Vivification de formules propositionnelles clausales

Cédric Piette, Youssef Hamadi, Lakhdar Saïs

Vivification de formules propositionnelles clausales

Cédric Piette[1] **Youssef Hamadi**[2] **Lakhdar Saïs**[1]

[1] Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens

[2] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom

{piette,sais}@cril.fr youssefh@microsoft.com

Résumé

Dans cet article, nous présentons une nouvelle technique de prétraitement d'une formule booléenne mise sous forme normale conjonctive (CNF). À la différence de la plupart des approches actuelles, notre procédure vise à améliorer la capacité de propagation des clauses de la formule tout en produisant un nombre limité de clauses utiles. Plus précisément, un test incomplet de redondance est effectué sur chaque clause d'une formule. Celui-ci est basé sur la propagation unitaire, et peut conduire soit à la production d'une sous-clause ou à la génération d'une nouvelle clause, via les techniques d'apprentissage implémenté dans les solveurs modernes. Cette nouvelle approche est comparée empiriquement au meilleur préprocesseur actuel, en terme de réduction de la formule considérée et d'amélioration pratique de sa résolution par l'un des meilleurs solveurs SAT.

Abstract

In this paper, we present a new way to preprocess Boolean formulae in Conjunctive Normal Form (CNF). In contrast to most of the current pre-processing techniques, our approach aims at improving the filtering power of the original clauses while producing a small number of additional and relevant clauses. More precisely, an incomplete redundancy check is performed on each original clauses through unit propagation, leading to either a sub-clause or to a new relevant one generated by the clause learning scheme. This preprocessor is empirically compared to the best existing one in term of size reduction and the ability to improve a state-of-the-art satisfiability solver.

1 Introduction

Depuis plusieurs années, les traitements préliminaires de formules CNF ont été de plus de plus étudiées par la communité SAT. Cet intérêt peut être expliqué par différents facteurs. Tout d'abord, la capacité des préprocesseurs à réduire les formules CNF de grande taille provenant de l'encodage de problèmes réels permet d'accroître la robustesse des solveurs SAT. En outre, l'élimination de clauses ou l'ajout de nouvelles clauses pertinentes peut permettre de mieux guider la recherche dans des zones importantes de d'espace de recherche, et renforcer l'efficacité des schémas d'apprentissage. Les techniques de prétraitement permettent également de gérer efficacement différents types de structures contenues dans les formules CNF, qu'il serait difficile de prendre en considération pendant la recherche.

L'une des techniques de prétraitement les plus efficaces, appelée `SatElite` [7], est à présent intégrée dans les meilleurs solveurs SAT actuels, tels que `Minisat` [8] et `RSAT` [14]. Il est maintenant admis que les performances de ces solveurs sont dans la majorité des cas grandement améliorées par ce prétraitement particulier, à tel point que `SatElite` est de plus en plus souvent utilisé par les participants de la compétition SAT.

Ainsi, prétraiter une formule CNF avant sa résolution est maintenant connu comme une étape importante, et un grand nombre de préprocesseurs ont été

proposés par le passé. L'un des premiers algorithmes de prétraitement efficace, appelé `3-Resolution`, a été proposé en conjonction du solveur `Satz` [12]. Celui-ci consiste à ajouter à la formule toutes les clauses résolvantes de taille inférieure ou égale à 3, jusqu'à saturation. De plus, toutes les clauses subsumées sont détectées et supprimées de la CNF. Un préprocesseur moins gourmand en ressources a ensuite été proposé dans [3]. Du nom de `2-SIMPLIFY`, il a été développé afin d'améliorer l'efficacité des solveurs sur les problèmes issus du monde réel, qui contiennent souvent un grand nombre de clauses binaires (i.e. contenant exactement 2 littéraux). Sommairement, l'idée est donc d'utiliser ces clauses binaires pour construire un graphe d'implication, duquel on peut déduire des clauses unitaires par le calcul de la fermeture transitive. Si de telles clauses sont produites, elles sont propagées et ce procédé est itéré jusqu'à ce qu'un point fixe soit atteint. Plus tard, un nouveau préprocesseur baptisé `HyPre` a généralisé `2-SIMPLIFY` en utilisant l'hyper-résolution pour déduire de nouvelles clauses binaires [1]. De plus, cet algorithme détecte et substitue les littéraux équivalents de manière incrémentale. En conséquence, `HyPre` se montre plus efficace que son prédécesseur pour la plupart des formules traitées.

La « classique » procédure DP, basée sur l'élimination de variables par résolution, a également été considérée comme prétraitement d'une CNF. À cause de sa complexité exponentielle en espace, un schéma plus faible a été adopté par la procédure `NiVER` [15]. Celle-ci élimine les variables par résolution si ce calcul ne conduit pas à augmenter le nombre de littéraux de la CNF. Appliquer `NiVER` avant de résoudre une formule permet de grandement améliorer les performances des solveurs modernes. Celle-ci a ensuite été améliorée par l'utilisation d'une nouvelle règle de substitution et de *signatures* de clauses [7], pour former le désormais célèbre préprocesseur `SatElite`.

A l'heure actuelle, seuls les préprocesseurs basés sur l'élimination de variables, comme `SatElite`, via une forme limitée de résolution sont maintenant greffés aux solveurs modernes. En effet, les autres techniques visent à modifier la formule CNF par l'ajout/le retrait de clauses, en conservant généralement le même ensemble de variables. L'un des principaux problèmes de ces techniques est qu'il est difficile de mesurer la pertinence d'une clause, qu'elle soit ajoutée ou éliminée, par rapport à l'étape de résolution. Certaines stratégies d'élimination de clauses peuvent rendre une formule plus difficile à résoudre. De manière similaire, l'ajout de nouvelles clauses peut conduire à l'augmentation de la complexité spaciale sans réduire l'espace de recherche. En effet, dans certains cas, les clauses ajoutées ne font que gêner le solveur par la génération d'information redondante inutile.

Dans cet article, nous revisitons cette catégorie de préprocesseurs, en utilisant uniquement des formes de résolution qui permettent de substituer les clauses de la formule par des clauses plus contraintes. En d'autres termes, notre but principal de renforcer, ou *vivifier*, les clauses redondantes d'une formule CNF. Dans cet objectif, nous appliquons une forme limitée de test de redondance sur chaque clause de la formule CNF, pour dériver, ou à défaut faire l'approximation de l'une de ses sous-clauses minimalement redondantes. L'approche proposée permet également de bénéficier des techniques d'apprentissage des solveurs modernes pour produire de nouvelles résolvantes qui sont conditionnellement ajoutées à la formule.

Ce papier est organisé comme suit : la section suivante présente les définitions de bases et les notations utilisées à propos de la logique propositionnelle et SAT. Dans la section 3, l'intérêt pratique de différentes techniques de simplifications est discuté. Ensuite, des formes particulères de résolution, dissimulées par la propagation unitaire, sont présentées, et une méthode incomplète pour leur production est proposée. Le préprocesseur résultant est détaillé et évalué dans la section 4. Enfin, nous concluons par des perspectives de recherches futures.

2 Définitions et notations

Nous établissons ici quelques définitions et notations utilisées dans la suite de ce papier. Soit $\mathcal{L}$ le langage propositionel standard, construit sur un ensemble fini de variables Booléennes. Une formule propositionnelle est sous forme normale conjonctive (CNF) si elle peut être représentée par un ensemble (interprété comme conjonction) de clauses, où une clause est un ensemble (interprété comme disjonction) de littéraux, un littéral étant une variable propositionnelle ou sa négation. L'ensemble des variables apparaissant au sein d'une formule CNF Σ est noté $Var(\Sigma)$. $Lit(\Sigma)$ est défini comme l'ensemble $\{x, \neg x | x \in Var(\Sigma)\}$. Pour un ensemble de littéraux L, $\bar{L}$ est défini comme suit : $\{\neg l | l \in L\}$.

Une *interprétation* ρ d'une formule CNF Σ est une application de $Var(\Sigma)$ dans l'ensemble des valeurs de vérité $\{vrai, faux\}$. Elle est en outre appelée *modèle* de Σ ssi elle donne la valeur *vrai* à Σ (noté $\rho \models \Sigma$). Si une formule CNF admet au moins un modèle, alors est dite *satisfiable*, sinon elle est dite *insatisfiable*. SAT est le problème qui consiste à décider si une formule CNF donnée est satisfiable ou non.

Soit $c_a = \{l_{a_1}, \ldots, l_{a_n}, l\}$ et $c_b = \{l_{b_1}, \ldots, l_{b_m}, \neg l\}$ deux clauses. La clause $c = \{l_{a_1}, \ldots, l_{a_n}, l_{b_1}, \ldots, l_{b_m}\}$ est une conséquence logique, (appelée *résolvante*) de

c_a et c_b. Cette règle de production est appelée *résolution* et est notée $\otimes_R$. La résolvante c est notée $c_a \otimes_R c_b$. Dans la suite, nous considérerons uniquement les résolvantes fondamentales, c'est-à-dire les clauses produites ne contenant pas un littéral et son opposé. La plupart des techniques utilisées pour la résolution de SAT (par exemple la procédure DP, la propagation unitaire, les techniques d'apprentissage, etc.) sont basées sur une application implicite ou explicite de la résolution. C'est clairement le cas de la plupart des préprocesseurs, dont celui présenté dans cet article.

Soient c et c' deux clauses de Σ. On dit que c' (resp. c) subsume (resp. est subsumée par) c (resp. c') ssi $c' \subset c$. Toute clause subsumée peut être ôtée de Σ tout en préservant sa satisfiabilité. Soit Σ une formule CNF et $x \in Lit(\Sigma)$. On définit $\Sigma|_x$ comme la formule simplifiée par l'affectation de x à la valeur *vrai*. De plus, on définit récursivement $PU(\Sigma)$ comme suit : (1) $PU(\Sigma) = \Sigma$ si Σ ne contient pas de clause unitaire, (2) $PU(\Sigma) = \bot$ si Σ contient deux clauses unitaires $\{x\}$ et $\{\neg x\}$, (3) sinon, $PU(\Sigma) = PU(\Sigma|_x)$ avec x apparaissant dans une clause unitaire de Σ. Une clause c est impliquée par propagation unitaire de Σ, noté $\Sigma \models_{PU} c$, si $PU(\Sigma|_{\bar{c}}) = \bot$.

Dans la section suivante, les principales stratégies de prétraitement sont discutées, et une forme limitée de résolution permettant la production de clauses plus contraintes que les clauses de la formule originale est présentée.

3 Prétraiter une formule CNF

3.1 Ajouter et/ou supprimer des clauses ?

Deux principales catégories de préprocesseurs ont été proposées, les deux étant basées sur la résolution. La première vise à éliminer des variables de la formule via une application partielle de la procédure DP [6]. Plus précisément, seules les variables pouvant être éliminées en conservant une formule de taille « raisonnable » (par rapport à la taille de la formule initiale) sont exhaustivement traitées par résolution. `SatElite` appartient à cette catégorie de préprocesseurs.

Le principe de la seconde catégorie est de modifier la formule par l'ajout et/ou le retrait de clauses, en conservant en général l'ensemble des variables de la formule. La plupart du temps, la production de nouvelles clauses est effectuée par résolution. Par exemple, `HyPre` effectue de l'hyper-résolution pour produire de nouvelles clauses binaires [1] qui sont ajoutées à la formule. Ces nouvelles clauses représentent de l'information redondante par rapport à la formule CNF initiale, cette information supplémentaire aidant en général le solveur dans son parcours de l'espace de recherche.

Récemment, une nouvelle approche introduite dans [9] a permis la détection et le retrait de certaines clauses redondantes, c'est-à-dire de clauses c de Σ telles $\Sigma \backslash \{c\} \models c$. Assez clairement, effectuer un tel test n'est pas efficace, d'un point de vue calculatoire. En conséquence, la redondance est uniquement testée à travers l'application de la propagation unitaire. De ce fait, cette approche est incomplète, mais permet le retrait d'un grand nombre de clauses redondantes en temps polynomial. Toutefois, tout comme les techniques basées sur l'ajout de clauses, le préprocesseur résultant peut ralentir l'étape de résolution.

Le principal problème de ces techniques est qu'il est difficile de caractériser quelles clauses redondantes vont d'avérer utiles. En effet, un compromis doit être fait entre la gestion d'un grand nombre de clauses, ce qui ralentit les implémentations de DPLL, et leur pertinence, c'est-à-dire leur capacité à déclencher des propagations de littéraux. En effet, il est bien connu que de l'information redondante peut effectivement aider les solveurs SAT ; par exemple, le schéma d'apprentissage des solveurs modernes, produisant une clause résolvante particulière après chaque conflit, peut être vu comme un ajout dynamique d'information redondante pendant la recherche. Cette stratégie d'apprenstissage est maintenant considérée comme l'une techniques clés des méthodes actuelles [2], ce qui montre l'intérêt de l'information redondante pour la résolution pratique de SAT. Néanmoins, une simple expérimentation consistant à ajouter à une formule CNF toutes les clauses apprises durant une résolution précédente la rend en général plus difficile à résoudre. Dès lors, comment s'assurer qu'une approche préliminaire ajoutant des clauses va effectivement accélérer le processus de résolution ?

A priori, une solution intéressante consisterait en la génération efficace de sous-clauses de la formule. De cette façon, il y a ni ajout ni retrait de clauses, mais substitution des clauses existantes par de plus contraintes. Par rapport aux procédés actuels de résolution de SAT, un tel calcul peut avoir de grands avantages : non seulement il permettrait d'accroître le nombre de propagations unitaires sans clause supplémentaire à gérer, mais conduirait également à la génération de clause apprises plus courtes, en réduisant la *raison* des littéraux propagés. Plusieurs techniques ont déjà été proposées pour la production de telles sous-clauses. Il est par exemple proposé dans [5] de générer des clauses résolvantes et de ne conserver que celles qui subsument au moins une clause de l'instance originale, dans le but de la substituer. Dans la section suivante, une nouvelle approche visant à tester plus systématiquement si une clause peut être raccourcie est présentée. De surcroît, cette technique peut bé-

néficier des stratégies d'apprentissage communément implémentées au sein des solveurs pour produire de nouvelles clauses intéressantes.

3.2 Une première réponse : raccourcir les clauses existantes

La façon dont est encodé un problème en CNF est cruciale pour sa résolution pratique, et peut conduire à des différences d'ordre exponentielles en ressources. Analyser les différentes formes de modélisation est maintenant une piste de recherche très active (cf. par exemple [10]).

Toutefois, même avec de « bonnes » modélisations, certaines clauses d'une formule CNF peuvent être redondantes. On dit qu'une clause est redondante si elle peut être inférée à partir des autres clauses de la formule CNF. Une telle clause peut être éliminée de la formule tout en préservant sa satisfiabilité. Cependant, comme mentionné plus haut, ce retrait de clauses redondantes d'une formule CNF peut mener à une dégradation de sa résolution pratique. Dans l'approche proposée dans ce papier, le test de redondance est effectué uniquement pour raccourcir des clauses en éliminant certains de ses littéraux.

Toutefois, tester si une clause est redondante est une tâche de complexité CoNP-complet [13]. Par conséquent, une technique incomplète mais linéaire en temps a été adoptée. En effet, ce test est uniquement effectué à travers la propagation unitaire. Plus formellement, une clause c d'une formule Σ est redondante modulo la propagation unitaire (noté $Red_{PU}(\Sigma, c)$), ssi $\Sigma \backslash \{c\} \models_{PU} c$. Évidemment, si $Red_{PU}(\Sigma, c')$, et $c' \subset c$, alors on a également $Red_{PU}(\Sigma, c)$. L'opposé n'est bien sûr pas vérifié. Cette observation conduit à une nouvelle définition de redondance minimale de clauses. On dit qu'une clause $c \in \Sigma$ est minimalement redondante modulo PU ssi $\nexists c' \subset c$ t.q. $Red_{PU}(\Sigma, c')$.

L'un des buts principaux de notre processus de vivification est de trouver pour chaque clause redondante l'une de ses sous-clauses minimales redondantes. Par ailleurs, le schéma classique d'apprentissage est utilisé pour dériver de nouvelles clauses intéressantes. Notre but est donc d'éliminer des littéraux des clauses d'une formule.

Plus précisément, quand une clause est vérifiée quant à sa redondance, elle est retirée de sa formule CNF, et l'opposé de chacun de ses littéraux est affecté un à un suivant leur ordre lexicographique. Si l'application de la propagation unitaire permet d'atteindre un conflit, alors la clause testée est redondante modulo PU. Étant données une formule CNF Σ et l'une de ses clauses $c = \{l_1, l_2, \ldots, l_n\}$. En supposant que l'ordre dans lequel sont affectés est $(\neg l_1, \ldots, \neg l_n)$, deux cas sont à considérer :

1. $\exists i \in \{1, \ldots, n-1\}$ t.q. $\Sigma \backslash \{c\} \cup \{\neg l_1, \ldots, \neg l_i\} \vdash_{PU} \bot$
   Dans ce cas, on a :
   $\Sigma \backslash \{c\} \vdash_{PU} c'$ avec $c' = (l_1 \vee \ldots \vee l_i)$

   Cette nouvelle clause c' subsume strictement c. Ainsi, la clause originale peut être substituée par la nouvelle clause déduite. Évidemment, c' n'est pas nécessairement minimalement redondante pour la propagation unitaire. En effet, un ordre d'affectation différent des littéraux $\{l_1, l_2, \ldots, l_i\}$ pourrait conduire à la production d'une clause encore plus petite. En fait, une nouvelle clause η peut être générée par un parcours *complet* du graphe d'implication associé à Σ et des affectations des littéraux $\{\neg l_1, \ldots, \neg l_i\}$. Un tel parcours exhaustif du graphe d'implication garantit que la clause η contient exclusivement des littéraux de c'. Par conséquent, η est une sous-clause de $(l_1 \vee \ldots \vee l_i)$.

2. Comme la propagation unitaire est effectuée après chaque affectation, si l'un des littéraux non encore traités est affecté par cette opération de filtrage, alors une sous-clause peut être déduite. Trivialement, quand ce phénomène se produit, le littéral propagé est affecté soit positivement (il satisfait la clause retirée de la CNF) ou négativement (il est falsifié dans cette clause). En considérant, i et j avec $1 \leq i < j \leq n$, les deux cas possibles sont :

   - $\Sigma \backslash \{c\} \cup \{\neg l_1, \ldots, \neg l_i\} \vdash_{PU} \neg l_j$
     Dans ce cas, on peut déduire :
     $\Sigma \backslash \{c\} \vdash_{PU} (l_1 \vee \ldots \vee l_i \vee \neg l_j)$

     En appliquant la résolution entre cette nouvelle clause et c (en utilisant la variable l_j), on obtient :
     $(l_1 \vee \ldots \vee l_j \vee \ldots \vee l_n) \otimes_R (l_1 \vee \ldots \vee l_i \vee \neg l_j) =$
     $(l_1 \vee \ldots \vee l_{j-1} \vee l_{j+1} \vee \ldots \vee l_n)$. Cette nouvelle clause subsume clairement c. Ainsi, la clause initiale peut être substituée par la nouvelle clause déduite.

   - $\Sigma \backslash \{c\} \cup \{\neg l_1, \ldots, \neg l_i\} \vdash_{PU} l_j$
     Dans ce cas, on peut déduire :
     $\Sigma \backslash \{c\} \vdash_{PU} (l_1 \vee \ldots \vee l_i \vee l_j)$

     Encore une fois, la clause produite subsume c et permet de « lui ôter » certains littéraux.

De cette manière, à partir de l'affectation itérée de l'opposé des littéraux d'une clause, une sous-clause peut être produite. Ce calcul peut clairement être greffé au sein d'un solveur moderne, et bénéficier d'une structure de données paresseuse pour être effectué. De plus, pendant la recherche, certaines affectations

peuvent mener à un conflit. Comme expliqué précédemment, lorsque ce cas se produit, la procédure peut faire appel à l'analyse de conflit implémentée dans ces même solveurs pour produire une nouvelle clause (*nogood*) en temps polynomial.

En considérant les règles définies plus haut et l'utilisation de la partie apprentissage des solveurs SAT, une formule CNF peut être *vivifiée*, c'est-à-dire rendue plus simple à résoudre. Dans la section suivante, nous présentons l'implémentation qui a été réalisée à partir de ces règles de production de sous-clauses et les choix techniques qui ont été adoptés.

4 Vivification d'une formule

4.1 Choix techniques

Dans cette section, différents paramètres pratiques sont discutés, certains d'entre eux étant le fruit de tests d'expérimentaux intensifs.

Tout d'abord, les idées proposées dans les sections précédentes supposent de tester les clauses de la formule dans le but de les réduire. Cependant, si un littéral est oté d'une clause, de nouvelles propagations peuvent être produites par cette nouvelle clause, rendant caduques les échecs rencontrés lors des tests des précédentes clauses. Ainsi, dès qu'un test permet la production d'une nouvelle sous-clause, toutes les autres sont de nouveau testées, à travers une nouvelle *itération* de l'algorithme.

Ensuite, la technique de production de sous-clauses présentée induit que l'ordre dans lequel les opposés des littéraux d'une clause sont affectés est important. Assez clairement, pour assurer une réduction maximale de la clause, tous les ordres possibles doivent être testés. Cela conduit malheureusement à un calcul relativement lourd ; nous avons donc adopté une stratégie incomplète qui consiste à n'essayer qu'un ordre particulier. En pratique, une variante de l'heuristique de choix de variable MOMS (Maximum Occurrences on clauses of Minimum Size) [11] est utilisée pour trier les littéraux et essayer de maximiser le nombre de littéraux impliqués par propagation unitaire.

La seule utilisation de cette heuristique conduit cependant à un ordre très similaire des littéraux d'une itération à la suivante. Même si une clause n'est testée de nouveau que si au moins une autre clause de la formule a été raccourcie, le tri selon MOMS ne semble pas une bonne solution, car la procédure ne bénéficierait pas des possibles multiples itérations de l'algorithme. Pour diversifier la recherche, une dose de *randomisation* a été ajoutée comme suit : en supposant que les littéraux d'une clause soient triés selon MOMS, deux d'entre eux sont choisis aléatoirement et inversés dans cet ordre.

Enfin, quand un conflit survient, la clause testée $c = (l_1 \vee \ldots \vee l_n)$ est substituée par la sous-clause $c' = (l_1 \vee \ldots \vee l_i)$ (cf. section 3.2). Comme mentionné plus haut, un parcours complet du graphe d'implication peut mener à la détection d'une clause encore plus réduite, mais pour des raisons d'efficacité, ce calcul n'est pas effectué en pratique. Dans notre implémentation, le schéma classique d'apprentissage est utilisé pour générer un *nogood* η correspodant au premier UIP (Unit Implication Point) du graphe d'implication. Si cette nouvelle clause η subsume la sous-clause c', alors la clause c' est substituée par η. Dans le cas contraire, la clause η est ajoutée à la formule seulement si sa cardinalité (en terme de nombre de littéraux) est strictement inférieure à la taille de la clause testée. Comme nos résultats expérimentaux le montrent, cette stratégie permet de n'ajouter qu'un nombre limité de *nogoods* ($< 5\%$ du nombre de clauses initiales) qui se montrent particulièrement utiles pour la future recherche exhaustive.

En considérant ces choix, un nouveau préprocesseur polynomial baptisé `ReVivAl` (pour pReprocessing based on Vivification Algorithm) a été développé. Cette méthode est décrite dans l'Algorithme 1. Sommairement, pour chaque clause c d'une formule CNF Σ donnée en entrée, c est retirée de Σ et l'opposé de chacun de ses littéraux est alternativement affecté avec l'utilisation de la propagation unitaire (boucle de la ligne 5 à 37). De plus, différentes vérifications à propos des autres variables de la clause (qui « devraient » ne pas être encore affectées) et la présence d'un conflit sont faites, comme présenté dans la section 3.2 (tests des lignes 14, 16, 20, 23, 27 et 31). L'ordre dans lequel sont littéraux sont affectés est donné par la fonction *choisis_un_litteral* qui choisit juste le littéral non affecté ayant le plus haut score dans score MOMS, où deux littéraux ont leur score inversé. Tant qu'une clause est réduite (*reduit* mis à *vrai*), le processus est poursuivi avec toutes les autres clauses.

Notons que notre implémentation a été intégrée à un solveur moderne, permettant l'utilisation des structures de données et des mécanismes les plus récents destinés à la résolution de SAT. Ainsi, chaque test de redondance d'une clause, qui se traduit par une suite d'affectations, bénéficie de l'efficacité des techniques de *watched literals*. De la même manière, l'ajout conditionnel de clauses par notre technique préliminaire est effectuée *via* les fonctions d'apprentissage habituellement appelées par le solveur à chaque conflit. Cette exploitation des structures et techniques implémentées dans les méthodes exhaustives permet non seulement une implémentation à moindre coût de notre méthode au sein de la plupart des solveurs actuels, mais

Algorithme 1 : Vivification d'une formule CNF

Données : Σ : une formule CNF
Résultat : une formule CNF vivifiée

```
1  début
2      change ⟵ vrai ;
3      tant que change faire
4          change ⟵ faux ;
5          pour chaque c ∈ Σ faire
6              Σ ⟵ Σ\{c} ;
7              Σ_b ⟵ Σ ;
8              c_b ⟵ ∅ ;
9              reduit ⟵ faux ;
10             tant que (non(reduit) et (c ≠ c_b))
               faire
11                 l ⟵ choisis_un_litteral(c\c_b) ;
12                 c_b ⟵ c_b ∪ {l} ;
13                 Σ_b ⟵ (Σ_b ∪ {¬l}) ;
14                 si ⊥ ∈ PU(Σ_b) alors
15                     c_l ⟵ analyse_de_conflit() ;
16                     si c_l ⊂ c alors
17                         Σ ⟵ Σ ∪ {c_l} ;
18                         reduit ⟵ vrai ;
19                     sinon
20                         si |c_l| < |c| alors
21                             Σ ⟵ Σ ∪ {c_l} ;
22                             c_b ⟵ c ;
23                         si c ≠ c_b alors
24                             Σ ⟵ Σ ∪ {c_b} ;
25                             reduit ⟵ vrai ;
26                 sinon
27                     si ∃(l_s ∈ (c\c_b)) t.q. l_s ∈
                       PU(Σ_b) alors
28                         si (c\c_b) ≠ {l_s} alors
29                             Σ ⟵ Σ ∪ {c_b ∪ {l_s}} ;
30                             reduit ⟵ vrai ;
31                     si ∃(l_s ∈ (c\c_b)) t.q. ¬l_s ∈
                       PU(Σ_b) alors
32                         Σ ⟵ Σ ∪ {c\{l_s}} ;
33                         reduit ⟵ vrai ;
34                 si Non(reduit) alors
35                     Σ ⟵ Σ ∪ {c} ;
36                 sinon
37                     change ⟵ vrai ;
38     retourner Σ ;
39 fin
```

lui confère également leur efficacité pour les différents tests effectués.

L'approche `ReVivAl` est évaluée empiriquement dans la section suivante.

4.2 Expérimentations

Nous avons comparé `ReVivAl` avec le préprocesseur qui est à ce jour considéré comme la meilleure approche préliminaire, c'est-à-dire `SatElite`. Le solveur `RSAT` [14] a été choisi comme méthode exhaustive, car il est reconnu comme une implémentation robuste et s'est montré très adapté à la résolution efficace de problèmes structurés à la dernière compétition SAT. Toutes nos expérimentations ont été menées sur des processeurs Intel Xeon 3GHz sous Linux CentOS 4.1. (noyau 2.6.9) avec une limite de mémoire vive de 2Go. Pour toutes ces expérimentations, une limite de temps de 3 heures CPU a été respectée. Au delà de cette limite, the note *time out* est reportée.

Nous avons comparé ces préprocesseurs, à la fois au niveau de la réduction de la formule donnée en entrée et de leur impact sur l'efficacité de RSAT. Cette comparaison a été faite sur un très grand jeu de *benchmarks* issus des compétitions SAT, de la SAT Race, SATLIB et d'autres sources : plus de 5000 instances ont été utilisées pour ces tests expérimentaux qui ont nécessité près de 600 jours de temps CPU. Un échantillon des résultats obtenus, contenant des exemples auxquels nous feront référence par la suite est proposé dans la Table 1 Les résultats expérimentaux complets sont en outre disponibles à l'adresse :
`http://www.cril.fr/~piette/preprocessor.html`

La première partie de la Table 1 indique le nom du problème testé, ainsi que le nombre de clauses (*#cla*) et de littéraux (*#lit*) qu'il contient. Les deux autres parties de la table sont similaires (une pour chaque préprocesseur) et contiennent le temps de prétraitement (en seconde), la taille de la formule résultante en terme de nombre de clauses et littéraux, et le temps (en sec.) nécessaire à RSAT pour décider de la satisfiabilité de la formule prétraitée. En outre, pour `ReVivAl`, le nombre d'itérations effectuées et de clauses apprises sont reportées dans les colonnes intitulées « #ite » et « #nogood », respectivement. Pour une formule donnée, la meilleure approche est celle dont la somme du temps de prétraitement et de résolution est la plus petite. Les résultats de la meilleure approche sont reportés en gras.

Tout d'abord, concentrons-nous sur les instances pouvant être résolues simplement en étant prétraitées. En effet, il existe de tels problèmes, dont certains ont été proposés dans les compétitions SAT et/ou aux SAT Races. Etant données les caractéristiques des algorithmes exécutés, quand l'un d'eux (ou les 2)

Instance		SatElite			ReVivAl				
nom	(#cla,#lit)	temps	(#cla,#lit)	temps RSAT	temps	(#cla,#lit)	temps RSAT	#ite	#nogood
pbl-00250	(32700,256765)	0,33	(31115,245053)	time out	**59,8**	(34815,219076)	**613,89**	30	2785
velev-fvp-sat-3.0-07	(1012271,2979665)	**4,05**	(998048,3017403)	**9,84**	61,53	(990394,2928889)	16,04	2	19
alu4mul.miter	(30465,103040)	0,1	(30392,102976)	915,91	**17,99**	(28992,90194)	**670,66**	15	502
Composite-024BitPrimes-1	(11158,49842)	0,02	(10087,44762)	7403,21	**0,56**	(10728,35545)	**2013,16**	20	163
Composite-024BitPrimes-0	(11158,49842)	0,02	(10016,44487)	733,2	**0,48**	(10395,34668)	**16,59**	18	132
velev-eng-uns-1.0-04	(66654,188252)	**0,5**	(62239,201530)	**11,67**	60,09	(57518,157260)	14,24	15	89
3bitadd_31	(31310,86676)	0,38	(31186,108004)	5058,73	**10,17**	(33125,83346)	**1,71**	28	1815
SAT_dat.k1	(3868,9928)	0	Polynomial	-	0	Polynomial	-	-	-
c3540mul.miter	(33199,112244)	**0,13**	(33066,112216)	**1635,29**	3,3	(27206,80134)	2186,01	13	815
logistics-rotate-10t5	(338789,680799)	2,45	(317194,687554)	1250,5	**11,67**	(277860,558081)	**151,64**	1	0
ezfact16_3	(1113,4089)	0	(990,3586)	0	0	Polynomial	-	-	-
ezfact48_8	(11001,41369)	0,02	(9215,34333)	61,99	**1,37**	(9582,27880)	**46,71**	18	315
ezfact48_9	(11001,41369)	0,02	(10532,39464)	204,94	**1,26**	(9558,27858)	**82,46**	17	344
ezfact64_1	(19785,74601)	0,06	(16716,62498)	time out	**2,77**	(17265,50532)	**3049,66**	15	549
abb313GPIA-8-c	(426860,2561106)	1,47	(421719,2521104)	366,29	**60,88**	(404007,2340042)	**12,05**	28	236
qg7-10	(33736,89626)	**0,04**	(11038,27452)	**0,01**	0,15	Polynomial	-	-	-
color-10-3	(6475,25200)	**0,08**	(6175,32600)	**1,28**	0,06	(6475,25200)	19,41	1	0
grieu-vmpc-s05-27r	(96849,253854)	0,15	(96849,253854)	104,02	**25,87**	(96482,239730)	**51,76**	9	154
ferry12	(32199,71303)	0,21	(30268,68540)	8,28	**0,5**	(30405,67168)	**1,87**	1	12
mod2c-3cage-unsat-9-1	(464,1856)	**0**	(464,1856)	**2152,31**	0	(472,1533)	2942,17	9	8
544707209399nw	(18031,53975)	0,12	(16032,49234)	1477,7	**2,26**	(14768,34277)	**814,4**	9	628
rand_net40-60-10	(14321,33560)	0,1	(10778,29243)	486,48	**3,74**	(14321,33152)	**421,97**	8	0
abb313GPIA-8-cn	(693640,2080902)	**16,73**	(388404,2307599)	**143**	23,27	(677607,2017201)	2321,44	12	19
equilarge_m1	(11489,33442)	0,07	(11158,41295)	time out	0,12	(11489,33164)	time out	5	0
hanoi5u	(73777,160717)	**0,29**	(61778,135270)	**183,74**	1,18	(59467,129001)	284,83	1	9
shuffling-2-s1765005333	(30465,103040)	0,13	(30392,102976)	1380,07	**23,93**	(28975,89789)	**1023,51**	11	592
lksat-n2200. . .s1262048766	(7524,22572)	0,04	(6322,19609)	201,38	**0,19**	(7629,20439)	**59,91**	16	105
3pipe_3_ooo	(33270,95618)	**0,19**	(31735,101212)	**9,67**	30,86	(31150,87665)	9,13	33	60
gripper12	(30746,68144)	0,12	(28060,62815)	time out	**0,26**	(27871,61632)	**676,2**	1	2
gripper13	(40461,89385)	0,16	(37437,83384)	1300.36	**0,37**	(37195,82030)	**1183,14**	1	4

TAB. 1 – SatElite VS ReVivAl

réussi(ssen)t à prouver l'(in)cohérence d'une formule CNF, cela signifie clairement que le problème est soluble en temps polynomial (indiqué par *Polynomial* dans la table). L'intérêt de ces formules pour les comparaisons expérimentales de méthodes systématiques pour SAT est sujet à débat, car elles ne présentent aucune difficulté combinatoire, ce qui *devrait* être le point de comparaison clé des approches exhaustives. Parmi les formules CNF testées, `SatElite` (respectivement `ReVivAl`) a prouvé 35 (resp. 167) instances polynomiales. Notons de plus que pour les 2 préprocesseurs, ces calculs sont le plus souvent effectués en quelques secondes (cf. par exemple `SAT_dat.k1`, `ezfact16_3`).

Ensuite, considérons la taille de la formule CNF après avoir été prétraitée. Des différences notables peuvent être observées entre les 2 approches. En effet, d'un côté, l'objectif de `SatElite` est l'élimination de variables sans augmentation de la taille de la formule CNF (en nombre de clauses). En conséquence, si le nombre de clauses de la formule obtenue est environ le même, elle possède en général un nombre plus élevé de littéraux. A l'opposé, `ReVivAl` tente de minimiser la taille des clauses de la formule et d'en ajouter un nombre limité. De ce fait, la formule simplifiée peut parfois contenir un nombre légèrement supérieur de clauses, mais le nombre moyen de littéraux par clause est typiquement réduit, rendant l'exploitation de ces clauses plus efficace par rapport à la propagation unitaire. Par exemple, sur le problème `alu4mul.miter` qui contient 30465 clauses et 103040 littéraux (taux #lit/#cla = 3,38), `SatElite` élimine des variables en conversant environ le même nombre de clauses et de littéraux alors que `ReVivAl` retourne une plus petite formule CNF en nombre de clauses (28992) pour un taux égal à 3,11. Dans certains cas, `SatElite` fournit une formule avec un taux bien supérieur à celui d'origine (cf. `3pipe_3_ooo` et `3bitadd_31`), mais cela ne se produit pas avec `ReVivAl`.

D'une manière plus générale, en écartant les problèmes ne pouvant être résolus par l'effet d'aucun des deux préprocesseurs en conjonction de RSAT, un gain de temps de 18,8% peut être observé en faveur de `ReVivAl`. De surcroît, avec l'usage de `SatElite`, RSAT ne parvient pas à résoudre 2508 instances de notre jeu de tests en 3 heures de temps (prétraitement *et* solveur), tandis que ce même solveur n'a échoué que pour 2457 instances avec notre approche. Toutefois, même si le plus souvent le traitement de `ReVivAl` a un meilleur effet que celui de `SatElite`, des contre-exemples peuvent évidemment être trouvés (`hanoi5u` et `abb313GPIA-8-cn`). Néanmoins, de nombreuses classes de problèmes sont typiquement plus sensibles à `ReVivAl`, qui semble supérieur à `SatElite` à accélérer la résolution par RSAT. Par exemple, sur les encodages booléens de problèmes de factorisation de circuits `ezfact-*`, sur les problèmes `Composite-*BitPrimes` suggérés comme challenge pour les solveurs SAT 1997 par Cook et Mitchell [4], et sur les problèmes de plannification `gripper*`, l'approche proposée dépasse clairement `SatElite`.

5 Conclusion

Dans cet article, nous avons présenté une nouvelle technique de prétraitement de formules CNF basée sur une forme limitée de résolution, dissimulée par la propagation unitaire et l'analyse de conflit. Notre approche, appellée vivification, utilise de manière originale des tests de redondance pour produire des sous-clauses et ajouter de nouvelles clauses issues des techniques d'apprentissage des solveurs modernes. Son efficacité a été illustrée par des expérimentations intensives avec une implémentation récente et robuste des techniques actuelles (RSAT). Une comparaison avec le meilleur préprocesseur actuel (`SatElite`) montre que notre algorithme, `ReVivAl`, simplifie efficacement certaines instances et familles d'instances, comme les problèmes d'encodage de factorisation de circuits ou de plannification.

Ces résultats ouvrent de nombreuses perspectives de recherche. L'intérêt pratique de la vivification a été empiriquement démontré, mais il nous est apparu que la combinaison de techniques de prétraitement peut permettre des améliorations encore plus grandes pour la résolution d'un problème. Une combinaison particulière de `ReVivAl` et de `SatElite` a d'ailleurs permis de dépasser toutes les approches proposées à la première phase de qualification de la SAT RACE 2008, en résolvant 48 instances sur les 50 proposées. La sélection dynamique de préprocesseurs par des techniques hyper-heuristiques semble être une piste prometteuse de recherche. Enfin, une intégration fine de `ReVivAl` au sein de solveurs modernes et son utilisation périodique à chaque redémarrage (*restart*) de la résolution sont plannifiées comme futures études.

Références

[1] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355, Santa Margherita Ligure (Italie), 2003. Springer.

[2] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International*

Joint Conference on Artificial Intelligence (IJCAI'03), pages 1194–1201, Acapulco (Mexique), 2003.

[3] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. In *17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 515–522, Seattle (USA), 2001.

[4] Stephen A. Cook and David G. Mitchell. Finding hard instances of the satisfiability problem : A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35, 1997.

[5] Sylvain Darras, Gilles Dequen, Laure Devendeville, Bertrand Mazure, Richard Ostrowski, and Lakhdar Sais. Using boolean constraint propagation for sub-clause deduction. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 757–761, Sitges (Espagne), 2005.

[6] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3) :201–215, Juillet 1960.

[7] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 61–75, St. Andrews (Écosse), 2005.

[8] Niklas Eén and Niklas Sorensson. Minisat home page http ://www.cs.chalmers.se/cs/research/formalmethods/minisat.

[9] Olivier Fourdrinoy, Eric Grégoire, Bertrand Mazure, and Lakhdar Sais. Eliminating redundant clauses in SAT instances. In *Proceedings of The Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07)*, pages 71–83, Brussels (Belgique), 2007.

[10] Alexander Hertel, Philipp Hertel, and Alasdair Urquhart. Formalizing dangerous SAT encodings. In *10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, pages 159–172, Lisbon (Portugal), 2007.

[11] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and artificial intelligence*, 1 :167–187, 1990.

[12] Chu-Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference of Principles and Practice of Constraint Programming (CP'97)*, pages 341–355, 1997.

[13] Paolo Liberatore. Redundancy in logic i : Cnf propositional formulae. *Artificial Intelligence*, 163(2) :203–232, 2005.

[14] Knot Pipatsrisawat and Adnan Darwiche. RSAT 2.0 : SAT solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[15] Sathiamoorthy Subbarayan and Dhiraj Pradhan. NiVER : Non increasing variable elimination resolution for preprocessing SAT instances. *7th International Conference on Theory and Applications of Satisfiability Testing Conference (SAT'04)*, pages 276–291, 2004.