HAL
archives-ouvertes.fr

Utilisation de la Propagation de Contraintes Booléennes pour la Production de Sous-Clauses

Sylvain Darras, Gilles Dequen, Laure Devendeville, Bertrand Mazure, Richard Ostrowski, Lakhdar Saïs

▶ **To cite this version:**

Sylvain Darras, Gilles Dequen, Laure Devendeville, Bertrand Mazure, Richard Ostrowski, et al.. Utilisation de la Propagation de Contraintes Booléennes pour la Production de Sous-Clauses. Premières Journées Francophones de Programmation par Contraintes, CRIL - CNRS FRE 2499, Jun 2005, Lens, pp.69-78. inria-00000058

HAL Id: inria-00000058
https://hal.inria.fr/inria-00000058

Submitted on 25 May 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Actes JFPC 2005

Utilisation de la propagation de contraintes booléennes pour la production de sous-clauses

S. Darras [1] **G. Dequen** [1] **L. Devendeville** [1] **B. Mazure** [2] **R. Ostrowski** [2] **L. Saïs**[2]

[1] LaRIA CNRS, Univ. de Picardie, 33 rue Saint Leu, 80039 Amiens Cedex 1, France
[2] CRIL CNRS, Univ. d'Artois, rue Jean Souvraz SP-18, 62307 Lens Cedex, France
{darras,dequen,devendeville}@u-picardie.fr {mazure,ostrowski,sais}@cril.univ-artois.fr

Résumé

La propagation de contraintes booléennes (BCP) est la technique la plus utile et la plus utilisée dans les solveurs SAT. Dans cet article, nous proposons une autre utilisation de cette technique dans le but de réduire, en termes de nombre de clauses et de longueur des clauses, la formule initiale. En considérant le graphe d'implications généré par la procédure BCP comme un arbre de résolution, nous pouvons déduire des sous-clauses de la formule initiale. Nous montrons ensuite, comment une telle extension peut être implémentée dans les solveurs actuels où la procédure BCP est utilisée à chaque nœud de l'arbre de recherche. Nous présentons une première implémentation de cette approche dans le cadre d'un pré-traitement pour le solveur Zchaff. Pour finir, des résultats comparatifs préliminaires montrant les points forts et les faiblesses de l'approche sont fournis sur certaines classes d'instances de nature structurées.

Mots-clés : SAT, Propagation de contraintes booléennes, raisonnement et recherche, subsomption.

1 Introduction

Les récents progrès sur la résolution pratique du problème SAT permettent aujourd'hui de résoudre de grandes instances issues d'applications réelles qui sont codées sous forme normale conjonctive (CNF) (e.g. [7, 8, 11, 2]). De nombreux problèmes sont proposés et des compétitions (e.g. Dimacs'93, Beijing'96, SAT'01-04) sont organisées sur ces instances SAT exprimants les contraintes intrinsèques à ces problèmes. La grande taille des instances SAT issues d'applications réelles est désormais à la portée des solveurs actuels tel que Zchaff [12], et nous permet d'envisager l'utilisation de la logique propositionnelle pour exprimer et résoudre pratiquement ces types d'instances. Ces progrès montrent que le pire cas (en terme de complexité) se produit rarement en pratique sur ce type de problème. Pour la plupart, les instances SAT issues de problèmes réels contiennent certaines formes de structures permettant aux solveurs spécialisés de les résoudre efficacement. Lors du codage, ces structures créent généralement des ensembles de clauses partageant de nombreuses variables communes comparativement aux instances aléatoires. Prenons par exemple la formule booléenne $x_1 \wedge x_2 \rightarrow y_1 \wedge y_2 \wedge \ldots \wedge y_n$. Il résulte de la mise sous forme clausale de cette formule n clauses partageant les deux variables x_1 et x_2. Ainsi, on peux voir qu'après avoir affecté la valeur $faux$ à y_i, les autres clauses deviennent redondantes et peuvent être éliminées par subsomption en utilisant la clause $\neg x_1 \vee \neg x_2$.

Notre intuition est que si l'on considère ces instances au cours de la recherche, à chaque nœud de l'arbre de résolution, de nombreuses clauses deviennent redondantes et certaines sous-clauses peuvent être déduites, diminuant la taille de la formule.

L'objectif principal de cet article est de montrer comment il est possible d'utiliser cette structure au cours de la recherche. Nous nous focalisons sur la déduction de sous-clauses qui peut nous permettre de réduire la formule à sa « véritable » taille et nous proposons une méthode de complexité polynomiale en temps et en espace constant.

La propagation des contraintes (ou propagation unitaire) est l'un de ces processus de déduction. Elle peut être considérée comme une forme restrictive de résolution et un cas particulier de la règle de subsomption. Le BCP est un paradigme important pour SAT. En effet, la plupart des solveurs pour SAT se basent sur la procédure de Davis-Putnam-Logemann-Loveland (DPLL) [3] où le BCP est appliqué à chaque nœud de l'arbre. Sur de nombreuses instances, 90% de l'espace de recherche est exploré en utilisant le BCP. Ce processus étant important, de nombreux travaux sont menés dans le but d'améliorer cette procédure (e.g. Zchaff)

et d'étendre son utilisation pratique. D'autres techniques de simplifications (e.g. [5, 1]), d'heuristiques de branchement (e.g. [9, 4]), d'analyse de conflits (e.g. [10, 12]) et de déduction de dépendances fonctionnelles (e.g. [6]) sont basées sur le BCP.

Dans cet article, nous proposons une nouvelle exploitation de la propagation des contraintes dans le but de simplifier les formules booléennes. Plus précisement, le graphe généré par le proccessus de la propagation des contraintes peut être vu comme un arbre de résolution encodant des clauses de la formule initiale et de nouvelles résolvantes. L'ensemble de toutes ces résolvantes peut être exponentiel dans le pire cas par rapport à l'ensemble de clauses encodées dans le graphe de contraintes. Afin d'éviter ce problème, nous considérons dans cet article une approche travaillant en espace constant et en temps polynomial permettant d'obtenir un sous-ensemble de ces résolvantes. Nous montrons ensuite comment cette approche peut être utilisée dans les dernières générations de solveurs pour SAT.

L'article est organisé de la façon suivante. Après quelques définitions préliminaires, nous présentons et discutons des relations entre la résolution et la propagation des contraintes booléennes. Ensuite, nous présentons une approche permettant la production de sous-clauses travaillant en espace constant et utilisant le graphe des contraintes.

Enfin, nous présentons quelques résultats expérimentaux concernant l'intégration dynamique de notre approche dans les solveurs SAT.

2 Définitions et notations

Une *formule CNF* Σ est un ensemble (interprété comme une conjonction) de *clauses*, où chaque clause est un ensemble (interpreté comme une disjonction) de *littéraux*. Un littéral est une variable propositionnelle positive ou négative. On note $var(\Sigma)$ (resp. $lit(\Sigma)$) l'ensemble des variables (resp. littéraux) apparaissant dans Σ. Une *clause unitaire* est une clause ne contenant qu'un seul littéral appelé *littéral unitaire*. Une clause binaire contient deux littéraux associés à deux variables différentes. Une clause contenant des littéraux de variables différentes est appelée fondamentale. Elle est appelée tautologique lorsqu'elle contient deux littéraux opposés. La taille d'une formule Σ est donnée par $|\Sigma| = \sum_{c \in \Sigma} |c|$, où $|c|$ est le nombre de littéraux de c.

Pour la suite, nous utilisons formule (resp. variable) à la place de formule CNF (resp. variable propositionnelle). De plus, nous définissons la négation d'un ensemble A de littéraux comme l'ensemble $\bar{A}$ correspondant à l'ensemble des littéraux opposés. On note $A_{\vee}$ (resp. $A_{\wedge}$) la disjonction (resp. conjonction) de tous les littéraux de A.

Une *interprétation* d'une formule Σ est une affectation des valeurs de vérité $\{vrai,faux\}$ des variables. Cette interprétation est dite partielle si et seulement si un sous-ensemble des variables de $var(\Sigma)$ est affecté. Un *modèle* d'une formule est une interprétation qui satisfait la formule. En conséquence, SAT revient à trouver un modèle d'une formule s'il existe ou à prouver qu'il n'en existe aucun.

Une formule ψ est une conséquence logique de ϕ (noté $\phi \models \psi$) si et seulement si tout modèle de ϕ est un modèle de ψ.

Soient c_1 et c_2 deux clauses de Σ.

i) *règle de résolution* : s'il existe un littéral l (appelé pivot de la résolvante) tel que $l \in c_1$ et $\neg l \in c_2$, alors une *résolvante* sur l entre c_1 et c_2 peut être définie comme $res(l,c_1,c_2) = c_1 \backslash \{l\} \cup c_2 \backslash \{\neg l\}$.

ii) *règle de subsomption* : lorsque $c_1 \subseteq c_2$ (c.à.d. c_1 est une sous-clauses de c_2), alors c_1 subsume c_2.

La règle de résolution (resp. subsomption) conduit à une nouvelle formule $\Sigma \cup res(l,c_1,c_2)$ (resp. $\Sigma \backslash c_2$) équivalente à Σ pour SAT. Une résolvante $r = res(l,c_1,c_2)$ est dite resolvante subsumante si et seulement si $\exists c \in \Sigma$ tel que r subsume c. En appliquant itérativement la règle de résolution, nous obtenons une preuve par résolution permettant de conclure à la non satisfaisabilité d'une formule. $\pi = \{c_1,c_2,\ldots,c_n\}$ est une résolution par dérivation de c_n d'une formule Σ si et seulement si c_i est soit une clause de Σ ou dérivée par application de la règle de résolution de c_j et c_k avec $j,k < i$. π est une résolution par réfutation, lorsque c_n est une clause. La taille de π est donnée par le nombre de ses clauses.

Pour une formule Σ et un littéral $l \in lit(\Sigma)$, nous définissons $\Sigma(l) = \{c | c \in \Sigma, \{l,\neg l\} \cap c = \emptyset\} \cup \{c \backslash \{\neg l\} | c \in \Sigma, \neg l \in c\}$ comme l'affectation à $vrai$ de l. Pour simplifier, $\Sigma(l_1)(l_2)\ldots(l_n)$ sera noté $\Sigma(l_1,l_2,\ldots,l_n)$.

La propagation des contraintes booléennes consiste à affecter de manière itérative tous les littéraux unitaires jusqu'à rencontrer une clause vide ou avoir affecté tous les littéraux. $\Sigma_{bcp}(l)$ est la formule obtenue par l'application du BCP sur $\Sigma(l)$. L'ensemble des littéraux unitaires produit par le BCP sur $\Sigma(l)$ est défini par $UPL(\Sigma,l)$. Le BCP peut être vu comme une forme restreinte de résolution. À chaque étape, une résolvante subsumante $res(l,\{l\},c_2) = c_2 \backslash \{\neg l\}$ est produite. Enfin, le BCP est une étape de simplification importante pour DPLL.

La procédure DPLL consiste à parcourir en profondeur un arbre binaire de recherche sur l'ensemble des affectations possibles. Après avoir simplifié la formule par l'application du BCP et avoir fixé les littéraux purs (ceux apparaissant soit exclusivement positivement soit exclusivement négativement), une *variable de décision* est choisie et affectée successivement à $faux$ et à $vrai$ cette variable.

3 Exploitation du BCP pour la production de sous-clauses

Dans ce paragraphe, nous montrons comment exploiter le BCP dans le but de produire des sous-clauses. Tout d'abord, nous présentons le graphe d'implications généré

par le BCP et ses deux possibles représentations en arbre de résolution.

3.1 Propagation des contraintes booléennes et graphe d'implications

Un *graphe d'implication* (GI) est un graphe orienté acyclique représentant l'ensemble des littéraux propagés. Un graphe de contraintes (défini ci-dessous) est généré en fonction d'une formule et d'un ensemble de littéraux de décision donnés.

Définition 1 (Graphe d'implications) *Soit Σ une formule et I un ensemble de littéraux de décision. Un graphe d'implications associé à Σ et I est un graphe orienté acyclique étiqueté $\mathcal{G}_{gi}(\Sigma,I) = (\mathcal{V},\mathcal{E})$ où :*

1. *$\mathcal{V} = \{l | l \in I \cup UPL(\Sigma,I)\}$ un ensemble de sommets.*
2. *$\mathcal{E} = \{\langle l_j,l_i \rangle |$*
   $\exists c = \{\neg l_1,\ldots,\neg l_{i-1},l_i,\neg l_{i+1},\ldots,\neg l_n\} \in \Sigma,$
   $c \cap \mathcal{V} = \{l_i\}$, $c \cap \overline{\mathcal{V}} = \{\neg l_1,\ldots,\neg l_{i-1},\neg l_{i+1},\ldots,\neg l_n\}$ et $j \in \{1,\ldots,i-1,i+1,\ldots,n\}$. Chaque arc $\langle v_i,v_j \rangle$ est étiqueté par une clause c, i.e. label($\langle v_i,v_j \rangle$)= c.

Dans la définition du GI, chaque nœud correspond à une affectation de variable. L'ensemble des littéraux associés aux prédécesseurs d'un sommet l ($pred(l)$) correspond aux affectations qui ont propagé l. Tous les arcs d'un $l' \in pred(l)$ à l sont étiquetés par la même clause c (noté $cl(l)$). La clause c et l'ensemble des littéraux $pred(l)$, nous donne la raison de son implication. Remarquons qu'en général, un littéral l peut être impliqué par différentes clauses et littéraux (c.à.d. raisons). Lorsque toutes les raisons appartiennent au graphe, celui ci est dit complet; sinon il est dit incomplet. Dans le cas de graphes implications complets, $pred(l)$ est un sur-ensemble où chaque élément correspond à une raison particulière. Pour plus de clarté dans la présentation, nous considérons dans cet article des graphes d'implications incomplets.

Dans un graphe d'implications $\mathcal{G}_{gi}(\Sigma,I)$, les sommets correspondant aux littéraux de décision I n'ont aucun arc entrant et sont appelés sommets sources ($sources(\mathcal{G}_{gi})$). Les sommets ne comportant aucun sommet sortant sont appelés sommets puits ($puits(\mathcal{G}_{gi})$). Lorsqu'un conflit apparait, $\mathcal{G}_{gi}$ contient deux sommets puits étiquetés par deux littéraux opposés.

Mentionnons que les graphes d'implications ont été largement utilisés par les solveurs SAT dans le but d'apprendre des clauses, appelées *nogoods*, à partir de conflits [12] ou d'effectuer un retour non-chronologique. [10]. Dans ce contexte, chaque littéral de décision l est étiqueté par un *niveau de décision* α correspondant au niveau d'affectation de ce dernier. À chaque étape, les littéraux propagés par le BCP, à un niveau donné, reçoivent la même étiquette. Pour un ensemble de littéraux de décision $I = \{l_1,l_2,\ldots,l_{i-1},l_i\}$, lorsqu'un conflit apparaît sur un littéral l_i, le littéral $\neg l_i$ est impliqué (c.à.d. $\Sigma_{bcp}(\{l_1,l_2,\ldots,_{i-1}\}) \vDash \neg l_i$).

algorithme 1 GI2AR(**entrées** : $\mathcal{G}_{gi} = (\mathcal{V},\mathcal{A})$: GI, $l \in \mathcal{V}$; **sortie** $\mathcal{G}_{ar} = (\mathcal{N},\mathcal{E})$: AR)

1: $r = \overline{pred(l)} \cup \{l\}$
2: $\mathcal{N} = \{r\}$, $\mathcal{E} = \emptyset$
3: $transformation(r,pred(l),\mathcal{G}_{ar},\mathcal{G}_{gi})$

Donc $\neg l_i$ est étiqueté du même niveau que l_{i-1} et $\forall k, 1 \leq k < i$ des arcs $\langle l_k,\neg l_i \rangle$ sont ajoutés et étiquetés par une nouvelle clause impliquée $\{\neg l_1 \ldots \neg l_{i-1},\neg l_i\}$. Un tel graphe d'implications généré par rapport à un ensemble ordonné de littéraux de décision est appelé graphe d'implications ordonné (GIO).

Nous donnons dans la suite une nouvelle transformation d'un graphe d'implications vers une forme spéciale d'arbre de résolution.

3.2 Graphe d'implication et arbre de résolution

Dans ce paragraphe, nous montrons une façon de transformer un graphe d'implications en un arbre de résolution (AR).

Les définitions ci-dessous donnent une description générale d'un arbre de résolution.

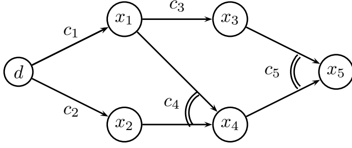
Définition 2 (Arbre de résolution) *Un arbre de résolution associé à graphe d'implications $\mathcal{G}_{gi}(\Sigma,I)$ et à un littéral $d \in \mathcal{V}$ est un graphe orienté acyclique $\mathcal{G}_{ar} = (\mathcal{N},\mathcal{E})$ tel que :*

- *chaque nœud $n \in \mathcal{N}$ est étiqueté par une résolvante.*
- *chaque arc $a = \langle n_i,n_j \rangle \in \mathcal{E}$ est étiqueté par une clause $c \in \Sigma$. La résolvante associée au noeud n_j est obtenue à partir de c et la clause étiquetant n_i.*
- *la racine est étiquetée par la clause $c_d = d \cup \overline{pred(d)}$.*
- *une clause étiquetant un nœud n (sauf la racine) est obtenue par résolution entre la clause étiquetant le père de n et la clause étiquetant l'arc reliant le nœud n à son père.*

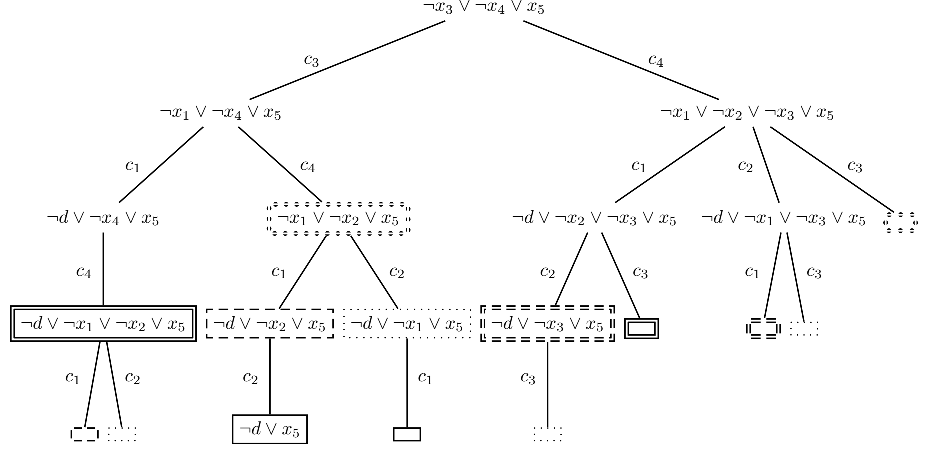
Comme mentionné au paragraphe 2, le BCP peut être vu comme une forme restrictive de résolution. À chaque étape une résolvante subsumante est produite entre une clause unitaire et d'autres clauses de la formule. Cette première transformation (non décrite dans ce papier) nous donne l'arbre de résolution décrivant précisement cette forme restrictive de résolution.

L'algorithme 1 décrit une autre transformation d'un graphe d'implications vers un arbre de résolution à partir d'un sommet v du graphe d'implications. Cette nouvelle transformation nous donne une image originale du BCP dans le sens où elle mène à des méthodes à base de résolution très efficaces.

Exemple 1 *Pour illustrer nos propos, considérons le Graphe d'implication de la Figure 1(a) obtenu à partir de Σ_1*



(a) GI de Σ_1



(b) AR de Σ_1

FIG. 1 – *Du graphe d'implications à l'arbre de résolution*

---

algorithme 2 transformation (**entrées** : r: nœud, s: ensemble de littéraux, $\mathcal{G}_{gi} = (\mathcal{V},\mathcal{A})$: GI ; **entrée-sortie**: $\mathcal{G}_{ar} = (\mathcal{N},\mathcal{E})$: AR)

1: **if** $s \neq \emptyset$ **then**
2: let $s = \{l_1, l_2, \ldots, l_k\}$
3: **for** $i = k$ downto 1 **do**
4: let $p \in pred(l_i)$
5: $c = label_{gi}(\langle p, l_i \rangle)$
6: $s' = (s \backslash \{l_i\}) \cup pred(l_i)$
7: $r' = res(l_i, r, c)$
8: $\mathcal{N} = \mathcal{N} \cup \{r'\}$
9: $\mathcal{E} = \mathcal{E} \cup \{\langle r, r' \rangle\}$ t.q. $label_{ar}(\langle r, r' \rangle) = c$
10: $transformation(r', s', \mathcal{G}_{ar}, \mathcal{G}_{gi})$
11: **end for**
12: **end if**

---

en affectant d à $Vrai$, avec :

$$\Sigma_1 = \left\{ \begin{array}{ll} c_1 : \neg d \vee x_1 & c_4 : \neg x_1 \vee \neg x_2 \vee x_4 \\ c_2 : \neg d \vee x_2 & c_5 : \neg x_3 \vee \neg x_4 \vee x_5 \\ c_3 : \neg x_1 \vee x_3 & c_6 : \neg d \vee \neg x_3 \vee x_5 \vee x_6 \\ \multicolumn{2}{c}{c_7 : \neg x_1 \vee x_2 \vee x_5} \end{array} \right\}$$

Remarque 1 *On remarque que dans l'arbre de résolution de la figure 1(b), les nœuds (sauf la racine) sont étiquetés par des nouvelles clauses. Afin de mettre en avant les sous-arbres identiques, deux nœuds encadrés de manière similaire représentent le même sous-arbre. Comme l'exploitation du graphe d'implications commence par le sommet étiqueté x_5, on remarque que toutes les nouvelles clauses contiennent ce littéral. En appliquant l'algorithme 1 sur l'ensemble des littéraux du graphe d'implications, nous obtenons un ensemble d'arbres de résolutions différents (c.à.d. une forêt).*

Dans l'algorithme 1, la transformation commence par le sommet l du GI (ligne 2), un nouveau nœud n étiqueté par la clause r construite à partir de l et $\overline{pred(l)}$ (la négation des littéraux étiquetant ses prédécesseurs) est ajouté à l'arbre de résolution. Ensuite, on effectue un appel à l'algorithme 2 avec le noeud courant r et l'ensemble s des littéraux prédécesseurs de l dans $\mathcal{G}_{gi}$. À chaque étape, on considère le noeud courant r et l'ensemble de littéraux s restant à traiter. Pour chaque littéral l_i de s, un nœud étiqueté par la résolvante r' obtenue à partir de r et c (ligne 5) et un arc $\langle r, r' \rangle$ étiqueté par c sont ajoutés à l'arbre de résolution (lignes 4-10), le processus est répété récursivement avec la nouvelle résolvante r', et s' obtenue à partir de s en remplaçant le littéral courant par ses prédécesseurs.

Il est à noter que pour chaque littéral l_i (ligne 3), toutes les combinaisons de ses ancêtres dans le graphe d'implication sont considérées, car pour un même appel r et s restent inchangés. En conséquence, l'algorithme 1 a une complexité exponentielle dans le pire cas et peut conduire à un arbre de résolution de taille exponentiel. Afin de pallier à ce problème, nous présentons, dans la suite, une approche ne considérant qu'un sous-ensemble de résolvantes possibles (c-à-d une partie de l'arbre de résolution) qui utilise un espace mémoire constant et qui est de complexité polynomiale en temps.

3.3 Propagation de contraintes et production de sous clauses

Dans ce paragraphe, nous présentons une approche en espace constant et polynomiale en temps. En effet, nous ne considérons qu'un sous-ensemble pertinent de résolvantes de l'arbre de résolution proposée. Plus précisement, une résolvante n'est prise en compte que si elle subsume (soit directement, soit par résolution) des clauses de la formule originale Σ. De plus, en pratique l'arbre de résolution n'est pas explicitement construit.

En premier lieu, fixons quelques définitions et propriétés.

Définition 3 *Soit Σ une formule, $l \in lit(\Sigma)$ et $\mathcal{G}_{ar} = (\mathcal{N},\mathcal{E})$ un arbre de résolution obtenu à partir de $\mathcal{G}_{ig}(\Sigma,l)$. Une clause c est l-sous-inférée de Σ (noté $\Sigma_l \models^* c$) si $\exists c' \in \Sigma$ tel qu'une des conditions suivantes soit satisfaite,*

1. *$c \in \mathcal{N}$ et $c \subset c'$*
2. *$\exists c'' \in \mathcal{N}$ tel que $c = res(p,c',c'') \subset c'$ où $p \in c'$ et $\neg p \in c''$*

Proposition 1 *Soit Σ une formule et $l \in lit(\Sigma)$. Si $\Sigma_l \models^* c$ alors $\Sigma \models c$*

Preuve 1 *Par construction de $\mathcal{G}_{ar} = (\mathcal{N},\mathcal{E})$ de $\mathcal{G}_{ig}(\Sigma,l)$ tous les nœuds de $\mathcal{N}$ sont étiquetés par une résolvante obtenue à partir des clauses de Σ. En conséquence, $\forall d \in \mathcal{N}$, nous avons $\Sigma \models d$. De plus, dans la définition de $\Sigma_l \models^* c$, on distingue deux cas. Dans le premier, $c \in \mathcal{N}$, alors $\Sigma \models c$. Dans le second, $c = res(p,c',c'')$ où $c' \in \Sigma$ et $c'' \in \mathcal{N}$, alors $\Sigma \models c$.*

Proposition 2 *Soient Σ une formule et $l \in lit(\Sigma)$. $\Sigma_l \models^* c$ peut être établi en $O(|\Sigma| \times |var(\Sigma)|)$.*

Preuve 2 *Nous donnons ici une idée de la complexité (pour plus de détails voir l'algorithme 3). Pour calculer $\Sigma_l \models^* c$, on applique d'abord la propagation unitaire sur $\Sigma \wedge l$, puis on calcule $\mathcal{G}_{gi} = (\mathcal{V},\mathcal{E})$. Ce calcul est effectué en temps linéaire. Ensuite, on cherche à trouver une clause $c' \in \Sigma$ telle que $c \subset c'$ ou $res(c,c') \subset c'$. Pour cela, on ne considère que les clauses contenant des littéraux de $UPL(\Sigma,l)$; sinon une telle clause ne peut être l-sous-inférée. Maintenant, considérons une clause c' vérifiant la condition précédente. Pour déterminer la l-sous-inférence, pour chaque littéral $p \in c' \cap \mathcal{V}$, on effectue un parcours de $\mathcal{G}_{gi}$ en commençant par le sommet $v \in \mathcal{V}$ étiqueté par p. Une première clause r composée de p et $\overline{pred(v)}$ est calculée. À cette étape nous avons $r \in \Sigma$. L'étape suivante consiste à générer pour chaque sommet $w \in pred(v)$ une nouvelle résolvante $r = res(w,r,cl(w))$. D'après la définition 3, r et c' sont testés et on peut distinguer trois cas :*

1. *$r \subset c'$ (subsomption directe);*
2. *$res(c,c') \subset c'$ (résolvante subsumante) ;*
3. *Les deux premiers cas ne s'appliquent pas.*

Dans les deux premiers cas, c' est réduite. La recherche continue sur la sous-clause déduite afin de la réduire d'avantage. Dans le dernier cas, la recherche s'effectue en prenant un prédécesseur d'un littéral de r n'apparaissant pas dans c'. En conséquence, un seul passage dans $\mathcal{G}_{gi}$ est nécessaire. Cette opération est réalisée en $O(n+m)$ où $n = |\mathcal{V}|$ et $m = |\mathcal{A}|$. Comme on considère chaque clause de Σ, et que pour chaque clause le nombre de passage dans $\mathcal{G}_{gi}$ est borné par la longueur de la clause, la complexité globale dans le pire cas de ce traitement est de $O(|\Sigma| \times (n+m)) = O(|\Sigma| \times |var(\Sigma)|)$.

Nous proposons maintenant une approche polynomiale afin de déduire des sous-clauses durant la recherche, basée sur la proposition 2.

4 Déduction de sous-clauses au cours de la recherche

Dans cette section, nous présentons une approche pratique pour la déduction de sous-clauses à partir d'une formule Σ et d'un graphe d'implication associé. La déduction de sous-clauses, telle que peuvent le permettre des traitements locaux classiques comme le *look-ahead* [5], peut fournir une aide précieuse aux heuristiques de branchement utilisées dans les algorithmes de type DPLL, permettant une détection plus rapide d'inconsistances locales ayant alors pour effet de réduire la taille de l'arbre de recherche. Décrivons l'algorithme $GetSubclause$ (Algorithme 3), utilisé pour simplifier une formule Σ grâce au graphe d'implication associé $\mathcal{G}_{ig}$. On note $\mathcal{V}_\alpha$ l'ensemble des littéraux affectés au niveau de décision courant α. Soient $y \in \mathcal{V}_\alpha$ et $A \subset \mathcal{V}$ tels que $A_\wedge \rightarrow y$.

Considérant une clause c de Σ, la fonction $GetSubclause$ trouve s'il en existe, des sous-clauses subsumant c (voir ligne 7 de l'algorithme 3) et de nouvelles clauses dont la résolvante avec c subsume c (voir ligne 1 de l'algorithme 3). Lorsqu'une subsomption directe est trouvée, la clause c est subsumée par l'implication $A_\wedge \rightarrow y$, donc c contient y et tous les littéraux de $\bar{A}$. Dans ce cas, nous ne pouvons plus produire de résolvante subsumante de c. Nous pouvons toutefois espérer trouver un sous-ensemble $B = \{x_1,...,x_k\} \subset A$ tel que $x_1 \wedge ... \wedge x_k \rightarrow y$. S'il existe $x_1 \wedge ... \wedge x_k \rightarrow y$, alors $\forall z | (z \in A, z \notin \{x_1,...,x_k\}), \exists x_{i_1},...,x_{i_l} \subset B | x_{i_1} \wedge ... \wedge x_{i_l} \rightarrow z$. C'est-à-dire, si l'on trouve une subsomption $B_\wedge \rightarrow y$ plus courte que $A_\wedge \rightarrow y$, alors tous les littéraux qui apparaissent dans A mais pas dans B sont impliqués par un sous-ensemble des littéraux de B. Pour trouver une telle subsomption, nous devons donc traiter les prédécesseurs des littéraux de $A \backslash B$. Puisque ces littéraux ont des prédecesseurs dans B, ils ont nécessairement été affectés après une partie des littéraux de B. Pour accroître la probabilité de trouver une telle implication à ce stade de l'analyse, la fonction $choice$ retourne la dernière

algorithme 3 $GetSubclause$ (**entrée** $\mathcal{G}_{gi} = (\mathcal{V},\mathcal{A})$: GI, A : ensemble de littéraux, y : littéral, c : clause, α: niveau de décision)

1: **if** $\exists x_r \in \bar{A} \cup \{y\} | \neg x_r \in c$ **and** $\forall x \in (\bar{A} \cup \{y\}) - \{x_r\}, x \in c$ **then**
2: $\Sigma = \Sigma - \{c\} \cup \{c - \{\neg x_r\}\}$
3: **if** $pred(x_r) \neq \emptyset$ **then**
4: $GetSubclause(\mathcal{G}_{gi}, A - \{x_r\} \cup pred(x_r), y, c, \alpha)$
5: **end if**
6: **else**
7: **if** $\forall x \in \bar{A} \cup \{y\}, x \in c$ **then**
8: $\Sigma = \Sigma - \{c\} \cup \{\bar{A}_\vee \vee y\}$
9: $x = choice(A)$
10: **if** $pred(x) \neq \emptyset$ **then**
11: $GetSubclause(\mathcal{G}_{gi}, A - \{x\} \cup pred(x), y, c, \alpha)$
12: **end if**
13: **else**
14: Choisir $x \in A | x \notin c$ **and** $\neg x \notin c$
15: **if** $pred(x) \neq \emptyset$ **then**
16: $GetSubclause(\mathcal{G}_{gi}, A - \{x\} \cup pred(x), y, c, \alpha)$
17: **end if**
18: **end if**
19: **end if**

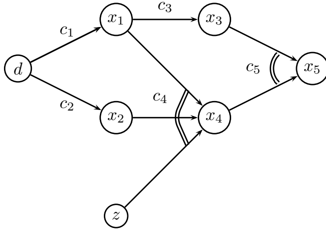


FIG. 2 – *IG de la formule Σ_2 modifiée section 4.1. La source est le littéral d*

variable affectée parmi les variables de A. Dans tous les cas, même si nous ne trouvons pas cette subsomption à ce stade, nous la mettrons en évidence lorsque nous appellerons $GetSubclause$ à partir des variables de $A \backslash B$. Soit z une variable de ce sous-ensemble. Lorsque nous trouverons $B_z = \{z_1, ..., z_p\}$ ($B_z \subset B$) tel que $B_{z\wedge} \rightarrow z$, nous aurons donc déduit la clause $c_z : \neg z_1 \vee ... \vee \neg z_p \vee z$. Sachant que $z_1, ..., z_p, z \in A$, la clause déduite de $A_\wedge \rightarrow y$ contient les littéraux $\neg z_1, ..., \neg z_p, \neg z$. La résolvante entre la clause déduite de $A_\wedge \rightarrow y$ et c_z permet donc de supprimer le littéral z. Il en est de même pour tous les littéraux de $A \backslash B$.

Pour conclure, chercher toutes les subsomptions à partir du graphe $\mathcal{G}_{ig}$ consiste en : $\forall y \in \mathcal{V}_\alpha$, $\forall c \in \Sigma | y \in c$, $GetSubclause(\mathcal{G}_{ig}, pred(y), y, c, \alpha)$.

Reprenons l'exemple 1 décrit précédemment. Pour essayer de déduire une sous-clause de c_6 en considérant les implications de x_5 à travers le Graphe d'Implication de la Figure 1(a), nous allons tout d'abord considérer $x_3 \wedge x_4 \rightarrow x_5$, qui est équivalent à la clause $\neg x_3 \vee \neg x_4 \vee x_5$. On peut voir que la variable x_3 appartient à l'implication actuelle et à la clause c_6. Puisque x_4 n'appartient pas à c_6, toute implication contenant ce littéral ne subsume pas c_6. Nous allons alors chercher des littéraux de c_6 dans les prédécesseurs de x_4 : x_1 et x_2. Suivant notre principe, puisque x_1 et x_2 ne sont pas des littéraux de c_6, remontons vers leur antécédent d. Nous en déduisons $d \wedge x_3 \rightarrow x_5$, et la clause correspondante $c'_6 : \neg d \vee \neg x_3 \vee x_5$ subsume c_6.

En continuant la recherche sur c'_6 avec l'ensemble courant $A = \{x_3, d\}$, la fonction $choice$ choisit la dernière variable affectée de A : x_3. Parcourant $\mathcal{G}_{gi}(\Sigma_1, \{d\})$, x_1 et d sont successivement visités et l'implication $d \rightarrow x_5$ est déduite. La sous-clause correspondante : $c''_6 : \neg d \vee x_5$ subsume directement c_6.

Cette double subsomption de c_6 en c'_6 puis en c''_6 a été rendue possible grâce au choix de x_3 par la fonction $choice$. En effet, si cette fonction avait choisi la variable d, le traitement se serait arrêté immédiatement sans aucune déduction supplémentaire, d n'ayant aucun antécédent. La subsomption de c'_6 en c''_6 aurait tout de même été décelée plus tard dans l'analyse du Graphe d'Implication, lors de l'appel à $GetSubclause(\mathcal{G}_{ig}, pred(x_3), x_3, c, \alpha)$. Nous aurions alors mis en évidence l'implication (triviale) $x_1 \rightarrow x_3$ puis $d \rightarrow x_3$, équivalent à la clause $\neg d \vee x_3$ et dont la résolvante avec c'_6 est $c''_6 : \neg d \vee x_5$.

En appliquant cette technique sur la clause c_7 à partir du même Graphe d'Implication $\mathcal{G}_{ig}(\Sigma_1, \{d\})$, nous pouvons déduire l'implication $x_1 \wedge x_2 \rightarrow x_5$. La résolvante entre la clause correspondant à cette implication et c_7 est $c_r : \neg x_1 \vee x_5$, qui subsume c_7.

En conclusion, ce Graphe d'Implication nous a permis de réduire c_6 de deux littéraux et c_7 d'un littéral. De telles réductions pourront déclencher plus rapidement des propagations unitaires, et ainsi accélérer la résolution. En effet, avec notre formule Σ_1 d'origine, l'affectation de x_5 à $Faux$ n'engendrait aucune propagation unitaire. Par contre, avec les clauses c_6 et c_7 réduites respectivement à $\neg d \vee x_5$ et $\neg x_1 \vee x_5$, l'affectation de x_5 à $Faux$ propage les affectations de d à $Faux$ et de x_5 à $Vrai$.

4.1 Subsomption locale

En considérant une utilisation dynamique de la recherche de subsomptions pour des algorithmes de type DPLL, l'algorithme 3 décrit précédemment ne trouve que des subsomptions valables dans tout l'arbre de résolution. Etant donné que le nombre de subsomptions de ce type est restreint, l'algorithme 3 peut être amélioré de manière à trouver des subsomptions valables uniquement pour une partie de l'arbre de recherche, délimitée par un niveau de décision. Soit β ce niveau de décision pour une subsomption donnée, celle-ci sera effacée (*i.e.* la clause d'origine sera restaurée) lorsqu'un backtrack à un niveau de décision inférieur ou égal à β sera effectué.

Instance	S/I	Zchaff nœuds	Zchaff tps	Pré-traitement (180s) +Zchaff				Pré-traitement (300s) +Zchaff			
				nœuds	subs	var. fixées	tps_z	nœuds	subs	var. fixées	tps_z
SAT.dat.k90	S	N/A	N/A	N/A	373	2 917	N/A	**6 157 239**	**598**	**4 419**	**12 964**
logistics.b	S	**3 810**	**0**	760	458	315	0	464	545	388	0
logistics.c	S	**9 577**	**0**	4 007	549	279	0	2 998	597	352	0
abp4-...-403	I	2 843 489	9 751	2 369 440	1 368	359	6 770	**1 675 832**	**1 458**	**380**	**4 508**
abp1-...-402	I	3 046 922	11 509	**2 353 237**	**1 462**	**265**	**5 710**	2 229 379	1 600	450	6 698
2bitadd_10	I	**60 605**	**40**	60 605	0	0	41	60 605	0	0	40
2bitadd_11	S	**7 870**	**1**	7 870	0	0	0	7 870	0	0	0
longmult10	I	711 397	1 385	605 584	46	117	1 427	**558 171**	**48**	**147**	**1 047**
longmult11	I	1 194 889	2 916	**1 005 477**	**47**	**116**	**2 348**	1 136 339	47	141	2 410
longmult12	I	1 164 158	2 926	**1 020 640**	**48**	**124**	**2 556**	1 108 196	48	137	2 818
longmult13	I	1 567 022	3 871	1 180 017	49	129	3 040	**1 145 954**	**49**	**137**	**2 618**
longmult15	I	**625 534**	**675**	498 430	37	19	626	384 451	51	170	489
bf0432-007	I	**864**	**0**	741	300	215	0	741	300	215	0
bf2670-001	I	**64**	**0**	99	190	184	0	99	190	184	0
flat200-10	S	**14 202**	**2**	14 202	0	0	2	14 202	0	0	3
flat200-100	S	**1 157**	**0**	1 157	0	0	0	1 157	0	0	0
difp_19_3_wal_rcr	S	2 030 996	2342	**1 257 747**	**330**	**431**	**1 819**	1 914 713	349	433	5 834
difp_20_99_arr_rcr	S	370 262	488	**290 264**	**279**	**51**	**279**	290 264	279	51	290
difp_20_3_wal_rcr	S	**64380**	**5**	718 328	318	401	361	718 328	318	401	374

TAB. 1 – *Résultats préliminaires*

algorithme 4 $GetSubClauseLevel$(**entrée** $\mathcal{G}_{gi} = (\mathcal{V},\mathcal{A})$: GI, A : ensemble de littéraux, y : littéral, c : clause, α: niveau de décision)

1: **if** $\exists x_r \in \bar{A} \cup \{y\} | \neg x_r \in c$ **and** $\forall x \in ((\bar{A} \cup \{y\}) - \{x_r\}) \cap \mathcal{V}_\alpha, x \in c$ **then**
2: $\Sigma = (\Sigma - \{c\} \cup \{c - \{\neg x_r\}\})_{dl > max_{x \notin c, x \in A \cap (\mathcal{V} - \mathcal{V}_\alpha)}(dl_x)}$
3: **if** $pred(x_r) \neq \emptyset$ **then**
4: $GetSubClauseLevel(\mathcal{G}_{ig}, A - \{x_r\} \cup pred(x_r), y, c, \alpha)$
5: **end if**
6: **else**
7: **if** $\forall x \in (\bar{A} \cup \{y\}) \cap \mathcal{V}_\alpha, x \in c$ **then**
8: $\Sigma = (\Sigma - \{c\} \cup \{(\bar{A} \cap c)_\vee \vee y\})_{dl > max_{x \notin c, x \in A \cap \mathcal{V} - \mathcal{V}_\alpha}(dl_x)}$
9: $x = choice(A)$
10: **if** $pred(x) \neq \emptyset$ **then**
11: $GetSubClauseLevel(\mathcal{G}_{ig}, A - \{x\} \cup pred(x), y, c, \alpha)$
12: **end if**
13: **else**
14: Choisir $x \in A | x \notin c$ **and** $\neg x \notin c$
15: **if** $pred(x) \neq \emptyset$ **then**
16: $GetSubClauseLevel(\mathcal{G}_{ig}, A - \{x\} \cup pred(x), y, c, \alpha)$
17: **end if**
18: **end if**
19: **end if**

Comme dans la version précédente, considérant une clause c de Σ et l'ensemble des littéraux $\mathcal{V}_\alpha$ affectés au niveau de décision courant α, la fonction $GetSubClauseLevel$ recherche des subsomptions de c (voir lignes 1 et 7 de l'algorithme 4) disponibles tant que tous les littéraux de $\bar{A}$ n'appartenant pas à c et dont le niveau de décision est différent de α gardent leur valeur de vérité. Lorsqu'un littéral l de A a été affecté à un niveau de décision inférieur à α, et s'il n'appartient pas à c, il peut être considéré comme ne faisant pas partie de A tant qu'il reste affecté à cette valeur (*i.e.* aucun backtrack n'est déclenché sur l). $A \backslash \{l\}$ peut donc être utilisé pour produire des sous-clauses de c.

Pour illustrer cette méthode, considérons le Graphe d'Implication de la Figure 4, obtenu à partir de la formule Σ_2 en affectant d à $Vrai$ au niveau de décision α, z ayant été affecté à $Vrai$ au niveau de décision $\beta < \alpha$. Σ_2 est obtenue à partir de Σ_1 (formule de l'exemple 1), en substituant à la clause c_4 la clause $c'_4 : \neg x_1 \vee \neg x_2 \vee \neg z \vee x_4$.

À partir de ce graphe, l'implication $x_1 \wedge x_2 \wedge z \rightarrow x_5$ peut être déduite. La clause correspondante est $c' : \neg x_1 \vee \neg x_2 \vee \neg z \vee x_5$. La résolvante entre c' et c_7 est $c_r : \neg x_1 \vee \neg z \vee x_5$ qui ne subsume pas c_7 car z n'apparaît pas dans cette clause. Cependant, puisque z a été affecté à un niveau de décision inférieur, on peut considérer que c'_4 est uniquement composée des littéraux $\neg x_1 \vee \neg x_2 \vee x_4$ tant que z garde sa valeur actuelle. Ainsi, la résolvante $c_r : \neg x_1 \vee x_5$ subsume c_7 jusqu'au prochain backtrack à un niveau de décision inférieur ou égal à β. L'implication $d \wedge z \rightarrow x_5$ peut aussi être déduite, et correspond à la clause $c'' : \neg d \vee \neg z \vee x_5$. De la même manière, c'' subsume c_6 si l'on considère que z garde sa valeur actuelle, auquel cas c'' est équivalent à $\neg d \vee x_5$ (pour les niveaux de décision supérieur à β). c_6 est donc directement subsumé.

Ce Graphe d'Implication permet de réduire c_6 de deux littéraux et la clause c_7 d'un littéral, pour les niveaux de décision supérieurs à β.

5 Résultats expérimentaux

L'approche que nous proposons pour la déduction de sous-clauses dans cet article peut être utilisée à chaque nœud de l'arbre de recherche DPLL. Rappelons que notre technique de simplification est envisageable dès lors qu'il existe

au moins une clause de longueur deux dans la formule, permettant ainsi la construction du graphe d'implication nécessaire. Dans cette section, nous fournissons des résultats comparatifs préliminaires montrant l'impact de cette méthode sur un ensemble de benchmarks provenant de problèmes structurés et industriels. D'un point de vue pratique, l'application exhaustive de la déduction de sous-clause à chaque nœud de l'arbre n'est pas envisageable étant donné son coût de calcul pénalisant. Il est alors nécessaire de limiter empiriquement voir heuristiquement cette simplification de la formule afin d'obtenir un gain pratique. Notre principal objectif dans cette section sera de montrer l'influence de notre approche sur le nombre de nœuds de l'arbre de recherche. Bien que ces expérimentations soient préliminaires, nous discuterons également des aspects pratiques de nos résultats en mentionant les améliorations éventuelles constatées sur le temps de calcul. Les résultats comparatifs fournis dans le tableau 1 montrent l'impact d'une restriction de la technique de déduction de sous-clauses à un simple pré-traitement sur l'ensemble de formules que nous avons sélectionné. Pour la plupart, ces formules sont des benchmarks provenant de problèmes industriels. Ainsi, les familles de formules « longmult* », « difp* » et « abp* » sont des instances de problèmes de « Bounded Model Checking ». Pour les autres formules de notre ensemble, nous avons privilégié, entre autres problèmes « dimacs », des formules décrivant des problèmes de coloration de graphes « flat* ». Ces résultats expérimentaux ont été réalisés sur des AMD Athlon 2000+ disposant de 512Mo de RAM sous un système Linux. Ainsi, le tableau 1 permet de comparer les performances obtenues en terme de nombre de nœuds et de temps de calcul, par l'un des meilleurs démonstrateurs SAT actuel : « zchaff »[1] lorsque notre pré-traitement est appliqué. La colonne intitulée « S/I » précise la nature logique de la formule où « S » et « I » signifient respectivement « Satisfaisable » et « Insatisfaisable ». Pour la mise en place des conditions expérimentales, la durée maximum de résolution autorisée pour la résolution d'un problème a été fixée arbitrairement à 4 heures. Au delà de cette limite, la mention « N/A » figure dans le tableau. Les deux premières colonnes de notre tableau de résultats fournissent pour chaque benchmark sélectionné le nombre de nœuds et le temps de calcul obtenu par le solveur zchaff. Comme nous l'avons mentionné plus haut, l'implémentation que nous avons faite de la déduction de sous-clause ne concerne ici que la tâche de pré-traitement de la formule. Etant donné son caractère relativement coûteux, nous avons fixé arbitrairement une limite de temps d'application de ce pré-traitement au delà de laquelle la production de clauses subsumantes s'arrête et où le travail de résolution de zchaff peut débuter. Nous avons fixé cette limitation dans le tableau de résultats 1 à 180 et 300 secondes. Cette limitation arbitraire résulte d'une étude expérimentale ne figurant pas dans cet article. Les colonnes intitulées « subs » représentent le nombre de subsomptions trouvées dans le temps alloué au pré-traitement. Les colonnes intitulées « var. fixées » fournissent le nombre de variables affectées grâce au pré-traitement ou plus précisément le nombre de sous-clauses unaires déduites. Enfin, les colonnes « tps_z » représentent le temps de calcul nécessaire à zchaff pour résoudre l'instance. Le temps de pré-traitement dans ce cas est à ajouter à celui de zchaff pour avoir une idée précise du gain ou de la perte obtenue par notre approche. Pour ces colonnes le temps limite autorisé pour zchaff a été fixé a 4 heures moins le temps de pré-traitement. Les colonnes mises en gras dans le tableau représentent l'option ayant le meilleur temps de calcul global. Pour les instances de type « Bounded Model Checking », « longmult* » et « abp* », les gains en terme de nombre de nœuds peuvent atteindre près de 50% par rapport aux résultats de zchaff seul. Notons, néanmoins, que la production de sous-clauses est fortement soumise à la nature même de la formule. Prenons par exemple les instances de coloration de graphes, « flat* », qui sont très peu sensibles à la déduction de subsomptions. Rappelons toutefois que ces résultats préliminaires sont obtenus à partir de la déduction de sous-clauses appliquée à la racine de l'arbre de recherche. Cette technique peut être appliquée à tous les nœuds de l'arbre de recherche et nécessite dans ce cas une gestion dynamique. La déduction de sous-clauses peut alors être vue comme une nouvelle forme d'apprentissage.

[1] zchaff version 2004.11.15

6 Conclusion et travaux futurs

Dans cet article, nous avons présenté une nouvelle extension de la portée de la propagation unitaire. Nous avons décrit deux façons possibles de transformer le graphe d'implication en arbre de résolution. La seconde transformation nous donne une autre vision du BCP, généralement considéré comme une forme limitée de résolution. En effet, de nombreuses résolvantes intéressantes peuvent être générées grâce au graphe d'implication du BCP, analysé par une technique basée sur la résolution. Pour qu'une telle extension soit envisageable de manière pratique, nous avons montré que trouver un sous-ensemble de résolvantes qui amène à une subsomption se fait en un temps polynômial, et peut se greffer à d'autres techniques de résolution de type DPLL.

Nos expérimentations préliminaires sont encourageantes. Sur plusieurs familles d'instances, une réduction du nombre de nœuds a été obtenue. Pour confirmer l'intérêt de l'approche proposée, des expérimentations plus poussées sont nécessaires. Par ailleurs, il est également nécessaire de poursuivre nos travaux dans le sens d'une gestion dynamique de la production de sous-clauses à tous les nœuds de l'arbre DPLL. Cela induit qu'il sera nécessaire, compte tenu du coût de traitement de notre approche, d'évaluer la perti-

nence des résolvantes déduites en fonction de leur potentiel pratique.

Références

[1] F. Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI*, 2002.

[2] A. Biere, E. Clarke, R. Raimi, and Y Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–72, 1999.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5:394–397, 1962.

[4] G. Dequen and O. Dubois. kcnfs: An efficient solver for random k-SAT formulae. In *International Conference on Theory and Applications of Satisfiability Testing (SAT), Selected Revised Papers, LNCS*, volume 6, pages 486–501, may 2003.

[5] O. Dubois, P. André, Y. Boufkhad, and J. Carlier. Sat versus unsat. In *Second DIMACS Challenge*, pages 415–436, 1996.

[6] E. Grégoire, B. Mazure, R. Ostrowski, and L. Sais. Automatic extraction of functional dependencies. In *SAT'04*, May 2004.

[7] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI'92*, pages 359–363, Vienna, Austria, 1992.

[8] T. Larrabee. Efficient generation of test patterns unsing boolean satisfiability. *IEEE Transaction on CAD*, 11:4–15, 1992.

[9] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI'97*, pages 366–371, Nagoya (Japan), August 1997.

[10] J.P.M. Silva and K.A. Sakallah. Grasp - a new search algorithm for satisfiability. In *CAD'96*, 1996.

[11] J.P.M. Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC'00*, June 2000.

[12] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01*, pages 279–285, San Jose, CA (USA), November 2001.