Compresser des arbres de recherche UNSAT à l'aide d'un système de cache

Anthony Blomme, Daniel Le Berre, Anne Parrain, Olivier Roussel

▶ **To cite this version:**

Anthony Blomme, Daniel Le Berre, Anne Parrain, Olivier Roussel. Compresser des arbres de recherche UNSAT à l'aide d'un système de cache. Journées Francophones de Programmation par Contraintes (JFPC'23), Jul 2023, Strasbourg, France. pp.38-45. hal-04425245

HAL Id: hal-04425245

https://univ-artois.hal.science/hal-04425245v1

Submitted on 30 Jan 2024

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.

Compresser des arbres de recherche UNSAT à l'aide d'un système de cache

Anthony Blomme[1], Daniel Le Berre[1], Anne Parrain[1], Olivier Roussel[1]

[1] Univ. Artois, CNRS, Centre de Recherche en Informatique de Lens (CRIL), F-62300 Lens, France
{blomme, leberre, parrain, roussel}@cril.fr

5 mai 2023

Résumé

Afin de fournir aux utilisateurs de solveurs SAT des preuves d'incohérence petites et faciles à comprendre, nous présentons des techniques de mise en cache pour identifier des sous-preuves redondantes et réduire la taille des arbres de preuve UNSAT. Dans un arbre de recherche, nous élaguons les branches correspondant à des sous-formules qui ont été prouvées incohérentes auparavant dans l'arbre. Pour ce faire, nous utilisons un cache inspiré par les compteurs de modèles et nous l'adaptons au cas des formules incohérentes. Nous discutons de l'implémentation de ce cache dans un solveur CDCL et un solveur DPLL. Cette approche peut réduire drastiquement l'arbre de preuve UNSAT de plusieurs instances des compétitions SAT'02 et SAT'03.

Mots-clés

SAT, IA Explicable, Solveur.

Abstract

In order to provide users of SAT solvers with small, easily understandable proofs of unsatisfiability, we present caching techniques to identify redundant subproofs and reduce the size of some UNSAT proof trees. In a search tree, we prune branches corresponding to subformulas that were proved unsatisfiable earlier in the tree. To do so, we use a cache inspired by model counters and we adapt it to the case of unsatisfiable formulas. The implementation of this cache in a CDCL and a DPLL solver is discussed. This approach can drastically reduce the UNSAT proof tree of several benchmarks from the SAT'02 and SAT'03 competitions

Keywords

SAT, Explainable AI, Solver.

1 Introduction

Depuis deux décennies, les solveurs SAT ont été fréquemment utilisés pour résoudre des problèmes NP-complets, et sont donc devenus courants dans de nombreuses applications [1]. Cependant, avec la hausse de la complexité des programmes informatiques, il y a un besoin toujours plus important que ces programmes soient capables de fournir des explications à leurs résultats. Ce besoin concerne naturellement tout type de solveur et donc les solveurs SAT dans notre cas. De ce fait, il est maintenant attendu que les solveurs SAT fournissent des explications car ceux-ci ne peuvent plus être utilisés comme de simples boîtes noires. Quand une instance est cohérente, le modèle trouvé peut être donné comme explication, et compressé en le réduisant à un impliquant premier [5]. Quand l'instance est incohérente, donner une explication est plus difficile car il faut alors montrer qu'aucune solution ne peut être trouvée. Certaines formes d'explication ont été proposées pour prouver l'incohérence d'une formule. Par exemple, nous pouvons envisager de donner un *MUS* (pour *Minimal Unsatisfiable Subset*) [9] à l'utilisateur, ce qui donne l'origine de l'incohérence, ou encore un certificat d'incohérence exprimé dans un format particulier tel que DRAT [17]. Ce dernier enregistre les étapes importantes d'un solveur et peut ensuite être contrôlé par un vérificateur indépendant. Cependant, ces techniques peuvent présenter un intérêt limité pour l'utilisateur car, dans le premier cas, il n'y a aucune garantie qu'un MUS soit plus petit que la formule complète et, dans le second cas, un certificat peut avoir un nombre exponentiel d'étapes. Dans les deux cas, ces formes de preuves ne peuvent pas être facilement comprises par l'utilisateur.

Dans cet article, nous nous focalisons sur des formules incohérentes. Notre but est de compresser significativement l'arbre de recherche d'un solveur afin d'obtenir une preuve suffisamment petite pour pouvoir être donnée comme explication à l'utilisateur. Nous nous concentrons sur la reconnaissance de motifs réguliers en raison de leur impact potentiellement important sur la taille de l'arbre, et aussi parce qu'ils peuvent être expliqués individuellement et indépendamment à l'utilisateur. Un cache peut être utilisé pour reconnaître ces motifs. Si la sous-formule courante a déjà été explorée et prouvée incohérente, alors la branche peut être élaguée. Notre objectif est de réduire la taille de l'explication, et cela ne nous dérange pas de passer du temps sur cette tâche si cela nous permet d'obtenir une bonne compression. Par conséquent, nous n'excluons pas des techniques coûteuses comme les oracles NP, tant qu'elles nous permettent de réduire la taille de la preuve.

Cet article est organisé de la façon suivante. Dans la section 2, nous introduisons certaines notions fondamentales. Dans la section 3, après un exemple dédié au problème de pigeons (PHP), nous discutons de l'intégration d'un cache

pour formules incohérentes dans les solveurs SAT, en considérant deux architectures de solveur. Ensuite, nous présentons quelques résultats expérimentaux dans la section 4. Enfin, nous concluons et nous présentons quelques pistes de recherche.

2 Préliminaires

Une *variable* booléenne v peut être soit vraie soit fausse. Un *littéral* est soit une variable v soit sa négation $\neg v$. Une *clause* est une disjonction (ou un ensemble) de littéraux et une formule sous *Forme Normale Conjonctive* (CNF) est une conjonction (ou un ensemble) de clauses. Une *affectation* est une fonction d'un ensemble de variables vers les valeurs de vérité 0 (pour *faux*) ou 1 (pour *vrai*). Une clause est satisfaite par une affectation si elle contient au moins un littéral l assigné à vrai. Une formule est satisfaite par une affectation si et seulement si toutes ses clauses sont satisfaites. Décider s'il existe une affectation qui satisfait une formule CNF donnée est connu comme le *problème de la cohérence booléenne* (SAT), qui est NP-complet [2]. La formule est *SAT* s'il est possible de trouver une telle affectation et elle est *UNSAT* sinon. Soit une affectation I, $F_{|I}$ désigne la formule simplifiée par I : les clauses satisfaites sont supprimées de la formule et les littéraux falsifiés sont supprimés des clauses restantes. Une *clause unitaire* est une clause c qui contient uniquement un littéral non falsifié l, qui doit donc être assigné à vrai. La clause c peut ensuite être considérée comme la *raison* de l'affectation de l et va être désignée $reason(l)$. Appliquer cette opération jusqu'à ce qu'il ne reste plus de clauses unitaires est appelé la *propagation unitaire*. Étendre une affectation avec un littéral sans raison est appelé une *décision*. Il est possible d'associer un *niveau de décision* à chaque littéral propagé ou décidé. Celui-ci est défini comme le nombre de décisions prises par le solveur avant l'affectation du littéral considéré. Une fonction $DL(l)$ fournit le niveau auquel le littéral l a été décidé ou propagé.

Les solveurs SAT sont des programmes informatiques capables de résoudre le problème SAT. Les premiers solveurs SAT capables de résoudre ce problème reposaient sur l'architecture *Davis Putnam Logemann Loveland* (DPLL) [4, 3]. Il y a deux décennies, une nouvelle architecture appelée *Conflict Driven Clause Learning* (CDCL) est apparue adaptée à la résolution de problèmes structurés et a fait des solveurs SAT des oracles couramment utilisés pour résoudre des problèmes NP-complets [1]. Les solveurs SAT explorent un arbre de recherche, dans lequel un chemin de la racine aux feuilles est une affectation partielle, et les feuilles correspondent à des clauses falsifiées (que l'on nomme conflit) quand la formule est incohérente. Alors que les approches DPLL explorent un arbre binaire en branchant sur les valeurs de vérité des variables, les solveurs CDCL utilisent l'analyse de conflit et l'apprentissage de clause pour diriger la recherche [11].

3 Redondance au cours de la recherche

Détecter des sous-arbres communs dans un arbre de recherche n'est pas nouveau : par exemple dans les compteurs de modèles [16, 14] les sous-arbres communs sont utilisés pour mettre en cache des nombres de modèles déjà calculés. Dans notre cas, nous voulons implémenter une idée similaire, mais en ciblant les formules incohérentes. Dans ce contexte, le cache contiendra des formules prouvées UNSAT, que nous espérons retrouver pendant la recherche. Un élément du cache est appelé une *entrée*. Le temps nécessaire à la réalisation de la compression n'est pas important à ce stade. Nous étudions en priorité les capacités de compression.

3.1 Exemple introductif

Les problèmes de pigeons sont des problèmes incohérents classiques connus pour être difficiles pour les solveurs et présentant de nombreuses symétries [8]. Le problème consiste à assigner $n+1$ pigeons à n pigeonniers avec les contraintes qu'un pigeon doit être associé à un pigeonnier et qu'un pigeonnier ne peut pas accueillir plus d'un pigeon. Pour ce problème, nous définissons les variables $x_{i,k}$, avec $i \in \{1, \dots n+1\}$ et $k \in \{1, \dots n\}$, qui indiquent que le pigeon i est associé au pigeonnier k. La première contrainte peut donc être encodée en utilisant une clause de taille n pour chaque pigeon : $C_{1,n} = \bigwedge_{1 \leq i \leq n+1}(x_{i,1} \vee \dots \vee x_{i,n})$. Pour la seconde, nous devons créer toutes les exclusions mutuelles entre deux pigeons et pour un pigeonnier spécifique : $C_{2,n} = \bigwedge_{1 \leq i < j \leq n+1} \bigwedge_{1 \leq k \leq n}(\neg x_{i,k} \vee \neg x_{j,k})$. Avec ces considérations, un problème de pigeons pour une valeur n (PHP_n) est défini comme $PHP_n = C_{1,n} \wedge C_{2,n}$. Quand la variable $x_{1,k}$ est assignée à $vrai$ et propagée, nous nous retrouvons avec un problème de pigeons de taille $n-1$. Cela se produit quand on explore les n façons de placer le premier pigeon. Une fois que le premier sous-problème PHP_{n-1} a été exploré, il peut être ajouté au cache et les $n-1$ autres sous-problèmes peuvent être reconnus. Cette méthode peut être répétée récursivement jusqu'à rencontrer le problème PHP_2. Seulement deux branches, une décision et sa négation, sont nécessaires pour complètement explorer ce dernier. La figure 1 illustre l'imbrication des sous-problèmes de pigeons.

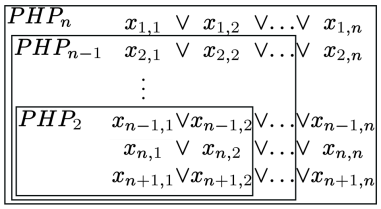

FIGURE 1 – Problème de pigeons de taille n.

Considérons par exemple le problème PHP_4 et une heuristique qui décide négativement la première variable non assignée. Cette heuristique va d'abord assigner $\neg x_{1,1}$, $\neg x_{1,2}$ et $\neg x_{1,3}$. Après ces trois décisions, la clause $x_{1,1} \vee x_{1,2} \vee$

$x_{1,3} \vee x_{1,4}$ va propager $x_{1,4}$. Cette dernière affectation va également propager les littéraux $\neg x_{2,4}$, $\neg x_{3,4}$, $\neg x_{4,4}$ et $\neg x_{5,4}$. Nous devons maintenant explorer le problème PHP_3. L'heuristique va ensuite décider $\neg x_{2,1}$ puis $\neg x_{2,2}$ et la clause $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$ va propager $x_{2,3}$. Cela va aussi propager $\neg x_{3,3}$, $\neg x_{4,3}$ et $\neg x_{5,3}$. Nous arrivons alors au problème PHP_2. Ce dernier sera entièrement exploré en décidant $\neg x_{3,1}$ puis en inversant cette décision. Les deux branches vont mener à un conflit. Comme nous avons prouvé que le problème PHP_2 est UNSAT, nous pouvons l'enregistrer dans le cache. En inversant les décisions $\neg x_{2,2}$ puis $\neg x_{2,1}$, nous allons obtenir à chaque fois un problème PHP_2 basé sur différentes variables. En consultant le cache, nous savons que nous avons déjà exploré ce problème à un renommage de variables près et nous pouvons directement conclure que ces deux branches sont incohérentes. Nous pouvons maintenant stocker le problème PHP_3 et un comportement similaire va se produire en inversant les décisions $\neg x_{1,3}$, $\neg x_{1,2}$ et $\neg x_{1,1}$. Après cela, la recherche va s'arrêter et l'instance va être considérée incohérente. La figure 2 montre l'arbre de recherche obtenu avec cette méthode. On remarque qu'il contient une branche unique avec toutes les décisions. Au final, nous avons 5 détections d'isomorphisme pour un total de 7 branches.

Avec cette méthode, il est possible d'avoir un total de $\sum_{y=2}^{n-1} y = ((n-2)(n+1))/2$ détections d'isomorphisme pour PHP_n. En ajoutant les deux branches de PHP_2, nous avons un total de $(((n-2)(n+1))/2) + 2$ branches.

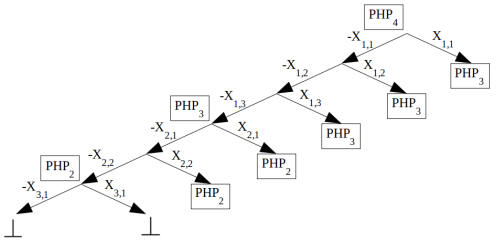

FIGURE 2 – Branche unique attendue en résolvant le problème PHP_4. Les propagations ont été omises.

3.2 Cache pour UNSAT

Pour généraliser le résultat du problème de pigeons, nous avons besoin de trouver une façon de détecter qu'une sous-formule donnée a déjà été trouvée dans l'arbre de recherche. Les compteurs de modèles [7] utilisent ce genre de fonctionnalité pour éviter de calculer plusieurs fois le nombre de modèles d'une même sous-formule (cela inclut le cas UNSAT, pour lequel le nombre de modèles est 0). Pour ce faire, ils utilisent une représentation normalisée de la sous-formule. Celle implémentée dans le compteur de modèles Cachet [14] assure que deux sous-formules avec les mêmes clauses sont considérées identiques même si ces clauses ne sont pas dans le même ordre ou si elles n'ont pas les mêmes indices. Cependant, cette technique ne va pas fonctionner sur notre exemple du problème de pigeons. En effet, dans ce cas, les sous-formules ne sont pas identiques, elles sont basées sur différentes variables. Il nous faut alors supporter la notion d'égalité modulo un renommage. Il y a aussi un problème spécifique lié à la mise en cache de formules UNSAT : une formule est UNSAT si elle contient une sous-formule UNSAT. Nous ne cherchons donc plus seulement des formules identiques, mais aussi des formules sous-sommées par une entrée du cache. Dans ce contexte, le cache ne peut plus être implémenté avec un dictionnaire. Nous devons vérifier séquentiellement toutes les entrées pour lesquelles la formule considérée a au moins autant de clauses de chaque taille. Ces deux fonctionnalités (inclusion et renommage) peuvent être implémentées en résolvant un problème NP-complet de sous-isomorphisme de graphe quand nous interrogeons le cache. Pour ce faire, nous encodons les formules sous forme de graphes de manière classique : les littéraux correspondent à des nœuds de la même couleur et les clauses correspondent à des nœuds dont la couleur dépend de leur taille. Une arête connecte les littéraux opposés et une clause est reliée à chacun de ses littéraux. La figure 3 montre un exemple de cette représentation. Chaque couleur est ici représentée par une forme différente.

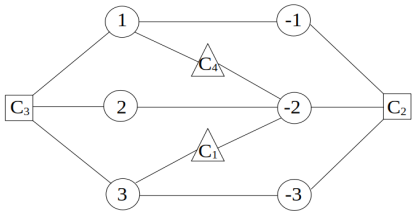

FIGURE 3 – Graphe correspondant à $F = (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2 \vee \neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2)$. Les littéraux sont représentés par des cercles, les clauses binaires/ternaires par des triangles/carrés.

3.3 Sources d'incohérence

Un nœud de l'arbre de recherche est identifié par l'interprétation I des variables menant à ce nœud. Quand la sous-formule $F_{|I}$ obtenue à un nœud est incohérente, notre but est de l'enregistrer dans le cache. Cependant, nous ne voulons pas stocker la sous-formule complète dans le cache, mais seulement un sous-ensemble incohérent de $F_{|I}$. En d'autres termes, nous voulons enregistrer un sous-ensemble incohérent de $F_{|I}$, mais pas nécessairement un sous-ensemble minimal (MUS) car cela serait trop coûteux et vraisemblablement peu rentable. Les solveurs SAT modernes sont capables de fournir une source d'incohérence d'une formule F quand elle est UNSAT (ce que l'on nomme un noyau UNSAT [18]). Toutefois, un tel noyau est généralement donné à la fin de la recherche et il correspond à la formule complète. En revanche, nous devons générer localement un noyau incohérent pour tout nœud de l'arbre de telle sorte que $F_{|I}$ est incohérent. Pour obtenir ce sous-ensemble incohérent, il nous faut collecter les clauses identifiées comme conflits ou utilisées dans les propagations menant à ces conflits. Dans la suite de l'article, nous appellerons *sources* d'une sous-formule incohérente

$F_{|I}$ les clauses *initiales* de F utilisées par le solveur pour prouver l'incohérence de $F_{|I}$. Cet ensemble sera désigné par $S(F, I)$. Ces sources peuvent facilement être obtenues en récoltant récursivement la raison de chaque propagation menant aux conflits. Ce processus est en essence similaire à l'analyse de conflit des solveurs CDCL, sauf que l'on ne réalise aucune étape de résolution. Un point important est que les sources ne peuvent contenir que des clauses de la formule initiale. Dans un solveur CDCL, si une clause apprise apparaît dans les sources, elle est remplacée par l'ensemble des clauses initiales qui l'ont générée. Les sources peuvent être définies formellement comme suit.

Définition 1 *Nous définissons d'abord la source d'une clause $S(C)$. Quand C est une clause initiale, $S(C) = \{C\}$. Quand L est une clause apprise, $S(L)$ est l'ensemble des clauses initiales de F qui apparaissent dans la dérivation de L par résolution.*
Soit $F_{|I}$ une sous-formule incohérente et $\{I_1, \dots I_m\}$ l'ensemble des branches développées par le solveur pour prouver cette incohérence. Chaque $F_{|I_j}$ contient un conflit C_j. Nous définissons $S_0(F, I_j) = \{C_j\}$ et $S_{i+1}(F, I_j) = S_i(F, I_j) \cup \{S(reason(l)) | l \in c \wedge c \in S_i(F, I_j) \wedge DL(l) \geq DL(I_j)\}$. Cette séquence a un point fixe désigné $S(F, I_j)$. Enfin, les sources $S(F, I)$ de $F_{|I}$ sont définies comme $S(F, I) = \cup_j S(F, I_j)$.

Par construction, $S(F, I)_{|I}$ est incohérente car elle contient toutes les clauses initiales utilisées par le solveur pour prouver l'incohérence de $F_{|I}$. Nous avons aussi $S(F, I)_{|I} \subseteq F_{|I}$. Par conséquent $S(F, I)_{|I}$ est un noyau incohérent de $F_{|I}$. Dans un solveur DPLL, $S(F, I)$ peut être obtenu en collectant les sources des deux nœuds fils $S(F, I \cup \{l\})$ et $S(F, I \cup \{\neg l\})$ et en ajoutant les clauses propagées par l, un littéral apparaissant dans les sources des nœuds fils. Ce point sera discuté dans la section 3.5. Dans un solveur CDCL, les sources sont obtenues en collectant toutes les clauses utilisées dans l'analyse de conflit, et en remplaçant chaque clause apprise par ses sources (i.e. les clauses collectées au niveau du conflit qui a généré cette clause apprise).

3.4 Le cas CDCL

L'architecture CDCL est actuellement l'approche état-de-l'art pour la résolution pratique de SAT [11]. Il est donc logique de mettre en œuvre le cache sur cette architecture. Cependant, cela pose quelques soucis. Un solveur CDCL explore l'arbre de recherche de manière non chronologique, puisque chaque fois qu'il apprend une nouvelle clause, le solveur revient au niveau de décision ayant propagé un littéral grâce à cette clause (ce n'est pas toujours le cas des solveurs CDCL implémentant un retour en arrière chronologique [13]). L'exemple de la figure 4 illustre ce comportement. Considérons la formule propositionnelle $F' = \{x \vee y \vee z, \neg x \vee y\} \cup F$. Nous ne savons pas si F est cohérente ou non. Supposons que le solveur prenne d'abord des décisions sur des variables différentes de x, y, z, puis la décision $\neg z$ qui supprime le littéral z dans la première clause. Après cela, d'autres décisions sont prises et le solveur décide ensuite $\neg y$, ce qui supprime le littéral y dans les deux premières clauses. À ce moment-là, comme nous devons satisfaire à la fois x et $\neg x$, un conflit va être dérivé par propagation unitaire. Après l'analyse de conflit, le solveur apprend la clause $y \vee z$, qui est la résolvante des deux premières clauses, retourne à la décision $\neg z$ et propage directement le littéral y, qui satisfait les deux premières clauses. Notez que la recherche non chronologique ignorera les nœuds entre les décisions $\neg y$ et $\neg z$. Ainsi, la cohérence des sous-formules correspondantes demeure inconnue. En d'autres termes, l'étape de retour en arrière dans un solveur CDCL n'indique pas que les sous-formules associées à des nœuds inexplorés sont incohérentes. C'est ce que notre exemple illustre : la décision y aurait pu être prise à n'importe quel niveau entre les décisions $\neg z$ et $\neg y$. Dans ce cas, les deux premières clauses auraient été satisfaites et la cohérence de la sous-formule n'aurait plus dépendu que de la cohérence de F. Cela a une conséquence sur la façon dont nous remplissons notre cache : nous ne pouvons ajouter une entrée au cache que quand nous somme sûrs que la formule simplifiée courante est UNSAT et que cela a été prouvé par le solveur. Dans un solveur CDCL, cela ne se produit qu'aux feuilles de l'arbre, quand nous rencontrons une clause conflictuelle. Les sources d'incohérence sont calculées en parcourant le graphe d'implication à partir de la clause conflit. Comparé à l'analyse de conflit classique, ce calcul produit un sous-ensemble des clauses initiales. Les clauses apprises sont remplacées par les clauses initiales nécessaires à les produire. D'une certaine manière, les sources que nous calculons « déplient » les clauses apprises en clauses initiales. En pratique, au fur et à mesure que la recherche progresse, la taille des sources trouvées augmente. Les entrées du cache sont créées en calculant les sources des feuilles que l'on simplifie en utilisant les décisions courantes et la propagation unitaire. Pendant cette simplification, nous excluons les littéraux propagés par une clause apprise, sinon nous obtiendrions toujours une clause vide. Nous avons réalisé quelques expérimentations initiales avec ce cadre. Pour les instances de problème de pigeons, nous avons pu retrouver un petit arbre similaire à celui montré à la figure 2 en adaptant l'heuristique pour qu'elle branche d'abord positivement sur les variables apparaissant le plus fréquemment dans la formule (plutôt que négativement comme l'heuristique de base de Minisat).

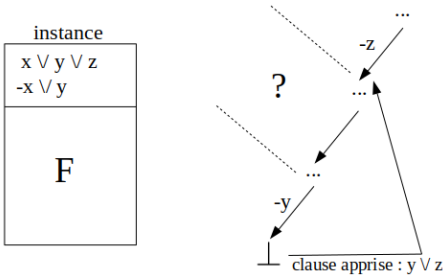

FIGURE 4 – Exemple illustrant le problème induit par la recherche non chronologique des solveurs CDCL.

Par contre, cette heuristique n'est pas performante sur les instances considérées. De plus, l'utilisation de notre cache était limitée à une étape de post-traitement, car CDCL a besoin d'une raison pour revenir en arrière. Changer la façon dont la raison est calculée modifie l'exploration de l'espace de recherche et peut donc agrandir l'arbre de recherche final. Par conséquent, nous avons aussi considéré l'approche DPLL qui ne souffre pas de ce problème.

3.5 Le cas DPLL

Dans un solveur de type DPLL, lorsque les deux fils d'un nœud correspondant à une interprétation I ont été explorés (une branche pour la décision l, une autre pour la décision $\neg l$) et que les deux sont incohérents, nous savons que $F_{|I}$ est aussi incohérent et les sources $S(F, I)$ peuvent être obtenues comme présenté dans la section 3.3. $S(F, I)$ simplifié par I est incohérent et ajouté au cache. Quand un nouveau nœud identifié par l'interprétation I est exploré, la première étape est de vérifier dans le cache s'il y a une entrée E qui est inclue dans la formule $F_{|I}$ courante à un renommage de littéraux près. S'il existe E dans le cache et σ un renommage des littéraux tel que $\sigma(E) \subseteq F_{|I}$, alors $F_{|I}$ est nécessairement incohérente puisque E est incohérente. Ce test peut être traduit en problème de sous-isomorphisme de graphe (voir section 3.2). Si on en a trouvé un, l'ensemble F' des clauses de $F_{|I}$ qui correspondent à des clauses de E est facilement obtenu en faisant correspondre des nœuds à des clauses. $F'_{|I}$ est incohérente mais en général F' peut être cohérente. En effet, F' doit être complété avec les clauses nécessaires pour propager les littéraux effacés dans F' au niveau de décision courant pour obtenir une formule incohérente. Prenons par exemple un problème de pigeons P encodé par les clauses $\{C_1, C_2, \ldots, C_n\}$ et considérons la formule F définie par $\{\neg x \vee y, \neg x \vee \neg y, x \vee C_1, C_2, \ldots, C_n\}$. Supposons aussi que le problème de pigeons P soit déjà présent dans le cache. À partir de F, si on branche sur y, $\neg x$ est propagé, et la formule simplifiée contient maintenant P qui est reconnu comme une entrée du cache. Les clauses de F correspondant à P sont $F' = \{x \vee C_1, C_2, \ldots, C_n\}$. Si on branche sur $\neg y$, nous obtenons aussi $F' = \{x \vee C_1, C_2, \ldots, C_n\}$ de la même manière. Cependant, F' est cohérente car la première clause de P peut être neutralisée par x. Pour retrouver une formule incohérente, nous devons ajouter toutes les clauses utilisées pour propager $\neg x$ sur les deux branches, ce qui signifie que nous devons ajouter $\{\neg x \vee y, \neg x \vee \neg y\}$ à F' pour obtenir une formule incohérente, qui est la source $S(F, \emptyset)$ et donc F peut maintenant être ajoutée comme entrée du cache. Il est à souligner que les vérifications du cache ont un coût élevé : nous résolvons plusieurs fois un problème NP-complet. Cependant, comme notre objectif n'est pas d'accélérer le temps de résolution mais plutôt de réduire la taille de l'arbre de recherche, il est acceptable de passer un long moment à chercher dans le cache si, au final, l'arbre généré est suffisamment petit. Quand une nouvelle entrée est ajoutée dans le cache, nous pouvons utiliser le plus grand niveau de décision présent dans les sources pour réaliser le retour en arrière. L'idée ici est d'éviter de revenir sur des décisions non impliquées dans le conflit. Ces nœuds nous donneraient la même entrée à ajouter au cache que l'entrée courante. Nous pouvons alors revenir au niveau de décision trouvé de cette manière. Si nous revenons à une décision qui n'a pas encore été inversée, alors nous inversons cette décision. Sinon nous ajoutons une nouvelle entrée au cache et nous répétons cette procédure. À titre d'exemple, considérons une formule F qui contient les clauses $\{a \vee b \vee c, a \vee b \vee \neg c, a \vee \neg b \vee c, a \vee \neg b \vee \neg c\}$ et l'interprétation $I = \langle \neg a, x, y, z \rangle$. Alors les décisions b et $\neg b$ vont toutes les deux mener à un conflit, ce qui signifie que $F_{|I}$ est incohérente de même que $S(F, I)_{|I} = \{b \vee c, b \vee \neg c, \neg b \vee c, \neg b \vee \neg c\}$. Tant que a est falsifié, ces clauses incohérentes restent dans la formule et nous pouvons donc revenir au niveau de la décision $\neg a$. Notez que même si cette technique de retour en arrière est similaire à l'analyse de conflit des solveurs CDCL, il y a tout de même quelques différences. Tout d'abord nous ne réalisons aucune étape de résolution, et donc aucune coupe dans le graphe d'implication (UIP). Ensuite nous ne sommes autorisés à passer que les sous-arbres dont nous savons qu'ils sont incohérents. Une autre différence est le fait que les clauses utilisées durant une analyse de conflit sont une explication de la clause apprise alors que les sources sont une explication de l'incohérence de la sous-formule.

4 Résultats expérimentaux

Nous avons implémenté l'approche proposée à l'aide de Minisat [6]. Nous avons désactivé la simplification de la base de clauses pour garder les clauses initiales pendant toute la recherche. Nous avons aussi désactivé les redémarrages qui construisent une séquence d'arbres de recherche. Pour l'approche DPLL, nous avons également désactivé l'apprentissage de clause et l'analyse de conflit. Cette dernière est remplacée par une procédure dédiée à la collecte des sources. L'activité d'une variable, qui est mise à jour pour chaque nouvelle clause apprise dans un solveur CDCL classique, est ici mise à jour à chaque fois qu'une nouvelle clause est ajoutée aux sources dans notre contexte. Pour l'approche CDCL, nous avons à la fois la procédure d'analyse de conflit et la procédure de collecte de sources (l'heuristique et l'apprentissage de clauses sont inchangées par rapport à Minisat). Le Glasgow Subgraph Solver (abrégé en GSS) [12] est appelé pour calculer des sous-isomorphismes de graphes, et donc questionner notre cache. Nous avons utilisé des instances des compétitions SAT'02 (partie *submitted*) [15] et SAT'03 (parties *handmade* et *industrial*) [10]. Nous avons choisi ces instances parce que nous avions besoin d'instances « faciles » pour Minisat, comme notre approche a une complexité de calcul élevée. Un résumé des résultats obtenus peut être trouvé dans la table 1. Minisat est évidemment bien plus efficace que nos approches avec du post-traitement ou un cache intégré (décrites plus tard) à cause du coût élevé de notre cache. Le DPLL avec post-traitement est clairement moins efficace que le CDCL avec post-traitement sur ces instances. Par contre, intégrer directement le cache à l'intérieur d'un solveur DPLL four-

TABLE 1 – Résumé de nos expérimentations. Pour chaque compétition, nous donnons le nombre d'instances connues pour être UNSAT et le nombre d'instances résolues par Minisat en 1 minute (instances faciles). Ensuite, nous donnons le nombre d'instances résolues dans chaque expérimentation. Les nombres entre parenthèses indiquent le nombre d'arbres de recherche avec une branche unique trouvés.

Compétition	#UNSAT	Minisat (1min)	DPLL (15min)		CDCL (15min)
			Post-traitement	Cache intégré	Post-traitement
SAT'02	381	276	40 (4)	106 (42)	78 (11)
SAT'03	198	78	15 (13)	87 (53)	39 (28)

nit des résultats significativement meilleurs que le CDCL avec post-traitement. Cela permet même de résoudre des instances que Minisat ne peut pas résoudre en 4 heures (e.g. les instances de la famille Urquhart).

4.1 Post-traitement des traces

Dans cette section, nous nous intéressons au potentiel de compression de notre approche. Il est donc nécessaire de comparer l'arbre avec et sans cache. La seule façon de faire est dans un premier temps de résoudre le problème et stocker l'arbre de recherche et dans un second temps de lancer le mécanisme de cache sur l'arbre stocké. Ainsi, il est facile de comparer l'arbre initial et l'arbre obtenu en utilisant le cache. En pratique, stocker l'arbre de recherche peut mener à des fichiers de très grande taille, c'est pourquoi nous simulons le post-traitement directement dans le solveur. Nous imposons un temps limite de 2 secondes pour chaque appel au GSS lorsque nous essayons d'identifier un sous-isomorphisme de graphe. Les deux approches DPLL et CDCL ont été testées avec un temps limite de 15 minutes sur nos instances. Certains résultats individuels de ces expérimentations sur quelques familles d'instances peuvent être trouvés dans la table 2. Nous comparons les nombres de conflits, donc de branches, des deux arbres de recherche. Le rapport entre ces deux nombres représente le pouvoir de compression de notre approche. La taille d'une instance est la somme des longueurs de ses clauses. Une distribution des rapports obtenus par les deux approches peut être trouvée dans la table 3. Le rapport de compression peut être très bon (plus petit que 10^{-3}), notamment pour l'approche CDCL. Malheureusement, cela n'arrive que pour un petit sous-ensemble des instances, principalement les familles marg, Urquhart et xor_chain, qui sont très structurées. Pour l'approche DPLL, le post-traitement s'est comporté principalement comme attendu et décrit dans la section 3.1 pour les problèmes de pigeons. La seule différence vient de l'heuristique utilisée dans Minisat, qui décide négativement la première variable et décide ensuite négativement les variables en commençant par la dernière et dans l'ordre décroissant. Donc, après que le problème PHP_{n-1} a été ajouté dans le cache, il est reconnu $n-1$ fois avec la première variable assignée (ce problème a été ajouté juste avant PHP_{n-1}). Ce problème est aussi reconnu quand on inverse la première décision. Ce comportement crée une branche additionnelle et donc le nombre de branches trouvé diffère de un comparé au nombre de branches attendu. Concernant les instances des compétitions SAT, pour les familles marg, Urquhart et xor_chain, nous avons souvent obtenu un arbre de recherche avec une branche unique comme montré dans la figure 2. Pour ces instances, quand le solveur ajoute une nouvelle entrée au cache après une certaine décision, elle est souvent reconnue après la négation de cette décision. Cela nous permet d'élaguer de nombreuses branches et cela explique les bons rapports obtenus. Ce n'est pas strictement le cas pour certaines instances (e.g. marg2x6.cnf et x1_16.cnf) mais nous avons obtenu des arbres très courts pour elles. Par exemple, la figure 5 montre l'arbre de recherche obtenu par notre approche quand le cache est utilisé dans les deux approches DPLL et CDCL. Les boîtes en violet représentent l'ajout d'une nouvelle entrée dans le cache et les boîtes en vert correspondent à la détection d'une entrée. Le label « i x » (« isomorphisme x«) signifie que l'élément enregistré à « cache x » a été reconnu.

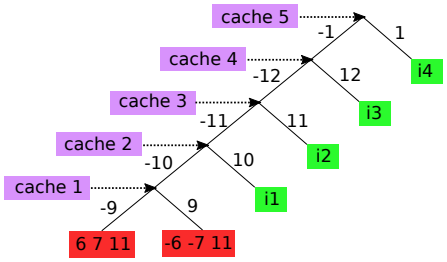

FIGURE 5 – Arbre de recherche pour marg2x2.cnf avec le cache.

4.2 Solveur avec cache intégré

Comme seconde expérimentation, nous avons utilisé le cache pendant la recherche elle-même. Nous n'avons pu réaliser qu'une implémentation pour DPLL, puisque générer une clause conflit depuis une correspondance du cache est une question ouverte à ce stade. Nous avons considéré les mêmes instances que précédemment et toujours avec un temps limite de 15 minutes par instance. Un extrait pertinent des résultats est donné dans la table 4. Pour un total de 95 instances, le solveur développe un arbre avec une branche unique comme présenté à la figure 2. La bonne compression précédemment obtenue pour les familles marg, Urquhart et xor_chain est toujours obtenue, également sur des instances plus grandes. Nous avons observé que le solveur DPLL avec cache intégré peut résoudre en moins de 15 minutes des instances que Minisat ne parvient pas à résoudre en plus de 4 heures. C'est le cas de certaines instances Urquhart de SAT'02 par exemple. Arrêter la recherche dès qu'une entrée du cache est détectée

TABLE 2 – Résultats expérimentaux concernant le pouvoir de compression de notre approche. Pour chaque instance, nous fournissons sa taille en nombre de littéraux, et le nombre de conflits trouvés sans et avec cache pour les deux approches DPLL et CDCL. Un tiret indique un dépassement de temps limite. Le rapport de compression est le nombre de conflits avec cache divisé par le nombre de conflits sans cache.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport de compression	Conflits (sans cache)	Conflits (cache)	Rapport de compression
PHP_7	448	$6.8\ 10^3$	23	$3.4\ 10^{-3}$	$5.6\ 10^3$	853	$1.5\ 10^{-1}$
PHP_{12}	2,028	-	-	-	-	-	-
marg2x6.sat03-1444	528	$5.2\ 10^5$	21	$4.0\ 10^{-5}$	$3.0\ 10^4$	20	$6.6\ 10^{-4}$
marg3x3add8.sat03-1449	1,056	-	-	-	$1.8\ 10^5$	32	$1.8\ 10^{-4}$
Urquhart-s3-b9	1,240	$5.1\ 10^5$	20	$3.9\ 10^{-5}$	$1.9\ 10^4$	21	$1.1\ 10^{-3}$
Urquhart-s3-b3	2,152	-	-	-	$1.6\ 10^6$	29	$1.8\ 10^{-5}$
x1_16	364	$6.0\ 10^4$	18	$3.0\ 10^{-4}$	$2.2\ 10^3$	20	$9.1\ 10^{-3}$
x1_24	556	-	-	-	$2.0\ 10^5$	78	$3.9\ 10^{-4}$
3col20_5_6	646	33	20	$6.0\ 10^{-1}$	27	27	1
3col40_5_5	1,286	756	198	$2.6\ 10^{-1}$	118	72	$6.1\ 10^{-1}$
homer06	1,800	$5.1\ 10^5$	195	$3.8\ 10^{-4}$	-	-	-
homer17	3,718	-	-	-	-	-	-

TABLE 3 – Distribution des rapports pour les techniques de post-traitement pour les approches DPLL et CDCL.

Rapport	Non résolu	[1 ;0.75[	[0.75 ;0.5[	[0.5 ;0.25[	[0.25 ;10^{-1}[	[10^{-1} ;10^{-2}[	[10^{-2} ;10^{-3}[	$\leq 10^{-3}$
DPLL	524	4	5	11	11	5	6	13
CDCL	462	38	13	5	3	6	7	45

permet de résoudre bien plus d'instances que le DPLL initial, notamment pour les familles testées dans la première expérimentation. Cependant, comme la taille du cache ne fait qu'augmenter au fur et à mesure de la recherche, essayer de reconnaître une entrée du cache peut devenir très coûteux, même avec le temps limite de 2 secondes. De plus, comme les sous-formules peuvent aussi être très grandes, trouver un isomorphisme peut prendre plus de temps que la limite imposée. Pour des grandes instances, certains appels au GSS ont peut être été annulés et il se peut que nous soyons passés à côté de certains isomorphismes, et donc compressions. Cela se produit par exemple pour les problèmes de pigeons plus grands que PHP_{16}.

5 Conclusion

Notre objectif dans ce travail est d'élaguer le plus possible les branches d'un arbre de recherche UNSAT pour réduire sa taille. Pour ce faire, nous avons proposé un cache inspiré par ce qui existe déjà pour les compteurs de modèles. L'idée est d'enregistrer des sous-formules UNSAT et d'essayer de les reconnaître plus tard dans l'arbre de recherche pour éviter d'explorer plusieurs sous-parties identiques de l'arbre. Nous avons présenté une méthode syntaxique basée sur la détection de sous-isomorphismes de graphe. Nous avons vu qu'il est possible d'obtenir des rapports plutôt bons et des preuves courtes et même un arbre de recherche avec une branche unique pour certaines familles d'instances, notamment celles avec beaucoup de symétries ou similarités mais il n'est toujours pas clair si cette approche peut fonctionner sur un large éventail d'instances. Nous avons proposé une implémentation de ce cache sur les deux architectures DPLL et CDCL. Si cette dernière présente des résultats prometteurs, elle ne passe pour l'instant pas à l'échelle car nous n'avons pu l'implémenter que comme un post-traitement. L'intégration du cache directement dans le solveur DPLL a permis de réduire drastiquement beaucoup plus d'arbres de recherche, en incluant des problèmes que Minisat n'est pas capable de résoudre. Malheureusement, générer une clause conflit depuis une correspondance du cache est une question ouverte à ce stade. Nous pouvons envisager quelques pistes pour améliorer notre approche. Tout d'abord, nous avons considéré une approche basée sur la gestion d'un cache mais nous n'avons pas implémenté la possibilité de supprimer les entrées qui semblent inutiles. Cette fonctionnalité est disponible dans les compteurs de modèles pour éviter que la mémoire ne dépasse une certaine limite. Cela pourrait être un bon ajout à notre approche. Nous n'avons considéré que deux heuristiques (celle de Minisat et une variante). D'autres heuristiques pourraient être essayées, comme celles utilisées dans les compteurs de modèles. Concernant la détection de sous-isomorphismes de graphe, il pourrait être intéressant de collecter certaines informations pendant un appel au GSS et essayer de les réutiliser dans de futurs appels. De plus, nous nous intéressons à la détection d'entrées du cache dans lesquelles certains littéraux sont falsifiés, comme celles-ci sont aussi incohérentes. En effet, l'une des raisons pour lesquelles le CDCL ne produit pas souvent un arbre avec une branche unique est que la sous-formule courante n'est pas directement une entrée du cache mais une avec des littéraux falsifiés. Finalement, nous recherchons d'autres formes de redondance pour compresser des arbres de recherche UNSAT dans un cadre plus général.

TABLE 4 – Résultats expérimentaux quand le cache est utilisé pendant la recherche. Pour chaque instance, nous donnons les nombres de conflits, d'entrées dans le cache, d'appels au GSS qui ont trouvé un isomorphisme ainsi que le nombre total d'appels et le nombre d'appels abandonnés (i.e. qui ont dépassé le temps limite de deux secondes). Le nombre entre parenthèses indique le nombre d'entrées différentes du cache reconnues par isomorphisme. Nous fournissons aussi le temps passé par le solveur (sans détection d'isomorphisme) et le temps cumulé de tous les appels au GSS. Tous les temps sont en secondes.

	DPLL (cache intégré)						
Instance	Conflits	Taille du cache	Sous-isomorphismes de graphe	Appels	Abandons	Temps (recherche)	Temps (GSS)
PHP_7	23	22	21 (6)	21	0	0.007	0.180
PHP_{12}	68	67	66 (11)	66	0	0.071	5.728
PHP_{16}	122	121	120 (15)	120	0	0.274	63.741
marg2x6.sat03-1444	21	20	17 (17)	18	0	0.004	0.162
marg3x3add8.sat03-1449	26	25	22 (22)	24	0	0.024	0.813
marg6x6.sat03-1456	86	85	84 (84)	84	0	0.134	7.446
Urquhart-s3-b9	20	19	18 (18)	18	0	0.009	0.175
Urquhart-s3-b3	29	28	27 (27)	27	0	0.024	0.486
Urquhart-s5-b5	94	93	92 (91)	101	0	0.292	36.967
x1_16	18	17	14 (14)	42	0	0.005	0.419
x1_24	25	24	23 (23)	23	0	0.037	0.779
x2_80.sat03-1605	395	394	393 (318)	2,257	121	0.919	427.492
3col20_5_6	12	11	6 (3)	31	0	0.004	0.178
3col40_5_5	357	319	235 (41)	52,583	0	1.451	564.310
homer06	111	105	98 (27)	420	40	0.495	116.096
homer17	363	348	352 (92)	1,691	211	3.249	712.465

Remerciements

Le premier auteur est en partie financé par la région Hauts-de-France. Ce travail a été soutenu par le projet CPER Data de la région Hauts-de-France.

Références

[1] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*. 2021.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM*, pages 151–158, 1971.

[3] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[5] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *FMCAD 2013*, pages 46–52, 2013.

[6] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT 2003*, pages 502–518, 2003.

[7] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability - Second Edition*, pages 993–1014. 2021.

[8] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39 :297–308, 1985.

[9] Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP 2015*, pages 173–182, 2015.

[10] Daniel Le Berre and Laurent Simon. The essentials of the SAT 2003 competition. In *SAT 2003*, pages 452–467, 2003.

[11] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability - Second Edition*, pages 133–182. 2021.

[12] Ciaran McCreesh, Patrick Prosser, and James Trimble. The glasgow subgraph solver : Using constraint programming to tackle hard subgraph isomorphism problem variants. In *ICGT 2020*, pages 316–324, 2020.

[13] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *SAT 2018*, pages 111–121, 2018.

[14] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004*, 2004.

[15] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2002 competition. *Ann. Math. Artif. Intell.*, 43(1) :307–342, 2005.

[16] Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In *SAT 2006*, pages 424–429, 2006.

[17] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim : Efficient checking and trimming using expressive clausal proofs. In *SAT 2014*, pages 422–429, 2014.

[18] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker : Practical implementations and other applications. In *DATE 2003*, pages 10880–10885, 2003.