Réparation locale de réfutation de formules SAT

Nicolas Prcovic

Réparation locale de réfutation de formules SAT

Nicolas Prcovic
LSIS - Aix-Marseille Universités
nicolas.prcovic@lsis.org

Résumé

Nous traitons du problème de la démonstration qu'une formule booléenne est insatisfiable. Nous présentons un schéma de réparation locale de preuve d'insatisfiabilité par résolution. L'idée principale est de considérer les preuves linéaires dont le nombre de résolutions autorisées est bornée par une constante comme étant équivalentes à des problèmes de satisfaction de contraintes (CSP) dont chaque solution est une réfutation de la formule booléenne. Dès lors, toute technique de réparation locale applicable aux CSP peut être réutilisée a priori, une fois définis le voisinage d'une "réfutation à réparer" et sa proximité par rapport à une véritable réfutation. Nous proposons des critères pour évaluer les réfutations à réparer ainsi que des opérateurs de voisinage qui réaffectent une variable du CSP (changement de clause) ou permutent des valeurs de variables (déplacement de clause dans la preuve).

Abstract

We address the problem of proving that a boolean formula is unsatisfiable. We present a local repair scheme which proves unsatisfiability by resolution. The main idea is to consider the linear proofs which number of allowed resolutions is bounded by a constant as being equivalent to constraint satisfaction problems (CSP), which solutions are refutations of the boolean formula. Therefore, any local repair technique applicable to CSP can be reused, once defined the neighborhood of a "refutation to be repaired" and its proximity to a true refutation. We propose criteria to evaluate the proofs to be repaired and define neighborhood operators that re-assign a variable of CSP (change of a clause) or swap the values of variables (move of a clause inside the proof).

1 Introduction

Parmi les 10 défis posés à la communauté par Selman et al [8] en 1997, le cinquième, consistant à trouver un algorithme de recherche locale efficace pour démontrer l'insatisfiabilité d'une formule booléenne (ie, sa réfutation), est le seul à n'avoir pas donné lieu à des résultats satisfaisants, malgré quelques tentatives intéressantes [6, 1]. Dans cet article, nous partons du résultat de [3], où il est démontré que le problème de trouver la plus petite réfutation par résolution d'une formule booléenne est NP-difficile *quand la longueur de la preuve est une fonction polynomiale* du nombre de variables. Nous proposons une nouvelle approche consistant à transformer la question sous la forme d'un problème de satisfaction de contraintes (CSP) dont chaque solution est une réfutation de la formule booléenne qu'il représente. L'avantage de cette transformation est que l'on peut immédiatement appliquer n'importe quelle méthode de recherche de solution d'un CSP à notre problème, notamment une recherche locale.

Informellement, il s'agit de construire une preuve *linéaire* par résolution dont la longueur est bornée par une constante k. Une preuve est linéaire quand toute nouvelle résolvante est inférée à partir de la résolvante précédemment inférée et d'une autre clause [5]. Dans ce contexte, construire une réfutation, c'est choisir k clauses successives de telle manière que la dernière résolvante soit la clause vide. Il s'agit donc bien de reformuler un problème de co-NP en un problème NP-complet. Ceci est possible (et n'implique pas que co-NP = NP) car nous restreignons la question initiale à celle d'une *preuve par résolution qui soit polynomialement bornée* par le nombre de variables de la formule. Cette restriction n'est pas gênante dans la mesure où, d'une part, les réfutations sont en général basées sur la résolution, et d'autre part, il faut que la taille de la preuve soit présuposée polynomialement bornée pour qu'un algorithme de recherche de cette preuve puisse s'exécuter en un temps raisonnable.

Dans un premier temps, nous ferons des rappels sur les formules booléennes et les méthodes pour déterminer leur insatisfiabilité. Puis, nous formaliserons comment, à partir d'une formule booléenne, nous pouvons

définir le CSP dont chaque solution est une réfutation linéaire de cette formule. Ensuite, nous définirons ce qu'est le voisinage d'une dérivation et comment évaluer sa proximité à une réfutation. Enfin, nous comparerons notre approche à l'existant.

2 Notions préliminaires

Une formule booléenne, sous sa forme dite CNF, est un ensemble de clauses. Une clause est un ensemble de littéraux. Un littéral est soit une variable booléenne, soit sa négation. On définit Var(l) comme étant la variable du littéral l. Les variables peuvent être affectées à la valeur vrai ou faux. Une clause représente une disjonction de l'ensemble de ses littéraux. Une formule SAT représente une conjonction de ses clauses. Un modèle d'une formule ϕ est une affectation de variables qui est tel que ϕ est vraie. Une formule n'ayant pas de modèle est dite insatisfiable. La clause vide, notée $\square$, est toujours fausse et si une formule la contient alors elle est insatisfiable. Si une clause contient un littéral l et son opposé $\neg l$, elle est toujours vraie et on l'appelle *tautologie* sur Var(l).

Les systèmes formels de preuve propositionnelle permettent de dériver d'autres formules à partir d'une formule ϕ, grâce à des règles d'inférence. En particulier, le système de Robinson [7] permet de dériver des clauses induites à partir des clauses d'une formule ϕ grâce à la règle de *résolution* :

$$C_1, C_2 \vdash C_1 \backslash \{l\} \cup C_2 \backslash \{\bar{l}\}$$

On notera $C_1 \otimes C_2$ la clause inférée par résolution de C_1 avec C_2, qu'on appellera *résolvante* sur Var(l) de C_1 et C_2. L'intérêt d'un système basé sur la résolution est qu'il permet d'établir des réfutations de formules (ie, des preuves de leur insatisfiabilité) dans la mesure où une formule est insatisfiable si et seulement si il existe une suite de résolutions qui dérive la clause vide.

Une dérivation peut se représenter grâce à un arbre binaire dont les feuilles sont des clauses de ϕ, les nœuds internes sont des résolvantes et la racine est la clause dérivée (cf figure 1).

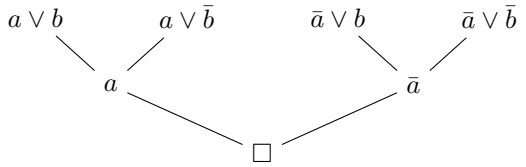

FIG. 1: Un arbre de resolution

Parmi les techniques de dérivation, certaines restreignent les possibilités d'appliquer la résolution tout en maintenant leur complétude. En particulier, les techniques de dérivation *linéaire* [5] obligent toute nouvelle résolvante à être inférée à partir de la précédente. Formellement, une dérivation linéaire d'une formule ϕ est une séquence de clauses (R_1, ..., R_n) telle que $R_1 \in \phi$ et chacun des autres R_i est la résolvante entre R_{i-1} (son parent proche) et une clause de ϕ (son parent d'entrée) ou une résolvante R_j ($j < i$) précédemment inférée (son ancêtre éloigné). R_1 est la clause de base. R_n est la clause dérivée. Si $R_n = \square$, nous avons une réfutation linéaire (cf figure 2). La SL-résolution[4] est un exemple d'algorithme permettant de rechercher systématiquement la réfutation linéaire d'une formule. Le principe de cet algorithme est d'effectuer une recherche arborescente en profondeur d'abord qui à chaque pas choisit une nouvelle clause à ajouter à la fin de la dérivation courante. La nouvelle clause doit permettre de produire une nouvelle résolvante non tautologique. L'algorithme s'arrête quand la clause vide est produite ou qu'il a essayé suffisamment de possibilités de construire une réfutation linéaire pour établir que la formule est non réfutable (et donc satisfiable).

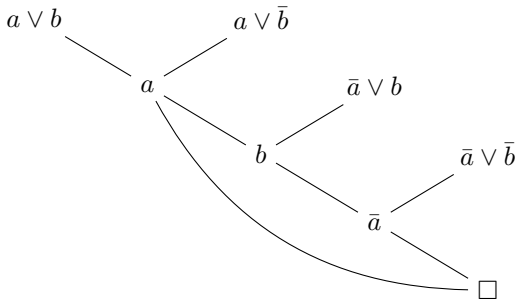

FIG. 2: Une résolution linéaire

Un problème de satisfaction de contraintes (CSP) est un triplet (X, D, C) où X=$\{x_1, ..., x_k\}$ est un ensemble de variables, D=$\{D_1, ..., D_k\}$ est l'ensemble des domaines dans lesquels ces variables peuvent prendre respectivement leur valeur, et C est un ensemble de contraintes qui expriment la compatibilité des affectations des variables. Une solution du CSP est une affectation de toutes les variables à une valeur de leur domaine telle que toutes les contraintes de C sont respectées.

3 Réfutation d'une formule SAT comme solution d'un CSP

Nous allons reformuler notre problème de réfutation d'une formule SAT sous la forme d'un CSP, uniquement dans le but de montrer concrètement comment

il peut se ramener à un problème NP-complet. Nous partons du fait que pour construire une dérivation linéaire de longueur k d'une formule $\phi = \{C_1, ..., C_m\}$, il faut choisir k clauses qui vont permettre d'effectuer la dérivation $\{C_{m+1}, ..., C_{m+k}\}$. Si on appelle x_i le choix du numéro de la i^e clause permettant d'inférer C_{m+i}, on a $x_i \in D_i = \{1, ..., m+i-1\}$. L'affectation de tous les x_i représente une réfutation linéaire si la contrainte $C_{x_1} \otimes C_{x_2} \otimes ... C_{x_k} = \square$ est respectée, avec $C_{m+1} = C_{x_1}$ et $\forall i > m+1$, $C_i = C_{i-1} \otimes C_{x_{i-m}}$. Ceci constitue un CSP dont la seule contrainte est globale et se vérifie en temps polynomial.

Notre objectif étant de définir une méthode de réparation de réfutation, nous pouvons en théorie imaginer commencer par affecter les variables x_i de façon à obtenir une dérivation d'une clause qui ne soit pas vide puis à modifier les affectations pour obtenir pas à pas une réfutation. Cependant, la difficulté est que nous ne pouvons pas choisir indépendamment les valeurs des variables car il faut qu'à chaque étape de la dérivation on puisse appliquer une résolution, qui nécessite qu'un littéral apparaisse positivement dans une clause et négativement dans l'autre. Or, la liberté de changer n'importe quelle affectation est un critère important pour concevoir un algorithme de recherche locale. C'est pourquoi nous choisissons de généraliser la règle d'inférence de telle manière qu'elle puisse s'appliquer à *tout* couple de clause. Pour ceci, nous intégrons une règle d'inférence qu'on retrouve classiquement dans les sytèmes logiques formels, la *règle d'affaiblissement* :

$$C_1, C_2 \vdash C_1 \cup C_2$$

Cette règle n'est pas intégrée dans les méthodes de réfutation basée sur la résolution car $C_1 \cup C_2$ est subsumée par C_1 et par C_2 et ne doit donc pas être utilisée à la place de C_1 ou C_2 pour dériver la clause vide. En particulier, si on peut effectuer une résolution entre C_1 et C_2 sur l, effectuer un affaiblissement de C_1 et C_2 produit une clause égale à la résolvante de C_1 et C_2 augmentée de $\{l, \bar{l}\}$. Nous introduisons cependant les affaiblissements car ils sont logiquement corrects et que nous pourrons les supprimer ultérieurement par réparation. Ceci nous conduit donc à remplacer la résolution par une règle d'inférence qui la généralise en prenant en compte la possibilité d'affaiblissement :

$$C_1, C_2 \vdash C_1 \backslash \{l\} \cup C_2 \backslash \{\bar{l}\} \text{ si } l \in C_1 \text{ et } \bar{l} \in C_2 \text{ et } C_1 \cup C_2 \text{ sinon}$$

En définissant l'opérateur binaire $\odot$ associatif à gauche qui est tel que $C = C_1 \odot C_2$ ssi $C_1, C_2 \vdash C$, la contrainte globale devient $C_{x_1} \odot C_{x_2} \odot ... C_{x_k} = \square$. Nous obtenons ainsi la propriété très utile que toute affectation de l'ensemble des variables x_i permet d'obtenir la dérivation (logiquement correcte) d'une clause.

Remarquons que, contrairement aux méthodes habituelles de réfutations basées sur la résolution, nous ne rejetons pas les tautologies inférées. Dans le cadre habituel, une tautologie $\{l, \bar{l}\} \cup C$ peut être écartée car une résolution sur l avec une clause $\{l\} \cup C'$ donne la résolvante $\{l\} \cup C \cup C'$ qui est subsumée par $\{l\} \cup C'$. Nous conservons les tautologies car elles sont logiquement correctes et que nous pourrons les supprimer ultérieurement par réparation.

A partir de maintenant, nous appelerons *clause inférée* le résultat d'une *inférence* qui consiste soit en une résolution, qui produit une résolvante, soit un affaiblissement, qui produit une *clause affaiblie*.

Réduction de la combinatoire

Nous avons un moyen de réduire la combinatoire du problème. Tel qu'il est défini jusqu'à présent, il s'agit de choisir k fois parmi au moins m et au plus $m+k$ valeurs. Or, k étant un polynôme $P(m)$, la combinatoire est en $\Theta(P(m)^{P(m)})$. Nous avons un moyen de réduire fortement cette complexité grâce à une restriction supplémentaire utilisée notamment dans la SL-résolution : les résolutions effectuées avec des ancêtres éloignés sont limitées à celles qui produisent une clause subsumant le parent proche (ie, tous les littéraux de l'ancêtre éloigné sont dans le parent proche sauf un littéral dont l'opposé est présent dans le parent proche, littéral qui permet la résolution). De plus, quand une résolution avec un ancêtre éloigné est possible, il faut l'effectuer immédiatement. Cette restriction conserve la complétude de la résolution linéaire et supprime totalement les choix relatifs aux résolutions avec ancêtres éloignés. La preuve de la complètude est donnée dans [4]. Nous pouvons donc modifier notre CSP ainsi : les variables x_i représentent les choix des *clauses d'entrée* effectués au cours de la dérivation. Entre deux choix, on applique itérativement toutes les résolutions subsumantes possibles avec ancêtre éloigné. Dans [4], il est donné une méthode (les chaînes de littéraux) pour le faire très efficacement. Remarquons qu'en conséquence, un CSP à k variables représente une dérivation de longueur supérieure ou égale à k : k inférences avec clauses d'entrée intercalées avec un certain nombre de résolutions avec un ancêtre éloigné. La combinatoire se réduit maintenant à $m^{P(m)}$.

Il est bien clair que la contrainte qui caractérise maintenant notre CSP est complexe et très particulière. Il n'y a donc pas véritablement d'intérêt pratique à reformuler notre problème sous la forme d'un CSP puis à le donner tel quel à un solveur de CSP. Cependant, par la suite, nous conserverons l'appellation CSP quand il est pratique pour la compréhension de notre méthode de considérer notre problème sous cette forme.

4 Schéma de réparation locale de réfutation

Maintenant que nous avons défini notre CSP, nous pouvons imaginer ré-utiliser directement n'importe quel algorithme de recherche locale connu dédié à la recherche rapide de solutions de CSP. Ce qui justifie l'utilisation d'une méthode de réparation de réfutation linéaire (que nous abrégerons dorénavant par RRL) est le présupposé qu'une modification locale a un impact global limité. Or, si nous considérons les arbres de résolution, nous constatons qu'ils induisent un ordre partiel sur les résolutions à effectuer. En l'occurrence, dans une réfutation linéaire qui se termine par la résolution entre l et $\bar{l}$, toutes les résolutions dérivant l peuvent être effectuées avant ou après celles dérivant $\bar{l}$. La dérivation de l n'influe pas sur celle de $\bar{l}$ et la réparation de l'une n'impacte l'autre que partiellement. Une modification locale de dérivation peut donc ne modifier que partiellement la clause dérivée.

Voici le schéma très général qu'ont les algorithmes de réparation si on les applique à notre contexte. Une fois générée une dérivation initiale D, on cherche itérativement un voisin D' de D qui améliore la qualité de D et on remplace D par D'. On s'arrête si la clause vide est dans D ou après un certain nombre d'itérations. Si D n'a pas de voisin qui l'améliore, on utilise une technique d'échappement de ce miminum local. Nous avons donc en premier lieu à définir ce qui fait la qualité d'une dérivation (ie, sa proximité à une réfutation) et ce qu'est le voisinage d'une dérivation.

4.1 Evaluation de la qualité d'une dérivation

Voici trois classes de critères permettant d'évaluer la proximité d'une dérivation par rapport à une réfutation :

- les critères liés à la taille des clauses produites :
  - La taille de la plus petite clause produite doit être minimisée. Nous devons considérer toutes les clauses inférées et non la dernière car il est possible que la clause vide apparaisse en cours de dérivation. Si deux dérivations ont leur clause la plus courte de taille égale, on favorisera la dérivation qui contient le plus de clauses de cette taille.
  - La largeur de la dérivation doit être minimisée. La largeur d'une dérivation est égale à la taille de sa plus grande clause. Dans [2], il est démontré que les réfutations courtes ont nécessairement une largeur faible. Si deux dérivations ont même largeur, on favorisera la dérivation qui contient le moins de clauses de cette largeur.
  - la taille moyenne des clauses doit être la plus faible possible. Ce critère prend en compte la taille des clauses qui ne sont pas extrêmes.
- les critères liés à la nature des clauses inférées :
  - le nombre de tautologies doit être minimisé car toute inférence entre une tautologie et une clause C produit une clause subsumée par C.
  - le nombre de clauses affaiblies doit être minimisé car elles sont subsumées par leur clause parente.
  - le nombre de clauses produites subsumées par des clauses d'entrée ou des clauses précédemment inférées doit être minimisées. Ce critère généralise le précédent car une clause peut être subsumée sans être le résultat d'un affaiblissement. Cependant le surcoût de vérification de subsumption doit être mis en balance avec les gains d'efficacité espérés grâce à une meilleure évaluation de la dérivation, ceci afin de déterminer s'il faut l'inclure en pratique.
- les critères liés à l'occurrence des littéraux :
  - Le nombre de littéraux "purs" (relativement aux clauses utilisées dans la dérivation) doit être minimisé. En effet, si un littéral apparaît dans une dérivation mais jamais son opposé, il sera toujours présent dans toutes les clauses inférées après son introduction.
  - De façon plus générale, parmi l'ensemble des clauses qui sont introduites *après* une clause inférée, il faut qu'il apparaisse au moins une fois l'opposé de chacun des littéraux de la clause inférée.

Parmi ces critères, les plus importants sont celui de la taille de la plus petite clause et celui du nombre de littéraux purs. En effet, une réfutation contient nécessairement la clause vide (de taille 0) et un nombre nul de littéraux purs. Par contre, elle peut éventuellement contenir des clauses longues, des affaiblissements et des tautologies. Notre fonction d'évaluation ordonne donc les dérivations en fonction des critères suivants, dans cet ordre (en cas d'égalité de deux dérivations sur un critère, on examine le suivant) : minimisation de la plus petite clause, maximisation du nombre de plus petites clauses, minimisation du nombre de littéraux purs, minimisation de la largeur de la dérivation, maximisation du nombre de clauses de cette largeur, minimisation du nombre de tautologies, minimisation du nombre d'affaiblissements. Cet ordre est partiellement arbitraire et d'autres combinaisons mériteraient d'être étudiées.

4.2 Voisinage d'une dérivation

Habituellement, on définit le voisin d'une affectation des variables comme étant une affectation qui diffère

d'une seule valeur. L'avantage est qu'il existe une séquence de voisins qui permet de relier toute affectation à toute autre, ce qui garantit à l'algorithme de recherche locale de pouvoir explorer tout l'espace de recherche. Cependant, dans notre cas, nous pouvons aussi considérer en plus un autre type de modification qui consiste à déplacer une clause (ou une suite de clauses) à un autre endroit de la dérivation, ce qui consiste à permuter les valeurs de variables du CSP. La mise en oeuvre de cette seconde possibilité se justifie dans la mesure où déplacer une clause peut moins perturber la dérivation que la remplacer par une autre clause.

Dit autrement, une dérivation résulte donc d'un *arrangement* de clauses prises parmi les clauses d'entrée du problème et nous pouvons nous intéresser d'abord à trouver une *combinaison* de clauses d'entrées puis à les ordonner.

Si k est la taille de la dérivation et m est le nombre de clauses de la formule, le nombre de façons de changer une clause de la dérivation est $k.m$ et le nombre de façons de déplacer une clause dans la dérivation est de l'ordre de k^2 (chacune des k clauses peut s'insérer dans un des $k-1$ autres emplacements).

4.2.1 Remplacement de clause

Nous nous intéressons dans un premier temps aux conditions de présence d'une clause dans la dérivation sans pour l'instant nous soucier de sa place. La condition la plus évidente est que, pour chacun de ses littéraux, il doit exister une autre clause de la dérivation qui contienne son littéral opposé (si on veut obtenir une réfutation). Une heuristique de choix de clause à retirer consiste donc à trouver une clause dont un littéral est "pur" en évitant autant que possible que la disparition des autres littéraux fasse que d'autres littéraux de la combinaison de clauses courante ne deviennent purs. De façon complémentaire, une heuristique d'introduction d'une nouvelle clause doit si possible introduire des littéraux opposés aux littéraux actuellement purs de la combinaison de clauses courante et éviter d'introduire des littéraux purs.

Il y a très certainement d'autres heuristiques de remplacement de clauses à découvrir.

4.2.2 Déplacement de clause

Voici les effets que peut avoir le déplacement d'une clause :

- Supprimer un affaiblissement : reculer une clause affaiblissante C à un endroit de la dérivation postérieur à l'introduction d'une clause dont un littéral est l'opposé d'un de ceux de C.
- Supprimer une tautologie : si l_2 et $\bar{l_2}$ sont dans une résolvante sur l_1, faire remonter la clause d'entrée avant l'introduction du littéral sur l_2 et après l'introduction du littéral sur l_1.
- Diminuer la taille d'une suite de clauses inférées : si aucun des nouveaux littéraux introduits par une clause d'entrée à l'étape i n'est effacé (par résolution) avant l'étape j ($j > i$), déplacer la clause d'entrée à l'étape $j-1$ permet de supprimer la présence des nouveaux littéraux entre les étapes i et j.

Il faut faire cependant attention que la disparition d'un affaiblissement ou d'une tautologie ou la diminution de la taille d'une suite de clauses peut provoquer l'apparition d'un autre affaiblissement ou d'une autre tautologie. Voici des cas de modifications qui apportent un gain clair grâce à un déplacement d'une clause. On appelle C_i la clause d'entrée introduite dans la dérivation D à l'étape i et R_{i+1} la clause inférée à partir de C_i et R_i.

- Cas où C_i est affaiblissante (cf figure 3). Soit C_j la première clause d'entrée introduite après C_i où un des littéraux $\bar{l_1}$ de C_j est opposé à un littéral l_1 de C_i. C_j produit une résolvante car $l_1 \in R_j$. Si R_{j+1} est une tautologie sur l_1 (car la résolution a été faite sur une autre variable) alors déplacer C_i à l'étape j (donc juste après C_j) fait disparaître la tautologie car l_1 n'est plus dans R_j. Il est nécessaire que R_{j+1} soit une tautologie sinon le déplacement de C_i derrière C_j fait que R_{j+1} devient un affaiblissement et nous n'aurions alors fait que déplacer l'endroit où se trouve l'affaiblissement.

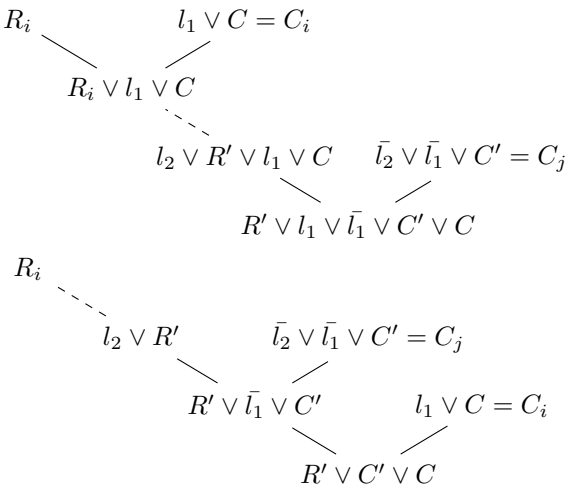

FIG. 3: Déplacement provoquant la suppression d'un affaiblissement et d'une tautologie : C_i passe juste après C_j.

– Cas où C_i permet une résolution sur l_1, $l_1 \in C_i$ (cf figure 4). Soit C_j la première clause d'entrée introduite après C_i où un des littéraux $\bar{l_2}$ de C_j est opposé à un littéral l_2 de C_i. Si R_{j+1} ne contient pas le littéral l_1, alors placer C_i à l'étape $j-1$ (juste avant C_j) permet de supprimer $C_i \backslash \{l_1\}$ de tous les R_h, $i < h < j$, c'est-à-dire de diminuer la taille de ces clauses inférées. Si R_{j+1} contient le littéral l_1, alors il faut remonter avant l'étape j jusqu'à ce que la clause inférée ne contienne plus l_1. En effet, si elle contenait l_1 alors l'absence de C_i avant elle ferait qu'elle deviendrait une tautologie sur l_1.

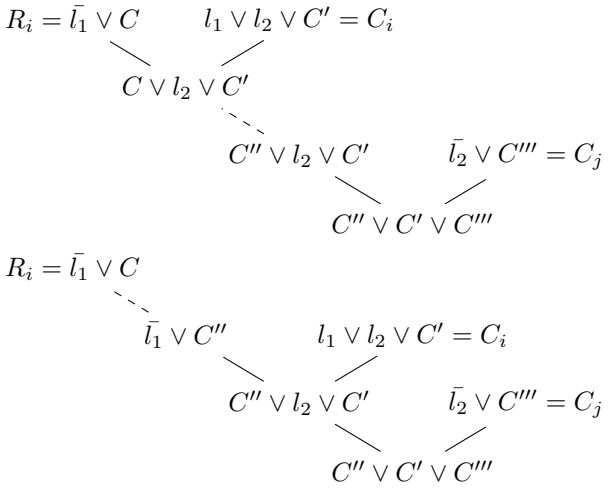

FIG. 4: Déplacement provoquant le raccourcissement d'une séquence de clauses inférées : C_i passe juste avant C_j.

Ce sont les deux cas de déplacement à effet purement positif que nous avons découvert pour l'instant mais nous pensons qu'il en reste à découvrir. Notamment, il faudrait aussi examiner les possibilités de déplacer des blocs de clauses consécutives.

5 Algorithme de réparation de réfutation linéaire

Nous avons d'abord défini l'algorithme le plus simple possible (cf figure 5). Nous partons d'une dérivation initiale D (ligne 1) et nous cherchons une meilleure dérivation `max-iterations` fois ou tant que nous n'avons pas dérivé la clause vide. Nous remplaçons une clause par une autre (ligne 4 et 5) et nous cherchons un meilleur ordre possible de clauses présentes dans la dérivation. Si cette dérivation est mieux évaluée, elle remplace la précédente (ligne 7). Une fois sorti de la boucle, nous retournons la dérivation produite.

Nous allons détailler les étapes de la méthode en précisant les options que nous avons écartées et celles que nous avons finalement retenues pour l'instant.

```
RRL(F, max-iterations)
(1) D = derivation-initiale(F)
(2) tantque clause vide not in D
    et nb-iterations < max-iterations
(3)     e = eval(D)
(4)     i = numero-clause-a-retirer(D)
(5)     c = clause-a-ajouter(F)
(6)     D' = reordonne(D \ {c_i} U {c})
(7)     si eval(D') > e alors D = D'
(8)     incrementer nb-iterations
(9) retourne D
```

FIG. 5: Algorithme de réparation de réfutation linéaire

La taille de la dérivation

Après quelques expérimentations, nous avons fixé la taille k de la dérivation à la valeur $2.m$. Cela signifie que nous nous attendons à ce que chaque clause puisse apparaître en moyenne deux fois par réfutation. Si nous voulons faire un algorithme qui ne dépend pas de k, il faudrait simplement le relancer en augmentant la valeur de k à chaque fois. Comme k doit rester un polynôme de m, il faudrait ajouter un multiple ou un diviseur de m à k à chaque relance.

Dérivation initiale

Nous pouvons affecter aléatoirement les clauses d'entrée de la dérivation initiale. Mais en pratique, nous avons constaté qu'une telle dérivation contenait essentiellement des affaiblissements qui étaient aussi souvent des tautologies. Nous avons donc aussi essayé de générer une dérivation initiale de la meilleure qualité possible grâce à un algorithme peu coûteux. Nous avons choisi d'utiliser un algorithme glouton qui construit itérativement une dérivation, à partir d'une dérivation vide, en rajoutant à chaque fois une clause d'entrée à la fin de la dérivation précédente, comme le ferait la SL-résolution mais sans backtracker. Concrètement, à l'itération i, nous inférons une nouvelle clause R_{i+1} à partir de la dernière clause inférée R_i et d'une clause d'entrée C_i que nous choisissons ainsi : il faut que R_{i+1} soit une résolvante non tautologique la plus courte possible et qu'elle ne soit pas subsumée par une résolvante précédente. Si toute résolvante non subsumée est tautologique, on accepte la plus courte. Si aucune résolution n'est possible, on choisit l'affaiblissement le plus court. En pratique, nous avons

constaté que les dérivations produites ainsi ne contenaient presque que des résolutions non tautologiques.

Remplacement des clauses

Pour l'instant, pour des raisons de simplicité, la clause à remplacer est choisie au hasard ainsi que la clause la remplaçant. Ceci garantit une bonne diversification dans le choix des clauses mais ne permet pas de trouver rapidement les bonnes clauses[1].

Déplacement des clauses

Le réordonnancement des clauses se fait en testant toutes les façons $k.(k-1)$ de déplacer une clause. Dès que le déplacement d'une clause produit une meilleure évaluation de la dérivation, le déplacement est effectué et on recommence. Lorsqu'il n'y a plus moyen de déplacer une clause en améliorant l'évaluation de la dérivation, on a terminé le réordonnancement. Cette manière de procéder permet d'améliorer la qualité des dérivations mais elle s'avère très coûteuse par rapport à ce qu'elle apporte. Nous avons donc restreint les choix de clauses à déplacer à celles responsables d'affaiblissement et de tautologie. Même avec cette restriction, le procédé reste coûteux. Faute de temps, nous n'avons pas encore pu expérimenter un réordonnancement qui n'effectuerait que les deux types de modifications présentées en section 4.2.2. Nous verrons s'ils permettent une amélioration de la dérivation proche de celle que nous obtenons déjà tout en réduisant fortement le temps de réordonnancement. Mais, par ailleurs, il faudra aussi envisager la possibilité de déplacer un bloc consécutif de clauses plutôt qu'une seule à la fois.

La RRL comme pré-processus

Même si l'objectif de la RRL est d'obtenir une réfutation, elle peut s'arrêter avant d'avoir pu dériver la clause vide. Dans ce cas, son utilisation peut quand même être utile si elle a produit des clauses courtes, unaires ou binaires. En effectuant une propagation unitaire, nous réduisons la taille de la formule initiale. En ajoutant toutes les clauses binaires à la formule initiale, nous favorisons le filtrage des clauses pour la méthode de résolution qui prendrait le relais.

C'est pourquoi, dans la dernière version de notre algorithme, nous mémorisons toutes les clauses binaires et unaires produites qui n'étaient pas dans la formule. Lorsque la RRL est terminée, si elle a échoué à trouver une réfutation mais qu'elle a produit de nouvelles clauses courtes, nous simplifions la formule en effectuant une propagation unitaire et en supprimant les clauses à littéral pur ou subsumée par une clause courte, puis nous relançons la RRL. De cette façon, nous sommes parvenus à réfuter plus de formules qu'auparavant.

[1] Nous n'avons pas encore eu le temps de tester une heuristique de choix de clause basée sur les principes donnés en section 4.2.1.

6 Travaux connexes et comparaisons expérimentales

Nous ne connaissons que deux autres tentatives de faire de la recherche locale pour démontrer l'insatisfiabilité : GUNSAT [1] et RANGER [6]. Ces deux algorithmes, bien qu'ayant des raffinements différents, sont basés sur le même principe itératif : parmi un ensemble de clauses d'entrées et de résolvantes précédemment produites, choisir au mieux deux clauses qui produiront une nouvelle résolvante (la clause la plus courte dans RANGER, un critère plus complexe dans GUNSAT). Aucun de ces deux algorithmes n'exhibe une réfutation et se contente d'essayer d'obtenir la clause vide. En imaginant qu'on associe à chaque résolvante deux pointeurs sur les clauses qui l'ont produite, on pourrait considérer que ces méthodes maintiennent un ensemble de sous-arbres (ie, de morceaux de preuves) dont certains sont suceptibles d'appartenir à la réfutation de la formule. Nous voyons bien alors ce qui différencie ces méthodes de la nôtre. Alors que nous restons focalisés sur une seule dérivation que nous nous autorisons à modifier à n'importe quel endroit, GUNSAT et RANGER essaient de fusionner des sous-arbres de preuves parmi un certain nombre de candidats. Nous avons donc des approches très différentes dont il n'est pas aisé de déterminer a priori celle qui pourrait être la plus efficace.

En fait, notre méthode doit surtout être comparée à la SL-résolution dans la mesure où la RRL est à la SL-résolution ce que sont les méthodes locales de recherche de modèles à DPLL : le sacrifice de la systématicité due à la liberté de modification d'une solution imparfaite en échange de la possiblité d'aller plus directement vers une solution. La SL-résolution ne revenant que sur le dernier choix de clause (qu'elle a ajouté à la fin de la dérivation), un mauvais choix de clause en début de dérivation peut mener à explorer inutilement un grand espace de recherche.

Premières expérimentations

Nous avons d'abord voulu vérifié l'efficacité relative de LRR par rapport à sa version systématique, la SL-résolution, et un autre algorithme classique basé sur la résolution, DP60. Nous avons pu constater que la SL-résolution ne parvenait à résoudre que de petits problèmes. Par exemple, aucun quand il y a 16 va-

riables. Par rapport à DP60, LRR devient meilleure lorsque le nombre de variables augmente.

#var	SL-res	DP60	LRR
10	3.9	0.18	0.57
16	-	0.68	1.08
20	-	2.26	1.67
24	-	13.6	10.0

FIG. 6: Comparaison entre la SL-resolution, DP60 et LRR sur 100 problèmes aléatoires insatisfiables au pic de difficulté (#clauses/#vars = 4.25).

Nous avons aussi voulu comparer LRR avec GUNSAT. GUNSAT est très souvent bien meilleure (typiquement 10 fois plus rapide) que LRR sauf sur quelques instances dans lesquelles elle reste coincée extrêmement longtemps (il faut l'arrêter avant qu'elle ait terminé) et qui augmente quelque peu artificiellement la moyenne des temps.

7 Conclusion et perspectives

Nous avons introduit une nouvelle méthode de preuve d'insatisfiabilité d'une formule SAT par une recherche locale. Elle imite parfaitement les méthodes locales destinées à trouver des solutions dans la mesure où nous avons fait en sorte de reformuler efficacement une restriction raisonnable de la question de l'insatisfaibilité d'une formule SAT en une question sur la consistance d'un CSP. Comme nous sommes face à un CSP très particulier mais qui permet de résoudre un problème très général, il est utile de raffiner et d'adapter autant que possible les méthodes de réparation locales traditionnelles. Ainsi, nous avons proposé une définition du voisinage d'une dérivation qui diffère de celle traditionnellement utilisée (consistant à modifier la valeur d'une seule variable) dans la mesure où nous incluons la possibilité de permuter les valeurs de variables (ie, les places des clauses dans la dérivation). De même, nous avons proposé des critères d'évaluation de proximité d'une affectation à une solution qui prenaient en compte le contexte (ie, la réfutation). Comme notre approche diffère grandement des techniques habituelles, nous savons qu'il faudra du temps pour étudier et perfectionner ses différents aspects afin de la rendre compétitive face aux méthodes traditionnelles de type DPLL qui bénéficient de plusieurs dizaines d'années d'efforts d'amélioration. Nous espérons néanmoins que le potentiel que nous lui voyons attirera l'attention d'autres membres de notre communauté, qui souhaiteront explorer les nombreux moyens de la rendre plus efficace, comme nous-même allons continuer à le faire.

Remerciements

Ce travail a été soutenu par un programme blanc de l'ANR (projet UNLOC).

Merci à Richard Ostrowski de m'avoir suggéré l'utilisation de la règle d'affaiblissement.

Références

[1] Gilles Audemard and Laurent Simon. Gunsat : A greedy local search algorithm for unsatisfiability. In *IJCAI*, pages 2256–2261, 2007.

[2] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2) :149–169, 2001.

[3] Kazuo Iwama. Complexity of finding short resolution proofs. In *MFCS*, pages 309–318, 1997.

[4] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4) :227–260, 1971.

[5] Donald W. Loveland. A linear format for resolution. In *Symposium on Automatic Demonstration*, pages 143–163, 1970.

[6] Steven David Prestwich and Inês Lynce. Local search for unsatisfiability. In *SAT*, pages 283–296, 2006.

[7] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[8] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. In *IJCAI (1)*, pages 50–54, 1997.