

# Importance de la sémantique dans le codage CNF de contraintes de cardinalité : application au diagnostic de SED

---

Anbulagan

Alban Grastien

NICTA et Australian National University  
Canberra, Australia

{anbulagan|alban.grastien}@nicta.com.au

## Résumé

Le codage d'un problème a un impact énorme sur le temps de calcul en SAT : un solveur SAT peut résoudre très facilement un problème codé d'une certaine manière, et éprouver des difficultés pour le même problème codé d'une autre manière. Pire, un codage peut favoriser le temps de calcul d'un solveur, et se montrer au contraire inadapté pour un autre solveur. Dans cet article, nous nous concentrons sur l'expression en CNF de contraintes de cardinalité : étant donné un ensemble  $V$  de variables propositionnelles, étant donnée une valeur entière  $k$ , exactement  $k$  variables de  $V$  doivent être évaluées à *vrai*. Nous proposons de nouvelles manières de modéliser ces contraintes en cherchant à grouper les variables dont la sémantique est proche. Nous étudions ces codages sur des problèmes de diagnostic de système à événements discrets (SED). Des problèmes jusqu'ici insolubles peuvent être à présent résolus à la fois par des procédures systématiques ou stochastiques. Les résultats mettent également en lumière l'existence d'un codage qui convienne aux deux types d'algorithme.

## 1 Introduction

Grâce aux progrès réalisés au cours de la dernière décennie, les algorithmes de satisfiabilité propositionnelle (SAT) peuvent être utilisés dans un nombre croissant d'applications, que ce soit le diagnostic, la planification, l'ordonnancement des tâches, la vérification automatique tant au niveau circuit intégré que logiciel, et bien d'autres. Un problème dans un de ces domaines peut être résolu par un algorithme dont l'une des composantes est un solveur SAT. Dans ce papier, nous nous intéressons au problème de diagnostic de système à événements discrets (SED).

Le diagnostic de SED consiste à déterminer les possibles évolutions d'un système partiellement observable pour déterminer tout dysfonctionnement et, dans ce cas, isoler et identifier le ou les problèmes. Nous avons proposé l'utilisation d'algorithmes SAT pour résoudre ce type de problèmes [4, 5]. Nos résultats ont montré que SAT offrait des performances bien supérieures aux algorithmes classiques de diagnostic. Cependant, près de 30% des problèmes proposés n'ont pas pu être résolus, en particulier les problèmes utilisant des observations partiellement ordonnées. Des investigations supplémentaires ont démontré que la modélisation des contraintes de cardinalité utilisée dans cette étude pouvait être fortement améliorée, ce qui a motivé le présent travail.

Dans cet article, nous proposons donc différentes manières d'encoder la contrainte de cardinalité, en cherchant en particulier à grouper les variables dont la sémantique est proche. Nous étudions ces codages sur le problème de diagnostic de SED. Nos expérimentations montrent un net progrès à la fois pour les algorithmes SAT de types systématique et stochastique. Les résultats mettent également en lumière l'existence d'un codage qui convient aux deux types d'algorithme.

Ce papier est découpé comme suit. Nous donnons d'abord une présentation bibliographique de la contrainte de cardinalité. Ensuite, nous présentons rapidement le problème de diagnostic de SED. Puis, nous proposons différentes manières d'encoder la contrainte de cardinalité. Nous évaluons l'efficacité de ces codages en section 5 et proposons une explication des résultats.

## 2 Contrainte de cardinalité en SAT

La contrainte de cardinalité est un cas particulier du problème de contraintes pseudo-booléennes dont les meilleurs solveurs utilisent SAT [3]. Étant donné un ensemble  $V$  de  $n$  variables propositionnelles, la contrainte indique que exactement  $k$  variables doivent être évaluées à *vrai*.

Les deux règles qui suivent permettent d'encoder cette contrainte sous forme de clauses sans utiliser de variable auxiliaire :

1. Pour tout sous-ensemble  $\{v_1, \dots, v_{k+1}\} \subseteq V$  de taille  $k+1$ , au moins une variable doit être évaluée à *faux* :  $\neg v_1 \vee \dots \vee \neg v_{k+1}$ .
2. Pour tout sous-ensemble  $\{v_1, \dots, v_{n-k+1}\} \subseteq V$  de taille  $n-k+1$ , au moins une variable doit être évaluée à *vrai* :  $v_1 \vee \dots \vee v_{n-k+1}$ .

Cette représentation de la contrainte ne nécessite pas de variable propositionnelle supplémentaire mais requiert cependant  $\frac{n!}{(k+1)! \times (n-k-1)!} + \frac{n!}{(k-1)! \times (n-k+1)!}$  clauses. Ce codage ne peut donc être utilisé que pour  $k = 1$ . Ce cas particulier est fréquent puisqu'il permet de représenter par exemple le fait qu'un composant ne peut être que dans un seul état à un instant donné. Des travaux ont été conduits pour ce problème, en particulier ceux de Marques-Silva et Lynce [6] qui tendent à prouver qu'il existe de meilleurs codages que les deux règles si dessus. Pour notre part, nous avons besoin de spécifier cette contrainte pour des valeurs différentes de  $k$ , et étudions donc d'autres modélisations.

Nous nous intéressons donc à la méthode introduite par Bailleux et Boufkhad [9] et étudiée par la suite par Sinz [11] qui utilise un *totaliseur*<sup>1</sup>. Le totaliseur est un arbre dont les feuilles sont étiquetées par les variables propositionnelles de  $V$ . Chaque nœud  $a$  de l'arbre est étiqueté par  $n(a)$  variables auxiliaires, notées  $a_1, \dots, a_{n(a)}$ , qui représentent le nombre de variables affectées à *vrai* aux feuilles du sous-arbre de  $a$ . Les variables à la racine représentent donc le nombre de variables de  $V$  assignées à *vrai*. Il est alors facile de contraindre ces variables pour satisfaire la contrainte de cardinalité.

Pour construire un totaliseur, il faut savoir comment encoder la valeur associée à chaque nœud et comment spécifier que la valeur à chaque nœud égale la somme des valeurs à chacun de ses deux fils. La littérature scientifique propose deux codages pour représenter un nombre entre 0 et  $K$  et l'addition.

**Codage binaire** C'est le codage utilisé par exemple dans [12]. Le nombre associé à ce nœud est  $\sum_{j \in \{1, \dots, n(a)\}} (v(a_j) \times 2^{j-1})$  où  $v(a_j) = 1$  si  $a_j$  est

affecté à *vrai* ou  $v(a_j) = 0$  sinon. Ce codage ne nécessite que  $\lfloor \log_2(K) \rfloor$  variables.

La contrainte  $a + b = c$  utilise  $\lfloor \log_2(K) \rfloor - 1$  variables auxiliaires  $z_j$  qui représentent la retenue dans le calcul de  $c_{j-1}$ , et est modélisée par un nombre en  $\log_2(K)$  de clauses qui encodent les formules suivantes :  $z_j = (a_{j-1} \wedge b_{j-1}) \vee (z_{j-1} \wedge (a_{j-1} \vee b_{j-1}))$  et  $c_j = a_j \oplus b_j \oplus z_j$ .

**Codage unaire** Ce codage a été proposé par Bailleux et Boufkhad [9]. La valeur associée à  $a$  est supérieure à  $p$  ssi  $p \leq n(a)$  et  $a_p$  est affectée à *vrai*. Ce codage requiert  $K$  variables.

La contrainte  $a + b = c$  est représentée par les propriétés suivantes :  $(a \geq p) \wedge (b \geq q) \Rightarrow (c \geq p + q)$  et  $(a < p + 1) \wedge (b < q + 1) \Rightarrow (c < p + q + 1)$ . Ce codage de l'addition ne nécessite pas de variable auxiliaire supplémentaire mais  $K^2$  clauses de taille 3 plus  $K^2$  clauses de taille 2. Malgré une représentation moins compacte que le codage binaire, le codage unaire s'est montré plus efficace dans les expérimentations.

## 3 Diagnostic par SAT

Notre étude s'appuie sur le problème de diagnostic de systèmes à événements discrets (SED) [8]. Nous présentons succinctement ce problème et montrons comment il est lié au codage de contraintes de cardinalité en SAT. Plus de détails sur l'utilisation d'algorithmes SAT pour le diagnostic de SED peuvent être trouvés dans [4].

Considérons un système qui peut être modélisé par un SED, c'est-à-dire un automate fini dont les transitions sont étiquetées par les événements associés au changement d'état. Cet automate est représenté de manière symbolique : chaque état est modélisé par une affectation des *variables d'états*, et les transitions sont régies par des *règles* qui précisent (i) quelle précondition les variables de l'état doivent satisfaire pour permettre la transition, (ii) quels sont les effets de la transition sur l'affectation des variables d'états, et (iii) quels événements sont associés à la transition. Un comportement du système est représenté par une séquence d'états et de transitions sur le SED, appelée *trajectoire*.

Le *superviseur* cherche à surveiller le système pour détecter l'occurrence de fautes, qui sont des événements particuliers, et les identifier le cas échéant. Certains événements sont *observables*, ce qui signifie que leur occurrence correspond à l'émission d'une observation, par exemple une alarme. La tâche du superviseur consiste à comparer le modèle et la séquence d'observations reçues pour retrouver ce qui s'est passé dans le système et quelles fautes ont eu lieu. Généralement, le système est seulement très partiellement

<sup>1</sup>Les auteurs de cet article sont preneurs de toute meilleure traduction de *totalizer*.

observable, et plusieurs trajectoires sont possibles. Le superviseur peut alors décider de ne considérer que la trajectoire la plus probable, par exemple celle comportant le plus faible nombre de fautes. C'est cette trajectoire que nous essayons de retrouver.

Dans [4], nous avons proposé un algorithme pour résoudre des problèmes de diagnostic de SED en utilisant SAT. Le principe est le suivant. On considère que la taille maximale de la trajectoire  $n$  est connue. C'est généralement possible directement grâce au nombre d'observations reçues par le superviseur. On crée alors les variables propositionnelles  $s^i$  (resp.  $e^i$  et  $r^i$ ) qui représentent l'affectation  $s = vrai$  de la variable d'état  $s$  (resp. l'occurrence de l'événement  $e$  et le tirage de la règle  $r$ ) au pas de temps  $i$  de la trajectoire.

Considérons à présent la liste  $\{f_1, \dots, f_m\}$  d'événements de faute du système. Notre approche consiste à chercher une trajectoire (1) sur le SED (2) cohérente avec les observations et (3) qui minimise le nombre d'occurrences d'événements de faute. Il faut donc trouver une affectation aux variables  $s^i$ ,  $e^i$  et  $r^i$  présentées précédemment de manière à satisfaire les trois contraintes précédentes :

1. L'affectation des variables doit correspondre à une trajectoire sur le SED. On introduit donc un ensemble de clauses noté  $Mod(0, n)$  qui assure que soient respectées les contraintes entre la valuation des variables d'état et l'occurrence des événements, et ce au travers des règles tirées.
2. La trajectoire doit être compatible avec les observations. Est donc défini un ensemble de clauses noté  $Obs(0, n)$  qui est satisfiable étant donné une valuation des variables  $s^i$ ,  $e^i$  et  $r^i$  si et seulement si les observations générées par la trajectoire sont compatibles avec les observations effectivement reçues. Ces clauses peuvent comporter des variables auxiliaires si les observations sont incertaines.
3. Enfin, la trajectoire doit minimiser le nombre de fautes. Pour ce faire, nous définissons un ensemble de clauses noté  $Que_k$  qui est satisfiable si et seulement si exactement  $k$  variables sont affectées à *vrai* parmi  $\mathcal{F} = \{f_1, \dots, f_m\} \times \{1, \dots, n\}$  où  $\langle f_i, j \rangle = f_i^j$ .

L'algorithme cherche d'abord une solution pour le problème SAT  $Mod(0, n) \cup Obs(0, n) \cup Que_0$ , c'est-à-dire une trajectoire sans faute. Si une telle trajectoire existe, le problème est satisfiable et la solution correspond à la trajectoire recherchée. Si une telle trajectoire n'existe pas, le problème n'est pas satisfiable. L'algorithme lance alors une nouvelle recherche pour le problème  $Mod(0, n) \cup Obs(0, n) \cup Que_k$  avec une valeur de  $k$  qui augmente jusqu'à ce qu'une solution soit trouvée.

C'est l'expression de la contrainte  $Que_k$  qui nous intéresse dans ce papier.

## 4 Codage de contrainte de cardinalité

Une étude préliminaire de la modélisation de contrainte de cardinalité nous a amené à la conclusion que deux paramètres orthogonaux gouvernent la construction du totaliseur. Le premier paramètre, que nous choisissons de noter  $\alpha$ , correspond à la manière d'encoder le nombre associé à chaque nœud. Le second paramètre n'a jusqu'ici et à notre connaissance jamais été clairement identifié et concerne l'ordre dans lequel les additions sont effectuées.<sup>2</sup> Ainsi, étant données les quatre variables  $a$ ,  $b$ ,  $c$  et  $d$ , est-il préférable de spécifier  $(a + b) + (c + d) = k$  ou  $((a + c) + d) + b = k$ ? Ce paramètre est noté  $\beta$ .

Nous définissons 23 codages de la contrainte de cardinalité. Parmi ces codages, 21 d'entre eux sont des combinaisons de 3 codage de nœuds ( $B$ ,  $U$  et  $A$ ) et 7 ordres d'addition ( $R$ ,  $CF$ ,  $CI$ ,  $CG$ ,  $TF$ ,  $TI$  and  $TG$ ). Nous considérons également deux encodages hybrides ( $UCTI$  and  $UACG$ ). Ces 23 encodages sont résumés dans la Table 1 et détaillés dans cette section.

$\beta \rightarrow$	$R$	$C$			$T$		
$\alpha \downarrow$		$F$	$I$	$G$	$F$	$I$	$G$
$B$	$BR$	$BCF$	$BCI$	$BCG$	$BTf$	$BTI$	$BTG$
$U$	$UR$	$UCF$	$UCI$	$UCG$	$UTf$	$UTI$	$UTG$
$A$	$AR$	$ACF$	$ACI$	$ACG$	$ATf$	$ATI$	$ATG$
hybride		$UCI \cup UTI$ ( $UCTI$ )			$UCG \cup ACG$ ( $UACG$ )		

TAB. 1 – Liste des codages de contrainte de cardinalité.

### 4.1 Codage des nœuds

Nous reprenons les codages binaire (noté  $B$ ) et unaire (noté  $U$ ) présentés en section 2. Nous ajoutons par ailleurs un nouveau codage unaire différent de  $U$  par la manière de représenter la contrainte  $a + b = c$ .

Ce codage se base sur l'interprétation suivante du codage unaire : chaque variable auxiliaire codant le nœud  $b$  correspond à un nombre 1 si la variable est évaluée à *vrai* et 0 sinon, et la valuation de  $b$  est la somme de ces nombres. Alors, plutôt que d'ajouter  $b$  à  $a$  en une seule fois, chaque nombre  $b_j$  est ajouté de manière séquentielle, les propriétés sur les valuations de  $a_i$  et  $b_j$  permettant de simplifier les formules (voir Figure 1). Plus formellement, on définit  $c^j$  comme l'addition de  $a$  avec les  $j$  premières variables de  $b$ . Clairement,  $c^0 = a$ . L'addition est représentées par :  $c_i^j \leftrightarrow c_i^{j-1} \vee (c_{i-1}^{j-1} \wedge b_j)$ . Remarquons que  $b_i \rightarrow b_{i-1}$

<sup>2</sup>La seule mention est la distinction entre totaliseurs séquentiels et parallèles [11].

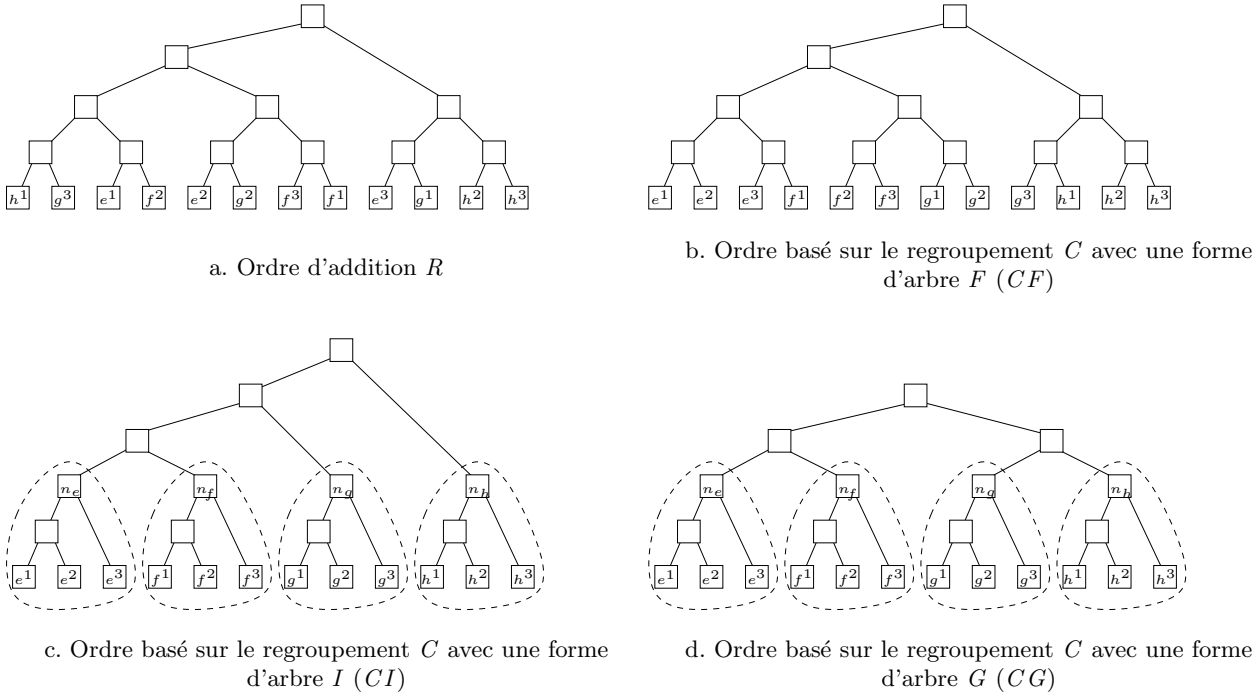


FIG. 2 – Exemple d'ordres d'addition.

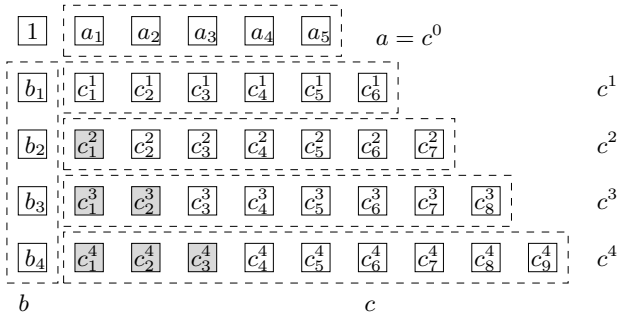


FIG. 1 – Principe de l'addition pour le codage  $A$ .

et donc  $c_i^j = c_i^i$  pour  $j > i$ ; les variables représentées en gris sur la figure sont donc redondantes et peuvent être supprimées. Pour trois nombres  $a$ ,  $b$  et  $c$  de même valeur maximale  $K$ , ce codage requiert  $\frac{K \times (K-1)}{2}$  variables auxiliaires,  $K^2$  clauses de taille 3 et  $K^2$  clauses de taille 2. Nous notons ce codage  $A$ .

Le codage de contrainte devrait satisfaire autant que possible la propriété d'efficacité connue sous le nom de consistance d'arc généralisée. Cette propriété implique que pour chaque variable et chacune des assignations autorisées de la variable, il existe un modèle tel que cette assignation est associée à la variable. En d'autres termes, la propagation unitaire doit assigner toutes les variables qui peuvent l'être au sein des clauses de la contrainte de cardinalité. Le codage  $U$  satisfait cette

propriété, mais pas le codage  $B$ .

La preuve de cette propriété pour  $A$  repose sur la preuve utilisée en [9]; il suffit alors de prouver que les lemmes 21 et 22 de cet article sont satisfaits. De même, il suffit de prouver ces résultats localement sur un nœud de l'arbre. Nous considérons ici  $a$ ,  $b$  et  $c$  trois nombres encodés avec respectivement  $K_a$ ,  $K_b$  et  $K_a + K_b$  variables, et la contrainte  $a + b = c$  encodée par  $A$ .

*Propagation en avant* : soit  $p_1 + p_2 = p \leq K_a$ ,  $q_1 + q_2 = q \leq K_b$ . Si  $a_{p_1}$  et  $b_{q_1}$  sont assignés à *vrai* et  $a_{K_a - p_2}$  et  $b_{K_b - q_2}$  sont assignés à *faux*, alors on peut montrer facilement que :

- pour tout  $i \in \{0, \dots, q_1\}$ ,  $c_{p_1+i}^i$  est assigné à *vrai* par propagation,
- pour tout  $i \in \{q_1, \dots, K_b\}$ ,  $c_{p_1+q_1}^i$  est assigné à *vrai* par propagation,
- pour tout  $i \in \{0, \dots, K_b - q_2\}$ ,  $c_{K_a - p_2+i}^i$  est assigné à *faux* par propagation, et
- pour tout  $i \in \{q_1, \dots, K_b\}$ ,  $c_{K_a + K_b - p_2 - q_2}^i$  est assigné à *faux* par propagation,

ce qui satisfait le lemme 21.

*Propagation en arrière* : ajoutons maintenant qu'il existe une valeur  $\gamma \in \{\text{vrai}, \text{faux}\}$  et pour tout  $l \in \{p_1 + q_1 + 1, \dots, K_a + K_b - p_2 - q_2 - 1\}$ , la valeur de  $c_l$  est assignée à  $\gamma$ . On peut montrer facilement que :

- pour tout  $i \in \{K_b - q_2, \dots, K_b\}$ , pour tout  $j \in \{p_1 + q_1 + 1, \dots, K_a + K_b - p_2 - q_2 - 1\}$ ,  $c_j^i$  est assignée à  $\gamma$  par propagation,
- pour tout  $i \in \{q_1, \dots, K_b - q_2\}$ , pour tout  $j \in$

$\{p_1 + q_1 + 1, \dots, K_a + i - p_2 - 1\}$ ,  $c_j^i$  est assignée à  $\gamma$  par propagation,

- pour tout  $i \in \{q_1, \dots, K_b - q_2\}$ ,  $b_i$  est assignée à  $\gamma$  par propagation,
- pour tout  $i \in \{1, \dots, q_1\}$ , pour tout  $j \in \{p_1 + i + 1, \dots, K_a + i - p_2 - 1\}$ ,  $c_j^i$  est assignée à  $\gamma$  par propagation, et
- pour tout  $j \in \{p_1 + j + 1, \dots, K_a - p_2 - 1\}$ ,  $a_j$  est assignée à  $\gamma$  par propagation,

ce qui satisfait le lemme 22.

## 4.2 Ordre de l'addition

Nous proposons 7 ordres d'addition. Certains ordres sont représentés sur la Figure 2 pour un problème de diagnostic avec quatre événements de faute  $e$ ,  $f$ ,  $g$  et  $h$ , et trois dates 1 à 3. Les autres ordres peuvent se déduire facilement.

Le premier ordre consiste tout simplement à construire un arbre binaire équilibré, et placer les variables de manière aléatoire au niveau des feuilles. Cet ordre est noté  $R$  et représenté sur la Figure 2a. Les six ordres d'addition suivants sont définis par combinaison d'une forme d'arbre (trois formes proposées) et d'un placement de variable sur les feuilles (deux placements proposés).

Contrairement à  $R$  qui utilisait un placement aléatoire, les deux placements que nous proposons visent à regrouper les variables de sémantique proche. Dans le problème de diagnostic de SED, on peut choisir de regrouper les variables qui correspondent au même composant ( $C$ , utilisé dans les figures 2b à 2d) ou à la même date ( $T$ ).

Nous considérons trois formes d'arbres :

- F** Nous générons un arbre équilibré dont les feuilles sont attribuées conformément à l'ordre de placement choisi (Figure 2b).
- I** Nous générons un sous-arbre pour chaque groupe de variables. L'addition est ensuite effectuée de manière incrémentale, ce qui génère un *peigne* dont les feuilles sont les sous-arbres (Figure 2c).
- G** Nous générons un sous-arbre pour chaque groupe de variables. Nous générons ensuite un arbre équilibré dont les feuilles sont les sous-arbres (Figure 2d).

## 4.3 Modélisations hybrides

Pour un (sous-)problème et un solveur donnés, le codage  $Que_k^i$  d'une contrainte de cardinalité peut être plus facile à résoudre qu'un autre codage  $Que_k^j$ . La tendance peut s'inverser pour un autre (sous-)problème. Nous proposons donc d'étudier des codages hybrides utilisant deux codages équivalents de

la même contrainte de cardinalité :  $Que_k^1 \cup Que_k^2$ . L'idée est que si le solveur est capable de déterminer quel codage rend le problème plus facile pour un (sous-)problème donné, il pourra bénéficier des avantages de chacun des codages et trouver une solution plus facilement. Dans ce papier, nous considérons les combinaisons  $UCI \cup UTI$  (noté  $UCTI$ ) et  $UCG \cup ACG$  (noté  $UACG$ ).

## 5 Validation empirique et analyse

### 5.1 Génération des instances CNF

Nous évaluons les 23 codages présentés dans la Table 1 sur les problèmes de diagnostic de SED déjà étudiés dans [4]. Nous avons choisi les problèmes les plus difficiles, à savoir :

- problèmes satisfiables : daté-dur-s, total-moyen-s, total-dur-s, partiel-moyen-s, partiel-dur-s ;
- problèmes non satisfiables : daté-moyen-u, daté-dur-u, total-facile-u, total-moyen-u, total-dur-u, partiel-moyen-u, partiel-dur-u.

Voici une description rapide des différences entre ces problèmes. Le premier paramètre (daté, total, partiel) indique le type d'observations disponibles pour le diagnostic. Les problèmes de type daté et total ont uniquement des clauses unitaires dans  $Obs(0, n)$ , ce qui rend le problème plus facile. De plus, la valeur  $n$  du nombre de pas de temps est mieux estimée dans les problèmes de type daté, ce qui réduit encore la complexité. Le deuxième paramètre (facile, moyen, dur) concerne la trajectoire à diagnostiquer, et est donc utilisé lors de la génération de la trajectoire. Pour les problèmes de type dur, les observations sont plus ambiguës et la trajectoire est plus difficile à retrouver.

Chaque problème comporte 20 instances correspondant à un nombre de fautes entre 1 et 20. Chaque problème satisfiable contient donc une contrainte  $Que_k$  et chaque problème non satisfiable contient une contrainte  $Que_{k-1}$  avec  $k \in \{1, \dots, 20\}$ .

Par ailleurs, nous proposons une modification dans le codage utilisé pour représenter le SED. Cette extension se base sur l'opération de *hyper-resolution* et apparaît quand toutes les règles associées à un événement particulier ont un même effet. Formellement, on a les clauses suivantes :  $\neg e \vee r_1 \vee \dots \vee r_k$  et  $\forall i \in \{1, \dots, k\}$ ,  $\neg r_i \vee a$ , ce qui implique  $\neg e \vee a$ . Cette amélioration a un coût de calcul très faible puisque déterminé directement depuis le SED, et génère pour notre exemple une augmentation du nombre de clauses d'environ 1% seulement. Nous la notons H.

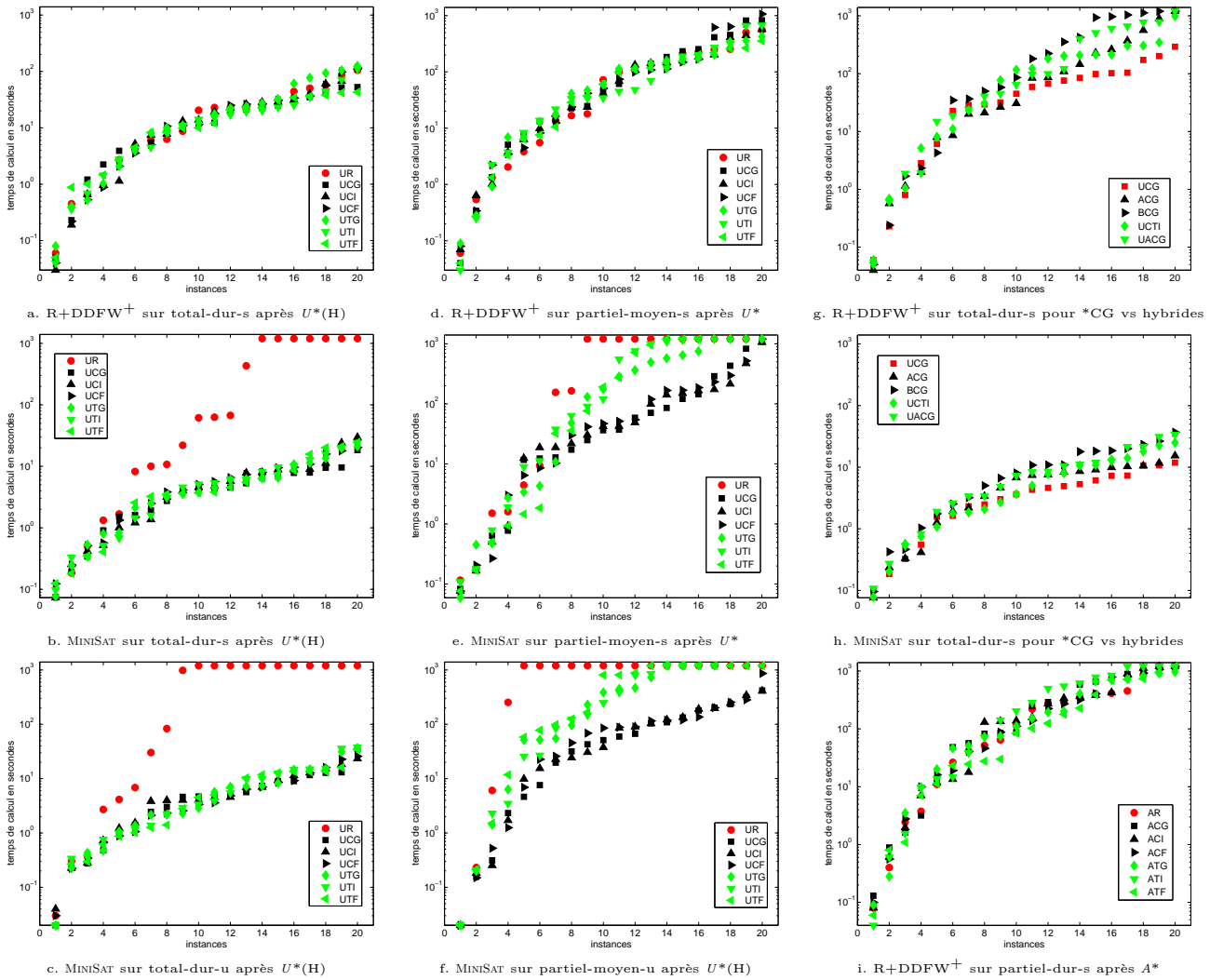


FIG. 3 – Performances des solveurs SAT pour certains problèmes de diagnostic de SED

## 5.2 Sélection des solveurs SAT

Nous avons sélectionné le solveur R+DDFW<sup>+</sup> [7] et le solveur MINISAT v2 [2] pour représenter respectivement la recherche locale stochastique (SLS) et la recherche systématique basée sur l’algorithme DPLL [1].<sup>3</sup> Le but est de comparer les résultats des deux types d’algorithmes pour déterminer s’ils réagissent de la même manière aux codages proposés.

DDFW<sup>+</sup> est un algorithme par pondération de clauses, qui adapte le poids des clauses en fonction du degré de stagnation de la recherche. La version R+DDFW<sup>+</sup> du solveur est une amélioration qui intègre un prétraitement par calcul de résolvantes. Ce prétraitement génère des clauses supplémentaires de

<sup>3</sup>Le solveur DDFW<sup>+</sup> peut être récupéré depuis son développeur, et MINISAT v2 est disponible à cette adresse : <http://minisat.se/MiniSat.html>.

taille inférieure ou égale à 3 dans la formule d’origine, avant d’appliquer les règles de propagation unitaire. Ce solveur a été choisi pour ses excellents résultats dans [7], où il surpasse tous les autres solveurs SLS. D’ailleurs, nos expérimentations initiales ont très rapidement montré une claire amélioration par rapport au solveur R+RSAPS utilisé dans [4] qui par exemple ne pouvait résoudre que 8 des 20 problèmes de total-dur-s.

Les solveurs de type DPLL sont connus pour leur capacité à résoudre des problèmes industriels qui possèdent un grand nombre de clauses et de variables et des structures difficiles. MINISAT est un des meilleurs algorithmes de cette catégorie. Aussi, nous utilisons MINISAT v2 avec simplification par élimination de variable, qui nous est apparu le plus efficace dans nos premiers tests.

### 5.3 Résultats

Les expérimentations comportent la résolution de 15 640 problèmes et ont donc été effectuées sur un cluster composé de 8 Intel Duo Core 2.4GHz avec 4 Go de mémoire RAM. Nous avons alloué 1 200 secondes pour chaque processus.

La Table 2 présente le résumé des performances des solveurs SAT pour les différents codages de la contrainte de cardinalité et avec (temps(H)) ou sans (temps()) l'utilisation de l'hyper-résolution, pour les problèmes satisfiables et non satisfiables. Chaque nombre représente le temps de calcul total pour 100 instances satisfiables ou 140 instances non satisfiables. Le temps de calcul est limité à 1 200 secondes. Par la suite, nous présentons des résultats plus précis sur la Figure 3 et les Tableaux 3,4. Sur les Tableaux 2 à 4, les nombres indiqués entre parenthèses représentent les instances non résolues en 1 200 secondes. Remarquons qu'à cause de cette limite, les temps de calculs pour les moins bonnes modélisations sont écrasés. Nous examinons à présent les résultats.

#### 5.3.1 Du point de vue du codage

**Le codage des nœuds** La Table 2 corrobore les résultats de [9], à savoir que, bien que nécessitant plus de variables intermédiaires, les problèmes utilisant le codage unaire ( $U^*$  and  $A^*$ ) sont résolus plus rapidement que les problèmes codés en binaire ( $B^*$ ), y compris pour les problèmes non satisfiables. Le solveur SLS semble être le plus sensible, puisque le temps de calcul pour les problèmes  $U^*$  est environ un tiers du temps nécessaire pour  $B^*$ .

Le codage de l'addition  $A$  proposé dans ce papier n'améliore pas en moyenne les performances par rapport à  $U$ ; les temps de calcul sont mêmes dégradés de 30% pour R+DDFW<sup>+</sup>. Cependant, la Figure 3i montre que le codage  $A$  permet de résoudre toutes les instances du problème partiel-dur-s qui est le plus difficile des problèmes satisfiables de notre expérimentation pour certains ordre des additions et par le solveur R+DDFW<sup>+</sup>.

**L'ordre des additions** On s'intéresse d'abord aux résultats de MINISAT. Les résultats de la Table 2 et de la Figure 3 sont sans ambiguïté : utiliser un ordre aléatoire ( $*R$ ) pour placer les variables dans le totaliseur rend le problème beaucoup plus difficile à résoudre. Les Figures 3b et 3e confirment que le temps de calcul de MINISAT est amélioré de deux ordres de magnitude si on choisit n'importe quel autre ordre.<sup>4</sup> Une analyse plus approfondie des informations fournies par

<sup>4</sup>Ce résultat n'apparaît pas dans la Table 2 à cause de la limite de temps de 1 200 secondes.

Solveur	Codage	Temps()	Temps(H)
R+DDFW <sup>+</sup> sur 5 problèmes satisfiables avec 100 instances	BR	(17) 39 471	(15) 25 435
	BCG	(18) 37 881	<b>(13) 23 805</b>
	BCI	(19) 40 638	(17) 26 660
	BCF	(16) 38 538	(16) 25 943
	BTG	(19) 37 174	(16) 26 355
	BTI	(32) 54 793	(18) 35 024
	BTF	(21) 38 722	(13) 24 751
	Total B	(142) 287 217	<b>(108) 187 973</b>
	UR	(3) 14 144	<b>7 151</b>
	UCG	(2) 12 782	(1) 9 380
	UCI	(1) 14 139	(3) 11 016
	UCF	(2) 14 970	(1) 9 861
	UTG	11 050	(2) 10 245
	UTI	(2) 15 267	(1) 10 453
	UTF	(1) 10 567	(1) 9 728
	Total U	(11) 92 919	<b>(9) 67 834</b>
	AR	16 973	(2) 12 399
	ACG	(3) 16 844	<b>12 257</b>
	ACI	(2) 16 952	(2) 13 204
	ACF	(1) 16 022	(1) 13 150
ATG	15 427	(3) 12 958	
ATI	(5) 23 903	(5) 16 691	
ATF	(2) 15 060	(1) 11 888	
Total A	(13) 121 181	<b>(14) 92 547</b>	
UCTI	(2) 15 518	<b>10 151</b>	
UACG	(6) 24 142	<b>(4) 16 705</b>	
MINISAT sur 5 problèmes satisfiables avec 100 instances	BR	(52) 67 177	(50) 67 963
	BCG	(16) 24 319	(14) 24 039
	BCI	(16) 23 981	<b>(14) 22 948</b>
	BCF	(17) 25 976	(16) 25 574
	BTG	(23) 31 875	(23) 32 936
	BTI	(23) 32 128	(23) 31 865
	BTF	(25) 32 371	(24) 32 232
	Total B	(172) 237 827	<b>(164) 237 557</b>
	UR	(42) 55 159	(42) 54 407
	UCG	<b>(7) 14 287</b>	(8) 14 534
	UCI	(9) 15 002	(8) 15 303
	UCF	(9) 15 534	(7) 15 361
	UTG	(12) 20 459	(13) 20 013
	UTI	(15) 24 507	(15) 23 228
	UTF	(17) 24 679	(15) 24 818
	Total U	(111) 169 627	<b>(108) 167 664</b>
	AR	(42) 53 683	(40) 52 262
	ACG	(9) 15 874	(9) 15 824
	ACI	<b>(8) 15 588</b>	(10) 15 814
	ACF	(10) 16 989	(10) 17 943
ATG	(13) 24 188	(14) 21 654	
ATI	(15) 24 393	(17) 24 891	
ATF	(16) 23 999	(18) 24 407	
Total A	<b>(113) 174 714</b>	(118) 172 795	
UCTI	(11) 17 239	<b>(9) 17 495</b>	
UACG	(10) 16 454	<b>(8) 16 559</b>	
MINISAT sur 7 problèmes non satisfiables avec 140 instances	BR	(67) 87 505	(65) 86 590
	BCG	(14) 23 828	(13) 23 283
	BCI	(13) 21 987	(15) 24 023
	BCF	(14) 23 638	<b>(12) 23 492</b>
	BTG	(25) 33 285	(26) 34 635
	BTI	(23) 32 920	(23) 33 034
	BTF	(27) 35 374	(26) 34 910
	Total B	(183) 258 537	<b>(180) 259 967</b>
	UR	(59) 79 510	(59) 79 900
	UCG	(8) 13 698	<b>(6) 13 331</b>
	UCI	(7) 13 376	(8) 13 844
	UCF	(9) 14 349	(9) 15 086
	UTG	(16) 23 406	(16) 23 029
	UTI	(17) 24 731	(15) 23 238
	UTF	(19) 26 699	(17) 25 436
	Total U	(135) 195 769	<b>(130) 193 864</b>
	AR	(64) 82 574	(60) 81 965
	ACG	(8) 15 200	<b>(8) 14 356</b>
	ACI	(9) 15 479	(8) 15 526
	ACF	(8) 15 786	(9) 16 508
ATG	(19) 26 035	(18) 28 117	
ATI	(17) 25 343	(17) 24 842	
ATF	(19) 27 088	(20) 27 913	
Total A	(144) 207 505	<b>(140) 209 227</b>	
UCTI	<b>(9) 15 788</b>	(10) 18 306	
UACG	<b>(8) 14 368</b>	(9) 15 638	

TAB. 2 – Résumé des performances des solveurs SAT

Problème	MINISAT		R+DDFW <sup>+</sup>	
	UCI()	UTF(H)	UCI()	UTF(H)
daté-dur-s	96	91	664	192
total-moyen-s	36	56	1 253	732
total-dur-s	124	136	2 654	351
partiel-moyen-s	2 716	(6) 10 936	2 585	3 123
partiel-dur-s	(9) 12 029	(9) 13 599	(1) 6 983	(1) 5 330

TAB. 3 – Comparaison du temps de calcul lorsque la difficulté du problème satisfiable augmente.

MINISAT montre que les clauses apprises durant la recherche sont en moyenne deux fois plus grandes pour les problèmes *\*R* que pour les autres problèmes. On voit aussi que l'ordre des variables *C* (*\*C\**) est préférable à l'ordre *T* (*\*T\**) puisque l'utilisation de ce codage réduit le temps de calcul d'un quart voire d'un tiers.

Voici notre interprétation de ces résultats. Nous pensons que la raison pour laquelle le solveur est plus efficace avec certains codages est que les variables auxiliaires ont une sémantique utile pour le problème de diagnostic. Considérons par exemple la Figure 2d ; notons  $v_1$  la première variable auxiliaire associée au nœud étiqueté  $n_e$ . Cette variable doit être évaluée à *vrai* si et seulement si l'événement  $e$  a lieu dans la trajectoire calculée à n'importe quel moment. Or, si se trouve que souvent, dans un (sous)-problème de diagnostic, on peut déterminer facilement et avec certitude qu'un événement donné a eu lieu, ou au contraire n'a pas eu lieu, dans un intervalle de temps donné. Ainsi, la valeur de cette variable  $v_1$  peut être souvent très facilement calculée, permettant de simplifier le problème général. Ainsi, si l'on sait que l'événement  $e$  a eu lieu à une date inconnue, on peut rapidement remonter cette information pour indiquer que la valeur de la racine est au moins 1, tandis qu'il est beaucoup plus difficile d'en faire autant avec l'arbre de la Figure 2a.

Si les variables sont ordonnées par date, la sémantique de la variable  $v_1$  sur le nœud  $n_i$  est la suivante :  $v_1$  est affectée à *vrai* si une faute a eu lieu au pas de temps  $t = i$ . Cette affirmation, ou son contraire, est beaucoup plus difficile à démontrer, en particulier si le pas de temps associé à chaque observation est variable. C'est notamment le cas lorsque les observations sont partiellement ordonnées : puisqu'on ne sait pas si l'événement observable  $o_1$  a eu lieu avant l'événement observable  $o_2$ , les deux possibilités sont codées dans la CNF avec un pas de temps d'occurrence variable. Ce phénomène est illustré par la Table 4 où le codage *UTG* réagit beaucoup moins bien que *UCG* au passage aux observations partiellement ordonnées.

Intéressons-nous maintenant au solveur SLS. Il est amusant de constater que l'ordre des additions n'a cette fois-ci presque pas d'impact sur les performances

de ce type d'algorithme. Il y a cependant un contre-exemple avec *\*TI* (en particulier pour  $A$  et  $B$ ). Rappelons que la forme d'arbre  $I$  consiste à additionner en série les nœuds  $n_l$ , où  $l$  représente soit un événement de faute soit un pas de temps, comme présenté en Figure 2c. Si le nombre  $m$  de nœuds  $n_l$  est important, on obtient une chaîne de dépendance de taille  $m$  contre  $2 \times \log_2(m)$  pour les codages équilibrés. Or, les algorithmes de type SLS fonctionnent en échangeant la valeur d'un littéral sur les clauses non satisfaites jusqu'à obtenir une solution, propageant ainsi les effets du conflit. Si l'effet d'un conflit doit être propagé sur une longue chaîne, le temps nécessaire pour le faire est plus élevé. C'est d'ailleurs une remarque déjà effectuée récemment par Prestwich [10]. Les événements de faute dans notre *benchmark* ne sont que 20, et la chaîne de dépendance est donc réduite pour *\*CI*. En revanche, le nombre de pas de temps varie selon les problèmes et peut dépasser les 300 unités pour les problèmes comportant 20 occurrences de fautes, ce qui explique la difficulté pour résoudre des problèmes *\*TI*.

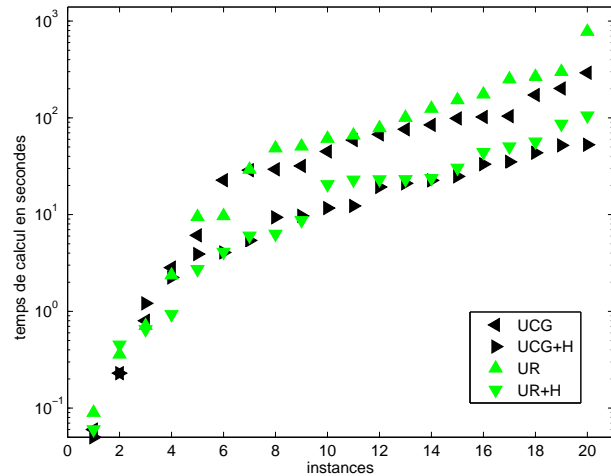


FIG. 4 – R+DDFW<sup>+</sup> sur le problème total-dur-s avec/sans hyper-résolution

**Le codage utilisant l'hyper-résolution** Les clauses additionnelles créées par la technique d'hyper-résolution H représentent environ 1% des clauses du problème original. Ces clauses n'améliorent pas le temps de calcul pour le solveur MINISAT, mais réduisent celui-ci d'environ 30% pour le solveur R+DDFW<sup>+</sup>. L'explication à ce résultat est comme suit. La taille des instances d'un problème est réduite environ 30% par le prétraitement intégré dans le solveur R+DDFW<sup>+</sup>, tandis que le prétraitement utilisé dans le solveur MINISAT ne réduit presque pas la taille. La Figure 4 confirme que le temps de calcul de R+DDFW<sup>+</sup> est amélioré d'un

ordre de magnitude, pour plupart des instances, apres l'integration d'hyper-resolution.

**Les codages hybrides** Le but du codage hybride est de tirer parti des avantages de plusieurs codages. Pour atteindre cet objectif, il n'est pas necessaire que le temps de resolution de chaque probleme hybride soit plus court que le meilleur temps de resolution mais, en revanche, le temps moyen devrait etre plus faible. Clairement, ce n'est pas le cas ici puisque le codage hybride  $UCTI$  requiert +10% de temps pour MINISAT et R+DDFW<sup>+</sup> par rapport au codage  $UCI$ , tandis que le codage  $UACG$  resiste mieux pour MINISAT mais moins bien pour R+DDFW<sup>+</sup>. Ces resultats sont confirmes par les Figures 3g et 3h ou le temps de calcul pour  $UACG$  est le plus souvent au dessus des temps de calcul pour  $UCG$  et  $ACG$ . L'objectif de ce codage n'est donc pas atteint.

Nous pensons que cet echec vient du fait que les deux codages utilises dans les codages hybrides sont trop semblables. Les qualites interessantes du second codage ne differant pas notablement des qualites du premier codage, les benefices de ce second codage ne compensent pas le surcoût engendré par l'ajout des variables auxiliaires et des clauses supplémentaires.

### 5.3.2 Du point de vue du solveur

On ne peut évidemment comparer les temps de calcul que pour les problèmes satisfiables. Rappelons que l'algorithme de diagnostic doit prouver que certains problèmes sont non satisfiables (pour des valeurs de  $k$  trop faibles), et les solveurs SLS ne sont pas suffisants pour un algorithme de diagnostic. Ceci étant précisé, la Table 2 montre que les temps de calcul pour l'algorithme DPLL et l'algorithme SLS sont comparables, ce qui est nouveau par rapport à [4] où les solveurs SLS étaient beaucoup moins efficaces.

La Table 3 compare plus précisément les temps de calcul des deux solveurs pour différents types de problèmes. Les problèmes les plus simples sont résolus plus rapidement par MINISAT, tandis que R+DDFW<sup>+</sup> est plus efficace sur les problèmes difficiles et parvient même à résoudre tous les problèmes satisfiables pour certains codages.

L'efficacité de R+DDFW<sup>+</sup> s'explique d'une part par l'impact du calcul des résolvants qui génère un grand nombre de clauses unitaires au cours de la phase de prétraitement; des expérimentations initiales pour choisir les solveurs montrent que l'intégration de ce prétraitement fait gagner un ordre de grandeur au temps de calcul. D'autre part R+DDFW<sup>+</sup> tire parti d'un mécanisme consistant à redistribuer le poids associé aux clauses pendant la recherche.

Problème	$UR()$	$UCG()$	$UCF()$	$UTG()$
daté-moyen-u	(3) 3 906	25	30	32
daté-dur-u	(5) 9 315	97	103	100
total-facile-u	(4) 5 644	18	20	21
total-moyen-u	(4) 6 808	41	47	51
total-dur-u	(11) 14 491	110	127	198
partiel-moyen-u	(16) 19 342	1 980	2 178	(7) 10 641
partiel-dur-u	(16) 20 003	(8) 11 428	(9) 11 844	(9) 12 364

TAB. 4 – Temps de calcul de MINISAT pour les problèmes non satisfiables, classé par codages de type  $U^*$ .

Concernant les instances non satisfiables, la Table 4 montre une tendance similaire à celle des problèmes satisfiables. MINISAT est maintenant capable de résoudre la plupart des problèmes, à part les plus difficiles, notamment partiel-dur-u.

### 5.3.3 Résolution des instances les plus difficiles

Nous nous intéressons maintenant à la résolution des instances les plus difficiles des problèmes partiel-dur-s et partiel-dur-u.

La Table 5 présente les résultats du problèmes partiel-dur-s pour les plus grandes valeurs de  $k$  qui sont les instances satisfiables les plus difficiles. Dans le cas présent, nous avons donné 5 heures pour la résolution de chaque instance. On voit que les deux solveurs arrivent à résoudre toutes les instances et que R+DDFW<sup>+</sup> résiste bien à l'augmentation de complexité.

Instance	MINISAT		R+DDFW <sup>+</sup>	
	$UCG()$	$UCG(H)$	$UCG()$	$UCG(H)$
18 fautes	5 773	4 118	1 439	1 799
19 fautes	7 790	8 078	984	690
20 fautes	13 542	7 465	1 936	819

TAB. 5 – Comparaison du temps de calcul (en secondes) du problème partiel-dur-s pour différents nombres de fautes, avec le codage  $UCG$

Instance	$UR()$	$UCG()$	$UCG(H)$
13 fautes	>180 000	918	795
14 fautes	>180 000	4 201	3 505
15 fautes	>180 000	1 625	1 233
16 fautes	>180 000	1 706	1 376
17 fautes	>180 000	3 352	3 325
18 fautes	>180 000	2 686	2 890
19 fautes	>180 000	4 574	7 380
20 fautes	>180 000	4 369	7 312

TAB. 6 – Temps de calcul (en secondes) de MINISAT pour les problèmes partiel-dur-u pour différents nombres de fautes, avec les codages  $UR$  et  $UCG$

Dans la Table 6, nous présentons les résultats du problème partiel-dur-u pour les grandes valeurs de  $k$  qui sont les instances non satisfiables les plus difficiles.

Cette fois-ci, nous avons attribué 50 heures pour la résolution de chaque instance. On voit nettement que les instances avec le codage  $UR$  sont beaucoup plus difficiles que celles avec  $UCG$  ou  $UCG(H)$ . Ces résultats confirment l'importance de la sémantique dans le codage CNF de contraintes de cardinalité, particulièrement pour le solveur de type DPLL.

### 5.3.4 Résumé

Les résultats montrent une nette progression de la liste de problèmes qui peuvent désormais être résolus par des solveurs systématiques ou stochastiques. Il apparaît clairement que la représentation unaire ( $U^*$ ) est préférable aux deux autres représentations. Par ailleurs,  $R+DDFW^+$  accepte n'importe quelle forme d'arbre pourvu que celui-ci soit équilibré, tandis que MINISAT préfère des arbres regroupant les variables associées au même composant.

Dans le cadre du diagnostic de SED, nous suggérons donc d'utiliser le codage  $UCF$  ou  $UCG$  qui satisfait ces deux contraintes des algorithmes. En outre, l'implémentation devrait inclure les clauses générées par hyper-resolution.

## 6 Conclusions et perspectives

Dans ce papier, nous avons étudié un certain nombre de codages de cardinalité de contraintes en tirant parti de la sémantique des variables. Nous avons proposé des codages basés sur le totaliseur, et avons étudié comment le codage de chaque nœud du totaliseur ainsi que l'ordre dans lequel l'addition est effectuée influencent le temps de résolution du problème SAT. Les résultats confirment que le codage unaire est le plus efficace. Ils démontrent également que les solveurs systématiques sont très sensibles à l'ordre dans lequel les additions sont effectuées, tandis que les solveurs stochastiques préfèrent éviter les longues chaînes de dépendance, ces deux préférences étant généralement indépendantes.

Ces résultats insistent sur l'importance critique, par ailleurs déjà connue, de la modélisation des problèmes SAT sur les performances dans la résolution. Cette modélisation doit s'appuyer sur la sémantique du problème, particulièrement pour le solveur de type DPLL. La contrainte de cardinalité étant utilisées dans de nombreux domaines comme la planification classique ou la vérification de circuit, les codages présentés dans ce papier peuvent être adaptés pour ces problèmes.

Finalement, nous voudrions étendre nos recherches dans la direction proposée par Marques-Silva and Lynce [6] à savoir comment modifier les solveurs SAT pour qu'ils utilisent la sémantique des variables dans la formule CNF pour améliorer les temps de réponse.

## Remerciements

Ce travail est supporté par NICTA au sein des projets SuperCom et G12/CPP. NICTA est subventionné par le Gouvernement Australien représenté par le *Department of Broadband, Communications and the Digital Economy* et l'*Australian Research Council* au travers du programme *ICT Centre of Excellence*.

## Références

- [1] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of ACM*, 5 :394–397, 1962.
- [2] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. of 6th SAT*, pages 502–518, 2004.
- [3] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2 :1–26, 2006.
- [4] A. Grastien et Anbulagan. Résolution d'un problème de diagnostic de systèmes à événements discrets par SAT. In *Proc. of Troisièmes Journées Francophones de Programmation par Contraintes (JFPC 2007)*, pages 387–396, 2007.
- [5] A. Grastien et Anbulagan et J. Rintanen et E. Keilareva. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proc. of 19th AAAI*, pages 305–310, 2007.
- [6] J. Marques-Silva et I. Lynce. Towards robust CNF encodings of cardinality constraints. In *Proc. of 13th CP*, pages 483–497, 2007.
- [7] A. Ishtaiwi et J. Thornton et Anbulagan et A. Sattar et D. N. Pham. Adaptive clause weight redistribution. In *Proc. of 12th CP*, pages 229–243, 2006.
- [8] G. Lamperti et M. Zanella. *Diagnosis of Active Systems*. Kluwer Academic Publishers, 2003.
- [9] O. Bailleux et Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proc. of 9th CP*, pages 108–122, 2003.
- [10] S. Prestwich. Variable dependency in local search : Prevention is better than cure. In *Proc. of 10th SAT*, pages 107–120, 2007.
- [11] C. Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proc. of 11th CP*, pages 827–831, 2005.
- [12] J. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68 :63–69, 1998.