Recherche arborescente bornée

Philippe Jégou
LSIS-CNRS
Av. Escadrille Normandie-Niemen
13397 Marseille Cedex 20
email : `philippe.jegou@univ.u-3mrs.fr`

Cyril Terrioux
LSIS-CNRS
39 rue Joliot-Curie
13453 Marseille Cedex 13
email : `cyril.terrioux@univ.u-3mrs.fr`

Résumé

Nous proposons une méthode pour la résolution de CSPs basée à la fois sur les techniques de recherche arborescente et sur la notion de décomposition arborescente du réseau de contraintes. Cette approche mixte nous permet de définir un cadre pour l'énumération dont on espère qu'il bénéficiera d'une part des avantages des techniques d'énumération pour leur efficacité pratique, et d'autre part, des garanties en termes de complexité théorique qu'offrent les méthodes de décomposition de CSPs. Les résultats expérimentaux que nous avons obtenus nous ont permis de nous assurer de l'intérêt pratique de cette approche.

1 Introduction

Dans cet article, nous considérons le formalisme CSP classique basé sur la donnée d'un ensemble X de variables x_1,x_2, $\ldots x_n$, qui doivent être affectées dans leur domaine fini respectif D_i, en satisfaisant un ensemble C de contraintes qui expriment des restrictions sur les différentes affectations possibles. Une solution est une affectation de chaque variable qui satisfait toutes les contraintes. Il est bien connu que ce problème est NP-complet.

La méthode de base pour la résolution de CSPs est fondée sur l'énumération de type "backtracking", qui pour être efficace doit s'appuyer sur les techniques de filtrage et sur l'exploitation d'heuristiques pour le choix des variables et valeurs à affecter prioritairement. Cette approche, bien que très souvent efficace en pratique, s'avère de complexité en théorie en $O(m.d^n)$ où n et m désignent respectivement le nombre de variables et le nombre de contraintes de l'instance considérée alors que d est la taille maximum des domaines.

De nombreux travaux ont été développés avec l'objectif de fournir des bornes de complexité meilleures que les précédentes en exploitant les caractéristiques particulières des instances comme par exemple l'acyclicité du réseau de contraintes [Fre82, DP87]. Les meilleures bornes connues à ce jour sont données par la "tree-width" d'un CSP, i.e. un paramètre associé au graphe de contraintes. Différentes méthodes ont été proposées comme le *Tree-Clustering* [DP89] (voir [GLS00] pour un état de l'art et une étude comparative concernant ces méthodes). Le Tree-Clustering est basé sur la notion de décomposition arborescente (tree-decomposition) du graphe de contraintes. Cette méthode consiste à fournir une représentation fondée sur un recouvrement des contraintes par des cliques dont l'agencement est acyclique. Cette nouvelle structure doit être équivalente en termes d'ensemble de solutions. La meilleure décomposition conduit à une complexité en $O(n.d^{w+1})$, où w est la tree-width du réseau [RS86]. Selon les instances, le gain effectif en termes de temps de résolution peut être considérable par rapport aux approches fondées sur l'énumération. Toutefois, la complexité en espace, insignifiante dans le cas du backtracking car généralement linéaire, peut rendre ce type d'approche complètement inutile en pratique. Cette complexité peut cependant être réduite à $O(n.s.d^s)$ où s est la taille maximum des séparateurs minimaux du graphe de contraintes [DF01].

L'objet de cette contribution est de proposer une voie alternative avec pour ambition de tirer profit de l'efficacité pratique des techniques de backtracking, tout en garantissant des bornes de complexité similaires à celles offertes par les approches structurelles. L'idée centrale est de circonscrire le backtracking en guidant le choix des variables grâce aux caractéristiques de la structure d'une décomposition arborescente du réseau. Cette approche exploite deux notions. La première est celle de "nogood structurel". Il s'agit d'un nogood au sens classique i.e. une affectation partielle de l'ensemble des variables qui ne peut pas être étendue à une solution [SV94], mais dont la découverte sera fondée par l'exploitation de propriétés structurelles. Un tel nogood servira à élaguer l'arbre de recherche en évitant l'exploration de sous-arborescences inconsistantes. La seconde notion est celle de "good structurel". Un good est une affectation partielle qui possède au moins une extension consistante sur une partie bien identifiée du problème. Un good sera détecté sur la base de critères structurels. L'élagage induit par les goods sera utilisé pour couper des branches de l'arbre de recherche en évitant d'explorer des sous-arbres consistants. D'un certain point de vue, on peut estimer que l'exploitation des goods conduit à produire des "forward-jump" dans l'arbre de recherche, par analogie avec la notion classique et opposée de backjumping. Notons dès à présent que ces notions de goods et de nogoods basées sur des propriétés structurelles ont déjà été introduites dans [BM96] mais celles présentées ici sont cependant formellement différentes.

L'exploitation de la structure à travers les notions de goods et de nogoods structurels est à la base de notre schéma de résolution énumérative. Nous montrerons comment cette approche peut garantir une borne de complexité en temps qui sera $O(n.d^{w+1})$ sous l'hypothèse de disposer d'une décomposition arborescente, tout en limitant la complexité en espace à $O(n.s.d^s)$. Ces bornes sont évaluées dans le pire des cas. Aussi, nous montrerons que notre approche est toujours plus efficace que le Tree-Clustering car elle requiert toujours moins de temps et moins d'espace. Les résultats expérimentaux que nous avons obtenus confirment d'ailleurs ces caractéristiques.

Dans la partie 2, nous introduisons des notations et des résultats sur les CSPs ainsi que des notions de théorie des graphes exploitées dans les méthodes fondées sur les décompositions arborescentes. La partie 3 présente la méthode, fournit ses fondements théoriques et donne en outre une évaluation de la complexité théorique et quelques comparaisons sur ce plan. Pour des raisons de place, cette contribution ne présente pas de résultats expérimentaux. Ceux-ci sont résumés dans la dernière partie (paragraphe 4) qui expose également les liens pouvant exister avec des travaux proches ainsi que les perspectives offertes par cette approche. Enfin, et toujours pour des raisons de place, aucune preuve n'est fournie dans cet article.

2 Préliminaires

2.1 Notations

Formellement, un *Problème de Satisfaction de Contraintes* (CSP) est défini par un quadruplet $\mathcal{P} = (X,D,C,R)$ où $X = \{x_1,x_2,\ldots,x_n\}$ est un ensemble fini de variables et $D = \{D_1,D_2,\ldots,D_n\}$ un ensemble fini de domaines tels que D_i est l'ensemble fini des valeurs que la variable x_i peut prendre. $C = \{C_1,C_2,\ldots,C_m\}$ est un ensemble fini de contraintes telles qu'une contrainte C_i est définie par les variables $\{x_{i_1},x_{i_2},\ldots,x_{i_{j_i}}\}$ sur lesquelles elle porte et $R = \{R_1,R_2,\ldots,R_m\}$ est un ensemble fini de relations sur les domaines des variables de chaque contrainte, i.e. une relation R_i est associée à chaque C_i avec $R_i \subseteq D_{i_1} \times \ldots \times D_{i_{j_i}}$. La relation R_i définit les affectations autorisées pour les variables, i.e. les affectations qui permettent de satisfaire la contrainte C_i.
Étant donné un quadruplet, différentes questions peuvent être formulées, comme le problème de décision associé à l'existence d'une solution, c'est-à-dire d'une affectation des variables satisfaisant toutes les contraintes. Dans ce cas, la question posée est de savoir s'il existe une fonction $f : X \to \cup_{i=1}^{n} D_i$ telle que $\forall i, 1 \leq i \leq m, (f(x_{i_1}),f(x_{i_2}),\ldots,f(x_{i_{j_i}})) \in R_i$. Si une telle fonction existe, alors f est une solution de $\mathcal{P}$. Le problème CSP est NP-complet.
Dans la suite, nous appellerons *CSP binaire* toute instance de CSP dont l'arité des contraintes est deux. Pour les CSPs binaires (toute contrainte concerne une paire de variables), l'objet mathématique correspondant au réseau de contraintes est un graphe (X,C), dont les sommets et les arêtes sont étiquetés respectivement par les domaines et les relations ; il est appelé *graphe de contraintes*. Pour les CSP généraux (il n'y a pas de restriction sur l'arité des contraintes), l'objet mathématique est l'*hypergraphe de contraintes*. Dans ce papier, l'étude est restreinte aux CSPs binaires afin de simplifier le propos. Toutefois, ce travail peut s'étendre sans difficulté aux CSPs d'arité quelconque.

2.2 Décomposition arborescente de CSPs

Les principaux travaux sur les CSPs peuvent être scindés en trois domaines principaux, qui ne sont cependant pas incompatibles. Les techniques de simplification par filtrage, l'optimisation des algorithmes de backtracking, et les méthodes de décomposition basées en général sur l'exploitation des classes polynomiales fondées sur les propriétés structurelles.

La méthode de base pour la résolution, généralement appelée *Backtracking Chronologique*, affecte à chaque variable une valeur de son domaine, en testant la consistance de l'affectation courante - compatibilité de la nouvelle affectation avec les précédentes - et en remontant aussi loin que nécessaire dans l'arbre de recherche si une inconsistance se présente, ou bien en l'étendant dans le cas contraire. Cette approche conduit à une explosion combinatoire. La complexité en temps dans le pire des cas est $O(m.d^n)$ tandis que la complexité en espace est en $O(n)$. Afin d'éviter l'inefficacité théorique mais surtout pratique d'une telle approche, de nombreuses techniques ont été développées. Par exemple, en simplifiant l'instance considérée par filtrage, avant ou pendant la résolution. Ou encore en déterminant les raisons d'un échec afin d'éviter que de tels échecs ne se reproduisent (*constraint learning* [Dec90], *nogood recording* [SV94]) ou afin de remonter aussi haut que possible dans l'arbre de recherche (*backjumping* [Gas79], *dependency directed backtracking* [SS77]). Par ailleurs, de nombreuses heuristiques ont été proposées afin de guider les algorithmes au niveau des choix des variables et des valeurs à affecter prioritairement. À ce jour, il n'y a ni algorithme, ni heuristique qui soit toujours meilleur que les autres, car selon les instances, certains algorithmes peuvent tirer parti de certaines particularités et se révéler inefficaces sur d'autres. Notons cependant que si l'on considère un ordre statique des variables (et/ou des valeurs), une comparaison formelle entre les algorithmes de backtracking peut être partiellement établie (voir [KvB97]). [CvB01] étend en partie ces résultats aux ordres dynamiques.

Les seules garanties existant en termes de complexité théorique avant la résolution d'un problème sont données par les méthodes de décomposition. Elles procèdent en isolant des parties *a priori* intraitables en temps polynomial, pour parvenir à une seconde étape qui garantira un temps de résolution polynomial. Ces méthodes exploitent en général les propriétés topologiques du graphe de contraintes et sont basées sur la notion de *décomposition arborescente* (tree-decomposition) de graphes [RS86], dont la définition est rappelée ci-dessous :

Définition 1 *Soit $G = (X,E)$ un graphe. Une décomposition arborescente de G est une paire $(\mathcal{C},\mathcal{T})$ avec $\mathcal{T} = (I,F)$ un arbre et $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque $\mathcal{C}_i$ est un nœud de $\mathcal{T}$ et vérifie :*

1. *$\cup_{i\in I}\mathcal{C}_i = X$,*
2. *pour toute arête $\{x,y\} \in E$, il existe $i \in I$ avec $\{x,y\} \subset \mathcal{C}_i$, et*
3. *pour tout $i,j,k \in I$, si k est un chemin de i à j dans $\mathcal{T}$, alors $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$.*

La largeur d'une décomposition arborescente $(\mathcal{C},\mathcal{T})$ est égale à $max_{i\in I}|\mathcal{C}_i| - 1$. La tree-width d'un graphe G est la largeur minimale sur toutes les décompositions arborescentes de G.

La complexité du problème de recherche d'une décomposition arborescente est NP-Hard [ACP87]. Toutefois, de nombreux travaux ont été développés dans cette direction [BG01]. Ceux-ci sont fréquemment basés sur l'exploitation de la notion de graphe *triangulé* (voir [Gol80] pour une introduction aux graphes triangulés). Les liens entre graphes triangulés et décompositions arborescentes sont évidents. En effet, étant donné un graphe triangulé, l'ensemble de ses cliques maximales $\mathcal{C} = \{\mathcal{C}_1,\mathcal{C}_2,\ldots\mathcal{C}_k\}$ de (X,E) correspond à la famille de sous-ensembles associée à cette décomposition. Comme un graphe quelconque $G = (X,E)$ n'est pas nécessairement triangulé, une décomposition arborescente peut être approximée en triangulant G. Nous appelons *triangulation* l'ajout à G d'un ensemble E' d'arêtes de telle sorte que $G' = (X,E \cup E')$ ne possède pas de cycle de longueur 4 ou plus sans corde (i.e. une arête joignant deux sommets non consécutifs dans

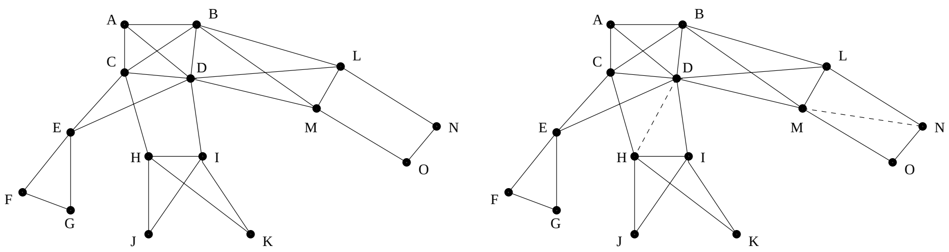

FIG. 1 – *À gauche, un graphe de contraintes sur 15 variables. À droite, ce graphe après une triangulation (lignes pointillées)*

le cycle). La largeur d'une triangulation G' du graphe G est égale à la taille maximum des cliques moins un dans le graphe résultant G'. La tree-width de G est alors égale à la largeur minimale pour toutes les triangulations.

Le graphe de la figure 1 n'est pas triangulé, mais une triangulation possible pour ce graphe est fournie ; la taille maximum des cliques est 4. C'est également une triangulation optimale, et donc la tree-width de ce graphe est 3. Dans la figure 2, l'arbre dont les nœuds correspondent aux cliques maximales du graphe triangulé constitue une décomposition arborescente possible pour le graphe de la figure 1. Nous avons ainsi $\mathcal{C}_1 = \{A,B,C,D\}$, $\mathcal{C}_2 = \{C,D,E\}$, $\mathcal{C}_3 = \{E,F,G\}$, $\mathcal{C}_4 = \{C,D,H\}$, $\mathcal{C}_5 = \{D,H,I\}$, $\mathcal{C}_6 = \{H,I,J\}$, $\mathcal{C}_7 = \{H,J,K\}$, $\mathcal{C}_8 = \{B,D,L,M\}$, $\mathcal{C}_9 = \{L,M,N\}$ et $\mathcal{C}_{10} = \{M,N,O\}$.

La méthode de décomposition de CSPs appelée *Tree-Clustering*, proposée par Dechter et Pearl [DP89], est basée sur ces notions (voir également [DF01] pour une description plus récente); elle procède en 4 étapes. La première réalise la triangulation du graphe de contraintes et la seconde calcule les cliques maximales du graphe triangulé (chaque clique correspond alors à un sous-problème). La troisième étape résout les différents sous-problèmes obtenus, et la dernière étape est constituée par la résolution du nouveau CSP général acyclique. L'idée directrice de cette méthode est de fournir un schéma systématique qui, pour tout CSP, permet de produire un CSP équivalent par un recouvrement de l'ensemble des contraintes avec pour objectif de construire un hypergraphe de contraintes acyclique. Ce dernier CSP pouvant alors être résolu en temps polynomial par rapport à sa taille.

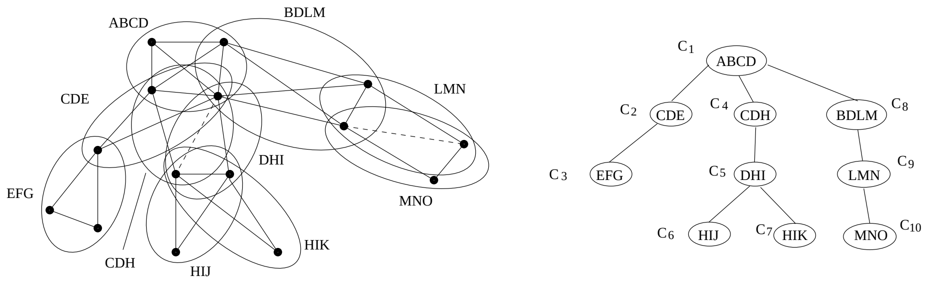

FIG. 2 – *À gauche, l'hypergraphe acyclique induit par les cliques maximales du graphe triangulé donné dans la figure 1. À droite, l'arbre associé à une décomposition arborescente de ce graphe.*

Cette méthode est généralement présentée [DP89] en utilisant l'approximation d'une triangulation optimale. Les étapes 1 et 2 sont réalisables en temps polynomial, plus précisément, en $O(n + m')$ où m' est le nombre d'arêtes du graphe après la triangulation ($m \leq m' < n^2$). Par ailleurs, l'arbre associé à l'hypergraphe acyclique peut être calculé en temps linéaire, étant données les cliques maximales. L'étape 3 est réalisable en $O(m.d^{w^+ +1})$ où w^+ est la taille moins un de la plus grande clique produite ($w^+ + 1 \leq n$). La dernière étape possède la même complexité. La complexité en espace, qui est bornée par le coût de stockage des solutions des sous-problèmes, peut être réduite à $O(n.s.d^s)$ où s est la taille maximum des séparateurs minimaux, c'est-à-dire, la taille de la plus grande intersection entre sous-problèmes ($s \leq w^+$). Finalement, notons que toute décomposition induit une valeur w^+, qui est telle que $w \leq w^+$ où w est la tree-width du graphe de contraintes initial.

Les figures 1 et 2 peuvent être considérées comme une illustration de cette méthode. Dans la figure 1, nous avons un graphe de contraintes. Lors de l'étape 1, la triangulation a ajouté deux arêtes (les lignes pointillées). Un recouvrement de ce graphe par des cliques maximales définit un hypergraphe acyclique. Chaque clique maximale définit un sous-problème.

Bien que théoriquement intéressante, cette méthode n'a pas encore démontré tout son intérêt pratique, même s'il est clair que pour certaines classes de CSPs, elle peut fournir une approche utile [DF01]. Une raison du manque d'efficacité pratique du Tree-Clustering est vraisemblablement la lourdeur de l'approche, en particulier l'espace mémoire qu'il requiert. Pour le cas où toutes les solutions sont recherchées, il peut être utile. Par contre, pour le simple test de consistance, ou bien si seulement une solution est recherchée, il sera préférable d'utiliser des algorithmes énumératifs tels que Forward-Checking (noté FC [HE80]), Real Full Look-Ahead (noté RFLA [Nad88]) ou bien Maintaining Arc-Consistency (noté MAC [SF94]).

Dans le paragraphe suivant, nous montrons comment la référence à la décomposition structurelle permet de proposer une procédure de recherche basée sur l'énumération tout en garantissant des bornes de complexité similaires à celles offertes par le Tree-Clustering.

3 La méthode BTD

3.1 Présentation

La méthode BTD (pour Backtracking sur Tree-Decomposition) procède par une recherche énumérative qui est guidée par un ordre partiel statique préétabli à partir d'une décomposition arborescente du réseau de contraintes. Aussi, la première étape de BTD consiste en un calcul de décomposition arborescente ou d'une approximation de décomposition arborescente. L'ordre partiel considéré permet d'exploiter quelques propriétés structurelles du graphe pendant la recherche afin d'élaguer certaines branches de l'arbre de recherche. En fait, ce qui distingue BTD des autres techniques de backtraking concerne les points suivants :

- l'ordre d'instanciation des variables est induit par une décomposition arborescente du graphe de contraintes,
- certaines parties de l'espace de recherche ne seront plus visitées dès que leur consistance sera connue (notion de *good structurel*),

– certaines parties de l'espace de recherche ne seront plus visitées si on sait que l'instanciation courante, bien que consistante, conduit à un échec (notion de *nogood structurel*).

Notons dès à présent que cette méthode peut être implémentée avec des variantes plus sophistiquées que le Backtracking de base, comme par exemple FC, ou MAC (ou d'autres algorithmes encore).

3.2 Justifications formelles

Soit $\mathcal{P} = (X,D,C,R)$ une instance telle que (X,C) est un graphe, avec $\mathcal{A} = (\mathcal{C},\mathcal{T})$ une décomposition arborescente (ou une approximation) où $\mathcal{T} = (I,F)$ est un arbre. Nous supposerons que les éléments de $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ sont indicés à partir de la notion de *numérotation compatible* :

Définition 2 *Une numérotation sur $\mathcal{C}$ compatible avec une numérotation préfixée de $\mathcal{T} = (I,F)$ dont $\mathcal{C}_1$ est la racine est appelée numérotation compatible $N_{\mathcal{C}}$.*

L'exemple de décomposition arborescente donné dans la figure 2 présente une numérotation compatible sur $\mathcal{C}$. Nous notons $Desc(\mathcal{C}_j)$ l'ensemble de variables appartenant à l'union des descendants $\mathcal{C}_k$ de $\mathcal{C}_j$ dans l'arbre enraciné dans $\mathcal{C}_j$, $\mathcal{C}_j$ inclus. Par exemple, $Desc(\mathcal{C}_4) = \mathcal{C}_4 \cup \mathcal{C}_5 \cup \mathcal{C}_6 \cup \mathcal{C}_7 = \{C,D,H,I,J,K\}$. La numérotation $N_{\mathcal{C}}$ définit un ordre partiel des variables qui permet d'établir un ordre d'énumération sur les variables de $\mathcal{P}$:

Définition 3 *Un ordre $\preceq_X$ sur les variables de X tel que $\forall x \in \mathcal{C}_i, \forall y \in \mathcal{C}_j$, avec $i < j$, $x \preceq_X y$ est un ordre d'énumération compatible.*

Dans l'exemple, l'ordre alphabétique $A,B,\ldots,N,O$ est un ordre d'énumération compatible. La décomposition arborescente avec la numérotation N_C permet de clarifier quelques relations dans le graphe de contraintes.

Théorème 1 *Soit $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ (avec $i < j$). Il n'existe pas d'arête $\{x,y\}$ dans le graphe (X,C) où $x \in (\cup_{k=1}^{j-1}\mathcal{C}_k) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ et $y \in Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$.*

Par exemple, soit $i = 1$, $j = 4$, et $\mathcal{C}_4$ un fils de $\mathcal{C}_1$. Il n'y a pas d'arête dans G entre $(\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3) \backslash (\mathcal{C}_1 \cap \mathcal{C}_4) = \{A,B,C,D,E,F,G\} \backslash \{C,D\} = \{A,B,E,F,G\}$ et $Desc(\mathcal{C}_4) \backslash (\mathcal{C}_1 \cap \mathcal{C}_4) = \{C,D,H,I,J,K\} \backslash \{C,D\} = \{H,I,J,K\}$.

En termes de CSP, cela signifie qu'il n'y a pas de contrainte joignant ces deux sous-ensembles de variables et par conséquent ces deux sous-problèmes. De fait, les relations de compatibilité entre les instanciations passent uniquement par le séparateur $\mathcal{C}_i \cap \mathcal{C}_j$.

La méthode BTD est basée sur un ordre d'énumération compatible et sur ce premier théorème. Considérons une instanciation consistante $\mathcal{A}$ des variables de $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_i \cup \mathcal{C}_{i+1} \cup \ldots \cup \mathcal{C}_{j-1}$, avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$. Du fait de la définition des ordres compatibles, l'énumération se poursuit sur les variables de la descendance $Desc(\mathcal{C}_j)$ à l'exception de celles qui appartiennent à $\mathcal{C}_i \cap \mathcal{C}_j$. Deux cas se présentent alors, selon qu'il existe ou non une extension consistante de l'instanciation sur $Desc(\mathcal{C}_j)$:

– **Il n'existe pas d'extension consistante**. Dans ce cas, la raison de l'inconsistance est essentiellement liée à l'insatisfaction de contraintes reliant deux variables de

$Desc(\mathcal{C}_j)$, ou (ou non exclusif) une variable de cet ensemble et une variable qui la précède dans cet ordre, c'est-à-dire qui appartient à $\mathcal{C}_i \cap \mathcal{C}_j$ (cf. théorème 1). Dans ce cas, si une nouvelle instanciation consistante $\mathcal{A}'$ telle que $\mathcal{A}'$ et $\mathcal{A}$ sont identiques sur $\mathcal{C}_i \cap \mathcal{C}_j$ est essayée, son extension sur $Desc(\mathcal{C}_j)$ conduira au même échec, indépendamment de ce qui précède. En fait, l'instanciation restreinte à $\mathcal{C}_i \cap \mathcal{C}_j$ peut être considérée comme un *nogood* au sens usuel du terme, bien qu'ici, il ait été trouvé sur la base de critères structurels. Ce nogood peut alors être exploité lors des développements ultérieurs de l'arbre de recherche.

– **Il existe une extension consistante**. Par un raisonnement similaire au précédent, il est possible de montrer que toute instanciation identique sur $\mathcal{C}_i \cap \mathcal{C}_j$ conduira à un succès sur $Desc(\mathcal{C}_j)$, parce qu'elle est indépendante de ce qui précède. Cette affectation peut être considérée comme un *good* au sens où, sur une partie du problème, $Desc(\mathcal{C}_j)$, cette instanciation possède une extension consistante. Comme les nogoods, les goods peuvent être enregistrés et utilisés lors des recherches ultérieures, autorisant des sauts dans l'arbre de recherche (*forward-jumping*), qui permettent de poursuivre l'énumération sur les variables situées après celles de $Desc(\mathcal{C}_j)$ dans l'ordre d'énumération compatible.

Les travaux les plus proches de notre approche sont ceux de Bayardo et Miranker dans [BM94] dont l'étude est limitée à la résolution de CSPs binaires arborescents. Toutefois, BTD peut être considérée comme une généralisation de leur travail puisque leur goods et leurs nogoods sont des instanciations de variables tandis que les nôtres correspondent à des affectations d'ensembles de variables (les séparateurs). Dans [BM96], Bayardo et Miranker proposent une autre généralisation des goods et des nogoods qui n'est pas basée sur les séparateurs mais sur des ensembles d'ancêtres sur la base d'un graphe de contraintes ordonné. Formellement, ce travail est différent du nôtre, bien que l'exploitation des goods et des nogoods pendant la recherche soit similaire à la nôtre (nous y revenons dans la partie 4).

Nous définissons maintenant formellement notre notion de goods et de nogoods fondée sur les séparateurs.

Définition 4 *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, un* ***good*** *(resp.* ***nogood****) de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, noté $g(\mathcal{C}_i/\mathcal{C}_j)$ (resp. $ng(\mathcal{C}_i/\mathcal{C}_j)$), est une affectation consistante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$ telle qu'il existe (resp. il n'existe pas) d'extension consistante de $\mathcal{A}$ sur $Desc(\mathcal{C}_j)$.*

Le lemme suivant et son corollaire montrent que les interactions entre un sous-problème enraciné en $\mathcal{C}_j$ et le reste du CSP transitent via l'intersection entre $\mathcal{C}_j$ et son père $\mathcal{C}_i$. Ces propriétés sont à l'origine des coupes (pour les nogoods) et des sauts (pour les goods) qui seront réalisés dans l'arbre de recherche.

Lemme 1 *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, étant donné $Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, toute instanciation consistante $\mathcal{B}$ de $Desc(\mathcal{C}_j)$ est compatible avec toute instanciation $\mathcal{A}$ de Y ssi $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$.*

Ce lemme conduit directement au corollaire suivant :

Corollaire 1 *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, toute instanciation consistante $\mathcal{B}$ de $Desc(\mathcal{C}_j)$ est compatible avec toute instanciation $\mathcal{A}$ de $(X \backslash Desc(\mathcal{C}_j)) \cup \mathcal{C}_i$ ssi $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$.*

L'exploitation des goods peut alors être formalisée comme suit :

Lemme 2 (saut par les goods (forward-jumping)) *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, étant donné $Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, alors pour tout $g(\mathcal{C}_i/\mathcal{C}_j)$, toute instanciation consistante $\mathcal{A}$ de Y telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g(\mathcal{C}_i/\mathcal{C}_j)$ possède une extension consistante sur $Desc(\mathcal{C}_j)$.*

Ainsi, si une instanciation partielle $\mathcal{A}$ est telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est un good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, alors il n'est pas nécessaire d'étendre la recherche sur $Desc(\mathcal{C}_j)$. Aussi, l'énumération se poursuit alors sur les variables du premier $\mathcal{C}_k$ localisé à l'extérieur de $Desc(\mathcal{C}_j)$, par exemple le premier frère de $\mathcal{C}_j$, s'il en existe un.

Lemme 3 (coupe par les nogoods) *Étant donnés $\mathcal{C}_i$ et $\mathcal{C}_j$ l'un de ses fils, étant donné $Y \subset X$ tel que $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, alors pour tout $ng(\mathcal{C}_i/\mathcal{C}_j)$, il n'existe pas d'affectation $\mathcal{A}$ de Y telle que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = ng(\mathcal{C}_i/\mathcal{C}_j)$ et telle que $\mathcal{A}$ possède une extension consistante sur $Desc(\mathcal{C}_j)$.*

3.3 L'algorithme de base

La méthode que nous proposons sur la base des notions précédentes peut être implémentée de plusieurs façons, selon le filtrage associé (ou non) à l'énumération. Cependant, les mécanismes seront similaires. La méthode BTD explore l'espace de recherche en utilisant un ordre compatible $\preceq_X$, qui débute sur les variables de $\mathcal{C}_1$. Dans un $\mathcal{C}_i$, l'énumération procède classiquement. Par ailleurs, quand toutes les variables sont affectées en satisfaisant toutes les contraintes concernées, nous obtenons une instanciation consistante $\mathcal{A}$ des variables de $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_i$. La recherche se poursuit alors sur les variables du premier fils $\mathcal{C}_{i+1}$ de $\mathcal{C}_i$, s'il en existe un. Plus généralement, nous considérons le cas d'un fils $\mathcal{C}_j$ de $\mathcal{C}_i$. Nous testons alors si $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est un good ou un nogood et l'action appropriée est alors exécutée :

- Dans le cas d'un nogood, nous modifions l'instanciation courante de $\mathcal{C}_i$.
- Dans le cas d'un good, un "forward-jump" est réalisé, afin de poursuivre l'énumération sur la première variable localisée après celles de $Desc(\mathcal{C}_j)$. La figure 3 illustre le cas d'un forward-jump, en supposant que $\mathcal{A}[\mathcal{C}_4 \cap \mathcal{C}_5] = \mathcal{A}[\{D,H\}]$ est un good. Nous montrons dans la partie (a) le saut réalisé dans l'ordre d'énumération compatible, et dans la partie (b), la poursuite de la recherche par rapport à la structure de l'instance.
- Dans les autres cas, i.e. $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ n'est ni un good ni un nogood, $\mathcal{A}$ doit être étendue de manière consistante sur les variables de $Desc(\mathcal{C}_j)$. Si tel est le cas, $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est enregistré en tant que good ; dans le cas contraire, si $\mathcal{A}$ ne peut être étendue de manière consistante, le nogood $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ est enregistré.

La figure 4 décrit l'algorithme BTD restreint au test de consistance de CSP : il retourne **True** si l'instanciation consistante $\mathcal{A}$ peut être étendue en une instanciation consistante sur $V_{\mathcal{C}_i}$ et sur la descendance de $\mathcal{C}_i$; **False** dans les autres cas. $V_{\mathcal{C}_i}$ représente des variables non affectées de $\mathcal{C}_i$ et G et N respectivement les ensembles de goods et de nogoods enregistrés. Cet algorithme est bien sûr exécuté après le calcul ou l'approximation d'une décomposition arborescente du graphe de contraintes.

Pour des raisons de place, nous ne donnerons pas de preuve de cet algorithme. Celle-ci, bien que fastidieuse, se fait assez facilement par induction sur le nombre de variables apparaissant dans la descendance de $\mathcal{C}_i$ exceptées celles déjà affectées et qui sont présentes

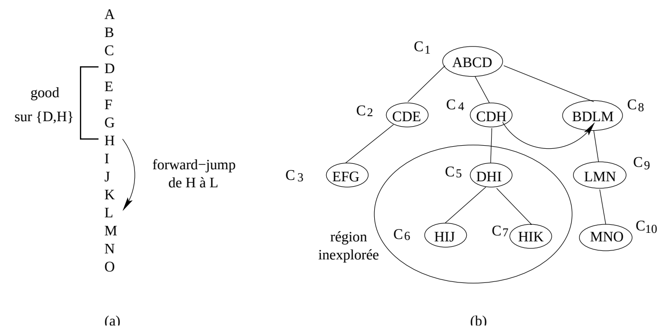

FIG. 3 – *Exemple de forward-jump avec un good $\mathcal{A}[\mathcal{C}_4 \cap \mathcal{C}_5]$ sur $\{D,H\}$. Dans (a), nous montrons le saut dans l'ordre d'énumération, tandis qu'en (b) nous voyons le saut réalisé dans la structure du problème.*

dans $\mathcal{C}_i$. La preuve est fondée sur l'exploitation des propriétés des goods et nogoods structurels.

Cette première version de BTD est basée sur le Backtracking Chronologique dont on connaît la relative inefficacité. Aussi, afin de rendre notre approche opérationnelle en pratique, est-il nécessaire de doter BTD de techniques de filtrage comme la consistance d'arc ou le forward-checking. Là encore, les propriétés de l'approche, sous réserve de se limiter à des filtrages n'altérant pas la structure du réseau de contraintes, nous garantissent la validité de ces extensions. Intuitivement, il faut considérer que les différents filtrages transiteront nécessairement par les séparateurs du graphe, et qu'en conséquence, l'exploitation des goods et des nogoods demeurera correcte. Comme extension naturelle, nous proposons **FC-BTD** (respectivement **MAC-BTD**) qui consiste en un BTD couplé avec un filtrage de type Forward-Checking (resp. de type consistance d'arc).

Bien que permettant déjà un Backtracking non chronologique, il est malgré tout possible d'étendre encore BTD avec le *Backjumping* au sens de [Gas79]. Ceci conduit alors à trois extensions directes selon qu'un filtrage est ou non utilisé et selon sa puissance (de type FC ou MAC) : **BTD-BJ**, **FC-BTD-BJ** et **MAC-BTD-BJ**.

Enfin, notons que dans la version présentée, l'algorithme BTD se limite à la construction d'une instanciation consistante partielle dont on a la garantie qu'elle pourra s'étendre à une solution du CSP traité. Pour le cas où une solution est recherchée, la modification de l'algorithme ne changera en rien les résultats de complexité que nous donnons ci-dessous, seule l'efficacité chronométrique pouvant alors s'en trouvée altérée.

3.4 Complexités en temps et en espace

Dans ce qui suit, nous supposerons qu'une décomposition arborescente du graphe de contraintes (ou bien son approximation) est disponible. Les paramètres de la complexité seront donc relatifs notamment aux caractéristiques de cette décomposition supposée connue.

Théorème 2 *Soient s la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ et $w^+ + 1$ la taille de la plus grande clique. La complexité en temps de BTD est de $O(n.s^2.m.d^{w^++1})$ et celle en espace de $O(n.s.d^s)$.*

```
BTD(A,C_i,V_{C_i})
If V_{C_i} = ∅
Then
    If Fils(C_i) = ∅ Then Return True
    Else
       Consistance ← True
       F ← Fils(C_i)
       While F ≠ ∅ and Consistance Do
            Choisir C_j dans F
            F ← F\{C_j}
            If A[C_j ∩ C_i] est un good de C_i/C_j dans G Then Consistance ← True
            Else
               If A[C_j ∩ C_i] est un nogood de C_i/C_j dans N Then Consistance ← False
               Else
                  Consistance ← BTD(A,C_j,C_j\(C_j ∩ C_i))
                  If Consistance
                  Then Enregistrer le good A[C_j ∩ C_i] de C_i/C_j dans G
                  Else Enregistrer le nogood A[C_j ∩ C_i] de C_i/C_j dans N
               EndIf
            EndIf
       EndWhile
       Return Consistance
    EndIf
Else
   Choisir x ∈ V_{C_i}
   d_x ← D_x
   Consistance ← False
   While d_x ≠ ∅ and ¬Consistance Do
        Choisir v dans d_x
        d_x ← d_x\{v}
        If ∄c ∈ C_i telle que c viole A ∪ {x ← v}
        Then Consistance ← BTD(A ∪ {x ← v},C_i,V_{C_i}\{x})
        EndIf
   EndWhile
   Return Consistance
EndIf
```

FIG. 4 – *L'algorithme BTD.*

On constate que les complexités en temps et en espace de BTD sont comparables à celles du Tree-Clustering. On peut aussi montrer que BTD développe moins de nœuds (ou autant dans le pire des cas) que le Backtracking Chronologique (BT) et que le Tree-Clustering (TC). Notons cependant que ces comparaisons ne sont rendues possibles que sous l'hypothèse que BT et TC utilisent un même ordre variables/valeurs que BTD et que TC exploite la même décomposition arborescente que BTD.

Théorème 3 *Étant donné un ordre compatible, BTD développe autant de nœuds que BT.*

Comme BT, BTD s'arrête dès que la consistance du problème est connue. Par ailleurs, TC construit toutes les affectations consistantes sur tous les $\mathcal{C}_i$. De plus, BTD ne développe pas nécessairement d'affectation sur la descendance d'un $\mathcal{C}_i$, pour le cas où un good (ou un nogood) aurait été exploité. Le gain de BTD en termes de nœuds est lié à ces propriétés. On obtient ainsi la propriété qui suit :

Théorème 4 *Étant donné un ordre compatible, BTD développe au plus autant de nœuds que TC, si celui-ci utilise BT pour la résolution des sous-problèmes $\mathcal{C}_i$.*

Notons enfin que si FC ou MAC étaient couplés avec BTD plutôt que BT, le théorème 3 conserverait sa validité. Par ailleurs, la complexité en temps serait identique, sous réserve d'un facteur multiplicatif correspondant au coût du filtrage considéré (dans le même esprit que l'analyse proposée dans [Lar00]).

4 Discussion et conclusion

L'ambition de BTD est de tirer profit de la relative efficacité pratique des méthodes fondées sur le backtracking, tout en garantissant des bornes de complexités similaires à celles offertes par les techniques de décomposition. Les éléments sur la complexité théorique présentés précédemment montrent que ces objectifs sont a priori atteints. Il reste cependant à nous assurer de l'efficacité pratique de l'approche. Pour des raisons d'espace, nous ne présenterons pas dans cet article les travaux que nous avons mené sur ce plan (ceux-ci sont disponibles sur simple requête auprès des auteurs), les tendances générales étant résumées ci-dessous.

Afin de nous assurer de la pertinence de BTD, les expérimentations ont été menées sur trois types de jeux de données :

- Les CSPs aléatoires classiques.
- Des CSPs aléatoires structurés. Il s'agit de CSPs aléatoires dont la génération a été guidée de sorte à ce que les instances engendrées recèlent des propriétés structurelles en rapport avec la décomposition arborescente (tree-width limitée notamment).
- Des instances du monde réel (FullRLFAP archive [1]).

Pour le cas des CSPs aléatoires classiques, BTD, en fait FC-BTD ou MAC-BTD, obtiennent des résultats identiques à ceux de FC ou de MAC. Cela nous a permis de nous assurer que l'exploitation de la structure n'engendrait pas de ralentissement significatif de l'efficacité en temps pour le cas où il n'existerait pas a priori de bonnes propriétés structurelles. Ce point doit constituer l'une des garanties des méthodes fondées sur l'hybridation. Pour le cas des CSPs aléatoires recelant de bonnes propriétés structurelles, nous avons observé une amélioration significative de l'efficacité de la résolution dans le cas d'une utilisation de FC-BTD (respectivement MAC-BTD) comparativement à FC (resp. MAC). Afin de nous assurer que ces résultats n'étaient pas le seul fait d'un parcours raisonné de l'espace lié à ses propriétés structurelles, une comparaison avec FC-CBJ a également été menée. Nous avons ainsi observé que FC-CBJ développe autant de nœuds que FC-BTD, mais qu'il le fait plus lentement que FC-BTD au niveau chronométrique. L'exploitation des goods et des nogoods joue alors un rôle capital. Enfin, sur les instances du monde réel (CELAR), BTD obtient soit de meilleurs résultats que les algorithmes classiques, soit des résultats comparables.

Par ailleurs, et de façon prévisible, pour ces différentes classes d'instances, nous avons pu observer que le Tree-Clustering est inopérant pour deux raisons. D'une part, son temps d'exécution est bien trop important, et d'autre part, l'espace requis pour le stockage des solutions des sous-problèmes est prohibitif, rendant cette méthode totalement inexploitable.

Il semblerait ainsi que BTD constitue une approche permettant d'exploiter les caractéristiques structurelles de certains CSPs, sans pour autant hériter des désagréments inhérents aux méthodes fondées sur la décomposition en termes de complexité en espace.

1. nous remercions le Centre d'Electronique de l'Armement (CELAR)

Pour nous assurer de cela, sur le plan théorique, mais aussi pour vérifier l'originalité de BTD, nous avons recherché parmi les travaux les plus proches les éléments qui permettent de les distinguer.

Ces travaux peuvent être classés selon trois orientations principales :

- Le Backtracking exploitant les goods et les nogoods au sens de Bayardo et Miranker [BM94, BM96] .
- Le Tree-Clustering [DP89] et ses améliorations théoriques [GLS00].
- Les approches hybrides recherchant un compromis entre le Tree-Clustering (ou la consistance adaptative [DP89]) et le Backtracking [DF01, Lar00].

Comme indiqué dans la présentation de BTD, les travaux les plus proches sont ceux de Bayardo et Miranker [BM94, BM96]. Notons que notre approche peut être considérée comme une généralisation naturelle de [BM94] puisque leur étude est limitée aux CSPs binaires acycliques (forêts). Par rapport à [BM96], alors que l'exploitation des goods et des nogoods est similaire à la nôtre, nos notions de goods et de nogoods sont formellement différentes. Dans [BM96], un good (ou un nogood) est défini par rapport à une variable x_i et à un ordre sur les sommets. Un good (ou un nogood) est une affectation d'un ensemble de variables qui précèdent x_i dans l'ordre et qui sont connectées à au moins une variable appartenant à l'ensemble des descendants de x_i dans la décomposition arborescente. Cette définition est alors formellement différente de la nôtre. Par exemple, si nous considérons un graphe de contraintes triangulé, et $x_i \in \mathcal{C}_j$, la dernière variable dans $\mathcal{C}_j$, alors un good (ou un nogood) sera une affectation de $\mathcal{C}_j \backslash \{x_i\}$. Ainsi, l'espace requis pour Learning-Tree-Solve (l'algorithme de [BM96]) est alors $O(n.d^{w^+ +1})$ ($w^+ + 1$ étant la taille du plus grand $\mathcal{C}_j$) alors que l'espace requis pour BTD est limité à $O(n.d^s)$ où s est la taille du plus grand séparateur. La complexité en temps de Learning-Tree-Solve est $O(exp(w^+ + 1))$ comme BTD.

Par ailleurs, l'intérêt pratique de Learning-Tree-Solve n'est pas présenté dans [BM96]. De plus, dans [BP00], Bayardo et Pehoushek rappellent les avantages supposés de l'exploitation des nogoods pour le test de consistance, tout en évoquant la difficulté pour fournir une implémentation efficace des goods au sens de [BM96], implémentation qui n'a d'ailleurs pas été présentée ni dans [BM96] ni dans [BP00].

Le travail de Baget et Tognetti [BT01] peut être considéré comme une approche similaire à la nôtre. En effet, dans leur méthode, les clusters sont définis par les composantes biconnexes, et les goods et nogoods - les auteurs n'utilisent pas ces expressions - sont limités à des affectations de variables, celles qui séparent les composantes biconnexes. La complexité en temps de leur méthode est alors $O(n.d^k)$ avec k la taille maximum des composantes biconnexes. Dans ce cas, on a $w^+ + 1 \leq k$. Si nous considérons le graphe de contraintes de la figure 1, nous avons deux composantes biconnexes, $\{E,F,G\}$ et $\{A,B,C,D,H,I,J,K,L,M,N,O\}$, et donc, $k = 12$ alors que $w^+ = 3$. Néanmoins, Baget et Tognetti indiquent différents moyens pour améliorer leur approche en exploitant une généralisation aux composantes k-connexes. Notons enfin qu'il n'y a pas de résultat expérimental dans [BT01].

BTD est principalement basé sur la décomposition arborescente. Aussi, les travaux proches du Tree-Clustering et de ses améliorations sont de fait pertinents dans cette comparaison. Notamment, dans [GLS00], une amélioration du Tree-Clustering est présentée de même qu'une comparaison théorique entre les principales méthodes de décomposition. Ces résultats peuvent indiquer des pistes pour des améliorations (théoriques) de BTD bien que leur incidence pratique demeure plus qu'hypothétique.

BTD peut être considéré comme une approche hybride réalisant un compromis entre l'efficacité pratique en temps et l'économie de ressources en espace. Dans [DF01], Dechter et El Fattah présentent un schéma réalisant un compromis temps-espace. Ce schéma permet de proposer une gamme d'algorithmes dont le tree-clustering et la méthode du coupe-cycle (linéaire en espace) seraient les deux extrêmes. Une autre idée séduisante dans ce travail est fournie par la possibilité de modifier la taille des séparateurs afin de minimiser l'espace. Dans les expérimentations que nous avons réalisées, nous avons d'ailleurs exploité cette idée afin de minimiser la taille des goods et des nogoods. Finalement, notons que leurs résultats expérimentaux sont limités à l'unique évaluation des paramètres structurels (w^+ et s) sur des instances structurées du monde réel (circuits combinatoires), et aucun résultat sur l'efficacité de la résolution effective des instances n'est présenté.

Enfin, dans [Lar00], Larrosa propose une méthode hybride basée sur la Consistance Adaptative (Adaptive Consistency [DP89]) et sur le Backtracking (ou FC ou MAC). La Consistance Adaptative (AdCons) s'appuie sur un schéma général d'élimination de variables qui opère en remplaçant des ensembles de variables par de nouvelles contraintes qui "résument" les effets des variables éliminées. AdCons possède les même bornes de complexité en temps et en espace que le Tree-Clustering. Aussi, l'aspect exponentiel de la complexité en espace constitue-t-il une limitation considérable pour l'intérêt pratique de l'algorithme. L'idée de Larrosa consiste à limiter la taille des nouvelles contraintes produites par AdCons à un paramètre k. Si des contraintes d'arité supérieure doivent être produites, alors cette production sera remplacée par une phase d'énumération (BT, FC, MAC, ...). Cette approche hybride permet de borner la taille de l'espace à $O(d^k)$ mais au détriment de la complexité en temps qui se trouve alors majorée par $O(exp(z(k)+k+1))$. Ici $z(k)$ est un paramètre structurel induit par k et par la largeur du graphe de contraintes ; il vérifie $z(k)+k<n$. Notons que dans le cas des graphes de contraintes ayant une densité de 6 pour cent, et en limitant la valeur de k à 2, l'auteur obtient des résultats intéressants sur des CSPs aléatoires classiques.

Pour conclure, nous indiquons quelques pistes pour les extensions potentielles de BTD. La première, concerne sa généralisation aux CSPs généraux. Celle-ci ne pose aucun problème, ni théorique, ni pratique car elle s'obtient immédiatement par construction. Une voie plus prometteuse est celle de la résolution de CSPs valués. Enfin, la comparaison théorique entre BTD et BT (respectivement FC-BTD vs FC et MAC-BTD vs MAC) devrait être étendue dans le futur afin de considérer différents ordres.

Références

[ACP87] S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embedding in a k-tree. *SIAM Journal of Discrete Mathematics*, 8:277–284, 1987.

[BG01] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal clique trees. *Artificial Intelligence*, 125:3–17, 2001.

[BM94] R. J. Bayardo and D. P. Miranker. An optimal backtrack algorithm for tree-structered constraint satisfaction problems. *Artificial Intelligence*, 71:159–181, 1994.

[BM96] R. J. Bayardo and D. P. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraints Satisfaction Problem. In *Proceedings of 13th National Conference on Artificial Intelligence*, pages 298–304, 1996.

[BP00] R. Bayardo and J. Pehoushek. Counting Models using Connected Components. In *Proceedings of AAAI 2000*, pages 157–162, 2000.

[BT01] J.-F. Baget and Y. Tognetti. Backtracking Throught Biconnected Components of a Constraint Graph. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 291–296, 2001.

[CvB01] X. Chen and P. van Beek. Conflict-Directed Backjumping Revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.

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

[DF01] R. Dechter and Y. El Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125:93–118, 2001.

[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.

[Fre82] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29:24–32, 1982.

[Gas79] J. Gaschnig. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report, 1979.

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

[Gol80] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[HE80] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[KvB97] G. Kondrak and P. van Beek. A Theorical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[Lar00] J. Larrosa. Boosting Search with Variable Elimination. In *CP 2000*, 2000.

[Nad88] B. Nadel. *Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms*, pages 287–342. In *Search in Artificial Intelligence*. Springer-Verlag, 1988.

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

[SF94] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of eleventh ECAI*, pages 125–129, 1994.

[SS77] R. Stallman and G. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[SV94] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.