

Élimination des redondances dans les algorithmes de résolution de SAT

Eliminating redundancies in SAT search algorithms

Richard Ostrowski

Bertrand Mazure

Lakhdar Saïs

Éric Grégoire

CRIL - CNRS

Université d'Artois
Rue Jean Souvraz – SP 18
F-62307 Lens Cedex France

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

Résumé

L'analyse de conflits est considérée comme un paradigme essentiel pour l'efficacité des algorithmes de recherche, en particulier pour résoudre des problèmes issus d'applications réelles. La structure intrinsèque présente dans le codage des problèmes réels (contrairement aux problèmes aléatoires) amène souvent les algorithmes de résolution classique à résoudre les mêmes sous-problèmes (i.e. les mêmes parties de l'espace de recherche). Cette redondance dans l'exploration de l'espace de recherche a amené le développement de solveurs efficaces intégrant différentes formes d'analyse de conflits. Dans ce papier, le principe de recherche complémentaire initialement présenté par Purdom dans le but d'éviter certaines formes de redondances est revisité et étendu. Pour des raisons de complexité de mise en œuvre, ce principe n'étant utilisé jusqu'à présent que dans un cadre spécifique et restreint. Nous montrons qu'il est possible de l'intégrer efficacement sous sa forme la plus générale dans les algorithmes de résolution de SAT. Plus précisément, nous montrons que l'exploration redondante de l'espace de recherche peut être évitée sans ajouter explicitement de nouvelles contraintes. En plus des coupures (retours-arrière) induites par les contraintes implicites, nous montrons qu'il est possible de déduire sous certaines conditions des littéraux unitaires, importants pour la suite de la recherche. Des résultats expérimentaux intéressants ont été obtenus sur de nombreuses classes d'instances, illustrant ainsi l'intérêt pratique de cette approche.

Mots Clés

Satisfiabilité, Procédure de Davis et Putnam, SAT

Abstract

Conflict analysis is a powerful paradigm of backtrack search algorithms, in particular for solving satisfiability

problems arising from practical applications. Accordingly, most recent satisfiability solvers implement forms of conflict analysis, at least to some extent. In this paper, a complementary search principle initially introduced by Purdom [18] is revisited and extended. Contrary to his author's a priori analysis, it is shown very efficient from a practical point of view in that it allows search trees in SAT solving to be pruned in a significant way while obeying an interesting time and space trade-off. More precisely, we show that redundancies during the search process can be avoided without adding new constraints explicitly. Moreover, the technique can be used not only to prune branches in the search tree, but also to derive implied literals. Extensive experimental results illustrate the feasibility and practical interest of this approach.

Keywords

SAT, Davis and Putnam's algorithm.

1 Introduction

SAT, i.e. décider de la satisfiabilité d'une formule booléenne sous forme normale conjonctive, est un problème NP complet [3] et un paradigme de base dans beaucoup de domaines d'intelligence artificielle comme le raisonnement automatique et non monotone et dans divers secteurs d'informatique comme la conception et la vérification de circuits (VLSI).

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 ([14, 13, 19, 23, 4, 2, 20, 22, 15]).

Alors qu'il reste une forte compétition dans le but d'implémenter des solveurs efficaces pour résoudre les instances k -SAT aléatoires [8], 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. Pour ces instances il est clair que l'analyse de conflits (*apprentissage* dans les domaines SAT et CSP) joue un rôle fondamental dans l'implémentation de solveurs SAT efficaces. De plus, d'autres techniques comme la technique de « *restart* » [10] et l'utilisation de structures paresseuses, comme l'implémentation des « *watched literals* » [16] permettent d'améliorer considérablement les performances des solveurs.

L'analyse de conflits permet d'une part d'effectuer un retour arrière non chronologique et d'autre part d'ajouter des informations dans la base de connaissances (appelées « *no-goods* ») dans le but de couper des branches dans l'arbre de recherche. De nombreuses approches d'analyse de conflits ont été développées dans différents domaines de l'Intelligence Artificielle (IA), notamment dans les systèmes de maintien de cohérence, des problèmes de satisfaction de contraintes (CSP) et en déduction automatique. Dans le cadre SAT, les premiers résultats autour de l'intégration de ces techniques d'apprentissage dans les algorithmes de résolution sont apparus dans [21] et [1]. Ces techniques se sont révélées efficaces dans la résolution d'instances difficiles issues d'applications réelles encodant d'une certaine manière des connaissances structurelles.

Dans ce papier, le principe de recherche complémentaire initialement proposé par Purdom [18] est ré-analysé et appliqué de manière originale et productive. Le principe de Purdom peut être interprété comme une technique d'apprentissage, dans le sens où il est utilisé pour éviter de rencontrer les mêmes conflits. Il diffère des techniques d'apprentissage classiques par le fait que la partie redondante de l'espace de recherche est connue à l'avance (à chaque point de choix). Ceci permet d'éliminer la partie redondante de l'espace de recherche de manière prospective (« *lookhead* ») ou rétrospective (« *lookback* »). Comme le note Purdom lui-même, la recherche complémentaire joue un rôle important d'un point de vue théorique sauf que pour être exploitée il faut ajouter explicitement des contraintes obtenues à partir de la négation d'une sous-formule CNF dont la transformation en formule CNF est coûteuse. Cette contrainte de mise en œuvre pratique a été citée par Gallo et Urbani [9] : “*Purdom's branching criterion succeeds in reducing the size of the search tree but a price must be paid. In fact, the formula must be transformed into the standard form of set of clauses, which might be quite costly*”.

Répondant à cette limitation pratique, nous proposons dans ce papier une mise en œuvre efficace du principe de Purdom dans les algorithmes de type DPPL. De plus, une nouvelle manière de dériver des littéraux unitaires est proposée et des résultats expérimentaux montrent l'efficacité de cette approche.

Le papier est organisé de la manière suivante. Dans un

premier temps, nous rappelons rapidement quelques définitions liées au problème SAT. Ensuite, la recherche des redondances dans l'arbre de recherche est mise en évidence en utilisant le principe de recherche complémentaire de Purdom. Enfin, après avoir mis en évidence les principaux inconvénients de cette technique, nous montrons comment elle peut être utilisée efficacement dans des solveurs SAT. Des résultats expérimentaux montrant les performances de cette méthode sont présentés sur un panel d'instances [6, 11]. Nous concluons par la présentation de quelques perspectives à ces travaux.

2 Définitions et notations

Le problème SAT consiste à déterminer la satisfiabilité d'une formule booléenne mise sous forme normale conjonctive (CNF).

Un *littéral* est une variable propositionnelle positive (l) ou négative ($\neg l$). Une formule CNF est une conjonction de clauses. Une *clause* est une disjonction de littéraux. Une formule sous forme normale disjonctive (DNF) est quant à elle une disjonction de conjonction de littéraux. Un ensemble de variables propositionnelles (resp. clauses) apparaissant dans une formule Σ est noté $var(\Sigma)$ (resp. $cl(\Sigma)$). Une *interprétation* d'une formule booléenne est une fonction $I : var(\Sigma) \rightarrow \{vrai, faux\}$ représentée par un ensemble de littéraux : $I = \{l/I[l] = vrai\} \cup \{\neg l/I[l] = faux\}$. Si toutes les variables apparaissent dans l'interprétation, cette interprétation est dite *complète* ; sinon elle est dite *partielle*. L'ensemble de littéraux satisfaisant (resp. falsifiant) une clause c étant donnée une interprétation (partielle) I est noté $sat(I, c)$ (resp. $unsat(I, c)$) et est défini par $I \cap \{l/l \in c\}$ (resp. $I \cap \{\neg l/l \in c\}$). De façon similaire, l'ensemble des littéraux non affectés d'une clause c dans une interprétation est noté $unset(I, c)$ et est défini par $\{l/l \in c \text{ and } l \notin sat(I, c) \cup unsat(I, c)\}$. Une interprétation I satisfait une clause c quand $|sat(I, c)| \geq 1$.

On dit qu'une clause c est *sursatisfaite* par I quand $|sat(I, c)| \geq 2$. L'ensemble vide de clauses (noté $\{\}$) est interprété comme *vrai* et une clause vide (notée $()$) est interprétée comme *faux*.

Une interprétation I est un *modèle* de Σ si et seulement si toutes les clauses de Σ sont satisfaites par I (i.e. $I[\Sigma] = vrai$).

Le problème SAT revient à décider si une formule CNF admet un modèle ou à prouver qu'il n'existe pas de modèle pour cette formule.

La longueur d'une clause C_i ($|C_i|$) est le nombre de littéraux différents apparaissant dans C_i . Une clause de taille 2 (resp. 1) est appelée clause *binnaire* (resp. *unaire*). La *taille* $|\mathcal{F}|$ d'une formule CNF \mathcal{F} est donnée par $\sum_{i=1}^{cl(\Sigma)} |C_i|$. Un littéral *unitaire* (resp. *pur* ou *monotone*) est un littéral apparaissant dans une clause unaire (resp. le littéral opposé n'apparaît pas dans la formule). La propagation unitaire (resp. des littéraux purs) est le processus de simplification de la formule qui consiste à détecter et à propager les littéraux unitaires (resp. purs). Ces procédures sont

connues comme étant la propagation des contraintes booléennes (BCP).

On appelle $\Sigma \wedge x$, noté $\Sigma(x)$, une formule obtenue à partir de Σ en affectant à x la valeur de vérité *vrai*. Formellement $\Sigma(x) = \{C \mid C \in \Sigma, \{x, \neg x\} \cap C = \emptyset\} \cup \{C \setminus \{\neg x\} \mid C \in \Sigma, \neg x \in C\}$. Par extension on note $\Sigma(x_1, x_2, \dots, x_i)$ la formule obtenue par simplifications successives de Σ par x_1, x_2, \dots, x_i . De même on note $\Sigma_{UP}(x_1, x_2, \dots, x_i)$ la formule obtenue en appliquant en plus la propagation unitaire après chaque étape de simplification de Σ par x_k avec $k \leq i$.

3 La procédure de recherche de Davis et Putnam

Les algorithmes complets actuels pour la résolution pratique du problème SAT sont basés sur la procédure de Davis et Putnam et plus précisément sur la version proposée par Davis, Logemann et Loveland [5], plus connue sous le nom DPLL. La procédure DPLL est un algorithme énumératif dont le but est de générer un arbre de recherche.

Les deux points importants de cet algorithme sont la fonction de simplification de la formule et celle du choix de la prochaine variable à affecter.

1. *Simplifier()* implémente la procédure de propagation de contraintes booléennes (BCP). Cette procédure permet d'obtenir une formule simplifiée qui reste équivalente pour la satisfiabilité. D'autres techniques de simplification (en temps polynômial) peuvent être utilisées. (par exemple résolution sur les clauses binaires, résolution bornée, etc.).
2. *ChoixDeVariable()* choisit une variable non affectée comme point de choix. En général on utilise des heuristiques pour ce choix de variable. La variable est sélectionnée en utilisant des connaissances syntaxiques comme la longueur des clauses, le nombre d'occurrences des variables, etc. Par exemple, l'heuristique sélectionnant la variable apparaissant le plus souvent dans les clauses les plus courtes (MOM) s'avère être souvent la plus efficace. Plus précisément, pour chaque variable non affectée l , on attribue un score $f(l)$ de la manière suivante : $f(l) = \sum_{l \in c} w(c)$ où $w(x) = 2^{-|C|}$ [12]. La prochaine variable est choisie parmi celles maximisant la fonction de sélection : $H(x) = f(x) * f(\neg x) + \min(f(x), f(\neg x))$. D'autres fonctions de pondération des clauses et de sélection ont été également proposées (ex. [7]).

4 Recherche complémentaire

Le principe de recherche complémentaire de Purdom peut s'énoncer de la façon suivante :

Définition 1 Soit Σ une formule CNF, et $l \in \text{var}(\Sigma)$, Σ peut être réécrit de manière équivalente comme $(l \vee \alpha) \wedge (\neg l \vee \beta) \wedge \gamma$ où $\alpha = \{c \setminus \{l\} \mid l \in c \text{ et } c \in \Sigma\}$,

fonction DPLL

Entrée: Une formule CNF Σ

Résultat: *VRAI* si Σ est SAT, *FAUX* sinon

```

begin
   $\Sigma \leftarrow \text{Simplifier}(\Sigma)$ 
  if  $() \in \Sigma$  then
    return FAUX
  endif
  if  $\Sigma = \{\}$  then
    return VRAI
  endif
   $l \leftarrow \text{ChoixDeVariable}(\Sigma)$ 
  if DPLL( $\Sigma(l)$ ) then
    return VRAI
  else
    return DPLL( $\Sigma(\neg l)$ )
  endif
end

```

fonction Simplifier

Entrée: Une formule CNF Σ

Résultat: Σ simplifiée par BCP

```

begin
  if  $\exists$  un littéral unitaire  $l$  then
    return Simplifier( $\Sigma(l)$ )
  endif
  if  $\exists$  un littéral pur  $l$  then
    return Simplifier( $\Sigma(l)$ )
  endif
  return  $\Sigma$ 
end

```

Algorithme 1 – algorithme DPLL

$$\beta = \{c \setminus \{\neg l\} \mid \neg l \in c \text{ et } c \in \Sigma\} \text{ et } \gamma = \{c \setminus \{l, \neg l\} \mid c = \emptyset \text{ et } c \in \Sigma\}.$$

Si l'on considère l'arbre de recherche de la procédure classique DPLL, $\Sigma(l)$ et $\Sigma(\neg l)$ peuvent avoir une intersection non vide. Plus précisément, les interprétations satisfaisant à la fois α et β sont visitées deux fois, comme le montre la figure 1.

La propriété qui suit est proposée dans le but d'éviter cette forme de redondance.

Propriété 1 (Purdom[18]) Soit Σ une formule CNF, l un littéral apparaissant dans Σ tel que $\Sigma = (l \vee \alpha) \wedge (\neg l \vee \beta) \wedge \gamma$, alors Σ est satisfiable ssi $\Sigma(l)$ est satisfiable ou $\Sigma(\neg l) \wedge \neg \beta$ est satisfiable.

Dans la propriété précédente, on peut constater que $\neg \beta$ est une formule sous forme normale disjonctive. Afin d'en déduire les contraintes correspondantes, on doit la transformer en CNF. Cette transformation malheureusement a un

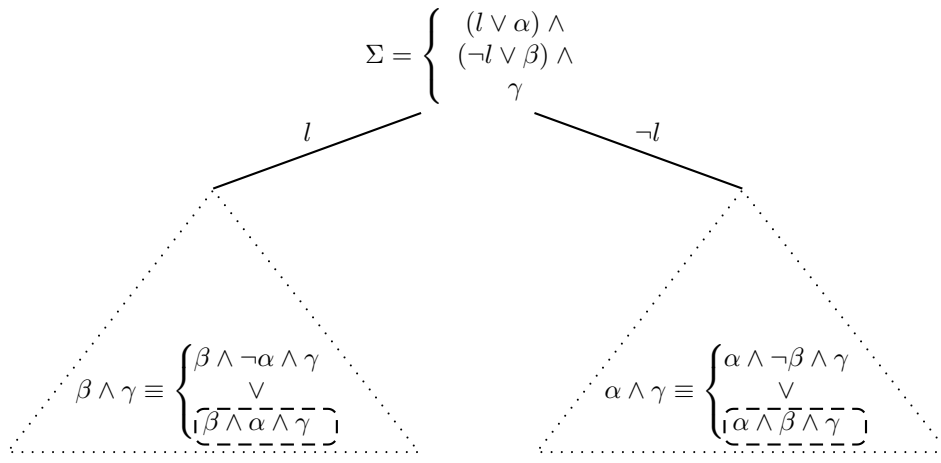


FIG. 1 – Recherche redondante dans l’arbre de recherche classique de DPL

prix en temps et en espace comme l’a indiqué Gallo et Urbani [9]. Pour cette raison, ces auteurs indiquent que cette propriété n’est utilisable que lorsque la partie β est réduite à une seule clause (le littéral $\neg l$ n’apparaît qu’une seule fois dans Σ). La négation d’une telle clause est un ensemble de clauses unitaires. Néanmoins, choisir une telle variable comme variable de branchement peut conduire à avoir un arbre de recherche qui ne sera pas équilibré pour le reste de la recherche, alors que les heuristiques les plus utilisées tendent à choisir la ou les variables de branchement qui équilibrent le plus l’arbre de recherche.

Remarque 1 Si l est un littéral pur dans Σ alors $\beta = \{\}$. Comme un ensemble vide de clauses est interprété comme *vrai*, $\neg\beta$ est *faux*. Dans ce cas, on peut dériver la propriété des littéraux purs : si l est un littéral pur apparaissant dans Σ , alors Σ est satisfiable ssi $\Sigma \wedge l$ est satisfiable.

Exemple 1 Considérons l’ensemble Σ de clauses suivant :

$$\Sigma = \begin{aligned} &(a \vee b \vee c) \wedge \\ &(a \vee d \vee \neg f) \wedge \\ &(\neg a \vee d \vee \neg e) \wedge \\ &(\neg a \vee f \vee g) \wedge \\ &(\neg a \vee \neg h) \wedge \\ &(\neg d \vee f) \end{aligned}$$

Σ peut être réécrit de manière équivalente de la façon suivante :

$$\Sigma = (a \vee \alpha) \wedge (\neg a \vee \beta) \wedge \gamma, \text{ où } \alpha = (b \vee c) \wedge (d \vee \neg f), \\ \beta = (d \vee \neg e) \wedge (f \vee g) \wedge (\neg h) \text{ et } \gamma = (\neg d \vee f).$$

L’ensemble des contraintes additionnelles est donné par :
 $\neg\beta = (\neg d \vee \neg f \vee h) \wedge (\neg d \vee \neg g \vee h) \wedge (e \vee \neg f \vee h) \wedge (e \vee \neg g \vee h).$

L’exemple précédent montre le principal inconvénient du principe de Purdom qui ne permet pas dans le cas général, en apparence, d’en faire une utilisation pratique.

5 La recherche complémentaire revisitée

Dans cette section, nous montrons comment il est possible d’exploiter de manière efficace cette propriété dans une procédure DPLL.

Pour illustrer l’importance de cette approche, considérons la formule de l’exemple 2 représentant une fonction booléenne que l’on retrouve dans de nombreuses instances issues d’applications réelles.

Exemple 2 Soit Σ l’ensemble de clauses :

$$\Sigma = \begin{aligned} &(y \vee \neg x_1 \vee \dots \vee \neg x_i \dots \vee \neg x_n) \wedge \\ &(\neg y \vee x_1) \wedge \\ &\dots \\ &(\neg y \vee x_i) \wedge \\ &\dots \\ &(\neg y \vee x_n) \end{aligned}$$

On remarque que Σ encode une fonction booléenne de la forme : $y = \wedge(x_1, \dots, x_i, \dots, x_n)$. y est appelé variable de *sortie* (*dépendante*) et $x_i, 1 \leq i \leq n$ sont appelées les variables d’*entrées* (*indépendantes*).

De la propriété 1, on peut en déduire les deux cas de figure suivants :

- x_i comme variable de branchement : Σ est satisfiable ssi $\Sigma(x_i)$ est satisfiable ou $\Sigma(\neg x_i) \wedge (y \wedge x_1 \wedge \dots \wedge x_{i-1} \wedge x_{i+1} \wedge \dots \wedge x_n)$. Dans ce cas on en déduit un ensemble de littéraux unitaires.
- y comme variable de branchement : Σ est satisfiable ssi $\Sigma(y)$ est satisfiable ou $\Sigma(\neg y) \wedge (\neg x_1 \vee \dots \vee \neg x_{i-1} \vee \neg x_{i+1} \vee \dots \vee \neg x_n)$. La dernière clause est redondante et appartient à $\Sigma(\neg y)$. Elle est par conséquent inutile.

Comme le montre l’exemple 2, on peut déduire des informations intéressantes lorsque l’on choisit une variable d’entrée (indépendante) comme variable de branchement sachant que la valeur de vérité des variables dépendantes

(sorties) sont automatiquement déduites des variables d'entrées de la fonction correspondante. Plus généralement la complexité de résolution du problème est fonction du nombre de variables indépendantes [17].

Montrons à présent comment la recherche complémentaire peut être utilisée dans des procédures de type DPLL.

Tout d'abord, lorsqu'une nouvelle amélioration est implémentée dans une procédure DPLL, elle concerne généralement l'un des points suivants :

1. *simplification* : comment simplifier la formule ?
2. *règle de branchement* : comment sélectionner la prochaine variable à affecter ?
3. *analyse de conflits* : que doit-on faire en cas de conflits ?

L'intégration de la propriété 1 dans des procédures DPLL répond aux points 1 et 3.

De la propriété 1, on remarque que l'ensemble des clauses obtenu de $\neg\beta$ est ajouté quand $\Sigma(l)$ est insatisfiable (c-à-d. quand on effectue un retour arrière sur le littéral l , et que nous explorons la branche droite de l'arbre de recherche). On le voit clairement dans l'algorithme 2 où les clauses β sont empilées au début de l'exploration de la branche droite de l'arbre de recherche. La pile est mise à jour lorsque l'on effectue un retour arrière.

Partant de la dualité entre SAT et UNSAT, nous pouvons aisément déduire qu'il n'est pas nécessaire d'ajouter $\neg\beta$. Lorsque l'ensemble β est satisfait (c-à-d. les clauses contenant $\neg l$ sont sursatisfaites) la formule $\neg\beta$ est insatisfiable. Par conséquent, il suffit de mettre simplement dans la pile des pointeurs sur les clauses où $\neg l$ apparaît et de vérifier au cours de la recherche si ces clauses sont sursatisfaites. Ce test est réalisé en maintenant pour chaque clause le nombre de littéraux qui la satisfait. En d'autres termes les contraintes $\neg\beta$ sont considérées sans les ajouter explicitement dans la formule.

Définition 2 Soit Σ une formule CNF, on appelle interprétation courante générée par DPLL, notée I_c , l'interprétation $\{l_1, l_{1_1}, \dots, l_{1_{n_1}}, l_2, l_{2_1}, \dots, l_{2_{n_2}}, \dots, l_i, l_{i_1}, \dots, l_{i_{n_i}}\}$ tels que l_1, \dots, l_n sont des littéraux sélectionnés par la fonction *ChoixDeVariable()* et $l_{j_1}, \dots, l_{j_{n_j}}$ ($\forall j \in [1..i]$) les littéraux propagés par la fonction *Simplifier()* au point de choix l_j . On note I_d l'interprétation courante I_c restreinte aux littéraux l_1, \dots, l_i .

Définition 3 Soient Σ une formule CNF et $I_d = \{l_1, \dots, l_{i-1}, l_i\}$. Un littéral l_k avec $k \leq i$ est actif de I_d si et seulement si $\Sigma_{UP}(l_1, \dots, l_{k-1}, \neg l_k)$ est insatisfiable.

Pour plus de lisibilité, nous notons $\Sigma_i = \Sigma_{UP}(l_1, \dots, l_{i-1})$ et β_i l'ensemble de sous-clauses de Σ_i tel que $(l_i \vee \beta_i) \in \Sigma_i$.

fonction *DPLL_R*

Entrée: Une formule CNF Σ , une pile s de clauses

Résultat: *VRAI* si Σ est SAT, *FAUX* sinon

```

begin
   $\Sigma \leftarrow$  Simplifier_R( $\Sigma, s$ )
  if  $() \in \Sigma$  then
    return FAUX
  endif
  if  $\Sigma = \{\}$  then
    return VRAI
  endif
   $l \leftarrow$  ChoixDeVariable( $\Sigma$ ) /*  $\Sigma = (l \vee \alpha) \wedge (\neg l \vee \beta) \wedge \gamma$  */
  if DPLL_R( $\Sigma(l), s$ ) then
    return TRUE
  else
    push( $s, \beta$ )
    if DPLL_R( $\Sigma(\neg l), s$ ) then
      return VRAI
    endif
    pop( $s$ )
    return FAUX
  endif
end

```

fonction *Simplifier_R*

Entrée: Une formule CNF Σ , une pile s de clauses

Résultat: Σ simplifiée par BCP et par les propriétés 2 et 3

```

begin
  if  $\exists$  un littéral unitaire  $l$  then
    return Simplifier_R( $\Sigma(l), s$ )
  endif
  if  $\exists$  un littéral pur  $l$  then
    return Simplifier_R( $\Sigma(l), s$ )
  endif
  foreach  $\beta \in s$  do
    if  $\forall c \in \beta$ ,  $c$  est satisfait then
      return  $\{\}$  /* propriété 2 */
    endif
    if une seule clause  $c$  de  $\beta$  n'est pas satisfait then
       $\Sigma \leftarrow \Sigma \bigcup_{i=1}^n \{-l_i\}$  s.t.  $l_i \in \text{unset}(c)$ 
      return Simplifier_R( $\Sigma, s$ ) /* propriété 3 */
    endif
  endforeach
  return  $\Sigma$ 
end

```

Algorithme 2 – DPLL_R algorithm

Propriété 2 Soient Σ une formule CNF et une interprétation $I_d = \{l_1, \dots, l_{i-1}, l_i\}$. Si $\exists \beta_j (j \leq i)$ tel que $I_d[\beta_j] = \text{vrai}$ et l_j est un littéral actif de I_d alors Σ_{i+1} est insatisfiable.

La propriété 2 montre clairement comment exploiter, en pratique, le principe de la recherche complémentaire. En effet, à chaque nœud de l’arbre de recherche, il est possible de détecter l’insatisfiabilité de la formule courante et ainsi d’effectuer un retour arrière.

Un autre aspect important est la possibilité de déduire des littéraux unitaires. Lorsque toutes les clauses $c \in \beta$ sont satisfaites exceptée une seule clause c , alors on peut impliquer la négation de littéraux non affectés de cette clause (propriété 3).

Propriété 3 Soient Σ une formule CNF et une interprétation $I_d = \{l_1, \dots, l_{i-1}, l_i\}$. $\forall k \leq i$ si $\exists b \in \beta_k$ tel que le littéral l_k est actif dans I_d et $\forall c \in \beta_k$ tel que $c \neq b$ et $I_d[c] = \text{vrai}$, alors $\Sigma_{i+1} \models \{\neg l/l \in \text{unset}(I_d, b)\}$.

En conséquence la fonction *Simplifier()* est modifiée. À chaque étape, on vérifie s’il n’existe pas un ensemble de clauses satisfait (par l’interprétation courante) parmi les ensembles de clauses empilés précédemment, auquel cas on effectue un retour arrière. Sinon si l’on trouve un ensemble de clauses dont une seule clause n’est pas satisfaite, on propage tous les opposés des littéraux de cette clause. Pour plus de détails sur l’implémentation voir l’algorithme 2.

6 Résultats expérimentaux

Dans cette section, nous montrons quelques résultats expérimentaux de l’algorithme 2 sur un ensemble de classes d’instances. Le but de ces tests est de montrer la faisabilité et l’intérêt de notre approche. Pour cela, nous avons utilisé une version basique de la procédure DPLL sans aucune technique ou optimisation particulière comme l’on peut trouver dans les meilleures implémentations actuelles des DPLL (comme la technique de « restart » ou les techniques classiques d’analyse de conflits, etc.).

Nous avons fait une comparaison entre la procédure basique DPLL et celle qui implémente la propriété 1 sur des instances issues des bibliothèques de compétitions DIMACS et SAT [6, 11]. Certaines de ces instances sont issues d’applications réelles et d’autres sont de nature académique.

Dans la table 1, nous montrons l’intérêt de notre approche sur certaines classes d’instances et les gains que l’on peut obtenir en nombre de nœuds (*#nuds*) et en temps¹ (*tps*). Pour chaque instance considérée, nous avons détaillé le nombre de variables (*#V*), le nombre de clauses (*#C*), SAT (resp. UNSAT) si l’instance est satisfiable (resp. insatisfiable). Pour l’algorithme DPLL_R, nous avons également indiqué le nombre de coupures (*#coup.*) réalisées par la propriété 2 et le nombre de littéraux impliqués (*#imp.*) par la propriété 3.

¹Un temps CPU limite a été fixé à 12 heures sur des PC à 2,4Ghz.

Tout d’abord, pour toutes les instances testées, on remarque que la taille de l’arbre de recherche (en nombre de nœuds) est inférieure ou égale à la version de base DPLL. On remarque un surcoût raisonnable dans le cas où la coupure ne s’applique pas.

Sur de nombreuses classes d’instances, des améliorations importantes sont obtenues en temps et/ou en nombre de nœuds. Par exemple, on peut noter que l’instance *grid-10-15* a été résolue en environ demi-heure par DPLL_R alors que douze heures n’ont pas suffi à la version DPLL pour résoudre l’instance.

Par ailleurs, nous avons également constaté l’application très fréquente des propriétés 2 et 3 sur des instances de très grande taille où le temps limite utilisé est insuffisant pour leur résolution. Ceci nous conduit à envisager d’intégrer les propriétés 2 et 3 dans les meilleurs solveurs actuels afin d’élargir la classe des problèmes traitables en pratique.

7 Conclusions et perspectives

Dans ce papier, nous avons montré qu’il est possible d’éviter certaines redondances dans la recherche effectuée par des procédures classiques DPLL en intégrant efficacement le principe de recherche complémentaire proposé par Purdom. Notre approche s’avère très efficace d’un point de vue pratique dans le sens où elle permet de couper des branches dans l’arbre de recherche tout en ayant un bon compromis en temps et en espace. En effet, certaines des redondances rencontrées au cours de la recherche peuvent être évitées sans pour autant rajouter de nouvelles contraintes (clauses). Ce qui nous permet d’éviter les inconvénients dus à leur production et à leur ajout dans la base de clauses. De plus, cette technique permet non seulement d’élaguer l’arbre de recherche en effectuant des retours-arrières, mais aussi de déduire des nouveaux littéraux unitaires utiles pour la suite de la recherche.

Les résultats expérimentaux obtenus montrent l’intérêt de notre approche pour résoudre certaines classes d’instances structurées. Ceci nous conduit à envisager son intégration dans les meilleurs solveurs actuels de SAT comme Zchaff [16], et élargir ainsi la classe des problèmes traitables en pratique. Signalons que les solveurs actuels utilisent de nombreux paradigmes importants comme les « restart », l’apprentissage à partir d’échecs en ajoutant des « no-goods », des structures de données particulières, etc.

Une autre piste concernerait le développement d’heuristiques adaptées dans le but d’utiliser encore plus efficacement cette coupure dans l’arbre de recherche. C’est-à-dire, choisir un ordre d’instanciation des variables, favorisant l’application du principe de recherche complémentaire. L’exemple 2 montre bien la relation importante existant entre l’ordre d’instanciation des variables et l’effet important du choix de la variable dans le but de déduire des informations intéressantes. Enfin, nous pensons qu’il serait intéressant d’étudier les relations pouvant exister entre le principe de recherche complémentaire et les techniques d’apprentissage à partir d’échecs.

instance	#V	#C	SAT	DPL		DPL_R			
				#noeuds	tps	#noeuds	#coup.	#imp.	tps
2bitadd_10	590	1422	UNSAT	850301547	26112	433140747	67939200	54558000	16230
2bitadd_11	649	1562	UNSAT	-	-	581730242	105208878	71168070	36695
2bitadd_12	708	1702	UNSAT	-	-	581730267	105208878	71168070	38109
2bitcomp_5	125	310	SAT	75	0.00	75	0	15	0.00
2bitmax_6	252	766	SAT	2226	0.07	2194	60	0	0.07
qg0-7	686	6816	SAT	3747681	1370	3030547	388947	339487	1217
ii32b4	381	6918	SAT	1620	0.63	1560	35	2	0.65
ii32d2	404	5153	SAT	68217	10.34	67338	261	94	11.41
aim-100-1_6-no-1	100	160	UNSAT	13873294	87.38	5521257	226738	1934381	43.71
aim-100-1_6-yes1-1	100	160	SAT	20	0.00	17	2	2	0.00
aim-100-2_0-no-1	100	200	UNSAT	4475735	32.16	2047429	99946	796458	19.01
aim-100-2_0-yes1-1	100	200	SAT	38965	0.33	16803	532	5021	0.17
aim-200-1_6-yes1-1	200	320	SAT	25	0.00	23	0	6	0.00
aim-200-2_0-yes1-1	200	400	SAT	1309265	23.78	547623	14212	106007	11.41
aim-50-1_6-no-3	50	80	UNSAT	17611	0.06	4693	594	1895	0.02
aim-50-1_6-yes1-3	50	80	SAT	54	0.00	51	2	7	0.00
aim-50-2_0-no-1	50	100	UNSAT	3791	0.02	2598	196	700	0.01
aim-50-2_0-yes1-1	50	100	SAT	358	0.00	212	14	49	0.00
c499_gr_rcs_w5	1560	15777	UNSAT	239	0.25	239	0	8	0.25
c499_gr_rcs_w6	1872	18870	SAT	5070	1.53	4445	51	673	1.62
example2_gr_rcs_w5	2220	23144	UNSAT	19565	19.75	17411	2045	1491	19.80
example2_gr_rcs_w6	2664	27684	SAT	787	0.13	787	0	1	0.14
dlx1_c	295	1592	UNSAT	36952631	662.53	24874151	1153	252691	563.77
2dlx_cc_mc_ex_bp_f2_bug011	4824	48160	SAT	3565	13.81	3565	0	0	23.06
2dlx_cc_mc_ex_bp_f2_bug012	6598	72227	SAT	20645	15.76	10751	26	0	13.07
grid_05_05	48	81	UNSAT	1012	0.01	242	83	20	0.00
grid_05_10	98	171	UNSAT	69884	1.25	10172	5223	585	0.20
grid_10_10	198	361	UNSAT	42035094	1713.35	654006	284255	14674	27.94
grid_10_15	298	551	UNSAT	-	-	29343066	11272261	319736	1902
grid_10_20	398	741	UNSAT	-	-	367080720	128902981	52173448	32615

TAB. 1 – DPLL contre DPLL_R

8 Remerciements

Ce travail est soutenu en partie par l’IUT de Lens et la Région Nord/Pas-de-Calais dans le cadre du programme TACT-TIC.

Références

- [1] Jr. Bayardo, J. Roberto, and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI’97)*, pages 203–208, Providence (Rhode Island, USA), Juillet 1997.
- [2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Design Automaton Conference, DAC’99*, 1999.
- [3] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York (USA), 1971.
- [4] J.M. Crawford and A.B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Twelfth National Conference on Artificial Intelligence, AAAI’94*, 1994.
- [5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5 :394–397, 1962.
- [6] Dimacs. Second Challenge on Satisfiability Testing organized by the Center for Discrete Mathematics and Computer Science of Rutgers University, 1993. <http://dimacs.rutgers.edu/Challenges/>.
- [7] O. Dubois, P. André, Y. Bouffkhad, and J. Carlier. Sat versus unsat. In D.S. Johnson and M.A. Trick, editors, *Second DIMACS Challenge, 1993*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pages 415–436, 1996.
- [8] O. Dubois and G. 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), Août 4–10 2001.
- [9] G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming*, 7(1) :45–61, Juillet 1989.
- [10] C.P. Gomes, S. Bart, and H. Kautz. Boosting combinatorial search through randomization. In *Procee-*

dings of the 15th National Conference on Artificial Intelligence, (AAAI'98), pages 431–437, 1998.

- [11] H. Hoos and T. Stützle. The satisfiability library (satlib). <http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/>.
- [12] R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1 :167–187, 1990.
- [13] H. Kautz and B. Selman. Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, Vienna, Austria, 1992.
- [14] T. Larrabee. Efficient generation of test patterns using boolean satisfiability. *IEEE Transaction on CAD*, 11 :4–15, 1992.
- [15] F. Massacci and L. Marraro. Logical cryptanalysis as a sat-problem : Encoding and analysis of the u.s. data encryption standard. *Journal of Automated Reasoning*, 2000.
- [16] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
- [17] R. Ostrowski, E. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *Proc. of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, pages 185–199, Ithaca (N.Y.), Septembre 2002. LNCS 2470, Springer Verlag.
- [18] P. W. Purdom. Solving satisfiability with less searching. *IEEE transactions on pattern analysis and machine intelligence*, PAMI-6(4) :510–513, Juillet 1984.
- [19] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 43(2) :125–155, 1989.
- [20] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification*, pages 480–494, 2000.
- [21] J.P.M. Silva and K.A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, 1996.
- [22] J.P.M. Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the IEEE/ACM Design Automation Conference, DAC'00*, Juin 2000.
- [23] H. Zhang and J. Hsiang. Solving open quasigroup problems by propositional reasoning. In *International Computer Symposium, Hsinchu, Taiwan*, 1994.