Combiner les Redémarrages, Nogoods et Décompositions pour la Résolution de CSP *

Philippe Jégou **Cyril Terrioux**
Aix-Marseille Université, LSIS UMR 7296
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{philippe.jegou, cyril.terrioux}@lsis.org

Résumé

Du point de vue théorique, les méthodes exploitant des décompositions (arborescentes) constituent une approche pertinente pour la résolution de CSP quand la largeur (arborescente) des réseaux de contraintes est petite. Dans ce cas, elles ont souvent montré leur intérêt pratique. Cependant, parfois, un mauvais choix pour le cluster racine (une décomposition arborescente est un arbre de clusters) peut dégrader significativement les performances de résolution.

Dans cet article, nous mettons en avant une explication de cette dégradation et nous proposons une solution reposant sur les techniques de redémarrage. Ensuite, nous présentons une nouvelle version de l'algorithme BTD (pour Backtracking with Tree-Decomposition [8]) intégrant des techniques de redémarrage. D'un point de vue théorique, nous prouvons que des nld-nogoods réduits peuvent être mémorisés et exploités durant la recherche et que leur taille est plus restreinte que celle des nld-nogoods enregistrés par MAC+RST+NG [9]. Nous décrivons également comment les (no)goods structurels peuvent être exploités quand la recherche redémarre à partir d'une nouvelle racine. Enfin, nous montrons empiriquement l'intérêt de l'approche proposée.

1 Introduction

Un problème de satisfaction de contraintes (CSP, voir [14] pour un état de l'art) est un triplet (X, D, C), où $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $D = (d_{x_1}, \ldots, d_{x_n})$ est une liste de domaines finis, à raison d'un domaine par variable, et $C = \{C_1, \ldots, C_e\}$ est un ensemble fini de e contraintes. Chaque contrainte C_i est un couple $(S(C_i), R(C_i))$, où $S(C_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ est la *portée* de C_i, et $R(C_i) \subseteq d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$ est sa *relation de compatibilité*. L'*arité* de C_i est $|S(C_i)|$. Un CSP est dit *binaire* si toutes ses contraintes sont d'arité 2. La structure d'un réseau de contraintes est représentée par un hypergraphe (un graphe dans le cas binaire), appelé *hypergraphe de contraintes*, dont les sommets correspondent aux variables et les arêtes aux portées des contraintes. Dans cet article, pour simplifier le propos, nous ne traitons que du cas binaire. Cependant ce travail peut facilement s'étendre au cas non-binaire en exploitant la 2-section [1] (aussi appelée graphe primal) de l'hypergraphe de contraintes, ce que nous ferons d'ailleurs pour nos expérimentations lors desquelles nous considérerons des CSP binaires et non-binaires. De plus, sans perte de généralité, nous supposons que le graphe de contraintes est connexe. Pour simplifier les notations, dans la suite, nous noterons le graphe $(X, \{S(C_1), \ldots S(C_e)\})$ par (X, C). Une affectation sur un sous-ensemble de X est dite *cohérente* si elle ne viole aucune contrainte. Déterminer si un CSP possède une *solution* (i.e. une affectation cohérente de toutes les variables) est connu pour être NP-complet. Aussi, de nombreux travaux ont été réalisés pour rendre la résolution plus efficace en pratique en utilisant des algorithmes énumératifs optimisés qui peuvent exploiter des heuristiques, de l'apprentissage de contraintes, du retour en arrière non-chronologique, des techniques de filtrage basées sur la propagation de contraintes, etc. La complexité en temps de ces approches est naturellement exponentielle, au moins en $O(n.d^n)$ avec n le nombre de variables et d la taille du plus grand domaine.

Généralement, afin de garantir une résolution effi-

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet TUPLES (ANR-2010-BLAN-0210).

cace, la plupart des solveurs exploitent simultanément plusieurs techniques parmi celles précédemment citées (comme les heuristiques, l'apprentissage de contraintes ou le filtrage). De plus, souvent, ils tirent également profit de l'utilisation de techniques de redémarrage. En particulier, les techniques de redémarrage permettent habituellement de réduire l'impact de mauvais choix effectués par des heuristiques (comme l'heuristique d'ordonnancement des variables) ou de l'apparition du phénomène de *queue lourde*. Elles ont été récemment introduites dans le cadre des CSP (e.g. dans [9]). Pour des raisons liées à l'efficacité, elles sont généralement exploitées avec des techniques d'apprentissage (comme la mémorisation de nld-nogoods [9]).

Dans cet article, nous introduisons pour la première fois, les techniques de redémarrage dans des méthodes de résolution de CSP par décomposition. Les méthodes par décomposition (e.g. [4, 8]) résolvent les CSP en prenant en compte certaines propriétés du réseau de contraintes. Souvent, elles reposent sur la notion de décomposition arborescente de graphes [12]. Dans ce cas, leur intérêt est relatif à leur complexité théorique en temps, i.e. $d^{w^+ + 1}$ avec w^+ la largeur de la décomposition arborescente considérée. Comme le calcul d'une décomposition arborescente optimale est NP-Difficile, les décompositions employées sont généralement produites à l'aide de méthodes heuristiques et donc fournissent une approximation d'une décomposition optimale. Quand le graphe possède de bonnes propriétés topologiques, et donc quand w^+ est petit, ces méthodes permettent de résoudre de grandes instances, e.g. les instances d'allocation de fréquence [3]. D'un point de vue pratique, elles obtiennent des résultats prometteurs sur de telles instances. Cependant, leur efficacité peut être significativement dégradée par de mauvais choix effectués par des heuristiques. Afin de présenter ce problème, nous considérerons ici la méthode BTD [8] qui constitue une référence dans l'état de l'art pour ce type d'approche [11].

Au niveau de BTD, la décomposition arborescente considérée et le choix d'un cluster racine (i.e. du premier cluster étudié) induisent un ordre particulier sur les variables. Aussi, sachant l'impact significatif que l'ordre sur les variables peut avoir sur l'efficacité d'un solveur, ce choix de cluster racine est crucial. Dans [7], l'approche proposée consiste à choisir l'ordre sur les variables avec plus de liberté, mais l'efficacité dépend toujours du choix du cluster racine. Dans la prochaine section, nous expliquons pourquoi il est difficile de proposer un choix de cluster racine convenable. En conséquence, afin de réduire l'impact du choix de racine sur l'efficacité pratique, nous proposons une alternative basée sur les techniques de redémarrage. Nous présentons alors une nouvelle version de BTD utilisant ces techniques. D'un point de vue théorique, nous prouvons ensuite que des nld-nogoods réduits peuvent être enregistrés durant la recherche et que leur taille est restreinte par rapport à celle des nld-nogoods mémorisés par MAC+RST+NG [9]. Nous décrivons aussi comment les (no)goods structurels peuvent être exploités quand la recherche redémarre à partir d'un nouveau cluster racine. Enfin, nous montrons expérimentalement l'intérêt de l'approche proposée.

La section 2 rappelle le cadre de BTD et décrit l'algorithme BTD-MAC [1]. Ensuite, la section 3 présente l'algorithme BTD-MAC+RST. Dans la section 4, nous évaluons l'intérêt des redémarrages pour la résolution de CSP à l'aide de méthodes par décomposition, avant de conclure dans la section 5.

2 La méthode BTD

La méthode BTD [8] constitue une référence dans l'état de l'art lorsqu'il s'agit de résoudre des CSP en exploitant la structure de leur graphe de contraintes [11]. Elle repose sur la notion de décomposition arborescente de graphes [12].

Définition 1 *Une* décomposition arborescente *d'un graphe $G = (X, C)$ est un couple (E, T) avec un arbre $T = (I, F)$ et une famille $E = \{E_i : i \in I\}$ de sous-ensembles de X, tel que chaque sous-ensemble (appelé cluster) E_i est un nœud de T et vérifie :*

(i) $\cup_{i \in I} E_i = X$,

(ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et

(iii) pour tout $i, j, k \in I$, si k est sur un chemin de i à j dans T, alors $E_i \cap E_j \subseteq E_k$.

La largeur d'une décomposition arborescente (E, T) est égale à $max_{i \in I}|E_i| - 1$. La largeur arborescente *(ou* tree-width*) w de G est la largeur minimale sur toutes les décompositions arborescentes de G.*

Etant donnés une décomposition arborescente (E, T) et un cluster racine E_r, nous notons $Desc(E_j)$ l'ensemble des sommets (variables) appartenant à l'union des descendants E_k de E_j dans le sous-arbre enraciné en E_j, E_j inclus. La figure 1(b) présente un arbre dont les nœuds correspondent aux cliques maximales du graphe de la figure 1(a). Il s'agit d'une décomposition arborescente possible pour ce graphe. Aussi, nous avons $E_1 = \{x_1, x_2, x_3\}$, $E_2 = \{x_2, x_3, x_4, x_5\}$, $E_3 = \{x_4, x_5, x_6\}$, $E_4 = \{x_3, x_7, x_8\}$, $Desc(E_1) = X$ et $Desc(E_2) = \{x_2, x_3, x_4, x_5, x_6\}$.

1. Cet algorithme n'a jamais été décrit précisément dans la littérature. L'algorithme MAC-BTD évoqué dans [8] correspond en fait à BTD-RFL (i.e. BTD basé sur Real-Full Lookahead).

Comme la taille maximale des clusters est 4, la largeur arborescente de ce graphe est 3.

Etant donné un ordre compatible $<$ sur les cluster (i.e. un ordre qui peut être produit par un parcours en profondeur d'abord de T à partir du cluster racine E_r), BTD effectue une recherche énumérative en utilisant un ordre $\preceq$ sur les variables (dit *compatible*) tel que $\forall x \in E_i$, $\forall y \in E_j$, avec $E_i < E_j$, $x \preceq y$. Autrement dit, l'ordre sur les clusters induit un ordre partiel sur les variables puisque les variables de E_i sont affectées avant celles de E_j si $E_i < E_j$. Pour l'exemple de la figure 1, $E_1 < E_2 < E_3 < E_4$ (resp. $x_1 \preceq x_2 \preceq x_3 \preceq \ldots \preceq x_8$) est un ordre compatible possible sur E (resp. X). En pratique, l'algorithme BTD débute sa recherche en affectant de façon cohérente les variables du cluster racine E_r avant d'explorer un cluster fils. Quand il explore un nouveau cluster E_i, il n'affecte que les variables de $E_i \backslash (E_i \cap E_{p(i)})$ [2].

Afin de résoudre chaque cluster, BTD peut exploiter n'importe quel algorithme qui ne modifie pas la structure du graphe de contraintes. Par exemple, BTD peut reposer sur l'algorithme MAC (pour Maintaining Arc-Consistency [15]). Durant la résolution, MAC peut prendre deux types de décisions : des *décisions positives* $x_i = v_i$ qui affectent la valeur v_i à la variable x_i (nous notons $Pos(\Sigma)$ l'ensemble des décisions positives de la suite de décisions Σ) et des *décisions négatives* $x_i \neq v_i$ qui assurent que x_i ne pourra pas être affectée à la valeur v_i. Soit $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ (où chaque δ_j peut être une décision positive ou négative) la suite de décisions courante. Une nouvelle décision positive $x_{i+1} = v_{i+1}$ est choisie et un filtrage par cohérence d'arc (AC) est accompli. Si aucun domaine ne devient vide, la recherche continue avec la prise d'une nouvelle décision positive. Sinon, la valeur v_{i+1} est supprimée du domaine $d_{x_{i+1}}$, et un filtrage par AC est réalisé. Si un échec se produit à nouveau, MAC revient en arrière et change la dernière décision positive $x_\ell = v_\ell$ en $x_\ell \neq v_\ell$. Concernant BTD-MAC (i.e. BTD utilisant MAC pour résoudre chaque cluster), nous pouvons constater que la prochaine décision positive porte nécessairement sur une variable du cluster courant E_i, du moment où une variable de E_i ne figure pas dans une décision positive, et que seuls les domaines des variables futures présentes dans $Desc(E_i)$ sont susceptibles d'être impactés par le filtrage par AC (car $E_i \cap E_{p(i)}$ est un séparateur du graphe de contraintes et que toutes ses variables ont déjà été affectées).

Quand BTD a instancié de façon cohérente toutes les variables d'un cluster E_i, il tente de résoudre chaque sous-problème enraciné en un des clusters fils E_j de E_i. Plus précisément, pour un fils E_j et une suite de décisions courante Σ, il essaie de résoudre le sous-problème induit par les variables de $Desc(E_j)$ et l'ensemble de décisions $Pos(\Sigma)[E_i \cap E_j]$ (i.e. l'ensemble des décisions positives impliquant une variable de $E_i \cap E_j$). Une fois ce sous-problème résolu, en prouvant soit l'existence, soit l'inexistence de solution, il mémorise soit un good structurel, soit un nogood structurel. Formellement, étant donnés un cluster E_i et E_j un de ses fils, un *good structurel* (resp. *nogood structurel*) de E_i par rapport à E_j est une affectation cohérente A de $E_i \cap E_j$ telle que A peut (resp. ne peut pas) être étendue de façon cohérente sur $Desc(E_j)$ [8]. Dans le cas particulier de BTD-MAC, l'affectation cohérente A est représentée par la restriction de l'ensemble des décisions positives de Σ sur $E_i \cap E_j$, à savoir $Pos(\Sigma)[E_i \cap E_j]$. Ces (no)goods structurels peuvent être employés plus tard dans la recherche pour éviter d'explorer une partie redondante de l'arbre de recherche. En effet, dès que la suite de décisions courante Σ contient un good (resp. nogood) de E_i par rapport à E_j, BTD n'a plus besoin de résoudre à nouveau le sous-problème induit par $Desc(E_j)$ et $Pos(\Sigma)[E_i \cap E_j]$ car il a déjà établi que ce sous-problème possédait au moins une solution (resp. aucune). Dans le cas d'un good, il poursuit sa recherche avec le prochain cluster fils. Dans celui d'un nogood, il revient en arrière. Par exemple, considérons un CSP de 8 variables $x_1, \ldots, x_8$ ayant chacune pour domaine $\{a, b, c\}$ et dont le graphe de contraintes et une décomposition arborescente possible sont donnés à la figure 1. Supposons que la suite de décisions courante $\Sigma = \langle x_1 = a, x_2 \neq b, x_2 = c, x_3 = b \rangle$ a été construite selon un ordre sur les variables compatible avec l'ordre sur les clusters $E_1 < E_2 < E_3 < E_4$. BTD essaie de résoudre le sous-problème enraciné en E_2 et une fois résolu, il mémorise $\{x_2 = c, x_3 = b\}$ comme un good ou un nogood structurel de E_1 par rapport à E_2. Si, plus tard, BTD étudie la suite de décisions $\langle x_1 \neq a, x_3 = b, x_1 = b, x_2 \neq a, x_2 = c \rangle$ et que celle-ci est cohérente, il continuera sa recherche avec le prochain fils de E_1 (à savoir E_4), si $\{x_2 = c, x_3 = b\}$ a été mémorisé sous la forme d'un good, ou il revient en arrière à la dernière décision de E_1 si $\{x_2 = c, x_3 = b\}$ correspond à un nogood.

L'algorithme 1 correspond à l'algorithme BTD-MAC. Initialement, la suite de décisions courante et les ensembles N et G de (no)goods structurels mémorisés sont vides et la recherche débute avec les variables du cluster racine E_r. Etant donnés un cluster courant E_i et la suite de décisions courante Σ, les lignes 16-23 consistent à explorer le cluster E_i en affectant les variables de V_{E_i} (avec V_{E_i} l'ensemble des variables non-instanciées du cluster E_i) comme le ferait MAC tandis que les lignes 1-14 permettent de gérer les fils de E_i et donc d'utiliser et d'enregistrer des (no)goods struc-

2. On suppose que $E_i \cap E_{p(i)} = \emptyset$ si E_i est le cluster racine.

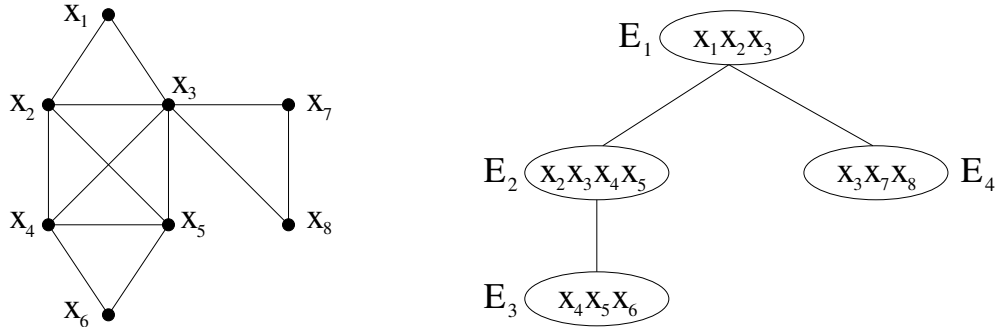

FIGURE 1 – Un graphe de contraintes de 8 variables (a) et une décomposition arborescente optimale (b).

Algorithme 1: BTD-MAC (**InOut** : $P = (X, D, C)$: CSP ; **In** : Σ : suite de décisions, E_i : cluster, V_{E_i} : ensemble de variables ; **InOut** : G : ensemble de goods, N : ensemble de nogoods)

```
1  if V_{E_i} = ∅ then
2      result ← true
3      S ← Sons(E_i)
4      while result and S ≠ ∅ do
5          Choose a cluster E_j ∈ S
6          S ← S\{E_j}
7          if Pos(Σ)[E_i ∩ E_j] is a nogood of E_i w.r.t. E_j in N
           then result ← false
8          else if Pos(Σ)[E_i ∩ E_j] is not a good of E_i w.r.t.
           E_j in G then
9              result ← BTD-MAC(P,Σ,E_j,E_j\(E_i ∩ E_j),G,N)
10             if result then
11                 Record Pos(Σ)[E_i ∩ E_j] as good of E_i w.r.t.
                   E_j in G
12             else
13                 Record Pos(Σ)[E_i ∩ E_j] as nogood of E_i
                   w.r.t. E_j in N
14     return result
15 else
16     Choose a variable x ∈ V_{E_i}
17     Choose a value v ∈ d_x
18     d_x ← d_x\{v}
19     if AC (P,Σ ∪ ⟨x = v⟩) ∧ BTD-MAC(P, Σ ∪ ⟨x = v⟩, E_i,
       V_{E_i}\{x}, G, N)= true then return true
20     else
21         if AC (P,Σ ∪ ⟨x ≠ v⟩) then
22             return BTD-MAC(P,Σ ∪ ⟨x ≠ v⟩,E_i,V_{E_i},G,N)
23         else return false
```

turels. BTD-MAC(P,Σ,E_i,V_{E_i},G,N) renvoie *true* s'il parvient à étendre Σ de façon cohérente sur les variables de $Desc(E_i)\backslash(E_i\backslash V_{E_i})$, *false* sinon. Sa complexité en temps est $O(n.s^2.e.\log(d).d^{w^+ + 2})$ pour une complexité en espace en $O(n.s.d^s)$ avec w^+ la largeur de la décomposition arborescente considérée et s la taille de la plus grande intersection entre deux clusters.

D'un point de vue pratique, généralement, BTD résout efficacement les CSP ayant une petite largeur arborescente [6, 7, 8]. Cependant, parfois, un mauvais choix de cluster racine peut dégrader significativement la qualité de la résolution. Le choix du cluster racine est crucial dans la mesure où il impacte directement l'ordre sur les variables, en particulier le choix des premières variables. Aussi, afin de faire un choix plus éclairé, nous avons sélectionné quelques instances de la compétition de solveurs de 2008 [3] pour lesquelles nous avons lancé BTD à partir de chaque cluster de la décomposition arborescente considérée. Nous avons d'abord constaté que, pour une même instance, les temps d'exécution varient de plusieurs ordres de grandeur selon le cluster racine choisi. Par exemple, pour l'instance scen11-f12 (qui est la plus facile de la famille scen11), BTD ne parvient à prouver l'absence de solution que pour 75 des 301 choix de racine possibles. Ensuite, nous avons observé que résoudre certains clusters (pas nécessairement le cluster racine) et leurs sous-problèmes se révèle être plus coûteux pour certains choix de racine que pour d'autres. Cela s'explique par le fait que le choix du cluster racine induit un ordre particulier sur les clusters et les variables. En particulier, comme pour un cluster E_i, BTD ne tient compte que des variables de $E_i\backslash(E_i \cap E_{p(i)})$, il ne manipule pas le même ensemble de variables pour E_i selon la racine choisie. Malheureusement, il semble utopique de vouloir proposer un choix de racine basé uniquement sur les propriétés de l'instance à résoudre car ce choix est intimement lié à l'efficacité de la résolution. Dans [7], une approche a été proposée pour choisir l'ordre sur les variables avec plus de liberté mais son efficacité demeure dépendante du choix de racine. Limiter l'impact du choix de racine est donc une nécessité. Dans la section 3, nous proposons une solution exploitant des techniques de redémarrage.

3 Exploitation des redémarrages au sein de BTD

Il est bien connu que n'importe quelle méthode exploitant des techniques de redémarrage doit autant que

3. Voir http ://www.cril.univ-artois.fr/CPAI08.

possible éviter d'explorer plusieurs fois les mêmes parties de l'espace de recherche et que la randomisation et l'apprentissage sont deux voies possibles pour atteindre ce but. Au niveau de l'apprentissage, BTD exploite déjà des (no)goods structurels. La première question qui se pose est de savoir s'il est possible de réutiliser des (no)goods structurels après un redémarrage et si oui, sous quelles conditions. De plus, suivant le moment où se produit le redémarrage, nous n'avons aucune garantie qu'un (no)good aura déjà été mémorisé. Aussi, une autre forme d'apprentissage est requise si on veut garantir une bonne efficacité pratique. Ici, nous considérons les nld-nogoods réduits (pour negative last decision nogoods) dont l'intérêt pratique a été mis en avant dans l'algorithme MAC+RST+NG [9]. Nous rappelons d'abord la notion de nogood dans le cas de MAC :

Définition 2 ([9]) *Etant donnés un CSP* $P = (X, D, C)$ *et un ensemble de décisions* Δ, $P_{|\Delta}$ *est le CSP* (X, D', C) *avec* $D' = (d'_{x_1}, \ldots, d'_{x_n})$ *tel que pour chaque décision positive* $x_i = v_i$, $d'_{x_i} = \{v_i\}$ *et pour chaque décision négative* $x_i \neq v_i$, $d'_{x_i} = d_{x_i} \backslash \{v_i\}$. *Pour le cas où* x_i *n'apparaît pas dans* Δ, *on a* $d'_{x_i} = d_{x_i}$.
Δ *est un* nogood *de* P *si* $P_{|\Delta}$ *n'a pas de solution.*

Dans la suite, nous supposerons que pour toute variable x_i et toute valeur v_i, la décision positive $x_i = v_i$ est toujours considérée avant la décision $x_i \neq v_i$.

Proposition 1 ([9]) *Soit* $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ *une suite de décisions prises le long d'une branche de l'arbre de recherche développé durant la résolution d'un CSP* P. *Pour toute sous-suite* $\Sigma' = \langle \delta_1, \ldots, \delta_\ell \rangle$ *préfixe de* Σ *telle que* δ_ℓ *est une décision négative, l'ensemble* $Pos(\Sigma') \cup \{\neg\delta_\ell\}$ *est un nogood (appelé un* nld-nogood réduit*) de* P *avec* $\neg\delta_\ell$ *la décision positive correspondant à* δ_ℓ.

Autrement dit, étant donnée une suite de décisions Σ prises le long d'une branche d'un arbre de recherche, chaque nld-nogood réduit caractérise la visite d'une partie inconsistante de cet arbre de recherche. Quand un redémarrage se produit, un algorithme comme MAC+RST+NG peut mémoriser de nouveaux nld-nogoods réduits et les exploiter ensuite pour éviter d'explorer à nouveau des parties déjà visitées de l'arbre de recherche. Il a été établi dans [9] que le calcul et l'utilisation de ces nld-nogoods réduits peuvent être faits efficacement, notamment en les stockant sous la forme d'une contrainte globale avec un propagateur spécifique pour établir la cohérence d'arc.

L'utilisation d'apprentissage au sein de BTD peut mettre en danger sa validité dès qu'on ajoute au problème initial une contrainte dont la portée n'est incluse dans aucun cluster. Par conséquent, la mémorisation des nld-nogoods réduits dans une contrainte globale portant sur toutes les variables, comme proposé dans [9], est impossible. Cependant, en exploitant les propriétés d'un ordre compatible sur les variables, la proposition 2 montre que cette contrainte globale peut être décomposée en une contrainte globale par cluster.

Proposition 2 *Soit* $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ *une suite de décisions prises le long d'une branche de l'arbre de recherche développé durant la résolution d'un CSP* P *à l'aide d'une décomposition arborescente* (E, T) *et d'un ordre compatible sur les variables. Soit* $\Sigma[E_i]$ *la sous-suite construite en ne considérant que les décisions de* Σ *concernant des variables de* E_i. *Pour toute sous-suite* $\Sigma'_{E_i} = \langle \delta_{i_1}, \ldots, \delta_{i_\ell} \rangle$ *préfixe de* $\Sigma[E_i]$ *telle que* δ_{i_ℓ} *est une décision négative et que chaque variable de* $E_i \cap E_{p(i)}$ *apparaît dans une décision de* $Pos(\Sigma'_{E_i})$, *l'ensemble* $Pos(\Sigma'_{E_i}) \cup \{\neg\delta_{i_\ell}\}$ *est un nld-nogood réduit de* P.

Preuve : Soient P_{E_i} le sous-problème induit par les variables de $Desc(E_i)$ et Δ_{E_i} l'ensemble des décisions de $Pos(\Sigma_{E_i})$ relatives aux variables de $E_i \cap E_{p(i)}$. Comme $E_i \cap E_{p(i)}$ est un séparateur du graphe de contraintes, $P_{E_i|\Delta_{E_i}}$ est indépendant du reste du problème P. Considérons $\Sigma[E_i]$ la sous-suite de Σ qui ne contient que des décisions impliquant des variables de E_i. D'après la proposition 1 appliquée à $\Sigma[E_i]$ et $P_{E_i|\Delta_{E_i}}$, $Pos(\Sigma'_{E_i}) \cup \{\neg\delta_{i_\ell}\}$ est nécessairement un nld-nogood réduit. □

Il en découle les deux corollaires suivants qui permettent de majorer la taille des nogoods produits et de les comparer à ceux produits par la proposition 1.

Corollaire 1 *Etant donnée une décomposition arborescente de largeur* w^+, *les nogoods réduits produits par la proposition 2 sont de taille au plus* $w^+ + 1$.

Corollaire 2 *Sous les mêmes hypothèses que la proposition 2, pour chaque nld-nogood réduit* Δ *produit par la proposition 1, il existe au moins un nld-nogood réduit* Δ' *produit par la Proposition 2 tel que* $\Delta' \subseteq \Delta$.

En mémorisant des (no)goods structurels, BTD exploite déjà une forme particulière d'apprentissage. Tout (no)good structurel d'un cluster E_i par rapport à un cluster fils E_j est par définition orienté de E_i vers E_j. Cette orientation est induite directement par le choix d'un cluster racine. Quand un redémarrage se produit, BTD peut choisir un cluster différent comme cluster racine. Si c'est le cas, nous devons considérer des (no)goods structurels avec différentes orientations. La proposition 3 établit comment ces (no)goods structurels peuvent être utilisés de façon valide quand BTD exploite des techniques de redémarrage.

Proposition 3 *Un good structurel de E_i par rapport à E_j peut être utilisé si le choix courant de cluster racine induit que E_j est un fils de E_i.*
Un nogood structurel de E_i par rapport à E_j peut être utilisé quel que soit le choix de cluster racine.

Preuve : Soit un good Δ de E_i par rapport à E_j produit pour un cluster racine E_r. Par définition d'un good structurel, le sous-problème $P_{E_j|\Delta}$ possède au moins une solution et sa définition ne dépend que de Δ et du fait que E_j est un fils de E_i. Par conséquent, pour n'importe quel choix de cluster racine pour lequel E_j est un fils de E_i, Δ sera encore un good structurel de E_i par rapport à E_j et pourra donc être utilisé pour ne pas explorer des parties redondantes de l'espace de recherche.

Concernant les nogoods structurels, tout nogood structurel Δ de E_i par rapport à E_j est un nogood, et donc toute suite de décisions Σ telle que $\Delta \subseteq Pos(\Sigma)$ ne pourra être étendue en une solution, indépendamment du choix de cluster racine. Ainsi, les nogoods structurels sont utilisables quel que soit le choix de racine. □

Il s'ensuit que contrairement aux nogoods, pour les goods, l'orientation doit être prise en compte. Il serait donc plus pertinent des les appeler des *goods structurels orientés*.

L'algorithme 3 décrit l'algorithme BTD-MAC+RST qui exploite des techniques de redémarrage tout en mémorisant des nld-nogoods réduits et des (no)goods structurels. L'exploitation des techniques de redémarrage peut être vue comme le choix d'un cluster racine (ligne 3) et l'exécution d'une nouvelle instance de BTD-MAC+NG (ligne 4) à chaque redémarrage jusqu'à ce que le problème soit résolu en montrant l'(in)existence de solution. L'algorithme 2 présente l'algorithme BTD-MAC+NG. Comme BTD-MAC, étant donnés un cluster E_i et une suite de décisions courante Σ, BTD-MAC+NG explore le cluster E_i (lignes 16-27) en affectant les variables de V_{E_i} (avec V_{E_i} l'ensemble des variables non instanciées de E_i). Quand E_i est instancié de manière cohérente, il gère les fils de E_i et donc utilise et mémorise des (no)goods structurels (lignes 1-14). Les (no)goods structurels utilisés peuvent avoir été mémorisés durant l'appel courant à BTD-MAC+NG ou durant un appel précédent. En effet, si le premier appel à BTD-MAC+NG s'effectue avec des ensembles vides pour les ensembles G et N de goods et nogoods structurels, G et N ne sont pas réinitialisés à chaque redémarrage. Notons que leurs emplois (lignes 7-8) se font en accord avec la proposition 3. Ensuite, à la différence de BTD-MAC, BTD-MAC+NG peut stopper sa recherche aussitôt que la condition de redémarrage est atteinte (ligne 21). Si c'est le cas, il mémorise des nld-nogoods réduits par rapport à la suite de décisions Σ restreinte aux décisions impliquant les variables de E_i (ligne 22) en accord avec la proposition 2. La détection et la mémorisation des nld-nogoods réduits s'effectuent selon la méthode proposée dans [9] appliquée à la sous-suite de décisions de Σ n'impliquant que des décisions concernant des variables du cluster E_i. Nous considérons qu'une contrainte globale est associée à chaque cluster E_i pour manipuler les nld-nogoods enregistrés au niveau de E_i et que leur exploitation s'effectue au travers d'un propagateur spécifique quand la cohérence d'arc est appliquée (lignes 19 et 25) comme dans [9]. La condition de redémarrage peut porter sur des paramètres globaux (e.g. le nombre de retours en arrière accomplis depuis le début de l'appel courant à BTD-MAC+NG), des locaux (e.g. le nombre de retours en arrière accomplis dans le cluster courant ou le nombre de (no)goods structurels mémorisés) ou une combinaison des deux. BTD-MAC+NG(P,Σ,E_i,V_{E_i},G,N) renvoie :

- *true* s'il parvient à étendre Σ de façon cohérente sur les variables de $Desc(E_i)\backslash(E_i\backslash V_{E_i})$,
- $false$ s'il prouve que Σ ne peut pas s'étendre de façon cohérente sur les variables de $Desc(E_i)\backslash(E_i\backslash V_{E_i})$,
- *unknown* si un redémarrage se produit.

BTD-MAC+RST(P) renvoie *true* si P possède au moins une solution, $false$ sinon.

Théorème 1 *BTD-MAC+RST est correct, complet et termine.*

Preuve : BTD-MAC+NG diffère de BTD-MAC par l'exploitation des techniques de redémarrage, par l'enregistrement de nld-nogoods réduits et au niveau des ensembles G et N qui ne sont pas nécessairement vides au départ. Quand un redémarrage se produit, la recherche est arrêtée et des nld-nogoods réduits sont mémorisés de façon valide en accord avec la proposition 2. Concernant les (no)goods structurels, N et G ne contiennent que des (no)goods structurels valides et leurs usages (lignes 7-8) s'effectuent en accord avec la proposition 3. Par conséquent, comme BTD-MAC est correct et termine et que ces propriétés ne sont pas remises en cause par les différences entre BTD-MAC et BTD-MAC+NG, il en est de même pour BTD-MAC+NG. Concernant la complétude, comme BTD-MAC est complet, BTD-MAC+NG l'est aussi sous réserve qu'aucun redémarrage ne se produise. Par ailleurs, la survenue d'un redémarrage ne fait qu'interrompre la recherche. Elle ne remet donc pas en cause le fait que s'il avait eu une solution dans la partie de l'espace de recherche explorée par l'appel courant à BTD-MAC+NG, celui-ci aurait trouvé cette solution.

Algorithme 2: BTD-MAC+NG (**InOut** : $P = (X, D, C)$: CSP ; **In** : Σ : suite de décisions, E_i : cluster, V_{E_i} : ensemble de variables ; **InOut** : G : ensemble de goods, N : ensemble de nogoods)

```
1  if V_{E_i} = ∅ then
2      result ← true
3      S ← Sons(E_i)
4      while result = true and S ≠ ∅ do
5          Choose a cluster E_j ∈ S
6          S ← S\{E_j}
7          if Pos(Σ)[E_i ∩ E_j] is a nogood in N then
           result ← false
8          else if Pos(Σ)[E_i ∩ E_j] is not a good of E_i w.r.t.
           E_j in G then
9              result ←
               BTD-MAC+NG(P,Σ,E_j,E_j\(E_i ∩ E_j),G,N)
10             if result = true then
11                 Record Pos(Σ)[E_i ∩ E_j] as good of E_i w.r.t.
                   E_j in G
12             else if result = false then
13                 Record Pos(Σ)[E_i ∩ E_j] as nogood of E_i
                   w.r.t. E_j in N
14     return result
15 else
16     Choose a variable x ∈ V_{E_i}
17     Choose a value v ∈ d_x
18     d_x ← d_x\{v}
19     if AC (P,Σ ∪ ⟨x = v⟩) ∧ BTD-MAC+NG(P,
       Σ ∪ ⟨x = v⟩, E_i, V_{E_i}\{x}, G, N)= true then return
       true
20     else
21         if must restart then
22             Record nld-nogoods w.r.t. the decision sequence
               Σ[E_i]
23             return unknown
24         else
25             if AC (P,Σ ∪ ⟨x ≠ v⟩) then
26                 return
                   BTD-MAC+NG(P,Σ ∪ ⟨x ≠ v⟩,E_i,V_{E_i},G,N)
27             else return false
```

Comme BTD-MAC+RST ne fait que réaliser plusieurs appels à BTD-MAC+NG, il est forcément correct. Au niveau de la complétude, si l'appel à BTD-MAC+NG n'est pas interrompu par un redémarrage (ce qui est nécessairement le cas du dernier appel à BTD-MAC+NG si BTD-MAC+RST termine), la complétude de BTD-MAC+NG implique celle de BTD-MAC+RST. Par ailleurs, l'enregistrement de nld-nogoods réduits à chaque redémarrage garantit de ne pas explorer une partie de l'espace de recherche déjà explorée par un précédent appel à BTD-MAC+NG. Il s'ensuit qu'au fil des appels successifs à BTD-MAC+NG, on va explorer une partie de plus en plus réduite de l'espace de recherche. Ainsi, la terminaison et donc la complétude de BTD-MAC+RST sont garanties par l'enregistrement non limité de nld-nogoods effectué durant les différents appels à BTD-MAC+NG ainsi que par la terminaison et la complétude de BTD-MAC+NG. □

Algorithme 3: BTD-MAC+RST (**In** : $P = (X, D, C)$: CSP)

```
1 G ← ∅ ; N ← ∅
2 repeat
3     Choose a cluster E_r as root cluster
4     result ← BTD-MAC+NG (P,∅,E_r,E_r,G,N)
5 until result ≠ unknown
6 return result
```

Théorème 2 *BTD-MAC+RST a une complexité en temps en* $O(R.((n.s^2.e.\log(d) + w^+.N).d^{w^+ +2} + n.(w^+)^2.d))$ *et une complexité en espace en* $O(n.s.d^s + w^+.(d + N))$ *avec* w^+ *la largeur de la décomposition arborescente considérée, s la taille de la plus grande intersection* $E_i \cap E_j$*, R le nombre de redémarrages et N le nombre de nld-nogoods réduits mémorisés.*

Preuve : BTD-MAC sans nld-nogoods a une complexité en temps en $O(n.s^2.e.\log(d).d^{w^+ +2})$. D'après les propositions 4 et 5 de [9], mémoriser et gérer les nld-nogoods de taille au plus n peut s'effectuer respectivement en $O(n^2.d)$ et $O(n.N)$. Comme, d'après le corollaire 1, la taille des nld-nogoods est au plus $w^+ + 1$, ces deux opérations peuvent être accomplies respectivement en $O((w^+)^2.d)$ et $O(w^+.N)$. BTD-MAC+RST réalise au plus R appels à BTD-MAC. Par conséquent, on obtient une complexité en temps pour BTD-MAC+RST en $O(R.((n.s^2.e.\log(d) + w^+.N).d^{w^+ +2} + n.(w^+)^2.d))$.

En exploitant la structure de données proposée dans [9], la complexité en espace pour stocker les nld-nogoods réduits est en $O(w^+.(d+N))$ dans le pire des cas car, selon le corollaire 1, BTD-MAC+RST mémorise N nogoods de taille au plus $w^+ + 1$. Concernant la mémorisation des (no)goods structurels, BTD-MAC+RST possède la même complexité en espace que BTD, à savoir $O(n.s.d^s)$. Donc, au total, la complexité en espace est $O(n.s.d^s + w^+.(d+N))$. □

Si BTD-MAC+RST emploie une politique de redémarrage géométrique [16] basée sur le nombre de retours en arrière autorisés (i.e. un redémarrage se produit dès que le nombre de retours en arrière effectués dépasse le nombre de retours en arrière autorisés qui est initialement fixé à n_0 et augmenté d'un facteur multiplicatif r à chaque redémarrage), nous pouvons majorer le nombre de redémarrage :

Proposition 4 *Etant donnée une politique de redémarrage géométrique basée sur le nombre de retours en arrière avec un nombre de retours en arrière initialement fixé à* n_0 *et un facteur multiplicatif r, le nombre de redémarrage R est majoré par* $\left\lceil \frac{\log(n)+(w^+ +1).\log(d)-\log(n_0)}{\log(r)} \right\rceil$.

Preuve : Dans le pire des cas, le nombre de retours en arrière est majoré par $n.d^{w^+ +1}$ car on a au plus n

clusters et que le nombre de retours en arrière pour un cluster donné est au plus $O(d^{w^+ +1})$. Au $i$ème redémarrage, le nombre de retours en arrière autorisé est $n_0.r^i$. Dans le pire des cas, BTD-MAC+RST termine dès que $n_0.r^i \geq n.d^{w^+ +1}$, i.e. dès que $i \geq \frac{\log(n)+(w^+ +1).\log(d)-\log(n_0)}{\log(r)}$. □

4 Expérimentations

Dans cette section, nous évaluons l'intérêt pratique des redémarrages pour la résolution de CSP grâce à des méthodes par décomposition. Dans ce but, nous comparons BTD-MAC+RST avec BTD-MAC, MAC et MAC+RST+NG sur 647 instances (d'arité quelconque) issues de la compétition de solveurs de 2008. Les instances sélectionnées sont celles qui possèdent des décompositions arborescentes convenables (i.e. celles avec un rapport n/w^+ au moins égal à 2). Les décompositions arborescentes sont calculées avec l'algorithme Min-Fill [13] qui est considéré comme une des meilleures heuristiques de l'état de l'art [5]. Le temps d'exécution de BTD-MAC(+RST) inclut le temps de calcul de la décomposition arborescente. Tous les algorithmes exploitent l'heuristique de choix de variables *dom/wdeg* [2]. Pour le choix du cluster racine, nous avons testé plusieurs heuristiques. Nous présentons ici les deux meilleures :

- RW : on choisit le cluster qui maximise la somme des poids des contraintes dont la portée intersecte le cluster (les poids considérés sont ceux de dom/wdeg). Cette heuristique est aussi utilisée pour BTD-MAC.
- RA : on choisit alternativement soit le cluster qui contient la prochaine variable selon l'heuristique dom/wdeg appliquée à toutes les variables et qui maximise la somme des poids des contraintes dont la portée intersecte le cluster, soit le cluster suivant en classant les clusters dans l'ordre décroissant du rapport nombre de contraintes sur taille du cluster moins un.

Ces deux heuristiques visent à suivre le principe du first-fail. Le second cas de l'heuristique RA permet d'apporter de la diversité à la recherche. Les politiques de redémarrage utilisées reposent toutes sur le nombre de retours en arrière autorisés. Les valeurs présentées ici sont celles qui fournissent les meilleurs résultats parmi les valeurs testées. Plus précisément, pour MAC+RST+NG, nous exploitons une politique géométrique pour laquelle le nombre initial de retours en arrière autorisés est 100 et le facteur multiplicatif 1.1. BTD-MAC+RST avec RW utilise une politique géométrique avec un facteur de 1.1 et initialement 50 retours en arrière sont autorisés. Pour RA, nous appliquons une politique géométrique avec un facteur 1.1

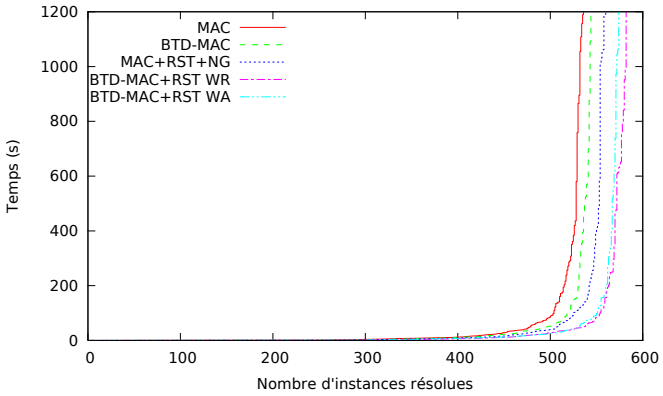

FIGURE 2 – Nombre total d'instances résolues par chaque algorithme.

et un nombre initial de retours en arrière autorisé de 75 quand le cluster est choisi selon la première règle. Dans le cas de la seconde, nous utilisons un nombre constant de retours en arrière autorisés fixé également à 75. Les expérimentations sont effectuées avec nos propres solveurs implémentés en C++, sur un PC basé sur Linux avec un processeur Intel Pentium IV cadencé à 3,2 GHz et 1 Go de mémoire. Le temps d'exécution est limité à 1 200 s (sauf pour la table 1).

La figure 2 présente le nombre total d'instances résolues par chacun des algorithmes considérés. D'abord, nous pouvons noter que, pour BTD-MAC+RST, les deux heuristiques RW et RA se comportent de manière similaire. Ensuite, il apparaît clairement que BTD-MAC+RST résout plus d'instances que les autres algorithmes. Par exemple, BTD-MAC+RST résout 582 instances en 15 863 s avec RW (resp. 574 instances en 13 280 s avec RA) alors que MAC+RST+NG n'en résout que 560 en 16 943 s. Sans les techniques de redémarrage, le nombre d'instances résolues est encore plus faible avec 536 instances en 18 063 s pour MAC et 544 instances en 13 256 s pour BTD-MAC.

Afin de mieux analyser le comportement des différents algorithmes, nous considérons maintenant les résultats obtenus par famille d'instances [4]. La table 2 fournit le nombre d'instances résolues et la somme des temps d'exécution pour ces instances pour chacun des algorithmes considérés tandis que la table 3 présente la somme des temps d'exécution en ne considérant que les instances résolues par tous les algorithmes. Nous pouvons d'abord constater, que pour certains types d'instances, comme les instances de la famille *graph coloring*, l'utilisation de techniques de redémar-

4. Notons que nous ne prenons pas en compte toutes les instances d'une famille, mais seulement celles qui ont une décomposition arborescente adéquate (cf. $n/w^+ \geq 2$).

rage ne permet pas d'améliorer l'efficacité de BTD-MAC+RST par rapport à MAC+RST+NG ou BTD-MAC. Par contre, pour les autres familles considérées, on peut observer que BTD-MAC+RST obtient des résultats intéressants. Ces bons résultats sont parfois dus à la décomposition arborescente (e.g. pour les familles *dubois* ou *haystacks*) car ils sont proches de ceux de BTD-MAC. De même, dans certains cas, ils résultent principalement de l'emploi de techniques de redémarrage (e.g. pour les familles *jobshop* ou *geom*) et ils sont alors voisins de ceux de MAC+RST+NG. Enfin, dans d'autres cas, BTD-MAC+RST tire pleinement partie à la fois de la décomposition arborescente et des techniques de redémarrage (e.g. pour les familles *renault*, *superjobshop* ou *scen11*). Dans de tels cas, il surclasse clairement les trois autres algorithmes. Par exemple, il est deux fois plus rapide que MAC+RST+NG pour résoudre les instances de la famille *scen11*, qui contient les instances d'allocation de fréquence les plus difficiles [3]. La table 1 présente les temps d'exécution de MAC+RST+NG et de BTD-MAC+RST pour ces instances. Nous pouvons remarquer que BTD-MAC ne résout que les trois plus simples. Cela s'explique par un mauvais choix de cluster racine. Il s'avère que, pour toutes les instances de cette famille, la plupart des choix de cluster racine conduisent à passer beaucoup de temps à résoudre certains sous-problèmes. Aussi, les techniques de redémarrages se révèlent ici très utiles.

Enfin, nous avons observé que BTD-MAC+RST est généralement plus efficace sur les instances ne possédant pas de solution que MAC+RST. Par exemple, si on considère les instances dépourvues de solution qui sont résolues par tous les algorithmes, il nécessite 4 260 s pour les résoudre contre 7 105 s pour MAC+RST. Un tel phénomène s'explique en partie par l'utilisation des décompositions arborescentes. En effet, si BTD-MAC+RST explore, au début de la recherche, un cluster sans solution, il peut rapidement conclure que le problème entier n'en a pas non plus.

Instance	MAC+RST+NG	BTD-MAC+RST
scen11-f12	0,51	0,30
scen11-f11	0,50	0,30
scen11-f10	0,65	0,35
scen11-f9	1,32	1,54
scen11-f8	1,60	1,78
scen11-f7	12,93	6,81
scen11-f6	20,23	9,86
scen11-f5	102	45,72
scen11-f4	397	202
scen11-f3	1 277	609
scen11-f2	3 813	1 911
scen11-f1	9 937	5 014

TABLE 1 – Temps de résolution en s (sans limite) pour l'instance scen11.

5 Conclusion

Dans cet article, nous avons d'abord décrit comment intégrer MAC dans BTD. Nous avons ensuite montré comment il est possible d'améliorer les méthodes de résolution exploitant des décompositions en leur ajoutant le concept de redémarrage. Cela nous a conduit à proposer une version étendue de BTD, à savoir la méthode BTD-MAC+RST. Pour cela, nous avons d'abord décrit comment les nogoods classiques peuvent être incorporés dans une méthode de résolution par décomposition tout en préservant la structure induite par la décomposition considérée. Ensuite, nous avons introduit la notion de *good structurel orienté*. En effet, si les nogoods structurels peuvent être directement utilisés quand BTD effectue des redémarrages, les goods doivent vérifier certaines propriétés liées à l'ordre d'exploration d'une décomposition arborescente, d'où la nécessité de les orienter. Dans la dernière partie du papier, les expérimentations ont clairement démontré l'intérêt pratique de l'utilisation des redémarrages dans les méthodes de résolution par décomposition. Cela permet, dans les faits, de dépasser le problème induit par l'ordre d'exploration des clusters, qui, très souvent, nuit de façon significative à l'efficacité pratique. Ces résultats ont également prouvé que l'ajout des redémarrages au sein de BTD permet de surclasser significativement les algorithmes MAC et MAC+RST+NG quand la topologie du réseau de contraintes possède une largeur convenable.

Comme perspectives de ce travail, nous pensons d'abord que des améliorations sont possibles au niveau des politiques de redémarrage qui peuvent être vues sous un nouvel éclairage. Il serait particulièrement intéressant de définir de nouvelles politiques spécifiques au cas des décompositions en considérant des politiques locales ou des combinaisons de politiques locales et globales mais aussi, d'adapter d'autres stratégies comme les redémarrages de Luby [10]. De plus, nous pouvons envisager des choix plus éclairés pour le choix du cluster racine en exploitant des informations plus riches et spécifiques aux décompositions, comme par exemple, le nombre de (no)goods dans un cluster. Enfin, cette approche pourrait être appliquée à un niveau supérieur. Aujourd'hui, nous exploitons des redémarrages pour contourner le problème du choix du cluster racine. Toutefois, l'efficacité des méthodes de résolution par décomposition est également fortement liée à la qualité de la décomposition employée. Aussi, pour contourner le problème du choix d'une décomposition convenable, nous pourrions envisager d'utiliser une nouvelle décomposition à chaque redémarrage. Tout en offrant plus de liberté, cette alter-

Famille	#inst.	MAC		BTD-MAC		MAC+RST+NG		BTD-MAC+RST			
								RW		RA	
		#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps
dubois	13	5	2 232	13	0,03	5	2 275	13	0,04	13	0,05
geom	83	83	415	83	819	83	479	83	468	83	460
graphColoring	39	29	1 989	33	1 291	29	2 783	34	2 825	33	2 769
haystacks	46	2	5,82	8	169	2	4,43	8	172	8	172
jobshop	46	37	617	35	469	46	14,87	46	13,15	46	10,93
renault	50	50	23,89	50	86,81	50	24,30	50	22,96	50	24,73
pret	8	4	250	8	0,05	4	552	8	0,06	8	0,05
scens11	12	8	1 632	3	1,25	9	537	10	878	10	882
Super-jobShop	46	19	1 648	21	1 179	33	2 315	34	1 553	27	449
travellingSalesman-20	15	15	191	15	229	15	214	15	346	15	294

TABLE 2 – Nombre total d'instances résolues et temps total de résolution en s par famille pour chaque algorithme.

Famille	#inst.	MAC	BTD-MAC	MAC+RST+NG	BTD-MAC+RST	
					RW	RA
dubois	5 / 13	2 232	0,01	2 275	0,01	0,01
graphColoring	27 / 39	951	1 051	1 308	846	1 277
haystacks	2 / 46	5,82	0	4,43	0	0,01
jobshop	33 / 46	392	468	5,63	5,10	4,48
pret	4 / 8	250	0,01	552	0,02	0
rlfapScens11	3 / 12	2,75	1,25	1,66	0,95	1,10
Super-jobShop	16 / 46	1 275	830	14,83	9,60	16,04

TABLE 3 – Temps de résolution en s pour chaque algorithme pour les instances résolues par tous les algorithmes.

native engendre des questions plus complexes car les nogoods, structurels ou non, et les goods ne seraient utilisables qu'en respectant des conditions encore plus restrictives.

Références

[1] C. Berge. *Graphs and Hypergraphs*. Elsevier, 1973.

[2] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[3] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

[4] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[5] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *CP*, pages 777–781, 2005.

[6] P. Jégou, S.N. Ndiaye, and C. Terrioux. 'Dynamic Heuristics for Backtrack Search on Tree-Decomposition of CSPs. In *IJCAI*, pages 112–117, 2007.

[7] P. Jégou, S.N. Ndiaye, and C. Terrioux. Dynamic Management of Heuristics for Solving Structured CSPs. In *CP*, pages 364–378, 2007.

[8] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[9] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal. Recording and Minimizing Nogoods from Restarts. *JSAT*, 1(3-4) :147–167, 2007.

[10] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47(4) :173–180, 1993.

[11] J. Petke. *On the bridge between Constraint Satisfaction and Boolean Satisfiability*. PhD thesis, University of Oxford, 2012.

[12] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[13] D. J. Rose. A graph theoretic study of the numerical solution of sparse positive denite systems of linear equations. In *Graph Theory and Computing*, pages 183–217. R.C. Read (ed.), Academic Press, New York, 1973.

[14] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[15] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *ECAI*, pages 125–129, 1994.

[16] T. Walsh. Search in a small world. In *IJCAI*, pages 1172–1177, 1999.