

Contraintes de cardinalité cachées dans les preuves d'insatisfaisabilité

Valentin Montmirail, Marie Pelleau, Jean-Charles Régin, Laurent Simon

▶ **To cite this version:**

Valentin Montmirail, Marie Pelleau, Jean-Charles Régin, Laurent Simon. Contraintes de cardinalité cachées dans les preuves d'insatisfaisabilité. 15es Journées Francophones de Programmation par Contraintes, Jun 2019, Albi, France. hal-02271389

HAL Id: hal-02271389
https://hal.univ-cotedazur.fr/hal-02271389

Submitted on 26 Aug 2019

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

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

Contraintes de cardinalité cachées dans les preuves d'insatisfaisabilité

Valentin Montmirail[1] **Marie Pelleau**[1] **Jean-Charles Régin**[1] **Laurent Simon**[2]
[1] Université Côte d'Azur, CNRS, I3S, Nice, France
[2] Université de Bordeaux, CNRS, LaBRI, Talence, France
prénom.nom@univ-cotedazur.fr lsimon@labri.fr

Résumé

Les solveurs SAT sont utilisés avec succès dans de nombreuses applications combinatoires et du monde réel. Il n'est maintenant pas rare de lire qu'une preuve mathématique implique des centaines de gigaoctets de traces de solveur SAT. L'ampleur de la preuve commence à constituer une véritable limite à l'approche globale. Dans ce travail, nous proposons de rechercher des contraintes de haut niveau dans les preuves UNSAT. Un travail similaire a été proposé il y a quelques annéespour les contraintes de cardinalité les plus simples. Nous étendons cette idée à un cas plus général (sans limitation sur les bornes des contraintes de cardinalité) en généralisant l'algorithme de Bron & Kerbosh (pour l'énumération de cliques dans les graphes) aux hypergraphes. Nous démontrons expérimentalement la capacité de notre approche à trouver des contraintes de cardinalité dans les grandes preuves, ouvrant ainsi un nouveau moyen de générer des preuves plus courtes et compréhensibles pour les problèmes difficiles.

Abstract

SAT solvers are successfully used in many real-world and combinatoric applications. It is now not uncommon to read that a mathematical proof implied hundreds of gigabytes of SAT solver traces. The size of the proof by itself begins to be a real limit to the whole approach. In this work, we propose to search for higher-level constraints in UNSAT proofs. A similar work was proposed a few years ago, but only on the original formulafor the most simple cardinality constraints. We extend this idea to a more general case (with no limitations on the bounds of the cardinality constraints) by generalizing the Bron & Kerbosh algorithm (for clique enumeration in graphs) to hypergraphs. We experimentaly demonstrate the ability of our approach to find for cardinality constraints in large proofs, opening a new way of generating more shorter and human-understandable UNSAT proofs of hard problems.

1 Introduction

Le succès des solveurs SAT dans la résolution de problèmes réels et combinatoires repose sur leur capacité à construire efficacement de preuves UNSAT. La plupart des solveurs actuels, basés sur le framework *Conflict-Driven Clause Learning* (CDCL) [22], peuvent produire des dizaines de milliers de lemmes par seconde, grâce à une intégration de nombreux ingrédients (2-Watched Literals, VSIDS, prétraitement, redémarrages, *etc.*) Cependant, même si les ingrédients sont bien connus, la stratégie globale d'un solveur est encore inconnue et est difficilement expliquée. En conséquence, quand une formule s'avère être UNSAT, le seul moyen de prouver le résultat est de "rejouer" les étapes élémentaires du solveur, en veillant à ce que chaque clause de la preuve puisse être déduite par résolution.

Beaucoup de progrès ont été réalisés ces dernières années dans la génération, la manipulation et la vérification des preuves UNSAT. Cependant, la plupart des efforts sont consacrés à la conception d'outils [13, 26] afin de garantir de la correction des solveurs SAT. La limite de ces approches est néanmoins la taille et l'expressivité de la preuve. Jusqu'à présent, il n'est pas possible de l'exploiter pour comprendre la preuve ou même l'expliquer. Il n'est maintenant plus rare de voir qu'une preuve mathématique implique des centaines de tera-octets de traces d'un solveur [16, 14]. Cependant, vérifier que la preuve est correcte ne donne aucun indice sur le fonctionnement du solveur. Plus important encore, la taille de la preuve, elle-même commence à être une véritable limite à l'ensemble de l'approche.

Dans ce travail, nous proposons de rechercher des contraintes de haut niveau dans les preuves. Nous montrons que, même si ces contraintes ne sont pas présentes dans la formule originale, les solveurs SAT ajoutent

généralement ces contraintes *in intenso* au cours de leur recherche. Il existe de nombreux travaux sur la détection d'informations dans des CNF. Nous pouvons citer en exemple l'idée de détecter les contraintes de cardinalité en utilisant la propagation unitaire [4], des équations booléennes [23], des variables dépendantes [12], *etc.* Cependant, il y a moins de travaux concernant l'analyse de la preuve générée. L'un d'eux était centré sur le pourcentage de temps consacré par le solveur à des calculs inutiles pour la preuve finale [24], un autre sur le lien entre les mesures de complexité de la preuve et la difficulté des instances [18]. Un travail récent [10] analyse le graphe de dépendance généré par la résolution pour détecter quel ensemble de clauses apprises est nécessaire pour dériver la contradiction finale.

Afin de trouver de la structure dans les preuves UNSAT, nous proposons de rechercher des contraintes de cardinalité dans la preuve DRAT [26] produite par un solveur CDCL, en étendant les approches précédentes à un cas plus général (sans limite sur les bornes des contraintes de cardinalité). Notre algorithme de détection est une première contribution : la taille de la formule dans laquelle la recherche est lancée est nettement plus grande que dans les travaux précédents. Ceci est rendu possible grâce à une généralisation de l'algorithme de Bron & Kerbosh (pour l'énumération de cliques dans les graphes) [8] aux hypergraphes que nous proposons dans cet article. Nous montrons expérimentalement que notre approche peut trouver des contraintes de cardinalité dans de grandes preuves. De plus, nous obtenons des preuves étonnamment plus courtes et compréhensibles sur certains cas typiques. Il est étonnant de constater que, même lors de la résolution d'instances aléatoires, les solveurs ont tendance à produire de nombreuses contraintes de cardinalité dans les preuves. Nous nous attendons à ce que cette découverte ouvre de nouvelles améliorations afin de comprendre comment les solveurss se comportent.

2 Préliminaires

Définissons quelques notions sur la logique propositionnelle et la théorie des graphes.

2.1 Logique propositionnelle et SAT

Nous considérons que le lecteur est familiarisé avec la logique propositionnelle et les bases des solveurs SAT de type *conflict-driven clause learning* (CDCL). Nous orientons le lecteur intéressé vers [5] pour une introduction approfondie du sujet. Nous présentons quelques notions essentielles pour plus de clarté.

Remarque 2.1 – Nous appelons clause *positive* (resp. *négative*), une clause contenant uniquement des littéraux positifs (resp. négatifs). Nous appelons *uniforme* une clause négative ou positive.

Les solveurs CDCL utilisent le principe de résolution à chaque analyse de conflit. Si le certificat pour les instances satisfiables est simplement l'affectation de variables évaluant la formule à vrai, le certificat pour les formules non satisfaisables s'appuie indirectement sur la règle de résolution. Chaque fois que le solveur CDCL apprend une nouvelle clause, le lemme obtenu par résolution est enregistré et fera partie de la preuve.

DRAT (deletion resolution asymmetric tautology) [26] est un exemple populaire de système de preuve clausal, le standard des preuves non satisfaisantes dans la résolution pratique de SAT, qui est une généralisation du format DRUP (deletion reverse unit propagation) [13]. DRAT autorise l'ajout d'une clause s'il s'agit d'une clause dite *Resolution Asymmetric Tautology* (RAT) [17]. Comme il est possible de vérifier efficacement si une clause est une RAT et que les RAT couvrent une grande partie des clauses redondantes, le système de preuve DRAT est très puissant. L'importance de la redondance est de certifier que si nous obtenons $\perp$ en ajoutant uniquement des lemmes qui ne changent pas la satisfiabilité de la formule, alors par transitivité, la formule est insatisfiable. RAT est défini comme suit :

Définition 2.1 (*Resolution Asymmetric Tautology* [17])**.** Soit Σ une formule et C une clause. Alors, C est une *resolution asymmetric tautology* en fonction de Σ s'il contient un littéral l tel que, pour chaque clause $d \in \Sigma|_{\overline{l}}$, la propagation unitaire sur la négation des littéraux de $(d \setminus \{l\})$ sur Σ dérive un conflit.

Il est relativement facile d'émettre une preuve DRAT à partir d'un solveur CDCL. Les solveurs CDCL gèrent une base de données contenant les clauses d'origine, et la création d'une preuve DRAT consiste généralement à imprimer chaque modification sur la base de données (*c.-à-d.* la suppression d'une clause ou l'ajout de nouvelles). Maintenant que nous avons introduit comment les solveurs SAT peuvent produire une preuve, passons en revue certaines notions de la théorie des graphes.

2.2 Théorie des graphes

La première notion dont nous avons besoin est la notion d'hyperarête, définie comme suit :

Définition 2.2 (Hypergraphe [3])**.** Soit $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ un ensemble fini. Un *hypergraphe* sur $\mathcal{X}$ est une famille $H = (E_1, E_2, \ldots, E_m)$ de sous-ensembles de $\mathcal{X}$ tel que :

$$E_i \neq \emptyset \qquad (i = 1, 2, \ldots, m) \tag{1}$$

$$\bigcup_{i=1}^{m} E_i = \mathcal{X} \tag{2}$$

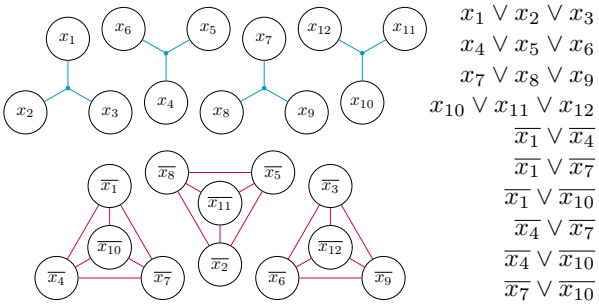


Figure 1 – Le problème des pigeonniers en encodage naïf et sa représentation en hypergraphe.

Les éléments de $\mathcal{X}$ sont appelés *sommets*, et les ensembles $E_1, E_2, \ldots, E_m$ sont les *arêtes* de l'hypergraphe (ou *hyperarêtes*). L'*ordre* de H est le nombre de sommets. Un hypergraphe est dit *simple* si aucune arête n'en contient une autre ($E_i \subseteq E_j \implies i = j$).

Définition 2.3 (Hypergraphe uniforme)**.** Le *rang* est $r(H) = \max_j |E_j|$, l'*anti-rang* est $s(H) = \min_j |E_j|$. H est un hypergraphe *uniforme* si $r(H) = s(H)$. Un hypergraphe simple de rang r est également appelé r-uniforme ou r-hypergraphe.

Définition 2.4 (Hypergraphe r-uniforme complet)**.** L'hypergraphe *r-uniforme complet* ou *r-complet*, noté K_n^r, est l'hypergraphe contenant n sommets dans lequel chaque sous-ensemble de taille r des sommets représentent une arête. On observe facilement que le nombre d'arêtes dans K_n^r est $\binom{n}{r} = \frac{n!}{r!(n-r)!}$

Remarque 2.2 – Dans la suite, nous appelons *clique* $\mathcal{C}$ d'ordre n un hypergraphe r-complet K_n^r. Par abus de notation, $\mathcal{C}$ correspond à un sous-ensemble de sommets.

Maintenant que nous avons rappelé suffisamment de définitions de la théorie des graphes, passons au codage en graphe d'une preuve SAT et à la détection des contraintes qui s'y trouvent. L'idée pour représenter la formule est : tout ensemble de clauses (*c.-à-d.* une formule d'entrée ou une preuve DRAT), peut être représenté à l'aide d'un hypergraphe où les sommets correspondent aux littéraux et les arêtes aux clauses.

Exemple 2.1 – Le fameux exemple des pigeonniers, avec l'encodage naïf, correspond à l'hypergraphe de la Fig. 1. Ici, il y a 4 pigeons et 3 pigeonniers.

3 Détection des contraintes de cardinalité

La détection des contraintes de cardinalité dans les formules exprimées en CNF avait déjà été proposée dans [4]. Le but de ce travail était d'accélérer le temps de résolution des instances difficiles à résoudre en proposant une approche à la volée. Les contraintes sont détectées lors de la recherche du solveur SAT et n'ont pas lors d'une approche post-mortem sur la preuve.

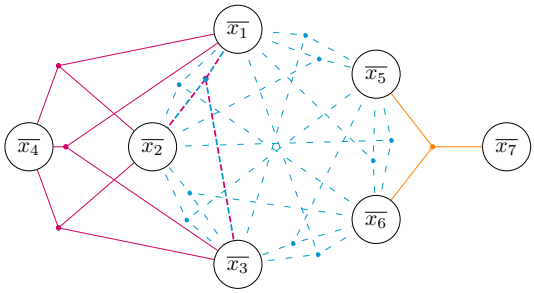


Figure 2 – Hypergraphe de la CNF de l'exemple 3.1.

3.1 Détection pour les clauses uniformes

Si on représente les clauses négatives binaires de l'encodage naïf de la contrainte AtMost-1 par un graphe, il correspond à une clique. Nous étendons cette affirmation aux clauses d'arité k. Une clause d'arité k ($\overline{x_1} \vee \overline{x_2} \vee \cdots \vee \overline{x_k}$) est équivalente à :

$$\begin{aligned}(1 - x_1) + \cdots + (1 - x_k) &\geq 1\\ k - x_1 - \cdots - x_k &\geq 1\\ k - 1 &\geq x_1 + \cdots + x_k\end{aligned}$$

En d'autres termes, Une clause d'arité k ($\overline{x_1} \vee \overline{x_2} \vee \cdots \vee \overline{x_k}$) est équivalente à une contrainte AtMost-$(k-1)$. Toute CNF peut être représentée sous la forme d'un hypergraphe dont les sommets correspondent aux littéraux et les arêtes aux clauses.

Exemple 3.1 – Considérons la CNF suivante :

$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$	$\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}$	$\overline{x_1} \vee \overline{x_2} \vee \overline{x_5}$
$\overline{x_1} \vee \overline{x_2} \vee \overline{x_6}$	$\overline{x_1} \vee \overline{x_3} \vee \overline{x_4}$	$\overline{x_1} \vee \overline{x_3} \vee \overline{x_5}$
$\overline{x_1} \vee \overline{x_3} \vee \overline{x_6}$	$\overline{x_1} \vee \overline{x_5} \vee \overline{x_6}$	$\overline{x_2} \vee \overline{x_3} \vee \overline{x_4}$
$\overline{x_2} \vee \overline{x_3} \vee \overline{x_5}$	$\overline{x_2} \vee \overline{x_3} \vee \overline{x_6}$	$\overline{x_2} \vee \overline{x_5} \vee \overline{x_6}$
$\overline{x_3} \vee \overline{x_5} \vee \overline{x_6}$	$\overline{x_5} \vee \overline{x_6} \vee \overline{x_7}$	

Cette instance peut être représentée par l'hypergraphe figure 2.

Proposition 3.1. *Dans un hypergraphe r-uniforme représentant les clauses négatives d'une CNF, une clique $\mathcal{C}$ d'ordre n avec $n \geq r$ correspond à une contrainte AtMost-$(r-1)$, autrement dit* $\sum_{x \in \mathcal{C}} x \leq r - 1$.

Démonstration.

Par récurrence. Dans un r-hypergraphe représentant les clauses négatives d'une CNF, une hyperarête est une clique particulière d'ordre r et correspond à une contrainte AtMost-$(r-1)$. Supposons que la propriété soit valable pour une clique d'ordre n. Soit $\mathcal{C} = \{x_1, x_2, \ldots, x_{n+1}\}$ une clique d'ordre $n+1$. On note $\mathcal{C}_{\setminus x}$ la clique $\mathcal{C}$ dans laquelle le sommet x et toutes les arêtes auxquelles il appartient ont été supprimés. Pour tout $x \in \mathcal{C}$, $\mathcal{C}_{\setminus x}$ est une clique d'ordre n. Ainsi, la clique $\mathcal{C}$ correspond à $n+1$ contraintes AtMost-$(r-1)$, et chaque sommet $x \in \mathcal{C}$ apparaît dans exactement n contraintes (toutes les cliques sauf $\mathcal{C}_{\setminus x}$). Si nous additionnons toutes les contraintes AtMost, nous obtenons :

$$\begin{aligned} nx_1 + nx_2 + \cdots + nx_{n+1} &\leq (n+1)(r-1) \\ x_1 + x_2 + \cdots + x_{n+1} &\leq \lfloor \tfrac{(n+1)}{n}(r-1) \rfloor \\ x_1 + x_2 + \cdots + x_{n+1} &\leq (r-1) \end{aligned}$$

Par conséquent, dans un r-hypergraphe représentant les clauses négatives de la CNF, une clique correspond à une contrainte AtMost-$(r-1)$. □

Exemple 3.2 – L'hypergraphe de la figure 2 a une clique d'ordre 5 ($\{\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_5}, \overline{x_6}\}$, en bleu), 6 cliques d'ordre 4 ($\{\overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}\}$, en rose, et les 5 incluses dans la clique d'ordre 5) et 25 cliques d'ordre 3 ($\{\overline{x_5}, \overline{x_6}, \overline{x_7}\}$, en orange, et les 24 incluses dans les cliques d'ordre 4). En ne gardant que les cliques maximales, nous obtenons 3 cliques correspondant aux contraintes suivantes :

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &\leq 2 \\ x_1 + x_2 + x_3 + x_5 + x_6 &\leq 2 \\ x_5 + x_6 + x_7 &\leq 2 \end{aligned}$$

Il est évident que ce principe fonctionne de manière similaire dans le cas d'une contrainte AtLeast-k. Une clause d'arité k $(x_1 \vee x_2 \vee \cdots \vee x_k)$ équivaut à $(x_1 + x_2 + \cdots + x_k \geq 1)$. En d'autres termes, une clause d'arité k $(x_1 \vee x_2 \vee \cdots \vee x_k)$ est équivalente à une contrainte AtLeast-1.

Proposition 3.2. *Dans un hypergraphe r-uniforme représentant les clauses positives d'une CNF, une clique $\mathcal{C}$ d'ordre n avec $n \geq r$ correspond à une contrainte AtLeast-$(n-r+1)$, autrement dit* $\sum_{x \in \mathcal{C}} x \geq n - r + 1$.

3.2 Rechercher toutes les cliques maximales

La recherche de cliques maximales est un problème difficile. Le problème CLIQUE est l'un des 21 problèmes complets de Karp et, un graphe à n sommets peut contenir jusqu'à $3^{\frac{n}{3}}$ cliques maximales [21].

Pour traiter la difficulté intrinsèque de notre problème, nous proposons ici de généraliser le fameux algorithme de Bron & Kerbosch (BK) [8] dans le cas des hypergraphes. À notre connaissance, cela n'a pas été fait auparavant. Comme indiqué dans [20], il est généralement supposé que cela ne peut pas être fait efficacement. En effet, la notion de voisin est plus complexe dans les hypergraphes que dans les graphes classiques, et sa généralisation n'est pas simple. Cependant, cela peut être décidé efficacement dans la pratique.

Algorithme 1 : Hyper-Bron-Kerbosch

```
1  Hyper-BK (Courant, Candidats, Exclus)
2      si Candidats = ∅ et Exclus = ∅ alors
3          signale Courant est une clique maximale
4      pour chaque sommet v dans Candidats
         faire
5          Courant ← Courant ∪ {v}
6          VoisinC ← γ(Courant)
7          Hyper-BK(Courant, Candidats ∩
             VoisinC, Exclus ∩ VoisinC)
8          Courant ← Courant \ v
9          Candidats ← Candidats \ {v}
10         Exclus ← Exclus ∪ {v}
```

Nous proposons la définition suivante du voisinage de clique afin de généraliser l'algorithme BK [8].

Définition 3.1 (Voisinage de clique)**.** Soit $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ un ensemble fini de sommets et soit $H = (E_1, E_2, \ldots, E_m)$ un hypergraphe. Pour une clique $\mathcal{C}$ et $\mathcal{Y} \subseteq \mathcal{X} \setminus \mathcal{C}$, nous notons par $\gamma(\mathcal{C})$, le *voisinage de clique* de $\mathcal{C}$, un ensemble de sommets tel que si un sommet est ajouté à $\mathcal{C}$, elle forme une nouvelle clique :

$$\gamma(\mathcal{C}) = \{y \in \mathcal{Y} \,|\, \forall N \subseteq \mathcal{C} \text{ avec } |N| \leq r-1, \\ \exists E_i \in H \text{ t.q. } N \cup \{y\} \in E_i\}$$

Dans cette définition, $\mathcal{Y}$ est introduit pour l'efficacité du calcul. Il peut toujours être défini par $\mathcal{X} \setminus \mathcal{C}$, nous proposons une amélioration à la fin de cette section. À partir de cette définition, nous proposons l'algorithme 1, basé sur l'algorithme BK [8], pour trouver toutes les cliques maximales dans un hypergraphe.

Lors du premier appel, *Courant* et *Exclus* ont pour valeur $\emptyset$, et *Candidats* contient tous les sommets de l'hypergraphe. *Courant* est le résultat temporaire qui est une clique, *Candidats*, l'ensemble des candidats possibles pouvant étendre la clique actuelle pour en former une nouvelle, et *Exclus* contient les sommets à partir desquels des cliques ont déjà été calculées (pour éviter de générer des cliques non maximales).

Il est clair que cet algorithme peut trouver toutes les cliques maximales dans un graphe simple, il n'est pas évident qu'il peut être étendu aussi facilement pour un

hypergraphe. Pour cela, prouvons qu'à chaque étape, *Courant* est une clique et que si elle est signalée, elle est maximale (correction). Prouvons aussi que toutes les cliques maximales sont trouvées (complétude).

Proposition 3.3. *À chaque étape de l'algorithme 1, Courant est une clique.*

Démonstration.

Un ensemble vide ou contenant un seul sommet est une clique. Supposons que *Courant* est une clique d'ordre $n \geq 2$. Soit *Candidats* est vide et nous revenons à un niveau supérieur où le dernier sommet ajouté est supprimé, et *Courant* reste une clique. Soit *Candidats* contient au moins un sommet v. *Candidats* est un sous-ensemble de $VoisinC$, alors par Déf. 3.1, v peut être ajouté à *Courant* de manière à former une nouvelle clique. □

Proposition 3.4. *Une clique est signalée par l'algorithme 1, si et seulement si elle est maximale*

Démonstration.

Après l'ajout d'un sommet à *Courant*, qui est une clique (Prop. 3.3), si *Candidats* devient vide, alors, soit *Exclus* est vide et *Courant* est maximale car aucun sommet ne peut être ajouté (ligne 3). Ou *Exclus* contient au moins un sommet v. *Exclus* est un sous-ensemble de $VoisinC$, alors par Def. 3.1, v peut être ajouté à *Courant* de manière à former une nouvelle clique. Donc la clique n'est pas maximale et Alg. 1 ne la signale pas. Il en va de même si le *Candidats* ne devient pas vide. □

Proposition 3.5. *Lorsqu'un candidat c est éliminé dans l'algorithme 1, il ne peut pas étendre Courant.*

Démonstration.

Un candidat c est supprimé grâce à l'intersection entre les ensembles *Candidats* et $VoisinC$. Si c n'est pas dans le voisinage de *Courant*, alors par Déf. 3.1, c ne peut pas augmenter *Courant* et est donc éliminé. □

À partir de ces propositions, nous pouvons maintenant proposer le théorème suivant :

Théorème 3.6. *L'algorithme 1 termine et trouve toutes les cliques maximales dans un hypergraphe.*

Démonstration.

La correction vient directement de Prop. 3.4, l'algorithme 1 ne renvoie que des cliques maximales. La complétude vient de Prop. 3.5, qui garantit que l'ensemble de tous les candidats possibles est testé étant donné qu'un candidat n'est éliminé que s'il ne peut pas faire partie de la clique actuelle. Ainsi, toutes les cliques sont essayées et signalées si elles sont maximales. Enfin, l'algorithme 1 se termine. Sinon, selon le lemme de Kőnig, *Candidats* est infini ou la pile d'appel est infinie. *Candidats* est un sous-ensemble de $VoisinC$ qui est un sous-ensemble de $\mathcal{X}$ qui est fini par Déf. 2.2. Ainsi, *Candidats* ne peut être infini. Comme la pile d'appels dépend de la taille de *Candidats*, elle ne peut pas être infinie. □

Comme indiqué lors de l'introduction de Def. 3.1, l'efficacité du calcul du voisinage de la clique dépend de l'ensemble des sommets considérés $\mathcal{Y}$. Nous proposons un filtrage effectué chaque fois que nous ajoutons un nouveau sommet v dans la clique. Pour tous les candidats c, nous savons qu'il nous faut au moins une hyperarête contenant v, c et les sous-ensembles de tous les sommets déjà présents dans la clique actuelle. En effet, considérons une clique actuelle d'ordre 4 dans un 3-hypergraphe : $\{x_1, x_2, x_3, x_4\}$, avec x_4 étant le dernier élément ajouté. Un sommet c reste candidat si les hyperarêtes suivantes existent $\{x_4, x_1, c\}, \{x_4, x_2, c\}, \{x_4, x_3, c\}$. Bien entendu, beaucoup plus d'hyperarêtes sont nécessaires pour étendre la clique à un ordre $n = 5$. Cependant, si une de ces hyperarêtes n'existent pas, il est certain que c ne peut pas étendre la clique et peut être ignoré.

Proposition 3.7. *Soit x_n le dernier sommet ajouté à Courant et prenons c un sommet avec $c \in \mathcal{X} \backslash Courant$. S'il existe de sous-ensemble $\texttt{comb} \subseteq Courant \setminus \{x_n\}$ avec $|\texttt{comb}| \leq r - 2$, de sorte qu'aucune hyperarête ne contienne $\{x_n, \texttt{comb}, c\}$, alors $c \notin Candidats$.*

Démonstration.

Directement de la définition 3.1. □

Une façon d'implémenter ce filtrage consiste à utiliser un filtre de Bloom [6]. C'est une structure de données probabiliste qui permet de vérifier si un élément est membre d'un ensemble. Des faux positives sont possibles, mais les faux négatifs ne le sont pas, une requête renvoie "éventuellement dans l'ensemble" ou "définitivement pas dans l'ensemble". Nous utilisons cette structure de données pour filtrer les candidats pour lesquels au moins un hyperarête est manquante, afin qu'ils ne soient pas considérés comme des candidats pour l'extension de la clique. La combinatoire peut être extrêmement volumineuse, il est plus efficace d'avoir des faux positifs à confirmer que de calculer le voisinage de la clique pour tous les sommets.

3.3 Cas des clauses mixtes

Dans la section précédente, nous avons émis l'hypothèse que nous avions affaire à des encodages de

$$
\begin{array}{ll}
 & x_1 + x_2 + x_4 + x_5 \leq 1 \\
\equiv & (x_1 + x_2 + x_3 \leq 1) \\
 & \wedge(\overline{x_3} + x_4 + x_5 \leq 1) \\
\equiv & (\overline{x_1} \vee \overline{x_2}) \\
 & (\overline{x_1} \vee \overline{x_3}) \quad \text{Missing :} \\
 & (\overline{x_2} \vee \overline{x_3}) \quad (\overline{x_1} \vee \overline{x_4}) \\
 & (x_3 \vee \overline{x_4}) \quad (\overline{x_1} \vee \overline{x_5}) \\
 & (x_3 \vee \overline{x_5}) \quad (\overline{x_2} \vee \overline{x_4}) \\
 & (\overline{x_4} \vee \overline{x_5}) \quad (\overline{x_2} \vee \overline{x_5})
\end{array}
$$

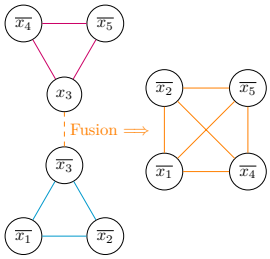


Figure 3 – Détection de clique avec clauses mixtes

cardinalité naïf en CNF : nous n'avions envisagé que des encodages avec des clauses uniformes. Cependant, cette hypothèse est clairement trop forte pour la recherche de contraintes de cardinalité dans une preuve ou simplement dans des cas plus complexes. Nous appelons *mixte* une clause faisant partie d'un encodage de contraintes de cardinalité contenant des littéraux positifs et négatifs. Pour expliquer comment nous pouvons contourner ce nouveau problème de détection, illustrons-le avec un exemple.

Dans la configuration de la figure 3, nous voyons qu'une recherche syntaxique simple échouera probablement à détecter une clique. Cependant, comme le montre la représentation graphique de ces clauses, nous pouvons *fusionner* les deux petites cliques pour en créer une plus grande. Ce principe s'appelle Merge AtMost-1 dans [4]. Nous proposons maintenant de généraliser ce résultat précédent aux contraintes AtMost-k et AtLeast-k. Tout d'abord, nous devons déterminer si notre approche peut récupérer des encodages *imbriqués*, qui sont probablement les encodages les plus courants pour les contraintes de cardinalité.

Remarque 3.1 – Toute contrainte de cardinalité $(x_1 + x_2 + \cdots + x_n \diamond k)$ avec $\diamond \in \{\leq, \geq, =\}$ en encodage imbriqué peut être retrouvée en fusionnant les littéraux utilisés pour scinder la contrainte d'origine, *c.-à-d.* en fusionnant des littéraux qui apparaissent positivement et négativement dans chaque paire de contraintes.

$$
\begin{array}{ll}
 & x_1 + x_2 + \cdots + x_n \diamond k \\
\equiv & (x_1 + \cdots + x_n + y_1 + \cdots + y_m \diamond c_1) \\
 & \wedge(\overline{y_1} + \cdots + \overline{y_m} \diamond c_2) \\
\equiv & \cdots \\
\equiv & (x_1 + y_1 + \cdots + y_m \diamond c_1) \wedge (x_2 + \overline{y_1} \diamond c_2) \wedge \ldots \\
 & \wedge(x_n + \overline{y_m} \diamond c_n)
\end{array}
$$

avec $\sum_i c_i = k + m$

Tout cela fonctionne bien lorsque la CNF (et la preuve) est bien formée, *c.-à-d.* qu'elle ne contient que des clauses de même taille, bien adaptées pour retrouver une clique. Cependant, cela ne suffit pas pour couvrir tous les cas, en particulier lors de la recherche dans les preuves, où les clauses unitaires apparaissent souvent. L'idée est de les utiliser comme des hyperarêtes spéciales afin de détecter de plus grandes cliques, comme ce qui a été fait dans la section 4.1 dans [4].

Remarque 3.2 – Toute clause de taille k est une clause de taille k' avec $1 \leq k \leq k'$. Cette clause peut être obtenue en introduisant autant de $\bot$ littéraux que nécessaire, *c.-à-d.* en considérant $C' = (C \vee \bot \vee \cdots \vee \bot)$, avec C' de taille k'.

En effet, comme expliqué dans la remarque 3.2, une clause plus petite peut être considérée comme une clause plus grande, un nœud spécial étant le nœud $\bot$. Lorsqu'un littéral l apparaît dans une clause unitaire dans la preuve, il peut être ajouté à toute arête, car $F \wedge l$ est égal à $F \wedge l \wedge (l \vee x)$ pour tout x.

3.4 à la recherche de meilleures explications

Dans notre quête d'explication la plus simple possible des preuves d'insatisfiabilité, il n'est pas suffisant de trouver autant de contraintes de cardinalité que possible. Nous devons trouver les contraintes les plus fortes. Illustrons ce problème avec, encore une fois, le problème bien connu des pigeonniers.

Exemple 3.3 – Voici les contraintes que nous pouvons extraire en recherchant des cliques dans l'exemple 2.1.

$$
\begin{array}{ll}
x_1 + x_4 + x_7 + x_{10} \leq 1 & x_1 + x_2 + x_3 \geq 1 \\
x_2 + x_5 + x_8 + x_{11} \leq 1 & x_4 + x_5 + x_6 \geq 1 \\
x_3 + x_6 + x_9 + x_{12} \leq 1 & x_7 + x_8 + x_9 \geq 1 \\
 & x_{10} + x_{11} + x_{12} \geq 1
\end{array}
$$

Le problème est insatisfiable, mais l'explication reste floue. Ce que nous aimerions idéalement avoir, c'est : $(\sum_{i=1}^{12} x_i \leq 3)$ et $(\sum_{i=1}^{12} x_i \geq 4)$.

Obtenir de telles contraintes structurées comme expliqué dans l'exemple 3.3 est assez simple, il suffit de faire la somme des AtLeast-k d'un côté et la somme des AtMost-k de l'autre.

Nous obtiendrons des contraintes de la forme $(\sum_{i=1}^{n} x_i \leq k)$ and $(\sum_{i=1}^{n} x_i \geq k')$. Si nous obtenons $k = k'$, nous pouvons remplacer ces deux contraintes par $(\sum_{i=1}^{n} x_i = k)$ qui est plus fort. Sinon, si nous obtenons $k' > k$, nous pouvons alors supprimer toutes les autres contraintes et ne garder que ces deux contraintes, car elles suffisent à expliquer l'insatisfaisabilité. Sinon, si nous obtenons $k < k'$, alors tout est en ordre et nous

ne pouvons rien faire d'autre que de stocker les résultats finaux. De toutes ces extractions et simplifications, nous pouvons obtenir la définition suivante qui sera utilisée dans la partie expérimentale.

Définition 3.2 (Explication-Cardinalité)**.** Une preuve DRAT est dite *cardinalité-expliquée*, si on obtient l'ensemble de contraintes suivant : $(\sum_{x \in C} x \leq Y)$ et $(\sum_{x \in C} x \geq Y')$ avec $Y > Y'$.

Cependant, dans le cas général, il ne suffit de faire la somme de toutes les contraintes jusqu'à ce que l'on obtienne une cardinalité-explication. Pour contourner ce problème, nous avons proposé une recherche gloutonne à travers les différentes contraintes afin d'en obtenir de plus fortes. Malheureusement, étant une heuristique, certaines instances pourraient être cardinalité-expliquées, mais ne le seront pas en raison de l'ordre choisi pour additionner les contraintes.

Nous avons deux types de critères pour choisir les contraintes à additionner. Premièrement, la diversification, si deux contraintes n'ont pas de variables en commun, elles peuvent être ajoutées en toute sécurité. Le deuxième critère est l'intensification. Deux contraintes qui ont exactement les mêmes variables peuvent également être additionnées ainsi que leurs bornes. Puis la borne est divisée par deux et arrondie, afin d'éviter un coefficient devant les variables de la contrainte. Nous effectuons cette recherche à la fois dans AtMost-k et AtLeast-k afin d'obtenir une cardinalité-explication ou une contrainte Equals-k, afin de supprimer en toute sécurité deux contraintes. En effet, si on obtient $(\sum_i x_i \leq k)$ et $(\sum_i x_i \geq k)$, on peut supprimer les deux contraintes et ajouter la contrainte suivante : $(\sum_i x_i = k)$.

4 Étude expérimentale

Nous avons implémenté l'approche proposée dans un solveur open source (écrit en C ++) `EXPResSO`, signifiant *EXplainable PRoof SOlver*. La structure de données pour manipuler l'hypergraphe est basée sur la bibliothèque HTD [1].

Le but de cette section d'évaluation est double. (1) nous devons évaluer les performances de notre méthode et (2) vérifier combien de contraintes de cardinalité nous pouvons trouver et combien nous pouvons compresser les preuves d'insatisfaisabilité en utilisant des contraintes de cardinalité. Toutes les preuves étudiées sont des preuves DRAT données par `Glucose` [2] et certifiées par DRAT-trim [26]. Les contraintes de cardinalité trouvées dans la formule sont exprimées sous le format XCSP3 [7], qui est le format standard pour les compétitions de programmation par contraintes (CP).

`Nacre` [11] 1.0.4 garantie l'insatisfiabilité de l'instance XCSP3. Lorsque la preuve est réécrite en tant qu'instance XCSP3, `Nacre` trouve toujours la contradiction par un simple contrôle d'arc cohérence (pas de recherche). `Glucose` et `EXPResSO` fonctionnent sur un cluster d'ordinateurs identiques dotés de deux processeurs Intel (R) E5-2670, 2,60 GHz avec 64 Go de mémoire. Pour chaque problème, nous fixons un timeout de 900 secondes et une limite mémoire de 32 Go.

4.1 Contraintes de cardinalité dans les preuves unsatisfiable des formules aléatoires

Les premiers résultats que nous présentons sont probablement les plus surprenants. Nous considérons ici des formules aléatoires uniformes (au seuil) tirées de SATLIB. Les résultats montrent clairement un nombre impressionnant de contraintes de cardinalité trouvées dans les preuves UNSAT d'instances aléatoires. La figure 5 illustre le (surprenant) gain. On peut s'attendre à un gain exponentiel sur ces preuves (la diagonale signifie aucun gain).

La figure 4 montre les fréquences de chaque $\leq k$ (resp. $\geq k$). Comme on le voit, on est loin de trouver que des contraintes de cardinalité triviales. Les cliques deviennent même de plus en plus grandes avec la taille des instances. C'est un résultat surprenant qui pourrait probablement s'expliquer par le fait que les solveurs SAT ont tendance à générer des preuves de plus en plus grandes avec le même ensemble de variables, permettant ainsi de trouver plus de cliques. Cependant, nous pouvons aussi raisonnablement supposer que le nombre important de contraintes de cardinalité témoigne de la stratégie de recherche particulière qui émerge de la recherche même du solveur CDCL, même si cette hypothèse mérite de nouvelles investigations.

Concentrons-nous maintenant sur la figure 6. Rechercher toutes les cliques maximales et ensuite fusionner toutes les contraintes obtenues a malheureusement un coût élevé. `Glucose` est capable de produire une preuve pour toutes les instances en moins de 10 secondes, alors que `EXPResSO` peut prendre des centaines de secondes pour rechercher toutes les cliques maximales. La qualité des explications trouvées dans la preuve DRAT vaut toutefois la peine. Par exemple, une autre expérimentation a indiqué que plus de 27% de toutes les preuves UNSAT contiennent deux contraintes contradictoires de cardinalité. Bien sûr, encore une fois, ces contraintes peuvent être d'un grand intérêt pour comprendre la structure de l'instance originale ou même la forme de la recherche elle-même.

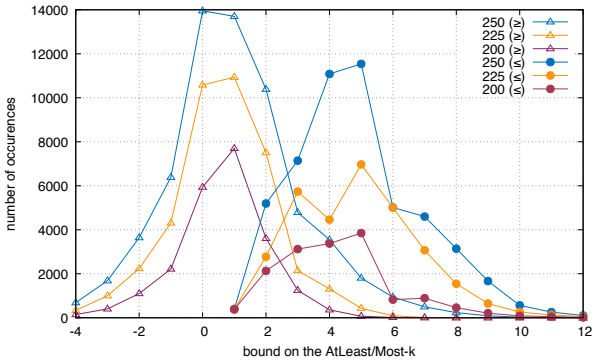


Figure 4 – Instances aléatoires : fréquences des contraintes de cardinalité pour $\leq k$ et $\geq k$

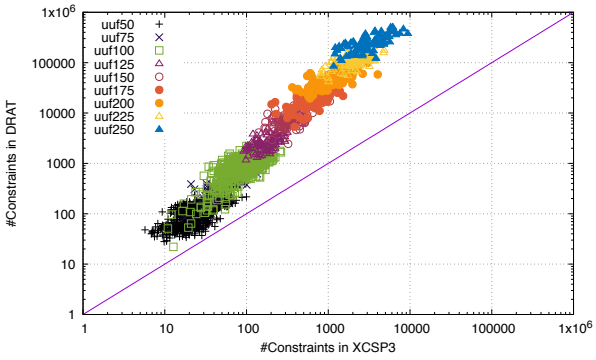


Figure 5 – Instances aléatoires : taille des preuves en CNF vs nb. des contraintes de cardinalité

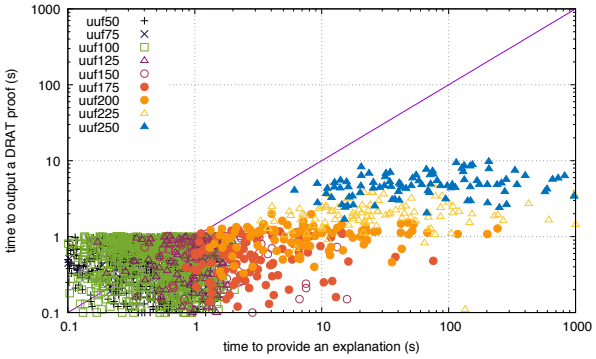


Figure 6 – Instances aléatoires : temps (s) pour `Glucose` et `EXPResSO`

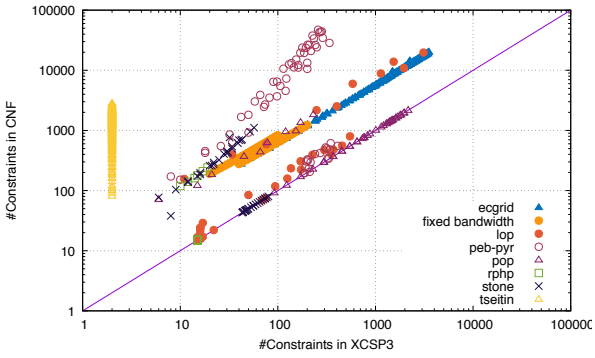


Figure 7 – Instances crafted : taille des problèmes d'entrée vs nb. de contraintes de cardinalité

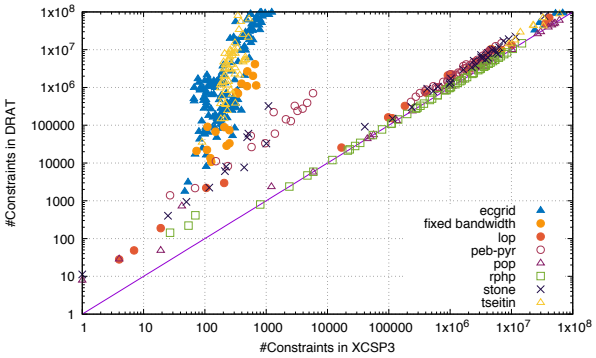


Figure 8 – Instances crafted : taille de la preuve (DRAT) vs nb. des contraintes de cardinalité

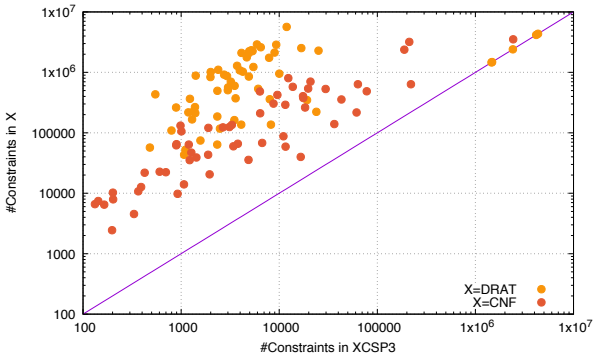


Figure 9 – Instances Industrielles : taille de la CNF (resp. DRAT) vs nb. des contraintes de cardinalité

Formula	Input #vars	Input #clauses	DRAT #size	XCSP3 #const
`hole20`	420	4 221	50 749	2
`hole30`	930	13 981	171 274	2
`hole40`	1640	32 841	405 899	–
`hole50`	2550	63 801	792 624	–
`tph8`	136	5 457	86 164	2
`tph12`	300	27 625	439 462	2
`tph16`	528	87 329	1 392 616	–
`tph20`	820	213 241	3 404 250	–
`Urquhart-b1`	106	714	179 330	1
`Urquhart-b2`	107	742	165 638	1
`Urquhart-b3`	121	1 116	299 910	1
`Urquhart-b4`	114	888	227 774	1

Table 1 – Comparaison des tailles entre la formule d'ori-

	`Glucose` min	`Glucose` med	`Glucose` max	`EXPResSO` min	`EXPResSO` med	`EXPResSO` max
Original	2448	66896	3520181	103	9598	3520181
Proof	33902	859211	4578043	511	10231	4578043

Table 2 – Instances Industrielles : analyse des tailles entre l'instance, la preuve et l'explication fournie

4.2 Contraintes de cardinalité dans les preuves de problèmes crafted

Les problèmes que nous considérons maintenant sont généralement générés pour donner du fil à retordre aux solveurs SAT basés sur la résolution. Nous avons utilisé dans cette section une sélection de problèmes décrits plus en détail dans [9, 19]. Commentons d'abord la Table 1, qui résume les résultats obtenus sur les preuves des problèmes de [19]. Les résultats, reportés dans la Table 1, sont vraiment frappants : pour les familles `hole*` et `tph*`, quand le délai imparti le permet, nous avons pu extraire une preuve de 2 contraintes. Notez que le post-traitement s'est fait instantanément, en résumant simplement toutes les contraintes afin d'obtenir une *explication-cardinalité* pour `hole*`, (nous obtenons $\sum_{i=1}^{n} x_i \leq 19$ et $\sum_{i=1}^{n} x_i \geq 20$. pour hole20) et `tph*` (contraintes similaires), une fois que nous avons éliminé par post-traitement les autres contraintes inutiles pour expliquer l'insatisfiabilité. Notez que nous ne nous sommes pas comparés avec la résolution étendue (ER) [25] ni DPR (une généralisation de DRAT) [15] car notre objectif ici n'est pas d'obtenir un système de preuve complet, mais simplement de traiter la preuve DRAT afin d'obtenir une explication plus courte de la réfutation. Maintenant, sur les instances proposées dans [9]. Les résultats sont représentés sur les figures 7 et 8. Ici encore, nous observons une grande compression sur les problèmes initiaux et aussi sur les preuves lorsque `EXPResSO` est capable de les gérer (les points sur la diagonale signifient que notre outil n'a pas été en mesure de réécrire les contraintes). Ce que nous pouvons voir, c'est que ces instances, qui sont difficiles à résoudre, génèrent des preuves qui peuvent en fait être réduites à très peu de contraintes de cardinalité.

4.3 Contraintes de cardinalité dans les preuves des problèmes industriels

Dans cette dernière série d'instances [10], nous nous concentrons sur les instances "industrielles" typiques. Ces instances ont été sélectionnées de telle sorte que leur preuve soient de taille raisonnable. Une fois de plus, nous étudions comment notre outil peut obtenir des contraintes de cardinalité dans les formules originales et dans les preuves. Les résultats sont résumés dans la Table 2 ainsi que sur la figure 9. Trois instances ont produit une preuve trop grande pour notre outil : leur nombre de contraintes de cardinalité correspond donc exactement au nombre de clauses (voir le trois points sur la diagonale de la figure). Cette dernière expérimentation conclut cette section en montrant à quel point notre méthode est générique : les contraintes de cardinalité sont légion dans toutes les preuves expérimentées, quelles que soient les origines des problèmes.

5 Conclusion

Dans cet article, nous avons montré que les preuves DRAT peuvent être restructurées et exprimées sous la forme de contraintes de cardinalité, même dans le cas d'instances générées aléatoirement. Pour ce faire, nous encodons la preuve sous la forme d'un hypergraphe où chaque lemme est une hyperarête et chaque littéral est un sommet. Ensuite nous cherchons toutes les cliques maximales, codant les contraintes de cardinalité. Nous avons généralisé l'algorithme de Bron & Kerbosch, pour énumérer les cliques maximales d'un hypergraphe et démontré son efficacité et sa *scalabilité* avec des centaines de sommets et des millions d'hyperarêtes.

Dans la recherche de preuves simples et efficaces, la communauté tend à proposer des systèmes de preuves strictement plus forts avec de nouvelles stratégies de recherche. Cependant, cela a un coût important : les procédures de recherche sont soit conçues pour des problèmes particuliers, soit moins génériques que les solveurs CDCL actuels. Ici, nous adoptons le point de vue opposé : nous gardons les performances des solveurs SAT et proposons de rechercher dans la trace des régularités exploitables. Ce travail est le premier à présenter une structure particulière et compréhensible des preuves de solveur SAT.

Cependant, notre approche a aussi des inconvénients : pour que la preuve soit expliquée, il faut d'abord la trouver, ce qui peut avoir un coût exponentiel (nous ne pouvons pas aller au-delà des limites de la résolution). La prochaine étape consisterait à piloter la recherche du solveur SAT grâce à un solveur de théorie cherchant des cliques maximales dans la preuve courante. Si une partie d'une explication de cardinalité est trouvée, il pourrait être intéressant de forcer la recherche vers le cas "opposé", ce qui permettrait de produire une preuve beaucoup plus courte ou de diviser la recherche en deux. Nous conjecturons également la présence de contraintes de cardinalité importantes dans la preuve d'insatisfaisabilité comme l'un des principaux témoins de la recherche du solveur SAT. Une grande hyperclique centrale pourrait expliquer la grande non-dégénérescence des graphes induits par la preuve rapportée dans [10]. Une autre étape consisterait à fournir un format certifié pouvant être indépendant vérifié avec un `checker`.

Remerciements

Une partie de ces travaux ont étés soutenues par le Ministère de l'Enseignement Supérieur et de la Recherche et partiellement financés par l'IDEX UCA[JEDI].

Références

[1] Michael Abseher, Nysret Musliu et Stefan Woltran : htd - A free, open-source framework for (customized) tree decompositions and beyond. *In Proc. of CPAIOR*, pages 376–386, 2017.

[2] Gilles Audemard et Laurent Simon : Predicting Learnt Clauses Quality in Modern SAT Solvers. *In Proc. of IJCAI*, pages 399–404, 2009.

[3] Claude Berge : *Hypergraphs : Combinatorics of Finite Sets*, volume 45 de *Mathematical Library*. Elsevier, 05 1984.

[4] Armin Biere, Daniel Le Berre, Emmanuel Lonca et Norbert Manthey : Detecting Cardinality Constraints in CNF. *In Proc. of SAT*, pages 285–301, 2014.

[5] Armin Biere, Marijn Heule, Hans van Maaren et Toby Walsh, éditeurs. *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[6] Burton H. Bloom : Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[7] Frédéric Boussemart, Christophe Lecoutre et Cédric Piette : XCSP3 : An Integrated Format for Benchmarking Combinatorial Constrained Problems. *CoRR*, abs/1611.03398, 2016.

[8] Coen Bron et Joep Kerbosch : Algorithm 457 : Finding All Cliques of an Undirected Graph. *Commun. ACM*, 16(9):575–577, 1973.

[9] Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström et Laurent Simon : Seeking practical CDCL insights from theoretical SAT benchmarks. *In Proc. of IJCAI*, pages 1300–1308, 2018.

[10] Rohan Fossé et Laurent Simon : On the Non-degeneracy of Unsatisfiability Proof Graphs Produced by SAT Solvers. *In Proc. of CP*, pages 128–143, 2018.

[11] Gael Glorian : Nacre. *In* Christophe Lecoutre et Olivier Roussel, éditeurs : *Proc. of 2018 XCSP3 Competition*, pages 85–85, 2019.

[12] Éric Grégoire, Richard Ostrowski, Bertrand Mazure et Lakhdar Sais : Automatic Extraction of Functional Dependencies. *In Proc. of SAT*, 2004.

[13] Marijn Heule, Warren A. Hunt Jr. et Nathan Wetzler : Trimming while checking clausal proofs. *In Proc. of FMCAD*, pages 181–188, 2013.

[14] Marijn J. H. Heule : Schur Number Five. *In Proc. of AAAI*, pages 6598–6606, 2018.

[15] Marijn J. H. Heule, Benjamin Kiesl et Armin Biere : Short Proofs Without New Variables. *In Proc. of CADE*, pages 130–147, 2017.

[16] Marijn J. H. Heule, Oliver Kullmann et Victor W. Marek : Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer. *In Proc. of SAT*, pages 228–245, 2016.

[17] Matti Järvisalo, Marijn Heule et Armin Biere : Inprocessing Rules. *In Proc. of IJCAR*, pages 355–370, 2012.

[18] Matti Järvisalo, Arie Matsliah, Jakob Nordström et Stanislav Zivny : Relating Proof Complexity Measures and Practical Hardness of SAT. *In Proc. of CP*, pages 316–331, 2012.

[19] Benjamin Kiesl, Adrián Rebola-Pardo et Marijn J. H. Heule : Extended Resolution Simulates DRAT. *In Proc. of IJCAR*, pages 516–531, 2018.

[20] Andreas Koeller : *Integration of Heterogeneous Databases : Discovery of Meta-Information and Maintenance of Schema-Restructuring Views*. Thèse de doctorat, Worcester Polytechnic Institute, dec 2001.

[21] J. W. Moon et L. Moser : On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

[22] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang et Sharad Malik : Chaff : Engineering an Efficient SAT Solver. *In Proc. of DAC*, pages 530–535, 2001.

[23] Richard Ostrowski, Éric Grégoire, Bertrand Mazure et Lakhdar Sais : Recovering and exploiting structural knowledge from CNF formulas. *In Proc. of CP*, pages 185–199, 2002.

[24] Laurent Simon : Post Mortem Analysis of SAT Solver Proofs. *In Proc. of POS*, pages 26–40, 2014.

[25] G. S Tseitin : *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer, 1983.

[26] Nathan Wetzler, Marijn Heule et Warren A. Hunt Jr. : DRAT-trim : Efficient Checking and Trimming Using Expressive Clausal Proofs. *In Proc. of SAT*, pages 422–429, 2014.