Intégration d'un Raisonnement d'Equivalence dans la Procédure Davis-Putnam

Chu Min Li
LaRIA, Université de Picardie Jules Verne
5 Rue du Moulin Neuf, 80000 Amiens
email : `cli@laria.u-picardie.fr`
URL : http://www.laria.u-picardie.fr/~cli

Résumé

Les clauses d'équivalence (Xors or modulo 2 arithmétique) représentent une structure commune lors du codage en SAT de nombreux problèmes réels et constituent un obstacle majeur pour la procédure Davis-Putnam (DP). Nous proposons une technique spéciale appelée *Raisonnement d'Equivalence* pour surmonter l'obstacle et montrons la performance d'une procédure DP renforcée par le raisonnement d'équivalence pour les instances SAT contenant des clauses d'équivalence provenant des problèmes de parité, de recherche de clé cryptographique et de vérification bornée de modèle. Nos résultats montrent que l'intégration d'un raisonnement d'équivalence facilite beaucoup de problèmes qui étaient hors de portée de la procédure DP. Nous comparons aussi le raisonnement d'équivalence avec les techniques CSP générales "regarder en arrière" sur les clauses d'équivalence.

1 Introduction

Considérons une formule propositionnelle $\mathcal{F}$ sous la forme normale conjonctive (CNF) sur un ensemble de variables booléennes $\{x_1, x_2, ..., x_n\}$. Le problème *satisfiabilité (SAT)* est de tester si les clauses dans $\mathcal{F}$ peuvent être satisfaites simultanément par une affectation de valeurs de vérité (0 ou 1) aux variables. SAT est fondamental dans beaucoup de domaines de l'informatique, de l'electronique et des mathématiques. Il est le premier problème NP-Complet [3].

Soit l avec ou sans indice un littéral. Une clause d'équivalence de longueur k peut être écrite comme suit :

$$l_1 \leftrightarrow l_2 \leftrightarrow ... \leftrightarrow l_k$$

où l'opérateur $\leftrightarrow$ est commutatif and associatif. La clause d'équivalence est équivalente à 2^{k-1} clauses CNF. Par exemple, une clause d'équivalence binaire est equivalente à 2 clauses CNF: $l_1 \vee \bar{l}_2$ et $\bar{l}_1 \vee l_2$, et une clause d'équivalence ternaire est equivalente à 4 clauses CNF: $l_1 \vee \bar{l}_2 \vee \bar{l}_3$, $\bar{l}_1 \vee \bar{l}_2 \vee l_3$, $\bar{l}_1 \vee l_2 \vee \bar{l}_3$, et $l_1 \vee l_2 \vee l_3$.

Quand on code un problème réel en SAT, on structure souvent le problème en couches et utilise des abréviations et des définitions, ce qui donnent souvent de nombreuses clauses d'équivalence (dans leur forme CNF) dans l'instance SAT finale. Malheureusement les clauses d'équivalence de longueur > 2 sont généralement très difficiles à résoudre pour la procédure Davis-Putnam (DP) [5], qui est la meilleur méthode complète pour SAT.

Il semble que les clauses d'équivalence donnent très peu de clauses unitaires tout au long de la résolution alors que la procédure DP rencontre souvent beaucoup de clauses unitaires lors de la résolution des autres problèmes, ce qui explique l'inefficacité de la procédure DP pour les clauses d'équivalence. D'un autre côté, fixer une variable dans les clauses d'équivalence produit souvent un certain nombre de littéraux équivalents à partir desquels on peux faire un raisonnement pour obtenir d'autres équivalences, ce qui peut remédier à l'inefficacité de la propagation unitaire. Dans ce papier, nous montrons comment intégrer le raisonnement d'équivalence dans la procédure DP pour résoudre les problèmes SAT contenant à la fois les clauses d'équivalence (appelées *partie EQ*) et d'autres clauses CNF (appelées *partie CNF*).

Ce papier est organisé de la façon suivante. Dans la section 2 nous définissons le raisonnement d'équivalence. Dans la section 3 nous présentons la procédure DP appelée *EqSatz* renforcée par le raisonnement d'équivalence. Notre approche était à l'origine motivée par le deuxième challenge formulé par Selman et al. [8] lors de l'IJCAI'97. La section 4 rapporte la performance de *EqSatz* pour le challenge. Dans la section 5 nous rapportons et comparons la performance de *EqSatz* avec les 4 autres DPs sur différents problèmes SAT contenant des clauses d'équivalence. Tous les résultats expérimentaux sont obtenus sur un Macintosh G3 300 Mhz muni d'une mémoire de 96 Mo sous système PPClinux, le temps est exprimé en seconde. La section 6 discute des travaux associés et la section 7 conclue.

2 Le Raisonnement d'Equivalence

Une clause d'équivalence peut être complémentée avec la propriété suivante:

$$\begin{aligned} \neg(l_1 \leftrightarrow l_2 \leftrightarrow ... \leftrightarrow l_k) &\equiv \bar{l}_1 \leftrightarrow l_2 \leftrightarrow ... \leftrightarrow l_k \\ &\equiv l_1 \leftrightarrow \bar{l}_2 \leftrightarrow ... \leftrightarrow l_k \\ &... \\ &\equiv l_1 \leftrightarrow l_2 \leftrightarrow ... \leftrightarrow \bar{l}_k \end{aligned} \tag{1}$$

Puisque toutes clauses d'équivalence de longueur > 3 peuvent être simplement transformées en clauses d'équivalence ternaires par l'ajout des nouvelles variables, nous ne considérons que les clauses d'équivalence ternaires et binaires dans ce papier. Nous définissons six règles d'inférence sur elles.

$$l_1,\ l_1 \leftrightarrow l_2 \leftrightarrow l_3 \ \vdash\ l_2 \leftrightarrow l_3 \tag{2}$$

$$\bar{l}_1,\ l_1 \leftrightarrow l_2 \leftrightarrow l_3 \ \vdash\ \neg(l_2 \leftrightarrow l_3) \tag{3}$$

$$l_1 \leftrightarrow l_1 \leftrightarrow l_2 \ \vdash\ l_2 \tag{4}$$

$$l_1 \leftrightarrow l_2 \leftrightarrow l_3,\ l_1 \leftrightarrow l_2 \leftrightarrow l_4 \quad \vdash \quad l_3 \leftrightarrow l_4 \tag{5}$$
$$l_1 \rightarrow (l_3 \leftrightarrow l_4), \bar{l}_1 \rightarrow (l_3 \leftrightarrow l_4) \quad \vdash \quad l_3 \leftrightarrow l_4 \tag{6}$$
$$l_1 \rightarrow (l_3 \leftrightarrow l_4), \bar{l}_1 \rightarrow (\bar{l}_3 \leftrightarrow l_4) \quad \vdash \quad l_1 \leftrightarrow l_3 \leftrightarrow l_4 \tag{7}$$

Notons que la partie droite de la règle 3 peut être réécrite sous la forme $\bar{l}_2 \leftrightarrow l_3$ ou $l_2 \leftrightarrow \bar{l}_3$ en utilisant la propriété 1. Toutes ces règles peuvent être réalisées par un nombre constant d'étapes de résolution après avoir écrit les clauses d'équivalence sous forme CNF. Nous appelons l'application de ces règles dans une formule $\mathcal{F}$ *le raisonnement d'équivalence.*

Les clauses d'équivalence binaires jouent un rôle particulier dans notre approche. Chaque fois que l'on obtient une équivalence binaire à partir de la règle 2, 3, 5, ou 6, l_1 ($\bar{l}_1$) est remplacé par l_2 ($\bar{l}_2$) dans $\mathcal{F}$. On peut montrer que le problème DIMACS[1] dubois* est résolu en temps linéaire par application répétée de la règle 5 et le remplacement des littéraux équivalents.

Les clauses d'équivalence ternaires sont représentées à la fois dans leur forme CNF et par une liste de trois variables concernées avec éventuellement un opérateur négation. Par exemple, la clause d'équivalence $\bar{x} \leftrightarrow \bar{y} \leftrightarrow \bar{z}$ est représentée par 4 clauses CNF et par $\neg(x, y, z)$.

La règle 7 est utilisée pour ajouter de nouvelles clauses d'équivalence ternaires à $\mathcal{F}$. Etant donné 3 clauses d'équivalence ternaires $x \leftrightarrow u \leftrightarrow v$, $y \leftrightarrow u \leftrightarrow w$, et $z \leftrightarrow v \leftrightarrow w$, on ajoute une clause unitaire x, la règle 2 appliquée à la première clause d'équivalence donne $u \leftrightarrow v$. On remplace alors u par v dans la deuxième clause d'équivalence avant d'appliquer la règle 5 pour obtenir $y \leftrightarrow z$. Donc $x \rightarrow (y \leftrightarrow z)$. De la même manière $\bar{x} \rightarrow (\bar{y} \leftrightarrow z)$. On obtient alors $x \leftrightarrow y \leftrightarrow z$, une clause d'équivalence à ajouter dans $\mathcal{F}$, en appliquant la règle 7. La règle 6 est aussi utilisée de cette façon pour déduire de nouveaux littéraux équivalents.

On peut montrer que le problème DIMACS pret*, bien que très difficile pour une implantation classique de la procédure DP, peut être résolu en temps linéaire en ajoutant successivement des nouvelles clauses d'équivalences dans $\mathcal{F}$.

3 *EqSatz*: Une Procédure DP Renforcée par le Raisonnement d'Equivalence

Nous implantons les 6 régles d'inférence définies dans la section 2 dans une procédure DP hautement optimisée appelée *Satz* [6]. Satz se compose d'un propagateur de clauses unitaires et d'une heuristique puissante pour sélectionner la prochaine variable de branchement afin de maximiser la réduction de l'espace de recherche. Satz renforcée par le raisonnement d'équivalence est appelée *EqSatz* et est résumée dans la figure 1. La figure 2 montre l'implantation du raisonnement d'équivalence. La figure 3 montre la règle de branchement de *EqSatz*.

Les clauses d'équivalence contenues dans la formule d'entrée sont sous la forme CNF. Les règles 2 et 3 sont exécutées par la propagation unitaire, ainsi le raisonnement d'équivalence commence par détecter syntaxiquement les littéraux équivalents produits par la propagation unitaire. Si l_1 est équivalent à l_2, il est remplacé par l_2. Notons que le remplacement

1. ftp://dimacs.rutgers.edu/pub/challenge/sat

peut produire une clause unitaire dans le cas où $l_1 \vee l_2$ est une clause dans $\mathcal{F}$. La propriété 1 est utilisée pour réécrire les clauses d'équivalence sous une forme appropriée afin d'appliquer les règles d'inférence.

Procedure $EqSatz(\mathcal{F})$
Debut
```
  Si F est vide, retourner "satisfiable";
  F:=Equivalency_Reasoning(UnitPropagation(F));
  si F contient une clause vide, retourner "insatisfiable".

  /* regle de branchement */
  selectionne une variable x dans F.
  si EqSatz(F ∪ {x}) est "satisfiable" alors retourner "satisfiable",
  sinon retourner le resultat de EqSatz(F ∪ {x̄}).
```
Fin.

procedure UnitPropagation($\mathcal{F}$)
Debut
```
  Tant qu'il n'y a pas de clause vide et une clause unitaire l existe,
  satisfaire l et simplifier F.
  Retourner F.
```
Fin.

FIG. 1 – *La Procédure DP EqSatz*

Le raisonnement d'équivalence est naturellement intégré dans l'heuristique de Satz. Etant donnée une variable libre x, *Satz* examine x par deux propagations unitaires expérimentales en ajoutant respectivement les clauses unitaires x et $\bar{x}$ dans $\mathcal{F}$ pour voir l'impact du branchement par x. Suivant cette ligne, *EqSatz* effectue un raisonnement d'équivalence expérimental après chaque propagation unitaire expérimentale pour regarder encore plus loin en avant et ajouter de nouvelles clauses d'équivalence dans $\mathcal{F}$ en utilisant les règles d'inférence 6 et 7.

Comme Satz, *EqSatz* essaie de brancher par la variable permettant de maximiser la réduction de l'espace de recherche tout en prenant le raisonnement d'équivalence en compte. Elle utilise 3 fonctions pour estimer la réduction de l'espace de recherche.

La première fonction est $nb\_fixed\_vars(\mathcal{F}_1, \mathcal{F}_2)$ qui donne le nombre de variables de $\mathcal{F}_2$ instantiées dans $\mathcal{F}_1$, la seconde est $nb\_eq\_pairs(\mathcal{F}_1, \mathcal{F}_2)$ qui donne le nombre de littéraux équivalents de $\mathcal{F}_2$ remplacés dans $\mathcal{F}_1$. La motivation est que l'instantiation d'une variable ou d'un nouveau littéral équivalent déduit tel que $l_3 \leftrightarrow l_4$ réduit l'espace de recherche de moitié. La troisième fonction $nb\_binary\_clauses(\mathcal{F}_1, \mathcal{F}_2)$ est définie en suivant la même ligne, mais la considération est un peu plus compliquée.

$$nb\_binary\_clauses(\mathcal{F}_1, \mathcal{F}_2) = \sum_{l \vee l' \text{ est dans } \mathcal{F}_1 \text{ mais non dans } \mathcal{F}_2} [f(\bar{l}) + f(\bar{l'})]$$

où $f(\bar{l})$ est le nombre d'occurrences de $\bar{l}$ dans $\mathcal{F}_2$.

Si $\mathcal{F}$ a n variables, elle a 2^n solutions possibles. Une clause binaire supprime 2^{n-2} solutions. Cependant deux clauses binaires ne suppriment pas 2^{n-1} solutions en général. En fait deux clauses binaires partageant un litéral identique ou n'ayant aucune variable en commun comme $x_1 \vee x_2$ et $x_1 \vee x_3$ ou $x_1 \vee x_2$ et $\bar{x}_3 \vee x_4$ suppriment respectivement

$2^{n-1} - 2^{n-3}$ et $2^{n-1} - 2^{n-4}$ solutions. Il n'y a que deux clauses binaires partageant un litéral complémenté comme $x_1 \vee x_2$ et $\bar{x}_1 \vee x_3$ suppriment 2^{n-1} solutions.

```
Procedure Equivalency_Reasoning(𝓕)
Debut
  𝓕:=Set_Equivalence_by_CNF_Clauses(𝓕);
  repeter
    /* Regle 4 */
    Tant qu'il y a une clause d'equivalence contenant deux lit-
    teraux identiques comme l1 ↔ l1 ↔ l3 et pas de clause vide faire
    debut
      satisfaire l3 et simplifier 𝓕;
      𝓕:=Set_Equivalence_by_CNF_Clauses(𝓕);
    fin;
    /* Regle 5 */
    tant qu'il y a deux clauses d'equivalence contenant deux litteraux identiques
    comme l1 ↔ l2 ↔ l3 et l1 ↔ l2 ↔ l4 mais pas de clause vide faire
      si l3 = ¬l4 ajouter une clause vide dans 𝓕
      sinon
      debut
        𝓕:=Substitute(l3, l4, 𝓕);
        𝓕:=Set_Equivalence_by_CNF_Clauses(𝓕);
      fin
  jusqu'a ce que une clause vide soit produite ou aucun changement
  ne se produit dans 𝓕
Fin.

procedure Set_Equivalence_by_CNF_Clauses(𝓕)
Debut
  Tant que 𝓕 contient deux clauses CNF binaires comme
  l1 ∨ l̄2 et l̄1 ∨ l2 mais pas de clause vide faire
    𝓕:=Substitute(l1, l2, 𝓕);
  retourner 𝓕
Fin.

procedure Substitute(l1, l2, 𝓕)
Debut
  remplacer toutes les occurrences de l1(l̄1) par l2(l̄2);
  𝓕:=UnitPropagation(𝓕); retourner 𝓕;
Fin.
```

FIG. 2 – *La Procédure du raisonnement d'équivalence*

Clairement les clauses binaires partageant un litéral complémenté suppriment beaucoup plus de solutions et ont plus de chance de conduire la DP à une contradiction où toutes les solutions sont supprimées. Donc une procédure DP doit brancher sur la variable permettant de générer le plus de clauses binaires partageant un littéral complémenté.

```
Pour chaque variable libre x faire
Soit F' and F'' deux copies de F
Debut
  F' := Equivalency_Reasoning(UnitPropagation(F' ∪ {x}));
  F'' := Equivalency_Reasoning(UnitPropagation(F'' ∪ {x̄}));
  Si F' et F'' contiennent toutes les deux une clause vide
    alors retourner "F est insatisfiable".
  sinon si F' contient une clause vide alors x := 0, F := F''
  sinon si F'' contient une clause vide alors x := 1, F := F';
  sinon
  debut
    /* Regle 6 */
    Pour chaque l3 ↔ l4 ∈ F' ∧ F''  F:=Substitute(l3,l4,F);
    /* Regle 7 */
    pour chaque l3 ↔ l4 ∈ F' et l̄3 ↔ l4 ∈ F'' ajouter x ↔ l3 ↔ l4 dans F.
    F:=Equivalency_Reasoning(F);
    si F contient une clause vide alors retourner "F est insatisfiable".
    soit w(l) le poid du litteral l,
    w(x) := nb_fixed_vars(F', F) + nb_eq_pairs(F', F) + nb_bin_cls(F', F)/2
    w(x̄) := nb_fixed_vars(F'', F) + nb_eq_pairs(F'', F) + nb_bin_cls(F'', F)/2;
  fin
Fin;
Pour chaque variable libre x faire H(x) := w(x̄) * w(x) * 1024 + w(x̄) + w(x);
Brancher sur la variable libre x tel que H(x) est le plus grand.
```

FIG. 3 – *La règle de branchement de EqSatz*

4 Résoudre le Challenge du Problème de Parité

EqSatz était à l'origine motivée par le challenge du problème de parité formulé par Selman et al. [8] lors de l'IJCAI'97. A notre connaissance, *EqSatz* est la seule procédure capable de résoudre toutes les 10 instances DIMACS par32* en un temps raisonnable.

Le tableau 1 montre la performance de *EqSatz* sur le challenge. Comme dans la section suivante, #cls et #eq_cls[2] dénotent respectivement le nombre total de clauses dans la formule d'entrée et le nombre de clauses d'équivalence ternaires dans la formule CNF d'entrée, une clause d'équivalence ternaire étant comptée comme 4 clauses CNF dans #cls. Comme on peut voir, toutes les instances contiennent une partie EQ large.

Le raisonnement d'équivalence rend *EqSatz* substantiellement plus rapide que Satz pour le challenge. Dans la section suivante, nous montrons qu'il est suffisament puissant pour rendre une procédure DP capable de résoudre d'autres problèmes qui étaient hors de portée de la DP.

2. *EqSatz* hérite de Satz un pré-traitement de la formule d'entrée qui cherche et ajoute les résolvants de longueur ≤ 3, ce qui peut supprimer certaines clauses d'équivalence. #eq_cls dénote le nombre de clauses d'équivalence ternaires restantes.

TAB. 1 – *Temps d'exécution et taille de l'arbre de recherche (t_size) de EqSatz pour le challenge du problem de parité*

Instance	#var	#cls	#eq_cls	time	t_size
par32-1-c	1315	5254	1097	1133	3672
par32-2-c	1303	5206	1085	50	209
par32-3-c	1325	5294	1107	3972	15123
par32-4-c	1333	5326	1115	793	1488
par32-5-c	1339	5350	1121	9265	38348
par32-1	3176	10277	1097	989	3089
par32-2	3176	10253	1085	241	651
par32-3	3176	10297	1107	8899	23827
par32-4	3176	10313	1115	827	2885
par32-5	3176	10325	1121	11855	35133

5 D'Autres Résultats Expérimentaux

Nous utilisons trois benchmarks de la littérature contenant des clauses d'équivalence pour évaluer l'impact du raisonnement d'équivalence dans une procédure DP et comparer la performance de *EqSatz* avec 4 procédures DP sur les même instances: le problème DIMACS pret* , le problème cryptographique de Massacci (Data Encryption Standard (DES[3])), et les problèmes de vérification bornée de modèle de Biere et al. (BMC[4]).

Les 4 DPs comparées sont Sato [13], Grasp [9], Relsat [1] et Satz. Sato, Grasp et Relsat utilisent à la fois les techniques “regarder en avant” (look-ahead) comme la propagation unitaire et l'heuristique pour choisir la prochaine variable de branchement et les techniques “regarder en arrière” (look-back) comme le retours en arrière intelligent et apprentissage, alors que Satz utilise uniquement les techniques “regarder en avant”. Donc la comparaison entre *EqSatz* et Satz dans l'experimentation illustre l'impact du raisonnement d'équivalence. La comparaison de *EqSatz* avec Sato, Grasp[5], et Relsat pourrait être considérée comme celle entre les techniques “regarder en arrière” et le raisonnement d'équivalence sur les instances contenant des clauses d'équivalence.

5.1 Performance sur le problème DIMACS pret*

Le problème a été élaboré par Pretolani, inspiré de la construction de Urquhart [10]. Nous modifions quelques constantes dans le générateur disponible au même site pour générer des instances plus grandes. Ces instances insatisfiables se composent uniquement de clauses d'équivalence.

EqSatz résout ce problème en temps linéaire. Le tableau 2 montre la performance des 5 procédures. Notons que bien que Sato, Grasp et Relsat sont substantiellement plus rapides que Satz pour ces instances, il semble qu'ils aient encore un comportement exponentiel.

3. http://www.uni-koblenz.de/~massacci

4. http://www.cs.cmu.edu/~modelcheck

5. Lors de la résolution de certaines instances, Grasp est arrêtée avant deux heures d'exécution à cause du manque de mémoire ou d'autres ressources. Son temps d'exécution est alors marqué par “?”

TAB. 2 – *Temps d'exécution de Sato, Grasp, Relsat, Satz et EqSatz pour le problème DIMACS pret**

#vars	300	450	600	750	1500	3000
#eq_cls	200	300	400	500	1000	2000
Grasp	9	31	82	166	1742	> 7200
Sato	28	37	102	> 7200	-	-
Relsat	1	2	4	10	72	696
Satz	> 7200	-	-	-	-	-
EqSatz	0	0	0	0	0	0

5.2 Performance sur les instances DES

Les instances DES a été élaborées par Massacci [7]. Ce sont le codage en SAT du problème de recherche de clé cryptographique et contient des clauses d'équivalence à partir de 3 tours. Nous rapportons seulement les instances de 3 tours, puisque aucun solveur SAT à notre connaissance ne résout les instances de plus de 3 tours. Les instances d'origine contiennent une grande quantité de variables sans clauses, donc nous les compressons en effectuant la résolution unitaire et l'élimination des variables monotones. Le tableau 3 montre la performance des 5 procédures DP pour les instances après la simplification. Toutes les instances sont satisfiables.

TAB. 3 – *Temps d'exécution pour les instances DES*

Instance	#var	#clause	#eq_cls	Sato	Grasp	Relsat	Satz	*EqSatz*
cnf-r3-b1-k1.1	1461	8966	48	871	?	1080	> 7200	995
cnf-r3-b1-k1.2	1450	8891	48	> 7200	?	454	> 7200	1023
cnf-r3-b2-k1.1	2855	17857	96	3	170	18	946	1276
cnf-r3-b2-k1.2	2880	17960	96	> 7200	183	22	1468	629
cnf-r3-b3-k1.1	4255	26778	144	> 7200	96	37	32	11
cnf-r3-b3-k1.2	4418	27503	144	> 7200	113	44	101	11
cnf-r3-b4-k1.1	5679	35817	192	> 7200	77	45	357	17
cnf-r3-b4-k1.2	5721	35963	192	> 7200	67	48	> 7200	18

EqSatz est une des procédures les plus rapides pour résoudre ces instances et est substantiellement plus rapide que Satz, illustrant l'impact du raisonnement d'équivalence pour résoudre ces instances même quand elles ne contiennent que très peu de clauses d'équivalence.

5.3 Performance pour les instances BMC

Les problèmes BMC a été élaborées par Biere et al. [2] et appaîraissent dans la vérification bornée de modèle (Bounded Model Checking). Toutes les instances sont insatisfiables. Nous sélectionnons les instances barrel* et queueinvar* les plus difficiles et une moitié des instances longmult* représentatives.

Le tableau 4 montre la performance des 5 DPs sur les instances BMC sélectionnées. *EqSatz* est substantiellement plus rapide que les 4 autres DPs sur les instances barrel* et queueinvar*. Sur les instances barrel* contenant une partie EQ plus importante, *EqSatz*

TAB. 4 – *Les instances BMC*

Instance	#var	#clause	#eq_cls	Sato	Grasp	Relsat	Satz	*EqSatz*
barrel5	1407	5383	870	25	?	264	293	0
barrel6	2306	8931	1476	281	?	4428	2461	1
barrel7	3523	13765	2310	530	?	> 7200	57	2
barrel8	5106	20083	3408	726	?	> 7200	5	3
barrel9	8903	36606	6408	> 7200	?	> 7200	> 7200	7
longmult1	791	2335	29	0	0	0	0	0
longmult3	1555	4767	87	41	1	1	0	1
longmult5	2397	7431	145	181	64	27	11	13
longmult7	3319	10335	203	348	?	3402	331	274
longmult9	4321	13479	261	733	?	> 7200	1948	1681
longmult11	5403	16863	319	1110	?	> 7200	4371	3050
longmult13	6565	20487	377	1916	?	> 7200	6965	3662
longmult15	7807	24351	435	2646	?	> 7200	> 7200	4876
queueinvar10	886	5622	51	15	27	20	12	3
queueinvar12	1112	7335	53	97	52	31	54	5
queueinvar14	1370	9313	55	576	101	162	250	10
queueinvar16	1168	6496	75	1398	109	93	1017	9
queueinvar18	2081	17368	70	> 7200	?	257	8	9
queueinvar20	2435	29671	72	> 7200	?	834	13	14

trouve la contradiction par le raisonnement d'équivalence sans branchement. Sur les instances longmult*, seule Sato est légèrement plus rapide que *EqSatz*.

6 Discussion et Travaux Associés

Les clauses d'équivalence constituent un obstacle majeur à une procédure DP. Par exemple, alors que la complexité de Satz sur les instances aléatoires les plus difficiles parraît être $O(2^{n/21})$, sa complexité sur le problème DIMACS pret* est $O(2^{n/3})$. Cependant les clauses d'équivalence réalisent une structure en commun dans le codage en SAT de nombreux problèmes réels. Massacci [7] a remarqué dans son codage en SAT du problème de recherche de clé cryptographique que chaque tour est séparé du prochain tour par un ensemble de clauses d'équivalences. Il a remarqué aussi que le problème devient difficile pour les techniques IA actuelles dès que les clauses d'équivalence apparaîssent.

L'apprentissage et le retour en arrière intelligent sont deux techniques CSP efficaces pour de nombreux problèmes structurés. Cependant, elles paraîssent être moins efficaces que la technique spécialisée du raisonnement d'équivalence pour la structure très spéciale des clauses d'équivalence. Par exemple, bien que elles rendent Sato, Grasp et Relsat substantiellement plus rapides que Satz pour le problème DIMACS pret*, le raisonnement d'équivalence rend la complexité de *EqSatz* linéaire pour ce problème.

Pour les instances contenant à la fois une partie EQ et une partie CNF, le raisonnement d'équivalence joue son rôle même si la partie EQ est très petite. Par exemple, *EqSatz* est significativement plus rapide que Satz et concurrence les meilleurs DPs renforcées de techniques "regarder en arrière" pour les instances DES dont la partie EQ représente seulement 2% de la formule entière. Pour le problème de parité 32-bit de DI-

MACS et le problème BMC barrel* contenant une partie EQ plus importante, *EqSatz* est substantiellement plus rapide que les autres DPs.

Ni le raisonnement d'équivalence ni les techniques CSP "regarder en arrière" ne permettent à une DP de résoudre les instances DES de 4 tours de Massacci dans un temps raisonnable. Il apparaît que ces instances contiennent d'autres structures. Nous pensons qu'une combinaison des techniques CSP "regarder en arrière" et du raisonnement d'équivalence est prometteuse pour résoudre ces instances.

Warners et Van Maaren [11] ont proposé une approche pour résoudre les instances qui ne contiennent que la partie EQ ou dont la partie EQ est beaucoup plus grande que la partie CNF. Cette approche a donné la première méthode capable de résoudre les 5 instances par32-i-c du challenge du problème de parité DIMACS dont la partie EQ représente plus de 80% de la formule entière. L'originalité de leur approche est l'élimination de clauses d'équivalence.

Etant donné un ensemble de clauses d'équivalence, Warners et Van Maaren sélectionnent une clause d'équivalence de longueur k, $x_1 \leftrightarrow l_2 \leftrightarrow ... \leftrightarrow l_k$, la réécrivent comme $x_1 \equiv l_2 \leftrightarrow ... \leftrightarrow l_k$ et remplacent dans toutes les autres clauses d'équivalence l'occurrence de $(\bar{x}_1)$ x_1 par $(\neg)l_2 \leftrightarrow ... \leftrightarrow l_k$, ce qui incrémente en général la longueur de ces clauses (par $k-2$ dans le pire de cas). x_1 est appelée *variable dépendante.*

Pour une formule qui ne contient pas de partie CNF comme les DIMACS pret* et dubois*, la clause sélectionnée peut être satisfaite facilement puisque sa variable dépendante n'apparaît pas ailleurs après le remplacement, donc elle peut être supprimée de $\mathcal{F}$. En supprimant successivement les clauses d'équivalence, $\mathcal{F}$ est résolue en temps polynomial. Pour les problèmes contenant à la fois la partie EQ et la partie CNF, le remplacement de la variable dépendante est limitée dans la partie EQ et la clause d'équivalence sélectionnée ne peut pas être supprimée si sa variable dépendante apparaît dans la partie CNF. Sinon la clause d'équivalence sélectionnée est supprimée.

L'algorithme de Warners et Van Maaren est divisé en deux phases pour résoudre une formule contenant à la fois la partie CNF et la partie EQ. Dans la première phase, l'opération présentée ci-dessus est appliquée à la partie EQ pour éliminer le maximum de clauses d'équivalence. Par exemple, les 5 instances par32-i-c contiennent plus de 1085 clauses d'équivalence mais il reste seulement 218 clauses d'équivalence après la première phase. (c'est à dire, il y a 218 variables dépendantes qui apparaîssent dans la partie CNF). Dans la deuxième phase, une procédure DP adaptée est exécutée sur la partie CNF et les 218 clauses d'équivalence. Donc la première phase peut être considérée comme un pré-traitement dans la racine d'un arbre de recherche, ce qui est à comparer avec le raisonnement d'équivalence de *EqSatz* à *chaque* noeud d'un arbre de recherche.

L'algorithme de Warners et Van Maaren est actuellement substantiellement plus rapide que *EqSatz* pour les 5 instances par32-i-c qui sont résolues en moins de 50 secondes [12]. L'efficacité semble être liée au fait qu'il y a peu de clauses d'équivalence (c'est à dire 218) après la première phase. Cependant quand la partie CNF est plus grande et contient plus de variables dépendantes, moins de clauses d'équivalence peuvent être supprimées. Dans ce cas, *EqSatz* est plus rapide que l'algorithme de Warners et Van Maaren. En fait, les 5 instances par32-i, les instances DES et les instances BMC sont dans ce cas. Par exemple, pour l'instance par32-1 qui contient 1548 clauses d'équivalence et 4085 autres clauses CNF, 1048 (plus longues) clauses d'équivalence restent après la première phase [12]. A notre connaissance, *EqSatz* est la seule procédure capable de résoudre toutes les 5 instances par32-i dans un temps raisonnable.

7 Conclusion

Les clauses d'équivalence représentent une structure commune dans le codage en SAT de nombeux problèmes réels. Ils constituent un obstacle majeur pour les procédures DP. Nous avons intégré un raisonnement d'équivalence dans Satz pour résoudre des problèmes contenant des clauses d'équivalence. Le raisonnement d'équivalence permet de remédier à l'inefficacité de la propagation unitaire sur les clauses d'équivalence. Intégré dans la règle de branchement, il la rend plus précise et plus puissante pour maximiser la réduction de l'espace de recherche. Enfin, les nouvelles clauses d'équivalence ajoutées dans $\mathcal{F}$ par le raisonnement d'équivalence réduit aussi considérablement l'espace de recherche.

Notre approche rend une procédure DP capable de résoudre un des 10 challenges formulés par Selman et al. et beaucoup d'autres problèmes réels. Les résultats expérimentaux suggèrent que le raisonnement d'équivalence est plus efficace que les techniques CSP générales "regarder en arrière" pour les structures spéciales des clauses d'équivalence.

8 Remerciements

Nous remercions Daniel Le Berre pour nous avoir informés les problèmes BMC, Patrice Seebold pour ses commentaires qui ont aidé à améliorer ce papier. Nous remercions également Sylvain Gérard pour l'amélioration du français de ce papier.

Références

[1] Bayardo Jr. R.J., Schrag R.C., Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In proceedings of AAAI-97, Providence, Rhode Island, July 1997.

[2] Biere A., Cimatti A., Clarke E., Zhu Y., Symbolic Model Checking without BDDs. In proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS. Springer-Verlag, 1999.

[3] Cook S.A., The Complexity of Theorem Proving Procedures. In *3rd ACM Symp. on Theory of Computing*, pages 151-158, Ohio, 1971.

[4] Crawford J.M., Kearns M.J., Schapire R.E., The Minimal Disagreement parity problem as a hard satisfiability problem. Draft version 1995.

[5] Davis M., Logemann G., Loveland D., A Machine Program for Theorem Proving. In Common. ACM 5, 1962, pp. 394-397.

[6] Li C.M., Anbulagan, Heuristics Based on Unit Propagation for Satisfiability Problems. In Proceedings of IJCAI-97, ISBN 1-55860-480-4, Page 366-371, Nagoya, Japan, August 1997.

[7] Massacci F., Using Walk-SAT and Rel-SAT for Cryptographic Key Search. In Proceedings of IJCAI-99, Page 290-295.

[8] Selman B., Kautz H., McAllester D., Ten challenges in propositional reasoning and search. In Proc. of IJCAI-97, ISBN 1-55860-480-4, Nagoya, Japan, August 1997.

[9] Silva J. P. M., Sakallah K. A., Conflict Analysis in Search Algorithms for Propositional Satisfiability. In Proc. of the Int. Conf. on Tools with Artificial Intelligence, November 1996.

[10] Urquhart A., Hard examples for resolution. Journal of the ACM, 34(1):209-219, January 1987.

[11] Warners J.P., Van Maaren H., A two phase algorithm for solving a class of hard satisfiability problems. Operations research letters 23 (1998) 81-88.

[12] Warners J.P., Personal communication, June 1999.

[13] Zhang H., An efficient propositional prover. In Proc. of int' Conf. on Automated Deduction, pp272-275, July 1997.