Un schéma générique d'algorithmes énumératifs avec (no)good recording pour la résolution bornée de CSP

Samba Ndiaye, Cyril Terrioux

Un schéma générique d'algorithmes énumératifs avec (no)good recording pour la résolution bornée de CSP

Samba Ndojh Ndiaye **Cyril Terrioux**

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20
{samba-ndojh.ndiaye, cyril.terrioux}@univ-cezanne.fr

Résumé

Ce papier présente un schéma générique d'algorithmes énumératifs pour la résolution de CSP. Ce schéma exploite des propriétés sémantiques et topologiques du réseau de contraintes afin de produire des goods et des nogoods. Il repose sur un ensemble de séparateurs du graphe de contraintes et plusieurs fonctions et procédures paramétrables de sorte à exploiter des heuristiques, des méthodes de filtrage, des techniques de retour en arrière intelligent, d'enregistrement de nogoods classiques ou de (no)goods structurels et des bornes de complexité théorique héritées des méthodes basées sur les décompositions de graphes. Selon les choix effectués, nous obtenons une famille d'algorithmes dont la complexité en temps est comprise entre $O(exp(w+1))$ et $O(exp(n))$ avec w la largeur d'arbre du graphe de contraintes et n le nombre de variables.

1 Introduction

Le formalisme CSP (problème de satisfaction de contraintes) offre un cadre puissant pour la représentation et la résolution d'une multitude de problèmes. Un CSP consiste en un ensemble de variables qui doivent être instanciées avec une valeur de leur domaine fini respectif tout en satisfaisant un ensemble de contraintes. Déterminer si une solution existe constitue un problème NP-Complet.

La méthode usuelle pour résoudre des CSP est basée sur une recherche énumérative qui, pour être efficace, requiert l'utilisation simultanée d'heuristiques et de techniques de filtrage. Cette approche, souvent efficace en pratique, a une complexité théorique en temps exponentielle en $O(exp(n))$ pour une instance ayant n variables. D'un point de vue pratique, FC [14] et MAC [20] sont parmi les plus efficaces. D'autre part, les méthodes structurelles (par exemple [1, 7, 12, 16]) exploitent des propriétés topologiques du graphe de contraintes et peuvent ainsi proposer les meilleures bornes théoriques de complexité en temps. Les meilleures bornes connues sont fournies par la "tree-width" (souvent notée w) et conduisent à une complexité en $O(exp(w+1))$ ($w < n$). Malheureusement, la complexité en espace, généralement linéaire pour les méthodes énumératives, peut rendre ces méthodes inutilisables en pratique.

Ce papier propose un schéma générique d'algorithmes énumératifs pour la résolution de CSP. Ce schéma, basé sur un ensemble de séparateurs du graphe de contraintes, exploite des propriétés sémantiques et topologiques du graphe de contraintes. Il utilise plusieurs procédures ou fonctions paramétrables de sorte à exploiter des heuristiques, des méthodes de filtrage, des techniques de retour en arrière intelligent, d'enregistrement de nogoods classiques ou de (no)goods structurels et des bornes de complexité théorique héritées des méthodes basées sur les décompositions de graphes.

Ce papier est organisé ainsi. La section 2 rappelle les notions de base sur les CSP et les graphes. Ensuite, dans la section 3, nous décrivons notre schéma générique d'algorithmes énumératifs. La section 4 présente une analyse de la complexité. Puis, la section 5 est consacrée à une discussion. Enfin, nous concluons et donnons quelques perspectives dans la section 6.

2 Préliminaires

Un *problème de satisfaction de contraintes* (CSP) est défini par un quadruplet (X, D, C, R). X est un ensemble $\{x_1, \ldots, x_n\}$ de n variables. Chaque variable x_i prend ses valeurs dans un domaine fini d_{x_i} issu de D. Les variables sont soumises aux contraintes de C. Chaque contrainte c est définie comme un ensemble $\{x_{c_1}, \ldots, x_{c_k}\}$ de variables. Une relation r_c de R est associée à chaque contrainte c de sorte que r_c représente l'ensemble des tuples autorisés sur $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Etant donné $Y \subseteq X$ tel que $Y = \{x_{i_1}, \ldots, x_{i_k}\}$, une *affectation* des variables de Y est un tuple $\mathcal{A} = (v_{i_1}, \ldots, v_{i_k})$ de $d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$. Nous noterons $X_{\mathcal{A}}$ l'ensemble des variables instanciées dans $\mathcal{A}$. Une affectation $\mathcal{A}$ est dite *partielle* si $X_{\mathcal{A}}$ est un sous-ensemble de X. Etant donnés $Y \subseteq X$ et une affectation $\mathcal{A}$, $\mathcal{A}[Y]$ représente l'affectation $\mathcal{A}$ restreinte aux variables de Y. Une contrainte c est dite *satisfaite* par $\mathcal{A}$ si $c \subseteq Y, \mathcal{A}[c] \in r_c$, *violée* sinon. Une affectation est dite *consistante* si elle ne viole aucune contrainte, *inconsistante* sinon. Une *solution* est une affectation de chacune des variables avec une valeur de son domaine qui satisfait toutes les contraintes. Etant donnée une instance CSP (X, D, C, R), le problème CSP consiste à déterminer s'il existe une solution. Ce problème est NP-Complet. La méthode usuelle de résolution de CSP est la méthode Backtrack. Cette méthode peut être significativement améliorée par l'emploi de filtrage, d'heuristiques, de techniques de retour en arrière intelligent ou d'apprentissage [5]. Dans ce papier, sans perte de généralité, nous ne considérerons que des contraintes binaires (c'est-à-dire des contraintes n'impliquant que deux variables). Par conséquent, la structure du CSP peut être représentée par le graphe (X, C), appelé *graphe de contraintes*. Les sommets de ce graphe sont les variables de X et il existe une arête entre deux sommets si les variables correspondantes sont liées par une contrainte.

A présent, nous rappelons quelques notions de la théorie des graphes. Un graphe (X, C) est *connexe* s'il existe une chaîne liant chaque paire de sommets. Etant donné un sous-ensemble X' de X, le *sous-graphe induit par* X' *du* graphe (X, C) est le graphe (X', C') avec $C' = \{\{x, y\} \in C,\ x, y \in X'\}$. Une composante connexe d'un graphe (X, C) est un sous-ensemble maximal V de X tel que le sous-graphe induit par V à partir du graphe (X, C) soit connexe (c'est-à-dire qu'il n'existe pas de sous-ensemble V' de X tel que $V \subset V'$ et le graphe induit par V' à partir de (X, C) soit connexe). Bien sûr, un graphe connexe ne possède qu'une seule composante connexe. Un séparateur d'un graphe connexe (X, C) est un sous-ensemble S de X tel que le sous-graphe induit par $X \backslash S$ à partir de (X, C) ait au moins deux composantes connexes. Un séparateur S d'un graphe (X, C) est dit *minimal* s'il n'existe pas de séparateur S' de (X, C) tel que $S' \subset S$ et les composantes connexes du sous-graphe induit par $X \backslash S$ à partir de (X, C) sont incluses dans les composantes connexes du sous-graphe induit par $X \backslash S'$ à partir de (X, C). Par exemple, dans le graphe connexe de la figure 1(a), $\{x_3\}$ est un séparateur minimal qui déconnecte le graphe en deux composantes connexes $\{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ et $\{x_5, \ldots, x_9\}$.

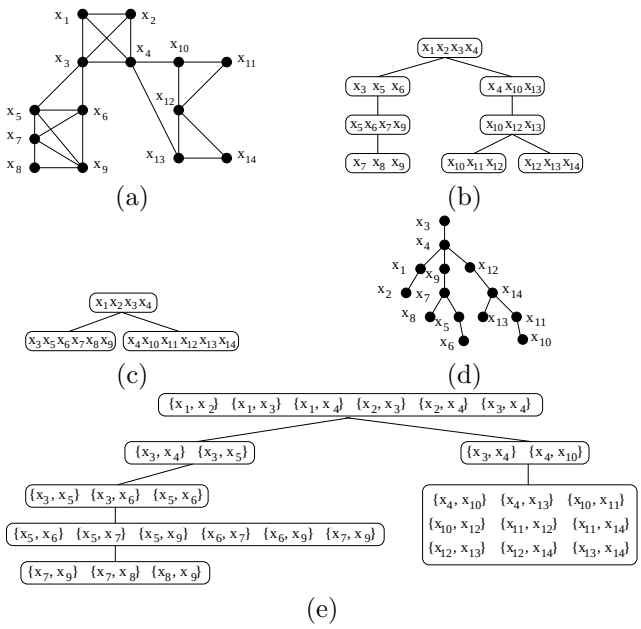

FIG. 1 – (a) Un graphe, (b) une décomposition arborescente, (c) un arbre BCC, (d) un arrangement en pseudo-arbre/arbre orienté et (e) une décomposition hinge.

3 Un schéma générique d'algorithmes énumératifs

Dans cette section, nous proposons un schéma générique appelé SBBT (pour Separator Based Back-Tracking). Ce schéma exploite les séparateurs du graphe de contraintes du CSP pour mémoriser des (no)goods structurels. Aussi, certaines parties de l'espace de recherche ne seront pas revisitées puisque leur (in)consistance sera connue. Dans la suite de l'article, nous considérerons un CSP $\mathcal{P} = (X, D, C, R)$ et son graphe de contraintes $G = (X, C)$. Etant donné un séparateur S_i de G, CC_{k,S_i} représentera une composante connexe du sous-graphe induit par $X \backslash S_i$ à partir de G. L'ensemble $SP_{k,S_i} = CC_{k,S_i} \cup S_i$ est appelé une *surcomposante connexe par rapport à* S_i. Les ensembles CC_{k,S_i} induisent des sous-

problèmes indépendants. Autrement dit, il n'existe aucune contrainte reliant deux variables de deux sous-problèmes induits distincts. Par exemple, le séparateur $S_1 = \{x_3\}$ déconnecte le graphe de la figure 1(a) en deux composantes connexes $CC_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ et $CC_{2,S_1} = \{x_5, \ldots, x_9\}$. Les surcomposantes connexes relatives à S_1 sont $SP_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}, x_3\}$ et $SP_{2,S_1} = \{x_5, \ldots, x_9, x_3\}$.

Nous pouvons définir un ensemble de séparateurs orientés en fournissant simplement la direction d'un séparateur (le séparateur racine) : étant donné un séparateur S_j orienté à partir de SP_{k,S_j}, chaque autre séparateur S_i est orienté à partir de la surcomposante connexe SP_{l,S_i} contenant S_j. Soit S_i un séparateur orienté à partir de SP_{l,S_i} dans un ensemble de séparateurs orientés. Une surcomposante connexe orientée relative à S_i est une surcomposante connexe SP_{t,S_i} relative à S_i différente de SP_{l,S_i}.

Le théorème 1 établit que les interactions entre les sous-problèmes induits par les surcomposantes connexes s'effectuent via le séparateur. Ainsi, les affectations sur ces sous-problèmes sont compatibles à partir du moment où elles sont égales au niveau du séparateur.

Théorème 1 *Soient S_i un séparateur, SP_{k_1,S_i} et SP_{k_2,S_i} deux surcomposantes connexes relatives à S_i. Une affectation $\mathcal{A}_1$ sur SP_{k_1,S_i} et une affectation $\mathcal{A}_2$ sur SP_{k_2,S_i} sont compatibles ssi $\mathcal{A}_1[S_i] = \mathcal{A}_2[S_i]$.*

Preuve : Puisque CC_{k_1,S_i} et CC_{k_2,S_i} induisent deux sous-problèmes indépendants, la compatibilité de deux affectations sur ces sous-problèmes passent forcément par les variables de S_i. Donc, ces affectations sont compatibles ssi elles sont égales sur S_i. □

Considérons une affectation consistante $\mathcal{A}$ sur un séparateur S_i et une surcomposante SP_{k,S_i}. Deux cas peuvent se produire. Si $\mathcal{A}$ ne possède aucune extension consistante sur CC_{k,S_i}, cette inconsistance provient uniquement de la violation de contraintes liant deux variables de CC_{k,S_i} ou une variable de S_i et une de CC_{k,S_i} car seul S_i connecte CC_{k,S_i} au reste du problème. Aussi, cette affectation sur S_i peut être considérée comme un nogood structurel puisque qu'aucune affectation partielle $\mathcal{B}$ avec $\mathcal{B}[S_i] = \mathcal{A}$ ne peut être étendue en une affectation consistante sur CC_{k,S_i}. De même, si $\mathcal{A}$ possède une extension consistante sur CC_{k,S_i}, cette affectation sur S_i peut être considérée comme un good structurel car n'importe quelle affectation $\mathcal{B}$ avec $\mathcal{B}[S_i] = \mathcal{A}$ peut être étendue de façon consistante sur CC_{k,S_i}. Ces notions de (no)goods structurels relatifs à une surcomposante connexe sont formellement définies ainsi :

Définition 1 *Soit S_i un séparateur. Un good (resp. nogood) structurel relatif à une surcomposante connexe SP_{k,S_i} est une affectation consistante sur S_i qui peut (resp. ne peut pas) être étendue de façon consistante sur le sous-problème induit par CC_{k,S_i}.*

Une variable x est dite *instanciable par le good* $\mathcal{A}$ relatif à SP_{k,S_i} si $x \in CC_{k,S_i}$. Le théorème 2 montre que certaines parties de l'espace de recherche peuvent être coupées grâce aux (no)goods structurels.

Théorème 2 *Soient S_i un séparateur, $\mathcal{A}$ une affectation sur S_i et $\mathcal{B}$ une affectation partielle consistante sur $X - CC_{k,S_i}$. Si $\mathcal{A}$ est un good (resp. un nogood) relatif à SP_{k,S_i} et $\mathcal{B}[S_i] = \mathcal{A}$, alors $\mathcal{B}$ peut (resp. ne peut pas) être étendue de façon consistante sur CC_{k,S_i}.*

Preuve : Si $\mathcal{A}$ est un good, alors $\mathcal{A}$ peut s'étendre de façon consistante sur CC_{k,S_i}. Soit $Sol_{\mathcal{A},SP_{k,S_i}}$ l'affectation consistante sur SP_{k,S_i} relative à ce good. Comme $\mathcal{B}[S_i] = \mathcal{A}$, $Sol_{\mathcal{A},SP_{k,S_i}}$ et $\mathcal{B}$ sont compatibles (d'après le théorème 1). Ainsi, $\mathcal{B}$ peut s'étendre de façon consistante sur CC_{k,S_i}. Si $\mathcal{A}$ est un nogood, $\mathcal{A}$ ne peut s'étendre de façon consistante sur CC_{k,S_i}. Puisque $\mathcal{B}[S_i] = \mathcal{A}$, si $\mathcal{B}$ possédait une extension consistante sur SP_{kS_i}, ce serait également une extension consistante du nogood (d'après le théorème 1), ce qui est impossible. Ainsi, il n'existe aucune extension consistante de $\mathcal{B}$ sur CC_{k,S_i}. □

Dans le schéma SBBT, $\mathcal{A}$ représente l'affectation partielle courante (qui est consistante), V l'ensemble des variables non instanciées, V_g l'ensemble des variables instanciables par des goods, x la variable courante, d_x le domaine initial de x, d son domaine courant, v la valeur courante de x, J l'ensemble des variables ayant causé les échecs précédents lors des tentatives d'extension de l'affectation courante. SBBT inclut plusieurs fonctions ou procédures. $Heuristic_{var}$ est l'heuristique d'ordonnancement des variables. Elle peut être définie de différentes manières afin d'exploiter plus ou moins la structure du problème. $Heuristic_{val}$ est l'heuristique d'ordonnancement des valeurs. $Check\_Good\_Nogood(\mathcal{A}', x, V, V'_g, J)$ vérifie pour chaque séparateur S_j devenant complètement instancié dans la nouvelle affectation courante $\mathcal{A}'$, si $\mathcal{A}'[S_j]$ est un good ou un nogood relatif au sous-problème SP_{k,S_j}. Dans le cas où $\mathcal{A}'[S_j]$ est un nogood relatif à SP_{k,S_j}, les variables de SP_{k,S_j} sont ajoutées à J car SP_{k,S_j} contient forcément les variables causant l'échec actuel. Ensuite, $false$ est retourné signifiant que, l'affectation $\mathcal{A}'$ ne peut pas conduire à une solution puisqu'elle contient un nogood. Dans le cas où $\mathcal{A}'[S_j]$ est un good relatif à SP_{k,S_j}, les variables de SP_{k,S_j} sont ajoutées à V'_g. Cet ensemble est retourné à SBBT si $\mathcal{A}'$ ne contient aucun nogood et ainsi ces variables deviennent instanciables

grâce à des goods. *Good_Recording*$(\mathcal{A}', x, V, V_g, J)$ enregistre $\mathcal{A}'[S_j]$ comme un good relatif à SP_{k,S_j} pour chaque SP_{k,S_j} devenant complètement instancié dans l'affectation courante. *Good_Cancel*(x, V, V_g) retire de V_g toutes les variables instanciables grâce à des goods contenant la variable x dont la valeur est sur le point d'être changée dans SBBT. La procédure *Failure*$(\mathcal{A}', x)$ retourne un ensemble de variables contenant celles qui sont actuellement en cause dans l'échec. *Nogood_Recording*$(\mathcal{A}, x, V, J)$ enregistre $\mathcal{A}[S_j]$ comme un nogood relatif à SP_{k,S_j} pour chaque séparateur S_j complètement instancié dans $\mathcal{A}$ tel que $x \in CC_{k,S_j}$ et CC_{k,S_j} est complètement instancié et est impliqué dans les raisons de l'échec (dans J). A titre d'exemple, les algorithmes 1 à 6 proposent une implémentation possible des différentes fonctions ou procédures respectant les spécifications énoncées ci-dessus. Elles permettent de définir une nouvelle méthode de résolution.

Le schéma générique d'algorithmes énumératifs SBBT résout récursivement le problème avec comme entrées $\mathcal{A}$, V et V_g. Il repose sur un ensemble de séparateurs et les surcomposantes connexes associées. Dans le cas où les séparateurs sont orientés, seules les surcomposantes connexes orientées sont considérées. SBBT renvoie $\emptyset$ si l'affectation $\mathcal{A}$ admet une extension consistante sur V, un ensemble J de variables causant les échecs sinon. $Heuristic_{var}$ choisit dans $V - V_g$ la prochaine variable x à instancier (ligne 3). Si le domaine courant d de x n'est pas vide, $Heuristic_{val}$ choisit une valeur v dans d. Dans le cas où l'extension $\mathcal{A}'$ de $\mathcal{A}$ n'est pas consistante, *Failure* ajoute à J l'ensemble (ou un sur-ensemble) des variables impliquées dans l'échec (ligne 16) et une nouvelle valeur (s'il en reste) est choisie grâce à $Heuristic_{val}$. Si $\mathcal{A}'$ est consistante, *Check_Good_Nogood*$(\mathcal{A}', x, V, V'_g, J)$ renvoie $false$ si $\mathcal{A}'$ contient un nogood impliquant l'affectation courante de x. $Heuristic_{val}$ choisit à nouveau une nouvelle valeur si le domaine de x n'est pas vide. Si aucun nogood n'est trouvé, *Check_Good_Nogood* retourne $true$ ainsi que l'ensemble V'_g contenant les variables instanciables grâce à des goods impliquant l'affectation courante de x. Ces variables sont alors ajoutées dans V_g. A la ligne 10, *Good_Recording* mémorise les éventuels nouveaux goods contenant x. Ensuite, SBBT est appelé récursivement par l'appel SBBT$(\mathcal{A}', V - \{x\}, V_g)$. Si $\mathcal{A}'$ ne possède aucune extension consistante, l'ensemble J' des variables impliquées dans l'échec est renvoyé et la valeur courante de x doit être changée. Puis, on retire de V_g les variables instanciables par des goods impliquant x. Si x est une des causes de l'échec, SBBT ajoute J' à J et une nouvelle valeur est choisie pour x (s'il en reste). Sinon, $J = J'$ et on effectue un saut en arrière vers une des variables causant l'échec (d'après J). Enfin, si le domaine courant

Algorithme 1 : SBBT(in : $\mathcal{A}, V$, in/out : V_g)

```
1  if V − V_g = ∅ then return ∅
2  else
3      x ← Heuristic_var(V − V_g)
4      d ← d_x ; J ← ∅ ; Backjump ← false
5      while d ≠ ∅ and Backjump = false do
6          v ← Heuristic_val(d)
7          d ← d − {v} ; A' ← A ∪ {x ← v}
8          if A' satisfies all constraints then
9              if Check_Good_Nogood (A', x, V, V'_g, J) then
10                 V_g ← V_g ∪ V'_g
11                 Good_Recording(A', x, V, V_g)
12                 J' ←SBBT(A', V − {x}, V_g)
13                 Good_Cancel(x, V, V_g)
14                 if x ∈ J' then J ← J ∪ J'
15                 else J ← J' ; Backjump ← true
16         else J ← J ∪ Failure(A', x)
17     Nogood_Recording (A, x, V)
18     return J
```

Algorithme 2 : Failure(in :$\mathcal{A}', x$)

```
1  return {x} ∪ {y ∉ V | c = {x, y} ∈ C and A' violates c}
```

Algorithme 3 : Check_Good_Nogood(in :$\mathcal{A}', x, V$,in/out :V'_g, J)

```
1  V'_g ← ∅
2  forall S_j ∈ Sep s.t. S_j ∩ V = {x} do
3      forall SP_{k,S_j} do
4          switch A'[S_j] do
5              case good related to SP_{k,S_j}
6                  V'_g ← V'_g ∪ CC_{k,S_j}
7              case nogood related to SP_{k,S_j}
8                  J ← J ∪ SP_{k,S_j} ; return false
9  return true
```

Algorithme 4 : Nogood_Recording (in :$\mathcal{A}, x, V$)

```
1  forall S_j ∈ Sep s.t. S_j ∩ V = ∅ do
2      forall CC_{k,S_j} s.t. x ∈ CC_{k,S_j} do
3          if J ∩ CC_{k,S_j} ≠ ∅ and CC_{k,S_j} ⊆ V then
4              Record A[S_j] as a nogood related to SP_{k,S_j}
```

Algorithme 5 : Good_Recording (in :$\mathcal{A}', x, V$, in/out :V_g)

```
1  forall SP_{k,S_j} s.t. SP_{k,S_j} ∩ (V − V_g) = {x} do
2      Record A'[S_j] as a good related to SP_{k,S_j}
3      V_g ← V_g ∪ CC_{k,S_j}
```

Algorithme 6 : Good_Cancel (in :x, V, in/out :V_g)

```
1  forall S_j ∈ Sep s.t. S_j ∩ V = {x} do
2      V_g ← V_g − ⋃_k CC_{k,S_j}
```

de x est vide ou qu'un saut en arrière se produit, *Nogood_Recording*$(\mathcal{A}, x, V, J)$ mémorise d'éventuels nouveaux nogoods contenant x puis SBBT renvoie J.

Théorème 3 *SBBT est correct, complet et termine.*

Preuve : SBBT repose sur BT qui est correct, complet et termine. Aussi, nous devons prouver que ces propriétés de BT ne sont pas altérées par les coupes réalisées grâce aux (no)goods et au backjumping de SBBT. Un good est mémorisé lorsqu'un sous-problème induit par un SP_{k,S_i} est complètement instancié dans l'affectation courante $\mathcal{A}$. Par conséquent, $\mathcal{A}[S_i]$ possède une extension consistante sur CC_{k,S_i}. Ainsi $\mathcal{A}[S_i]$ est un good structurel par rapport au sous-problème SP_{k,S_i}. Pour n'importe quelle affectation $\mathcal{B}$ telle que $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, nous savons que $\mathcal{B}$ peut s'étendre de façon consistante sur CC_{k,S_i} (d'après le théorème 2). Donc, continuer la recherche sur $V \backslash SP_{k,S_i}$ est correcte.

Concernant l'enregistrement de nogoods, nous savons que si certaines variables de CC_{k,S_i} sont instanciées avant toutes les variables de S_i, nous ne pouvons pas mémoriser l'affectation sur S_i comme un nogood lorsqu'elle ne peut pas s'étendre de façon consistante sur CC_{k,S_i}. En effet, SBBT n'a pas essayé toutes les affectations possibles sur CC_{k,S_i} quand il revient en arrière sur S_i. Par conséquent, un nogood n'est mémorisé que lorsque S_i est complètement instancié avant toute autre variable du sous-problème induit par un CC_{k,S_i} dans l'affectation courante $\mathcal{A}$ et que les raisons de l'échec de l'extension de $\mathcal{A}$ sur CC_{k,S_i} sont incluses dans le sous-problème induit par SP_{k,S_i}. Ainsi, puisque $\mathcal{A}[S_i]$ ne peut être étendue de façon consistante sur CC_{k,S_i}, $\mathcal{A}[S_i]$ est un nogood structurel. Pour n'importe quelle autre affectation $\mathcal{B}$ avec $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, nous savons que $\mathcal{B}$ ne peut s'étendre de façon consistante sur CC_{k,S_i} (d'après le théorème 2). Aussi, nous pouvons revenir en arrière car l'affectation courante ne peut s'étendre en une solution.

Enfin, lorsque SBBT échoue dans l'extension de l'affectation courante avec la variable x, il backjumpe sur la dernière variable instanciée dans J, l'ensemble (ou le sur-ensemble) de variables en cause dans l'échec. Les raisons de l'échec étant dans J, revenir ailleurs que sur cette variable conduira au même échec. Les propriétés de BT n'étant pas altérées par les coupes ajoutées, SBBT est correct, complet et termine. □

4 Analyse de complexité

La complexité de notre schéma générique SBBT dépend de l'ensemble de séparateurs et des fonctions et procédures employés. Par exemple, l'algorithme BT peut être obtenu dans SBBT en employant des procédures *Good_Recording* et *Nogood_Recording* vides et une fonction *Failure* naïve retournant simplement $X_{\mathcal{A}'}$. Utiliser un retour en arrière chronologique conduit généralement à rencontrer plusieurs fois les mêmes échecs. Dans SBBT, ces redondances peuvent être évitées en définissant et en revenant en arrière dans un ensemble de variables causant actuellement les échecs (structure de backjump (lignes 14-15) et fonction *Failure*). Cet ensemble peut être calculé de différentes façons (par exemple en exploitant la formule de CBJ [18] ou de GBJ [4]). De plus, l'ensemble de séparateurs et les fonctions ou procédures *Check_Good_Nogood*, *Good_Cancel*, *Good_Recording* et *Nogood_Recording* rendent également possible la réduction de l'espace de recherche à explorer en exploitant des propriétés sémantiques et topologiques du problème. Certaines parties de l'espace de recherche seront élaguées aussitôt leur (in)consistance connue. Par dessus tout, l'heuristique d'ordonnancement des variables (fonction $Heuristic_{var}$) s'avère extrêmement importante pour l'efficacité des algorithmes. Son degré de liberté peut être plus ou moins limité afin de tirer profit de l'exploitation de la structure du problème ou de l'efficacité des heuristiques dynamiques. Il est possible de combiner de multiples façons ces techniques afin de définir de nouveaux algorithmes ou de capturer très facilement des algorithmes existants comme BTD [16], BCC [1, 8], pseudo-tree search [9], Tree-solve et Learning Tree-solve [2], ou AND/OR Search Tree et AND/OR Search Graph [6].

Nous allons à présent rappeler ces différentes méthodes et montrer comment SBBT peut aisément les capturer. BTD [16] repose sur une décomposition arborescente du graphe de contraintes. Etant donné un graphe $G = (X, C)$, une *décomposition arborescente* [19] de G est un couple $(E, \mathcal{T})$ où $\mathcal{T} = (I, F)$ est un arbre avec I l'ensemble des nœuds et F celui des arêtes et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque sous-ensemble (appelé cluster) E_i est un nœud de $\mathcal{T}$ 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 une chaîne reliant i et j dans $\mathcal{T}$, alors $E_i \cap E_j \subseteq E_k$. La largeur d'une décomposition arborescente $(E, \mathcal{T})$ est égale à $max_{i \in I}|E_i| - 1$. La *largeur d'arbre* (ou *tree-width*) w de G est la largeur minimale sur toutes les décompositions arborescentes de G. La figure 1(b) propose une décomposition arborescente possible pour le graphe de la figure 1(a). BTD instancie les variables selon un ordre induit par la décomposition arborescente du graphe de contraintes utilisée. De plus, certaines parties de l'espace de recherche ne seront pas visitées à nouveau dès que leur (in)consistance est connue, grâce aux notions de *(no)good structurel*. Un good (resp. nogood) est une affectation partielle consistante sur un

ensemble de variables (un séparateur) qui peut (resp. ne peut pas) s'étendre de façon consistante sur la partie du CSP située après le séparateur. L'ordre sur les variables est calculé comme suit. Soit Y un ensemble de variables instanciées, $x_i \in E_i$, si $x_i \in Y$, alors $\forall E_j \in E$, tel que $i > j$, $\forall x_j \in E_j$, $x_j \in Y$. Aussi, BTD affecte une variable $x_i \in E_i$ ssi toutes les variables des clusters précédant E_i sont déjà instanciées. Sa complexité en temps est en $O(exp(w+1))$.

SBBT peut facilement capturer BTD, en utilisant comme ensemble de séparateurs orientés l'ensemble des intersections entre clusters dans la décomposition arborescente considérée, orientées à partir de la sur-composante connexe contenant le cluster racine et en exploitant l'heuristique $Heuristic_{var}$ pour choisir les variables dans le même ordre que BTD. Etant donnée une numérotation des clusters tel que E_1 soit le cluster racine, $Heuristic_{1,var}$ choisit comme prochaine variable à instancier la variable $x_i \in E_i$ telle que toutes les variables des clusters E_j avec $j < i$ soient déjà instanciées ou instanciables par des goods. Par conséquent, SBBT mémorise au moins les mêmes (no)goods structurels que BTD, ce qui permet de garantir la même borne de complexité en temps.

Théorème 4 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(w+1))$.*

L'heuristique $Heuristic_{2,var}$ est similaire à $Heuristic_{1,var}$, mais elle permet de choisir la prochaine variable dans toute une branche de la décomposition arborescente. Nous pouvons alors considérer qu'ainsi les clusters d'une même branche sont regroupés dans un même cluster. Cela revient donc à lancer $Heuristic_{1,var}$ sur cette nouvelle décomposition arborescente dont la largeur est $h-1$ où h est le nombre maximum de variables dans une branche de la décomposition initiale.

Théorème 5 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(h))$.*

De même l'heuristique $Heuristic_{3,var}$ est proche de $Heuristic_{1,var}$, mais nous choisissons la prochaine variable parmi les $w+k+1$ variables d'un chemin contenu dans une branche de la décomposition arborescente où k est une constante à paramétrer [15].

Théorème 6 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(2(w+k+1)-s^-))$, avec s^- la taille de la plus petite intersection de clusters.*

Concernant la méthode BCC [1, 8], elle repose sur les composantes biconnexes du graphe de contraintes. Une *composante biconnexe* (ou *bicomposante*) d'un graphe G est un sous-graphe maximum de G qui demeure connexe après la suppression de n'importe lequel de ses sommets. Le graphe des bicomposantes, obtenu en représentant chaque bicomposante par un nœud et en ajoutant une arête entre deux composantes si elles partagent un sommet, est un arbre (nous supposons le graphe de contraintes connexe) appelé un arbre BCC de G. La figure 1(c) présente un arbre BCC possible pour le graphe de la figure 1(a). BCC est basé sur cet arbre dont les nœuds sont naturellement ordonnés tels que les fils soient placés après leur père. Cet ordre naturel peut être obtenu aussi bien par un parcours en largeur qu'un parcours en profondeur de l'arbre. BCC instancie les variables selon un ordre statique *BCC-compatible* (compatible avec l'ordre naturel de l'arbre BCC considéré) : les variables de V_i sont instanciées avant celles de V_j si V_i et V_j sont des bicomposantes telles que $i < j$. Etant donné un ordre BCC-compatible, l'accesseur d'une bicomposante est sa plus petite variable dans l'ordre. Cet ordre sur les variables permet d'éviter certaines redondances. En fait, certaines valeurs des accesseurs sont marquées si on a déjà établi qu'elles pouvaient s'étendre de façon consistante sur les prochaines variables dans l'ordre. Aussi, la prochaine fois que ces valeurs seront affectées à ces variables, un forward-jump sera effectué vers une partie non explorée de l'espace de recherche. Si une valeur d'un accesseur ne peut être étendue de façon consistante sur une partie des prochaines variables selon l'ordre considéré, cette valeur est supprimée. De plus, lorsqu'un échec se produit, BCC backjumpe sur la dernière variable instanciée pouvant expliquer l'échec. Sa complexité en temps est $O(exp(k))$ avec k la taille de la plus grande bicomposante.

SBBT capture la méthode BCC en utilisant comme ensemble de séparateurs orientés l'ensemble des intersections entre bicomposantes de l'arbre BCC considéré et en employant le même ensemble de variables causant les échecs dans la fonction *Failure* et un ordre de variables BCC-compatible pour $Heuristic_{var}$ ($Heuristic_{BCC,var}$). Par conséquent, SBBT mémorise au moins les valeurs marquées (resp. supprimées) par BCC comme des goods (resp. nogoods) structurels. Par ailleurs, SBBT effectue les mêmes sauts en arrière que BCC. D'où le théorème suivant :

Théorème 7 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(k))$ avec k la taille de la plus grande bicomposante.*

La notion de décomposition hinge (figure 1(e)) est basée sur la notion d'ensemble hinge [13]. Soient $G = (X, C)$ un graphe connexe, $C' \subset C$ et $C'' \subset C - C'$. C'' est a-connecté par rapport à C' si $\forall e \in C'', \forall f \in C''$, il existe une séquence $e_1, \ldots, e_n$ d'arêtes de C'' telle

que $e_1 = e, e_n = f$ et $e_i \cap e_{i+1}$ n'est pas incluse dans $var(C')$ ($var(C')$ représente l'ensemble des variables reliées par les arêtes de C'). Soient $C' \subseteq C$ contenant au moins deux arêtes, $CC_1, \ldots, CC_m$ les composantes a-connexes de $G' = (X, C - C')$ induites par C'. C' est un hinge si pour tout $i = 1, \ldots, m$, il existe une arête $c_i \in C'$ telle que $var(CC_i) \cap var(C') \subseteq c_i$. Un hinge est minimal s'il ne contient aucun autre hinge. Une décomposition hinge de G est un arbre $\mathcal{T}$ vérifiant : (i) les nœuds de $\mathcal{T}$ sont des hinges minimaux de G, (ii) chaque arête de C est contenue dans au moins un nœud de $\mathcal{T}$, (iii) deux nœuds adjacents A et B de $\mathcal{T}$ partagent exactement une arête $c_i \in C$, $c_i = var(A) \cap var(B)$, (iv) les variables dans l'intersection de deux nœuds de l'arbre $\mathcal{T}$ sont contenues dans chaque nœud sur la chaîne les reliant. On appelle la Hinge width w_H d'un graphe de contraintes G la taille maximale des nœuds d'une décomposition hinge : c'est un invariant de G appelé degré de cyclicité. Pour une décomposition hinge donnée, les nœuds de l'arbre sont des hinges minimaux. Ils définissent donc des composantes connexes CC_i séparées du reste du problème par une unique arête c_i. On peut donc considérer l'ensemble de ces c_i comme ensemble de séparateurs. Dans le cadre des CSP binaires, une décomposition hinge peut être vue comme une décomposition arborescente en remplaçant chaque nœud C' de l'arbre par $var(C')$. Ainsi les intersections entre nœuds de l'arbre forment des séparateurs du graphe de contraintes. SBBT peut ainsi utiliser la structure tirée d'une décomposition hinge du graphe de contraintes de la même manière qu'une décomposition arborescente. Les intersections entre nœuds de l'arbre forment l'ensemble de séparateurs orientés comme précédemment avec la méthode BTD. On peut de même utiliser l'heuristique $Heuristic_{var}$ définie pour BTD. La complexité de SBBT est donc donnée par le théorème suivant.

Théorème 8 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(w_H))$.*

La méthode Pseudo-Tree Search (PTS [9]) utilise la notion de pseudo-arbre (figure 1(d)) qui permet de tenir compte de la structure du problème : dès que des parties du problème deviennent indépendantes lors de sa résolution alors elles sont résolues indépendamment. Un pseudo-arbre $\mathcal{T} = (X, C')$ de $G = (X, C)$, est un arbre orienté enraciné tel que toute arête de C n'appartenant pas à C' relie un sommet de X avec un de ses ancêtres dans $\mathcal{T}$. Les variables sont affectées selon l'ordre induit par $\mathcal{T}$: la racine est le point de départ et les sous-problèmes enracinés aux fils de la variable courante sont résolus récursivement et indépendamment.

La méthode Tree-Solve [2] est très voisine de PTS et repose sur la notion d'arrangement en arbre orienté [10]. Un arrangement en arbre orienté (figure 1(d)) d'un graphe $G = (X, C)$, est un arbre orienté enraciné $\mathcal{T} = (X, C')$ tel que deux sommets adjacents de G soient dans une même branche de $\mathcal{T}$ qui est un chemin de la racine à une feuille de l'arbre. Tree-Solve procède de la même manière que PTS sur un arrangement en arbre orienté du graphe de contraintes.

La méthode AND/OR Search Tree [6] repose sur le calcul d'un espace de recherche AND/OR défini suivant un pseudo-arbre du graphe de contraintes. Les indépendances entre sous-problèmes ainsi produits permettent de réduire exponentiellement la taille de l'espace de recherche. Soit $\mathcal{T} = (X, C')$ un pseudo-arbre de $G = (X, C)$. L'arbre de recherche AND/OR associé $S_{\mathcal{T}}(\mathcal{P})$ admet des niveaux alternés de nœuds AND et OR. Les nœuds OR x_i correspondent aux variables alors que les AND $< x_i, v_i >$ (ou v_i) sont les valeurs affectées aux variables dans leur domaine respectif. La racine de l'arbre AND/OR est le nœud OR donné par la racine de $\mathcal{T}$. Un nœud OR x_i a pour fils un nœud AND $< x_i, v_i >$ ssi $< x_i, v_i >$ est consistante avec l'affectation partielle définie sur le chemin de la racine de l'arbre au nœud x_i. Un nœud AND $< x_i, v_i >$ a pour fils un nœud OR x_j ssi x_j est un fils de x_i dans le pseudo-arbre. Une solution de $\mathcal{P}$ est un sous-arbre de l'arbre de recherche AND/OR contenant la racine de ce dernier et qui vérifie : s'il contient un nœud OR alors il contient au moins un de ses fils, s'il contient un nœud AND alors il contient tous ses fils et toutes ses feuilles sont consistantes. La méthode de résolution AND/OR Search Tree consiste donc à calculer un pseudo-arbre du graphe de contraintes et construire l'arbre de recherche AND/OR associ 'e. De ce fait, une recherche en profondeur d'abord pour trouver un sous-arbre solution suffit pour résoudre le problème.

SBBT capture PTS, Tree-solve et AND/OR Search Tree en utilisant une heuristique de choix de variables induite par un pseudo-arbre (PTS et AND/OR Search Tree) ou un arrangement en arbre orienté (Tree-Solve) du graphe de contraintes du CSP. En outre, les procédures *Good_Recording* et *Nogood_Recording* sont définies vides et la fonction *Failure* retournant $X_{\mathcal{A}'}$. L'ensemble de séparateurs peut être choisi quelconque.

Théorème 9 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(h))$ avec h la hauteur du pseudo-arbre ou de l'arrangement en arbre orienté.*

Les méthodes Tree-Solve et AND/OR Search Tree peuvent être améliorées en enregistrant des informations qui permettent d'éviter un grand nombre de redondances et réduire ainsi la taille de l'espace de recherche. La notion de séparateurs-parents définie dans [6] pour un pseudo-arbre admet une définition quasi-

similaire dans [2] d'ensemble de définition d'un sous-problème pour un arrangement en arbre orienté. Ces deux notions donnent en définitive un ensemble de séparateurs du graphe de contraintes. En effet, pour un nœud x_i de $\mathcal{T}$, un pseudo-arbre ou un arrangement en arbre orienté, l'ensemble des séparateurs-parents de x_i est formé par x_i et ses ancêtres dans $\mathcal{T}$ qui sont voisins dans G de ses descendants dans $\mathcal{T}$ alors que l'ensemble de définition de x_i est uniquement constitué de ces derniers. Des affectations identiques d'un séparateur mènent à la résolution d'un même sous-problème. Pour éviter ces redondances, il est possible d'enregistrer ces affectations ((no)goods : Learning Tree-solve). Dans le cas d'un arbre de recherche AND/OR, il est montré dans [6] que deux nœuds avec le même ensemble de séparateurs-parents sont racines de deux sous-problèmes identiques si les affectations sur les variables de l'ensemble séparateurs-parents sont identiques. On peut dès lors les fusionner et cette opération mène à un point fixe appelé graphe de contexte minimal de recherche AND/OR (AND/OR Search Graph).

Là où, les méthodes Tree-Solve et AND/OR Search Tree de base avaient une complexité spatiale linéaire, ces nouvelles versions ont une complexité en espace exponentielle en la largeur induite w^* du pseudo-arbre ou de l'arrangement en arbre orienté. Soient $G = (X, C)$ un graphe et $\mathcal{T} = (X, C')$ un pseudo-arbre ou un arrangement en arbre orienté de G. La largeur induite par $\mathcal{T}$ est égale à la largeur de $G = (X, C \cup C')$. Pour un ordre donné sur les nœuds de l'arbre, la largeur d'un nœud est donnée par le nombre de ses voisins le précédant dans l'ordre. La largeur de l'ordre est la largeur maximum des nœuds. La largeur d'un graphe est la largeur minimum sur l'ensemble des ordres possibles.

SBBT capture les méthodes Learning Tree-Solve et AND/OR Search Graph en utilisant comme ensemble de séparateurs orientés l'ensemble des ensembles de définition des sous-problèmes donnés par l'arrangement en arbre orienté (Learning Tree-Solve) ou l'ensemble des séparateurs-parents donnés par le pseudo-arbre (AND/OR Search Graph) et un ordre sur les variables donné par une numérotation préfixe de l'arbre pour $Heuristic_{var}$. Mais, cette fois-ci, les procédures *Good_Recording* et *Nogood_Recording* et la fonction *Failure* seront définies de manière usuelle. Les (no)goods enregistrés au niveau des séparateurs sont identiques à ceux de la méthode Learning Tree-Solve. Ils permettent, en outre, de retrouver l'ensemble des nœuds fusionnés dans le graphe de contexte minimal de la méthode AND/OR Search Graph.

Théorème 10 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en* $O(exp(w^*))$.

Nous voyons que SBBT peut facilement capturer plusieurs méthodes existantes. De plus, il est possible de définir de nouveaux algorithmes comme celui présenté dans la précédente section. Cette nouvelle méthode permet de calculer directement un ensemble de séparateurs et ainsi de s'assurer de bonnes propriétés (par exemple la taille ou le nombre de séparateurs ou la taille des composantes connexes). Dans la mesure où un ensemble de séparateurs définit une famille de décompositions arborescentes, il s'agit d'une structure plus générale. Il est également plus aisé de calculer une telle structure avec des propriétés adéquates que de calculer des décompositions arborescentes pour lesquelles un travail supplémentaire doit être accompli pour obtenir les mêmes propriétés. Cette méthode utilise aussi des techniques de backjumping et les notions de (no)goods structurels pour réduire la taille de l'espace de recherche en évitant certaines redondances. Par ailleurs, la fonction $Heuristic_{var}$ a un impact significatif sur le nombre de (no)goods enregistrés. Contrairement à des méthodes comme BTD ou BCC qui imposent certaines contraintes sur la fonction $Heuristic_{var}$, SBBT laisse une totale liberté à l'heuristique tout en continuant à exploiter des (no)goods. Cependant, nous ne pouvons fournir aucune garantie sur le nombre de (no)goods structurels mémorisés par SBBT. Aussi, il n'est plus possible de garantir de bonnes bornes de complexité car ces bornes dépendent des redondances évitées grâce aux (no)goods. En pratique, le nombre de (no)goods mémorisés peut toujours être considérable, mais, théoriquement, nous ne pouvons garantir que la même complexité en temps que BT.

Théorème 11 *Dans le cas général, la complexité en temps de SBBT est en* $O(exp(n))$.

La complexité en espace de SBBT ne dépend que de l'ensemble de séparateurs considéré puisque toutes les informations mémorisées sont des affectations sur les séparateurs. Le nombre de (no)goods enregistrés sur un séparateur S_i est donc majoré par $d^{|S_i|}$. Aussi, l'espace-mémoire requis est borné par le nombre maximal de (no)goods qu'on peut enregistrer sur les séparateurs.

Théorème 12 *Soit s la taille du plus grand séparateur. La complexité en espace de SBBT est en* $O(n.s.exp(s))$.

Nous avons montré, dans cette section, que la complexité en temps du schéma générique SBBT dépend de l'ensemble de séparateurs, de l'heuristique d'ordonnancement des variables et des fonctions ou procédures employées. De plus, selon les choix effectués, nous avons vu que SBBT était en mesure de capturer plusieurs méthodes existantes qui exploitent la structure du problème de différentes façons.

5 Discussion

Le schéma générique que nous proposons dans ce papier nous permet de couvrir un large spectre d'algorithmes selon les choix effectués au niveau de l'ensemble de séparateurs et des procédures et fonctions. Ce spectre inclut des algorithmes allant des méthodes structurelles (par exemple BTD, BCC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree, AND/OR Search Graph) aux méthodes purement énumératives comme BT. De plus, bien que la présentation de SBBT repose sur BT, l'emploi de techniques de filtrage ne remet pas en cause la correction, la terminaison ou la complétude de SBBT à partir du moment où le filtrage ne modifie pas le graphe de contraintes. Par exemple, nous pouvons définir une version de SBBT basée sur FC ou MAC. Cependant, un filtrage comme la consistance de chemin ne peut pas être utilisé dans la mesure où il est susceptible d'ajouter des contraintes et certains séparateurs pourraient ne plus l'être dans le nouveau graphe de contraintes.

Nous pouvons également montrer que SBBT peut facilement capturer GBJ [4] et CBJ [18] en définissant la fonction *Failure* de la bonne façon. Concernant les algorithmes d'apprentissage, SBBT se révèle proche de l'algorithme Nogood Recording (NR [21]). En fait, les nogoods structurels de SBBT sont un cas particulier des nogoods classiques exploités dans NR. Ils diffèrent principalement au niveau des justifications. Pour les nogoods structurels, les justifications reposent simplement sur les séparateurs et les sous-problèmes induits (c.-à-d. sur la structure du graphe de contraintes) au lieu des conflits rencontrés pour les nogoods classiques.

Enfin, le spectre d'algorithmes couvert par SBBT comprend les méthodes structurelles. Par exemple, il capture les méthodes PTS et AND/OR Search Tree si l'ordre d'affectation des variables est induit par un pseudo-arbre du graphe de contraintes, Tree-Solve s'il est induit par un arrangement en arbre orienté. Lorsque l'ensemble de séparateurs est calculé à partir d'une décomposition arborescente du graphe de contraintes, SBBT est équivalent à BTD. Si cet ensemble est basé sur les composantes biconnexes du graphe de contraintes, il est équivalent à BCC. De même, notre schéma générique capture la méthode Learning Tree-solve si l'ensemble des séparateurs est calculé à partir d'un arrangement en arbre orienté, et la méthode AND/OR Search Graph dans le cas où les séparateurs sont calculés à partir d'un pseudo-arbre. En revanche, tandis que la plupart de ces méthodes structurelles exploite des ordres statiques sur les variables, SBBT ne souffre pas de cet inconvénient. Il s'ensuit que la complexité en temps et la capacité à mémoriser des nogoods dépendent directement du degré de liberté accordé à l'heuristique de choix de variables. En effet, les nogoods ne sont mémorisés que lorsque cet enregistrement est correct, ce qui peut conduire à une diminution du nombre de nogoods mémorisés par rapport aux méthodes structurelles évoquées précédemment. Notons, que l'enregistrement de goods, lui, est totalement indépendant de l'ordre sur les variables.

Dans [17], les auteurs proposent une unification des algorithmes de résolution de CSP. Cette unification est en fait une caractérisation des algorithmes par leurs composantes P (propagation), L (learning : apprentissage), M (move : déplacement). Elle ne donne pas de définitions précises de ces composantes par soucis de généralité alors que SBBT en intègre plusieurs qui lui permettent de capturer un certain nombre de méthodes. Par ailleurs, dans [17], l'apprentissage se limite seulement aux nogoods.

6 Conclusion et perspectives

Dans ce papier, nous avons décrit un schéma générique d'algorithmes énumératifs appelé SBBT. Ce schéma exploite des propriétés sémantiques et topologiques du graphe de contraintes pour produire des (no)goods. En particulier, SBBT utilise un ensemble de séparateurs du graphe de contraintes. SBBT peut être modulé via l'exploitation d'heuristiques, de méthodes de filtrages, de mémorisation de nogoods classiques ou de (no)goods structurels, et de bornes théoriques de complexité en temps héritées des méthodes de décomposition du graphe de contraintes comme les décompositions arborescentes. Ainsi, le spectre d'algorithmes décrit par SBBT s'étend des méthodes structurelles (comme BTD, BCC, PTS, Tree-Solve, Learning Tree-Solve, AND/OO Search Tree ou AND/OR Search Graph) aux méthodes purement énumératives comme BT, GBJ ou CBJ. Aussi, la complexité en temps varie entre $O(exp(w+1))$ et $O(exp(n))$ pour un graphe de contraintes de n variables et dont la tree-width est w. La complexité en espace est $O(n.s.exp(s))$ avec s la taille du plus grand séparateur.

Même si la complexité en temps de SBBT dépend entre autres de l'ensemble de séparateurs utilisé et de l'heuristique de choix de variable, SBBT ne requiert aucune propriété particulière au niveau des séparateurs. Autrement dit, n'importe quel ensemble de séparateurs peut être exploité dans SBBT. Cependant, si cet ensemble de séparateurs repose sur quelques propriétés topologiques du graphe de contraintes (comme une décomposition arborescente ou une décomposition en composantes biconnexes), nous pouvons obtenir un algorithme plus puissant avec une meilleure borne de complexité en temps. Comme aucune condition n'est imposée sur l'ensemble de séparateurs, nous

pouvons facilement produire des algorithmes hybrides qui exploiteraient différentes propriétés topologiques selon la partie du graphe de contraintes considérée. Par exemple,les séparateurs pourraient être calculés à partir d'une décomposition arborescente sur une partie du problème et des bicomposantes sur une autre

De plus, l'heuristique d'ordonnancement des variables possède également une grande influence sur la capacité à mémoriser des nogoods. Plus l'heuristique est libre, moins on enregistrera de nogoods structurels. Comme ces nogoods structurels permettent d'éviter certaines redondances, leur mémorisation et leur utilisation ont un impact significatif sur l'efficacité de la résolution. De même, il est bien connu que les heuristiques de choix de variables jouent un rôle central dans l'efficacité des méthodes de résolution. Aussi, en pratique, il pourrait être intéressant d'exploiter certains compromis entre la liberté donnée à l'heuristique et la capacité à mémoriser des nogoods structurels. Le schéma SBBT est suffisamment puissant pour permettre une mise en œuvre aisée de tels compromis.

Concernant la suite de ce travail, il faudra dans un premier temps essayer de réduire l'influence de l'heuristique de choix de variables sur la capacité à enregistrer des nogoods. Une solution pourrait passer par l'exploitation de techniques proches de celles de l'algorithme Dynamic Backtracking [11]. Ensuite, nous devrons comparer SBBT avec d'autres méthodes structurelles ou purement énumératives afin de déterminer plus précisément quels sont les algorithmes existants que nous pouvons capturer avec SBBT. Concernant l'ensemble de séparateurs, dans cet article, SBBT est présenté avec un ensemble de séparateurs calculé statiquement. Aussi, une extension prometteuse serait de calculer cet ensemble de façon dynamique. Il faudra également mener une étude expérimentale afin de comparer les nouvelles méthodes données par SBBT aux méthodes de résolution déjà existantes, car pour celles qui sont capturées par ce cadre, des résultats sont disponibles dans la littérature. Enfin, il pourrait être utile d'étendre ce travail aux problèmes d'optimisation sous contraintes [3].

Remerciements

Ce travail a été soutenu par un programme blanc de l'ANR (projet STAL-DEC-OPT).

Références

[1] J.-F. Baget and Y. Tognetti. Backtracking Throught Biconnected Components of a Constraint Graph. In *Proc. of IJCAI*, pages 291–296, 2001.

[2] R. J. Bayardo and D. P. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraints Satisfaction Problem. In *Proc. of AAAI*, pages 298–304, 1996.

[3] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs : Basic properties and comparison. *LNCS*, 1106, 1996.

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

[5] R. Dechter. *Constraint processing.* Morgan Kaufmann Publishers, 2003.

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

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

[8] E. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 :755–761, 1985.

[9] E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proc. of IJCAI*, pages 1076–1078, 1985.

[10] F. Gavril. Some NP-complete Problems on Graphs. In *Proc. of the Conference on Information Sciences and Systems*, pages 91–95, 1977.

[11] M. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

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

[13] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66 :57–89, 1994.

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

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

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

[17] N. Jussien and O. Lhomme. Vers une unification des algorithmes de résolution de CSP. In *Proc. of JNPC*, pages 155–168, 2002.

[18] P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9 :268–299, 1993.

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

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

[21] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *IJAIT*, 3(2) :187–207, 1994.