Arbres de preuves minimaux de formules irredondantes

Olivier Bailleux, Yacine Boufkhad, Olivier Roussel
CRIL, rue de l'université,
SP 16, 62307 Lens cedex
email:`{bailleux,boufkhad,roussel}@cril.univ-artois.fr`

Résumé

L'une des difficultés en intelligence artificielle consiste à trouver un jeu de test approprié pour les algorithmes développés. En ce qui concerne le problème SAT, il existe actuellement un large consensus sur l'utilisation de formules aléatoires au seuil. En effet, pour une taille de clause fixée, il existe un rapport du nombre de clauses au nombre de variables (le seuil) pour lequel les formules générées aléatoirement par le modèle sont les plus difficiles. Pour pouvoir effectuer les tests en un temps raisonnable, les formules générées doivent être de taille modérée. Cet article montre, expérimentalement, que les formules aléatoires au seuil de taille réduite possèdent une propriété qui les rend plus faciles et que n'ont pas les formules de plus grande taille : la redondance. Une formule est redondante s'il existe une sous formule stricte qui lui est équivalente. Pour mesurer la difficulté de ces formules pour la procédure DLL indépendamment de toute heuristique, nous en calculons les arbres de preuves minimaux. Pour cela, nous donnons un algorithme efficace qui permet de déterminer la taille de l'arbre DLL minimum de toute formule CNF. Le rapport entre les tailles des arbres de l'algorithme SAT et les tailles des arbres minimaux est également étudié. Ces expériences permettent alors de conclure qu'il convient de tester les algorithmes SAT sur des formules aléatoires au seuil rendues irredondantes.

1 Introduction

Le problème SAT consiste à déterminer si une formule propositionnelle exprimée sous forme normale conjonctive peut être rendue vraie en choisissant convenablement une affectation des variables. Les algorithmes complets les plus efficaces connus à ce jour pour résoudre ce problème sont tous basés sur l'algorithme DLL [DLL62]. Ces algorithmes sont testés et optimisés sur une famille de formules faciles à générer et ne possédant aucune propriété particulière a priori : les formules aléatoires de clauses de longueur k. Parmi

*Ce travail a été soutenu par l'université d'Artois, l'I.U.T. de Lens ainsi que par la Région Nord Pas de Calais dans le cadre du programme Ganymède II et d'un contrat d'objectifs.

cette famille, les formules les plus difficiles se situent à un rapport du nombre de clauses au nombre de variables bien déterminé expérimentalement, le seuil de satisfiabilité, et c'est donc avec ce rapport que les formules de tests sont générées.

Nous montrons dans cet article que les formules aléatoires au seuil ont (au moins) une propriété, la redondance, qui les rend plus faciles. Une formule est dite redondante lorsqu'au moins une de ses clauses n'apporte pas de contrainte nouvelle, ce qui signifie que sa suppression ne change pas l'ensemble des solutions. Dans le cas contraire, elle est dite irredondante.

Le première partie de cet article met en évidence cette propriété de redondance en étudiant, expérimentalement, le nombre de clauses redondantes. Il est également montré que la proportion de clauses redondantes décroît lorsque le nombre de variables augmente. La deuxième partie compare alors la difficulté des formules redondantes et irredondantes pour l'algorithme DLL en utilisant une approche originale : le calcul des tailles d'arbres minimaux pour cet algorithme. À cette fin, nous proposons une procédure permettant de déterminer la taille du plus petit arbre de réfutation DLL qui puisse prouver l'inconsistance de la formule. Il est alors montré expérimentalement que les tailles des arbres minimaux sont plus importantes pour des formules irredondantes que pour des formules redondantes. Une dernière section tire les conclusions des parties précédentes et propose de tester les algorithmes SAT sur des formules irredondantes qui ont des propriétés plus proches des formules de grande taille.

2 Formules irredondantes

Le modèle de génération de formules aléatoires k-SAT est largement utilisé pour tester les performances des algorithmes de résolution de SAT. Ceci en raison de la facilité avec laquelle on peut générer les formules les plus difficiles dans ce modèle [MSL92, LT93]. Dans le modèle k-SAT les clauses sont générées de la façon suivante : chaque clause de longueur k est tirée uniformément et avec remise parmi les $2^n\binom{n}{k}$ clauses possibles. Dans les formules ainsi générées on peut avoir des clauses qui se répètent et des clauses qui n'ajoutent pas de contraintes supplémentaires à la formule. Nous nous intéressons dans cette section à la redondance dans ces formules et à son influence sur leur difficulté. Une clause $C \in \Sigma$ est dite *redondante* dans une formule Σ si Σ et $\Sigma - \{C\}$ sont équivalentes. Une formule est dite irredondante si aucune de ses clauses n'est redondante. Nous nous intéressons tout d'abord à rendre irredondante, une formule k-SAT générée selon le modèle ci-dessus. Ceci est réalisé en supprimant itérativement les clauses redondantes jusqu'à ce que la formule en soit exempte. Pour savoir si une clause est redondante, on utilise le fait suivant : une clause C est redondante dans une formule Σ si et seulement si C est un impliqué de $\Sigma - \{C\}$. En effet, si C est un impliqué de $\Sigma - \{C\}$ alors tous les modèles de $\Sigma - \{C\}$ satisfont C. Par conséquent, l'ajout de la clause C ne modifie pas l'ensemble de modèles de $\Sigma - \{C\}$. Inversement si Σ et $\Sigma - \{C\}$ sont équivalentes alors C ne contredit aucun modèle de $\Sigma - \{C\}$ et est donc un impliqué de $\Sigma - \{C\}$. La procédure de la figure 1 calcule pour une formule Σ donnée une formule irredondante Σ_{irred} équivalente, composée d'un sous ensemble des clauses de Σ. La formule résultante Σ_{irred} dépend de l'ordre dans lequel les clauses sont examinées. Comme on peut le voir sur la figure 1 qui résume l'algorithme, il a été choisi d'utiliser, simplement, l'ordre chronologique dans lequel les clauses ont été générées. Bien que cet ordre soit fixé, ce choix ne modifie pas les statistiques que l'on peut effectuer sur un ensemble de formules générées aléatoirement. En effet, toute formule irredondante équivalente à une formule Σ aurait pu être générée avec la même probabilité puisque la formule Σ peut être générée de façon équiprobable avec n'importe quel ordre de clauses.

Nous définissons le degré de redondance d'une formule comme le rapport

$$\rho = \frac{|\Sigma| - |\Sigma_{irred}|}{|\Sigma|}$$

Pour une formule Σ, il existe plusieurs sous-formules irredondantes Σ_{irred}, et donc, on peut attribuer à Σ plusieurs degrés de redondance. Une définition plus satisfaisante serait d'agréger ces rapports en un seul en utilisant un opérateur adapté. Elle est cependant expérimentalement impraticable et nous ne considérons pour chaque formule qu'un seul degré de redondance. Néanmoins, comme les expérimentations présentées dans cette section sont réalisées à partir d'un vaste échantillon de formules aléatoires, et que chaque formule a une égale probabilité d'être générée dans un certain ordre, aucun degré de redondance ne se trouve favorisé dans le calcul de la moyenne.

IRRED(Σ)
begin
  $\Sigma_{irred} = F$;
  for each clause $C \in \Sigma$
    if $(\Sigma - \{C\}) \cup \overline{C}$ is unsat **then**
      $\Sigma_{irred} = \Sigma_{irred} - C$;
    endif
  endfor
  return Σ_{irred} ;
end.

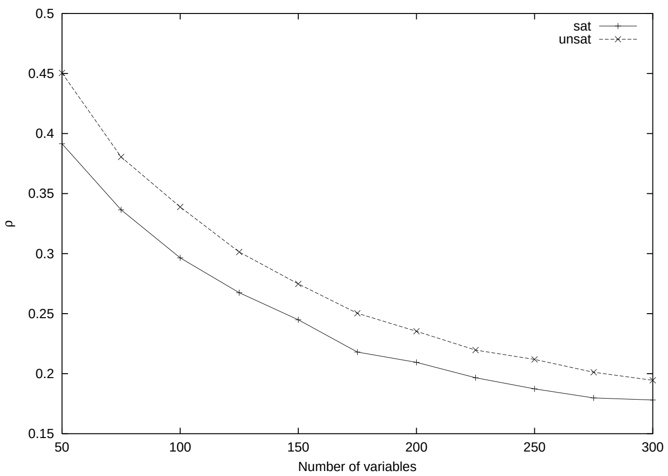
FIG. 1: Procédure IRRED

La courbe de la figure 2 représente les variations du degré de redondance en fonction du nombre de variables, le ratio du nombre de clauses sur le nombre de variables étant fixé et égal à 4.25. On constate une nette décroissance du niveau de redondance quand le nombre de variables augmente. Le degré de redondance semble tendre vers zéro lorsque le nombre de variable tend vers l'infini.

La courbe de la figure 3 représente quant à elle les variations du niveau de redondance en fonction du rapport C/V pour différentes valeurs du nombre de variables. Les formules sont irredondantes pour de faibles valeurs de C/V puis le niveau de redondance commence à augmenter à partir d'une valeur de C/V qu'on note $c_{red}(n)$. À partir de cette valeur, le niveau de redondance augmente jusqu'au seuil. Après le seuil, le niveau de redondance augmente selon une fonction linéaire du nombre de clauses car chaque clause rajoutée est redondante puisque presque toutes les formules sont non satisfiables. Le seuil de redondance $c_{red}(n)$ augmente avec le nombre de variables et tend vers le seuil de satisfiabilité.

De ces deux courbes, on déduit que les formules aléatoires tendent à être de moins en moins redondantes lorsque le nombre de variables augmente.

De plus des expérimentations rapportées dans [BR00] montrent que les formules irredondantes satisfiables sont beaucoup plus difficiles que les formules redondantes aussi



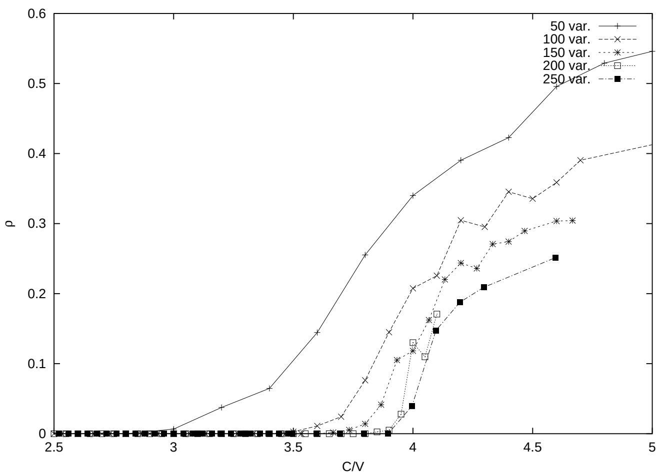
,

FIG. 2: Médiane du degré de redondance ρ des formules satisfiables et non satisfiables au seuil

bien pour les méthodes complètes que pour les méthodes stochastiques telles que walksat [SKC94]. Nous nous intéresserons dans la section suivante à la difficulté des formules non satisfiables irredondantes en comparant leurs arbres minimaux à ceux des formules redondantes correspondantes.

Arbres de preuves minimaux

En 1962, M. Davis, G. Logemann et D. Loveland ont proposé un algorithme nommé DLL [DLL62], de complexité en espace linéaire, permettant de résoudre SAT. Cet algorithme utilise deux règles d'inférences (la résolution unitaire et la règle des littéraux purs) et une règle d'énumération. La résolution unitaire permet de fixer tout littéral apparaissant seul dans une clause. La variable correspondante reçoit la valeur permettant de satisfaire cette clause. La règle des littéraux purs permet d'assigner aux variables dont toutes les occurrences sont positives (resp. négatives) la valeur vrai (resp. faux). La règle d'énumération consiste à résoudre récursivement la formule obtenue en fixant une variable à vrai, puis, si aucune solution n'est trouvée, la formule obtenue en fixant la même variable à faux. La procédure DLL est complète. Lorsque la formule d'entrée est non satisfiable, elle parcourt un arbre appelé DLL-réfutation, qui constitue une preuve de non satisfiabilité. Lorsque la formule est satisfiable, la procédure DLL exhibe une solution. La taille de l'arbre obtenu, en terme de nombre de nœuds binaires (énumérations) et/ou de nœuds unaires (résolutions unitaires et littéraux purs) est exponentielle dans le pire des cas [CS88] [BKPS98], même sous l'hypothèse peu plausible P = NP. En pratique, la taille de l'arbre produit par DLL dépend de la manière de choisir chaque variable de branchement. Les écarts constatés entre les tailles des arbres obtenus d'une part en choisissant les variables aléatoirement, d'autre part en utilisant les heuristiques (de complexité en temps polynomiale) les plus efficaces, atteignent plusieurs ordres de grandeur.



,

FIG. 3: Médiane du degré de redondance ρ des formules satisfiables et non satisfiables en fonction du rapport C/V et pour différents nombres de variables

Beaucoup d'algorithmes actuellement utilisés pour la résolution pratique de SAT (CSAT [DABC96], SATO [Zha97], SATZ [mLA97], TABLEAU [CA96]...) sont des améliorations de DLL incluant des heuristiques spécifiques pour le choix des variables de branchement et des techniques de détection précoce de certaines inconsistances.

Dans cette section, notre propos est de déterminer de manière exacte la taille minimale, en terme de nombre de nœuds binaires (i.e. de nombre d'applications de la règle d'énumération) d'une DLL-réfutation pour une formule CNF donnée.

Nous avons adopté une approche constructive de type "branch and bound" tirant parti des techniques de détection précoce d'inconsistances utilisées dans différents algorithmes de type DLL réputés performants tels que, sans être exhaustif, CSAT, SATZ ou TABLEAU. Nous proposons deux procédures ayant des complexités en temps respectives $O(2^n n!)$ et $O(3^n n!)$ et une complexité en espace polynomiale, dans le pire des cas. La première, MIN_DLL, est basée sur une recherche systématique. La deuxième procédure, RMIN_DLL, est une version "non monotone" plus efficace en pratique. Elle demeure complète, mais peut traiter deux fois de suite un même sous-problème en utilisant la première fois une borne de la taille supposée d'un arbre minimal plus optimiste que lors de la deuxième tentative. La première variante permet de traiter des formules 3CNF aléatoires de 45 variables, au seuil de satisfiabilité, avec un temps de calcul moyen de 120 secondes sur une machine 32 bits cadencée à 500Mhz. Dans les mêmes conditions, la variante non monotone requiert en moyenne 12 secondes. Les deux procédures peuvent être facilement modifiées pour exhiber une réfutation minimale.

La correction des deux procédures repose essentiellement sur les propriétés suivantes, dont nous omettrons les preuves pour rester concis :

1. Soit deux suites d'inférences (littéraux purs et/ou résolutions unitaires) transformant respectivement une formule CNF Σ en Σ_A et Σ_B telles qu'aucune nouvelle règle d'inférence ne soit applicable à Σ_A ni à Σ_B. Soit Σ_A et Σ_B contiennent la clause vide, soit $\Sigma_A = \Sigma_B$.

```
MIN_DLL(Σ,m)
begin
    if m ≤ 0 then return −1 endif ;
    if {} ∈ PROP(Σ) then return 0 endif ;
    if m ≤ 1 then return −1 endif ;
    Σ' := PROP(Σ) ; BMIN :=m ;
    let V be the set of the variables occuring in Σ' ;
    for each x in V do
        choose l ∈ {¬x, x} ;
        if {} ∈ PROP(Σ'[l]) then
            B1 := MIN_DLL(Σ'[¬l], BMIN − 1) ;
            if B1 ≠ −1 then BMIN := B1 + 1 endif
        else if {} ∈ PROP(Σ'[¬l]) then
            B1 := MIN_DLL(Σ'[l], BMIN − 1) ;
            if B1 ≠ −1 then BMIN := B1 + 1 endif
        else
            B1 :=MIN_DLL(Σ'[l], BMIN − 2) ;
            if B1 ≠ −1 then
                B2 := MIN_DLL(Σ'[¬l], BMIN − 1 − B1) ;
                if B2 ≠ −1 then BMIN := B1 + B2 + 1 endif
            endif
        endif
    endfor
    if BMIN < m return BMIN else return −1 endif ;
end.
```

FIG. 4: Procédure MIN_DLL

2. Soit T une DLL réfutation d'une formule Σ ayant pour racine la variable de branchement x et pour fils gauche (resp. droit) T_g (resp. T_d). Si T a une taille minimale sur l'ensemble des DLL-réfutations ayant pour racine x, alors T_g et T_d sont des DLL-réfutations de tailles minimales des formules obtenues à partir de Σ en branchant respectivement sur $\neg x$ et x.

La procédure MIN_DLL est présentée en figure 4. Elle utilise une procédure PROP qui accepte en paramètre une formule sous forme CNF Σ et retourne une formule obtenue à partir de Σ en effectuant des inférences de type résolution unitaire et/ou règle des littéraux purs jusqu'à ce qu'aucune règle d'inférence ne soit applicable. Appelée avec les paramètres Σ et m, MIN_DLL retourne -1 s'il n'existe aucune DLL-réfutation de Σ ayant moins de m nœuds binaires, sinon elle retourne la taille d'une DLL-réfutation de Σ minimisant le nombre de nœuds binaires. La procédure MIN_DLL se distingue de DLL par le fait qu'à chaque énumération, elle ne choisit pas une variable de branchement mais les essaie toutes. Elle utilise toutefois une heuristique pour définir l'ordre dans lequel les variables candidates sont traitées. Dans notre implantation, nous avons adopté un classement des variables basé sur la fonction de branchement de SATZ.

La procédure RMIN_DLL suit le même schéma que MIN_DLL excepté pour les appels récursifs situés dans la boucle "for". Elle exploite la propriété suivante : si T, de taille m_x, est une DLL réfutation de Σ minimale sur l'ensemble des DLL-réfutations de

Σ ayant pour racine la variable de branchement x et si $m_x < BMIN$, alors la somme des tailles des fils droit et gauche de T, T_d et T_g respectivement, vaut au plus $BMIN - 2$. Donc le plus petit de ces deux fils possède au plus $BMIN/2-1$ nœuds binaires. On peut donc calculer m_x de la manière suivante :

Soit l une assignation possible de x.

1. Brancher sur l et rechercher un arbre minimal T_l avec moins de $BMIN/2$ nœuds.
2. Si un tel arbre T_l existe, brancher sur $\neg l$ et rechercher un arbre minimal $T_{\neg l}$ ayant moins de $BMIN-1-|T_l|$ nœuds. Si un tel arbre existe, alors $m_x = |T_l|+|T_{\neg l}|+1$.
3. Si aucun arbre T_l n'a été trouvé alors brancher sur $\neg l$ et rechercher un arbre minimal $T_{\neg l}$ ayant moins de $BMIN/2$ nœuds. Si un tel arbre n'existe pas alors $m_x \geq BMIN$.
4. Si un arbre $T_{\neg l}$ ayant moins de $BMIN/2$ nœuds existe alors brancher à nouveau sur l et rechercher un arbre minimal ayant moins de $BMIN-1-|T_{\neg l}|$ nœuds (une borne moins optimiste que celle utilisée lors de la première recherche de T_l). Si un tel arbre n'existe pas alors $m_x \geq BMIN$ sinon $m_x = |T_l| + |T_{\neg l}| + 1$.

Précisons enfin que les procédures MIN_DLL et RMIN_DLL peuvent être facilement modifiées pour calculer des minorants et majorants des tailles des DLL-réfutation minimales à un coût moindre que le celui du calcul des valeurs exactes. On peut obtenir un majorant en limitant le nombre de variables de branchement essayées lors de chaque énumération. Un minorant peut être obtenu en utilisant une borne initiale m suffisamment petite pour permettre l'exécution de la procédure calcul en temps raisonnable. La valeur de retour -1 confirmera que m est un minorant de la taille d'une DLL-réfutation minimale de la formule traitée.

3 Expérimentations

L'algorithme CSAT ainsi que la procédure RMIN-DLL ont été testés sur des formules 3-SAT insatisfiables au seuil, générées selon le modèle classique pour des nombres de variables allant de 10 à 50. La complexité de la procédure RMIN-DLL a limité la taille des formules que nous avons pu tester en un temps raisonnable. Chacune de ces formules a été rendue irredondante en utilisant l'algorithme de la figure 1. MIN-DLL ainsi que CSAT ont alors été appliqués à la formule redondante et à son équivalent irredondant. La figure 5 présente, en fonction du nombre de variables, la taille moyenne des arbres générés par les algorithmes sur quelques centaines de formules générées.

Non seulement les formules irredondantes sont plus difficiles pour CSAT que les formules redondantes, mais elle ont également des arbres minimaux nettement plus grands. Par exemple, les arbres minimaux des formules irredondantes de 30 variables ont des tailles sensiblement identiques à celles des arbres minimaux des formules redondantes de 50 variables. On peut donc considérer qu'en moyenne, une formule irredondante de 30 variables présente la même difficulté intrinsèque (en terme de nombre de noeuds dans l'arbre) qu'une formule de 50 variables.

La figure 6 présente, sous forme de moyennes, les rapports $MI(\Sigma')/MR(\Sigma)$, $CI(\Sigma')/CR(\Sigma)$, $CR(\Sigma)/MR(\Sigma)$ et $CI(\Sigma')/MI(\Sigma')$, où $MR(\Sigma)$ représente le nombre de noeuds d'un arbre minimal d'une formule Σ, $MI(\Sigma')$ le nombre de noeuds d'un arbre minimal d'une formule irredondante Σ' obtenue à partir de Σ, $CR(\Sigma)$ le nombre de noeuds requis

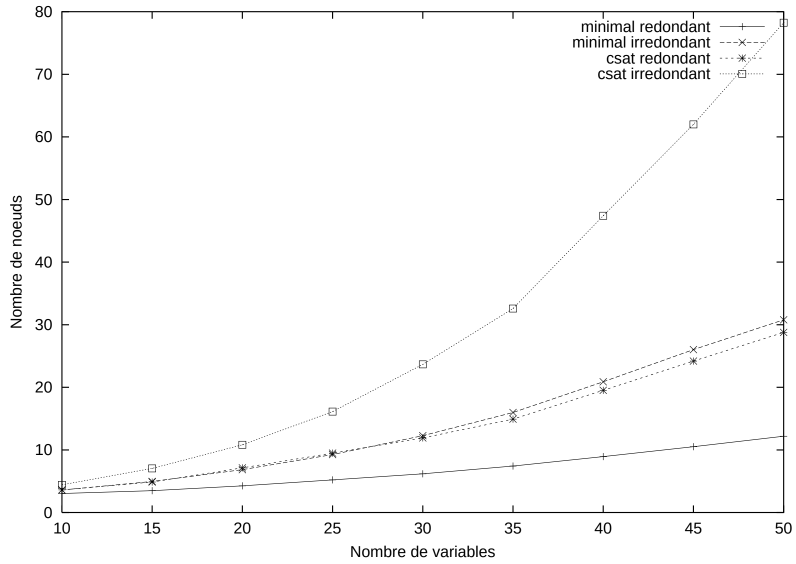


FIG. 5: Tailles moyennes des arbres minimaux et des arbres produits par CSAT pour des formules aléatoires au seuil et pour les formules irredondantes associées, en fonction du nombre de variables.

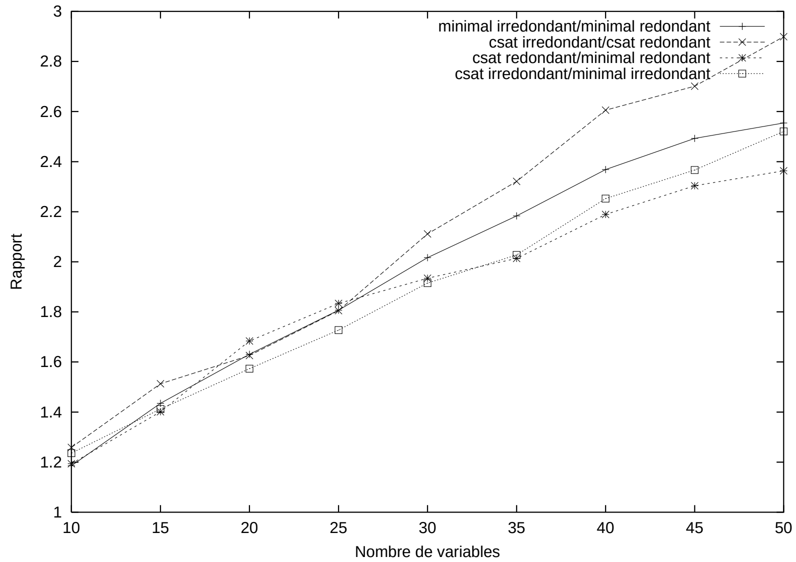


FIG. 6: Moyennes des rapports entre les tailles des arbres générés de deux manières (CSAT et arbres minimaux) pour les formules aléatoires au seuil et pour les formules irredondantes associées, en fonction du nombre de variables.

par CSAT pour résoudre Σ et $CI(\Sigma')$ le nombre de noeud requis par CSAT pour résoudre Σ'. Les mesures proviennent des échantillons utilisés figure 5.

Lorsque la taille des formules augmente, on constate les faits suivants :

- Le rapport $CI(\Sigma')/CR(\Sigma)$ augmente, donc la suppression des clauses redondantes affecte d'autant plus les performances de CSAT que la taille des formules est grande.
- On observe la même tendance avec les arbres minimaux même si pour les formules les plus grandes de nos échantillons, CSAT semble être plus affecté que les arbres minimaux par la suppression des clauses redondantes.
- Pour les formules redondantes comme pour les formules irredondantes, la taille des arbres de CSAT s'éloigne de plus en plus de la taille des arbres minimaux.

Ces résultats, combinés à l'étude de l'évolution de la redondance en fonction de la taille des formules de la section 2, justifient l'intérêt de construire un générateur de formules aléatoires irredondantes qui ont des caractéristiques plus semblables aux formules de grande taille que les formules redondantes.

4 Générateur de formules aléatoires irredondantes

On construit un ensemble irredondant de clauses en y ajoutant itérativement des clauses générées aléatoirement après avoir vérifié que l'ajout d'une clause ne rend pas l'ensemble redondant. Le problème principal qui doit être résolu est qu'il faut non seulement vérifier que la nouvelle clause n'est pas redondante mais aussi qu'elle ne rend pas redondante une clause qui est déjà dans la formule. Pour éviter de faire systématiquement des tests de redondance pour chaque clause, on utilise la propriété suivante : pour chaque clause irredondante C dans une formule Σ, il existe un impliquant I de $\Sigma - \{C\}$ qui contredit C. Nous appellerons un tel impliquant le témoin de l'irredondance de C dans Σ. À une étape quelconque de la procédure de génération, la formule courante Σ est irredondante et chacune de ses clauses a son témoin. Lorsqu'une nouvelle clause C est générée, sa redondance est testée. Si elle est irredondante alors l'algorithme teste si elle est satisfaite par tous les témoins des clauses dans la formule. Si elle n'est pas satisfaite par un témoin quelconque alors la redondance de la clause correspondant à ce témoin dans la formule $\Sigma \cup \{C\}$ doit être testée, et si elle est irredondante, alors un nouveau témoin doit être trouvé. L'algorithme essaye d'abord de modifier le témoin en y ajoutant des littéraux de façon à satisfaire C si possible. Alors un nouveau témoin est recherché. Si aucun témoin n'existe alors la clause C est rejetée. Ces étapes sont résumées ci-dessous :

- Tester la satisfiabilité de $\Sigma \cup \overline{C}$, si elle est non satisfiable alors rejeter la clause C. Sinon mémoriser l'impliquant trouvé par le test précèdent comme témoin pour C.
- Si C n'est pas redondante, tester si C est satisfaite par tous les témoins des clauses déjà dans Σ. Pour tout témoin t_i qui ne satisfait pas C (t_i correspondant à une clause C_i) :
  1. Soit il existe un littéral de C dont la variable correspondante n'est pas dans t_i, ajouter ce littéral à t_i de façon à satisfaire C.
  2. Soit il n'existe pas un tel littéral, auquel cas tester si C_i est irredondante dans $\Sigma \cup \{C\}$. Si elle l'est alors l'impliquant trouvé par ce test remplace t_i, sinon C est rejetée et tous les témoins sont remis à leurs valeurs initiales.

À la fin de ces étapes, si la clause C n'est pas rejetée, elle est ajoutée à l'ensemble de clauses courant. La terminaison de cet algorithme n'est pas garantie car il se peut qu'à une étape donnée, toutes les clauses tirées aléatoirement soient rejetées. Il est nécessaire de limiter le nombre de tentatives pour rajouter de nouvelles clauses. Si ce nombre est atteint l'algorithme répond qu'il a échoué dans la génération d'une formule. Ce qui serait notamment le cas, si la formule devient non satisfiable avant d'avoir atteint le nombre de clauses requis par l'utilisateur.

5 Conclusion

La recherche d'arbres minimaux est une méthode très intéressante qui permet d'étudier certaines propriétés de l'algorithme DLL et en particulier la difficulté de certaines formules indépendamment des heuristiques qui peuvent être utilisées dans sa mise en oeuvre. En parallèle avec les expérimentations sur l'un des algorithmes de satisfiabilité les plus performants, cette technique a permis également dans cet article de montrer clairement que les formules aléatoires irredondantes étaient beaucoup plus difficiles que les formules redondantes correspondantes pour DLL. Nous obtenons donc une double confirmation de cette difficulté accrue des formules irredondantes. Dans les deux cas, les formules rendues irredondantes sont significativement plus difficiles que les formules redondantes dont elles sont issues. Comme les formules aléatoires deviennent de moins en moins redondantes au fur et à mesure que leur taille grossit, on en déduit que les expériences sur les algorithmes de satisfiabilité doivent être menées sur des formules rendues irredondantes qui sont celles qui se rapprochent le plus par leur propriétés des grandes formules générées aléatoirement.

Références

[BKPS98] Paul Beame, Richard Karp, Toniann Pitassi, and Michael Saks. On the complexity of unsatisfiability proofs for random k-cnf formulas. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 561–571, 1998.

[BR00] Yacine Boufkhad and Olivier Roussel. Redundancy in random sat formulas. In *to appear in the Proceedings of AAAI 2000*, 2000.

[CA96] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3sat. *Artificial Intelligence*, 81 :31–57, 1996.

[CS88] V. Chvátal and E. Szemerédi. Many hard example for resolution. *Journal of the ACM*, pages 759–768, 1988.

[DABC96] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. Sat versus unsat. In *Clique, Coloring and Salisfiability : Second DIMACS implementation challenge, Volume 26 of DIMACS series in Discrete Mathematics and Computer Science*, pages 415–436. American Mathematical Society, 1996.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5 :394–397, 1962.

[LT93] T. Larabee and Y. Tsuji. Evidence for a Satisfiability Threshold for Random 3CNF Formulas. In *Proceedings AAAI Spring Symposia*, 1993.

[mLA97] Chu min Li and Anbulagan. Heuristic based on unit propagation for satisfiability problems. In *Proceedings of IJCAI 97*, pages 366–371, 1997.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and Easy Distribution of SAT Problems. In *Proceedings of AAAI'92*, pages 459–465, July 1992.

[SKC94] Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of AAAI 94*, pages 337–343, 1994.

[Zha97] H. Zhang. Sato : An efficient propositional prover. In *Proceedings of CADE 97*, pages 272–275, 1997.