Intégration d'une approche structurelle dans un cadre hybride pour la résolution de CSP

Loïc Blet, Samba Ndojh Ndiaye, Christine Solnon

Intégration d'une approche structurelle dans un cadre hybride pour la résolution de CSP

L. Blet[1,3] S. N. Ndiaye[1,2] C. Solnon[1,3]

[1] Université de Lyon - LIRIS
[2] Université Lyon 1, LIRIS, UMR5205, F-69622 France
[3] INSA-Lyon, LIRIS, UMR5205, F-69621, France

{loic.blet,samba-ndojh.ndiaye,christine.solnon}@liris.cnrs.fr

Résumé

[PV04] a proposé un cadre générique d'algorithmes permettant de combiner des techniques issues des approches complètes et incomplètes pour la résolution de problèmes de satisfaction de contraintes. Les auteurs ont défini une instanciation de ce cadre, DR(minDestroy), dont les premiers résultats sont très prometteurs. Cette instanciation, qui réalise une recherche locale dans l'ensemble des affectations partielles cohérentes, est capable de prouver l'incohérence de certains problèmes malgré une recherche a priori incomplète. Nous nous proposons d'étendre ce cadre et cette instanciation en y intégrant l'exploitation de la structure du problème. L'évaluation expérimentale menée a démontré un gain significatif sur des instances de CSP structurées.

Mots Clef

Satisfaction de contraintes, Graphe de contraintes, Décomposition arborescente, Recherche complète, Recherche locale.

Abstract

[PV04] proposed a generic framework allowing algorithms to combine techniques from both complete and incomplete methods for solving constraint satisfaction problems. The authors defined an instantiation of this framework, DR(minDestroy). Its first results are very promising. This instantiation performs a local search in the set of consistent partial assignments. It is able to prove the inconsistency of some problems even if its search is supposedly incomplete. We propose to extend this framework and this instantiation by integrating the use of the structure of the problem. Experimental evaluation shows a notable gain on structured CSP instances.

Keywords

Constraint satisfaction, Constraint graph, Tree-decomposition, Complete search, Local search.

1 Introduction

Les contraintes font partie de notre vie quotidienne, qu'il s'agisse par exemple de faire un emploi du temps, de remplir un camion de déménagement avec des cartons de tailles diverses, ou encore de planifier un trafic aérien. Un problème de satisfaction de contraintes (CSP) est défini par un ensemble fini de variables, chaque variable devant être affectée à une valeur de son domaine tout en respectant un ensemble de contraintes, qui restreignent les valeurs que peuvent prendre simultanément les variables.

De façon plus formelle, un problème de satisfaction de contraintes est défini par un triplet (X, D, C) tel que :

- X est un ensemble fini de variables ;
- D est une fonction qui associe à chaque variable $x_i \in X$ son domaine $D(x_i)$, c'est-à-dire l'ensemble fini des valeurs que peut prendre x_i ;
- C est un ensemble fini de contraintes. Chaque contrainte $c_j \in C$ est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.

Une *affectation* A attribue une valeur à des variables. Une affectation est dite *totale* si elle instancie toutes les variables du problème, elle est dite *partielle* si elle n'en instancie qu'une partie. Une affectation (totale ou partielle) est *cohérente* si elle ne viole aucune contrainte et *incohérente* si elle viole au moins une contrainte. Une *solution* est une affectation totale cohérente.

Ce problème est NP-difficile dans le cas général. Il existe deux grandes approches de résolution de CSP. D'une part, l'approche complète explore exhaustivement l'ensemble de toutes les affectations possibles jusqu'à trouver une solution ou prouver l'absence de solution. La méthode standard de cette approche est le backtracking (BT) qui construit de manière incrémentale une affectation tout en vérifiant à chaque étape si cette affectation est cohérente, c'est-à-dire respecte toutes les contraintes. La complexité en temps de BT est exponentielle en n le nombre de variables. Elle souffre du grand nombre de redondances

qu'elle génère. Plusieurs améliorations de cette méthode ont été proposées afin de réduire l'espace de recherche à travers des techniques de filtrage basées sur des cohérences locales [Wal75, Fre78] et des techniques de retour en arrière intelligent [Gas79, Dec90, Pro95]. Cela a conduit à des méthodes beaucoup plus efficaces telles que Forward Checking (FC[HE80]) et Maintaining Arc Consistency (MAC[SF94]). Malgré tout, les complexités de ces méthodes demeurent exponentielles en n. Une autre amélioration consiste à utiliser la structure du problème pour assurer des garanties théoriques fortes sur la complexité en temps, mais aussi rendre plus efficace la résolution pratique [FQ85, DP87, DP89, GLS00, Dar01, JT03, DM07].

Malgré ces améliorations, l'approche complète souffre de la taille de l'espace de recherche (d^n, d étant la taille maximale des domaines des variables). En effet, dans le cas de problèmes contenant un nombre important de variables, la durée de résolution par cette approche est trop grande. L'approche incomplète consiste à chercher une solution le plus rapidement possible (avec une complexité en temps polynomiale) en explorant les zones de l'espace de recherche les plus prometteuses [SKC94, LA87, GL97, MH97]. Cependant, elle est en général incapable de prouver l'incohérence d'un CSP si elle ne trouve pas une solution. Il existe différentes façons d'explorer l'espace de recherche de façon incomplète, la plus connue étant la recherche locale : partant d'une affectation complète incohérente, la recherche locale modifie itérativement cette affectation en changeant les valeurs de quelques variables, jusqu'à trouver une affectation cohérente.

[PV04] propose un cadre Generic Decision Repair [JL02] pour définir des méthodes hybrides qui utilisent à la fois des techniques issues de l'approche complète et de l'approche incomplète. Il permet de structurer la recherche à l'image de l'approche complète dans la construction d'une affectation partielle cohérente et de réduire cet espace grâce au filtrage des valeurs incompatibles avec cette affectation. Il autorise dans le même temps et à l'image de l'approche incomplète, une plus grande liberté dans la réparation d'une affectation ne pouvant mener à solution en modifiant la valeur d'une variable. Ce cadre permet l'exploration d'un grand nombre de combinaisons entre des techniques issues des approches complètes et incomplètes. Les auteurs ont défini une instanciation d'algorithme DR(minDestroy) qui réalise une recherche locale dans l'ensemble des affectations partielles consistantes. Malgré une recherche a priori incomplète, DR(minDestroy) est capable de prouver l'incohérence de certains problèmes grâce à l'enregistrement des justifications des échecs dans l'extension d'une affectation donnée. Elle a obtenu des premiers résultats très satisfaisants. Néanmoins, elle n'intègre pas l'exploitation de la structure du CSP qui pourrait permettre d'en améliorer la résolution. Le but de cet article est d'intégrer l'exploitation de la structure. Pour cela, nous allons utiliser la méthode BTD [JT03] qui facilite cette exploitation à travers un ordre d'affectation des variables donné par la structure du problème et l'enregistrement de (no)goods permettant de réduire les redondances. Par ailleurs, les bons résultats théoriques et pratiques de BTD justifient l'utilisation de cette approche.
La prochaine section présente le formalisme CSP, le cadre Generic Decision Repair et la méthode BTD. Dans la section suivante, nous définissons une paramétrisation du cadre générique qui capture BTD et une extension de cette dernière qui intègre la recherche locale. Enfin, nous présentons une étude expérimentale évaluant l'intérêt de l'approche.

2 Préliminaires

2.1 Un cadre générique pour la résolution de CSP

Pour proposer une nouvelle méthode pour résoudre les CSP il est intéressant de savoir dans quel cadre s'intègre cette méthode. Nous présentons ici un cadre générique qui englobe toutes les méthodes présentées dans cet article.
[PV04] propose un cadre d'algorithmes de résolution de CSP (voir algorithme 1).

Algorithme 1 : Generic Decision Repair

Entrées : Un CSP P et A une affectation initiale
1 **répéter**
2 cohérent ← Propager(P, A)
3 **si** *cohérent* **alors**
4 $r \leftarrow$ ÉtendreAffectation(P, A)
5 **si** $r = 1$ **alors**
6 **retourner** $vrai$
7 **sinon**
8 $r \leftarrow$ RéparerAffectation(P, A)
9 **si** $r = 0$ **alors**
10 **retourner** $faux$
11 **jusqu'à** *Arrêt()* ;
12 **retourner** ?

Il contient plusieurs fonctions paramétrables :
Propager : permet de filtrer les valeurs incompatibles avec l'affectation courante en utilisant un niveau de cohérence locale afin de détecter une incohérence éventuelle.
ÉtendreAffectation : permet d'étendre une affectation cohérente avec l'instanciation d'une nouvelle variable.
RéparerAffectation : modifie la valeur d'une variable dans une affectation qui ne peut mener à une solution.
Une bonne définition des fonctions paramétrables permet de capturer des méthodes complètes (BT, FC, MAC...) mais aussi des méthodes incomplètes (Incomplete Dynamic Backtracking [Pre02], Min conflicts [MPJL92]...). Ainsi, ce cadre nous offre ainsi un vaste espace de méthodes de résolution à explorer.
Dans [PV04], les auteurs proposent une instance de ce cadre, la méthode Decision Repair(minDestroy) (notée DR(minDestroy)). Pour cette dernière :

Propager est basée sur la technique du FC qui filtre les valeurs incompatibles avec l'affectation courante dans les domaines des variables non instanciées.

ÉtendreAffectation choisit la prochaine variable selon l'heuristique min domaine/degré qui choisit comme prochaine variable à affecter une qui minimise le ratio entre la taille du domaine courant et le nombre de contraintes portant sur cette variable.

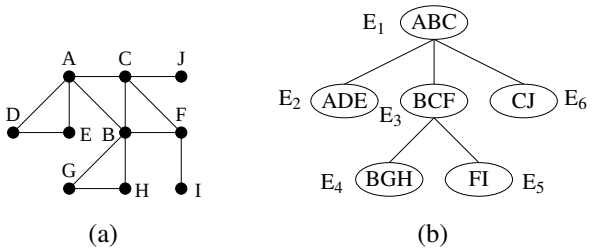
RéparerAffectation désaffecte une variable parmi celles qui sont en cause (au sens de CBJ [Pro95]) dans l'échec actuel et qui soit le moins en cause dans le filtrage de valeurs dans des variables non instanciées (cette heuristique (minDestroy) permet de garder le plus d'informations possibles dans l'optique de prouver l'incohérence du CSP).

DR(minDestroy) intègre une structure de gestion des variables en cause dans un échec (les justifications de cet échec) lui permettant au sein de la fonction RéparerAffectation de choisir la variable à désaffecter. Cette structure lui permet aussi de détecter l'incohérence de certaines parties de l'espace de recherche voire même du problème dans sa globalité. De ce fait, même s'il n'y a pas de garantie d'exploration exhaustive de l'espace recherche, la survenue d'un échec avec un ensemble de justifications vide permet de conclure à l'incohérence du CSP. Ceci constitue une originalité majeure pour une méthode de résolution a priori incomplète. En outre, cet algorithme a obtenu des résultats pratiques très satisfaisants. Et, nous cherchons ici à le rendre plus performant encore en y intégrant une exploitation de la structure à l'image de la méthode BTD.

2.2 L'algorithme BTD [JT03]

Dans le cas d'un CSP binaire où les contraintes lient au plus deux variables, la structure du problème est capturée notamment par le graphe des contraintes $G_P = (X, A)$ dont les sommets représentent les variables du CSP et les arêtes représentent les contraintes (i.e. il existe une arête entre deux sommets, s'il existe une contrainte qui lie les deux variables correspondantes). Dans le cas général, les contraintes peuvent lier plus de deux variables donc la structure est capturée par un hypergraphe de contraintes dont les sommets restent les variables du CSP et les hyperarêtes les contraintes (c'est-à-dire il existe une hyperarête contenant un ensemble de sommets ssi les variables correspondantes sont liées par une contrainte). Le cas binaire étant un cas particulier du cas général, nous nous plaçons directement dans ce dernier où BTD est basée sur une décomposition arborescente de la 2-section de l'hypergraphe de contraintes du CSP. La 2-section d'un hypergraphe $H = (X, B)$ est un graphe $G = (X, A)$ tel qu'il existe une arête entre deux sommets de G ssi ces deux sommets figurent ensemble dans une hyperarête de H.

Définition 1. *[RS86] Une* décomposition arborescente *d'un graphe $G = (S, A)$ est une paire (T, E), où $T = (N, F)$ est un arbre, et E est une fonction d'étiquetage associant à chaque sommet $p \in N$ un ensemble de sommets*



(a) (b)

FIGURE 1 – (a) Un graphe. (b) Une décomposition arborescente de ce graphe.

(appelé cluster) $E(p) \subseteq S$, tels que :

1. $\forall v \in S, \exists p \in N, v \in E(p)$;
2. $\forall (v, u) \in A, \exists p \in N, \{v, u\} \subseteq E(p)$;
3. *Pour tout sommet $v \in S$, l'ensemble $\{p \in N | v \in E(p)\}$ induit un sous-graphe connexe de T.*

La *largeur* d'une décomposition est la taille du plus grand ensemble $E(p)$ moins un. La *largeur d'arbre* du graphe G est la plus petite parmi celles de toutes les décompositions possibles de G. Une décomposition arborescente est optimale si sa largeur correspond à la largeur d'arbre du graphe considéré.

La figure 1 présente un graphe et une décomposition arborescente de ce graphe.

Une décomposition arborescente détecte l'indépendance entre différentes parties du problème afin de les résoudre séparément et donc plus efficacement. Cette résolution passe par un ordre d'affectation des variables compatible avec la décomposition, mais aussi par l'enregistrement de la cohérence ou de l'incohérence des sous-problèmes une fois que cela a été déterminé. Pour cela, BTD commence par choisir le cluster contenant le plus de variables comme racine pour débuter la recherche et construire de ce fait une arborescence de clusters. Les fils d'un cluster sont ordonnés par taille d'intersection croissante avec le cluster père [JNT07]. Sur l'exemple de la figure 1, E_1 est la racine de l'arborescence. Le problème enraciné en ce cluster racine contient la totalité du CSP. BTD affecte d'abord les variables contenues dans E_1, avant de choisir un sous-problème à résoudre, effectuant un parcours en profondeur de l'arborescence. En effet, chaque cluster fils du cluster racine (E_2, E_3 ou E_6) constitue à son tour la racine d'un sous-problème. Ce sous-problème est défini par l'ensemble des variables et des contraintes contenues dans le sous-arbre de clusters mais aussi par la valeur des variables du séparateur avec le cluster père (les variables contenues à la fois dans E_1 et dans le cluster fils). Tous ces sous-problèmes étant indépendants, ils peuvent être résolus séparément. Si le sous-problème enraciné en E_2 est cohérent, BTD enregistre ce *good structurel* qui lui permettra à l'avenir de ne pas le résoudre à nouveau. Par contre, si le sous-problème est incohérent, BTD enregistre un *nogood structurel*. Un (no)good structurel par rapport à un sous-

problème enraciné en un cluster E_i est défini par l'affectation des variables contenues dans le séparateur avec le cluster père de E_i.

La réduction du nombre de redondances assure une complexité en temps de BTD exponentielle en la largeur de la décomposition, alors que sa complexité en espace est exponentielle en la taille maximum des séparateurs entre clusters.

3 Extension du cadre Generic Decision Repair à l'exploitation de la structure

3.1 Capture de BTD dans le cadre Generic Decision Repair

Pour capturer la méthode BTD dans le cadre Generic Decision Repair, il faut tout d'abord y intégrer une décomposition arborescente du graphe de contraintes du CSP, mais aussi définir de manière adéquate les fonctions ÉtendreAffectation et RéparerAffectation.

L'ordre d'affectation de la fonction ÉtendreAffectation doit être compatible avec la décomposition. En effet, il faut pouvoir affecter les variables du cluster racine avant de choisir un des sous-problèmes enracinés en un des clusters fils de la racine pour ensuite résoudre de manière récursive ce sous-problème. Quant à la fonction RéparerAffectation, elle doit tout simplement revenir à la dernière variable instanciée au sein du cluster courant voire au sein du cluster père de celui-ci. A ce stade, on se rend compte qu'il manque à ce cadre la gestion de l'apprentissage de (no)goods. Nous faisons le choix d'intégrer celle-ci à ces deux fonctions.

Ainsi, ÉtendreAffectation (voir algorithme 2) choisit la prochaine variable à instancier dans le cluster courant E_i. Si toutes les variables de E_i sont déjà affectées, elle vérifie si les sous-problèmes éventuellement générés (enracinés au niveau des fils éventuels de E_i) ont déjà été résolus à travers la liste des (no)goods. Si un de ces sous-problèmes est connu comme étant incohérent, ÉtendreAffectation met la variable *cohérent* à faux afin de modifier l'affectation courante (qui ne peut mener à une solution) avec la fonction RéparerAffectation. Si par contre, aucun des sous-problèmes résolus n'est incohérent, alors elle étend l'affectation au cluster racine d'un sous-problème non résolu. Dans le cas où tous les sous-problèmes ont déjà été résolus et prouvés cohérents, alors le sous-problème enraciné en E_i est cohérent et ÉtendreAffectation enregistre ce good entre E_i et son cluster père. Puis, elle revient à ce dernier pour vérifier s'il admet d'autres sous-problèmes non encore résolus. Si tous les sous-problèmes associés au cluster racine (qui est déjà affecté) sont prouvés cohérents, alors le CSP est cohérent et la fonction ÉtendreAffectation retourne la valeur 1.

En cas d'incohérence, la fonction RéparerAffectation (voir algorithme 3) va permettre de modifier l'affectation courante en modifiant l'instanciation d'une variable du cluster courant E_i. Si toutes les variables de E_i ont déjà été désinstanciées, elle passe à son cluster père et enregistre au passage un nogood traduisant que le sous-problème enraciné en E_i est incohérent. En effet, avant de remonter au cluster père l'algorithme de résolution a tenté d'étendre l'affectation courante dans ce sous-problème de toutes les manières possibles sans succès. Si un échec persiste malgré le fait que toutes les variables du cluster racine sont désaffectées, cela prouve que le problème est globalement incohérent et la fonction RéparerAffectation renvoie la valeur 0.

Avec ces définitions de ces deux fonctions, nous capturons la méthode BTD dans le cadre Generic Decision Repair. Ce dernier permettant aisément de combiner des techniques issues des approches complètes et incomplètes, nous allons proposer une extension de BTD à la recherche locale, ce

Algorithme 2 : ÉtendreAffectation(P, A, E_i)

Entrées : P le CSP, A l'affectation courante, E_i le cluster courant

```
1  si Toutes les variables de E_i sont instanciées alors
2      si E_i admet un sous-problème incohérent (un nogood) alors
3          retourner 0
4      sinon
5          si E_i admet un sous-problème non résolu alors
6              Passer au cluster racine de ce sous-problème
7          sinon
8              si E_i est le cluster racine de P alors
9                  retourner 1
10             sinon
11                 Enregistrer un good entre E_i et son cluster père;
12                 Revenir au cluster père
13 sinon
14     Étendre A à une variable non encore instanciée de E_i
15 retourner −1
```

Algorithme 3 : RéparerAffectation(P, A, E_i)

Entrées : P le CSP, A l'affectation courante, E_i le cluster courant

```
1  si Aucune variable de E_i n'est instanciée alors
2      si E_i est le cluster racine de P alors
3          retourner 0
4      sinon
5          Enregistrer un nogood entre E_i et son cluster père;
6          Remonter au cluster père;
7  sinon
8      Changer l'affectation de la dernière variable affectée dans E_i;
9  retourner −1
```

Algorithme 4 : RéparerAffectation2(P, A, E_i)

Entrées : P le CSP, A l'affectation courante, E_i le cluster courant

```
si Aucune variable de E_i n'est en cause dans l'échec alors
    si E_i est le cluster racine de P alors
        retourner 0
    sinon
        Enregistrer un nogood entre E_i et son cluster père;
        Remonter au cluster père
sinon
    Changer l'affectation d'une variable de E_i en cause dans l'échec
retourner −1
```

qui, à notre connaissance, n'a encore jamais été réalisé.

3.2 Une extension de BTD à la recherche locale

Cette extension libère l'ordre de désaffectation des variables en passant d'un ordre chronologique, au choix d'une variable parmi toutes celles du cluster courant qui sont en cause dans l'échec actuel. Pour cela, nous modifions la fonction RéparerAffectation précédente. Au lieu de remonter à la dernière variable affectée dans le cluster courant voire dans son cluster père, nous allons remonter directement à une variable en cause dans l'échec toujours au sein du cluster (voir algorithme 4). Par ailleurs, le choix de la variable à désinstancier parmi celles du cluster courant qui sont en cause dans l'échec est totalement libre. Dans le cas où les raisons de l'échec ne se trouvent pas dans le cluster courant, mais dans son cluster père par exemple, la preuve est faite que le sous-problème enraciné en E_i est incohérent et RéparerAffectation2 enregistre un nogood entre E_i et son cluster père. Puis, elle remonte à ce cluster pour faire le choix de la variable à désaffecter parmi celles qui sont responsables de l'échec.
A l'image de DR(minDestroy), nous avons testé dans la partie expérimentale une version de cette fonction qui fait le choix de désinstancier une variable qui permettra de conserver le plus d'informations possibles sur les justifications des filtrages de valeurs. Ainsi, il sera plus aisé d'élaguer l'espace de recherche voire même de faire la preuve d'incohérence d'un problème.

4 Évaluation expérimentale

Nous avons comparé plusieurs algorithmes qui utilisent différemment la structure des CSP et diffèrent dans la rigidité de leur recherche. Nous décrivons ici les CSP que nous générons et les conditions des expérimentations.

Génération de CSP structurés Nous utilisons ici la même méthode que dans [JNT06], qui produit des instances avec un graphe des contraintes structuré (graphe dont la largeur d'arbre est beaucoup plus petite que le nombre de variables). Elle commence par générer un arbre de cliques contenant les n variables du CSP (chaque variable ayant un domaine de taille d) et ns cliques dont les tailles sont bornées par $w + 1$ et celles des intersections entre cliques par s. Ensuite, elle crée une contrainte liant toute paire de variables apparaissant dans une même clique et qui interdit t tuples. Enfin, elle retire un pourcentage p des contraintes du CSP préalablement construit. Cette méthode assure que la largeur arborescente du graphe de contraintes du CSP ainsi généré est bornée par w. Une classe d'instances est donc définie par la liste de ses paramètres (n, d, w, t, s, ns, p).
Nous avons généré 14 classes d'instances (voir tableau 1), chacune contenant 20 instances. Le tableau 1 récapitule les paramètres des classes. La difficulté dépend de la taille de l'instance (le nombre de variables et le nombre de valeurs dans les domaines), mais aussi et surtout de la dureté des contraintes, soit le nombre de tuples autorisés par contraintes. Lorsque la dureté est faible (resp. forte), le problème est généralement trivialement cohérent (resp. incohérent). Les instances les plus difficiles se trouvent dans la zone de transition de phase pour laquelle la dureté est intermédiaire. Nous avons essayer d'y situer nos classes d'instances.

Algorithmes testés Nous considérons les algorithmes suivants : FC [HE80], FC-BTD [JT03], DR(minDestroy) (noté DRmin [PV04]), DRminBTD (voir la partie 3.2) et FC-CBJ-BTD (Utilisation de CBJ [Pro95] à l'intérieur du cluster courant dans FC-CBJ).
Les algorithmes testés sont implémentés en C. La machine utilisée avait les caractéristiques suivantes : Intel(R) Xeon(R) CPU E5520 2.27GHz 64 bits, cache 8192 KB, RAM 16 Go.
Pour chaque algorithme et chaque classe, nous comparons le nombre d'instances résolues (en moins de 30 minutes).

Résultats expérimentaux Nous commençons par comparer DRmin et DRminBTD sur les 14 classes mélangées (voir figures 2, 3, 4 et 5). Nous notons de bien meilleures performances pour DRminBTD. Particulièrement pour prouver l'incohérence de CSP, l'utilisation de la structure est bénéfique. Cela vient de l'information supplémentaire qui est enregistrée en se basant sur la décomposition et qui permet de réduire de manière drastique les redondances.
Dans un second temps, nous avons cherché à évaluer le comportement de DRminBTD quand la structure du CSP est de moindre qualité (de grandes valeurs de w par rapport à n). Pour cela nous comparons DRmin et DRminBTD sur 4 classes de CSP de moins en moins structurés, la classe A étant celle contenant les CSP les plus structurés. Pour chaque classe nous générons 20 instances de CSP. Nous pouvons voir dans le tableau 2 que le comportement de DRminBTD tend vers celui de DRmin quand la qualité de la structure se dégrade. Malgré le coût engendré par la gestion de la décomposition, les performances de DRminBTD restent comparables à celles de DRmin quand le problème n'est pas structuré. Ainsi, DRminBTD est capable de tirer

classe	n	#D	%TI	#MC
1	150	25	34	15
2	150	25	38	15
3	150	25	41	15
4	150	25	46	15
5	250	20	27	20
6	250	20	29	20
7	250	20	32	20
8	250	20	37	20
9	250	25	34	15
10	250	25	37	15
11	250	25	40	15
12	250	25	45	15
13	500	20	30	15
14	500	20	34	15
A	80	10	46	10
B	80	10	30	20
C	80	10	23	30
D	80	10	17	40

Tableau 1 – Les classes de problèmes structurés générés. Chaque ligne donne successivement : le numéro de la classe, le nombre de variables (#X), la taille des domaines (#D), le pourcentage de tuples interdits par contrainte (%TI), la taille de la plus grande clique de la décomposition (#MC).

profit de la structure du CSP pour une résolution très efficace, mais ne ralentit pas cette résolution si la structure est de qualité moindre.

classe	DRmin		DRminBTD	
	C	I	C	I
A	5	3	**6**	**9**
B	6	**1**	**7**	**1**
C	**3**	0	**3**	**4**
D	4	**0**	**6**	**0**

Tableau 2 – Influence de la structure des CSP sur DRmin et DRminBTD. Nombre de problèmes trouvés cohérents (C), incohérents (I) sur 20 problèmes, le reste des instances ayant été arrêtées après 30 minutes

Pour mieux apprécier le comportement de DRminBTD nous le comparons à FC et FC-BTD sur les premières classes de CSP structurés que nous avons générés (voir figures 2, 3, 4 et 5). Nous notons que FC n'est pas compétitif face aux deux autres algorithmes sur ces instances. En effet, ces instances sont vraiment difficiles et l'absence d'exploitation de leur structure rend leur résolution très complexe (très longue). L'utilisation de la structure dans les autres approches améliore nettement leur efficacité. On voit par ailleurs qu'elles ont des résultats similaires. Néanmoins, FC-BTD se comporte mieux sur les instances incohérentes, alors que DRminBTD est plus efficace sur les cohérentes. Nous allons essayer de relier ce constat avec le nombre de nœuds visités et la durée d'exécution des algorithmes (voir figures 2, 3, 4 et 5).
Il apparaît clairement que FC-BTD développe bien plus de nœuds que DRminBTD pour résoudre un problème. Ce ré-

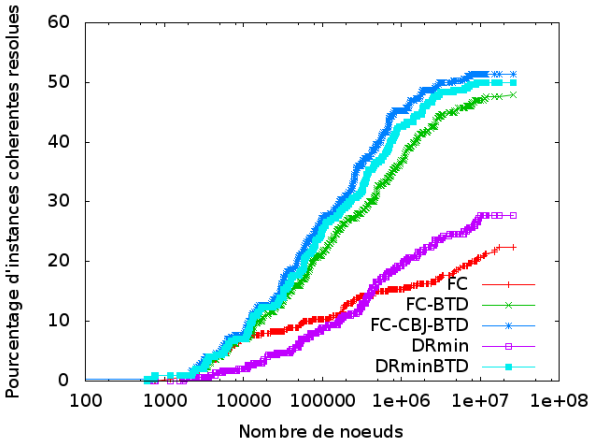


FIGURE 2 – Nombre de nœuds développés pour résoudre les problèmes trouvés cohérents

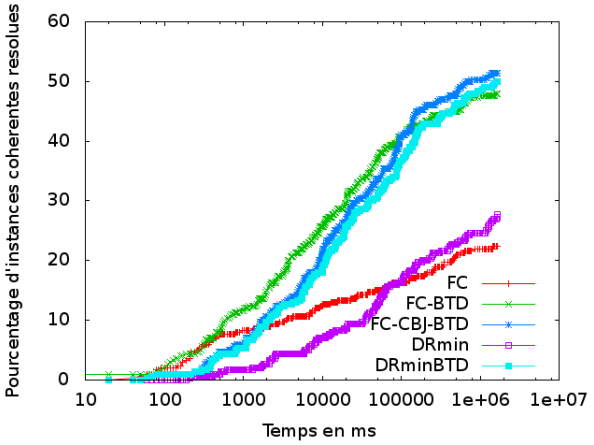


FIGURE 3 – Temps passé pour résoudre les problèmes trouvés cohérents

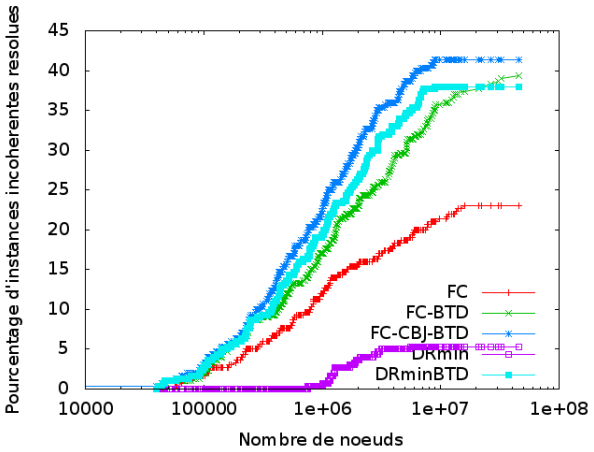


FIGURE 4 – Nombre de nœuds développés pour résoudre les problèmes trouvés incohérents

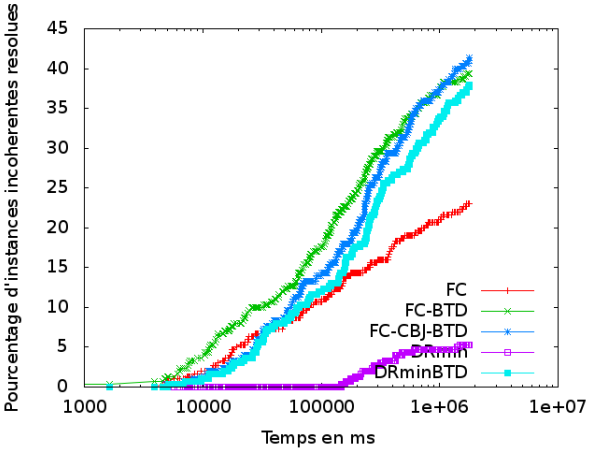


FIGURE 5 – Temps passé pour résoudre les problèmes trouvés incohérents

sultat est important car il permet de voir l'impact très positif de la recherche locale. Cependant, FC-BTD est plus rapide au final, même si on peut imaginer, avec les figures 3 et 5, que sur des problèmes plus difficiles DRminBTD surpasse FC-BTD en rapidité. Néanmoins, le nombre de tests de cohérence est beaucoup plus important dans DRminBTD du fait de la gestion des justifications des échecs. Le coût de cette gestion semble très important au niveau de chaque nœud développé. Il est donc important d'avoir une grande réduction du nombre de nœuds par rapport à FC-BTD pour assurer un temps de résolution meilleur.
La différence d'efficacité sur les instances cohérentes et incohérentes s'explique également par les orientations des deux approches. En effet, FC-BTD repose sur une recherche systématique plus apte à prouver l'incohérence grâce à l'utilisation du principe du "first fail" qui consiste à faire les choix les plus contraints d'abord. Cela permet de détecter le plus rapidement possible les incohérences dans la partie de l'espace de recherche en cours d'exploration. Par contre DRminBTD repose sur la recherche locale dans l'espace des affectations partielles qui est a priori incomplète. Même si le choix de la variable à désaffecter est faite de telle sorte qu'on puisse faire une preuve éventuelle d'incohérence, le principe de cette recherche reste malgré tout de tendre vers une solution au plus vite.
Par ailleurs l'algorithme FC-CBJ-BTD a obtenu les meilleures performances dans nos tests. Nous montrons dans les figures 2, 3, 4 et 5 une comparaison de ses résultats avec les autres algorithmes.

5 Conclusion

Nous avons proposé une extension de la méthode de résolution de CSP DR(minDestroy) [PV04] qui accomplit une recherche locale dans l'espace des affectations partielles. Cette extension intègre l'exploitation de la structure du problème pour accroître l'efficacité de sa résolution. Nous avons mené une étude expérimentale dont les résultats traduisent une amélioration considérable des performances pratiques de DR(minDestroy) dans le cas d'instances de CSP structurées. Malgré tout, une comparaison avec l'état de l'art confirme l'intérêt de l'approche au niveau du nombre de nœuds visités, mais révèle que le coût de gestion des justifications des échecs devra être réduit pour accroître le gain en temps.
Par ailleurs, il serait intéressant d'assouplir encore la recherche de DRminBTD, tout en gardant la gestion de la structure du CSP. Par exemple, nous pourrions désaffecter une variable en cause dans un échec sans nous limiter au cluster courant. Il serait aussi utile de définir une version de DRminBTD en modifiant l'enregistrement et l'effacement des justifications afin de garantir la complétude et des garanties sur la complexité en temps.

Références

[Dar01] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126 :5–41, 2001.

[Dec90] R. Dechter. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[DM07] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171 :73–106, 2007.

[DP87] R. Dechter and J. Pearl. The Cycle-cutset method for Improving Search Performance in AI Applications. In *Proceedings of the third IEEE on Artificial Intelligence Applications*, pages 224–230, 1987.

[DP89] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[FQ85] E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 1076–1078, 1985.

[Fre78] E. Freuder. Synthesizing constraint expressions. *CACM*, 21(11) :958–966, 1978.

[Gas79] J. Gaschnig. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :343–282, 2000.

[HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3) :263–313, 1980.

[JL02] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1) :21–45, 2002.

[JNT06] P. Jégou, S. N. Ndiaye, and C. Terrioux. Strategies and Heuristics for Exploiting Tree-decompositions of Constraint Networks. In *Inference methods based on graphical structures of knowledge (WIGSK'06), ECAI workshop*, pages 13–18, 2006.

[JNT07] P. Jégou, S.N. Ndiaye, and C. Terrioux. Dynamic heuristics for backtrack search on tree-decomposition of csps. In *IJCAI*, pages 112–117, 2007.

[JT03] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artif. Intell.*, 146 :43–75, May 2003.

[LA87] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing : theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[MH97] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers & OR*, 24(11) :1097–1100, 1997.

[MPJL92] S. Minton, A. Philips, M. D. Johnston, and P. Laird. Minimizing conflicts : A heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58 :241–249, 1992.

[Pre02] S. Prestwich. Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research*, 115 :51–72, 2002.

[Pro95] P. Prosser. Forward checking with backmarking. In *Constraint Processing, Selected Papers*, pages 185–204, London, UK, 1995. Springer-Verlag.

[PV04] C. Pralet and G. Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *CPAIOR*, pages 240–255, 2004.

[RS86] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[SF94] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *PPCP*, pages 10–20, 1994.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 337 – 343. MIT Press, 1994.

[Wal75] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. P. H. Winston, McGraw–Hill, New York, 1975.