Explications pour les Solveurs SAT

Anthony Blomme

UNIVERSITÉ D'ARTOIS

École Doctorale Sciences Technologies et Santé n° 585

Explications pour les solveurs SAT

THÈSE

présentée et soutenue publiquement le 18 décembre 2023

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Anthony Blomme

Composition du jury

Président :	Laurent Simon	Université de Bordeaux
Encadrants :	Daniel Le Berre	Université d'Artois
	Anne Parrain	Université d'Artois
	Olivier Roussel	Université d'Artois
Rapporteurs :	Olivier Bailleux	Université de Bourgogne
	Djamal Habet	Université Aix-Marseille

UNIVERSITÉ D'ARTOIS

Région Hauts-de-France

Centre de Recherche en Informatique de Lens – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : +33 (0)3 21 79 17 23
http://www.cril.univ-artois.fr

Mise en page avec memcril (B. Mazure, CRIL) et thloria (D. Roegel, LORIA).

Remerciements

Dans un premier temps, je voudrais remercier Laurent Simon, qui a accepté d'être président du jury de thèse, ainsi que les deux rapporteurs Olivier Bailleux et Djamal Habet, dont j'ai eu plaisir de lire les rapports. Je les remercie aussi pour les questions pertinentes et intéressantes posées au cours de la soutenance.

Je voudrais également remercier mes encadrants Daniel Le Berre, Anne Parrain et Olivier Roussel pour leur disponibilité et leurs conseils avisés au cours de ces trois années de thèse.

Je remercie également les autres membres du CRIL et tout particulièrement les autres doctorants pour la bonne ambiance qui régnait généralement dans les bureaux.

Finalement, je remercie mes parents qui m'ont toujours soutenu dans mes études et sans qui je ne serais pas allé aussi loin.

À mes parents

Table des matières

Partie II Contributions

Table des figures

Introduction Générale

L'informatique est un domaine en expansion continue depuis ses débuts, mais avec l'intelligence artificielle elle est de plus en plus présente dans notre vie et dans notre quotidien. Cela se caractérise notamment par le fait que de nombreuses décisions sont prises par des programmes informatiques et ce même pour des problèmes impliquant de grandes responsabilités ou impliquant des vies humaines comme la médecine.

Même si dans certains domaines, la prise de décision par un algorithme ne fait pas débat, pour des problèmes plus sensibles, notamment ceux touchant aux personnes, il est plus difficile d'accepter une décision provenant d'un logiciel. Cette réticence est aussi renforcée par l'apparente complexité des logiciels et par la difficulté de vérifier la validité de la décision. La loi de 1978 relative à l'informatique, aux fichiers et aux libertés[1], indique : « *Aucune décision produisant des effets juridiques à l'égard d'une personne ou l'affectant de manière significative ne peut être prise sur le seul fondement d'un traitement automatisé de données à caractère personnel* ».

Il semble donc que l'explication des résultats retournés soit une évolution naturelle des applications informatiques. En effet, de la même façon qu'il peut être demandé à une personne de justifier les choix qu'elle a faits, il devrait être tout à fait normal de demander à un logiciel d'expliquer les résultats qu'il a obtenus. Cela pourrait permettre de faire en sorte que les décisions prises par des programmes soient plus largement acceptées. En effet, cela a été évoqué dans le rapport Villani (partie 6) : « *A long terme, l'explicabilité des technologies de l'intelligence artificielle est l'une des conditions de leur acceptabilité sociale. C'est pourquoi la puissance publique doit agir de différentes manières : [(point 1)] développer la transparence et l'audit des algorithmes [(alinéa b)] en soutenant la recherche sur l'explicabilité de l'IA. Pour cela, il faut investir autour de trois axes de recherche : la production de modèles plus explicables, la production d'interfaces utilisateurs plus intelligibles, et enfin la compréhension des mécanismes à l'œuvre pour produire une explication satisfaisante* ». De plus, générer des explications peut également être un moyen de vérifier que le programme s'est comporté de la façon souhaitée.

Ce besoin devient même une exigence légale : le règlement (UE) 2016/679 du parlement européen et du conseil du 27 avril 2016, relatif à la protection des personnes physiques à l'égard du traitement des données à caractère personnel et à la libre circulation de ces données (alias Règlement Général sur la Protection des Données, ou RGPD) qui s'applique depuis le 25 mai 2018, précise dans ses considérations liminaires (numéro 71) que « *en tout état de cause, un traitement de ce type [automatisé de données à caractère personnel] devrait être assorti de garanties appropriées, qui devraient comprendre une information spécifique de la personne concernée ainsi que le droit d'obtenir une intervention humaine, d'exprimer son point de vue, d'obtenir une explication quant à la décision prise à l'issue de ce type d'évaluation et de contester la décision.* », mais aussi dans ses articles 14, paragraphe 2, alinéa g) et 15, paragraphe 1, alinéa h) que « *l'existence d'une prise de décision automatisée, y compris un profilage, visée à l'article 22, paragraphes 1 et*

1. Article 47 de https ://www.legifrance.gouv.fr/affichTexte.do ?cidTexte=JORFTEXT000000886460.

4, et, au moins en pareils cas, des informations utiles concernant la logique sous-jacente, ainsi que l'importance et les conséquences prévues de ce traitement pour la personne concernée. », ou encore dans son article 22, paragraphe 1, que « *la personne concernée a le droit de ne pas faire l'objet d'une décision fondée exclusivement sur un traitement automatisé, y compris le profilage, produisant des effets juridiques la concernant ou l'affectant de manière significative de façon similaire.* ».

Ce besoin d'explication concerne tout type de programme informatique. C'est aussi le cas pour les programmes issus de la recherche, comme par exemple les solveurs de contraintes, et plus particulièrement les solveurs SAT étudiés dans cette thèse. Si des travaux existent depuis longtemps pour concevoir des solveurs de contraintes explicables [JO01], l'enjeu aujourd'hui est d'apporter des explications aux réponses données par des systèmes existants. C'est notamment important pour les solveurs SAT dont l'usage s'est généralisé depuis une vingtaine d'années, d'abord dans le monde de la recherche puis dans la société.

Dans cette thèse, nous allons commencer par rappeler certaines notions importantes concernant la logique propositionnelle et les solveurs SAT. Cette première partie va également nous permettre d'étudier la notion d'explication et de voir ce qui peut constituer une justification pour un solveur SAT. Puis la partie suivante sera consacrée aux deux méthodes développées au cours de la thèse. La première méthode utilise une détection sémantique de problèmes de pigeons grâce à la reconnaissance de contraintes de cardinalité alors que la seconde repose sur la détection syntaxique de motifs récurrents par l'intermédiaire d'un système de cache. Dans les deux cas, le but est ici d'identifier comme UNSAT certaines branches de l'arbre de recherche. Cela va permettre d'élaguer une partie de l'arbre de recherche et donc de réduire sa taille. L'arbre de recherche sera ainsi potentiellement plus facile à comprendre et plus assimilable pour un être humain. Les diverses méthodes ont été expérimentées avec plusieurs configurations différentes. Enfin, nous pourrons conclure et évoquer quelques améliorations et futures pistes de recherche.

Première partie

Préliminaires

Chapitre 1

Le problème SAT et les solveurs SAT

1.1 Logique propositionnelle

Au cours de ce mémoire, nous allons considérer des notions basiques de logique propositionnelle que nous allons ici rappeler. Ces dernières nous permettront par la suite de définir le *problème de la cohérence booléenne*, ou *problème SAT* [DP60, Coo71, BHvMW21], qui nous a particulièrement intéressé tout au long de la thèse. Tout d'abord, nous rappelons les définitions suivantes :

Définition 1 (Variable propositionnelle)**.** *Une* variable propositionnelle *est un symbole auquel on associe une valeur de vérité qui sera soit* vrai *(1, True ou* $\top$*) soit* faux *(0, False ou* $\bot$*). On note* V *l'ensemble des variables propositionnelles.*

Définition 2 (Littéral)**.** *Un* littéral *désigne soit une variable propositionnelle* $v \in V$ *soit sa négation* $\neg v$*.*

Définition 3 (Clause)**.** *Une* clause *est une disjonction de littéraux, qui peut également être vue comme un ensemble de littéraux. La taille d'une clause correspond au nombre de littéraux présents dans celle-ci.*

Définition 4 (Formule)**.** *Une* formule *propositionnelle est définie récursivement de la manière suivante :*

- $\top$, $\bot$ *et les variables propositionnelles sont des formules.*
- *Soient* ϕ_1 *et* ϕ_2 *deux formules propositionnelles, alors* $\neg\phi_1$, $(\phi_1 \wedge \phi_2)$, $(\phi_1 \vee \phi_2)$, $(\phi_1 \rightarrow \phi_2)$, $(\phi_1 \leftrightarrow \phi_2)$ *sont aussi des formules.*

Les connecteurs logiques sont définis de manière habituelle.

Définition 5 (Formule CNF)**.** *Une formule propositionnelle est sous* Forme Normale Conjonctive *(ou* CNF *pour* Conjunctive Normal Form*) si elle correspond à une conjonction de clauses, qui peut aussi être vue comme un ensemble de clauses.*

Exemple 1. *Soit* $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$*.* ϕ *est une formule propositionnelle sous Forme Normale Conjonctive composée d'une clause de taille 3 et de deux clauses de taille 2. Cette formule est construite à partir des variables* x_1*,* x_2 *et* x_3*.* ϕ *peut également être représentée de la façon ensembliste suivante :* $\{\{x_1, x_2, x_3\}, \{\neg x_1, x_2\}, \{x_1, \neg x_2\}\}$*.*

Une formule CNF correspond donc à un ensemble de *contraintes* que nous devons satisfaire. Dans cette thèse, nous allons également utiliser certaines contraintes particulières, que sont les contraintes de cardinalité.

Définition 6 (Contrainte de cardinalité)**.** *Une* contrainte de cardinalité *est une fonction booléenne de la forme* $\sum_{i=1}^{n} l_i \otimes k$ *où* $\otimes \in \{<, \leq, =, \geq, >\}$*,* l_i *sont des littéraux et* k *est un entier représentant le* seuil*.*

Pour satisfaire une contrainte de cardinalité, il faut compter le nombre de littéraux l_i satisfaits dans la partie gauche de la contrainte et le comparer au seuil pour voir si l'opération utilisée au niveau du $\otimes$ est bien respectée. Toute clause de la forme $\bigvee_{i=1}^{n} l_i$ peut être représentée sous la forme d'une contrainte de cardinalité de la forme $\sum_{i=1}^{n} l_i \geq 1$.

Exemple 2. *La clause* $x_1 \vee x_2 \vee x_3$ *peut être réécrite sous la forme* $x_1 + x_2 + x_3 \geq 1$*. Cette contrainte est satisfaite si au moins un des littéraux* x_1*,* x_2 *ou* x_3 *est satisfait. La contrainte* $x_1 + x_2 + x_3 \leq 2$ *est satisfaite si au plus deux littéraux parmi* x_1*,* x_2 *et* x_3 *sont satisfaits.*

On s'intéressera notamment à deux contraintes de cardinalité spécifiques, $\sum_{i=1}^{n} l_i \geq 1$ que nous nommerons *at least 1* et $\sum_{i=1}^{n} l_i \leq 1$ que nous nommerons *at most 1*.

Lorsque l'on considère une formule ϕ, il est possible d'assigner certaines valeurs aux variables présentes dans la formule. L'ensemble des assignations ainsi effectuées s'appelle une *affectation* (ou *interprétation*) et celle-ci peut être utilisée pour simplifier ϕ. Une affectation peut être définie de la façon suivante.

Définition 7 (Affectation)**.** *Soit* V *l'ensemble des variables présentes dans une formule* ϕ*. Une* affectation α *est une fonction qui associe à chaque variable de* V *une valeur de vérité parmi* $\bot$ *(pour* $faux$*) ou* $\top$ *(pour* $vrai$*). En d'autres termes, nous avons* $\alpha : V \to \{\top, \bot\}$*. Il est également possible de représenter une affectation par l'ensemble des littéraux qu'elle satisfait (i.e.* v *si la variable* v *est assignée à* $vrai$*, et* $\neg v$ *si la variable* v *est assignée à* $faux$*). Une affectation est dite* complète *si toutes les variables de* V *ont reçu une valeur de vérité et* partielle *si ce n'est pas le cas.*

Exemple 3. *Soit* $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ *une formule CNF définie sur* $V = \{x_1, x_2, x_3\}$*. Alors* $\alpha_1 = \{x_1, x_2\}$ *est une affectation partielle et* $\alpha_2 = \{x_1, x_2, \neg x_3\}$ *est une affectation complète.*

Nous choisissons la représentation d'une affectation sous la forme de l'ensemble des littéraux satisfaits. Pour simplifier une formule CNF ϕ suivant une affectation α, nous considérons chaque clause C de ϕ. Si C est satisfaite, c'est-à-dire si au moins un des littéraux présents dans C est aussi présent dans α, nous supprimons la clause de ϕ. Dans tous les cas, nous supprimons aussi de C tous les littéraux falsifiés, i.e. ceux dont le littéral opposé est présent dans α. Nous notons $C_{|\alpha}$ la clause obtenue en réalisant cette opération sur la clause C avec l'affectation α et nous notons $\phi_{|\alpha}$ la formule propositionnelle simplifiée obtenue de cette manière. Lorsqu'une formule ϕ est simplifiée par une affectation α, si l'une des clauses restantes ne contient plus qu'un seul littéral l (on dit qu'elle est *unitaire*), alors celui-ci est nécessairement vrai et nous pouvons donc l'ajouter à α. Cette opération correspond à une *propagation unitaire* et la clause utilisée pour propager l peut être vue comme la *raison* de cette propagation. Réaliser des propagations jusqu'à ce qu'il ne reste plus aucune clause de taille 1 est ce que l'on nomme la *propagation unitaire*. On notera $\vdash_{UP}$ l'application de la propagation unitaire et $\phi \vdash_{UP} l$ lorsqu'un littéral l est obtenu par propagation unitaire.

Une clause est dite *satisfaite* si au moins l'un de ses littéraux est présent dans l'affectation courante. Une formule CNF est *satisfaite* si et seulement si toutes les clauses présentes sont satisfaites. On dit alors que l'affectation α qui a permis de satisfaire la formule CNF ϕ est un *modèle* de ϕ et on note cela par $\alpha \models \phi$. Nous donnons maintenant les définitions d'implication logique et d'impliquant premier :

Définition 8 (Implication logique)**.** *On dit qu'une formule* ϕ_1 implique logiquement *une formule* ϕ_2*, aussi noté* $\phi_1 \models \phi_2$ *si tout modèle de* ϕ_1 *est aussi un modèle de* ϕ_2*.*

Définition 9 (Impliquant premier)**.** *Soit* ϕ *une formule CNF et* α *une affectation.* α *est un* impliquant premier *si et seulement si* $\alpha \models \phi$ *et* $\forall \alpha' \subset \alpha$, $\alpha' \not\models \phi$*.*

En d'autres termes, une conjonction de littéraux α est un impliquant premier de ϕ si retirer n'importe quel littéral de α fait que cette affectation ne satisfait plus ϕ. Ainsi, un impliquant premier est un sous-ensemble minimal pour l'inclusion satisfaisant une formule donnée.

Exemple 4. *Soit* $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ *une formule CNF. Si on considère l'affectation* $\alpha = \{x_1\}$*, on peut alors simplifier* ϕ *et obtenir* $\phi_{|\alpha} = (x_2)$*. Ici, les première et troisième clauses ont été satisfaites par l'affectation et on a supprimé un littéral falsifié de la seconde clause. Si on étend maintenant cette affectation avec le littéral* x_2*, on a alors* $\alpha = \{x_1, x_2\}$ *et on satisfait la formule* ϕ *(on obtient* $\phi_{|\alpha} = \emptyset$*). De plus,* α *est une affectation partielle et est aussi un impliquant premier de* ϕ*. En ajoutant le littéral* $\neg x_3$*, l'affectation* $\alpha = \{x_1, x_2, \neg x_3\}$ *devient complète mais nous n'avons plus un impliquant premier de* ϕ*. Ici, le littéral* $\neg x_3$ *n'a pas de répercussion sur la cohérence de* ϕ *en présence des deux autres littéraux.*

Il est également possible d'effectuer des opérations entre clauses comme la *résolution* qui permet d'obtenir une nouvelle clause (que l'on appelle une *résolvante*) à partir de deux autres clauses dont on disposait déjà.

Définition 10 (Résolvante)**.** *Soit un littéral* l *et deux clauses de la forme* $l \vee A$ *et* $\neg l \vee B$ *où* A *et* B *sont deux disjonctions. La* résolvante *de ces deux clauses est la clause* $A \vee B$*.*

En d'autres termes, si les deux littéraux opposés de la même variable sont présents dans deux clauses différentes, alors il est possible de créer une nouvelle clause (appelée *résolvante*) constituée des autres littéraux présents dans les clauses utilisées pour cette opération (qui est appelée une *résolution*). Si les clauses utilisées pour l'étape de résolution contiennent d'autres littéraux complémentaires en dehors de celui qui est supprimé, alors la résolvante est tautologique (i.e. toujours vraie car elle contient deux littéraux opposés). Les résolvantes pouvant être obtenues à partir des clauses d'une formule ϕ sont conséquences logiques de cette formule.

Exemple 5. *Soient* $x_1 \vee x_2 \vee x_3$ *et* $\neg x_1 \vee x_4$ *deux clauses, alors la résolvante de ces deux clauses est* $x_2 \vee x_3 \vee x_4$*. De plus, nous avons que* $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \models x_2 \vee x_3 \vee x_4$*.*

1.2 Le problème SAT

1.2.1 Présentation

Avec ce que nous avons vu, il est possible de définir le problème SAT [DP60, Coo71, BHvMW21].

Définition 11 (Problème SAT)**.** *Le* problème de la cohérence booléenne *(ou* problème SAT *pour* Boolean SATisfiability*) correspond au problème de décision suivant :*

- *Entrée : Une formule CNF* ϕ *construite sur* n *variables propositionnelles*
- *Question : Existe-t-il une affectation* α *des* n *variables propositionnelles qui satisfait la formule* ϕ *?*

En d'autres termes, le problème SAT consiste à regarder si une formule CNF donnée admet au moins un modèle. S'il est possible de trouver une telle affectation, alors la formule est dite *cohérente* (ou *SAT*). Dans le cas contraire, si aucun modèle n'existe, elle est dite *incohérente* (ou *UNSAT*). Un programme capable de résoudre ce problème est appelé un *solveur SAT*. Dans le cas d'une instance cohérente, en plus de sa réponse, le solveur peut fournir un modèle s'il en dispose d'un. Si nous considérons l'algorithme DP [DP60], celui-ci effectue toutes les résolutions possibles et si on obtient une clause vide de cette manière, alors l'instance considérée est incohérente. Si toutes les résolvantes ont pu être obtenues sans obtenir la clause vide, alors l'instance est cohérente. Ainsi, cette approche est capable de dire qu'une instance est cohérente sans générer de modèle.

Il est intéressant d'utiliser le format CNF puisqu'il est possible de convertir toute formule en temps polynomial au format CNF par l'intermédiaire de la méthode de Tseitin [Tse66] ou de Plaisted-Greenbaum [PG86] qui ajoutent de nouvelles variables. La formule obtenue est équivalente en terme de cohérence et on peut retrouver un modèle de la formule initiale quand on projette les modèles sur les variables initiales.

Le problème SAT a été le premier problème à être prouvé NP-complet [Coo71], ce qui signifie que l'on peut vérifier si une solution est correcte en temps polynomial mais qu'il faut un temps exponentiel pour trouver une solution, sous l'hypothèse généralement admise que $\mathrm{P} \neq \mathrm{NP}$. Il est même considéré comme un problème classique en théorie de la complexité car de nombreux autres problèmes peuvent s'y réduire [GJ79]. Ainsi, si l'on arrive un jour à construire un solveur capable de résoudre le problème SAT en temps polynomial (et donc à prouver que $\mathrm{P} = \mathrm{NP}$), on résoudrait alors également tous ces autres problèmes en temps polynomial. Malgré cette complexité, les solveurs SAT modernes sont capables de résoudre des instances du problème SAT comprenant des millions de clauses [HK17]. Les solveurs SAT sont également utilisés pour résoudre des problèmes dans des domaines très variés, comme la vérification de matériel [Bie21] ou de logiciel [Kro21], la planification [PlanningAsSATHB] ou encore les dépendances de logiciels [ACTZ12]. De plus, les solveurs SAT sont aussi utilisés pour résoudre des problèmes mathématiques combinatoires ouverts tels que les triplets Pythagoriciens [HK17].

En 2002 eut lieu la première édition de la compétition SAT [SLH05] qui se déroule depuis chaque année et dont le but est d'évaluer les solveurs les plus performants du moment, de réunir de nouvelles instances sur lesquelles tester les solveurs mais aussi d'encourager la mise au point d'idées innovantes pour améliorer les performances des solveurs. Si les premiers solveurs présentés n'étaient pas très robustes, cette tendance a changé dès l'année suivante grâce à la mise en commun des instances utilisées et la réponse attendue (i.e. instance cohérente ou incohérente). De ce fait, les recherches autour du problème SAT sont relativement compétitives, ce qui explique les niveaux de performance atteints aujourd'hui. Parmi les solveurs récompensés à cette compétition, nous pouvons par exemple citer Glucose [AS18], Lingeling [Bie14], MapleSAT [LGPC16], CaDiCaL [BFFH20] (vainqueur de l'édition 2023) ou Kissat [BF22] (vainqueur de l'édition 2022).

Afin de pouvoir facilement partager des instances de test et comparer les performances des solveurs, il a été décidé d'utiliser un format standardisé pour représenter la formule CNF qu'un solveur doit résoudre. Cela permet également d'utiliser les solveurs comme des boîtes noires. Le format retenu est le format DIMACS, qui a été proposé lors de la compétition *DIMACS implementation challenge* [JT96] de 1993 et qui est encore utilisé actuellement lors des compétitions. Dans un fichier au format DIMACS, un préfixe, que l'on reconnaît par une ligne commençant par la lettre `p` précise le type de l'instance (`cnf` dans notre cas), puis le nombre de variables et ensuite de clauses de la formule considérée. Ensuite, chaque ligne va représenter une clause. Pour chaque clause, nous représentons les littéraux présents à l'intérieur de celle-ci par l'intermédiaire

de nombres positifs (pour les littéraux positifs) et de nombres négatifs (pour les littéraux négatifs). La valeur 0 indique la fin de la clause. Les lignes commençant par le caractère `c` sont des commentaires et sont donc ignorées par le solveur. La figure 1.1 montre un exemple de formule cohérente et un exemple de formule incohérente toutes les deux exprimées au format DIMACS. La première étape pour un solveur est donc de lire un fichier semblable afin de récupérer la formule qu'il doit résoudre. La résolution en elle-même dépendra de l'architecture du solveur. Au cours du temps, deux architectures principales de solveur SAT ont émergé et nous allons maintenant revenir sur chacune d'entre elles.

$(x_1 \vee x_3) \wedge$
$(x_1 \vee \neg x_3) \wedge$
$(x_2 \vee x_3) \wedge$
$(x_2 \vee \neg x_3)$

```
p cnf 3 4
1 3 0
1 -3 0
2 3 0
2 -3 0
```

$(x_1 \vee x_3) \wedge$
$(x_1 \vee \neg x_3) \wedge$
$(x_2 \vee x_3) \wedge$
$(x_2 \vee \neg x_3) \wedge$
$(\neg x_2 \vee x_4) \wedge$
$(\neg x_1 \vee \neg x_2 \vee \neg x_4)$

```
p cnf 4 6
1 3 0
1 -3 0
2 3 0
2 -3 0
-2 4 0
-1 -2 -4 0
```

FIGURE 1.1 – Exemples de formules CNF cohérente (à gauche) et incohérente (à droite) et leur représentation au format DIMACS.

1.2.2 L'architecture DPLL

L'architecture DPLL (pour *Davis-Putnam-Logemann-Loveland*) [DP60, DLL62] repose sur une exploration complète et en profondeur d'abord d'un arbre binaire de recherche, qui représente ici l'espace de recherche. Le fonctionnement global d'un solveur DPLL est résumé par l'algorithme 1. Le principe général est d'alterner des phases de simplification de la formule par propagation unitaire (ligne 1) et de prise de décision (i.e. une affectation arbitraire d'un littéral à Vrai - ligne 8) qui vont mener le solveur soit à un modèle, soit à un conflit. Dans un solveur, le *niveau de décision* correspond au nombre de décisions qui ont été prises. Lors du premier appel à la fonction *DPLL*, l'affectation α correspond à l'ensemble vide. La fonction *Propagate* effectue toutes les propagations unitaires possibles et renvoie la nouvelle affectation obtenue. Si un conflit est détecté lors de la propagations unitaire, cette fonction peut donner la clause conflit comme second résultat. Lors du premier appel à cette fonction, toutes les clauses unitaires présentes dans la formule initiale vont être propagées. À l'issue de cette phase :

— Si toutes les clauses ont pu être satisfaites, alors la recherche peut s'arrêter et nous pouvons indiquer que la formule considérée est cohérente (lignes 2 à 4) en donnant le modèle trouvé (α).
— Si la formule simplifiée contient la clause vide, alors un conflit a été détecté (ligne 5). Nous pouvons arrêter la recherche pour cet appel récursif en indiquant que nous avons rencontré un conflit (ligne 6). Lorsque nous allons revenir en arrière, nous allons inverser la polarité de la dernière décision qui n'a pas encore été inversée (ligne 11). Cette opération correspond à ce que l'on appelle un *backtrack*. Si à un moment donné, nous avons inversé toutes les décisions possibles dans l'arbre de recherche sans avoir trouvé de modèle, alors dans ce cas nous pouvons en déduire que l'instance est incohérente car nous avons exploré tout l'espace de recherche et qu'aucun modèle ne peut être trouvé.

— Dans les autres cas, le solveur peut prendre une nouvelle décision (ligne 8) et tester une première polarité (ligne 9). La seconde polarité ne sera testée que si aucun modèle n'a été détecté lors de l'exploration de la première polarité (lignes 10 à 12).

Algorithm 1 *DPLL*(ϕ, α)

ENTREE : ϕ - une formule CNF, α - l'affectation courante
SORTIE : La formule ϕ est-elle cohérente (SAT) ou incohérente (UNSAT) ?

```
1 : (α, C) ← Propagate(φ, α)
2 : if φ_|α = ∅ then
3 :    return SAT
4 : end if
5 : if C ≠ Undef then
6 :    return UNSAT
7 : end if
8 : d ← Decide()
9 : r ← DPLL(φ, α ∪ {d})
10 : if r = UNSAT then
11 :    r ← DPLL(φ, α ∪ {¬d})
12 : end if
13 : return r
```

Exemple 6. *À titre d'exemple, supposons que nous avons la formule* $\phi = (x_1 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)$ *et un solveur DPLL qui affecte à faux la première variable non assignée. Alors le solveur va commencer par prendre la décision* $\neg x_1$*, qui va permettre de propager* x_3 *grâce à la clause* $x_1 \vee x_3$*. Cela va nous mener à un conflit, puisque la clause* $x_1 \vee \neg x_3$ *ne peut plus être satisfaite, et nous allons donc inverser la décision* $\neg x_1$*. Une fois cela fait, le solveur va prendre la décision* $\neg x_2$*, qui va également propager* x_3 *par l'intermédiaire de la clause* $x_2 \vee x_3$ *et cela va de nouveau nous amener à un conflit. Après avoir inversé la décision* $\neg x_2$*, nous arrivons à un modèle puisque nous avons satisfait toutes les clauses. Ainsi, nous pouvons arrêter la recherche, indiquer que l'instance est cohérente et donner l'affectation* $\alpha = \{x_1, x_2\}$ *comme modèle. La figure 1.2 résume la recherche effectuée dans cet exemple. Sur cette figure, les arêtes bleu et bleu ciel représentent respectivement les décisions du solveur et les inversions de décision. Les arêtes vertes représentent les propagations unitaires. Ensuite, les pointillés représentent les retours en arrière. Finalement, les boîtes rouges représentent les conflits et la clause conflit est indiquée à l'intérieur de la boîte.*

Supposons maintenant que l'on a en plus des clauses de ϕ *les deux clauses* $\neg x_2 \vee x_4$ *et* $\neg x_1 \vee \neg x_2 \vee \neg x_4$*. Dans ce cas, après avoir inversé les deux décisions* $\neg x_1$ *et* $\neg x_2$*, nous pouvons propager* x_4 *avec la clause* $\neg x_2 \vee x_4$ *et nous arrivons à un conflit. À ce moment-là, toutes les décisions prises jusqu'à maintenant ont déjà été inversées. Nous avons donc exploré tout l'espace de recherche sans trouver de modèle, nous pouvons donc arrêter la recherche et indiquer que l'instance est incohérente. La figure 1.3 résume la recherche effectuée avec cette version de l'exemple. Nous utilisons le même code couleur que pour l'exemple précédent.*

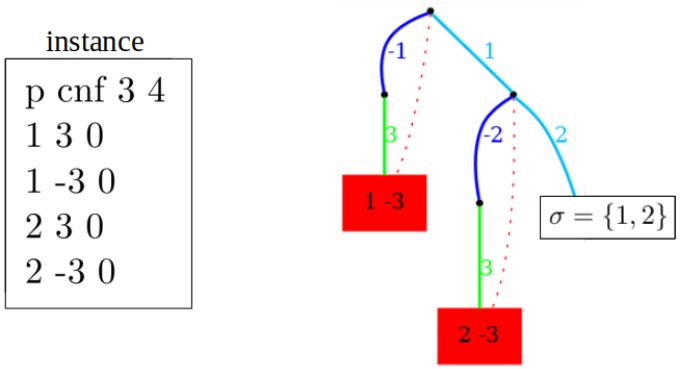


FIGURE 1.2 – Exemple d'arbre de recherche pour l'approche DPLL sur une instance cohérente.

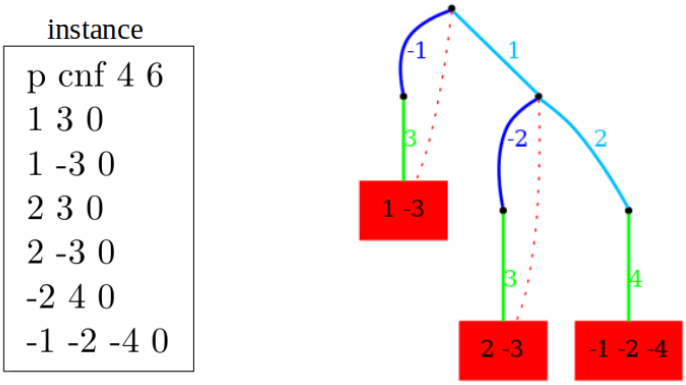


FIGURE 1.3 – Exemple d'arbre de recherche pour l'approche DPLL sur une instance incohérente.

À noter

Il est intéressant de remarquer que l'architecture DPLL permet de savoir pour un nœud de l'arbre si la formule correspondant à ce nœud est incohérente ou non. En effet, pour un nœud donné, si nous avons exploré les deux sous-arbres fils, l'un pour une décision et l'autre pour la négation de cette décision, et que nous les avons montrés comme étant incohérents (c'est-à-dire que toutes les branches de ces sous-arbres mènent à un conflit), alors nous pouvons faire remonter l'incohérence et en déduire que ce nœud est également incohérent.

Dans un solveur, la façon dont les décisions sont prises, ce que l'on nomme aussi *l'heuristique*, est une composante fondamentale qui va avoir un fort impact sur la manière dont l'espace de recherche va être exploré. En effet, pour une instance donnée, une heuristique qui favorise les propagations unitaires va diminuer le nombre de décisions à inverser lors des retours en arrière suite aux conflits, par rapport à une heuristique qui ne les favorise pas [Kul21].

1.2.3 L'architecture CDCL

Autour des années 2000, une nouvelle architecture appelée CDCL (pour *Conflict Driven Clause Learning*) [MLM21] est apparue par l'intermédiaire de solveurs tels que GRASP [SS99],

Chaff [MMZ+01] ou encore MiniSat [ES03]. Elle est actuellement l'approche état-de-l'art pour la résolution du problème SAT et a eu pour conséquence que les solveurs SAT sont devenus des programmes fréquemment utilisés comme oracles dans la résolution de certains problèmes NP-complets [MLM21]. Cette architecture a succédé à l'approche DPLL et est implémentée dans la plupart des solveurs SAT modernes. Contrairement à son prédécesseur, l'approche CDCL explore généralement l'espace de recherche de manière non chronologique en se basant sur des opérations telles que l'analyse de conflit et l'apprentissage de clause pour diriger la recherche. Nous allons maintenant revenir sur les principales opérations implémentées dans une architecture CDCL.

Nous supposons que la fonction *DL(l)* nous donne le niveau de décision auquel le littéral l a été affecté. Cette fonction peut également être utilisée pour récupérer le niveau de décision le plus élevé parmi les littéraux d'une clause. De même, nous supposons qu'une fonction $reason(l)$ nous donne la raison qui a permis de propager le littéral l.

L'analyse de conflit

Tout d'abord, le solveur GRASP [SS99] a introduit la notion d'*analyse de conflit* dans la recherche d'un solveur. Dans un solveur CDCL, à chaque fois qu'un conflit est rencontré, une étape dite d'analyse de conflit est déclenchée. Le but de cette opération est d'empêcher le solveur d'effectuer à nouveau une mauvaise combinaison de décisions. À l'origine, l'analyse de conflit a été présentée dans GRASP avec des coupes dans le graphe d'implication du solveur. Du point de vue de la logique, nous nous basons sur la notion de résolvante afin de réaliser l'analyse de conflit et construire une clause à apprendre. L'analyse de conflit va partir du conflit et va réaliser des résolutions en considérant les raisons qui ont mené à ce conflit, jusqu'à un certain point appelé un *UIP* (pour *Unique Implication Point*). Par la suite, le solveur Chaff [MMZ+01] a défini la notion de *1-UIP* (pour *First Unique Implication Point*). Nous savons que nous avons atteint ce point lorsque la résolvante que nous avons pour le moment contient au plus un littéral du niveau de décision courant. Il est possible de continuer les résolutions mais il a été montré que de meilleurs résultats sont obtenus lorsque des 1-UIP sont utilisés [ABH+08]. Cette opération va permettre de déduire une nouvelle clause qui pourra être apprise par le solveur. Cette *clause apprise* (par opposition aux *clauses initiales* qui sont celles déjà présentes à la base dans la formule considérée) va mettre en évidence un littéral impliqué précédemment dans la recherche mais qui n'était pas dérivable par propagation unitaire avec les clauses à disposition à ce moment-là. Le solveur va donc annuler les décisions qu'il a prises jusqu'à ce point pour ensuite effectuer une propagation unitaire à partir de celle induite par la clause nouvellement apprise. Cette opération, que l'on nomme un *backjump*, implique une exploration non chronologique de l'espace de recherche. Comme les clauses apprises sont obtenues grâce à des opérations de résolution, elles sont impliquées par la formule initiale et donc ajouter des clauses apprises ne change en rien la cohérence de la formule et ne change pas non plus les modèles pouvant être trouvés.

Exemple 7. *Afin d'illustrer l'analyse de conflit, nous considérons la formule* $\phi = (x_1 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$ *et nous supposons disposer d'un solveur CDCL qui décide les variables négativement en commençant par la dernière. Le solveur va donc commencer par décider* $\neg x_4$*, qui va permettre de propager successivement* $\neg x_2$ *grâce à la clause* $\neg x_2 \vee x_4$ *et ensuite* x_3 *grâce à la clause* $x_2 \vee x_3$*. Suite à ces propagations, la clause* $x_2 \vee \neg x_3$ *ne peut plus être satisfaite et nous avons donc rencontré un conflit. Effectuer une résolvante entre ce conflit et* $x_2 \vee x_3$ *nous permet d'obtenir d'obtenir la clause unitaire* x_2*. À ce moment là, nous pouvons arrêter l'analyse de conflit et apprendre la clause* x_2*. Nous pouvons aussi revenir à la racine de l'arbre et propager le littéral* x_2*, qui va nous permettre d'arriver à un*

nouveau conflit après quelques propagations. Ici, nous pouvons arrêter la recherche et indiquer que l'instance est incohérente car nous avons rencontré un conflit au niveau de décision 0. La recherche effectuée dans cet exemple est représentée à la figure 1.4. Ici, nous utilisons un code couleur comparable à celui utilisé avec les exemples de l'architecture DPLL sauf qu'ici nous n'avons d'inversion de décision. À la place, nous utilisons des arêtes oranges pour représenter les propagations suite à l'apprentissage d'une clause.

Si l'on avait eu un solveur DPLL, nous n'aurions pas appris la clause unitaire x_2 et suite au premier conflit, nous aurions inversé la décision $\neg x_4$. Suite à cela, le solveur aurait pu prendre une nouvelle décision, à savoir $\neg x_3$ dans notre cas. Cette décision nous aurait fait propager les littéraux x_1 (grâce à la clause $x_1 \vee x_3$) et x_2 (grâce à la clause $x_2 \vee x_3$) et nous serions arrivés à un conflit. Inverser la décision $\neg x_3$ nous aurait également menés à un conflit et aurait arrêté la recherche. Ainsi, nous aurions obtenu un arbre de recherche constitué de 3 branches alors qu'avec une approche CDCL nous avons obtenu un arbre de recherche à 2 branches.

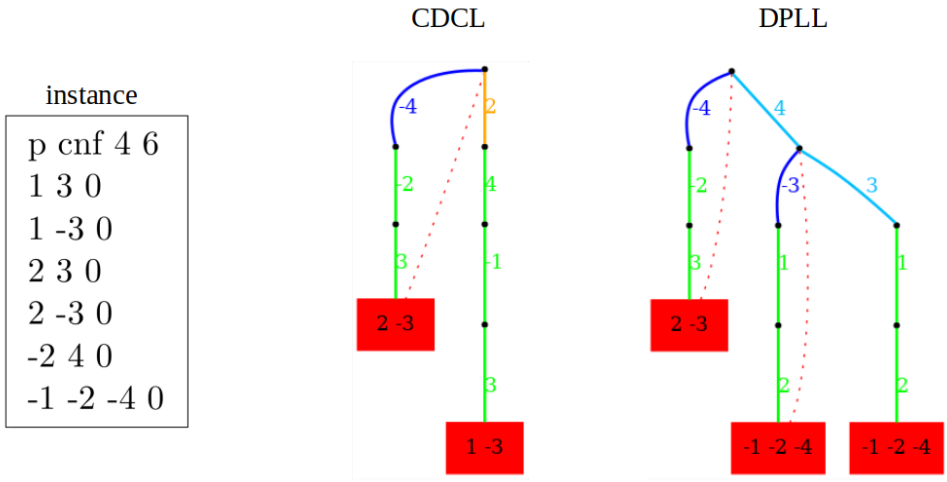


FIGURE 1.4 – Exemple d'arbre de recherche pour l'approche CDCL sur une instance incohérente. L'arbre de recherche pour l'approche DPLL avec la même heuristique est ensuite donné à titre de comparaison.

Cependant, pour les solveurs implémentant un retour arrière chronologique [NR18], le backjump ne s'effectue pas toujours de cette façon et il est possible de revenir au niveau de décision précédent. L'intérêt de cette méthode est de pouvoir détecter plus rapidement un modèle dans certains cas. En effet, si un modèle peut être trouvé à proximité du conflit où nous sommes arrivés (par exemple, en inversant la dernière décision), il peut être intéressant d'effectuer un retour arrière chronologique plutôt qu'un backjump classique qui pourrait éventuellement nous éloigner d'un modèle.

À noter

Contrairement à l'architecture DPLL, il n'est pas toujours possible de savoir si un nœud de l'arbre de recherche est UNSAT au cours de la recherche. L'exemple suivant permet d'illustrer cela.

Exemple 8. *Considérons une formule constituée des clauses $x \vee y \vee z$ et $\neg x \vee y$ ainsi que d'un*

ensemble F de clauses indépendantes des variables x, y et z. Comme nous nous situons au cours de la recherche, nous ne savons pas encore si l'instance considérée est cohérente ou non. Ensuite, considérons une heuristique qui va se comporter de la manière suivante : elle va d'abord prendre des décisions autres que x, y ou z, puis elle va décider $\neg z$. Cette décision va simplifier le littéral z présent dans la clause $x \vee y \vee z$. Ensuite, l'heuristique va prendre d'autres décisions avant de décider $\neg y$. De cette façon, le littéral y va être supprimé des clauses $x \vee y \vee z$ et $\neg x \vee y$, ce qui va mener à un conflit car il nous faut alors satisfaire à la fois x et $\neg x$. Une analyse de conflit va donc être enclenchée et celle-ci va nous donner la clause $y \vee z$, qui est la résolvante des clauses $x \vee y \vee z$ et $\neg x \vee y$. Cette clause apprise va nous faire revenir au niveau de la décision $\neg z$ pour nous faire propager directement y. Ce comportement est illustré à la figure 1.5

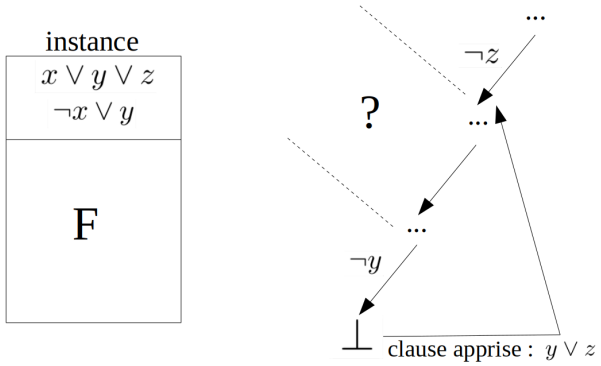


FIGURE 1.5 – Illustration du problème induit par la recherche non chronologique des solveurs CDCL.

Sur cet exemple, on voit bien que les nœuds entre les décisions $\neg z$ et $\neg y$ vont être ignorés et donc il n'est pas possible de savoir si ces nœuds ignorés sont UNSAT ou non. En d'autres termes, effectuer un backjump ne signifie pas forcément que les nœuds ignorés lors de cette opération sont UNSAT. Ce comportement est illustré par notre exemple. En effet, la décision y aurait pu être prise à n'importe quel moment après la décision $\neg z$. Cela aurait satisfait les clauses $x \vee y \vee z$ et $\neg x \vee y$ et la cohérence de la formule n'aurait plus dépendu que de F, qui peut très bien être cohérente.

Malgré cela, il est tout de même possible de déduire que certains nœuds d'un arbre de recherche sont UNSAT. En effet, au niveau des feuilles, nous savons qu'un conflit est apparu et qu'il n'est donc plus possible de satisfaire l'une des contraintes du problème. Nous pouvons donc en conclure que les feuilles de l'arbre sont UNSAT. De plus, si nous pouvons atteindre un nœud n_2 à partir d'un nœud n_1 en utilisant uniquement des propagations unitaires et si nous savons que le nœud n_2 est UNSAT, alors nous pouvons en déduire que n_1 est également incohérent. En d'autres termes, à partir d'un conflit, nous pouvons remonter dans la branche considérée jusqu'à atteindre une décision et considérer tous les nœuds intermédiaires comme étant UNSAT.

Watched literals, heuristique et redémarrages

Par la suite, le solveur Chaff [MMZ+01] a quant à lui introduit l'utilisation de structures dites paresseuses dans les solveurs CDCL afin de réaliser certaines opérations de manière efficace. Concernant la propagation unitaire, le solveur utilise des *watched literals* afin de ne réaliser des propagations unitaires que sous certaines conditions. En effet, pour chaque clause, nous n'allons observer que deux littéraux non encore affectés. À chaque affectation, nous ne considérons que

les clauses qui ont eu un de leurs deux watched literals falsifié. Si le second watched literal n'est pas satisfait, alors nous parcourons la clause à la recherche d'un littéral non encore affecté afin de remplacer celui qui vient d'être falsifié. Si on en trouve un, on effectue alors le remplacement. Sinon, cela signifie que le second watched literal doit être propagé car c'est la dernière possibilité pour pouvoir satisfaire la clause considérée. Cependant, si le second watched literal est lui aussi falsifié, alors nous avons trouvé un conflit. Utiliser ce type de structure fait que l'unique moyen de savoir si toutes les clauses ont été satisfaites (et donc si la formule considérée est cohérente) est de vérifier que toutes les variables présentes dans l'instance ont été affectées. Ainsi, un modèle retourné par un solveur CDCL correspond toujours à une affectation complète. Étant donné que la propagation unitaire est une opération extrêmement courante à l'intérieur des solveurs SAT, il est en effet intéressant de chercher à les effectuer de la manière la plus efficace possible.

Le solveur Chaff a également proposé l'utilisation d'une nouvelle heuristique nommée *VSIDS* (pour *Variable State Independent Decaying Sum*). Celle-ci attribut un score à chaque variable et décide en priorité les variables ayant le score le plus élevé. Chaque fois qu'un conflit est rencontré, le score des variables impliquées dans l'analyse de conflit est augmenté. Afin d'empêcher le score d'atteindre des valeurs élevées, les scores sont périodiquement divisés par une certaine valeur. Le but ici est d'essayer d'affecter en priorité les variables qui ont récemment mené à un conflit. Il est possible d'initialiser le score des variables au nombre d'occurrences dans les clauses initiales mais cela n'est pas toujours implémenté. Par la suite d'autres heuristiques ont été proposées comme par exemple EVSIDS (pour *exponential VSIDS*) [ES03], ou encore LRB (pour *Learning Rate Branching*) [LGPC16].

Les solveurs CDCL effectuent aussi régulièrement des *redémarrages*. Lors de cette opération, toutes les décisions prises et les propagations en découlant sont effacées et la recherche recommence depuis le début. Lors d'un redémarrage, les clauses apprises et les mises à jour de scores sont toutefois conservées par le solveur. Cela a pour but d'empêcher le solveur de rester bloquer dans un espace difficile de la recherche. Il existe de nombreuses heuristiques permettant de décider quand effectuer des redémarrages. Si nous n'avons aucune information concernant ces redémarrages, il est toujours possible de se baser sur la suite de Luby [LSZ93].

Algorithme

L'algorithme 2 résume le comportement global d'un solveur CDCL. Ici, le solveur va commencer par effectuer une première étape de propagation unitaire afin d'éliminer les clauses unitaires présentes dans la formule (ligne 3). Par la suite, une étape de propagation va être effectuée après chaque décision (ligne 3). En fonction du résultat de cette propagation, le solveur peut avoir deux comportements. Si une clause vide est apparue (ligne 4), nous avons alors un conflit. Si ce conflit est arrivé au niveau de décision 0, alors nous pouvons en conclure que l'instance est incohérente et arrêter la recherche (lignes 5 à 7). Sinon, nous pouvons lancer une analyse de conflit qui va nous donner une clause à apprendre (ligne 8). Lors de cette analyse de conflit, diverses mises à jour de scores vont avoir lieu. Ensuite, nous pouvons ajouter la clause apprise à la formule considérée et nous revenons en arrière (lignes 9 et 10). Pour ce faire, la fonction $Backjump(C, \alpha)$ va commencer par récupérer le niveau du retour en arrière. Ce dernier correspond au niveau de décision le plus élevé numériquement présent dans la clause apprise C et qui est différent du niveau de décision courant. Ensuite, l'affectation α est mise à jour en supprimant les décisions et propagations effectuées après le niveau de retour. Si la propagation unitaire s'est terminée sans rencontrer de conflit, alors nous pouvons prendre une nouvelle décision (ligne 18). À ce moment-là, si toutes les variables ont été affectées nous savons alors que la formule est cohérente et l'affectation courante est un modèle (lignes 19 à 21). Sinon, nous ajoutons la

décision dans l'affectation courante (ligne 22) et nous allons la propager pendant la prochaine propagation unitaire (ligne 3). Au cours de la recherche, le solveur va régulièrement effectuer des redémarrages (lignes 12 à 14). De plus, comme le nombre de clauses apprises peut être exponentiel, les solveurs vont régulièrement supprimer certaines clauses dans la base de clauses (lignes 15 à 17). Concernant les diverses fonctions présentes dans l'algorithme, nous ne donnons pas de détails d'implémentation car ces fonctions sont très dépendantes de la politique implémentée dans le solveur.

Algorithm 2 *CDCL*(ϕ)

ENTREE : ϕ - une formule CNF
SORTIE : La formule ϕ est-elle cohérente (SAT) ou incohérente (UNSAT) ?

```
1 : α ← ∅
2 : while true do
3 :    (α, C) ← Propagate(φ, α)
4 :    if C ≠ Undef then
5 :       if CurrentDecisionLevel() = 0 then
6 :          return UNSAT
7 :       end if
8 :       C1 ← AnalyzeConflict(C)
9 :       φ ← φ ∪ {C1}
10 :      α ← Backjump(C1, α)
11 :   else
12 :      if needRestart then
13 :         Restart()
14 :      end if
15 :      if needCleanDB then
16 :         CleanDB()
17 :      end if
18 :      l ← Decide()
19 :      if l = Undef then
20 :         return SAT
21 :      end if
22 :      α ← α ∪ {l}
23 :   end if
24 : end while
```

1.2.4 Autres améliorations internes des solveurs SAT

Une autre amélioration au niveau des solveurs SAT a été l'implémentation du solveur MiniSat [ES03]. Ce solveur est basé sur l'architecture CDCL et a été mis au point dans l'optique d'avoir un solveur facilement extensible et dont le code est facile à lire et compact. Ce solveur a remporté la compétition SAT 2005 grâce notamment à sa minimisation des clauses apprises [SB09] mais également grâce à son préprocesseur SatElite [EB05]. Les solveurs SAT modernes utilisent encore la notion de *trail* introduite dans MiniSat pour représenter l'arbre de recherche et des solveurs basés sur MiniSat, comme Glucose [AS18], étaient encore présents lors des dernières compétitions

SAT. De plus, le code de ce solveur est disponible en ligne et décrit dans un article [ES03]. À l'époque, les codes de certains solveurs étaient disponibles aux chercheurs. C'était le cas par exemple des solveurs GRASP, RELSAT, SATO, SATZ. Aujourd'hui, le code des solveurs participant à la compétition SAT doit être mis à disposition de la communauté scientifique, ce qui permet de tester de nouvelles idées sur la base de solveurs performants (il y a même une catégorie dédiée à cela dans les compétitions SAT récentes, la Minisat/Glucose/CaDiCaL hack track).

Parmi les autres améliorations testées pour améliorer l'architecture CDCL, il y a l'ajout de différents types de raisonnement comme par exemple la possibilité de détecter certains types de contraintes (e.g. des contraintes de cardinalité [BLLM14, ABC+19]).

Une autre amélioration de l'algorithme CDCL repose sur l'exploitation des symétries [Sak21] présentes dans la formule considérée :

Définition 12 (Symétrie)**.** *Soit une formule CNF ϕ et P l'ensemble des littéraux présents dans ϕ. Une permutation σ définie sur P ($\sigma : P \to P$) est une* symétrie *de ϕ si elle satisfait les conditions suivantes :*

— $\forall l \in P, \sigma(\neg l) = \neg\sigma(l)$
— $\sigma(\phi) = \phi$

En d'autres termes, une symétrie est une permutation des littéraux de la formule qui conserve la structure de la formule, et donc qui conserve les potentiels modèles existants. Il existe plusieurs manières différentes de briser les symétries d'une formule. Une possibilité est de compiler avant le début de la recherche toutes les symétries globales présentes dans la formule considérée et d'ajouter à la formule des clauses appelées *sbp* (pour *Symmetry Breaking Predicate*) [CGLR96]. Cette méthode a notamment été implémentée dans l'outil SHATTER [ASM06], puis améliorée dans BREAKID [DBBD16]. Il existe également des méthodes dynamiques pour briser les symétries. Des programmes tels que CDCLSym [MBCK18] ou encore CosySEL [SBDD23] vont compiler avant le début de la recherche toutes les symétries globales présentes dans la formule initiale et seront ensuite capables de les reconnaître à la volée au cours de la recherche. En définissant une notion d'ordre sur les affectations et en attribuant des états aux symétries, il est possible de savoir à tout moment de la recherche si une des symétries détectées permet de retrouver une affectation plus petite (au sens de la notion d'ordre définie) que celle actuelle. Si c'est le cas, alors nous pouvons couper la branche sur laquelle nous nous trouvons et revenir en arrière en brisant la symétrie grâce à un *sbp*. Encore une fois, cela est intéressant pour des instances incohérentes mais cela peut aussi retarder la découverte d'un modèle pour une instance cohérente.

1.2.5 Autres informations fournies par les solveurs SAT

Une des fonctionnalités majeures ajoutées aux solveurs SAT est la possibilité de générer un certificat UNSAT. Le principe de cette approche est d'enregistrer les étapes importantes d'un solveur et ensuite d'appeler un programme externe que l'on nomme *vérificateur* qui peut être certifié et dont le but est de vérifier que les étapes enregistrées sont correctes et que le solveur avait bien le droit de réaliser ces opérations. Si aucune étape ne pose problème, on peut alors considérer que la recherche du solveur s'est réalisée correctement et l'instance a été prouvée incohérente. Actuellement, le format de certificats le plus connu est le format *DRAT* (pour *Deletion Resolution Asymmetric Tautology*) [WHJ14]. C'est aussi le format qui est utilisé lors des compétitions SAT pour vérifier les résultats des solveurs participants sur des instances incohérentes [FHI+21]. Avant l'utilisation de certificats, lors des compétitions, on pouvait juste

vérifier que le modèle trouvé par le solveur était correct ou bien que les résultats étaient cohérents entre plusieurs solveurs. L'utilisation de ces certificats a donc été une avancée notable lors des compétitions puisque cela a permis de vérifier de manière formelle et automatisée les résultats des solveurs.

À noter

Certifier le code d'un solveur SAT moderne tout en préservant son efficacité est une tâche très difficile car les solveurs SAT utilisent nombre d'optimisations de bas niveau (voir par exemple le solveur SAT certifié à l'aide d'Isabelle IsaSAT [@IsaSAT]). Il est cependant possible de vérifier les réponses fournies par ces derniers à l'aide d'outils certifiés. En effet, pour une instance cohérente, nous pouvons vérifier que le modèle obtenu satisfait bien toutes les clauses de la formule considérée, et pour une instance incohérente, nous pouvons générer un certificat et utiliser un outil externe et certifié pour vérifier ce certificat. Ainsi, il est possible d'avoir confiance dans la réponse des solveurs SAT même si leur code n'est pas certifié.

Pour générer une preuve au format DRAT, un solveur CDCL peut simplement enregistrer toutes les modifications effectuées à la base de clauses. Chaque fois qu'une clause est apprise ou supprimée par le solveur, cette opération est enregistrée dans la preuve. Les clauses supprimées sont précédées de la lettre `d` afin de les différencier des clauses apprises qui ne sont précédées d'aucune lettre. Étant donné que l'on génère des certificats pour des instances incohérentes, la dernière clause apprise doit logiquement être la clause vide. Une fois que le solveur a terminé sa recherche et que la preuve a été générée, on peut appeler un vérificateur externe qui va vérifier que la preuve est correcte et notamment que les différentes clauses apprises étaient bien impliquées par la formule considérée. La vérification d'une étape dans un certificat au format DRAT se base sur les propriétés suivantes :

Définition 13 (RUP)**.** *Soit ϕ une formule et C une clause. On dit que la propriété* RUP *(pour* Reverse Unit Propagation*) est vérifiée pour C s'il est possible de montrer que $\phi \wedge \neg C \vdash_{UP} \bot$ en utilisant uniquement la propagation unitaire (i.e. il est possible d'obtenir la clause vide en utilisant uniquement la propagation unitaire).*

Définition 14 (RAT)**.** *Soit ϕ une formule et $C \vee p$ une clause. On dit que la propriété* RAT *(pour* Resolution Asymmetric Tautology*) est vérifiée pour $C \vee p$ si pour chaque clause $(D \vee \neg p) \in \phi$, alors la propriété RUP est vérifiée pour la résolvante $C \vee D$ (i.e. on peut montrer avec la propagation unitaire que $C \vee D$ est impliquée par ϕ).*

En d'autres termes, pour que la propriété RUP soit vérifiée, il doit être possible de retrouver un conflit juste en propageant les littéraux opposés de ceux présents dans une clause donnée. Pour que la propriété RAT soit vérifiée, il faut que, pour une clause donnée, toutes les résolvantes pouvant être obtenues à partir des clauses de la formule considérée vérifient la propriété RUP. Ainsi, lors de la vérification d'un certificat au format DRAT, toutes les clauses ajoutées doivent vérifier l'une des deux propriétés soit RUP soit RAT. La figure 1.6 donne un exemple simple de preuve au format DRAT pour une formule donnée. Ici, nous pouvons facilement vérifier que la preuve générée est correcte. En effet, si l'on ajoute $\neg x_1$ ou x_2 à la formule considérée et que l'on effectue une propagation unitaire, nous trouvons un conflit dans les deux cas. Cela n'est pas visible sur cet exemple mais l'ordre des clauses présentes dans une preuve DRAT est important. En effet, lorsqu'une clause C est ajoutée dans la preuve par le solveur, les clauses précédemment

ajoutées étaient présentes dans le solveur (elles ont été apprises) et étaient donc disponibles lors de la dérivation de C. Par contre, il se peut très bien que toutes les clauses enregistrées au cours de la recherche ne soient pas utiles dans la dérivation de la clause vide. Pour pallier ce souci, il est possible, lors de l'étape de vérification, de parcourir en sens inverse la preuve générée afin de repérer les étapes indispensables dans la dérivation de la clause vide [HJW13]. Cette façon de procéder peut donc permettre de repérer des étapes inutiles et donc de réduire la taille de la preuve générée.

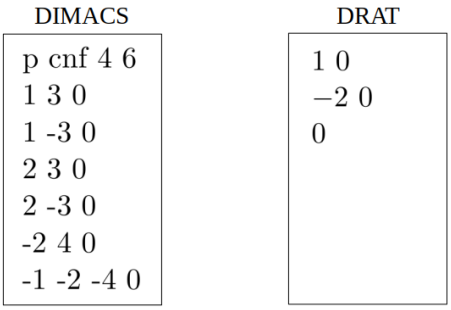


FIGURE 1.6 – Exemple de formule au format DIMACS et d'une preuve possible au format DRAT.

Par la suite, d'autres formats de certificats ont été proposés. Le dernier en date est le format *FRAT* [BCH22], qui permet d'enregistrer d'autres informations en plus de celles déjà présentes dans le format DRAT. La figure 1.7 donne un exemple de preuve au format FRAT pouvant être obtenu pour une instance incohérente donnée. Cet exemple est tiré de [BCH22]. Avec ce format, nous pouvons donner des informations telles que les clauses initiales de la formule (avec le préfixe `o`), celles présentes à la fin de la recherche (avec le préfixe `f`) ainsi que les clauses apprises par le solveur (avec le préfixe `a`). Il est toujours possible d'utiliser la lettre `d` pour indiquer des clauses qui ont été supprimées. De plus, lorsqu'une clause est apprise, il est possible d'indiquer quelles autres clauses ont permis de la dériver via le préfixe `l`. Le but de ce format est d'être plus flexible que le format DRAT et de pouvoir être étendu à d'autres systèmes de preuves. Par ailleurs, le fait de fournir plus d'informations dans le certificat peut permettre de faciliter la vérification.

Une autre fonctionnalité notable des solveurs SAT est la possibilité de générer un MUS [ZM03a, GMP06, GMP07, GMP08, LS08] pour une instance incohérente.

Définition 15 (MUS)**.** *Soit ϕ une formule incohérente ($\phi \models \bot$). $M \subseteq \phi$ est un* MUS *(pour* Minimal Unsatisfiable Subset*) si et seulement si $M \models \bot$ et $\forall M' \subset M$, $M' \not\models \bot$.*

En d'autres termes, un MUS correspond à un sous-ensemble minimal pour l'inclusion de clauses provenant de la formule initiale et qui est incohérent. Par conséquent, si l'on supprime une seule des clauses d'un MUS, nous devons donc retrouver un sous-ensemble de clauses qui est cohérent. Un MUS peut donc permettre de retrouver une source de l'incohérence d'une formule donnée. Au cours du temps, diverses méthodes ont été mises au point afin de faciliter l'extraction d'un MUS, comme par exemple des méthodes à base de rotation de modèle [BM11]. Une même formule peut admettre plusieurs MUS de tailles différentes. Il semble donc intéressant de retourner à l'utilisateur le plus petit MUS possible [LM04, OMA+04] mais cette opération est bien plus coûteuse que de trouver un seul MUS. Les MUS ne sont pas toujours adaptés pour fournir une explication de haut niveau. Par exemple, l'outil Alloy [2] ne génère pas forcément un

2. http ://alloytools.org.

FRAT

```
o 1                1 2 -3 0   |  f 1   1 2 -3 0
o 2               -1 -2 3 0   |  f 2  -2 -1 3 0
o 3                2 3 -4 0   |  f 3   2 3 -4 0
o 4               -2 -3 4 0   |  f 4  -2 -3 4 0
o 5              -1 -3 -4 0   |  f 5 -1 -3 -4 0
o 6                 1 3 4 0   |  f 6    1 3 4 0
o 7                -1 2 4 0   |  f 7   -1 2 4 0
o 8                1 -2 -4 0  |  f 8  1 -2 -4 0
a 9 -3 -4 0 1        5 1 8 0   |  f 9    -3 -4 0
a 10   -4 0 1     9 3 2 8 0    |  f 10      -4 0
a 11    3 0                    |  f 11       3 0
a 12   -2 0                    |  f 12      -2 0
a 13    1 0 1     12 11 1 0    |  f 13       1 0
a 14      0 1 13 12 10 7 0     |  f 14         0
```

FIGURE 1.7 – Exemple de preuve au format FRAT. Des espaces ont été ajoutés pour simplifier la lecture.

MUS de la formule UNSAT car il ne s'agit pas forcément d'une explication minimale de haut niveau dans ce contexte. À la place, il utilise une analyse spécifique de l'arbre de recherche UNSAT pour retrouver la source minimale d'incohérence de haut niveau d'une spécification incohérente [TCJ08]. Alloy est donc un outil de spécification formelle basé sur SAT mais qui a eu besoin de développer une notion d'explication propre à l'utilisateur qui n'est pas basée sur les MUS.

Au cours du temps, d'autres fonctionnalités ont été ajoutées aux solveurs SAT. Nous pouvons par exemple citer la possibilité d'utiliser des *sélecteurs*. Ce terme désigne des littéraux pouvant être ajoutés aux clauses et permettant facilement de les activer ou de les désactiver. C'est une manière de gérer l'incrémentalité, qui a été introduite par SATIRE [WKS01] et popularisée par MiniSat [ES03]. Enfin, des solveurs parallèles ont été proposés [HS18]. Parmi ceux-ci, nous pouvons citer par exemple la méthode *portfolio* qui utilise plusieurs solveurs différents en parallèle pour résoudre une même instance, ou l'approche *ManySat* [HJS09] qui utilise en parallèle différentes configurations d'un même solveur, avec éventuellement échange de clauses apprises pendant la recherche. Quand il faut obtenir un MUS ou un certificat dans ce cadre, nous pouvons le demander au solveur qui a résolu l'instance. Quand l'espace de recherche est découpé comme dans des approches type *Cube and Conquer* [HKWB11], les certificats d'incohérence peuvent être vérifiés de manière indépendantes. Travailler en parallèle avec une base de clauses apprises unique pour produire un certificat ou un MUS, comme dans Gimsatul [FB22], reste en pratique très difficile à réaliser.

1.3 Le problème #SAT

Une des variantes du problème SAT qui va nous intéresser dans ce mémoire est le problème *#SAT*, qui peut être défini de la façon suivante.

Définition 16 (Problème #SAT)**.** *Le* problème #SAT *correspond au problème suivant :*
- *Entrée : Une formule CNF ϕ construite sur n variables propositionnelles*
- *Question : Combien d'affectations différentes des n variables satisfont la formule ϕ ?*

Un programme capable de résoudre ce problème est appelé un *compteur de modèles*. De même que pour les solveurs SAT, il existe aussi une compétition pour les compteurs de modèles [FHH21]. Comme compteurs de modèles exacts notables, nous pouvons par exemple citer Cachet [SBB+04], SharpSAT [Thu06], SharpSAT-TD [KJ21] ou encore D4 [LM17]. Cette fois-ci, il ne suffit plus de chercher à trouver un modèle à une formule mais de compter combien de modèles différents cette formule peut admettre. Il n'est donc plus possible d'arrêter la recherche dès que l'on a trouvé un modèle. Il faut cette fois-ci explorer tout l'espace de recherche afin de trouver tous les modèles satisfaisant la formule considérée.

Une des techniques importantes implémentées dans les compteurs de modèles est l'utilisation d'un système de cache. Dès lors qu'une sous-formule a été complètement explorée et son nombre de modèles déterminé, celle-ci va être enregistrée à l'intérieur d'un cache avec son nombre de modèles, ce que nous appelons une *entrée*. Si à un autre moment de la recherche, la sous-formule courante est identique à une entrée du cache (ce qui est appelé un *hit*), alors nous pouvons directement utiliser le nombre de modèles enregistré et revenir en arrière. Ceci inclut le cas des sous-formules incohérentes, pour lesquelles le nombre de modèles enregistré est égal à 0. Pour utiliser ce système, il faut avoir une représentation des sous-formules pour pouvoir les sauvegarder. C'est le cas du compteur de modèles Cachet [SBB+04], qui utilise une représentation mettant bout à bout les clauses résiduelles de la formule, ou encore du compteur de modèles SharpSAT [Thu06], qui utilise une combinaison des indices de variables et de clauses restantes.

Exemple 9. *Soit* $\phi = (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2 \vee \neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2)$ *une formule CNF. Dans ce cas, Cachet va représenter* ϕ *par* $(1, 2, 3, 0, 1, -2, 0, -1, -2, -3, 0, -2, 3, 0)$ *(le* 0 *représente ici une fin de clause). Ici, nous partons d'une représentation DIMACS de la formule et nous trions les littéraux dans chaque clause et ensuite les clauses de la formule. Les littéraux sont triés par ordre de leur valeur absolue et pour une variable donnée, le littéral positif vient avant le littéral négatif. Si nous supposons que les clauses de* ϕ *sont numérotées de* 1 *à* 4*, alors SharpSAT va représenter* ϕ *par* $((1, 2, 3), (1, 2, 3, 4))$*. Cette fois, nous avons d'abord les indices de variables présentes dans* ϕ *(i.e.* $(1, 2, 3)$*) et ensuite les indices des clauses présentes (i.e.* $(1, 2, 3, 4)$*).*

À noter

La représentation de SharpSAT est plus compacte mais on perd des informations sur la sous-formule enregistrée. Ainsi, elle ne permet pas de reconnaître des sous-formules identiques si celles-ci sont basées sur des clauses différentes.

Enfin, afin de faire en sorte que le cache ne dépasse pas une certaine taille, certaines entrées considérées inutiles vont régulièrement être supprimées.

Concernant l'heuristique de choix de variables, nous pouvons par exemple citer l'utilisation dans Cachet de l'heuristique *VSADS* (pour *Variable State Aware Decaying Sum*) [SBK05] qui calcule le score de chaque variable grâce à une combinaison linéaire entre son score obtenu avec l'heuristique VSIDS et son nombre d'apparitions dans la sous-formule courante. Les compteurs de modèles utilisent également une étape de *décomposition* qui sépare la sous-formule courante en plusieurs sous-formules indépendantes qu'il faut alors résoudre. Si la formule complète est cohérente, alors toutes ces sous-formules le sont également et il faut donc calculer un nombre de modèles pour chacune d'entre elles. Le cache peut éventuellement permettre de trouver le nombre de modèles de certaines sous-formules. Le nombre de modèles de la formule courante peut être déterminé en multipliant entre eux les nombres de modèles des différentes sous-formules. En

revanche, si l'une d'entre elles est incohérente, alors nous savons que le formule complète est incohérente et nous pouvons arrêter la recherche sur cette branche.

Chapitre 2

Notion d'explication

2.1 Qu'est-ce qu'une explication ?

De nos jours, il y a un besoin toujours plus important que les programmes informatiques soient capables de justifier leurs résultats ou décisions, de la même manière qu'un être humain peut être amené à un moment donné à expliquer certaines de ses actions. Ce besoin a notamment été exprimé dans un certain nombre de textes de lois et rapports, comme indiqué en introduction. Il concerne donc tous types de solveurs, que ce soit les solveurs SAT, les solveurs pseudo-booléens ou autres. Par contre, actuellement, il n'existe pas véritablement de consensus sur ce qui peut constituer une explication pour un solveur SAT. Nous pouvons cependant remarquer qu'une explication peut souvent être vue comme un ensemble d'informations données à une personne pour lui permettre de mieux comprendre le fonctionnement d'un programme et le raisonnement utilisé pour arriver au résultat.

2.2 Propriétés d'une explication

Une des premières choses que l'on peut noter concernant les explications est que celles-ci peuvent varier en fonction de divers paramètres. En effet, en fonction du niveau auquel on se situe (au niveau des clauses ou de contraintes de haut niveau par exemple) ou encore du fondement même de l'explication (logique ou statistique par exemple), l'explication générée ne va pas être la même. Une autre source de variation va provenir de la personne à laquelle l'explication est destinée. En effet, l'explication peut être plus ou moins technique ou précise suivant le rôle dans l'application de la personne à qui on s'adresse : que ce soit une personne qui a contribué à modéliser les données, une personne qui a une responsabilité d'administration de tout ou partie de l'application ou une personne qui est simple utilisateur. Ces divers utilisateurs n'ont pas les mêmes connaissances et n'ont donc pas non plus les mêmes exigences en terme d'explication. C'est le point de vue choisi, par exemple, dans le logiciel IBEX (pour *Interactive Black-Box Explanations*) [HM20] qui intègre divers profils d'utilisateur et adapte l'explication générée pour des solveurs de contraintes (CSP) en fonction du profil sélectionné. Si l'explication générée par cet outil ne convient pas à l'utilisateur, ce dernier peut modifier certains paramètres sur son profil et demander à ce que IBEX génère une nouvelle explication.

Par ailleurs, le logiciel considéré peut être vu soit comme une boîte transparente, et dans ce cas il faut expliquer le comportement réel du logiciel, soit comme une boîte noire, et dans cet autre cas il faut en général utiliser une autre représentation se rapprochant le plus du logiciel réel ou qui produit des résultats similaires.

Du côté des sciences sociales, Tim Miller a proposé plusieurs propriétés des explications [Mil17]. Par exemple, une demande d'explication peut provenir d'un désir de vérification du comportement d'un programme ou bien lorsque le résultat n'est pas celui attendu (on parle alors d'explication *contrastive*). Dans ce dernier cas, l'utilisateur va chercher à trouver ce qui a provoqué ce résultat plutôt que celui souhaité, alors que s'il avait eu directement le résultat attendu, il s'en serait probablement contenté. Ensuite, parmi toutes les causes possibles d'un évènement, l'utilisateur n'est en général intéressé que par une partie d'entre elles (il ne s'agit d'ailleurs pas forcément des causes les plus probables). Cela peut indiquer qu'il n'est pas forcément nécessaire de donner une explication complète à l'utilisateur pour le satisfaire et que certaines simplifications peuvent être acceptées. Enfin, l'explication peut avoir une dimension sociale lorsqu'elle est le résultat d'un échange de connaissances entre un utilisateur et le système. L'utilisateur n'est pas forcément au courant de toutes les contraintes du problème ou de toutes les conséquences d'une action ou d'une décision. L'utilisateur et le système n'ont donc pas forcément la même modélisation du problème. Le but de cette conversation pourrait donc être de réconcilier ces deux visions.

2.3 Expliquer les résultats des solveurs SAT

Nous allons maintenant revenir sur chacun des résultats possibles d'un solveur SAT et étudier les explications qu'il est envisageable de donner.

2.3.1 Cas des instances cohérentes

Explications basées sur la logique

Lorsque le solveur a trouvé une solution à une instance cohérente, l'utilisateur peut se poser plusieurs types de questions concernant le modèle retourné. Par exemple, l'utilisateur peut demander : « Pourquoi ce modèle est-il une solution ? ». Dans ce cas, une explication simple consiste à indiquer que toutes les contraintes exprimées dans le problème donné ont été satisfaites. En effet, on peut facilement retrouver pour chaque clause de la formule CNF donnée en entrée un littéral de la solution qui satisfait cette clause. On peut aussi essayer de réduire le modèle retourné en calculant un impliquant premier afin d'éventuellement supprimer des littéraux inutiles à la satisfaction de la formule. Normalement, le fait de satisfaire toutes les contraintes exprimées doit aussi satisfaire l'utilisateur. Si la solution donnée par le solveur ne convient pas à l'utilisateur, cela signifie que ce dernier doit avoir des préférences qui n'ont pas été exprimées sous forme de contrainte. Dans ce cas, il peut devenir nécessaire de réviser le problème pour prendre en compte ces préférences. Ce problème n'a pas été étudié au cours de cette thèse. Une autre question possible est : « Pourquoi cette solution et pas une autre ? ». Cette fois-ci, nous pouvons éventuellement donner une justification logique aux littéraux impliqués par la formule, ce que l'on nomme son *backbone* [JLM15, BFW23]. Ce dernier désigne l'ensemble des littéraux présents dans tous les modèles possibles de la formule considérée. Dans ce cas, nous pouvons montrer que l'ajout de la négation de l'un de ces littéraux mène à une incohérence et nous pouvons donc nous ramener au cas des instances incohérentes, que nous détaillerons plus tard. Par contre, il n'est pas possible en général de donner de justification logique aux décisions du solveur. En effet, celles-ci se basent uniquement sur l'heuristique du solveur et ne peuvent être vues que comme des affectations arbitraires ayant pour but de faire avancer la recherche quand celle-ci est bloquée (i.e. quand elle ne peut plus avancer grâce à des propagations unitaires).

Nous pouvons aussi citer le cas des explications pas-à-pas [BGG21] qui permet de découper la résolution d'un problème de contraintes en plusieurs étapes. Cette technique a notamment été utilisée sur des grilles logiques.

Informations venant du solveur

Une autre approche que l'on peut aussi avoir avec les instances cohérentes consiste à récupérer certaines informations du solveur et notamment indiquer ce qui a permis d'effectuer certaines affectations. Par exemple, nous avons ajouté dans le solveur Sat4j [LP10] l'option « -Dcolor » qui permet d'ajouter des informations sur l'origine de l'affectation des littéraux d'un modèle par le solveur. Cela peut passer par une coloration comme celle présentée à la figure 2.1. Les premières catégories représentées sont classiques :

— UNASSIGNED (Gris) : les variables qui n'ont pas été affectées. Cette catégorie est toujours vide pour un solveur CDCL. Elle est utile si on calcule un impliquant premier.
— DECIDED (Vert) : les variables dont la valeur a été attribuée uniquement grâce à l'heuristique du solveur.
— PROPAGATED_ORIGINAL (Rouge) : les variables dont la valeur a été uniquement propagée depuis une clause initiale.
— PROPAGATED_LEARNED (Bleu) : les variables dont la valeur a été uniquement propagée depuis une clause apprise.

Lorsque le solveur parcourt l'espace de recherche, il se peut que par propagation il affecte un littéral qui a déjà été décidé par l'heuristique. Ce littéral est donc déjà présent dans l'affectation courante. Les catégories suivantes prennent en compte ce cas :

— DECIDED_PROPAGATED (Violet) : les variables dont valeur a été décidée par l'heuristique puis propagée par une clause initiale à un autre niveau de décision.
— DEDIDED_PROPAGATED_LEARNED (Cyan) : les variables dont valeur a été décidée par l'heuristique puis propagée par une clause apprise à un autre niveau de décision
— DECIDED_CYCLE (Fond vert) : les variables dont la valeur a été décidée par l'heuristique puis propagée par une clause initiale ou apprise au même niveau de décision.

De cette manière, nous pouvons visualiser pour chacun des littéraux ce qui a permis de l'affecter et notamment de savoir quel mécanisme interne du solveur (décision arbitraire ou propagation depuis une clause) a permis d'attribuer une valeur.

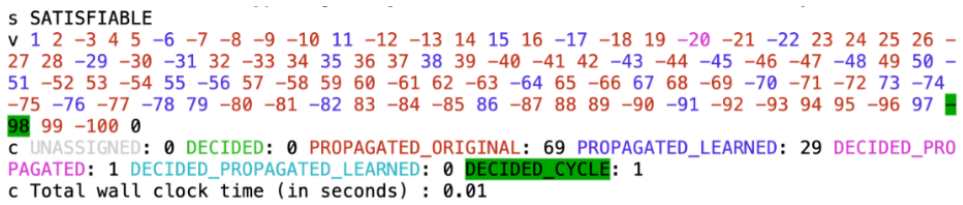


FIGURE 2.1 – Exemple d'informations sur l'origine des littéraux d'un modèle dans Sat4j.

Dans l'exemple 2.1 qui correspond au problème `aim-100-1_6-yes1-1.cnf`, on peut noter que la solution proposée découle du choix de deux affectations, $\neg x_{20}$ et $\neg x_{98}$. La majorité des autres affectations découlent de la propagation des clauses initiales (69/100). Les propagations sur les clauses apprises par le solveur (29/100), inconnues de l'utilisateur, complètent la solution proposée. Il s'agit de propagations qu'il est difficile de justifier à partir du problème original. Par ailleurs, même si nous pouvons déterminer quelles décisions ont permis d'obtenir un modèle pour une instance, nous ne pouvons pas véritablement en tirer d'information

supplémentaire.

Exemple 10. *L'exemple suivant a pour but d'illustrer l'impact de l'heuristique dans la recherche du solveur. Ici, nous considérons un ensemble de clauses* $C = \{a_i \vee b_i, a_i \vee c_i, \neg b_i \vee \neg c_i \vee d_i, \neg d_i \vee \neg a_{i+1}\}$ *pour* $i = 1 \cdots n-1$. *Si l'on considère maintenant un solveur dont l'heuristique va d'abord rendre vrai le littéral* $\neg a_1$, *celui-ci va ensuite propager les valeurs* b_1, c_1 *et ensuite* d_1. *Avec cette dernière affectation, nous pouvons en déduire* $\neg a_2$. *De la même manière, nous allons propager* $\neg a_3 \cdots \neg a_n$. *Ainsi, avec juste une seule décision, le solveur va être capable d'affecter toutes les autres variables par propagation. Considérons maintenant un autre solveur qui va prendre des décisions dans l'ordre inverse. Ce deuxième solveur va alors prendre successivement les décisions* $\neg a_n, \neg a_{n-1}, \cdots \neg a_1$. *Chaque* $\neg a_i$ *va permettre d'affecter les autres variables de même niveau (i.e. qui ont le même indice* i *que la dernière décision). Ainsi, cette fois-ci, il aura fallu* n *décisions pour attribuer une valeur à chacune des variables. Ces deux recherches sont représentées à la figure 2.2. Au final, les solutions trouvées sont identiques mais les arbres de recherches ont une taille bien différente.*

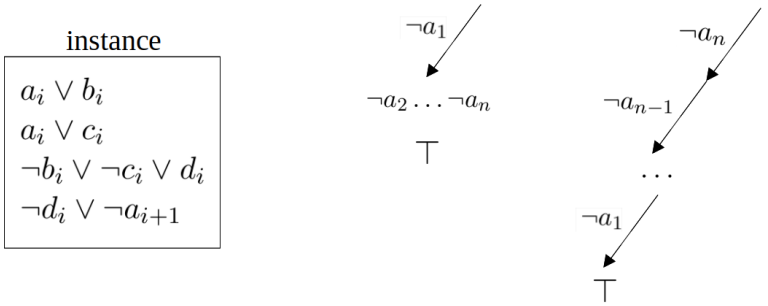


FIGURE 2.2 – Comparaison des arbres de recherche de deux solveurs différents. Les propagations concernant les variables b, c et d ont été omises.

À noter

Cet exemple permet d'illustrer que les décisions sont un artefact du solveur et non de la formule elle-même.

Ce dernier point vient du fait que le système d'inférence utilisé dans les solveurs SAT est la propagation unitaire. En effet, au cours de la recherche, un certain nombre de littéraux sont impliqués par la formule courante et la propagation unitaire n'est capable d'en détecter qu'une partie. Si un solveur disposait d'un système d'inférence plus puissant et capable de détecter tous ces littéraux impliqués localement, alors les branches présentes dans l'arbre de recherche développé par un tel solveur correspondraient chacune à un modèle. Si un tel solveur doit prendre une décision, cela signifie que plusieurs modèles sont disponibles et qu'il peut faire un choix entre ceux-ci.

Informations statistiques globales sur les modèles

On peut aussi imaginer donner une explication autre que logique à l'utilisateur. Nous pouvons utiliser par exemple des arguments tels que la probabilité pour qu'une variable soit affectée à vrai par exemple. Pour pouvoir faire cela, nous devons commencer par récupérer un ensemble

de modèles de l'instance considérée car nous ne pouvons plus travailler avec un seul modèle. Par exemple, supposons que nous obtenons les 5 modèles présents dans la table 2.3. À partir de là, nous pouvons effectuer divers calculs sur ces modèles et notamment calculer les probabilités des affectations des diverses variables. Un exemple de ce qui peut être obtenu pour notre exemple peut être visualisé à la table 2.1. Ainsi, nous pouvons observer par exemple les variables qui ont toujours obtenu la même affectation (des littéraux impliqués faisant partie du *backbone* de la formule). Il est également à noter que, même si c'est le cas ici, utiliser l'affectation la plus probable pour chacune des variables ne nous permet pas forcément de retrouver un modèle. Même si ces statistiques donnent une information supplémentaire à l'utilisateur, elles ne constituent pas une explication.

a1	a2	a3	a4	a5	a6	b1	b2	b3	b4	b5	b6
0	1	1	0	0	0	1	0	0	0	1	0
0	0	1	0	1	0	1	0	0	0	0	1
0	1	1	0	0	0	1	0	0	0	0	1
0	1	1	0	0	0	0	0	0	0	1	1
0	1	0	0	1	0	1	0	0	0	0	1

FIGURE 2.3 – Les cinq modèles d'une formule sous forme de tableau.

TABLE 2.1 – Probabilités d'affectation positive ou négative pour les variables de l'exemple 2.3.

Variable	Affectation positive	Affectation négative	Plus probable
a_1	0 %	100 %	$\neg a_1$
a_2	80 %	20 %	a_2
a_3	80 %	20 %	a_3
a_4	0 %	100 %	$\neg a_4$
a_5	40 %	60 %	$\neg a_5$
a_6	0 %	100 %	$\neg a_6$
b_1	80 %	20 %	b_1
b_2	0 %	100 %	$\neg b_2$
b_3	0 %	100 %	$\neg b_3$
b_4	0 %	100 %	$\neg b_4$
b_5	40 %	60 %	$\neg b_5$
b_6	80 %	20 %	b_6

2.3.2 Cas des instances incohérentes

Le cas des instances incohérentes est plus complexe car, cette fois-ci, il faut justifier l'impossibilité de trouver une solution, c'est-à-dire d'expliquer qu'il n'est pas possible de trouver une affectation des variables qui satisfasse toutes les clauses de l'instance. Dans cette partie, nous proposons un tour d'horizon des travaux qui se rapprochent de notre problématique.

Tout d'abord, nous pouvons par exemple citer l'utilisation de certificats. Cependant, comme on sait que le problème SAT est NP-complet, nous pouvons en déduire que dans le pire des cas, nous allons rencontrer un nombre exponentiel de conflits, et donc de clauses à apprendre, dans la recherche du solveur. Il est donc possible pour certaines instances de se retrouver avec un certificat contenant un nombre exponentiel d'étapes à vérifier, ce qui nous donne un fichier qui

n'est pas de taille assimilable pour un être humain et ce même avec l'étape de simplification. Par exemple, la preuve d'incohérence générée pour le problème des triplets Pythagoriciens [HK17] dépasse les 200TB. Même s'il est toujours possible d'automatiser l'étape de vérification, il est évident que de tels fichiers ne peuvent pas être parcourus par un être humain en raison de leur longueur. Certains systèmes de preuve comme les plans-coupes ou encore la Résolution Étendue peuvent permettre, en théorie, d'obtenir des preuves plus courtes [GN21]. De plus, même si les clauses exprimées initialement peuvent avoir un sens pour l'utilisateur, ce n'est pas forcément le cas des clauses apprises car ces dernières sont obtenues par résolution de plusieurs autres clauses. Ainsi, on peut obtenir des clauses apprises difficilement interprétables pour un être humain et qui pourtant sont indispensables pour prouver l'incohérence de la formule.

Dans le cadre de l'explication d'instances incohérentes, nous pouvons également imaginer donner un MUS à l'utilisateur. Un MUS permet de retrouver une source d'incohérence de l'instance considérée. Si le but de l'utilisateur était d'avoir une instance cohérente, fournir un MUS peut permettre d'indiquer sur quelle partie du problème il faut se focaliser. Cependant, il n'est pas toujours possible de supprimer des clauses lors de la génération d'un MUS et il est donc possible que l'on se retrouve avec un MUS identique à la formule considérée. Dans ce cas, nous ne pouvons pas tirer d'information supplémentaire du MUS que nous venons de récupérer.

2.4 Contexte de la thèse

Le travail réalisé au cours de cette thèse s'inscrit dans le cadre de l'IA explicable et nous nous sommes tout particulièrement intéressés à l'explication des résultats donnés par les solveurs SAT. Générer des explications pour des instances cohérentes est soit facile (montrer la satisfaction de toutes les clauses, obtenir un impliquant premier), soit impossible à expliquer d'un point de vue logique quand ce sont des artéfacts du solveur (décisions), soit se ramène à l'explication sur des instances incohérentes (prouver l'implication de formule). De ce fait, nous nous sommes focalisés sur le cadre des instances incohérentes. Afin d'expliquer l'incohérence d'une formule, nous avons adopté le cadre suivant : si nous pouvons compresser de manière significative un arbre de recherche (ou même un certificat d'incohérence), nous pouvons alors éventuellement obtenir quelque chose qui soit de taille plus assimilable pour un utilisateur et qui puisse servir d'explication. Pour effectuer cette opération, on peut imaginer plusieurs possibilités, comme par exemple supprimer des décisions ou propagations inutiles, réordonner certains nœuds de l'arbre de recherche, ou encore reconnaître des motifs récurrents au cours de la recherche. Par la suite, nous nous sommes focalisés sur cette dernière possibilité, qui nous semblait être la plus prometteuse et la plus à même de réaliser des compressions significatives. En effet, si nous disposons d'un arbre de recherche UNSAT de taille exponentielle, certaines clauses servent un nombre exponentiel de fois de conflit. De ce fait, on peut se demander si un même sous-ensemble incohérent de clauses peut être exploré de manière répétée.

Pour ce faire, nous avons étudié principalement deux techniques et que nous allons développer dans la suite de cette thèse. La première consiste à reconnaître dans l'arbre de recherche des formules UNSAT classiques comme des problèmes de pigeons, qui sont des problèmes simples à expliquer à l'utilisateur. Si une formule UNSAT est reconnue, alors nous pouvons directement arrêter la recherche sur cette branche et revenir en arrière. Cette technique peut notamment permettre d'associer une explication à chaque motif et il est aussi possible d'expliquer les motifs reconnus individuellement et indépendamment. La deuxième technique consiste cette fois-ci à reconnaître des sous-formules déjà prouvées UNSAT à l'intérieur de l'arbre. Par exemple, si à un moment donné de la recherche, nous reconnaissons une sous-formule que nous avons pré-

cédemment explorée et déterminée comme étant UNSAT alors nous pouvons conclure que la sous-formule courante est aussi incohérente. Nous pouvons alors arrêter la recherche à ce niveau et faire référence à la sous-formule reconnue. Le comportement de cette méthode est représenté à la figure 2.4. Ce principe est notamment utilisé dans les compteurs de modèles qui prennent aussi en compte le cas des instances cohérentes. Dans les deux cas, l'objectif est, à partir d'un arbre de recherche exponentiel, d'obtenir un arbre qui soit le plus compressé possible et dont certaines branches sont étiquetées soit par un problème UNSAT connu, soit par une sous-formule que nous avons précédemment explorée. Dans la suite de cette thèse, nous allons revenir sur chacune de ces deux techniques en présentant d'abord une approche sémantique de détection de problèmes de pigeons et ensuite une approche basée sur la détection syntaxique de formules stockées dans un système de cache. Un point important à noter est que, au vu de la complexité des approches considérées, nous mettons l'accent avant tout sur la possibilité de générer une explication plutôt que sur le temps nécessaire pour la générer. De ce fait, nous n'excluons pas l'utilisation de techniques coûteuses, comme les oracles NP, tant que celles-ci nous permettent d'obtenir une bonne compression.

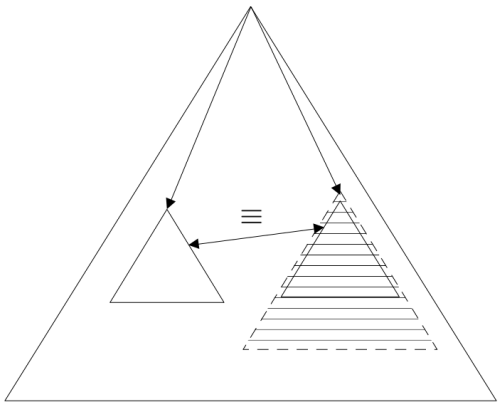


FIGURE 2.4 – Illustration de la méthode de reconnaissance de sous-formules déjà prouvées UNSAT. Quand on reconnaît une sous-formule UNSAT (à gauche sur le schéma) dans la sous-formule courante (à droite sur le schéma), on peut couper la branche correspondante.

Les approches proposées peuvent s'effectuer après la fin de la recherche ou bien pendant la recherche elle-même. L'avantage d'effectuer certains traitements après la recherche fait que nous pouvons partir d'une preuve ou d'un arbre stocké sous une forme ou une autre sans avoir à se soucier de certains aspects comme la gestion des retours en arrière. De plus, de cette manière, nous pouvons aussi imaginer nous affranchir de la chronologie de la recherche qui a été effectuée. En effet, comme nous disposons de l'arbre de recherche dans son intégralité, nous pouvons l'explorer comme nous le souhaitons. Cependant, appliquer les méthodes précédemment présentées au cours de la recherche présentent éventuellement l'avantage d'éviter l'exploration de certains sous-arbres potentiellement coûteux à explorer et donc d'éviter des calculs inutiles.

Deuxième partie

Contributions

Chapitre 3

Détection sémantique de problèmes de pigeons

3.1 Le problème des pigeons

Les problèmes de pigeons sont des problèmes incohérents classiques connus pour être difficiles pour les solveurs SAT et contenant de nombreuses symétries [Hak85]. De plus, les problèmes de pigeons sont représentatifs de différents problèmes d'appariement et nous pouvons donc espérer retrouver ce genre de problème dans différentes instances du problème SAT. Pour un problème de pigeons de taille n, que l'on notera PHP_n, le but est d'associer $n+1$ pigeons à n pigeonniers sachant que chaque pigeon doit être associé à un pigeonnier (contrainte *at least*) et qu'un pigeonnier ne peut pas accueillir plus d'un pigeon (contrainte *at most*). Une explication simple de l'incohérence d'un problème de pigeon consiste à indiquer que le nombre de pigeonniers disponibles est plus petit que le nombre de pigeons à placer. Une telle explication est facile à comprendre pour l'utilisateur. Cependant, les solveurs SAT ne peuvent pas fournir ce genre d'explication puisqu'il n'est pas possible de faire passer ces informations dans la modélisation du problème et, de plus, un solveur SAT n'est pas capable d'utiliser des arguments basés sur du comptage (même si il est possible d'encoder des problèmes de comptage). En pratique, pour pouvoir résoudre un problème de pigeons donné, un solveur SAT doit développer un arbre de recherche de taille exponentielle pour prouver l'incohérence de la formule [Hak85]. Un encodage à partir de clauses peut être réalisé de la manière suivante. Tout d'abord, 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. Ainsi, la contrainte qu'un pigeon doit être associé à un pigeonnier peut être encodée par un ensemble de $n+1$ clauses (une clause par pigeon) : $\phi_{1,n} = \bigwedge_{1 \leq i \leq n+1}(x_{i,1} \vee \dots \vee x_{i,n})$. Pour encoder le fait qu'un pigeonnier ne peut pas accueillir plus d'un pigeon, on peut créer pour chaque pigeonnier toutes les exclusions mutuelles possibles entre deux pigeons différents : $\phi_{2,n} = \bigwedge_{1 \leq i < j \leq n+1} \bigwedge_{1 \leq k \leq n}(\neg x_{i,k} \vee \neg x_{j,k})$. De cette manière, un problème de pigeons de taille n peut être défini comme $PHP_n = \phi_{1,n} \wedge \phi_{2,n}$.

Dans notre cas, nous souhaitons détecter ce type de problèmes de manière sémantique. Ici, plutôt que d'essayer de reconnaître toutes les clauses présentes dans un problème de pigeons (ce qui correspondrait à une détection syntaxique), nous voulons passer par la détection sémantique de certaines contraintes de cardinalité. Il est possible de voir chaque clause de $\phi_{1,n}$ comme une contrainte de cardinalité *at least 1*. Chaque clause $x_{i,1} \vee \dots \vee x_{i,n}$ peut être réécrite sous la forme $x_{i,1} + \dots + x_{i,n} \geq 1$. Concernant les clauses de $\phi_{2,n}$, nous pouvons aussi encoder le fait qu'un pigeonnier ne peut pas accueillir plus d'un pigeon par l'intermédiaire d'une contrainte *at most*

1. En effet, pour un pigeonnier k donné, l'ensemble des exclusions mutuelles $\bigwedge_{1\leq i<j\leq n+1}(\neg x_{i,k} \vee \neg x_{j,k})$ peut se réécrire en la contrainte $x_{1,k}+\cdots+x_{n+1,k} \leq 1$. La figure 3.1 donne une visualisation de ces contraintes.

Ces dernières contraintes peuvent être détectées sémantiquement grâce à la propagation unitaire. Dans [BLLM14], le but est de détecter des contraintes de cardinalité de taille quelconque dans une CNF pour remplacer les clauses correspondantes dans la formule, afin d'appliquer un solveur Pseudo-Booléen basé sur le système de preuve des plans coupes. Pour ce faire, les auteurs partent d'une contrainte de cardinalité obtenue à partir d'une clause de la formule, récupèrent les littéraux propagés à partir de chacun des littéraux de cette contrainte et utilisent les littéraux communs de ces propagations pour étendre la contrainte de cardinalité considérée. Il semble donc intéressant de partir en priorité de clauses binaires et d'essayer d'étendre les contraintes de cardinalité correspondantes. Considérons par exemple la formule $\phi = \{\neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_2 \vee \neg x_3\}$. Ici, la clause $\neg x_1 \vee \neg x_2$ peut se réécrire sous la forme $x_1 + x_2 \leq 1$. Avec cette contrainte, les auteurs effectuent une propagation unitaire depuis x_1 et une autre depuis x_2. Dans les deux cas, le littéral $\neg x_3$ est propagé. La contrainte de cardinalité considérée peut donc être étendue en $x_1 + x_2 + x_3 \leq 1$.

Dans notre cas, nous souhaitons détecter de manière syntaxique un ensemble de contraintes *at least 1* (correspondant à des clauses) et de manière sémantique un ensemble de contraintes *at most 1* (correspondant à des exclusions mutuelles). Un même littéral ne peut pas être présent dans plusieurs contraintes *at most 1* différentes et ces contraintes *at most 1* doivent toutes avoir la même taille. Chaque fois qu'une variable présente dans une contrainte *at most* est satisfaite, alors toutes les variables présentes dans cette même contrainte vont être négativement propagées. L'avantage de cette méthode est qu'une exclusion mutuelle peut être exprimée de différentes manière et sera quand même détectée. Le but ici va donc être de trouver un ensemble de clauses de la formule courante contenant les exclusions mutuelles nécessaires.

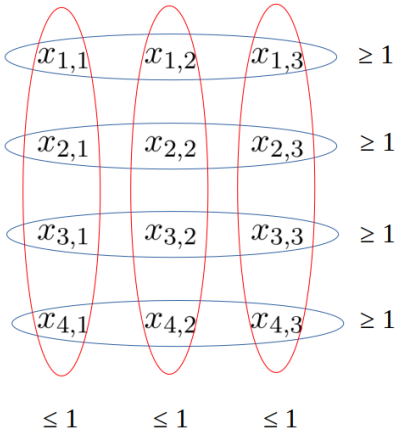


FIGURE 3.1 – Contraintes de cardinalité pour le problème PHP_3.

3.2 Détecter des problèmes de pigeons purs

Pour cette partie, nous allons émettre certaines hypothèses lorsque nous essayons de détecter des problèmes de pigeons. Tout d'abord, nous considérons que nous recherchons des pigeons purs, c'est-à-dire que toutes les clauses correspondant aux contraintes *at least 1* et *at most 1* nécessaires

à l'apparition d'un problème de pigeons sont présentes dans la formule courante.

L'idée de l'approche que nous allons présenter consiste à récupérer de manière syntaxique toutes les clauses *at least 1* d'un problème de pigeons tout en vérifiant de manière sémantique que toutes les exclusions mutuelles directes ou indirectes (correspondant aux contraintes *at most 1*) sont bien présentes dans les clauses que nous avons sélectionnées. Pour ce faire, nous allons sélectionner une clause de départ en supposant qu'elle fait partie d'un problème de pigeon, et nous allons essayer de voir quelles autres clauses du problème pourraient faire partie d'un problème de pigeons. Ces clauses doivent avoir des littéraux en exclusion mutuelle avec chacun des littéraux de la clause de départ et aussi en exclusion mutuelle avec chacun des littéraux des clauses déjà ajoutées au problème de pigeons. Nous construisons un ensemble de clauses candidates à l'aide des propagations des littéraux de la clause de départ, ce qui permet de retrouver les littéraux en exclusion. Si plusieurs exclusions mutuelles sont possibles, nous explorons toutes les combinaisons possibles pour retrouver un pigeon. Lorsque nous considérons une clause de départ de taille n, nous pouvons nous arrêter dès lors que nous avons sélectionné un total de $n+1$ contraintes *at least 1*. À ce moment-là, nous pouvons indiquer que nous avons détecté un problème de pigeons et arrêter la recherche sur la branche de l'arbre où nous nous trouvons car la formule correspondante est incohérente. Nous pouvons fournir une explication très simple de l'incohérence à l'utilisateur. Nous allons maintenant revenir en détail sur chacune des étapes de notre approche.

3.2.1 Détecter les littéraux en exclusions mutuelles

Dans un premier temps, depuis une clause de départ $D = \{d_1, d_2, \dots d_k\}$, nous cherchons des clauses dont les littéraux sont en exclusion avec chacun des littéraux de D. La première étape de notre algorithme de détection va consister à choisir la clause de départ D et à identifier pour chaque littéral d_i tous les littéraux $\neg l_i$ en exclusion avec d_i (i.e. tels que $\phi \models \neg l_i \vee \neg d_i$). Pour ce faire, on effectue des propagations unitaires à partir de d_i et pour chaque littéral $\neg l_i$ on mémorise quel d_i a permis de le propager. Un littéral $\neg l_i$ peut être obtenu de différentes manières, à partir de la propagation de différents d_i. Donc pour chaque $\neg l_i$, on mémorise l'ensemble des d_i qui permettent de le propager. En pratique, pour chaque littéral $\neg l_i$, nous stockons ces ensembles sous forme d'un vecteur de bits $(b_k \dots b_1)$ où le bit b_i est à 1 si et seulement si d_i est en exclusion mutuelle avec $\neg l_i$.

Exemple 11. *Soit $D = (d_1 \vee d_2 \vee d_3 \vee d_4)$ une clause de départ, $C = (l_1 \vee l_2 \vee l_3 \vee l_4)$ une autre clause et les exclusions mutuelles suivantes : $\neg d_1 \vee \neg l_2$ et $\neg d_3 \vee \neg l_2$. En effectuant des propagations unitaires depuis chaque littéral de D, nous allons nous rendre compte que le littéral $\neg l_2$ peut être propagé à la fois à partir de d_1 et à partir de d_3. Nous devons donc indiquer que le littéral l_2 est en exclusion mutuelle avec les deux littéraux d_1 et d_3. En pratique, cela va se traduire par l'utilisation du vecteur de bits* (0101) *pour le littéral l_2. Dans la notation, l'ordre des bits dans les vecteurs de bits est donc inversé par rapport à l'ordre des littéraux dans la clause de départ.*

Nous pouvons considérer des ensembles de bits de taille quelconque, mais en pratique nous nous limiterons à des vecteurs de 64 bits. On appellera *marque* ce vecteur de bits dans le reste du document. Cela permet de détecter des problèmes de pigeons allant jusqu'au problème PHP_{64}, ce qui nous semble une limite raisonnable. L'algorithme *CreateMarks* (voir algorithme 3) résume cette partie de notre approche. Initialement tous les ensembles sont vides (ligne 1). La fonction *InitializeMarks*() a pour but de créer un vecteur de bits avec uniquement des 0 pour chaque littéral de la formule. Ensuite, nous allons lancer une propagation unitaire à partir de chacun

des littéraux de la clause de départ et nous allons marquer les littéraux opposés à ceux que nous allons propager (ligne 4). Par exemple, si le littéral d_i de la clause de départ permet de propager un littéral l, alors nous allons forcer à 1 le bit d'indice i dans le vecteur de bits associé à $\neg l$ pour indiquer que $\neg l$ est en exclusion mutuelle avec d_i. La fonction $UnitPropagation(d_i, i, m)$ part donc d'un littéral d_i et de sa position i dans la clause de départ et réalise une propagation unitaire à partir de d_i tout en mettant à jour les marques stockées dans m.

À noter

Comme notre but est avant tout de vérifier l'existence de certaines exclusions entre littéraux, la fonction $UnitPropagation$ doit réaliser toutes les propagations possibles, même si cela signifie propager des littéraux opposés (ce qui devrait normalement mener à un conflit et arrêter la propagation).

Par ailleurs, les propagations effectuées à partir de chacun des littéraux de la clause de départ doivent être indépendantes et il faut donc pouvoir annuler ces propagations. Concernant maintenant les littéraux présents dans la clause de départ, nous les forçons à avoir un unique bit à 1 dans leur vecteur de bit (ligne 3).

En terme de complexité, dans cet algorithme, nous utilisons régulièrement la propagation unitaire. Dans le pire des cas, réaliser cette opération demande de parcourir toutes les clauses et celles-ci contiennent au maximum k littéraux. Ainsi, la complexité de la propagation unitaire est en $O(km)$, où k est la taille de la plus grande clause et m est le nombre de clauses de la formule. Dans notre fonction $CreateMarks(C)$, nous lançons une propagation unitaire depuis chacun des littéraux de la clause C, qui contient au maximum k littéraux. La complexité de l'algorithme $CreateMarks$ est donc en $O(k^2m)$.

Algorithm 3 *CreateMarks*(C)

ENTREE : C - une clause de départ
SORTIE : m - les marques créées

```
1 : m ← InitializeMarks()
2 : for i = 0..|C| − 1 do
3 :     m[C[i], i] ← 1
4 :     UnitPropagation(C[i], i, m)
5 : end for
6 : return m
```

Exemple 12. *À titre d'exemple, nous allons considérer le problème de pigeons PHP_4 que nous modifions légèrement comme le montre la figure 3.2. On suppose les exclusions mutuelles classiques par colonnes présentes. L'exclusion mutuelle $\neg x_{1,4} \vee \neg x_{2,4}$ est remplacée par une exclusion indirecte grâce aux deux clauses binaires $\neg x_{1,4} \vee y$ et $\neg y \vee \neg x_{2,4}$ (où y est une nouvelle variable). Cette exclusion indirecte est représentée par un arc en pointillés bleus. Nous ajoutons aussi les exclusions suivantes : $\neg x_{1,1} \vee \neg x_{2,2}$, $\neg x_{1,2} \vee \neg x_{2,1}$, $\neg x_{1,2} \vee \neg x_{3,3}$ et $\neg x_{1,4} \vee \neg x_{3,3}$. Ces nouvelles exclusions sont représentées par des arcs rouges. Enfin, nous ajoutons une clause dont le but est d'être éliminée au cours de notre approche : $x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee z$ (où z est une nouvelle variable). Par la suite, nous allons régulièrement revenir sur cet exemple.*

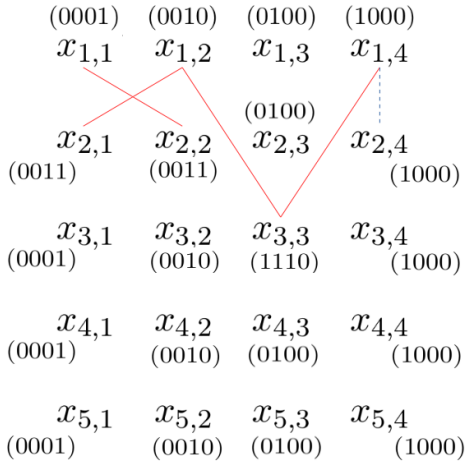


FIGURE 3.2 – Exemple utilisé pour illustrer l'algorithme de détection de pigeons purs.

Ici, nous pouvons par exemple choisir la clause $x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4}$ *comme départ de notre approche et elle recevra donc les marques* $(0001, 0010, 0100, 1000)$. *Par propagation unitaire, on montre que les littéraux ayant la même position dans chacune des clauses sont en exclusion avec le littéral correspondant dans la clause de départ. Par exemple, les littéraux* $x_{2,1}$, $x_{3,1}$, $x_{4,1}$ *et* $x_{5,1}$ *vont tous les quatre être en exclusion avec le littéral* $x_{1,1}$ *et vont donc recevoir la marque* (0001). *De même, les littéraux de la deuxième colonne reçoivent la marque* (0010), *ceux de la troisième colonne la marque* (0100), *et ceux de la quatrième colonne la marque* (1000). *Cependant, comme nous avons ajouté des exclusions mutuelles, certains littéraux vont cumuler des exclusions supplémentaires. C'est notamment le cas des littéraux* $x_{2,1}$ *et* $x_{2,2}$, *qui sont en exclusion à la fois avec le littéral* $x_{1,1}$ *et le littéral* $x_{1,2}$ *et qui vont finalement tous les deux recevoir la marque* (0011). *De même, le littéral* $x_{3,3}$ *est en exclusion avec les trois littéraux* $x_{1,2}$, $x_{1,3}$ *et* $x_{1,4}$ *et il va donc recevoir la marque* (1110). *Par ailleurs, le littéral* $\neg y$ *va recevoir la marque* (1000). *Les littéraux négatifs (sauf* $\neg y$*) ainsi que les littéraux* y *et* z *ne sont présents dans aucune exclusion.*

3.2.2 Énumérer les clauses candidates

Une étape importante de notre approche consiste à réaliser une forme de propagation sur les marques d'une clause. Dans un pigeon pur, comme chaque littéral d'une clause doit être en exclusion avec exactement un littéral de la clause de départ et comme tous les littéraux de la clause de départ doivent être présents dans des exclusions, nous savons que si un littéral dans une clause donnée est associé à un seul littéral l de la clause de départ, alors l ne peut plus être en exclusion avec les autres littéraux présents dans la clause considérée. Si à un moment, nous avons deux littéraux différents d'une clause chacun en exclusion avec un littéral de la clause de départ, alors ces exclusions doivent être différentes. Si ce n'est pas le cas, nous pouvons nous arrêter car un littéral de la clause de départ ne peut pas être en exclusion avec deux littéraux d'une même clause.

Exemple 13. *La figure 3.3 donne des exemples de ces propagations sur des vecteurs de bits. Tout d'abord, si nous avons l'ensemble de marques* $(1000, 1100, 0100, 0001)$ *pour une clause* $C = (l_1 \vee l_2 \vee l_3 \vee l_4)$ *donnée, nous remarquons que les premier et troisième vecteurs n'ont qu'un seul bit à* 1, *ce qui signifie que le littéral* l_1 *est en exclusion avec uniquement* d_4 *et le littéral*

l_3 est en exclusion avec uniquement d_3. De ce fait, ni d_3 ni d_4 ne peuvent plus être réutilisés pour former une exclusion mutuelle avec l_2 ou l_4. Les marques (1000) *et* (0100) *ne peuvent plus être utilisées par les autres vecteurs de bits de l_2 et l_4. Ainsi, le vecteur de bits de l_2 ne va plus avoir aucun bit à* 1 *et donc le littéral correspondant ne sera plus en exclusion avec aucun littéral de la clause départ D. Ainsi, la clause C ne peut pas apparaître dans un problème de pigeons. Par contre, si nous avons l'ensemble de marques* $(1000, 1110, 0100, 0001)$*, alors dans ce cas nous pouvons les simplifier et obtenir* $(1000, 0010, 0100, 0001)$.

$$(1000, 1100, 0100, 0001) \rightarrow \text{échec}$$
$$(1000, 1110, 0100, 0001) \rightarrow (1000, 0010, 0100, 0001)$$

FIGURE 3.3 – Exemples de propagations sur des vecteurs de bits.

L'algorithme *UnitPropagationBitmask* (voir algorithme 4) décrit cette procédure. Tout d'abord, nous initialisons un tableau permettant d'indiquer quels vecteurs de bits ont déjà été considérés (ligne 1). Tant qu'il existe un vecteur de bits avec un unique bit à 1 et que nous n'avons pas encore considéré (ligne 2), nous forçons à 0 le bit correspondant dans les autres vecteurs de bits de *marks* (ligne 3) et nous indiquons que nous avons fini de considérer ce vecteur de de bits (ligne 4). Une fois toutes les propagations possibles effectuées, nous pouvons donner le tableau avec les marques mises à jour (ligne 6).

Dans cet algorithme, le pire des cas se produit lorsque chaque passage dans le tableau de marques fait apparaître un nouveau vecteur avec un unique bit à 1 et que cela se produit pour chaque marque. C'est par exemple le cas pour le tableau de marques $(0001, 0011, 0111, 1111)$. Ainsi, la complexité de l'algorithme *UnitPropagationBitmask* est en $O(k^2)$, où k est la taille de la plus grande clause.

Algorithm 4 *UnitPropagationBitmask*(*marks*)

ENTREE : *marks* - un tableau de vecteurs de bits représentant les marques d'une clause
SORTIE : les marques mises à jour

1 : *done* ← tableau de booléens de la même taille que *marks* initialement tous à faux
2 : **while** $\exists v_i \in marks \mid v_i$ a un seul bit à 1 and not *done*$[i]$ **do**
3 : forcer le bit correspondant à 0 dans les autres marques de *marks*
4 : *done*$[i]$ ← *True*
5 : **end while**
6 : **return** *marks*

Exemple 14. *Dans notre exemple de problème de pigeons, nous savons que la clause* $x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4}$ *a pour marques* $(0001, 0010, 1110, 1000)$*. Nous remarquons que les littéraux* $x_{3,1}$*,* $x_{3,2}$ *et* $x_{3,4}$ *ont tous les trois un vecteur de bits avec un unique bit à* 1*. Nous forçons alors les bits correspondants à* 0 *dans le vecteur de bits de* $x_{3,3}$*. Ce littéral va devenir unitaire puisque son vecteur de bits va prendre la valeur* 0100*. Nous allons donc devoir effectuer une propagation depuis le vecteur de bits de* $x_{3,3}$ *mais cela ne va pas modifier les autres marques. À ce moment-là, nous avons considéré tous lesvecteurs de bits possibles et nous pouvons donc arrêter l'algorithme. Au final, les nouvelles marques de la clause considérée sont :* $(0001, 0010, 0100, 1000)$.

Les propagations précédentes ne permettent pas toujours d'associer un unique littéral d_i à chacun des littéraux l_i d'une clause. Si nous avons plusieurs possibilités pour un littéral, nous devons toutes les explorer. L'algorithme *CreateCombinations* (voir algorithme 5) détaille cette procédure. Tout d'abord, nous vérifions si le tableau de marques considéré contient un vecteur de bits avec uniquement des 0 (cela peut arriver quand nous avons deux vecteurs de bits identiques ayant un unique bit à 1) et si c'est le cas aucune combinaison ne peut être créée (lignes 1 à 3). Si tous les vecteurs de bits présents dans *marks* sont unitaires, alors nous avons trouvé une nouvelle combinaison possible (lignes 4 à 6). Si nous ne nous trouvons dans aucun des cas précédents, nous créons un tableau initialement vide visant à récupérer les combinaisons pouvant être créées à partir des marques courantes (ligne 7). Ensuite, pour chacun des vecteurs de bits v ayant au moins deux bits à 1 (ligne 8), nous devons considérer toutes les possibilités. Pour ce faire, pour chaque indice i de bit à 1 (ligne 9) dans v, nous sauvegardons la valeur de v (ligne 10) et nous forçons ensuite à 0 tous les bits de v sauf celui de i (ligne 11). Une fois cela fait, nous pouvons propager le nouveau vecteur de bits unitaire que nous venons de créer (ligne 12) et ensuite nous récupérer toutes les combinaisons pouvant être créées avec les marques mises à jour (ligne 13). Nous pouvons alors récupérer la valeur de v précédemment suvegardée (ligne 14). Une fois tous les vecteurs de bits considérés, nous pouvons donner les combinaisons créées (ligne 17).

Ici, le pire des cas survient lorsque chaque littéral de la clause C est en exclusion avec tous les littéraux de la clause de départ. Ainsi, il faut créer toutes les combinaisons possibles pour cette clause, qui est au maximum de longueur k. La complexité de l'algorithme *CreateCombinations* est donc en $O(k!)$, où k est la taille de la plus grande clause.

Algorithm 5 *CreateCombinations*(*marks*)

ENTREE : *marks* - les marques dont on veut créer les combinaisons
SORTIE : *combs* - Les combinaisons créées

```
1 : if ∃v ∈ marks | v = 0 then
2 :    return ∅
3 : end if
4 : if toutes les marques n'ont qu'un bit à 1 then
5 :    return {marks}
6 : end if
7 : combs ← ∅
8 : for v ∈ marks | v contient au moins deux bits à 1 do
9 :    for i indice de bit à 1 dans v do
10 :       backup ← v
11 :       forcer à 0 tous les bits de v sauf i
12 :       newMarks ← UnitPropagationBitmask(marks)
13 :       combs ← combs ∪ CreateCombinations(newMarks)
14 :       v ← backup
15 :    end for
16 : end for
17 : return combs
```

Exemple 15. *Dans notre exemple de problème de pigeons, nous savons que la clause* $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$ *a pour marques* $(0011, 0011, 0100, 1000)$. *Ici, il n'y a que les littéraux* $x_{2,1}$ *et* $x_{2,2}$ *qui ont des vecteurs de bits avec au moins deux bits à* 1. *Nous commençons par considérer le*

vecteur de bits de $x_{2,1}$*. Nous observons que son bit* b_0 *est à* 1 *et nous pouvons donc forcer à* 0 *les autres bits de son vecteur de bits ainsi que le bit correspondant dans les autres vecteurs de bits. Cette opération va rendre unitaire le vecteur de bits de* $x_{2,1}$ *(nous obtenons alors* 0001*) ainsi que celui de* $x_{2,2}$ *(nous obtenons alors* 0010*). À ce moment-là, tous les vecteurs de bits sont devenus unitaires et nous avons comme première combinaison :* $(0001, 0010, 0100, 1000)$*. Ensuite, nous observons que le bit* b_1 *de* $x_{2,1}$ *est également à* 1*. En procédant de la même manière qu'avec le bit* b_0*, nous rendons unitaire le vecteur de bits de* $x_{2,1}$ *(nous obtenons alors* 0010*) ainsi que celui de* $x_{2,2}$ *(nous obtenons alors* 0001*). De cette manière, nous obtenons une deuxième combinaison possible :* $(0010, 0001, 0100, 1000)$*. Si l'on considère maintenant le vecteur de bits de* $x_{2,2}$*, nous obtenons les mêmes combinaisons qu'avec* $x_{2,1}$*.*

Nous pouvons maintenant créer les clauses candidates pour tenter de construire un problème de pigeons à partir d'une clause de départ. Cette étape est détaillée par l'algorithme *CreateCandidates* (voir algorithme 6). Nous commençons par récupérer les marques en propageant les littéraux présents dans la clause de départ (ligne 1). Ensuite, nous effectuons une première sélection parmi les clauses de la formule. Nous ne retenons que les clauses qui sont de la même taille que la clause de départ et qui n'ont aucun littéral en commun avec cette clause de départ (ligne 2). Nous créons également une variable dont le but est de contenir toutes les clauses candidates créées (ligne 3) ainsi qu'une autre variable dont le but est de compter le nombre de clauses différentes utilisées pour créer des clauses candidates (ligne 4). Pour chacune des clauses que nous avons retenues, nous allons tenter de créer des clauses candidates. Nous commençons par récupérer les marques de la clause courante et nous effectuons une propagation sur celles-ci (ligne 6). Si aucun problème n'a été rencontré pendant la propagation (ligne 7), alors nous pouvons créer les combinaisons à partir des marques obtenues pour la clause courante (ligne 8). Si nous avons obtenu au moins une combinaison valide, nous pouvons augmenter notre compteur (lignes 9 à 11). Ensuite, nous récupérons chacune des combinaisons valides (ligne 12) et nous trions la clause courante en fonction des marques obtenues (ligne 13). Cela va nous donner une nouvelle clause candidate que nous pouvons prendre en compte (ligne 14). La fonction *SortClause*(*cl*, *comb*) a donc pour but de trier la clause *cl* en fonction des marques de *comb*. Ici, nous choisissons l'ordre des marques des littéraux de la clause de départ. Une fois toutes les clauses candidates créées, nous vérifions si nous avons utilisé suffisamment de clauses initiales différentes (i.e. le nombre de clauses utilisées plus la clause de départ doit être strictement plus grand que la taille de la clause de départ) et si c'est le cas, nous pouvons lancer la création du problème de pigeons (lignes 18 à 20) qui sera détaillée dans la section suivante.

Pour cet algorithme, nous commençons par appeler la fonction *CreateMarks*, qui est de complexité $O(k^2m)$, avec k la taille de la plus grande clause et m le nombre de clauses de la formule. Ensuite, pour chaque clause sélectionnée, nous appelons la fonction *UnitPropagationBitmask* (de complexité $O(k^2)$) puis la fonction *CreateCombinations* (de complexité $O(k!)$). De plus, nous devons trier chaque clause en fonction des différentes combinaisons de marques obtenues. Pour chaque combinaison, le tri est en $O(k \times log(k))$. Nous verrons par la suite que la fonction *PigeonHoleConstruction* a une complexité en $O(m^k)$. Ainsi, la complexité de l'algorithme *CreateCandidates* est en $O(k^2m+m(k^2+k!k\times log(k))+m^k)$, c'est-à-dire en $O(k!mk\times log(k)+m^k)$.

Exemple 16. *Dans notre exemple, nous pouvons tout d'abord éliminer toutes les clauses binaires puisqu'elles ne sont pas de la même taille que* $x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4}$ *qui est notre clause de départ. Pour la clause* $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$*, l'étape de propagation va nous donner les marques* $(0011, 0011, 0100, 1000)$*, ce qui nous permet de créer les deux combinaisons* $(0001, 0010, 0100, 1000)$ *et* $(0010, 0001, 0100, 1000)$*. Ainsi, nous allons respectivement créer les clauses candidates* $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$ *et* $x_{2,2} \vee x_{2,1} \vee x_{2,3} \vee x_{2,4}$*. Concernant la clause*

Algorithm 6 *CreateCandidates*(C)

ENTREE : C - la clause de départ
SORTIE : Est-il possible de détecter un problème de pigeons ?

1 : $marks \leftarrow CreateMarks(C)$
2 : $chose \leftarrow \{cl \mid cl \in \phi \wedge |cl| = |C| \wedge C \cap cl = \emptyset\}$
3 : $candidates \leftarrow \emptyset$
4 : $cpt \leftarrow 0$
5 : **for** $cl \in chose$ **do**
6 : $newMarks \leftarrow UnitPropagationBitmask(\{marks[l] \mid l \in cl\})$
7 : **if** $newMarks \neq \emptyset$ **then**
8 : $combinations \leftarrow CreateCombinations(newMarks)$
9 : **if** $combinations \neq \emptyset$ **then**
10 : $cpt \leftarrow cpt + 1$
11 : **end if**
12 : **for** $comb \in combinations$ **do**
13 : $newCandidate \leftarrow SortClause(cl, comb)$
14 : $candidates \leftarrow candidates \cup newCandidate$
15 : **end for**
16 : **end if**
17 : **end for**
18 : **if** $cpt + 1 > |C|$ **then**
19 : **return** $PigeonHoleConstruction(C, candidates, \{C\})$
20 : **end if**

$x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4}$*, la propagation va nous donner les marques* $(0001, 0010, 0100, 1000)$*, qui sera également la seule combinaison possible. Cette dernière va nous donner la clause candidate* $x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4}$*. Les clauses* $x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4}$ *et* $x_{5,1} \vee x_{5,2} \vee x_{5,3} \vee x_{5,4}$ *ont toutes les deux les marques* $(0001, 0010, 0100, 1000)$ *et seront donc considérées telles quelles comme clauses candidates car leurs marques sont toutes unitaires et déjà ordonnées. Concernant la clause* $x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee z$*, celle-ci a les marques* $(0001, 0010, 0100, 0000)$ *et la propagation va repérer qu'un vecteur de bits est vide. Cette clause ne va donc pas être prise en compte. Finalement, les clauses candidates correspondent à l'ensemble* $\{x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}, x_{2,2} \vee x_{2,1} \vee x_{2,3} \vee x_{2,4}, x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4}, x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4}, x_{5,1} \vee x_{5,2} \vee x_{5,3} \vee x_{5,4}\}$*. Ici, nous avons utilisé quatre clauses différentes pour générer les clauses candidates, ce qui est suffisant pour lancer l'étape de construction du problème de pigeons.*

3.2.3 Reconstruire le problème de pigeons

Comme nous permettons l'exclusion indirecte de littéraux (par propagation unitaire), il est possible que les littéraux d_i et l_i soient en exclusion indirecte, d_i étant un littéral de la clause de départ, soit $d_i \vdash_{UP} \neg l_i$. Cependant, cela ne signifie pas forcément que $l_i \vdash_{UP} \neg d_i$. Nous devons donc le vérifier pour nous assurer que nous avons vraiment détecté un problème de pigeons. Comme les littéraux des clauses sont triés par ordre de leur vecteur de bits, nous devons vérifier que le littéral d'indice i dans la clause à ajouter exclut bien les littéraux présents à l'indice i dans les clauses précédemment retenues. Cette fonction est détaillée par l'algorithme *CanSelect* (voir algorithme 7). Dans celui-ci, nous parcourons chacun des littéraux de la clause à ajouter (ligne 1) et nous récupérons les littéraux ayant le même indice dans les clauses du problème que nous construisons (ligne 2). Cela nous donne les littéraux devant être en exclusion mutuelle avec le littéral auquel nous sommes arrivés. Pour ce faire, nous récupérons les propagations pouvant être effectuées à partir du littéral considéré (ligne 3) et nous vérifions que, pour chaque littéral précédemment récupéré dans le problème que nous construisons, nous avons bien propagé sa négation (lignes 4 et 5). Si une exclusion est manquante, nous nous arrêtons en indiquant que la clause ne peut pas être ajoutée (ligne 6). Sinon, nous passons au littéral suivant dans la clause à ajouter. Si nous avons parcouru tous les littéraux de cette clause et que nous avons trouvé toutes les exclusions nécessaires, nous pouvons alors ajouter la clause (ligne 10). Le fait de passer la valeur -1 à la fonction *UnitPropagation* permet d'indiquer que l'on ne veut pas mettre à jour les marques des littéraux propagés.

Ici, le pire des cas survient lorsque nous considérons une clause C de taille k et que nous avons déjà un nombre k de clauses présentes dans le problème de pigeons que nous sommes en train de construire. Pour chacun des littéraux de C, nous devons donc retrouver un nombre k de littéraux par propagation unitaire. Or, la propagation unitaire a une complexité en $O(km)$, avec k la taille de la plus grande clause. La complexité de l'algorithme *CanSelect* est donc en $O(k^2 m)$.

Exemple 17. *Si nous reprenons notre exemple de problèmes de pigeons, nous pouvons choisir par exemple la clause* $x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4}$ *comme clause de départ. Elle sera donc présente de base dans le problème de pigeon que nous allons tenter de construire. Parmi les clauses candidates, nous pouvons par exemple tenter d'ajouter la clause* $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$*. Pour vérifier si cette action est valide, nous devons vérifier que les littéraux* $x_{2,1}$*,* $x_{2,2}$*,* $x_{2,3}$ *et* $x_{2,4}$ *propagent respectivement* $\neg x_{1,1}$*,* $\neg x_{1,2}$*,* $\neg x_{1,3}$ *et* $\neg x_{1,4}$*. Ici, si nous propageons chacun des littéraux de la clause que nous souhaitons ajouter, nous retrouvons bien les exclusions nécessaires grâce aux clauses binaires* $\neg x_{1,1} \vee \neg x_{2,1}$*,* $\neg x_{1,2} \vee \neg x_{2,2}$*,* $\neg x_{1,3} \vee \neg x_{2,3}$*,* $\neg x_{1,4} \vee y$ *et* $\neg y \vee \neg x_{2,4}$*. Nous pouvons donc ajouter cette clause au problème que nous sommes en train de construire.*

Algorithm 7 *CanSelect*(C, *pigeon*)

ENTREE : C - la clause candidate que l'on veut ajouter, *pigeon* - le problème de pigeons que nous sommes en train de construire
SORTIE : la clause C peut-elle être ajoutée à *pigeon* ?

```
1 : for i = 0 ... |C| − 1 do
2 :     find ← {cl[i] | cl ∈ pigeon}
3 :     propagations ← UnitPropagation(C[i], −1, ∅)
4 :     for l ∈ find do
5 :         if ¬l ∉ propagations then
6 :             return false
7 :         end if
8 :     end for
9 : end for
10 : return true
```

Imaginons que le problème que nous construisons contienne cette fois-ci la clause de départ et également la clause candidate $x_{2,2} \vee x_{2,1} \vee x_{2,3} \vee x_{2,4}$ (cela est possible car nous avons les exclusions mutuelles nécessaires). Si nous essayons maintenant d'ajouter la clause candidate $x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4}$, nous allons rencontrer un problème. En effet, en parcourant les littéraux de cette clause, nous remarquons par exemple que le littéral $x_{3,1}$ doit propager à la fois $\neg x_{1,1}$ et $\neg x_{2,2}$. Or $x_{3,1}$ et $x_{2,2}$ ne s'excluent pas mutuellement. Ainsi, il n'est pas possible d'ajouter cette clause au problème que nous avons construit actuellement. Un comportement similaire peut être observé avec les deux autres clauses candidates $x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4}$ et $x_{5,1} \vee x_{5,2} \vee x_{5,3} \vee x_{5,4}$.

Il ne nous reste maintenant plus qu'à construire le problème de pigeons à partir des clauses candidates créées. Cette partie est détaillée par l'algorithme *PigeonHoleConstruction* (voir algorithme 8). Si à un moment donné le nombre de clauses du problème de pigeons que nous construisons est plus grand que le nombre de littéraux dans la clause de départ, alors nous avons détecté un problème de pigeons et nous pouvons nous arrêter en indiquant cela (lignes 1 à 3). Sinon, nous considérons chacune des clauses candidates (ligne 4) et nous essayons de l'ajouter au problème de pigeons en construction (ligne 5). Si cela est possible, nous supprimons des clauses candidates celles qui ont des littéraux communs avec la clause candidate courante (ligne 6). Cela va au moins supprimer les clauses candidates générées à partir de la même clause initiale que la clause que nous venons d'ajouter. Si il est toujours possible de construire un problème de pigeons (ligne 7), alors nous faisons un nouvel appel de la fonction en ajoutant la nouvelle clause au problème en construction et aussi en considérant les clauses candidates filtrées (ligne 8). Ici la fonction *DifferentClauses*(*remain*) a pour but de donner le nombre de clauses initiales différentes utilisées pour créer les candidates de *remain*. On effectue un nouvel appel si la somme de ce nombre avec le nombre de clauses présentes dans le problème de pigeons en construction (il faut aussi considérer que nous allons lui ajouter une nouvelle clause) est strictement plus grand que la taille de la clause de départ. Si un problème de pigeons est détecté, il faut aussi arrêter tous les appels récursifs précédents (lignes 9 à 11). Si nous avons considéré toutes les clauses candidates sans trouver de problème de pigeons, nous devons aussi l'indiquer (ligne 15).

Pour cet algorithme, le pire des cas correspond au cas où nous avons dû considérer toutes les clauses de la formule et générer une unique clause candidate pour chaque clause initiale. De plus, toutes ces clauses candidates peuvent être utilisées pour étendre le problème de pigeons

en construction mais sans jamais réussir à détecter de problème de pigeons. Si nous avons considéré une clause de taille k, nous devons alors considérer toutes les façons de sélectionner k clauses candidates parmi les m clauses candidates créées. Ainsi, la complexité de l'algorithme *PigeonHoleConstruction* est en $O(C_m^k)$. Une autre façon de voir les choses consiste à considérer que, pour chacune des k clauses à sélectionner pour le problème de pigeons, nous allons parcourir les m clauses candidates. La complexité de l'algorithme peut donc se réécrire en $O(m^k)$.

Algorithm 8 *PigeonHoleConstruction*(C, *candidates*, *pigeon*)

ENTREE : C - la clause de départ, *candidates* - les clauses candidates, *pigeon* - le problème de pigeons que nous construisons
SORTIE : Est-il possible de construire un problème de pigeons ?

```
1 : if |pigeon| > |C| then
2 :    return ⊤
3 : end if
4 : for cand ∈ candidates do
5 :    if CanSelect(cand, pigeon) then
6 :       remain ← {cl | cl ∈ candidates ∧ cand ∩ cl = ∅}
7 :       if DifferentClauses(remain) + |pigeon| + 1 > |C| then
8 :          r ← PigeonHoleConstruction(C, remain, pigeon ∪ {cand})
9 :          if r = ⊤ then
10 :            return ⊤
11 :         end if
12 :      end if
13 :   end if
14 : end for
15 : return ⊥
```

Exemple 18. *Parmi les clauses candidates, nous pouvons par exemple commencer par essayer d'ajouter la clause* $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$ *à notre clause de départ. Nous avons déjà vu que cela est possible car nous avons toutes les exclusions nécessaires. Une fois cette clause ajoutée, nous pouvons la supprimer des clauses candidates ainsi que* $x_{2,2} \vee x_{2,1} \vee x_{2,3} \vee x_{2,4}$*. Il nous reste donc les trois clauses candidates* $x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4}$*,* $x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4}$ *et* $x_{5,1} \vee x_{5,2} \vee x_{5,3} \vee x_{5,4}$ *que nous allons pouvoir ajouter dans n'importe quel ordre. Une fois cela fait, le problème que nous construisons contient* 5 *clauses alors que la clause de départ contient* 4 *littéraux. Nous pouvons donc nous arrêter en indiquant que nous avons trouvé un problème de pigeons. Nous avons bien retrouvé le problème* PHP_4 *introduit au début de cet exemple.*

Avec cette façon de faire, nous pouvons déjà prendre en compte un certain nombre de cas particuliers. En effet, les problèmes de pigeons que nous détectons peuvent très bien mélanger littéraux positifs et négatifs et l'utilisation de nos vecteurs de bits permet d'ignorer l'ordre des littéraux à l'intérieur des contraintes. Par ailleurs, il est possible de détecter nos problèmes de pigeons même si des exclusions mutuelles sont présentes sous la forme de plusieurs contraintes. Cependant, notre approche échoue si, par exemple, une contrainte *at least 1* a été divisée en plusieurs clauses.

À noter

Même si la complexité dans le pire des cas est importante, elle est largement surestimée car elle ne tient pas compte du filtrage effectué. Le nombre de clauses en exclusion avec la clause de départ est souvent assez réduit et n'est jamais égal à m.

3.2.4 Résultats expérimentaux

Nous avons implémenté notre approche avec le langage de programmation Python. Toutes les méthodes évoquées ainsi qu'un algorithme de recherche inspiré de DPLL ont été implémentées. Nous ne nous sommes pas basés sur le code d'un solveur déjà existant pour cette partie. L'heuristique implémentée sélectionne la première variable non affectée. Nous avons sélectionné un certain nombre d'instances provenant des compétitions SAT'02 [SLH05] et SAT'03 [LS03] pour tester notre approche. Pour chaque instance, nous avons imposé un temps limite de 30 minutes.

Un extrait des résultats obtenus peut être observé dans la table 3.1 et des résultats plus globaux peuvent être trouvés dans la table 3.2. Pour les instances de la famille `homer` ainsi que certaines instances de routage, notre approche est particulièrement efficace car nous détectons un problème de pigeons à la racine, ce qui permet d'arrêter directement la recherche. Cependant, ce n'est pas le comportement qui a été le plus observé. En effet, dans la plupart des cas, lorsque l'on détecte quelque chose, nous ne retrouvons que des problèmes de pigeons de taille 2, ce qui ne permet pas de faire une compression significative. De plus, nous avons régulièrement observé des problèmes de performances avec notre approche. Celle-ci n'avait pas pour but initial d'être la plus performante possible et donc nous nous sommes souvent retrouvés dans le cas où une instance n'avait toujours pas fini d'explorer le premier nœud de l'arbre de recherche quand l'algorithme a été arrêté. Lors de cette première expérimentation, nous ne sommes pas parvenus à détecter des problèmes de pigeons en quantité et en taille suffisantes pour avoir une compression significative. Cependant, il reste possible que ces problèmes existent mais que nous soyons passés à côté de ceux-ci car pour la plupart des instances, nous n'avons pas pu explorer complètement l'arbre de recherche.

3.3 Faire de l'échantillonnage

3.3.1 Principe

Afin de pallier les importants problèmes de performance rencontrés et afin aussi de voir si les arbres de recherche développés contiennent des problèmes de pigeons, nous avons aussi envisagé une approche basée sur de l'échantillonnage pour limiter l'usage de la détection de pigeons très chronophage à quelques nœuds de l'arbre seulement. Pour cette approche, nous laissons tourner un solveur sur une instance donnée et nous sélectionnons un certain nombre de branches dans l'arbre de recherche obtenu. Par exemple, nous pouvons sélectionner une branche tous les x conflits rencontrés et nous pouvons aussi sélectionner un nombre maximum y de branches. Une fois ces branches sélectionnées, nous testons notre algorithme de détection de problèmes de pigeons purs en remontant ces branches, c'est-à-dire que nous testons d'abord les nœuds se trouvant juste avant le conflit puis ceux encore avant jusqu'à revenir à la racine. Il faut tout de même faire attention à ne pas lancer plusieurs fois l'algorithme de détection si nous

TABLE 3.1 – Résultats expérimentaux de notre algorithme de détection de problèmes de pigeons purs. Pour chaque instance, nous indiquons si l'algorithme de recherche a terminé dans le temps limite de 30 minutes et nous indiquons ensuite le nombre de nœuds explorés, les tailles de problèmes de pigeons détectés, le nombre total de problèmes reconnus (la valeur entre parenthèses indique le nombre de problèmes différents reconnus) et le temps en secondes. Un tiret indique qu'aucun problème de pigeons n'a été reconnu.

Instance	Terminé	Explorés	Tailles détectées	Quantité	Temps
homer06	oui	1	9	1 (1)	0.233
homer11	oui	1	10	1 (1)	1.015
homer16	oui	1	11	1 (1)	0.422
ca004	oui	513	2	121 (12)	19.681
ca008	non	11177	2	3077 (39)	1800
ca064	non	2051	2	339 (3)	1800
dp03u02	non	82	2	17 (2)	1800
dp05u04	non	8	-	0 (0)	1800
BMC (4)	non	5545	2	1320 (10)	1800
BMC (42)	non	7557	2	2554 (3)	1800
9symml_gr_rcs_w5	oui	1	5	1 (1)	59.949
c499_gr_rcs_w5	oui	1	5	1 (1)	48.158
term1_gr_rcs_w3	non	0	-	0 (0)	1800
am_4_4	non	24	-	0 (0)	1800
am_5_5	non	3	-	0 (0)	1800
hanoi4u	non	0	-	0 (0)	1800
hanoi5u	non	0	-	0 (0)	1800
rope_0003	oui	6339	2	2236 (36)	335.570
rope_0004	non	35981	2	5695 (47)	1800
rope_0005	non	129321	2	25975 (40)	1800

TABLE 3.2 – Résumé de nos expérimentations sur la détection de problèmes de pigeons. Pour chaque compétition, nous donnons le nombre d'instances incohérentes testées, le nombre d'instances résolues et le nombre d'instances sur lesquelles nous avons détecté au moins un problème de pigeons.

Compétition	#UNSAT	Résolues	Au moins une détection
SAT'02	72	21	31
SAT'03	5	0	0

retrouvons un nœud déjà exploré. L'idée ici est de tester en priorité des formules de petite taille, sur lesquelles il est donc plus facile de tester notre approche.

3.3.2 Résultats expérimentaux

Nous avons voulu tester notre approche basée sur de l'échantillonnage afin de déterminer si des problèmes de pigeons pouvaient réellement être trouvés dans les instances que nous testons. Pour cette expérimentation, nous avons choisi $x = y = 100$, i.e. nous sélectionnons une branche tous les 100 conflits rencontrés et nous sélectionnons au maximum 100 branches. Par ailleurs, la première branche est forcément sélectionnée pour l'échantillonnage afin de faire en sorte d'avoir tout de même au moins une branche à explorer, au cas où l'arbre de recherche serait trop petit. Les résultats sur un certain nombre d'instances sont visibles dans la table 3.3 et des

résultats plus globaux peuvent trouvés dans la table 3.4. Les résultats obtenus ne sont toujours pas très encourageants. En effet, sur les instances qui répondent, à part pour les instances `c499_gr_rcs_w5.cnf` et `am_4_4.cnf` sur lesquelles nous avons réussi à détecter des pigeons de taille 3, nous n'avons retrouvé que des pigeons de taille 2. De plus, nous avons toujours des problèmes de performances et nous n'explorons en général qu'un nombre limité de nœuds dans les branches sélectionnées.

TABLE 3.3 – Résultats expérimentaux de notre algorithme de détection de problèmes de pigeons purs lorsque l'on fait de l'échantillonnage. Pour chaque instance, nous indiquons si l'algorithme a terminé dans le temps limite de 30 minutes et ensuite nous donnons le nombre de nœuds explorés, les tailles de problèmes de pigeons détectés, le nombre total de problèmes reconnus (la valeur entre parenthèses indique le nombre de problèmes différents reconnus) et le temps en secondes. Un tiret indique qu'aucun problème de pigeons n'a été reconnu.

Instance	Terminé	Explorés	Tailles détectées	Quantité	Temps
lisa19_0_a	non	0	-	0 (0)	1800
lisa19_99_a	non	1	2	1 (1)	1800
lisa20_0_a	non	0	-	0 (0)	1800
ca004	oui	71	2	19 (4)	6.943
ca008	oui	1101	2	384 (11)	524.818
ca064	non	890	2	863 (2)	1800
dp03u02	non	217	2	217 (12)	1800
dp05u04	non	0	-	0 (0)	1800
BMC (4)	oui	960	2	10 (1)	235.754
BMC (42)	oui	835	2	495 (3)	1242.910
9symml_gr_rcs_w5	non	0	-	0 (0)	1800
c499_gr_rcs_w5	non	5	2, 3	5 (4)	1800
term1_gr_rcs_w3	non	0	-	0 (0)	1800
am_4_4	non	69	2, 3	62 (54)	1800
am_5_5	non	4	2	3 (3)	1800
hanoi4u	non	0	-	0 (0)	1800
hanoi5u	non	0	-	0 (0)	1800
rope_0003	oui	540	2	66 (25)	178.250
rope_0004	oui	913	2	78 (20)	264.480
rope_0005	oui	978	2	85 (17)	198.760

TABLE 3.4 – Résumé de nos expérimentations sur la détection de problèmes de pigeons avec échantillonnage. Pour chaque compétition, nous donnons le nombre d'instances incohérentes testées, le nombre d'instances résolues et le nombre d'instances qui ont détecté au moins un problème de pigeons.

Compétition	#UNSAT	Résolues	Au moins une détection
SAT'02	71	14	20
SAT'03	5	0	3

3.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la détection de problèmes de pigeons en détectant syntaxiquement les contraintes *at least 1* et sémantiquement les contraintes *at most*

1 (i.e. les exclusions mutuelles). Ces problèmes de pigeons sont des problèmes incohérents classiques et facilement explicables à un utilisateur. Notre approche se base sur la détection de contraintes de cardinalité et consiste à sélectionner des clauses d'une formule donnée tout en vérifiant l'existence de suffisamment d'exclusions mutuelles entre les littéraux de ces clauses. Nous avons également supposé que les clauses devant être sélectionnées sont toutes de la même taille et donc tous les littéraux présents dans celles-ci doivent être présents dans au moins une exclusion. Nous considérons que nous avons détecté un problème de pigeons dès lors que le nombre de clauses sélectionnées est plus grand que la taille de ces clauses. Si cela se produit au cours de la recherche d'un solveur, nous pouvons alors arrêter la recherche sur la branche courante.

Concernant nos expérimentations, nous avons implémenté cette approche à l'intérieur d'une recherche DPLL. Pour certaines familles d'instances, nous réussissons à détecter un problème de pigeons au niveau de la racine de l'arbre de recherche, ce qui permet de décider directement de l'incohérence du problème. Cependant, en règle générale, soit nous rencontrons des problèmes de performances liés à la complexité élevée de l'approche et à notre implémentation, soit nous ne détectons que des problèmes de pigeons de petite taille (ce qui ne permet pas de réaliser une compression significative de l'arbre de recherche). Comme en général nous n'explorons qu'une partie limitée de l'arbre de recherche et comme notre exploration est guidée par l'heuristique, il reste tout de même la possibilité que nous soyons passés à côté de problèmes de pigeons plus importants et donc plus à même de réaliser une compression significative. Nous avons également tenté une approche basée sur de l'échantillonnage pour tenter de trouver d'autres problèmes de pigeons mais celle-ci ne nous a pas véritablement apporté de nouveaux résultats.

Chapitre 4

Détection syntaxique de motifs récurrents

Les travaux concernant l'utilisation du cache en post-traitement et son intégration dans un solveur DPLL ainsi que les résultats associés ont été publiés et présentés à la conférence internationale ICAART 2023 [BLPR23] et aux Journées Francophones de Programmation par Contraintes (JFPC) 2023. L'intégration dans un solveur CDCL et l'encodage étendu ont été présentés au workshop international Pragmatics of SAT (PoS) 2023.

4.1 Exemple introductif : décomposition du problème des pigeons

Pour illustrer et motiver notre but, considérons que nous avons un problème de pigeons de taille n à résoudre. 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 reproduit quand on explore les n façons de placer le premier pigeon dans l'un des n pigeonniers disponibles. Une fois que le premier sous-problème PHP_{n-1} a été exploré, il peut être enregistré dans un 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 problème. La figure 4.1 illustre l'imbrication des sous-problèmes de pigeons.

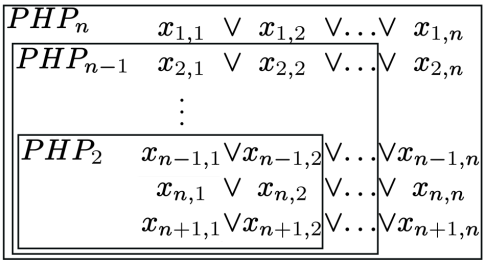


FIGURE 4.1 – Imbrication des sous-problèmes pour un problème de pigeons de taille n.

Considérons par exemple le problème PHP_4 et une heuristique qui décide négativement la première variable non assignée. Nous supposons disposer d'un système de *cache* nous permettant de stocker des sous-formules UNSAT et aussi de vérifier si l'une d'entre elles peut être considérée

comme équivalente à la sous-formule courante. Nous considérons un comportement semblable à un solveur DPLL, c'est-à-dire que pour revenir en arrière, nous inversons la dernière décision qui n'a pas encore été inversée. 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 $\neg x_{5,4}$ par l'intermédiaire de diverses exclusions mutuelles présentes dans la formule. Nous arrivons maintenant au problème PHP_3 que nous devons explorer car le cache est actuellement vide. 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 $x_{2,3}$. Les exclusions mutuelles vont alors propager $\neg x_{3,3}$, $\neg x_{4,3}$ et $\neg x_{5,3}$. Nous arrivons maintenant au problème PHP_2. Celui-ci sera entièrement exploré en décidant $\neg x_{3,1}$ puis en inversant cette décision. Ces 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 retrouver à chaque fois un problème PHP_2 mais basé sur des variables différentes. En consultant le cache, nous savons que nous avons déjà exploré ce problème à un renommage de variables près et nous pouvons directement en conclure que ces deux branches sont incohérentes. Par la suite, nous utiliserons le terme de *hit* pour désigner une telle reconnaissance. Nous pouvons maintenant stocker le problème PHP_3 dans le cache 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, comme plus aucune décision ne peut être inversée, la recherche va s'arrêter et l'instance va être considétée incohérente. La figure 4.2 montre l'arbre de recherche obtenu avec cette méthode. On remarque qu'il a une forme de peigne avec toutes les décisions sur une seule branche. Au final, nous avons 5 détections de sous-problèmes similaires pour un total de 7 branches.

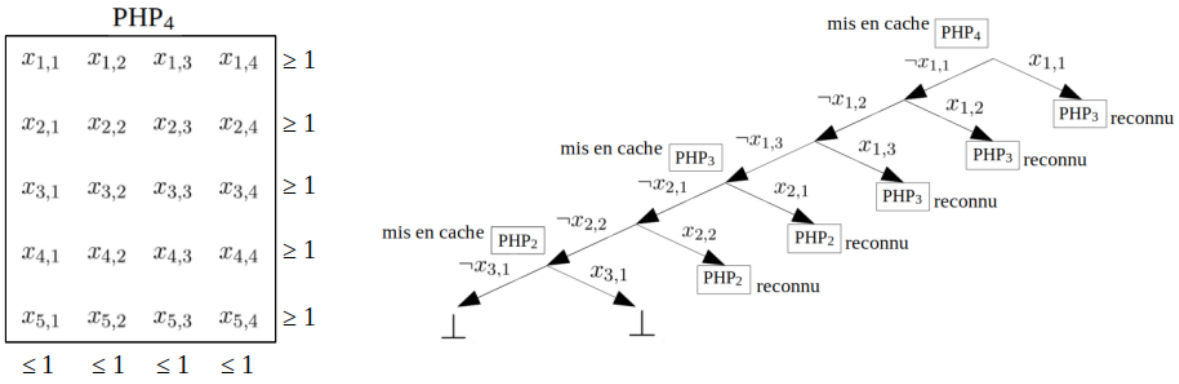


FIGURE 4.2 – Motif de peigne attendu en résolvant le problème PHP_4. Les propagations ont été omises.

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

4.2 Utiliser un cache de formules UNSAT

Pour généraliser le résultat obtenu pour les problèmes de pigeons, nous avons besoin de trouver une façon de détecter qu'une sous-formule donnée a déjà été trouvée précédemment dans l'arbre de recherche. Un compilateur de CNF vers OBDD tel que [HD04] utilise un cache

de sous-formules représentées sous forme d'OBDD pour cette tâche. Les compteurs de modèles utilisent aussi un cache 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 [SBB+04], que nous avons utilisée à un moment de la thèse, assure que deux sous-formules avec les mêmes clauses sont considérées identiques même si elles ne sont pas dans le même ordre ou si elles ne sont pas basées sur les mêmes clauses initiales. Cependant, les systèmes de cache sous forme d'OBDD et des compteurs de modèles ne vont 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 des variables différentes. Il nous faut alors supporter la notion d'égalité modulo un renommage. Il y a aussi un problème spécifique 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 contenant une entrée du cache. Dans ce contexte, le cache ne peut plus être implémenté avec un dictionnaire, comme cela pouvait être le cas par exemple avec la représentation de Cachet. 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 d'isomorphisme de sous-graphe [Coo71] quand nous interrogeons le cache.

Définition 17 (Problème d'isomorphisme de sous-graphe)**.** *Le* problème d'isomorphisme de sous-graphe *correspond au problème de décision suivant :*

— *Entrée : Un graphe appelé* pattern *et un second graphe appelé* target
— *Question : Est-ce que le graphe* pattern *apparaît dans le graphe* target *?*

Pour résoudre ce problème, il faut donc trouver un renommage σ des nœuds du graphe *pattern* vers ceux du graphe *target* qui conserve la structure du graphe *pattern*. Ainsi, pour toute arête (n_1, n_2) présente dans le graphe *pattern* (où n_1 et n_2 sont deux nœuds de ce graphe), alors l'arête $(\sigma(n_1), \sigma(n_2))$ doit être présente dans le graphe *target*. Il est également possible d'attribuer des *étiquettes* aux différents nœuds des graphes considérés et dans ce cas là, il n'est possible de faire correspondre entre eux que des nœuds ayant la même étiquette. Pour pouvoir utiliser ces isomorphismes, nous encodons les formules CNF sous forme de graphes de manière classique : les littéraux 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 4.3 montre un exemple de cette représentation. Chaque couleur est ici représentée par une forme géométrique différente. Par la suite, nous désignerons cette représentation comme l'*encodage classique*.

Il existe plusieurs possibilités pour remplir le cache. La première consiste à ajouter des formules UNSAT au fur et à mesure qu'on les découvre pendant la recherche. Cela correspond au comportement adopté dans notre exemple du problème de pigeons. Par la suite, nous allons principalement utiliser cette approche. Il existe cependant une seconde possibilité qui est basée sur l'utilisation d'un cache pré-rempli (sur le principe d'une bibliothèque d'ouvertures aux échecs). En d'autres termes, avant le début de la recherche, on enregistre des formules UNSAT classiques dans le cache. L'intérêt de cette méthode repose sur le fait que nous pouvons choisir quelles formules vont être ajoutées dans le cache et nous pouvons donc facilement leur attribuer des noms ou des explications prédéfinies. Par exemple, si l'on souhaite faire une détection syntaxique de problèmes de pigeons (en opposition à la détection sémantique présentée précédemment), on peut enregistrer à l'avance des problèmes de pigeons allant de celui de taille 2 et jusqu'à une certaine taille n et ensuite effectuer la recherche en tentant de les reconnaître.

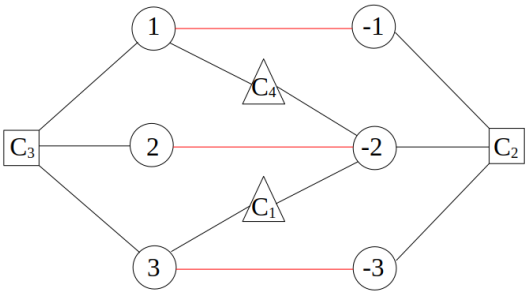


FIGURE 4.3 – Graphe correspondant à $\phi = (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2 \vee \neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2)$. Les littéraux sont représentés par des cercles, les clauses binaires/ternaires par des triangles/carrés. Une arête rouge relie les littéraux opposés.

Cependant, il est très difficile de savoir avant même le début de la recherche quels types de problèmes vont être présents dans une formule donnée et un même problème peut admettre plusieurs encodages différents. Cela rend cette seconde technique particulièrement délicate à mettre en œuvre en pratique. Il est également possible d'imaginer une méthode hybride combinant ces deux approches.

Cache vs symétries

Une autre façon de réduire l'arbre de recherche des problèmes de pigeons est d'ajouter des clauses dont le but est de briser les symétries [CGLR96], soit pendant une étape de pré-traitement [DBBD16], soit de manière paresseuse pendant la recherche [MBCK18, SBDD23]. Il serait également possible de rechercher des symétries de la formule courante à chaque nœud de l'arbre de recherche. Dans tous les cas, une première différence avec notre approche est que les symétries trouvées s'appliquent à la formule complète alors que les entrées du cache que nous identifions correspondent seulement à un sous-ensemble de la formule courante. Pour des formules symétriques, nous nous attendons à ce que les deux méthodes puissent grandement compresser l'arbre de recherche mais nous nous attendons aussi à ce que notre système de cache soit capable de compresser des formules non symétriques. Par exemple, considérons la formule $\phi = PHP_n \cup \phi'$ où la première clause de ϕ' est $C_0 = x_{1,1} \vee x_{1,2} \vee \cdots \vee x_{n+1,n}$ et les $n(n+1)$ autres clauses C_i sont obtenues à partir de C_0 en inversant les i premiers littéraux. ϕ' change le nombre d'occurrences de chaque littéral, ainsi ϕ n'est pas symétrique mais notre système de cache est capable de reconnaître les problèmes de pigeons présents.

Cache vs SDCL

Nous pouvons comparer notre approche à base d'un système de cache avec l'approche SDCL (pour *Satisfaction Driven Clause Learning*) [HKSB17]. Le but de cette approche est de détecter au cours de la recherche si l'exploration de la branche courante peut être arrêtée sans remettre en cause la cohérence de la formule, c'est à dire que si la branche mène à un modèle de la formule, alors il existe une autre partie de l'arbre de recherche qui permet de trouver un modèle. Pour ce faire, pour une formule ϕ une nouvelle formule appelée le *positive redux* est obtenue en ne conservant de ϕ que les clauses satisfaites par l'interprétation courante α et en supprimant de ces clauses tous les littéraux qui ne sont pas assignés dans α. Si cette formule à laquelle on ajoute une clause permettant de bloquer α est cohérente, alors il est possible de trouver une

autre affectation, un autre chemin dans l'arbre, qui permet de satisfaire exactement les mêmes contraintes. SDCL utilise un autre solveur SAT pour faire ce test, et utilise la clause bloquant α pour initier l'analyse de conflit.

Tout comme notre approche, SDCL utilise un oracle NP-complet dans un solveur CDCL avant de prendre une décision. Cependant, SDCL n'a pas pour objet de réduire un arbre de recherche incohérent, mais de permettre l'implémentation d'un nouveau système de preuve, appelé *propagation redundant (PR)*. Ce nouveau système de preuve permet par exemple de résoudre le problème des pigeons efficacement.

Le test SDCL est d'au plus un appel à un oracle NP-complet avant chaque décision. Dans notre cas, nous devons faire cet appel pour chaque élément du cache, qui grossit au cours de la recherche.

4.3 Sources d'incohérence

Un nœud de l'arbre de recherche est identifié par l'interprétation α des variables menant à ce nœud. Quand la sous-formule $\phi_{|\alpha}$ 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 $\phi_{|\alpha}$. En d'autres termes, nous voulons enregistrer un sous-ensemble incohérent de $\phi_{|\alpha}$, 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 ϕ quand elle est UNSAT (ce que l'on nomme un noyau UNSAT [ZM03a]). Toutefois, un tel noyau est généralement donné à la fin de la recherche et il peut correspondre à la formule complète. En revanche, nous devons générer localement un noyau incohérent pour tout nœud de l'arbre de telle sorte que $\phi_{|\alpha}$ 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. Par la suite, nous appellerons *sources* d'une sous-formule incohérente $\phi_{|\alpha}$ les clauses *initiales* de ϕ utilisées par le solveur pour prouver l'incohérence de $\phi_{|\alpha}$. Cet ensemble sera désigné par $Sources(\phi, \alpha)$. 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 18 (Sources)**.** *Nous définissons d'abord la source d'une clause $Sources(C)$. Quand C est une clause initiale, $Sources(C) = \{C\}$. Quand L est une clause apprise, $Sources(L)$ est l'ensemble des clauses initiales de ϕ qui apparaissent dans la dérivation de L par résolution.*

Soit $\phi_{|\alpha}$ une sous-formule incohérente et $\{\alpha_1, \dots \alpha_m\}$ l'ensemble des branches développées par le solveur pour prouver cette incohérence. Chaque $\phi_{|\alpha_j}$ contient un conflit C_j. Nous définissons $S_0(\phi, \alpha_j) = \{C_j\}$ et $S_{i+1}(\phi, \alpha_j) = S_i(\phi, \alpha_j) \cup \{Sources(reason(l))|l \in c \wedge c \in S_i(\phi, \alpha_j) \wedge DL(l) \geq DL(\alpha_j)\}$, où les fonctions $reason(l)$ et $DL(l)$ donnent respectivement la raison de la propagation du littéral l et le niveau de décision de l. Cette séquence admet un point fixe désigné $Sources(\phi, \alpha_j)$. Enfin, les sources $Sources(\phi, \alpha)$ de $\phi_{|\alpha}$ sont définies comme $Sources(\phi, \alpha) = \cup_j Sources(\phi, \alpha_j)$

Par construction, $Sources(\phi, \alpha)_{|\alpha}$ est incohérente car elle contient toutes les clauses initiales utilisées par le solveur pour prouver l'incohérence de $\phi_{|\alpha}$. Nous avons aussi $Sources(\phi, \alpha)_{|\alpha} \subseteq$

$\phi_{|\alpha}$. Par conséquent, $Sources(\phi, \alpha)_{|\alpha}$ est un noyau incohérent de $\phi_{|\alpha}$. Dans un solveur DPLL, $Sources(\phi, \alpha)$ peut être obtenu en collectant les sources des deux nœuds fils $Sources(\phi, \alpha \cup \{l\})$ et $Sources(\phi, \alpha \cup \{\neg l\})$ et en ajoutant les clauses servant à propager à partir de l un littéral apparaissant dans les sources des nœuds fils. 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).

4.4 Intégration dans un solveur DPLL

Dans un premier temps, nous détaillons l'intégration de notre système de cache à l'intérieur d'un solveur DPLL. Cette intégration est résumée par l'algorithme 9.

Algorithm 9 *CacheDPLL*(ϕ, α)

ENTREE : ϕ - une formule CNF, α - l'affectation courante
SORTIE : La formule ϕ est-elle cohérente (SAT) ou incohérente (UNSAT) ?, les sources de ϕ

```
1 : (α, C) ← Propagate(φ, α)
2 : if φ_|α = ∅ then
3 :    return (SAT, ∅)
4 : end if
5 : if C ≠ Undef then
6 :    S ← CollectSourcesConflict(C)
7 :    return (UNSAT, S)
8 : end if
9 : if (H ← HasIsomorphism(φ, α)) ≠ ∅ then
10 :   S ← CollectSourcesIsomorphism(H)
11 :   return (UNSAT, S)
12 : end if
13 : d ← Decide()
14 : (r, S_1) ← DPLL(φ, α ∪ {d})
15 : if r = UNSAT then
16 :    (r, S_2) ← DPLL(φ, α ∪ {¬d})
17 :    if r = UNSAT then
18 :       S ← S_1 ∪ S_2
19 :       AddToCache(S, α)
20 :    else
21 :       S ← ∅
22 :    end if
23 : else
24 :    S ← ∅
25 : end if
26 : return (r, S)
```

Si nous avons montré que la formule est cohérente, alors nous n'avons pas besoin de sources et nous pouvons arrêter la recherche (lignes 2 à 4, 21 et 24). Dans un solveur de type DPLL,

Algorithm 10 *CollectSourcesConflict*(C)

ENTREE : C - une clause de départ
SORTIE : S - les sources collectées sur la branche suite à l'arrivée d'un conflit

```
1 : S ← Sources(C)        // Voir définition 18
2 : for l ∈ C do
3 :    if reason(l) ≠ ∅ and UsedVariable[var(l)] = false then
4 :       UsedVariable[var(l)] ← true
5 :       S ← S ∪ CollectSourcesConflict(reason(l))
6 :    end if
7 : end for
8 : return S
```

Algorithm 11 *CollectSourcesIsomorphism*(H)

ENTREE : H - les clauses initiales identifiées dans un isomorphisme
SORTIE : S - les sources collectées sur la branche suite à la détection d'un isomorphisme

```
1 : S ← ∅
2 : for C ∈ H do
3 :    S ← S ∪ CollectSourcesConflict(C)
4 : end for
5 : return S
```

lorsque les deux fils d'un nœud correspondant à une interprétation α ont été explorés (une branche pour la décision l – ligne 14, une autre pour la décision $\neg l$ – ligne 16) et que les deux sont incohérents (lignes 15 et 17), nous savons que $\phi_{|\alpha}$ est aussi incohérent et les sources $Sources(\phi, \alpha)$ peuvent être obtenues comme présenté précédemment (ligne 18). $Sources(\phi, \alpha)$ simplifié par α est incohérent et ajouté au cache (ligne 19). La fonction $AddToCache(\phi)$ crée une nouvelle entrée dans le cache pour la formule ϕ. Quand un nouveau nœud identifié par l'interprétation α est exploré, la première étape est de vérifier si un conflit a été détecté (ligne 5). Si c'est le cas, nous collectons les sources qui ont mené à ce conflit et on indique que la formule est incohérente (lignes 10 et 11). En pratique, la fonction $CollectSourcesConflict(C)$ (détaillée à l'algorithme 10) collecte récursivement les raisons utilisées pour arriver à la clause C sur l'ensemble de la branche courante. Dans cette fonction, nous supposons disposer d'un tableau *UsedVariable* permettant d'indiquer pour chaque variable du problème si nous l'avons précédemment rencontrée. Les entrées de ce tableau sont initialement affectées à *faux*. Ensuite, nous pouvons vérifier dans le cache s'il y a une entrée E qui est incluse dans la formule $\phi_{|\alpha}$ courante à un renommage de littéraux près (ligne 9). La fonction $HasIsomorphism(\phi, \alpha)$ vérifie si un isomorphisme de sous-graphe peut être détecté entre une entrée du cache et la formule ϕ courante simplifiée par α. S'il existe E dans le cache et σ un renommage des littéraux tel que $\sigma(E) \subseteq \phi_{|\alpha}$, alors $\phi_{|\alpha}$ est nécessairement incohérente puisque E est incohérente. Ce test peut être traduit en problème d'isomorphisme de sous-graphe. Si on en a trouvé un, l'ensemble ϕ' des clauses de $\phi_{|\alpha}$ qui correspondent à des clauses de E est facilement obtenu en faisant correspondre les nœuds aux clauses. $\phi'_{|\alpha}$ est incohérente mais en général ϕ' peut être cohérente. En effet, ϕ' doit être complété avec les clauses nécessaires pour propager les littéraux effacés dans ϕ' au

niveau de décision courant pour obtenir une formule incohérente (ligne 10 - voir exemple 19). Nous pouvons ensuite indiquer que la formule courante est incohérente (ligne 11). En pratique, la fonction $CollectSourcesIsomorphism(H)$ (détaillée à l'algorithme 11) va parcourir les clauses identifiées dans l'isomorphisme, qui sont stockées dans H, et va ensuite lancer une collecte récursive (similaire à ce qui est réalisé lors de la collecte des sources suite à un conflit) à partir de chacune de ces clauses afin de collecter les potentielles raisons manquantes.

Exemple 19. *Prenons par exemple un problème de pigeons P encodé par les clauses $\{C_1, C_2, \ldots, C_n\}$ et considérons la formule ϕ 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 ϕ, 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 ϕ correspondant à P sont $\phi' = \{x \vee C_1, C_2, \ldots, C_n\}$. Si on branche sur $\neg y$, nous obtenons aussi $\phi' = \{x \vee C_1, C_2, \ldots, C_n\}$ de la même manière. Cependant, ϕ' est cohérente car la clause $x \vee C_1$ 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 ϕ' pour obtenir une formule incohérente, qui est la source $Sources(\phi, \emptyset)$ et donc ϕ 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 d'isomorphisme de sous-graphe [Coo71]. Cependant, comme notre objectif n'est pas d'accélérer le temps de résolution mais plutôt de réduire la taille de l'arbre de recherche, nous acceptons de passer beaucoup de temps 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 un retour en arrière. L'idée ici est d'éviter de revenir sur des décisions qui n'ont pas participé au conflit. Comme ces décisions n'ont pas modifié les sources que nous avons récupéré, ces nœuds nous donneraient alors la même entrée à ajouter au cache que l'entrée courante. Nous pouvons donc 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 ϕ 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 $\alpha = \{\neg a, x, y, z\}$. Alors les décisions b et $\neg b$ vont toutes les deux mener à un conflit, ce qui signifie que $\phi_{|\alpha}$ est incohérente de même que $Sources(\phi, \alpha)_{|\alpha} = \{b \vee c, v \vee \neg c, \neg b \vee c, \neg b \vee \neg c\}$. Tant que a est falsifié, ces clauses incohérentes restent dans la formule et nous pouvons donc revenir au niveau de la décision $\neg a$.

À noter

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.

4.5 Intégration dans un solveur CDCL

4.5.1 Cacher des sous-arbres UNSAT

Au cours de la présentation qui suit, nous nous référons à l'algorithme 12, qui décrit l'intégration de notre système de cache dans un solveur CDCL. L'architecture CDCL est actuellement l'architecture état de l'art pour la résolution du problème SAT et il est donc logique d'essayer d'intégrer notre système de cache avec cette architecture. Cependant, comme nous l'avons déjà évoqué, il n'est pas toujours possible de conclure qu'un nœud donné de l'arbre de recherche est UNSAT et cela a une conséquence forte sur la façon dont nous alimentons notre cache car nous ne pouvons ajouter une entrée que quand nous sommes sûrs que la formule courante est UNSAT et que cela a été prouvé par le solveur. Dans un solveur CDCL, cela ne se produit que lorsque nous arrivons à un conflit (ligne 5) ou lorsque nous reconnaissons une entrée du cache (ligne 11). C'est donc à ce moment-là que nous pouvons tenter d'ajouter une nouvelle entrée dans le cache (ligne 22). Si nous avons rencontré un conflit, les sources peuvent être obtenues en effectuant un parcours dans le graphe d'implication depuis la clause conflit (ligne 10). Si nous avons reconnu une entrée E du cache, l'ensemble ϕ' des clauses de $\phi_{|\alpha}$ qui correspondent aux clauses de E peuvent facilement être obtenues en faisant correspondre les nœuds du graphe avec les indices des clauses. Dans un premier temps, nous devons être capable de générer une clause conflit à partir de certains littéraux contenus dans les clauses reconnues par l'isomorphisme (ligne 16). Une fois cette clause conflit générée, il peut être nécéssaire d'effectuer un backjump avant l'analyse de conflit (ligne 18). Nous allons revenir plus en détail sur cette génération de clause conflit et ce potentiel retour en arrière dans les parties suivantes. Ensuite, nous pouvons ajouter à ϕ' les clauses requises pour la propagation des littéraux effacés dans ϕ' au niveau de décision courant (ligne 17). Comparé à l'analyse de conflit classique, les informations contenues dans les clauses apprises sont ici exprimées en termes de clauses initiales. D'une certaine façon, les sources que nous collectons « déplient » les clauses apprises en clauses initiales. En pratique, la taille des sources trouvées augmente à mesure que la recherche progresse. Nous créons des entrées de cache en collectant les sources des feuilles que l'on simplifie ensuite en utilisant les décisions courantes et la propagation unitaire. Cependant, si l'on considère toutes les décisions et propagations réalisées jusque là, comme nous nous trouvons au niveau d'une feuille, nous allons directement retrouver un conflit. Nous choisissons donc d'exclure les littéraux propagés par des clauses apprises afin d'éviter cette situation. De plus, si la formule obtenue contient une clause unitaire (ce qui peut arriver si une propagation pouvait à la fois se faire grâce à une clause apprise ou une clause initiale mais que le solveur a utilisé la clause apprise), nous ne créons pas d'entrée de cache pour cette formule puisqu'elle ne pourra jamais être reconnue. En effet, les vérifications dans le cache sont réalisées avant de prendre une décision et donc à la fin d'une propagation unitaire. Donc les formules que nous créons ne contiennent pas de clause unitaire.

4.5.2 Intégrer le cache avec l'analyse de conflit

Si nous ne souhaitons pas utiliser le système de cache en post-traitement mais plutôt à l'intérieur de la recherche elle-même afin d'éviter certaines explorations coûteuses, nous devons alors élaguer la branche courante dès lors qu'une entrée est reconnue. Un problème va alors provenir du fait que les solveurs CDCL ont besoin d'une clause falsifiée pour démarrer l'analyse de conflit. Nous devons donc pouvoir exprimer un hit sous forme d'un ensemble de littéraux falsifiés pour permettre cette analyse de conflit. Une première solution consiste simplement à

Algorithm 12 *CacheCDCL*(ϕ)

ENTREE : ϕ - une formule CNF
SORTIE : La formule ϕ est-elle cohérente (SAT) ou incohérente (UNSAT) ?

```
1 : α ← ∅
2 : while true do
3 :    conflictFound ← false
4 :    (α, C) ← Propagate(φ, α)
5 :    if C ≠ Undef then
6 :       if CurrentDecisionLevel() = 0 then
7 :          return UNSAT
8 :       end if
9 :       conflictFound ← true
10 :      S ← CollectSourcesConflict(C)
11 :   else if (H ← HasIsomorphism(φ, α)) ≠ ∅ then
12 :      if CurrentDecisionLevel() = 0 then
13 :         return UNSAT
14 :      end if
15 :      conflictFound ← true
16 :      C ← CreateConflictFromIsomorphism(H, α)
17 :      S ← CollectSourcesIsomorphism(H)
18 :      α ← BackjumpIfNeeded(C, α)
19 :   end if
20 :   if conflictFound = true then
21 :      C1 ← AnalyzeConflict(C)
22 :      AddToCache(S, α)
23 :      φ ← φ ∪ {C1}
24 :      α ← Backjump(C1, α)
25 :   else
26 :      if needRestart then
27 :         Restart()
28 :      end if
29 :      if needCleanDB then
30 :         CleanDB()
31 :      end if
32 :      l ← Decide()
33 :      if l = Undef then
34 :         return SAT
35 :      end if
36 :      α ← α ∪ {l}
37 :   end if
38 : end while
```

créer une clause falsifiée par toutes les décisions présentes dans le chemin depuis la racine jusqu'à la feuille courante. Cependant, cette approche n'a pas donné de bons résultats. Cette méthode est particulièrement inutile pour l'analyse de conflit car aucun littéral présent dans la clause créée n'a de raison. De cette manère, la clause peut contenir des littéraux non nécessaires au conflit.

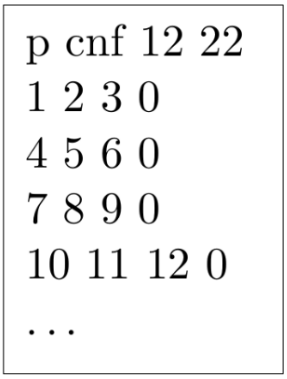
Nous proposons une approche alternative qui prend avantage de la notion de sources. Lorsque nous avons collecté les clauses initiales O correspondant à une entrée de cache, nous pouvons créer une clause composée de tous les littéraux falsifiés présents dans O. Une telle clause est par construction falsifiée par l'affectation courante. Par ailleurs, nous montrons que cette clause que nous générons est impliquée par la formule initiale et qu'il s'agit bien d'un conflit. En effet, nous savons que $O_{|\alpha} \subseteq \phi_{|\alpha}$ est incohérente, donc $O \wedge \alpha \models \bot$. Supposons que l'on appelle $SELECT(O, \alpha)$ les littéraux de α qui falsifient un littéral dans une clause de O. Donc, nous avons aussi $O \wedge SELECT(O, \alpha) \models \bot$ ou $O \models \neg SELECT(O, \alpha)$ puisque les littéraux de α satisfaisant des clauses de O ne contribuent pas à l'incohérence. Ainsi, utiliser $\neg SELECT(O, \alpha)$ comme clause conflit quand une entrée de cache est détectée est correct.

À titre d'exemple, considérons le problème PHP_3 et une heuristique qui décide négativement les variables en ordre inverse d'indice. La figure 4.4 illustre le comportement de cette heuristique. L'heuristique va commencer par décider $\neg x_{4,3}$ et ensuite $\neg x_{4,2}$. Cela va propager $x_{4,1}$ par l'intermédiaire de la clause $x_{4,1} \vee x_{4,2} \vee x_{4,3}$. Les exclusions mutuelles vont également propager $\neg x_{1,1}$, $\neg x_{2,1}$ et $\neg x_{3,1}$. À ce moment là, nous arrivons au problème PHP_2. Après l'avoir exploré, nous allons l'enregistrer dans le cache et le solveur va apprendre la clause unitaire $\neg x_{4,1}$, qui va nous faire revenir au début de l'arbre de recheche. Après avoir propagé $\neg x_{4,1}$, l'heuristique va de nouveau décider $\neg x_{4,3}$. La clause $x_{4,1} \vee x_{4,2} \vee x_{4,3}$ va cette fois-ci propager $x_{4,2}$ et les exclusions mutuelles $\neg x_{1,2} \vee \neg x_{4,2}$, $\neg x_{2,2} \vee \neg x_{4,2}$ et $\neg x_{3,2} \vee \neg x_{4,2}$ vont respectivement en déduire $\neg x_{1,2}$, $\neg x_{2,2}$ et $\neg x_{3,2}$. Nous retrouvons alors le problème PHP_2, que nous allons reconnaître grâce à celui que nous avons précédemment stocké. Les clauses $x_{1,1} \vee x_{1,2} \vee x_{1,3}$, $x_{2,1} \vee x_{2,2} \vee x_{2,3}$ et $x_{3,1} \vee x_{3,2} \vee x_{3,3}$ ainsi que certaines exclusions mutuelles vont correspondre à l'entrée du cache. Dans ces clauses, nous savons que les littéraux $x_{1,2}$, $x_{2,2}$ et $x_{3,2}$ ont été falsifiés. Nous pouvons donc créer la clause $x_{1,2} \vee x_{2,2} \vee x_{3,2}$, qui va être donnée en entrée de l'analyse de conflit. En effectuant des résolutions sur les exclusions mutuelles $\neg x_{1,2} \vee \neg x_{4,2}$, $\neg x_{2,2} \vee \neg x_{4,2}$ et $\neg x_{3,2} \vee \neg x_{4,2}$, nous en déduisons la clause unitaire $\neg x_{4,2}$ que nous allons pouvoir apprendre. Avec cette clause apprise, nous pouvons revenir au niveau de décision 0, propager $\neg x_{4,2}$ et continuer la recherche.

Il est tout de même à noter que nous pouvons nous trouver dans le cas où la clause conflit que l'on crée ne contient aucun littéral du niveau de décision courant, ce qui brise un invariant classique des solveurs CDCL. Cela peut arriver lorsque la décision courante n'est pas liée au hit du cache et cela peut signifier qu'une entrée du cache que l'on vient de créer aurait pu être détectée plus tôt dans l'arbre de recherche si l'entrée du cache avait été créée auparavant. Ce cas peut poser problème dans le fonctionnement du solveur. Le problème vient ici du fait que l'on n'a aucun littéral du niveau de décision courant et donc on ne peut pas appliquer l'analyse de conflit qui nécessite d'avoir au moins un littéral du niveau de décision courant (ce qui garantie un UIP).

Exemple 20. *À titre d'exemple, nous allons considérer une formule ϕ constituée des deux sous-formules $\phi_1 = \{x_1 \vee x_2 \vee x_7 \vee x_8, x_1 \vee \neg x_2, \neg x_1 \vee x_2, \neg x_1 \vee \neg x_2, x_7 \vee \neg x_8, \neg x_7 \vee x_8, \neg x_7 \vee \neg x_8\}$ et $\phi_2 = \{x_3 \vee x_4 \vee x_5 \vee x_6, x_3 \vee \neg x_4 \vee x_5 \vee x_6, \neg x_3 \vee x_4 \vee x_5 \vee x_6, \neg x_3 \vee \neg x_4 \vee x_5 \vee x_6\}$ (soit $\phi = \phi_1 \cup \phi_2$). De plus, nous considérons une heuristique qui décide négativement les variables dans l'ordre inverse.*

L'heuristique va donc commencer par décider $\neg x_8$, ce qui va permettre de propager $\neg x_7$



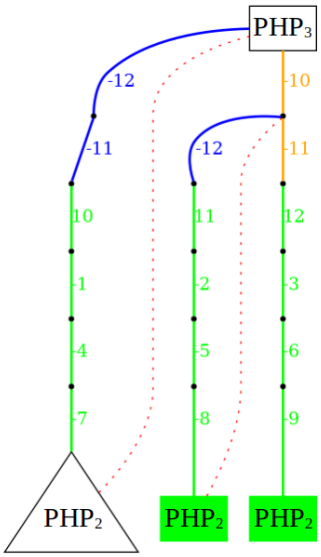


FIGURE 4.4 – Exemple pour illustrer l'intégration du système de cache avec l'analyse de conflit. Une flèche bleue indique une décision, une flèche noire représente une propagation par clause initiale et une flèche rouge représente une propagation par clause apprise.

grâce à la clause $\neg x_7 \vee x_8$. *À ce moment-là, la sous-formule* ϕ_1 *va se résumer à l'ensemble de clauses* $\{x_1 \vee x_2, x_1 \vee \neg x_2, \neg x_1 \vee x_2, \neg x_1 \vee \neg x_2\}$. *L'heuristique va ensuite prendre successivement les décisions* $\neg x_6$, $\neg x_5$ *et* $\neg x_4$. *La clause* $x_3 \vee x_4 \vee x_5 \vee x_6$ *nous oblige alors à propager* x_3. *Nous obtenons alors un conflit avec la clause* $\neg x_3 \vee x_4 \vee x_5 \vee x_6$. *L'analyse de conflit ne va réaliser qu'une seule résolvante entre la clause conflit et la clause* $x_3 \vee x_4 \vee x_5 \vee x_6$, *ce qui va nous permettre d'apprendre la clause* $x_4 \vee x_5 \vee x_6$. *Ces deux clauses utilisées dans l'étape de résolution vont également constituer les sources de la clause apprise. Ici, aucune mise en cache ne sera effectuée puisque la simplification par l'affectation courante va nous mener trivialement à un conflit. Nous revenons donc au niveau de la décision* $\neg x_5$ *et nous allons propager* x_4 *via la clause apprise. Cette propagation va également nous permettre de propager* x_3 *grâce à la clause* $x_3 \vee \neg x_4 \vee x_5 \vee x_6$. *La clause* $\neg x_3 \vee \neg x_4 \vee x_5 \vee x_6$ *devient alors conflictuelle et une nouvelle analyse de conflit va être démarrée. Nous allons partir de la clause conflit et effectuer des résolvantes avec d'abord la clause* $x_3 \vee \neg x_4 \vee x_5 \vee x_6$ *puis la clause apprise* $x_4 \vee x_5 \vee x_6$. *Cela va nous permettre d'apprendre la clause* $x_5 \vee x_6$. *Lors de la collecte des sources, nous devons remplacer la clause apprise* $x_4 \vee x_5 \vee x_6$ *par ses sources et nous collectons donc les sources* $\{x_3 \vee x_4 \vee x_5 \vee x_6, x_3 \vee \neg x_4 \vee x_5 \vee x_6, \neg x_3 \vee x_4 \vee x_5 \vee x_6, \neg x_3 \vee \neg x_4 \vee x_5 \vee x_6\}$. *Si l'on simplifie ces clauses par l'interprétation courante en excluant les littéraux propagés par clause apprise, nous ne prenons en compte que les décisions* $\neg x_6$ *et* $\neg x_5$ *pour la simplification. Nous obtenons alors* $\{x_3 \vee x_4, x_3 \vee \neg x_4, \neg x_3 \vee x_4, \neg x_3 \vee \neg x_4\}$ *que nous pouvons ajouter au cache. Nous pouvons maintenant revenir au niveau de la décision* $\neg x_6$ *et propager* x_5, *ce qui va satisfaire toutes les clauses de* ϕ_2.

Ici, lors de l'étape de vérification dans le cache, nous pouvons détecter un isomorphisme de sous-graphe avec l'entrée du cache précédemment créée. Les clauses concernées par cette reconnaissance sont les clauses $\{x_1 \vee x_2 \vee x_7 \vee x_8, x_1 \vee \neg x_2, \neg x_1 \vee x_2, \neg x_1 \vee \neg x_2\}$. *Si nous souhaitons compléter ces sources, nous remarquons qu'il ne nous manque que la raison de la propagation de* $\neg x_7$ *et nous pouvons donc ajouter la clause* $\neg x_7 \vee x_8$ *aux sources. Concernant*

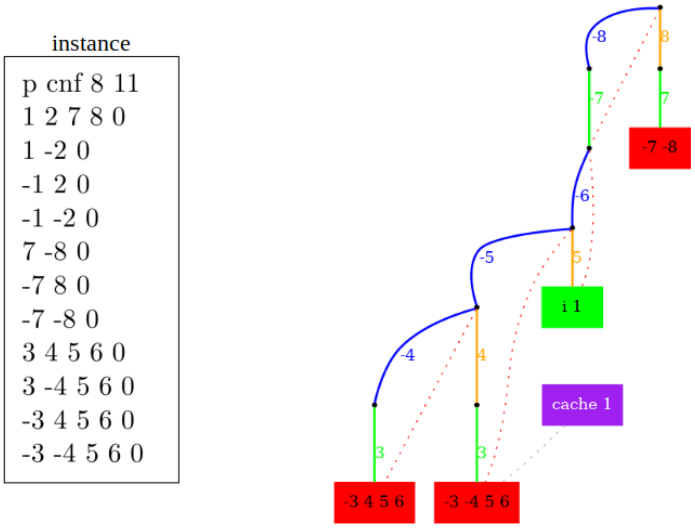


FIGURE 4.5 – Exemple d'arbre de recherche avec le système de cache intégré dans un solveur CDCL.

maintenant l'analyse de conflit, parmi les clauses reconnues, nous remarquons que seuls les littéraux x_8 et x_7 ont dû être simplifiés pour pouvoir reconnaître l'entrée du cache. Nous pouvons donc créer la clause conflit $x_7 \vee x_8$. Or, si nous démarrons une analyse de conflit, maintenant, celle-ci va s'arrêter directement et nous redonner la clause $x_7 \vee x_8$ car aucune de ses variables n'a été affectée au niveau de décision courant. Cela nous fait revenir au niveau de la décision $\neg x_8$ et nous devons alors propager x_7. Or, cette variable a déjà été affectée. Pour corriger cela, nous devons effectuer l'analyse de conflit juste après la décision $\neg x_8$ et la propagation $\neg x_7$ (i.e. le moment le plus haut dans l'arbre où la détection d'isomorphisme était possible). Cette fois-ci, l'analyse de conflit va effectuer une résolvante entre la clause conflit $x_7 \vee x_8$ et la clause $\neg x_7 \vee x_8$, ce qui va nous permettre d'apprendre la clause unitaire x_8. Nous allons donc revenir au niveau de décision 0, propager x_8 et continuer la recherche. La figure 4.5 montre l'arbre de recherche obtenu sur cet exemple.

Avec ce que nous venons de voir, nous pouvons donner le code de certaines fonctions introduites dans l'algorithme 12. Tout d'abord, la fonction *CreateConflictFromIsomorphism*(H, α) (détaillée à l'algorithme 13) va juste parcourir les clauses reconnues dans l'isomorphisme et récupérer les littéraux falsifiés dans celles-ci. Ensuite, la fonction *BackjumpIfNeeded*(C, α) (détaillée à l'algorithme 14) compare le niveau de décision le plus élevé pouvant être trouvé dans la clause conflit précédemment générée au niveau de décision courant. Si le niveau de décision de la clause conflit est plus petit, alors nous pouvons effectuer un backjump pour revenir à ce niveau.

4.6 Résultats expérimentaux pour les isomorphismes classiques

4.6.1 Environnement

Notre système de cache a été implémenté à l'intérieur de MiniSat [ES03]. Nous avons tout de même dû apporter quelques modifications à ce solveur afin de faciliter son intégration. Tout d'abord, nous avons désactivé les simplifications de la base de clauses afin de conserver l'intégralité des clauses présentes dans la formule au moment de la première décision. Cela vient du

Algorithm 13 *CreateConflictFromIsomorphism*(H, α)

ENTREE : H - les clauses initiales identifiées dans un isomorphisme, α - l'affectation courante
SORTIE : C - la clause conflit générée

```
1 : conflict ← ∅
2 : for C ∈ H do
3 :    for l ∈ C do
4 :       if α(literal) = false then
5 :          conflict ← conflict ∪ {l}
6 :       end if
7 :    end for
8 : end for
9 : return conflict
```

Algorithm 14 *BackjumpIfNeeded*(C, α)

ENTREE : C - la clause conflit générée, α - l'affectation courante
SORTIE : α - l'affectation éventuellement mise à jour

```
1 : if DL(C) < CurrentDecisionLevel() then
2 :    α ← BackjumpUntil(DL(C), α)
3 : end if
4 : return α
```

fait que les sources peuvent contenir certaines clauses pouvant être éliminées et donc nous avons besoin de ces clauses lors de la création d'une entrée de cache. Concernant maintenant l'approche CDCL en particulier, nous avons également désactivé les redémarrages qui construisent une séquence d'arbres de recherche. Pour l'approche DPLL, nous nous sommes basés sur le code de MiniSat dans lequel nous avons désactivé l'analyse de conflit et l'apprentissage de clauses. Dans tous les cas, la collecte des sources est réalisée à l'intérieur d'une procédure dédiée. Dans cette dernière, nous pouvons par exemple mettre à jour l'activité d'une variable lorsqu'un de ses littéraux a été propagé par une clause qui n'est pas encore dans nos sources et que nous devons donc ajouter. Cette nouvelle mise à jour d'activité peut éventuellement se cumuler avec celle de l'analyse de conflit de la version CDCL. Nous avons choisi d'utiliser le Glasgow Subgraph Solver (abrégé en GSS) [MPT20] pour calculer des isomorphismes de sous-graphes et donc interroger notre cache. Avant chaque décision du solveur, si le cache n'est pas vide, nous traduisons cette étape de vérification dans le cache en problèmes d'isomorphimes de sous-graphes que nous donnons ensuite au GSS. Lorsque les isomorphismes classiques sont utilisés, nous avons imposé un temps limite de 2 secondes pour chaque appel au GSS et de 15 minutes pour résoudre chaque instance (ce qui inclut le temps d'utilisation du GSS).

4.6.2 Instances de problèmes de pigeons

Tout d'abord, nous nous sommes intéressés au pouvoir de compression de notre approche sur des problèmes de pigeons. 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 et c'est pourquoi nous simulons le post-traitement directement dans le solveur. Des résultats individuels sur les approches DPLL et CDCL peuvent être trouvés dans la table 4.1. Pour ce type d'expérimentation, nous fournissons pour chaque instance sa taille en nombre de littéraux, et le nombre de conflits trouvés sans et avec cache pour les deux approches DPLL et CDCL. Nous comparons les nombres de conflits, donc de branches, des deux arbres de recherche. Le rapport de compression est le nombre de conflits avec cache divisé par le nombre de conflits sans cache.

Pour l'approche DPLL, le post-traitement s'est comporté principalement comme attendu pour les problèmes de pigeons et nous avons obtenu des peignes pour ces problèmes. Un exemple est donné à la figure 4.6 pour le problème PHP_4 avec l'approche DPLL. Sur cette figure, 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. L'étiquette « i x » (pour « isomorphisme x ») signifie que l'élément enregistré à « cache x » a été reconnu. La seule différence vient de l'heuristique utilisée dans MiniSat, qui décide négativement la première variable et décide ensuite 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 1 comparé au nombre de branches attendu.

Nous avons ensuite utilisé le cache pendant la recherche elle-même. Un extrait des expérimentations sur les problèmes de pigeons peut être trouvé dans la table 4.2 pour l'approche DPLL et dans la table 4.3 pour l'approche CDCL. Pour ce type d'expérimentation, nous donnons pour chaque instance les nombres de conflits, d'entrées créées 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

TABLE 4.1 – Résultats expérimentaux concernant le pouvoir de compression de notre approche avec les isomorphismes classiques sur les problèmes de pigeons.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport	Conflits (sans cache)	Conflits (cache)	Rapport
PHP_5	180	168	12	$7.1\ 10^{-2}$	151	47	$3.1\ 10^{-1}$
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	-	-	-	-	-	-

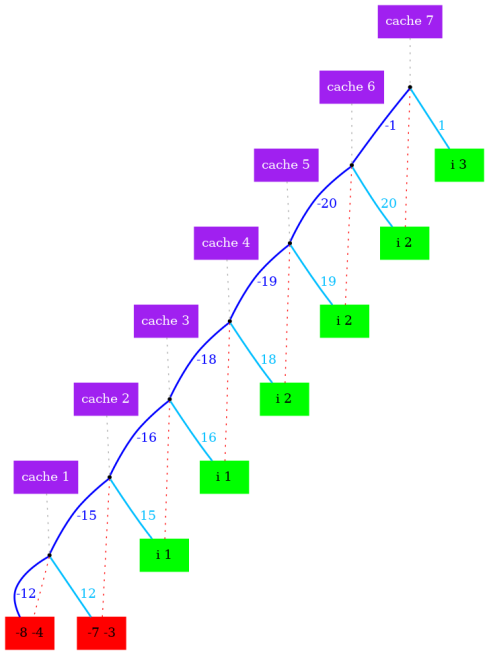


FIGURE 4.6 – Arbre de recherche pour le problème PHP_4 quand le cache est utilisé avec un solveur DPLL. Les propagations depuis des clauses initiales ont été omises.

(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.

TABLE 4.2 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur DPLL avec les isomorphismes classiques sur les instances de problèmes de pigeons.

	DPLL (cache intégré, isomorphismes classiques)						
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Abandons	Temps (recherche)	Temps (GSS)
PHP_5	12	11	10 (4)	10	0	0.012	0.072
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

Concernant les expérimentations sur le pouvoir de compression de notre approche, nous sommes limités dans la résolution d'instances de problèmes de pigeons car nous devons tout de

TABLE 4.3 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur CDCL avec les isomorphismes classiques.

	CDCL (cache intégré, isomorphismes classiques)					
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Temps (recherche)	Temps (GSS)
PHP_5	26	19	8 (2)	88	0.022	0.473
PHP_7	47	41	29 (8)	259	0.025	1.550
PHP_{12}	116	107	96 (20)	589	0.189	18.412
PHP_{16}	187	178	167 (32)	1020	0.731	166.088

même explorer l'arbre de recherche complet, qui est de taille exponentielle pour ces instances, et ce même si des branches peuvent être coupées suite à la reconnaissance d'une entrée de cache. En intégrant le cache directement dans le solveur, nous pouvons arrêter la recherche sur la branche courante dès qu'une entrée de cache est reconnue et cela nous permet de résoudre plus d'instances. Par contre, comme les sous-formules et entrées de cache peuvent devenir très grandes, dès que nous considérons des instances de pigeons plus grandes que celle de taille 16, nous avons une partie des appels au GSS qui atteint le temps limite de 2 secondes. Cela explique que nous n'arrivons pas à résoudre ces dernières instances dans le temps limite de 15 minutes.

4.6.3 Un autre exemple : les échiquiers mutilés

Nous allons maintenant nous intéresser à un autre exemple d'instances difficiles pour les solveurs SAT, à savoir les instances d'*échiquiers mutilés*. Ces instances étaient notamment présentes à la compétition SAT de 2018 [HJS19] et il a été montré que les solveurs SAT développent en général un arbre de recherche de taille exponentielle sur ces types de problèmes [Ale04]. Le principe de ce problème est le suivant : nous disposons d'un échiquier de taille $n \times n$ dont deux carrés placés sur les coins opposés d'une même diagonale ont été supprimés et nous souhaitons savoir s'il est possible de recouvrir l'échiquier ainsi obtenu avec des dominos de taille 2×1. Ce type de problème est incohérent et une explication simple consiste à indiquer que le nombre de cases noires et le nombre de cases blanches présentes sur l'échiquier sont différents. En effet, comme chaque domino va recouvrir exactement une case blanche et une case noire, nous devons alors disposer du même nombre de cases noires et de cases blanches si l'on veut pouvoir recouvrir la totalité de l'échiquier. Or ici ce n'est pas le cas car nous disposons de m cases de la même couleur que la diagonale dans laquelle les coins opposés ont été supprimés et $m + 2$ cases de l'autre couleur. Donc le problème est UNSAT. Cependant, il n'est pas possible d'obtenir une telle explication avec un solveur SAT puisque d'une part l'encodage ne donne pas ces informations et d'autre part un solveur SAT ne sait utiliser des arguments de comptage.

Au niveau de l'encodage, il faut créer une variable pour chaque frontière entre deux cases de l'échiquier. Si une de ces variables est affectée à vrai, cela signifie qu'un domino a été placé sur les deux cases correspondantes. Il faut faire en sorte que chaque case soit couverte par exactement un domino. Nous devons donc créer, pour chaque case, une contrainte $= 1$ avec les variables concernées par la case considérée. Ainsi, nous créons une clause contenant toutes ces variables pour la partie ≥ 1 et toutes les exclusions mutuelles pour la partie ≤ 1. Cela nous fait un total de deux clauses pour chaque case dans les coins, quatre clauses pour chaque case sur les bords et sept clauses pour chaque case interne. Par ailleurs, il a été montré qu'il était possible de légèrement simplifier cet encodage [dKvMW00]. En effet, nous pouvons nous contenter de la contrainte ≤ 1 pour les cases de la même couleur que la diagonale et de la contrainte ≥ 1 pour

les autres cases, ce qui permet de supprimer approximativement la moitié des clauses.

Pour ces instances, nous avons remarqué qu'il était possible à partir d'un échiquier mutilé de taille n de retrouver un échiquier mutilé de taille $n-2$ mais pas de taille $n-1$. Par exemple, la figure 4.7 montre des placements de dominos pour le problème d'échiquier mutilé de taille 5 permettant de retrouver un problème de taille 3. De plus, retrouver un sous-problème demande d'affecter positivement plusieurs variables différentes alors que, pour les problèmes de pigeons, nous n'avions besoin d'affecter positivement qu'une seule variable pour retrouver un sous-problème de pigeons. Il semble donc très peu probable de retrouver un arbre de décision avec une forme de peigne comme précédemment avec les problèmes de pigeons. Cependant, nous remarquons qu'il est tout de même possible de trouver des échiquiers équivalents puisque certaines configurations de dominos peuvent être placées de diverses manières. Par exemple, si l'on a deux dominos placés horizontalement, nous pouvons aussi très bien les placer verticalement.

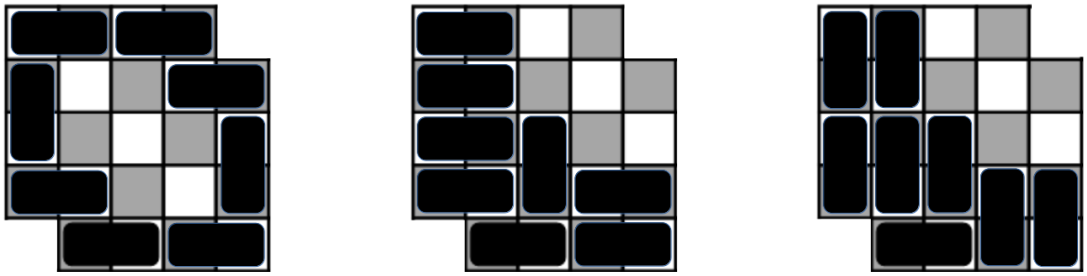


FIGURE 4.7 – Placements de dominos permettant de retrouver un problème d'échiquier mutilé plus petit.

Au niveau des expérimentations, nous avons supprimé la case supérieure droite ainsi que la case inférieure gauche. Des résultats pour certaines instances d'échiquier mutilé peuvent être observés dans la table 4.4 pour l'approche avec élagage après la recherche et dans les tables 4.5 et 4.5 pour les approches avec élagage pendant la recherche. L'heuristique de MiniSat va d'abord attribuer une valeur à la variable 1, qui dans notre cas correspond à la frontière entre les deux premières cases dans le coin supérieur gauche. Ainsi, affecter cette variable négativement ou positivement va respectivement correspondre au fait de poser un domino verticalement ou horizontalement dans le coin supérieur gauche de l'échiquier. Les deux configurations ainsi obtenues sont symétriques et cela nous permet de détecter un isomorphisme très tôt dans l'arbre de recherche. Cette détection nous permet d'élaguer approximativement la moitié de l'arbre. Cependant, comme la détection de problèmes plus petits est plus complexe qu'avec les problèmes de pigeons, nous résolvons moins d'instances. Ici, nous ne parvenons pas à résoudre des problèmes d'échiquiers mutilés plus grands que celui de taille 6.

TABLE 4.4 – Résultats expérimentaux concernant le pouvoir de compression de notre approche avec les isomorphismes classiques sur les instances d'échiquiers mutilés.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport	Conflits (sans cache)	Conflits (cache)	Rapport
mChess_5	240	45	15	$3.3\ 10^{-1}$	30	16	$5.3\ 10^{-1}$
mChess_6	388	105	35	$3.3\ 10^{-1}$	44	23	$5.2\ 10^{-1}$
mChess_7	568	-	-	-	473	202	$4.3\ 10^{-1}$

TABLE 4.5 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur DPLL avec les isomorphismes classiques sur les instances d'échiquiers mutilés.

	DPLL (cache intégré, isomorphismes classiques)						
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Abandons	Temps (recherche)	Temps (GSS)
mChess_5	13	12	5 (3)	46	0	0.008	0.261
mChess_6	27	25	14 (6)	287	0	0.026	2.341
mChess_7	-	-	-	-	-	-	-

TABLE 4.6 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur CDCL avec les isomorphismes classiques sur les instances d'échiquiers mutilés.

	CDCL (cache intégré, isomorphismes classiques)					
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Temps (recherche)	Temps (GSS)
mChess_5	15	9	2 (2)	38	0.010	0.222
mChess_6	23	17	1 (1)	75	0.014	0.774
mChess_7	218	123	8 (5)	11,197	0.677	152.480

4.6.4 Instances des compétitions SAT'02 et SAT'03

Concernant la suite des expérimentations en, nous avons sélectionné un total de 580 instances UNSAT provenant des compétitions SAT'02 (partie *submitted*) [SLH05] et SAT'03 (parties *hand-made* et *industrial*) [LS03]. Étant donné que nous résolvons régulièrement des problèmes d'isomorphismes de sous-graphes, qui sont NP-complets [Coo71], notre approche a une complexité de calcul élevée. Nous avions donc besoin d'instances « faciles » à résoudre pour MiniSat et c'est pourquoi nous avons sélectionné ces deux compétitions.

Élagage après la recherche

Dans cette section, nous nous intéressons au potentiel de compression de notre approche sur les instances des compétitions. Certains résultats individuels de ces expérimentations sur quelques familles d'expérimentations peuvent être trouvés dans la table 4.7. Une distribution des rapports obtenus par les deux approches peut être trouvée dans la table 4.8.

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 ces familles, nous avons souvent obtenu un arbre de recherche avec une forme de peigne comme présenté précédemment. 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 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 4.8 montre l'arbre de recherche obtenu par notre approche quand le cache est utilisé. Nous donnons ici l'arbre de recherche obtenu avec l'approche CDCL mais nous obtenons un arbre semblable avec l'approche DPLL.

TABLE 4.7 – Résultats expérimentaux concernant le pouvoir de compression de notre approche avec les isomorphismes classiques sur les instances des compétitions.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport	Conflits (sans cache)	Conflits (cache)	Rapport
marg2x6	528	$5.2\ 10^5$	21	$4.0\ 10^{-5}$	$3.0\ 10^4$	20	$6.6\ 10^{-4}$
marg3x3add8	1,056	-	-	-	$1.8\ 10^5$	32	$1.8\ 10^{-4}$
Urquhart-s3-b9	1,240	$5.2\ 10^5$	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	30	18	$6.0\ 10^{-1}$	27	27	1
3col40_5_4	1,286	413	122	$3.0\ 10^{-1}$	92	64	$7.0\ 10^{-1}$
3col60_5_1	1,926	-	-	-	826	498	$6.0\ 10^{-1}$
homer06	1,800	$5.0\ 10^5$	214	$4.3\ 10^{-4}$	-	-	-
homer07	2,178	$5.1\ 10^5$	35	$6.9\ 10^{-5}$	-	-	-
homer17	3,718	-	-	-	-	-	-
ca004	426	218	65	$3.0\ 10^{-1}$	43	43	1
ca008	940	-	-	-	144	144	1
dp03u02	2,376	63	58	$9.3\ 10^{-1}$	26	26	1
dp04u03	5,688	-	-	-	70	70	1
BMC (4)	1122	$1.7\ 10^4$	239	$1.4\ 10^{-2}$	38	38	1
BMC (23)	1122	$1.6\ 10^4$	278	$1.8\ 10^{-2}$	38	36	$9.5\ 10^{-1}$
BMC (61)	2,142	-	-	-	72	65	$9.0\ 10^{-5}$
ezfact16_2	4,089	365	300	$8.2\ 10^{-1}$	37	37	1
ezfact16_4	4,089	236	53	$2.2\ 10^{-1}$	252	252	1
ezfact16_5	4,089	105	37	$3.5\ 10^{-1}$	50	45	$0.9\ 10^{-5}$
rope_0001	180	26	11	$4.2\ 10^{-1}$	30	23	$7.7\ 10^{-1}$
rope_0002	360	248	15	$6.0\ 10^{-2}$	162	77	$4.8\ 10^{-1}$
rope_0003	540	-	-	-	795	795	-
grid_05_05	298	-	-	-	428	324	$7.6\ 10^{-1}$

TABLE 4.8 – Distribution des rapports pour les techniques de post-traitement pour les approches DPLL et CDCL sur les instances SAT'02 et SAT'03 lorsque les isomrophismes classiques sont utilisés.

Rapport	Non résolu	$[1\,;0.75[$	$[0.75\,;0.5[$	$[0.5\,;0.25[$	$[0.25\,;10^{-1}[$	$[10^{-1}\,;10^{-2}[$	$\leq 10^{-2}$
DPLL	521	5	6	12	11	5	20
CDCL	463	38	13	5	3	6	52

Élagage pendant la recherche

Nous avons ensuite utilisé le cache pendant la recherche elle-même. Un extrait pertinent des résultats avec les isomorphismes classiques est donné dans la table 4.9 pour l'approche DPLL ainsi que dans la table 4.10 pour l'approche CDCL. 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 l'intégration du cache directement dans le solveur permet de 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 sur la branche courante dès qu'une entrée du cache est détectée

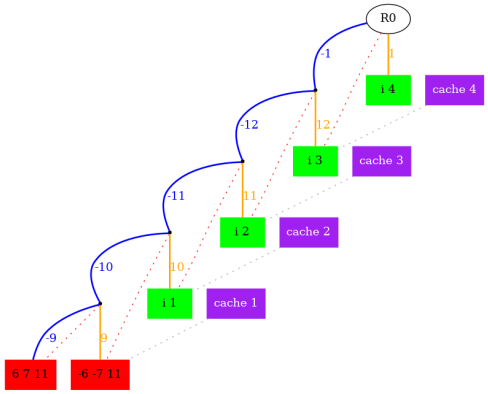


FIGURE 4.8 – Arbre de recherche pour l'instance `marg2x3.cnf` quand le cache est utilisé avec un solveur CDCL. Les propagations depuis des clauses initiales ont été omises.

permet de résoudre bien plus d'instances que le DPLL ou CDCL 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 et entrées de cache 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 peuvent être interrompus et il se peut que nous passions à côté de certains isomorphismes, et donc de potentielles compressions.

Cache pré-rempli avec des problèmes de pigeons

Nous avons également tenté une approche syntaxique de détection de problèmes de pigeons en utilisant un cache pré-rempli. Pour ce faire, avant le début de la recherche, nous remplissons le cache avec un certain nombre de problèmes de pigeons. Dans notre cas, nous avons considéré les problèmes de pigeons de tailles 2 à 64. Comme nous ne souhaitons détecter que des problèmes de pigeons, nous ne faisons plus de mise en cache une fois la recherche d'une solution commencée. Pour ces expérimentations, nous avons utilisé l'approche CDCL avec le cache intégré. Par ailleurs, nous avons considéré les mêmes instances que précédemment et nous avons utilisé les mêmes limites de temps (i.e. 2 secondes pour chaque appel au GSS et 15 minutes par instance). Une partie des résultats peut être consultée dans la table 4.11. Pour les instances de la famille `homer` et une instance de routage nous arrivons à détecter un problème de pigeons à la racine de l'arbre de recherche, ce qui permet de conclure directement que l'instance est incohérente et donc d'arrêter la recherche. En dehors de ces instances, il est rare de détecter des problèmes de pigeons dans les instances des compétitions SAT'02 et SAT'03 et lorsque cela arrive, nous ne détectons que des problèmes de petite taille.

Comparé à notre approche de détection de problèmes de pigeons précédemment présentée, nous remarquons que nous n'avons plus de détection sur les instances des familles `cmpadd`, `dinphil` et `BMC`. Cette différence peut être due au fait que nous utilisons ici une approche CDCL et que l'heuristique utilisée n'est pas la même.

TABLE 4.9 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur DPLL avec les isomorphismes classiques sur les instances des compétitions.

	DPLL (cache intégré, isomorphismes classiques)						
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Abandons	Temps (recherche)	Temps (GSS)
marg2x6	21	20	17 (17)	18	0	0.004	0.162
marg3x3add8	26	25	22 (22)	24	0	0.024	0.813
marg6x6	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	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
3col40_5_9	341	319	263 (49)	56,040	0	1.615	650.558
homer06	111	105	98 (27)	420	40	0.495	116.096
homer07	228	220	220 (56)	1111	69	1.320	328.307
homer17	363	348	352 (92)	1,691	211	3.249	712.465
ca002	10	9	1 (1)	19	0	0.003	0.103
ca004	56	53	28 (6)	1,260	0	0.046	7.239
dp02u01	7	6	0 (0)	4	0	0.002	0.026
dp03u02	56	47	5 (3)	1,147	0	0.086	40.822
BMC (4)	63	56	40 (15)	1,071	4	0.069	24.680
BMC (23)	61	57	40 (13)	1,028	7	0.075	27.129
ezfact16_4	46	32	19 (13)	719	0	0.293	55.174
ezfact16_5	49	43	27 (13)	1,274	0	0.436	91.899
rope_0001	16	13	10 (2)	60	0	0.009	0.422
rope_0002	15	14	11 (4)	46	0	0.012	0.951
rope_0004	117	109	82 (20)	5,812	0	0.219	454.973
grid_05_05	282	266	234 (110)	25,648	0	0.724	135.574

4.7 Détection étendue d'isomorphismes

Le problème de l'isomorphisme de sous-graphe nous permet d'identifier si une entrée du cache est contenue dans la formule courante à un renommage de variables près. En d'autres termes, pour une entrée de cache E et une formule courante ϕ', nous voulons déterminer s'il existe un renommage cohérent σ des littéraux (i.e. une permutation des littéraux telle que si $\sigma(l) = l'$ alors $\sigma(\neg l) = \neg l'$) tel que $\sigma(E) \subseteq \phi'$. Nous rappelons que ϕ', la formule courante, est obtenue en simplifiant la formule initiale, ce qui signifie enlever les clauses satisfaites et également les littéraux falsifiés dans chaque clause restante.

Cependant, nous avons observé que la formule courante peut contenir (à un renommage de variables près) une entrée du cache simplifiée par un sous-ensemble α_1 de l'interprétation courante α. Autrement dit, $\sigma(E)_{|\alpha_1} \subseteq \phi'$. Quand cela se produit, comme E est incohérente, $\sigma(E)_{|\alpha_1}$ est aussi incohérente et donc ϕ' est UNSAT. En pratique, cela peut arriver car nous recherchons des isomorphismes uniquement au niveau des décisions et non pas après chaque propagation, ce qui serait évidemment trop coûteux. Cela peut aussi se produire lorsque des littéraux apparaissant dans E sont assignés tôt dans la branche. Pour détecter ce cas, nous devons introduire une notion d'isomorphisme étendu.

TABLE 4.10 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur CDCL avec les isomorphismes classiques sur les instances des compétitions.

	CDCL (cache intégré, isomorphismes classiques)					
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Temps (recherche)	Temps (GSS)
marg2x6	20	17	18 (17)	44	0.007	0.341
marg3x3add8	32	25	20 (20)	55	0.022	0.524
marg6x6	86	84	84 (84)	276	0.181	15.190
Urquhart-s3-b9	21	18	17 (17)	38	0.009	0.329
Urquhart-s3-b3	29	26	27 (25)	59	0.023	0.861
Urquhart-s5-b5	95	91	91 (90)	259	0.367	55.682
x1_16	18	15	14 (14)	60	0.010	0.693
x1_24	40	35	32 (18)	202	0.022	1.665
x1_96	2177	471	106 (76)	8513	1.759	423.444
3col20_5_6	27	5	0 (0)	15	0.005	0.099
3col40_5_4	110	22	54 (3)	786	0.107	5.535
3col60_5_1	915	396	387 (1)	30,362	2.624	768.225
homer06	102	95	92 (20)	462	0.485	47.701
ca004	43	13	0 (0)	171	0.015	1.222
ca008	144	58	0 (0)	3,046	0.194	39.385
ca016	535	85	42 (2)	9,496	1.292	439.575
dp03u02	26	10	0 (0)	81	0.077	9.629
dp04u03	70	19	0 (0)	631	0.372	307.948
BMC (4)	38	1	0 (0)	291	0.306	9.057
BMC (23)	51	3	4 (1)	331	.0.096	3.244
BMC (61)	69	2	10 (1)	2,065	0.794	41.228
ezfact16_2	37	30	0 (0)	1,208	0.172	32.346
ezfact16_10	315	291	1 (1)	16,944	1.735	594.338
rope_0001	23	14	4 (2)	116	0.006	0.742
rope_0002	31	24	20 (4)	326	0.020	3.915
rope_0003	795	369	0 (0)	14,111	0.905	358.383
grid_05_05	395	336	35 (5)	21,345	1.856	326.346

À titre d'exemple, considérons la formule ϕ de la partie gauche de la figure 4.9 et contenant les clauses du problème PHP_4 dans lequel la clause $x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4}$ est remplacée par $a \vee b \vee x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4}$ (où a et b sont de nouvelles variables). Nous considérons également que le cache contient déjà PHP_4 et que $x_{2,4}$ a été assigné à faux tôt dans la branche courante. Quand a et b sont tous les deux assignés à faux, nous obtenons alors la formule de la partie droite de la figure 4.9 et il est clair que PHP_4 est maintenant contenu (d'une certaine manière) dans la formule courante. Cependant, syntaxiquement, cela n'est pas vrai si nous considérons la formule courante simplifiée. En effet, celle-ci ne contient pas la seconde clause de PHP_4, puisqu'elle a été réduite. Ainsi, un isomorphisme classique ne peut pas être détecté. Par contre, si a et b avaient été assignés avant $x_{2,4}$, cette identification aurait été possible. Il est clair que cette méthode est très sensible à l'ordre des affectations, et cela devrait être évité. On note aussi que les littéraux falsifiés a et b n'ont pas le même rôle que le littéral falsifié $x_{2,4}$ car les littéraux a et b doivent bel et bien être falsifiés pour pouvoir détecter le problème PHP_4 alors que le littéral $x_{2,4}$ doit être conservé.

TABLE 4.11 – Résultats expérimentaux quand le cache pré-rempli est utilisé pour détecter des problèmes de pigeons pendant la recherche d'un solveur CDCL avec les isomorphismes classiques.

	CDCL (cache pré-rempli, isomorphismes classiques)					
Instance	Conflits	Isomorphismes de sous-graphe	Tailles détectées	Appels	Temps (recherche)	Temps (GSS)
homer06	1	1 (1)	9	1	5.260	0.385
homer11	1	1 (1)	10	1	4.157	0.653
homer16	1	1 (1)	11	1	3.560	0.420
ca004	43	0 (0)	-	151	3.736	0.922
ca008	144	0 (0)	-	517	3.578	4.163
ca032	1,010	0 (0)	-	7,756	7.980	271.506
dp03u02	26	0 (0)	-	94	3.484	2.049
dp04u03	70	0 (0)	-	198	3.612	21.538
BMC (4)	38	0 (0)	-	819	3.569	8.021
BMC (42)	70	0 (0)	-	4,053	4.540	79.276
term1_gr_rcs_w3	1	1 (1)	3	2	3.980	1.080
am_4_4	5,950	0 (0)	-	14,741	22.071	532.403
rope_0003	830	3 (1)	2	2,158	3.960	13.356
rope_0004	1,079	1 (1)	2	3,487	4.104	25.890
rope_0005	1,557	2 (1)	2	5,406	4.640	48.014

$$
\begin{array}{r|l}
a \vee b \vee x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4} & x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4} \\
x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4} & x_{2,1} \vee x_{2,2} \vee x_{2,3} \\
x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4} & x_{3,1} \vee x_{3,2} \vee x_{3,3} \vee x_{3,4} \\
x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4} & x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4} \\
x_{5,1} \vee x_{5,2} \vee x_{5,3} \vee x_{5,4} & x_{5,1} \vee x_{5,2} \vee x_{5,3} \vee x_{5,4}
\end{array}
$$

FIGURE 4.9 – Une formule ϕ et la même formule ϕ simplifiée après avoir assigné $x_{2,4}$, a et b à faux. Nous avons exclu les clauses binaires implémentant les exclusions mutuelles du problème PHP_4.

4.7.1 Définition formelle

Soit ϕ la formule initiale, α l'interprétation courante et E une entrée du cache. Nous voulons déterminer si α peut être partitionné en deux sous-ensembles α_0 et α_1 tels que $\sigma(E)_{\alpha_1} \subseteq \phi'$ (avec $\phi' = \phi_{|\alpha_0 \cup \alpha_1}$). α_1 est utilisé pour simplifier à la fois $\sigma(E)$ et ϕ, α_0 est utilisé pour simplifier uniquement ϕ. En d'autres termes, nous voulons détecter si, en réordonnant les affectations, nous pouvons obtenir à un moment donné un isomorphisme classique : $\sigma(E) \subseteq \phi_{|\alpha_0}$. La figure 4.10 illustre la relation entre les différentes formules, les interprétations et les tests. Évidemment, nous ne connaissons pas α_0 et α_1. Il doit être noté que α_1 peut satisfaire une clause C de $\sigma(E)$, mais nous devons être sûrs que C est bien présente dans ϕ. Sinon la formule courante peut contenir uniquement un sous-ensemble strict de E, qui peut être cohérent. Cela signifie que l'on doit garder les clauses satisfaites lorsque nous effectuons nos tests. Par ailleurs, une clause $C \in E$ peut correspondre à une clause C' de ϕ même si C' contient des littéraux supplémentaires, tant que ceux-ci sont falsifiés par α_0. Pour formaliser cela, nous introduisons une relation d'inclusion spécifique $\sqsubseteq$ telle que $\sigma(E) \sqsubseteq \phi'$ implique que $\exists \alpha_0$ tel que $\sigma(E) \subseteq \phi_{|\alpha_0}$.

Soit une interprétation courante α et deux clauses C et C', $C \sqsubseteq C'$ si et seulement si $C \subseteq C'$ et $\forall l \in C' \backslash C$, $\alpha(l) = faux$ (l est assigné à faux dans α). Soit α l'interprétation courante et deux

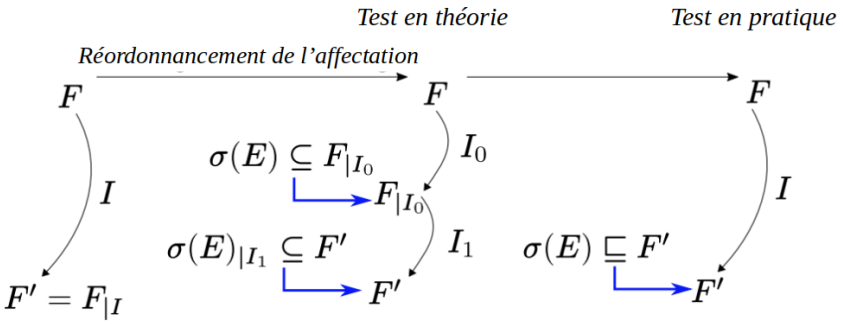


FIGURE 4.10 – Principe du test d'isomorphisme étendu.

formules ϕ_1 et ϕ_2, $\phi_1 \sqsubseteq \phi_2$ si et seulement si chaque clause C de ϕ_1 correspond à une clause C' de ϕ_2 tel que $C \sqsubseteq C'$. Dans notre cas, nous ne considérons que des appariements bijectifs pour simplifier les tests. Évidemment, $\sigma(E) \sqsubseteq \phi'$ implique que $\exists \alpha_0$ tel que $\sigma(E) \subseteq \phi_{|\alpha_0}$ car $\sqsubseteq$ ignore les littéraux falsifiés, qui appartiennent soit à α_0, auquel cas nous souhaitons les ignorer, soit à α_1, dans ce cas ils sont simplifiés à la fois dans $\sigma(E)$ et ϕ'.

Déterminer si un isomorphisme étendu de formule existe est toujours dans NP. En effet, si nous avons un oracle qui nous donne un renommage étendu et cohérent σ' qui fait correspondre un littéral l à un autre littéral si $l \notin \alpha_0$, ou le fait correspondre à vrai ou faux si $l \in \alpha_0$, alors vérifier si $\sigma(E) \sqsubseteq \phi'$ (σ est σ' restreint aux littéraux n'apparaissant pas dans α_0) revient à vérifier si $\sigma'(E) \subseteq \phi''$ où ϕ'' est obtenue en supprimant de ϕ tous les littéraux falsifiés par α_0. Ce dernier test est clairement polynomial.

Identifier des isomorphismes étendus pourrait être réalisé de diverses manières, par exemple en adaptant un solveur d'isomorphismes de sous-graphes. De notre côté, nous avons choisi une option plus simple consistant à encoder l'isomorphisme étendu sous la forme d'un problème d'isomorphisme de sous-graphe, même si l'encodage proposé n'est pas polynomial. Trouver un encodage polynomial est le sujet de travaux futurs.

4.7.2 Encodage étendu

Pour détecter ces isomorphismes étendus, nous proposons une extension de l'encodage des formules présenté à la figure 4.3. Comme nous souhaitons ajouter la possibilité d'assigner certains littéraux d'une entrée de cache à vrai ou faux, nous devons adapter notre encodage sous forme de graphe. Autoriser certains littéraux d'une clause donnée à être satisfaits est évident : nous encodons également les clauses satisfaites, i.e. nous ne supprimons pas les clauses satisfaites avant d'appliquer l'encodage sous forme de graphe sur la formule courante. Autoriser des littéraux d'une clause donnée à être falsifiés (i.e. deviner α_0) est plus complexe. En effet, un littéral falsifié peut soit être effacé (s'il appartient à α_0) soit conservé (s'il appartient à α_1) pour pouvoir correspondre à une entrée du cache. Comme α_0 et α_1 sont tous les deux inconnus, nous devons considérer les deux cas possibles, pour chaque littéral falsifié. Donc, pour une clause contenant n littéraux falsifiés, nous devons considérer 2^n variantes de la clause initiale (l'encodage est donc de taille exponentielle) en effaçant certains littéraux falsifiés (depuis le cas où aucun de ces littéraux n'a été effacé jusque celui où ils ont tous été supprimés) et en faisant en sorte de ne sélectionner qu'une seule variante. Cela permet au solveur d'isomorphismes de sous-graphes d'identifier quels littéraux appartienent à α_0. On remarque que le pire des cas correspond aux clauses utilisées comme des raisons puisque tous les littéraux sauf un vont être falsifiés (il n'y a pas de clause conflit à ce stade). Afin d'éviter de faire correspondre plusieurs variantes d'une

même clause à différentes clauses d'une entrée, nous devons introduire un nouveau type de nœud dans notre représentation graphique. Ces nouveaux nœuds, que nous appellerons *nœuds d'exclusion*, vont relier toutes les variantes d'une même clause dans la représentation de la formule courante. Pour les entrées de cache, comme il n'y a pas de variantes, nous ajoutons simplement un unique nœud d'exclusion par clause. Ainsi, pour chaque nœud d'exclusion n_1 dans une entrée de cache, nous allons essayer de le faire correspondre avec un nœud d'exclusion n_2 parmi ceux présents dans la représentation de la formule courante. De plus, nous devons faire correspondre l'arc reliant le nœud n_1 à un nœud de clause avec un seul des différents arcs reliant n_2 aux nœuds représentant les variantes d'une même clause. Ainsi, pour les autres nœuds de clause présents dans la représentation de l'entrée de cache considérée, nous ne pourrons plus utiliser les variantes reliées au nœud d'exclusion n_2 puisque ce dernier est déjà en correspondance avec un autre nœud de l'entrée de cache et qu'il n'est pas possible de faire correspondre un unique nœud d'exclusion dans la représentation de la formule avec plusieurs nœuds d'exclusion de l'entrée de cache (comme ces derniers vont recevoir des identifiants différents, ils doivent être mis en correspondance avec autant de nœuds d'exclusion différents dans la représentation de la formule). Un exemple graphique de cet encodage étendu est donné à la figure 4.11. Dans celui-ci, nous considérons une clause $C_5 = (x_1 \vee x_2 \vee x_3 \vee x_4)$. Si les littéraux x_3 et x_4 sont falsifiés dans l'interprétation courante, alors ce nouvel encodage va créer 4 variantes de C_5 : $C_{5,1} = (x_1 \vee x_2)$, $C_{5,2} = (x_1 \vee x_2 \vee x_3)$, $C_{5,3} = (x_1 \vee x_2 \vee x_4)$ et $C_{5,4} = (x_1 \vee x_2 \vee x_3 \vee x_4)$.

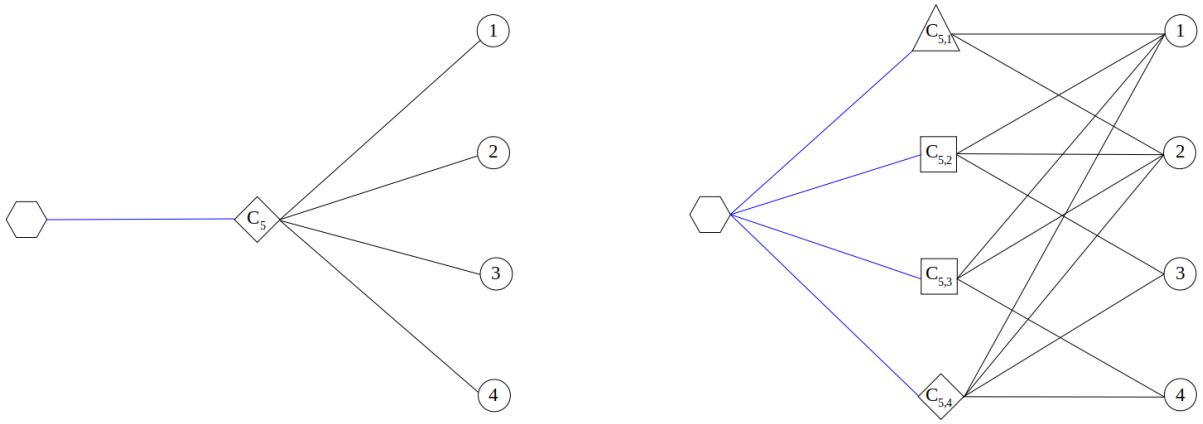


FIGURE 4.11 – Modélisation sous forme de graphe de la clause $C_5 = (x_1 \vee x_2 \vee x_3 \vee x_4)$ si elle est présente dans une entrée de cache (à gauche de la figure) ou dans la formule courante quand les littéraux x_3 et x_4 sont falsifiés (à droite de la figure). Les littéraux sont toujours représentés par des cercles. Les triangles et carrés représentent toujours des clauses binaires et ternaires. Un losange représente une clause de taille 4 et un hexagone représente un nœud d'exclusion.

Nous soulignons ici que nous créons les variantes spécifiquement pour une clause et indépendamment des autres clauses présentes dans la formule. Cela signifie qu'un littéral effacé à l'intérieur d'une clause peut très bien être conservé à l'intérieur d'une autre clause. Nous considérons que ce dernier point ne pose pas de problème dans notre approche. En effet, comme l'entrée de cache est UNSAT, nous pouvons la simplifier par des littéraux falsifiés tout en conservant l'incohérence. Ainsi, si un littéral doit être enlevé d'une clause donnée et conservé dans une autre clause pour pouvoir reconnaître une entrée du cache, dans les deux cas, ce littéral est falsifié. Dans le premier cas, c'est possible car on réduit les clauses de la formule courante des littéraux

falsifiés. Dans le second cas, c'est aussi possible car on peut falsifier les littéraux d'une formule incohérente, ici l'entrée du cache, la formule reste incohérente.

Exemple 21. *À titre d'exemple, supposons que nous avons une entrée de cache contenant les clauses* $(a \vee b \vee c)$ *et* $(d \vee e)$ *et que la formule courante contienne les clauses* $(l_1 \vee l_2 \vee f_1 \vee f_2)$ *et* $(l_3 \vee l_4 \vee f_1 \vee f_3)$*, où les littéraux* f_1*,* f_2 *et* f_3 *sont falsifiés dans l'interprétation courante. Nous pouvons par exemple faire correspondre les littéraux* a*,* b*,* d *et* e *avec respectivement les littéraux* l_1*,* l_2*,* l_3 *et* l_4*. Avec l'encodage étendu et les variantes créées dans la sous-formule, nous pouvons faire correspondre le littéral* c *avec* f_1*. Ainsi, le littéral* f_1 *doit être conservé dans la clause* $(l_1 \vee l_2 \vee f_1 \vee f_2)$ *mais pas dans* $(l_3 \vee l_4 \vee f_1 \vee f_3)$*. Les littéraux* f_2 *et* f_3 *doivent être supprimés dans tous les cas. Cela signifie que pour reconnaître cette entrée de cache, nous pouvons faire correspondre* $(a \vee b \vee c)$ *avec* $(l_1 \vee l_2 \vee f_1)$ *et* $(d \vee e)$ *avec* $(l_3 \vee l_4)$*.*

4.8 Résultats expérimentaux pour les isomorphismes étendus

4.8.1 Taille de l'encodage

Comme nous ne simplifions pas les clauses satisfaites et comme nous créons plusieurs variantes de certaines clauses, l'encodage étendu est bien plus large que l'encodage classique. Cependant, nous avons mesuré la taille des graphes créés pour quelques instances et nous avons observé que le nombre de variantes créées n'explose pas. Nous remarquons qu'il n'est pas nécessaire de générer les variantes plus grandes que la plus longue clause présente dans toutes les entrées du cache (comme nous utilisons le même encodage de la formule courante pour toutes les entrées du cache).

Nous avons étudié le nombre de variantes ajoutées à la formule courante pour quelques instances. Pour cette expérimentation, nous avons spécifiquement autorisé un temps limite de 30 minutes pour chaque appel au GSS et un temps limite de 15 heures pour le solveur. Les résultats de cette expérimentation peuvent être observés dans la table 4.12 pour l'approche DPLL et dans la table 4.13 pour l'approche CDCL. Les instances sélectionnées correspondent à des instances UNSAT pour lesquelles nous n'avons pas obtenu de réponse en utilisant les isomorphismes classiques. Pour chaque instance, nous donnons son nombre original de clauses, suivi par le nombre de graphes que l'instance a tenté de créer et le nombre de créations de graphe qui ont échoué. Nous donnons ensuite le nombre minimum et maximum de variantes créées. Enfin, nous donnons le nombre d'appels au GSS qui ont atteint le temps limite de 30 minutes.

En pratique, nous avons observé que le nombre de clauses ajoutées est, en général, raisonnable comparé au nombre original de clauses. Cependant, comme cet encodage ne supprime pas les clauses satisfaites et comme nous créons également plusieurs variantes de certaines clauses, les appels au GSS prennent souvent beaucoup de temps et il n'est pas rare que nous atteignons le temps limite de 30 minutes. Ainsi, nous n'explorons pas une grande partie de l'arbre de recherche pour ces instances. Concernant l'instance `hanoi4u.sat03 − 399.cnf`, à un certain moment de la recherche, le nombre de variantes est bien plus élevé que le nombre de clauses initiales et nous atteignons la limite du million de littéraux que nous avons arbitrairement imposée. Cela se produit à cause des longues clauses présentes dans la formule et qui créent de nombreuses variantes. Dans ce cas, nous arrêtons la génération du graphe et nous n'appelons pas le GSS. Cela explique le nombre élevé de générations de graphe qui ont été tentées mais qui ont été avortées pendant le processus. À titre de comparaison, nous donnons les mêmes informations lorsque les isomorphismes classiques sont utilisés dans la table 4.14 pour l'approche DPLL et

dans la table 4.15 pour l'approche CDCL. Cette fois, nous avons imposé un temps limite de 2 secondes pour chaque appel au GSS et un temps limite de 15 minutes par instance. Comme prévu, les graphes créés sont bien plus petits avec l'encodage classique.

TABLE 4.12 – Statistiques concernant la taille de l'encodage lorsque les isomorphismes étendus sont utilisés avec l'approche DPLL.

Instance	Clauses	Graphes	Échecs	Variantes (min)	Variantes (max)	Abandons
3col60_5_1	493	184	0	564	1062	16
am_4_4	1455	55	0	1510	2965	27
ezfact16_2	1070	155	0	1075	2529	5
Mat25	1968	35	0	2315	2819	16
hanoi4u	15919	30294	30289	35409	128332	10

TABLE 4.13 – Statistiques concernant la taille de l'encodage lorsque les isomorphismes étendus sont utilisés avec l'approche CDCL.

Instance	Clauses	Graphes	Échecs	Variantes (min)	Variantes (max)	Abandons
3col60_5_1	493	924	0	557	1867	0
am_4_4	1455	848	0	1574	2974	0
ezfact16_2	1070	106	0	1072	2253	0
Mat25	1968	34	0	2206	2993	17
hanoi4u	15919	10407	10407	113899	122340	0

TABLE 4.14 – Statistiques concernant la taille de l'encodage lorsque les isomorphismes classiques sont utilisés avec l'approche DPLL.

Instance	Clauses	Graphs	Fails	Clauses (min)	Clauses (max)	Aborts
3col60_5_1	493	260	0	223	405	97
am_4_4.sat03-360	1455	504	0	6	1372	0
ezfact16_2	1070	346	0	297	1029	0
Mat25	1968	59	0	849	1618	39
hanoi4u.sat03-399	15919	63	0	782	9429	19

TABLE 4.15 – Statistiques concernant la taille de l'encodage lorsque les isomorphismes classiques sont utilisés avec l'approche CDCL.

Instance	Clauses	Graphs	Fails	Clauses (min)	Clauses (max)	Aborts
3col60_5_1	493	2389	0	25	470	0
am_4_4.sat03-360	1455	7370	0	6	1310	0
ezfact16_2	1070	108	0	333	1010	0
Mat25	1968	82	0	747	1645	20
hanoi4u.sat03-399	15919	10407	0	112	12082	56

Ensuite, les figures 4.12, 4.13, 4.14, 4.15, 4.16 et 4.17 donnent le temps de réponse des appels au Glasgow Subgraph Solver lorsque les isomorphismes étendus sont utilisés. Pour chacune de ces figures, l'axe des abscisses correspond aux différents appels au GSS et l'axe des ordonnées indique le temps de réponse des appels. Ici, les temps de réponse ont été triés par ordre croissant. Pour la plupart de ces instances, nous remarquons que nous passons rapidement de l'état où nous avons

tout de suite une réponse à l'état où nous atteignons la limite de temps. Même si l'utilisation de l'encodage étendu exige de passer plus de temps sur l'étape de recherche d'isomorphisme, il semble peu utile d'y consacrer beaucoup plus de temps, puisque la très vaste majorité des appels terminent immédiatement.

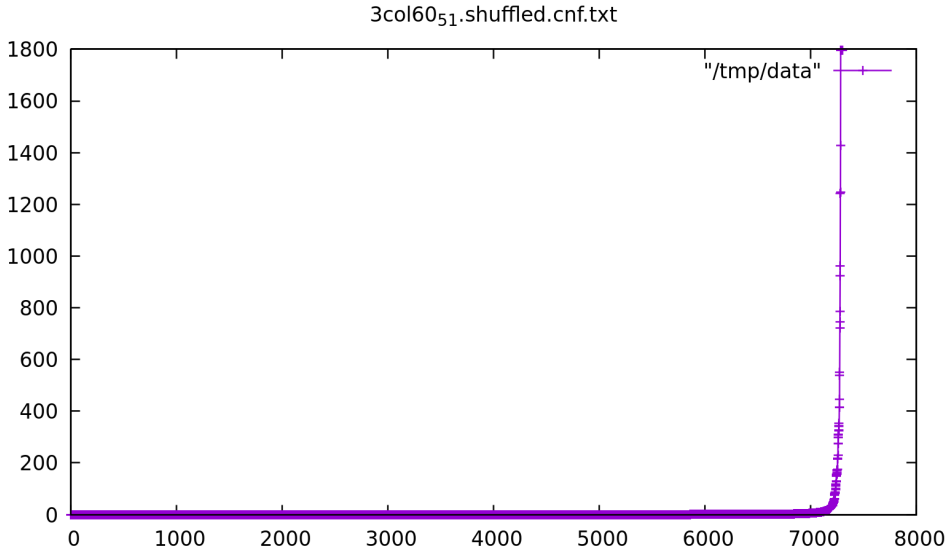


FIGURE 4.12 – Graphe représentant les temps des appels au Glasgow Subgraph Solver triés dans l'ordre croissant lorsque les isomorphismes étendus sont utilisés sur l'instance `3col60_5_1.cnf`.

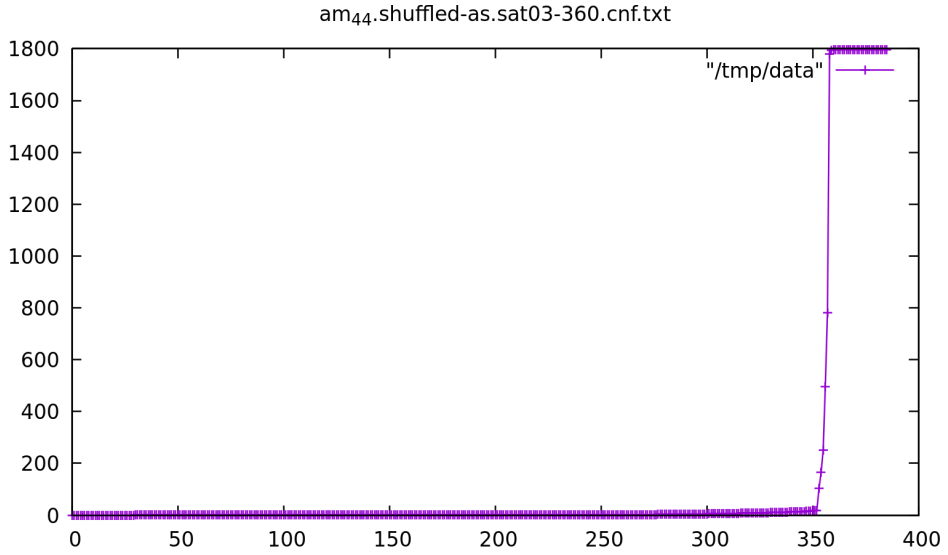


FIGURE 4.13 – Graphe représentant les temps des appels au Glasgow Subgraph Solver triés dans l'ordre croissant lorsque les isomorphismes étendus sont utilisés sur l'instance `am_4_4.cnf`.

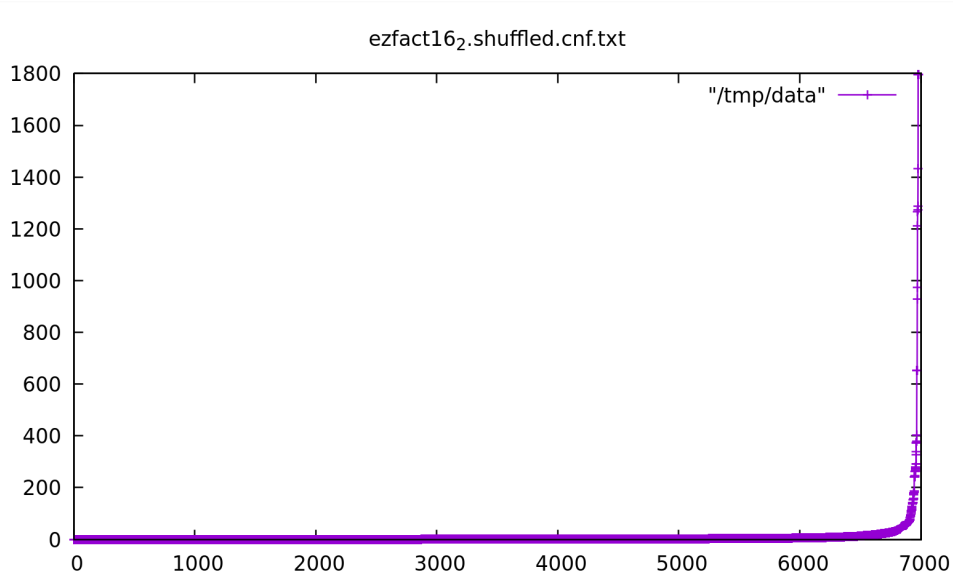


FIGURE 4.14 – Graphe représentant les temps des appels au Glasgow Subgraph Solver triés dans l'ordre croissant lorsque les isomorphismes étendus sont utilisés sur l'instance `ezfact16_2.cnf`.

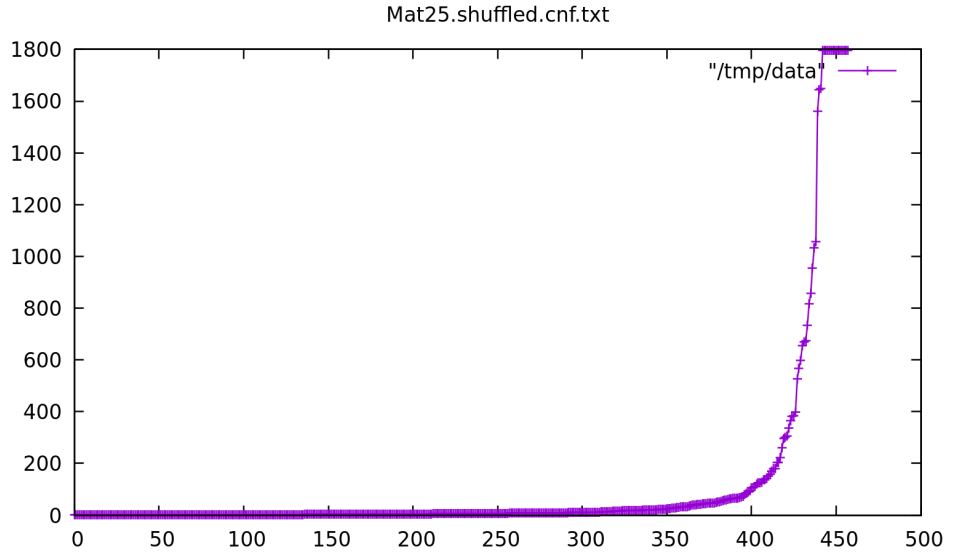


FIGURE 4.15 – Graphe représentant les temps des appels au Glasgow Subgraph Solver triés dans l'ordre croissant lorsque les isomorphismes étendus sont utilisés sur l'instance `Mat25.cnf`.

4.8.2 Instances de problèmes de pigeons

Nous avons réitéré certaines de nos expérimentations précédentes avec les isomorphismes étendus. Cette fois-ci, nous avons imposé un temps limite de 4 secondes pour chaque appel au GSS et de 30 secondes pour chaque instance (i.e. nous avons doublé les temps imposés pour les isomorphismes classiques). Des résultats concernant le pouvoir de compression lorsque les isomorphismes étendus sont utilisés sur les instances de problèmes de pigeons peuvent être trouvés dans la table 4.16. Ensuite, les figures 4.18 et 4.19 donnent les deux arbres de recherche

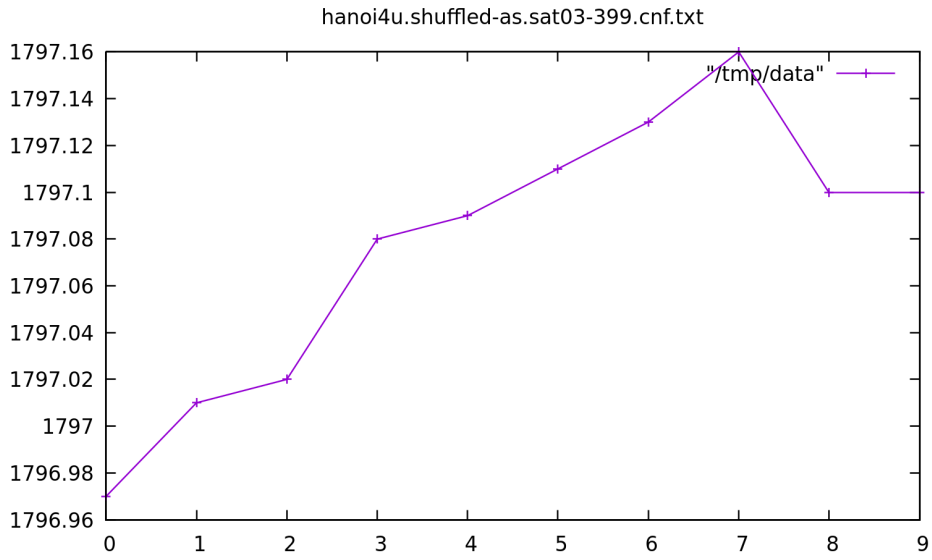


FIGURE 4.16 – Graphe représentant les temps des appels au Glasgow Subgraph Solver triés dans l'ordre croissant lorsque les isomorphismes étendus sont utilisés sur l'instance `hanoi4u.cnf`.

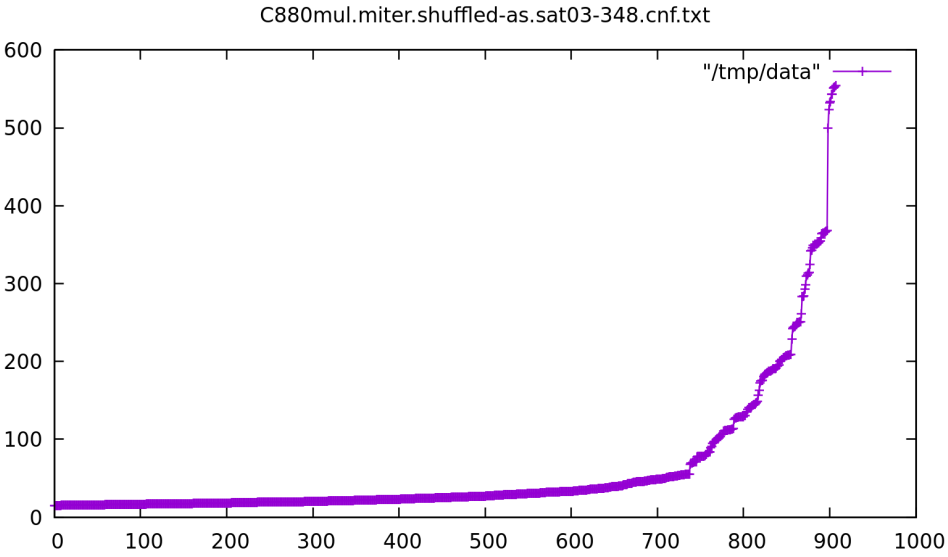


FIGURE 4.17 – Graphe représentant les temps des appels au Glasgow Subgraph Solver triés dans l'ordre croissant lorsque les isomorphismes étendus sont utilisés sur l'instance `c880mul.miter.cnf`.

développés pour le problème PHP_5 avec les deux types d'encodage. On remarque notamment que l'encodage étendu permet de détecter plus d'isomorphismes et de développer un arbre plus petit.

Nous donnons maintenant dans la table 4.17 des résultats de notre approche avec cache intégré lorsque les isomorphismes étendus sont utilisés dans une approche CDCL sur les problèmes de pigeons Concernant les problèmes de pigeons, nous n'avons pas pu retrouver d'arbre de recherche avec une forme de peigne avec ces instances mais nous obtenons néanmoins des

TABLE 4.16 – Résultats expérimentaux concernant le pouvoir de compression de notre approche avec les isomorphismes étendus sur les instances de problèmes de pigeons.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport	Conflits (sans cache)	Conflits (cache)	Rapport
PHP_4	100	28	8	$2.9\ 10^{-1}$	29	13	$4.5\ 10^{-1}$
PHP_5	180	168	12	$7.1\ 10^{-2}$	151	29	$1.9\ 10^{-1}$

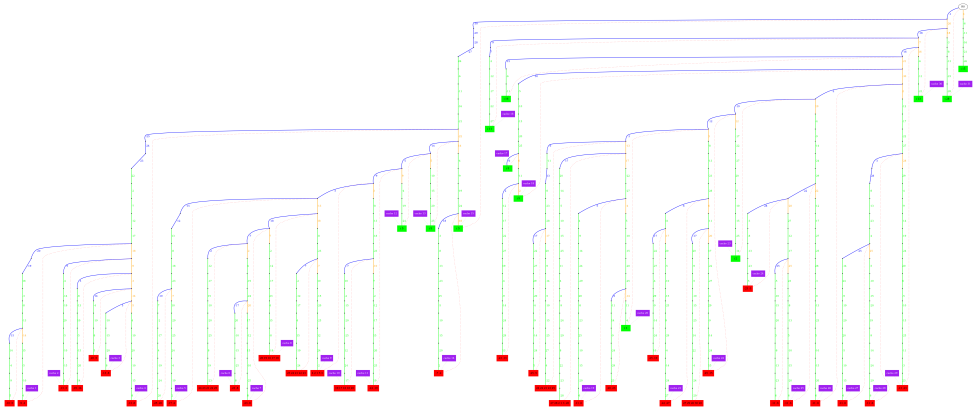


FIGURE 4.18 – Arbre de recherche développé pour le problème PHP_5 quand le cache est utilisé en post-traitement avec les isomorphismes classiques dans un solveur CDCL.

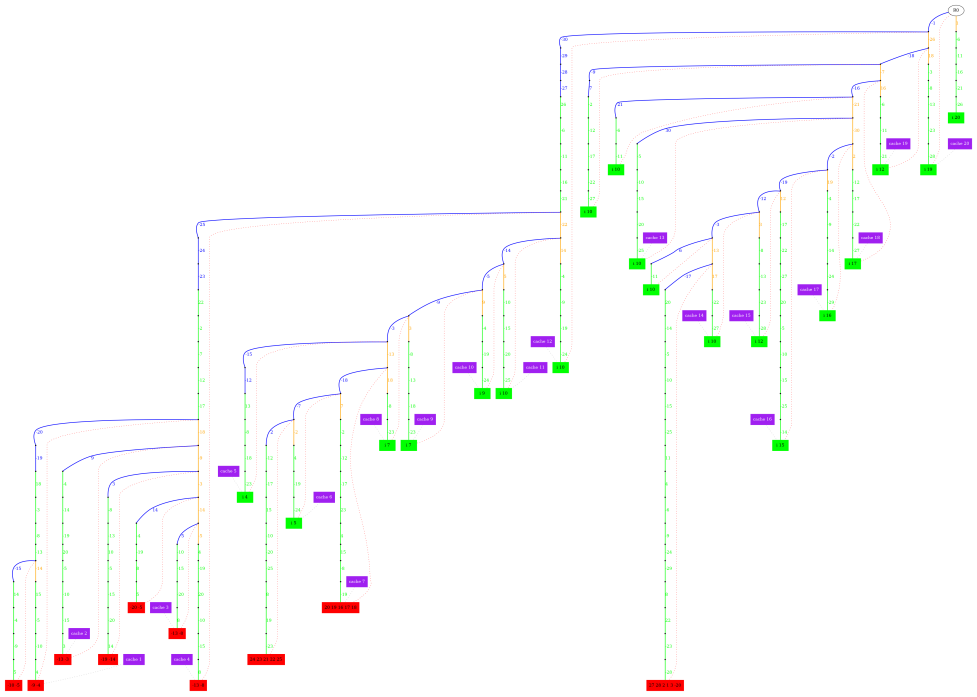


FIGURE 4.19 – Arbre de recherche développé pour le problème PHP_5 quand le cache est utilisé en post-traitement avec les isomorphismes étendus dans un solveur CDCL.

arbres relativement courts. Si nous considérons les arbres de recherche obtenus pour le problème

PHP_5 lorsque l'on utilise à la fois les isomorphismes classiques et étendus, comme on peut le voir avec les figures 4.20 et 4.21, nous observons que l'encodage étendu est capable de détecter plus d'isomorphismes et aussi de développer un arbre de recherche plus petit.

TABLE 4.17 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur CDCL avec les isomorphismes étendus sur les instances de problèmes de pigeons.

	CDCL (cache intégré, isomorphismes étendus)					
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Temps (recherche)	Temps (GSS)
PHP_4	13	8	5 (4)	29	0.016	0.198
PHP_5	23	17	16 (11)	95	0.017	1.183
PHP_7	42	35	35 (21)	391	0.083	220.329

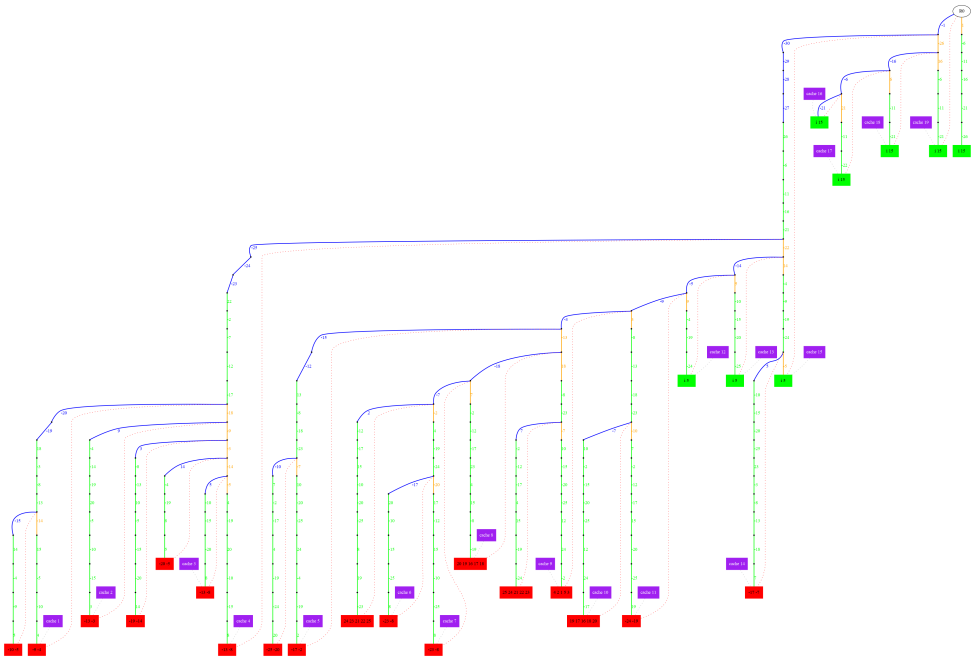


FIGURE 4.20 – Arbre de recherche développé pour le problème PHP_5 quand le cache intégré est utilisé avec les isomorphismes classiques dans un solveur CDCL.

4.8.3 Instances d'échiquiers mutilés

Concernant l'utilisation des isomorphismes étendus sur les instances d'échiquiers mutilés, nous donnons des résultats concernant le pouvoir de compression dans la table 4.18 et des résultats concernant l'intégration du cache directement dans le solveur dans la table 4.19. Les résultats sont relativement similaires à ce que nous avons obtenu avec les isomorphismes classiques. L'heuristique de MiniSat nous permet toujours de détecter un isomorphisme très tôt dans l'arbre de recherche, ce qui permet de supprimer une bonne partie des branches de l'arbre de recherche.

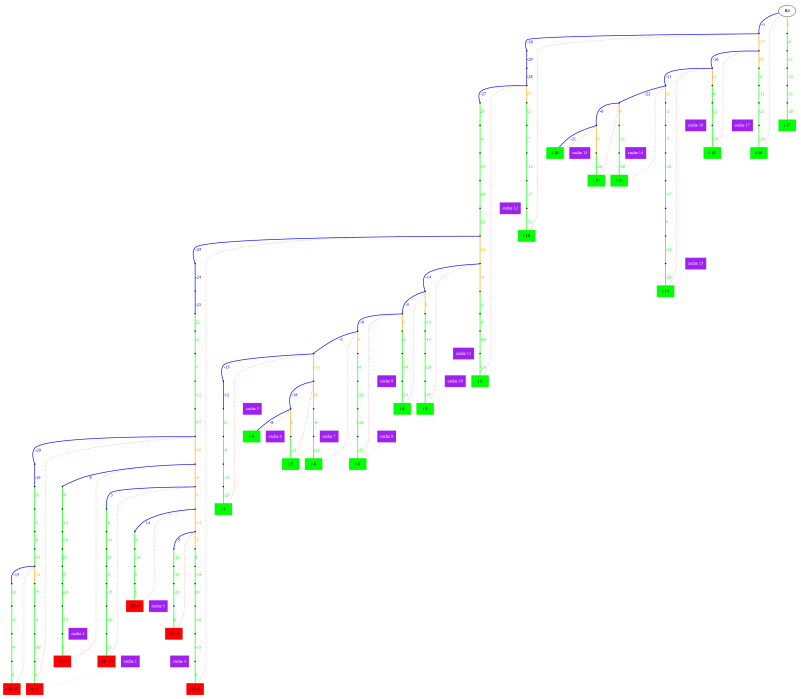


FIGURE 4.21 – Arbre de recherche développé pour le problème PHP_5 quand le cache intégré est utilisé avec les isomorphismes étendus dans un solveur CDCL.

TABLE 4.18 – Résultats expérimentaux concernant le pouvoir de compression de notre approche avec les isomorphismes étendus sur les instances d'échiquiers mutilés.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport	Conflits (sans cache)	Conflits (cache)	Rapport
mChess_5	240	45	14	$3.1\ 10^{-1}$	30	14	$4.7\ 10^{-1}$
mChess_6	388	105	32	$3.0\ 10^{-1}$	44	23	$5.2\ 10^{-1}$
mChess_7	568	-	-	-	-	-	-

TABLE 4.19 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur CDCL avec les isomorphismes étendus sur les instances d'échiquiers mutilés.

	CDCL (cache intégré, isomorphismes étendus)					
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Temps (recherche)	Temps (GSS)
mChess_5	13	8	5 (4)	51	0.015	0.678
mChess_6	23	17	1 (1)	103	0.036	4.200
mChess_7	-	-	-	-	-	-

4.8.4 Instances des compétitions

Élagage après la recherche

Nous donnons maintenant dans la table 4.20 certains résultats obtenus concernant le pouvoir de compression de notre approche lorsque les isomorphismes étendus sont utilisés et les taux de compression sont classés dans la table 4.21.

TABLE 4.20 – Résultats expérimentaux concernant le pouvoir de compression de notre approche avec les isomorphismes étendus sur les instances des compétitions.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport	Conflits (sans cache)	Conflits (cache)	Rapport
marg2x6	528	$5.2\ 10^5$	21	$4.0\ 10^{-5}$	$3.0\ 10^4$	20	$6.6\ 10^{-4}$
marg3x3add8	1,056	-	-	-	$1.8\ 10^5$	31	$1.7\ 10^{-4}$
Urquhart-s3-b9	1,240	$5.2\ 10^5$	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	-	-	-	$2.0\ 10^5$	32	$1.6\ 10^{-4}$
3col20_5_6	646	30	18	$6.0\ 10^{-1}$	27	24	$8.9\ 10^{-1}$
3col40_5_4	1,286	-	-	-	92	44	$4.8\ 10^{-1}$
3col60_5_1	1,926	-	-	-	-	-	-
homer06	1,800	-	-	-	-	-	-
homer07	2,178	$5.1\ 10^5$	29	$5.7\ 10^{-5}$	-	-	-
homer17	3,718	-	-	-	-	-	-
ca004	426	218	55	$2.5\ 10^{-1}$	43	43	1
ca008	940	-	-	-	144	133	$9.2\ 10^{-1}$
dp03u02	2,376	63	55	$8.7\ 10^{-1}$	26	26	1
dp04u03	5,688	-	-	-	70	70	1
BMC (4)	1122	$1.7\ 10^4$	60	$3.5\ 10^{-3}$	38	35	$9.2\ 10^{-1}$
BMC (23)	1122	$1.6\ 10^4$	55	$3.5\ 10^{-3}$	38	36	$9.5\ 10^{-1}$
BMC (61)	2,142	-	-	-	-	-	-
ezfact16_2	4,089	-	-	-	37	30	$8.1\ 10^{-1}$
ezfact16_4	4,089	236	53	$2.2\ 10^{-1}$	-	-	-
ezfact16_5	4,089	105	35	$3.3\ 10^{-1}$	50	39	$7.8\ 10^{-1}$
rope_0001	180	26	10	$3.8\ 10^{-1}$	30	13	$4.3\ 10^{-1}$
rope_0002	360	248	14	$5.6\ 10^{-2}$	162	24	$1.5\ 10^{-1}$
rope_0003	540	-	-	-	-	-	-
grid_05_05	298	-	-	-	428	125	$2.9\ 10^{-1}$

TABLE 4.21 – Distribution des rapports pour les techniques de post-traitement pour les approches DPLL et CDCL sur les instances SAT'02 et SAT'03 lorsque les isomorphismes étendus sont utilisés.

Rapport	Non résolu	$[1;0.75[$	$[0.75;0.5[$	$[0.5;0.25[$	$[0.25;10^{-1}[$	$[10^{-1};10^{-2}[$	$\leq 10^{-2}$
DPLL	530	4	2	10	9	3	22
CDCL	486	17	6	12	3	7	49

Élagage pendant la recherche

Nous donnons également certains résultats obtenus avec les isomorphismes étendus et une approche CDCL dans la table 4.22. Comme nous pouvons le voir, même si l'encodage peut permettre d'obtenir des arbres de recherche plus petits, nous remarquons également que cette approche est bien plus lente à cause de la taille de l'encodage.

TABLE 4.22 – Résultats expérimentaux quand le cache est utilisé pendant la recherche d'un solveur CDCL avec les isomorphismes étendus sur les instances des compétitions.

	CDCL (cache intégré, isomorphismes étendus)					
Instance	Conflits	Taille du cache	Isomorphismes de sous-graphe	Appels	Temps (recherche)	Temps (GSS)
marg2x6	20	17	18 (17)	50	0.057	3.331
marg3x3add8	31	24	21 (21)	74	0.067	14.081
marg4x4	41	39	39 (35)	186	0.108	130.496
Urquhart-s3-b9	21	18	17 (17)	50	0.039	2.959
Urquhart-s3-b3	29	26	27 (26)	67	0.269	17.958
x1_16	18	15	14 (14)	94	0.018	21.538
x1_24	29	24	22 (21)	117	0.044	60.011
x2_32	65	54	53 (40)	568	0.195	908.380
3col20_5_6	23	3	2 (2)	16	0.008	17.322
3col40_5_4	57	20	27 (4)	809	0.199	1474.762
3col40_5_10	101	51	68 (5)	4,459	0.581	1653.611
ca004	43	13	0 (0)	313	0.149	13.614
ca008	168	66	8 (3)	9,126	0.896	560.150
dp03u02	26	10	0 (0)	85	0.135	39.305
dp04u03	70	19	0 (0)	681	0.852	1406.251
BMC (4)	40	2	4 (1)	359	0.644	166.749
BMC (23)	52	3	6 (3)	347	0.526	58.319
ezfact16_2	41	35	11 (3)	1,094	0.408	904.826
ezfact16_5	18	3	11 (2)	128	0.216	34.186
rope_0001	13	10	8 (7)	52	0.018	1.860
rope_0002	20	16	15 (13)	117	0.069	24.230
rope_0004	82	38	37 (22)	1,028	0.472	957.178

4.9 Résultats globaux

Finalement, nous donnons un résumé de nos expérimentations sur les instances des compétitions dans la table 4.23 pour les isomorphismes classiques et dans la table 4.24 pour les isomorphismes étendus. MiniSat sans cache résoud évidemment bien plus d'instances que nos approches avec cache à cause du coût élevé du cache. On remarque que le fait d'intégrer le cache directement dans le solveur permet de résoudre bien plus d'instances que de faire du post-traitement car cela permet de couper l'espace de recherche. Cela permet par exemple de résoudre des instances que MiniSat ne parvient pas à résoudre en 4 heures (e.g. les instances de la famille `Urquhart`). Concernant les approches en post-traitement, l'approche CDCL est bien plus efficace que l'approche DPLL. Ces deux approches ont des performances assez semblables quand le cache est directement intégré. Par contre, l'ajout des isomorphismes étendus a ralenti le solveur et donc cette approche résoud bien moins d'instances que les approches utilisant des isomorphismes classiques.

4.10 Limites de l'approche

L'idée de départ de notre travail sur la compression des preuve est que, comme on a un nombre exponentiel de branches dans l'arbre, et une formule de taille « polynomiale », on devrait retomber plusieurs fois sur la même cause d'incohérence. Or, on peut construire une formule de taille polynomiale (par rapport à un paramètre n) et développer un arbre de recherche avec des

TABLE 4.23 – Résumé de nos expérimentations pour les isomorphismes classiques. 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 avec une forme de peigne trouvés.

Compétition	#UNSAT	MiniSat (1min)	DPLL (15min)		CDCL (15min)	
			Post traitement	Intégré classique	Post traitement	Intégré classique
SAT'02	382	276	42 (4)	106 (42)	78 (11)	113 (26)
SAT'03	198	78	17 (15)	87 (53)	39 (28)	72 (37)

TABLE 4.24 – Résumé de nos expérimentations pour les isomorphismes étendus. 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 forme de peigne trouvés.

Compétition	#UNSAT	MiniSat (1min)	DPLL (30min)	CDCL (30min)	
			Post traitement	Post traitement	Intégré étendu
SAT'02	382	276	32 (4)	60 (9)	51 (9)
SAT'03	198	78	18 (15)	34 (22)	38 (22)

nœuds qui sont tous différents (au sens non isomorphes). Prenons une famille de $2n$ formules F_i toutes UNSAT et toutes différentes (aucune formule F_i n'est isomorphe à une formule F_j, avec $i \neq j$). Ces formules doivent être de taille polynomiale par rapport à n. On peut construire facilement ces formules en prenant des formules aléatoires de taille limitée par une constante $P(n)$, avec P un polynôme.

On génère alors la formule ϕ contenant les clauses $(x_i \vee a_i \vee F_{2i})$ (pour i variant de 1 à n) et $(\neg x_i \vee a_i \vee F_{2i+1})$ (pour i variant de 1 à n) ainsi que la clause $(\neg a_1 \vee \neg a_2 \vee \neg a_3 \vee \cdots \neg a_n)$. Comme les F_j sont UNSAT, F_j est toujours faux et les 2 premières clauses peuvent se simplifier en $(x_i \vee a_i)$ et $(\neg x_i \vee a_i)$. Par résolution, on en déduit (a_i) (pour i variant de 1 à n), ce qui contredit la dernière clause. Donc, ϕ est UNSAT et de taille polynomiale par rapport à n.

Prenons maintenant un solveur DPLL qui branche d'abord sur les variables x_i. Sur chaque branche, après assignation des x_i, on a une formule qui contient des clauses de la forme $(a_i \vee F_j)$, plus la dernière clause. Par contre, d'une branche à l'autre, on a au moins un F_j qui est différent. De ce fait, à ce niveau de l'arbre, toutes les branches sont différentes et non isomorphes. On peut retrouver des isomorphisme plus bas dans l'arbre, mais en choisissant bien les F_i, on n'aura sans doute que des petits isomorphimes, sans grand intérêt. En résumé, on peut contruire une formule de taille polynomiale, et développer un arbre de recherche exponentiel avec des nœuds internes qui seront tous différents. Cette formule ne permettra pas d'obtenir des isomorphismes réellement pertinents.

4.11 Conclusion

Dans ce chapitre, nous avons présenté notre approche basée sur la détection de sous-formules déjà prouvées incohérentes pour réduire la taille d'un arbre de recherche. Pour ce faire, nous

utilisons un système de cache inspiré par ce qui existe déjà dans les compteurs de modèles, mais spécialisé au cadre des instances incohérentes. L'idée est d'enregistrer à divers moments de la recherche des sous-formules qui ont été prouvées UNSAT par le solveur et d'essayer de les reconnaître plus tard. Si cela arrive, nous pouvons alors couper la branche de l'arbre où nous nous trouvons. Nous utilisons les clauses apparaissant dans les conflits afin de générer des entrées de cache plus petites et donc plus faciles à détecter. En résolvant des problèmes d'isomorphismes de sous-graphes, nous sommes capables de détecter si une entrée du cache est présente dans la formule courante à un renommage près. Nous présentons une manière classique et aussi une manière étendue de gérer ces isomorphismes. Nous détaillons également l'intégration de cette approche dans les deux architectures DPLL et CDCL. L'exploration binaire et complète d'un solveur DPLL nous permet de savoir pour chaque nœud de l'arbre si il est UNSAT ou non, ce qui nous permet de créer des entrées de cache au niveau des nœuds de l'arbre, alors que l'exploration non chronologique des solveurs CDCL nous force à créer des entrées au niveau des feuilles de l'arbre. Cette différence a un fort impact sur le contenu du cache. Par ailleurs, si nous souhaitons intégrer le cache directement dans le solveur, nous devons être capables de générer un conflit si un isomorphisme est détecté.

Nous avons implémenté cette approche à l'intérieur du solveur MiniSat et nous l'avons testée sur un certain nombre d'instances incohérentes des compétitions SAT'02 et SAT'03. Les approches avec le cache intégré directement dans le solveur présentent de bien meilleurs résultats que celles à base de post-traitement et les approches utilisant des isomorphismes classiques résolvent plus d'instances que celles implémentant des isomorphismes étendus, même si il est parfois possible d'obtenir de plus petits arbres de recherche avec ces dernières. Pour certaines familles d'instances, nous arrivons à obtenir de bons taux de compression et même des arbres ayant une forme de peigne pour certaines instances. Nous sommes même capables de résoudre efficacement certaines instances que MiniSat ne parvient pas à résoudre en plus de quatre heures. Cependant, ces bons résultats ont été obtenus pour des instances structurées présentant de nombreuses symétries ou similarités et il n'est pas sûr que notre approche puisse fonctionner sur un panel plus large d'instances.

Conclusion Générale

L'intelligence artificielle est plus que jamais présente dans nos vies et dans la société. Celle-ci touche également de nombreux domaines, y compris ceux touchant aux vies humaines comme la médecine. Dans ce contexte, il est naturel de demander à ce qu'un programme informatique soit capable de justifier les résultats qu'il a obtenus afin de vérifier que tout s'est bien passé comme prévu et ainsi d'avoir une confiance accrue dans ces résultats. Ce besoin concerne tout type de programme informatique et a d'ailleurs été exprimé dans un certain nombre de textes de loi. C'est dans ce contexte que nous nous sommes intéressés à l'explication des résultats donnés par les solveurs SAT. Générer des explications pour des instances cohérentes est soit facile (montrer la satisfaction de toutes les clauses, obtenir un impliquant premier), soit impossible à expliquer d'un point de vue logique quand ce sont des artéfacts du solveur (décisions), soit se ramène à l'explication sur des instances incohérentes (prouver l'implication de formule). De ce fait, nous nous sommes focalisés sur le cadre des instances incohérentes.

Au cours de la thèse, nous avons donc adopté le contexte suivant : si nous pouvons identifier comme UNSAT certaines branches d'un arbre de recherche incohérent et de taille exponentielle, alors nous pouvons élaguer un certain nombre de branches et donc réduire sa taille. De cette manière, si l'on est capable d'obtenir une compression importante de l'arbre de recherche, celui-ci peut éventuellement être de taille convenable pour un utilisateur humain. Un point imporant à noter est que nous nous focalisons en priorité sur les posiblités de compression et non la complexité de calcul. Ainsi, nous n'excluons pas l'utilisation de techniques coûteuses comme l'appel à des oracles NP.

Au cours de la thèse, nous avons principalement étudié deux approches. Dans un premier temps, nous nous sommes intéressés à la détection de formules incohérentes classiques telles que les pigeons au cours de la recherche. Ces formules ont la particularité de pouvoir être expliquées facilement et indépendamment. Dans notre cas, nous avons proposé un algorithme pour détecter des problèmes de pigeons, qui sont des problèmes d'appariements simples à expliquer à l'utilisateur. Cet algorithme repose sur la détection syntaxique des contraintes *at least 1* et sur la détection sémantique des exclusions mutuelles du problème des pigeons. Ensuite, nous avons aussi étudié une approche basée sur la détection syntaxique de formules déjà prouvées UNSAT par le solveur. Cette fois-ci, nous avons utilisé un système de cache inspiré par le fonctionnement des compteurs de modèles, mais spécialisé au cadre des instances incohérentes, permettant le renommage des littéraux et détectant l'inclusion de l'entrée de cache dans la formule courante. L'idée ici est de régulièrement enregistrer dans le cache des sous-formules prouvées UNSAT par le solveur et de tenter de les reconnaître à un autre moment de la recherche. Nous utilisons des problèmes d'isomorphismes de sous-graphes pour reconnaître les entrées de cache. Cette approche a été implémentée dans les deux architectures DPLL et CDCL. Nous avons proposé une méthode pour reconnaître que la sous-formule courante est isomorphe à une entrée du cache (encodage classique) mais aussi pour reconnaître que cette sous-formule courante est isomorphe à une entrée du cache simplifiée par une partie de l'interprétation courante (encodage étendu).

Cependant, les approches proposées ont une complexité très élevée. En effet, d'un côté l'approche de détection de problèmes de pigeons implique de nombreuses combinatoires, et de l'autre la reconnaissance de sous-formules déjà prouvées UNSAT nécessite de régulièrement résoudre des problèmes NP-complets d'isomorphismes de sous-graphes. De ce fait, nous ne parvenons à résoudre qu'un nombre limité d'instances et nous n'obtenons de bons résultats que sur des instances provenant de familles particulières et très structurées. Par contre, sur ces instances, la compression est impressionnante et donne des preuves qui sont de taille réellement assimilable par l'utilisateur. De plus, le point de départ des approches proposées repose sur le fait que dans un arbre de recherche UNSAT de taille exponentielle, un même sous-ensemble incohérent de clauses va être exploré de manière répétée. Or, il peut s'avérer que cette hypothèse ne soit pas correcte puisqu'il est possible de construire des instances UNSAT où cela n'est pas le cas. Cela peut donc expliquer les résultats mitigés que nous avons obtenus. Pour les instances que nous parvenons à résoudre, nous arrivons tout de même à obtenir de bons taux de compression et des arbres de recherche courts (et même une forme de peigne pour certains d'entre eux).

Différentes pistes de futures recherches peuvent être envisagées pour améliorer les diverses approches présentées. Par exemple, concernant notre approche de détection de sous-formules déjà prouvées UNSAT, nous nous sommes basés sur l'utilisation d'un système de cache mais nous n'avons pas implémenté la possibilité de supprimer les entrées qui n'ont plus l'air d'être utiles. Cela pourrait être une bonne amélioration de notre approche. Il peut aussi être envisagé de tester d'autres heuristiques, et notamment celles implémentées dans les compteurs de modèles. En effet, les résultats obtenus dépendent grandement de l'heuristique utilisée au cours de la recherche puisque celle-ci définit la forme de l'arbre de recherche. Concernant maintenant la détection d'isomorphismes de sous-graphes, il pourrait être intéressant de collecter certaines informations pendant un appel au GSS et d'essayer de réutiliser ces informations au cours d'un futur appel. Utiliser le GSS de manière incrémentale pourrait permettre d'améliorer le temps d'exécution. Enfin, nous sommes aussi intéressés par la recherche d'autres formes de redondances au cours de la recherche afin de compresser des arbres de recherche UNSAT de manière plus générale.

Nous envisageons également d'utiliser nos techniques développées pour l'explication afin de réduire la taille d'un certificat d'incohérence en produisant une preuve au format FRAT ou veriPB. Cela nécessite de gérer le renommage des littéraux dans le format de preuve.

Les codes des différents programmes présentés sont disponibles sur un Software Heritage [3] et les résultats obtenus sont disponibles dans une archive Zenodo [4].

3. https ://archive.softwareheritage.org/swh :1 :dir :f374a7888c779e450f42bb56e7117e1719d71dbd ;origin=https ://github.com/antektek/TheseAnthonyBlommeCodes ;visit=swh :1 :snp :61d0fa234801b591b64e1f5f0a99d11620d72beb ;anchor=swh :1 :rev :c300198b0c9222a3d5b34a615a4446e28fcb8a94

4. https ://zenodo.org/records/10390156

Bibliographie

[ABC+19] Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z. Salamon, Josep Suy, et Mateu Villaret. « Automatic Detection of At-Most-One and Exactly-One Relations for Improved SAT Encodings of Pseudo-Boolean Constraints ». Dans Thomas Schiex et Simon de Givry, éditeurs, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 de *Lecture Notes in Computer Science*, pages 20–36. Springer, 2019. 1.2.4

[ABH+08] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, et Lakhdar Sais. « A Generalized Framework for Conflict Analysis ». Dans Hans Kleine Büning et Xishun Zhao, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 de *Lecture Notes in Computer Science*, pages 21–27. Springer, 2008. 1.2.3

[ACTZ12] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, et Stefano Zacchiroli. « Dependency solving : A separate concern in component evolution management ». *J. Syst. Softw.*, 85(10) :2228–2240, 2012. 1.2.1

[Ale04] Michael Alekhnovich. « Mutilated chessboard problem is exponentially hard for resolution ». *Theor. Comput. Sci.*, 310(1-3) :513–525, 2004. 4.6.3

[AS18] Gilles Audemard et Laurent Simon. « On the Glucose SAT Solver ». *Int. J. Artif. Intell. Tools*, 27(1) :1840001 :1–1840001 :25, 2018. 1.2.1, 1.2.4

[ASM06] Fadi A. Aloul, Karem A. Sakallah, et Igor L. Markov. « Efficient Symmetry Breaking for Boolean Satisfiability ». *IEEE Trans. Computers*, 55(5) :549–558, 2006. 1.2.4

[BCH22] Seulkee Baek, Mario Carneiro, et Marijn J. H. Heule. « A Flexible Proof Format for SAT Solver-Elaborator Communication ». *Log. Methods Comput. Sci.*, 18(2), 2022. 1.2.5

[BF22] Armin Biere et Mathias Fleury. « Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022 ». Dans Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, et Martin Suda, éditeurs, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 de *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022. 1.2.1

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, et Maximillian Heisinger. « CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020 ». Dans Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, et Martin Suda, éditeurs, *Proc. of SAT Competition*

2020 – Solver and Benchmark Descriptions, volume B-2020-1 de *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020. 1.2.1

[BFLW18] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, et Christoph Weidenbach. « A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality ». *J. Autom. Reason.*, 61(1-4) :333–365, 2018.

[BFW23] Armin Biere, Nils Froleyks, et Wenxi Wang. « CadiBack : Extracting Backbones with CaDiCaL ». Dans Meena Mahajan et Friedrich Slivovsky, éditeurs, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 de *LIPIcs*, pages 3 :1–3 :12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. 2.3.1

[BGG21] Bart Bogaerts, Emilio Gamba, et Tias Guns. « A framework for step-wise explaining how to solve constraint satisfaction problems ». *Artif. Intell.*, 300 :103550, 2021. 2.3.1

[BHvMW21] Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs. *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. 1.1, 1.2.1

[Bie14] Armin Biere. « Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling ». Dans Daniel Le Berre, éditeur, *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 de *EPiC Series in Computing*, page 88. EasyChair, 2014. 1.2.1

[Bie21] Armin Biere. Bounded Model Checking. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 739–764. IOS Press, 2021. 1.2.1

[BLLM14] Armin Biere, Daniel Le Berre, Emmanuel Lonca, et Norbert Manthey. « Detecting Cardinality Constraints in CNF ». Dans Carsten Sinz et Uwe Egly, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 de *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014. 1.2.4, 3.1

[BLPR23] Anthony Blomme, Daniel Le Berre, Anne Parrain, et Olivier Roussel. « Compressing UNSAT Search Trees with Caching ». Dans Ana Paula Rocha, Luc Steels, et H. Jaap van den Herik, éditeurs, *Proceedings of the 15th International Conference on Agents and Artificial Intelligence, ICAART 2023, Volume 3, Lisbon, Portugal, February 22-24, 2023*, pages 358–365. SCITEPRESS, 2023. 4

[BM11] Anton Belov et João Marques-Silva. « Accelerating MUS extraction with recursive model rotation ». Dans Per Bjesse et Anna Slobodová, éditeurs, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 37–40. FMCAD Inc., 2011. 1.2.5

[CGLR96] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, et Amitabha Roy. « Symmetry-Breaking Predicates for Search Problems ». Dans Luigia Carlucci Aiello, Jon Doyle, et Stuart C. Shapiro, éditeurs, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and*

Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996, pages 148–159. Morgan Kaufmann, 1996. 1.2.4, 4.2

[Coo71] Stephen A. Cook. « The Complexity of Theorem-Proving Procedures ». Dans Michael A. Harrison, Ranan B. Banerji, et Jeffrey D. Ullman, éditeurs, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. 1.1, 1.2.1, 1.2.1, 4.2, 4.4, 4.6.4

[DBBD16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, et Marc Denecker. « Improved Static Symmetry Breaking for SAT ». Dans Nadia Creignou et Daniel Le Berre, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 de *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016. 1.2.4, 4.2

[dKvMW00] Etienne de Klerk, Hans van Maaren, et Joost P. Warners. « Relaxations of the Satisfiability Problem Using Semidefinite Programming ». *J. Autom. Reason.*, 24(1/2) :37–65, 2000. 4.6.3

[DLL62] Martin Davis, George Logemann, et Donald W. Loveland. « A machine program for theorem-proving ». *Commun. ACM*, 5(7) :394–397, 1962. 1.2.2

[DP60] Martin Davis et Hilary Putnam. « A Computing Procedure for Quantification Theory ». *J. ACM*, 7(3) :201–215, 1960. 1.1, 1.2.1, 1.2.1, 1.2.2

[EB05] Niklas Eén et Armin Biere. « Effective Preprocessing in SAT Through Variable and Clause Elimination ». Dans Fahiem Bacchus et Toby Walsh, éditeurs, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 de *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. 1.2.4

[ES03] Niklas Eén et Niklas Sörensson. « An Extensible SAT-solver ». Dans Enrico Giunchiglia et Armando Tacchella, éditeurs, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 de *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. 1.2.3, 1.2.3, 1.2.4, 1.2.5, 4.6.1

[FB22] Mathias Fleury et Armin Biere. « Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses ». *CoRR*, abs/2207.13577, 2022. 1.2.5

[FHH21] Johannes Klaus Fichte, Markus Hecher, et Florim Hamiti. « The Model Counting Competition 2020 ». *ACM J. Exp. Algorithmics*, 26 :13 :1–13 :26, 2021. 1.3

[FHI+21] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, et Martin Suda. « SAT Competition 2020 ». *Artif. Intell.*, 301 :103572, 2021. 1.2.5

[GJ79] M. R. Garey et David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 1.2.1

[GMP06] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « Extracting MUSes ». Dans Gerhard Brewka, Silvia Coradeschi, Anna Perini, et Paolo Traverso, éditeurs, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 de *Frontiers in Artificial Intelligence and Applications*, pages 387–391. IOS Press, 2006. 1.2.5

[GMP07] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « MUST : Provide a Finer-Grained Explanation of Unsatisfiability ». Dans Christian Bessiere, éditeur, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 de *Lecture Notes in Computer Science*, pages 317–331. Springer, 2007. 1.2.5

[GMP08] Éric Grégoire, Bertrand Mazure, et Cédric Piette. « On Approaches to Explaining Infeasibility of Sets of Boolean Clauses ». Dans *20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), November 3-5, 2008, Dayton, Ohio, USA, Volume 1*, pages 74–83. IEEE Computer Society, 2008. 1.2.5

[GN21] Stephan Gocht et Jakob Nordström. « Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs ». Dans *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. 2.3.2

[GSS21] Carla P. Gomes, Ashish Sabharwal, et Bart Selman. Model Counting. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 993–1014. IOS Press, 2021.

[Hak85] Armin Haken. « The Intractability of Resolution ». *Theor. Comput. Sci.*, 39 :297–308, 1985. 3.1

[HD04] Jinbo Huang et Adnan Darwiche. « Using DPLL for Efficient OBDD Construction ». Dans *SAT 2004*, 2004. 4.2

[HJS09] Youssef Hamadi, Saïd Jabbour, et Lakhdar Sais. « ManySAT : a Parallel SAT Solver ». *J. Satisf. Boolean Model. Comput.*, 6(4) :245–262, 2009. 1.2.5

[HJS19] Marijn J. H. Heule, Matti Järvisalo, et Martin Suda. « SAT Competition 2018 ». *J. Satisf. Boolean Model. Comput.*, 11(1) :133–154, 2019. 4.6.3

[HJW13] Marijn Heule, Warren A. Hunt Jr., et Nathan Wetzler. « Trimming while checking clausal proofs ». Dans *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013. 1.2.5

[HK17] Marijn J. H. Heule et Oliver Kullmann. « The science of brute force ». *Commun. ACM*, 60(8) :70–79, 2017. 1.2.1, 2.3.2

[HKB19] Marijn J. H. Heule, Benjamin Kiesl, et Armin Biere. « Clausal Proofs of Mutilated Chessboards ». Dans Julia M. Badger et Kristin Yvonne Rozier, éditeurs, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 de *Lecture Notes in Computer Science*, pages 204–210. Springer, 2019.

[HKSB17] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, et Armin Biere. « PRuning Through Satisfaction ». Dans Ofer Strichman et Rachel Tzoref-Brill, éditeurs, *Hardware and Software : Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, volume 10629 de *Lecture Notes in Computer Science*, pages 179–194. Springer, 2017. 4.2

[HKWB11] Marijn Heule, Oliver Kullmann, Siert Wieringa, et Armin Biere. « Cube and Conquer : Guiding CDCL SAT Solvers by Lookaheads ». Dans Kerstin Eder, João Lourenço, et Onn Shehory, éditeurs, *Hardware and Software : Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 de *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011. 1.2.5

[HM20] Clément Henin et Daniel Le Métayer. « A Multi-layered Approach for Tailored Black-Box Explanations ». Dans Alberto Del Bimbo, Rita Cucchiara, Stan Sclaroff, Giovanni Maria Farinella, Tao Mei, Marco Bertini, Hugo Jair Escalante, et Roberto Vezzani, éditeurs, *Pattern Recognition. ICPR International Workshops and Challenges - Virtual Event, January 10-15, 2021, Proceedings, Part III*, volume 12663 de *Lecture Notes in Computer Science*, pages 5–19. Springer, 2020. 2.2

[HS18] Youssef Hamadi et Lakhdar Sais, éditeurs. *Handbook of Parallel Constraint Reasoning*. Springer, 2018. 1.2.5

[JLM15] Mikolás Janota, Inês Lynce, et Jo ao Marques-Silva. « Algorithms for computing backbones of propositional formulae ». *AI Commun.*, 28(2) :161–177, 2015. 2.3.1

[JO01] Narendra Jussien et Samir Ouis. « User-friendly explanations for constraint programming ». Dans Anthony J. Kusalik, éditeur, *Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01), Paphos, Cyprus, December 1, 2001*, 2001. (document)

[JT96] David S. Johnson et Michael A. Trick, éditeurs. *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 de *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 1996. 1.2.1

[KJ21] Tuukka Korhonen et Matti Järvisalo. « Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters (Short Paper) ». Dans Laurent D. Michel, éditeur, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 de *LIPIcs*, pages 8 :1–8 :11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 1.3

[Kro21] Daniel Kroening. Software Verification. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 791–818. IOS Press, 2021. 1.2.1

[Kul21] Oliver Kullmann. Fundaments of Branching Heuristics. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 351–390. IOS Press, 2021. 1.2.2

[LGPC16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, et Krzysztof Czarnecki. « Learning Rate Based Branching Heuristic for SAT Solvers ». Dans Nadia Creignou et Daniel Le Berre, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 de *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016. 1.2.1, 1.2.3

[LM04] Inês Lynce et João Marques-Silva. « On Computing Minimum Unsatisfiable Cores ». Dans *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. 1.2.5

[LM17] Jean-Marie Lagniez et Pierre Marquis. « An Improved Decision-DNNF Compiler ». Dans Carles Sierra, éditeur, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. 1.3

[LP10] Daniel Le Berre et Anne Parrain. « The Sat4j library, release 2.2 ». *J. Satisf. Boolean Model. Comput.*, 7(2-3) :59–6, 2010. 2.3.1

[LS03] Daniel Le Berre et Laurent Simon. « The Essentials of the SAT 2003 Competition ». Dans Enrico Giunchiglia et Armando Tacchella, éditeurs, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 de *Lecture Notes in Computer Science*, pages 452–467. Springer, 2003. 3.2.4, 4.6.4

[LS08] Mark H. Liffiton et Karem A. Sakallah. « Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints ». *J. Autom. Reason.*, 40(1) :1–33, 2008. 1.2.5

[LSZ93] Michael Luby, Alistair Sinclair, et David Zuckerman. « Optimal Speedup of Las Vegas Algorithms ». *Inf. Process. Lett.*, 47(4) :173–180, 1993. 1.2.3

[MBCK18] Hakan Metin, Souheib Baarir, Maximilien Colange, et Fabrice Kordon. « CDCLSym : Introducing Effective Symmetry Breaking in SAT Solving ». Dans *TACAS 2018, ETAPS*, pages 99–114, 2018. 1.2.4, 4.2

[Mil17] Tim Miller. « Explanation in Artificial Intelligence : Insights from the Social Sciences ». *CoRR*, abs/1706.07269, 2017. 2.2

[MLM21] João Marques-Silva, Inês Lynce, et Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. 1.2.3

[MM20] João Marques-Silva et Carlos Mencía. « Reasoning About Inconsistent Formulas ». Dans Christian Bessiere, éditeur, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4899–4906. ijcai.org, 2020.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, et Sharad Malik. « Chaff : Engineering an Efficient SAT Solver ». Dans *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. 1.2.3, 1.2.3

[MPT20] Ciaran McCreesh, Patrick Prosser, et James Trimble. « The Glasgow Subgraph Solver : Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants ». Dans Fabio Gadducci et Timo Kehrer, éditeurs, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, volume 12150 de *Lecture Notes in Computer Science*, pages 316–324. Springer, 2020. 4.6.1

[NR18] Alexander Nadel et Vadim Ryvchin. « Chronological Backtracking ». Dans Olaf Beyersdorff et Christoph M. Wintersteiger, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 de *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018. 1.2.3

[OMA+04] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, et Igor L. Markov. « AMUSE : a minimally-unsatisfiable subformula extractor ». Dans Sharad Malik, Limor Fix, et Andrew B. Kahng, éditeurs, *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 518–523. ACM, 2004. 1.2.5

[PG86] David A. Plaisted et Steven Greenbaum. « A Structure-Preserving Clause Form Translation ». *J. Symb. Comput.*, 2(3) :293–304, 1986. 1.2.1

[Rin21] Jussi Rintanen. Planning and SAT. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 765–789. IOS Press, 2021.

[Sak21] Karem A. Sakallah. Symmetry and Satisfiability. Dans Armin Biere, Marijn Heule, Hans van Maaren, et Toby Walsh, éditeurs, *Handbook of Satisfiability - Second Edition*, volume 336 de *Frontiers in Artificial Intelligence and Applications*, pages 509–570. IOS Press, 2021. 1.2.4

[SB09] Niklas Sörensson et Armin Biere. « Minimizing Learned Clauses ». Dans Oliver Kullmann, éditeur, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 de *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009. 1.2.4

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, et Toniann Pitassi. « Combining Component Caching and Clause Learning for Effective Model Counting ». Dans *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. 1.3, 4.2

[SBDD23] Sabrine Saouli, Souheib Baarir, Claude Dutheillet, et Jo Devriendt. « CosySEL : Improving SAT Solving Using Local Symmetries ». Dans *VMCAI 2023*, pages 252–266, 2023. 1.2.4, 4.2

[SBK05] Tian Sang, Paul Beame, et Henry A. Kautz. « Heuristics for Fast Exact Model Counting ». Dans Fahiem Bacchus et Toby Walsh, éditeurs, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 de *Lecture Notes in Computer Science*, pages 226–240. Springer, 2005. 1.3

[SLH05] Laurent Simon, Daniel Le Berre, et Edward A. Hirsch. « The SAT2002 competition ». *Ann. Math. Artif. Intell.*, 43(1) :307–342, 2005. 1.2.1, 3.2.4, 4.6.4

[SS99] João P. Marques Silva et Karem A. Sakallah. « GRASP : A Search Algorithm for Propositional Satisfiability ». *IEEE Trans. Computers*, 48(5) :506–521, 1999. 1.2.3

[TCJ08] Emina Torlak, Felix Sheng-Ho Chang, et Daniel Jackson. « Finding Minimal Unsatisfiable Cores of Declarative Specifications ». Dans Jorge Cuéllar, T. S. E. Maibaum, et Kaisa Sere, éditeurs, *FM 2008 : Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 de *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008. 1.2.5

[Thu06] Marc Thurley. « sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP ». Dans Armin Biere et Carla P. Gomes, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 de *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006. 1.3

[Tse66] G.S. Tseytin. « On the complexity of derivation in propositional calculus ». 1966. 1.2.1

[WHJ14] Nathan Wetzler, Marijn Heule, et Warren A. Hunt Jr.. « DRAT-trim : Efficient Checking and Trimming Using Expressive Clausal Proofs ». Dans Carsten Sinz et Uwe Egly, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 de *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. 1.2.5

[WKS01] Jesse Whittemore, Joonyoung Kim, et Karem A. Sakallah. « SATIRE : A New Incremental Satisfiability Engine ». Dans *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 542–545. ACM, 2001. 1.2.5

[ZM03a] Lintao Zhang et Sharad Malik. « Extracting small unsatisfiable cores from unsatisfiable boolean formulas ». Dans *SAT 2003 pre-proceedings*, 2003. Presented at SAT2003 but not part of the post-proceedings. 1.2.5, 4.3

[ZM03b] Lintao Zhang et Sharad Malik. « Validating SAT Solvers Using an Independent Resolution-Based Checker : Practical Implementations and Other Applications ». Dans *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society, 2003.