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. In *Abstract appears in "Abstracts of the Poster Sessions of IJCAI-97"*, Nagoya (Japan), 1997.
- [LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Nagoya (Japan), August 1997.
- [LB01] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, Boston University, Massachusetts, USA, June 14th-15th 2001.

- [Ole00] Shtrichman Oler. Tuning sat checkers for bounded model checking. In *Proceedings of Computer Aided Verification (CAV'00)*, 2000.
- [ORL02] Grégoire E. Mazure B. Ostrowski R. and Sais L. Recovering and exploiting structural knowledge from cnf formulas. In *Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, pages 185–199, Ithaca (N.Y.), 2002. LNCS 2470, Springer Verlag.
- [RSB99] Antoine Rauzy, Lakhdar Saïs, and Laure Brisoux. Calcul propositionnel : vers une extension du formalisme. In *Actes des Cinquièmes Journées Nationales sur la Résolution Pratique de Problèmes NP-complets (JNPC'99)*, pages 189–198, Lyon, 1999.
- [SAT02] Sat 2002 : Fifth international symposium on theory and applications of satisfiability testing, May 2002. <http://gauss.eecs.uc.edu/Conferences/SAT2002/>.
- [SAT03] Sat 2003 : Sixth international symposium on theory and applications of satisfiability testing, May 2003. <http://www.mrg.dist.unige.it/events/sat03/>.
- [WGS03] Ryan Williams, Carla P. Gomez, and Bart Selman. Backdoors to typical case complexity. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1173–1178, 2003.
- [ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD'2001*, pages 279–285, San Jose, CA (USA), November 2001.