A generic bounded backtracking framework for solving CSPs

Samba Ndojh Ndiaye and Cyril Terrioux

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

Abstract. This paper introduces a new generic backtracking framework for solving CSPs. This scheme exploits semantic and topological properties of the constraint network to produce goods and nogoods. It is based on a set of separators of the constraint graph and several procedures adjustable to exploit heuristics, filtering, backjumping techniques, classical nogood recording, topological (no)good recording, and topological complexity bounds inherited from methods based on graph decompositions like tree-decompositions. According to these choices, we obtain a family of algorithms whose time complexity is between $O(exp(w+1))$ and $O(exp(n))$ with w the tree-width of the constraint graph and n the number of variables.

1 Introduction

The CSP formalism (Constraint Satisfaction Problem) offers a powerful framework for representing and solving efficiently many problems. A CSP consists of a set of variables, which must be assigned in their respective finite domain, by satisfying a set of constraints. Determining if a solution exists is a NP-complete problem.

The usual method for solving CSPs is based on backtracking search, which, in order to be efficient, must use both filtering and heuristic techniques. This approach, often efficient in practice, has an exponential theoretical time complexity in $O(exp(n))$ for an instance having n variables. From a practical viewpoint, FC [1] and MAC [2] are among the most efficient ones. On the other hand, structural methods (e.g. [3–6]) exploit some topological properties of the constraint graph and can thus provide better theoretical time complexity bounds. The best known complexity bounds are given by the "tree-width" of a CSP (often denoted w) and lead to a time complexity in $O(exp(w+1))$ ($w < n$). Unfortunately, the space complexity, often linear for backtracking methods, may make such an approach unusable in practice.

This paper introduces a new generic backtracking framework for solving CSPs. This scheme based on a set of separators of the constraint graph, exploits semantic and topological properties of the constraint network to produce

(no)goods. It uses several adjustable procedures to exploit heuristics, filtering, backjumping techniques and good topological complexity bounds.

This paper is organized as follows. First, we provide the basic notions about CSPs and graphs. Then, we present our generic backtracking framework. Section 4 is devoted to a complexity analysis. Finally, we discuss about related works in section 5 before concluding and outlining future works in section 6.

2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined by a tuple (X, D, C, R). X is a set $\{x_1, \ldots, x_n\}$ of n variables. Each variable x_i takes its values in the finite domain d_{x_i} from D. The variables are subject to the constraints from C. Each constraint c is defined as a set $\{x_{c_1}, \ldots, x_{c_k}\}$ of variables. A relation r_c (from R) is associated with each constraint c such that r_c represents the set of allowed tuples over $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Given $Y \subseteq X$ such that $Y = \{x_{i_1}, \ldots, x_{i_k}\}$, an *assignment* on the variables of Y is a tuple $\mathcal{A} = (v_{i_1}, \ldots, v_{i_k})$ from $d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$. We denote by $X_{\mathcal{A}}$ the set of variables assigned in $\mathcal{A}$. An assignment $\mathcal{A}$ is said *partial* if $X_{\mathcal{A}}$ is a subset of X. Given $Y \subseteq X$ and an assignment $\mathcal{A}$, $\mathcal{A}[Y]$ represents the assignment $\mathcal{A}$ restricted to the variables of Y. A constraint c is said *satisfied* by $\mathcal{A}$ if $c \subseteq Y, \mathcal{A}[c] \in r_c$, *violated* otherwise. An assignment is said *consistent* if it does not violate any constraint, *inconsistent* otherwise. Given an instance (X, D, C, R), the CSP problem consists in determining if there is an assignment of each variable which satisfies each constraint. This problem is NP-complete. In this paper, without loss of generality, we only consider binary constraints (i.e. constraints which involve two variables). So, the structure of a CSP can be represented by the graph (X, C), called the *constraint graph*. The vertices of this graph are the variables of X and an edge joins two vertices if the corresponding variables share a constraint. The usual method for solving CSPs is based on backtracking search. The basic backtracking method is chronological Backtracking (denoted BT). It can be significantly improved by using filtering, heuristics, learning or backjumping techniques [7].

Now, we provide some notions about the graph theory. A graph (X, C) is *connected* if there exists a path linking every pair of vertices. Given a subset X' of X, the *subgraph induced by* X' *from* a graph (X, C) is the graph (X', C') with $C' = \{\{x, y\} \in C,\ x, y \in X'\}$. A connected component of a graph (X, C) is a maximal subset V of X such that the graph induced by V from (X, C) is connected (i.e. there is no subset V' of X such that $V \subset V'$ and the graph induced by V' from (X, C) is connected). Of course, a connected graph has a single connected component. A separator of a connected graph (X, C) is a subset S of X such that the subgraph induced by $X \backslash S$ from (X, C) has at least two connected components. A separator S of a graph (X, C) is said *minimal* if there is no separator S' of (X, C) such that $S' \subset S$. In the connected graph of figure 1(a), the set $\{x_3\}$ is a minimal separator that disconnects the graph into two connected components $\{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ and $\{x_5, \ldots, x_9\}$.

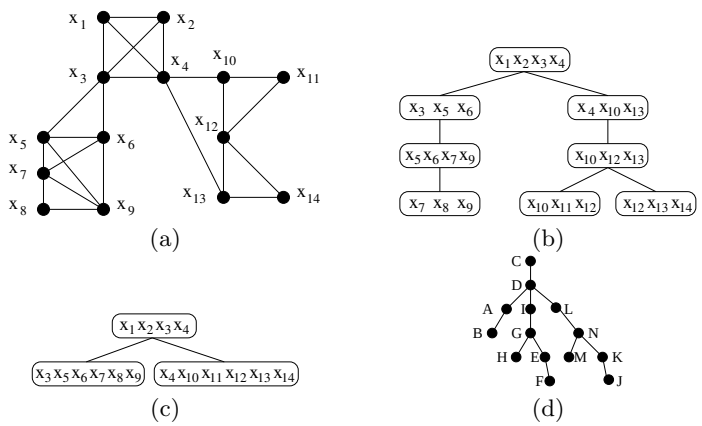

Fig. 1. (a) A graph, (b) a tree-decomposition, (c) a BCC tree, (d) a rooted-tree arrangement / pseudo-tree and (e) a hinge decomposition.

3 A generic backtracking framework

3.1 Theoretical foundations

In this section, we propose a new generic scheme of enumerative algorithms called SBBT (for Separator Based BackTracking). It exploits the separators of the constraint graph of the CSP to record structural (no)goods. Therefore, some parts of the problem will not be visited again since their (in)consistency is known. In this section, we consider a CSP $\mathcal{P} = (X, D, C, R)$ and its constraint graph $G = (X, C)$. Let S_i be a separator of G, CC_{k,S_i} denotes one of the connected components of the subgraph induced by $X \backslash S_i$ from G. A connected overcomponent related to S_i is the set $SP_{k,S_i} = CC_{k,S_i} \cup S_i$. The CC_{k,S_i} sets induce independent subproblems. There is no constraint linking two variables in two independent subproblems. For the graph of figure 1(a), $S_1 = \{x_3\}$ is a separator that disconnects G into two connected components $CC_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ and $CC_{2,S_1} = \{x_5, \ldots, x_9\}$. The connected overcomponents related to S_1 are $SP_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}, x_3\}$ and $SP_{2,S_1} = \{x_5, \ldots, x_9, x_3\}$.

We can define a directed set of separators by only providing the direction of one separator (root separator): let S_j be a separator directed from SP_{k,S_j}, each other separator S_i of the set is directed from the connected overcomponent SP_{l,S_i} containing S_j. Let S_i be a separator directed from SP_{l,S_i} in a directed set of separators, a directed connected overcomponent related to S_i is a connected overcomponent SP_{t,S_i} related to S_i different from SP_{l,S_i}.

Theorem 1 states that the interactions between the subproblems induced by the connected overcomponents pass through the separator. Thus, assignments on these subproblems are compatible if they are equal on the separator.

Theorem 1 *Let S_i be a separator, SP_{k_1,S_i} and SP_{k_2,S_i} two connected overcomponents related to S_i, $\mathcal{A}_1$ and $\mathcal{A}_2$ two assignments on SP_{k_1,S_i} and SP_{k_2,S_i}, $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible iff $\mathcal{A}_1[S_i] = \mathcal{A}_2[S_i]$.*

Proof: Since CC_{k_1,S_i} and CC_{k_2,S_i} induce independent subproblems, the compatibility of the two assignments pass through the variables of S_i. Therefore, they are compatible iff they are equal on S_i. □
Let us consider an assignment $\mathcal{A}$ on a separator S_i and a SP_{k,S_i}. Two cases can arise. If $\mathcal{A}$ has no consistent extension on CC_{k,S_i}, the reasons of this inconsistency is only due to constraints joining two variables in CC_{k,S_i} or a variable in S_i and another in CC_{k,S_i}, because CC_{k,S_i} is only connected to the rest of the problem by S_i. So, this assignment on S_i can be considered as a structural nogood since any partial assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}$ cannot be extended consistently on CC_{k,S_i}. Likewise, if $\mathcal{A}$ has a consistent extension on CC_{k,S_i}, this assignment on S_i can be considered as a structural good since any partial assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}$ can be extended consistently on CC_{k,S_i}.

We define formally below the notions of structural goods and nogoods related to connected overcomponents.

Definition 1 *Let S_i be a separator, a structural good (resp. nogood) related to a connected overcomponent SP_{k,S_i} is a consistent assignment on S_i that can (resp. cannot) be consistently extended on the subproblem induced by CC_{k,S_i}.*

A variable x is said *assignable by a good* $\mathcal{A}$ related to an overcomponent SP_{k,S_i} if $x \in CC_{k,S_i}$. Theorem 2 proves that some parts of the search space can be pruned by structural (no)goods.

Theorem 2 *Let S_i be a separator, $\mathcal{A}$ an assignment on S_i and $\mathcal{B}$ a partial consistent assignment on $X - CC_{k,S_i}$, If $\mathcal{A}$ is a good (respectively a nogood) and $\mathcal{B}[S_i] = \mathcal{A}$, then $\mathcal{B}$ can (resp. cannot) be consistently extended on CC_{k,S_i}.*

Proof: If $\mathcal{A}$ is a good, it can be extended consistently on CC_{k,S_i}. We denote by $Sol_{\mathcal{A},SP_{k,S_i}}$ the solution on SP_{k,S_i} related to the good. Since $\mathcal{B}[S_i] = \mathcal{A}$, $Sol_{\mathcal{A},SP_{k,S_i}}$ and $\mathcal{B}$ are compatible (according to theorem 1). Thus, $\mathcal{B}$ can be extended consistently on CC_{k,S_i}.

If $\mathcal{A}$ is a nogood, it cannot be extended consistently on CC_{k,S_i}. Since $\mathcal{B}[S_i] = \mathcal{A}$, if there is a consistent extension of $\mathcal{B}$ on SP_{kS_i}, it would be a consistent extension of the nogood (according to theorem 1): this is impossible. Thus, there is no consistent extension of $\mathcal{B}$ on CC_{k,S_i}. □

3.2 The generic scheme SBBT

In SBBT, $\mathcal{A}$ denotes the current partial assignment (which is consistent), V the set of unassigned variables, V_g the set of assignable variables thanks to goods, x the current variable, d_x the initial domain of x, d its current domain, v the current value of x, J the set of variables involved in the failures which have occurred during the extension of the current partial assignment. SBBT includes several functions and procedures. $Heuristic_{var}$ is the variable ordering heuristic. It can be defined in different ways to exploit more or less the problem structure. $Heuristic_{val}$ is the value ordering heuristic. $Check\_Good\_Nogood(\mathcal{A}', x, V, V'_g, J)$ checks, for each separator S_j becoming fully assigned in the new current assignment $\mathcal{A}'$, whether $\mathcal{A}'[S_j]$ is a good or a nogood related to a subproblem SP_{k,S_j}. In

Algorithm 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
```

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

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

case $\mathcal{A}'[S_j]$ is a nogood related to SP_{k,S_j}, the variables in SP_{k,S_j} are added to J because SP_{k,S_j} contains the variables causing actually this failure. Then, $false$ is returned meaning that, since $\mathcal{A}'$ contains a nogood, it cannot lead to a solution. In case $\mathcal{A}'[S_j]$ is a good related to SP_{k,S_j}, the variables in SP_{k,S_j} are added to V'_g. This set is returned to SBBT if there is no nogood in $\mathcal{A}'$ and thus they become assignable variables thanks to goods. *Good_Recording*$(\mathcal{A}', x, V, V_g, J)$ records $\mathcal{A}'[S_j]$ as a good related to SP_{k,S_j} for each SP_{k,S_j} becoming fully assigned in the current assignment. *Good_Cancel*(x, V, V_g) removes from V_g all assignable variables thanks to goods containing the variable x whose value is about to be unassigned in SBBT. The procedure *Failure*$(\mathcal{A}', x)$ returns a set of variables containing those that actually cause the failure. *Nogood_Recording*$(\mathcal{A}, x, V, J)$

Algorithm 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
```

Algorithm 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}
```

Algorithm 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}
```

records $\mathcal{A}[S_j]$ as a nogood related to SP_{k,S_j} for each separator S_j fully assigned in $\mathcal{A}$ such that $x \in CC_{k,S_j}$ and CC_{k,S_j} is fully unassigned and is involved in the reasons of the failure (in J). For example, the functions and procedures described in algorithms 1-6 propose a possible implementation of our generic scheme SBBT. Of course, they respect the specifications provided above. Note that this implementation defines a new enumerative algorithm.

SBBT solves recursively the subproblem with the inputs $\mathcal{A}$, V and V_g. It relies on a set of separators and the related connected overcomponents. In case this set is directed, only the directed connected overcomponents are considered. It returns $\emptyset$ if the assignment $\mathcal{A}$ admits a consistent extension on V, a set J of variables causing the failures otherwise. $Heuristic_{var}$ chooses the next variable x to assign in V (line 3). If the current domain d of x is not empty, $Heuristic_{val}$ chooses a value v in d. In case the extension $\mathcal{A}'$ of $\mathcal{A}$ is not consistent, *Failure* adds to J the set (or a superset) of variables involved in the failure (line 16) and $Heuristic_{val}$ chooses a new value (if any). If $\mathcal{A}'$ is consistent, *Check_Good_Nogood*$(\mathcal{A}', x, V, V'_g, J)$ returns $false$ if $\mathcal{A}'$ contains a nogood with the current value of x. $Heuristic_{val}$ chooses a new value if the domain is not empty. If no nogood is found, *Check_Good_Nogood* returns $true$ with the set V'_g containing the assignable variables thanks to goods with the current assignment of x. These variables are added in V_g. At line 10, *Good_Recording* records the possible new goods containing x. Then SBBT is recursively called on SBBT$(\mathcal{A}', V - \{x\}, V_g)$. If $\mathcal{A}'$ has no consistent extension, the set J' of variables involved in the failure is returned and the current value of x must be changed. So, first, *Good_Cancel* removes from V_g the assignable variables thanks to goods containing x. If x is involved in the failure, SBBT adds J' to J and a new value is chosen for x (if any). Otherwise, $J = J'$ and a backjump occurs to

Algorithm 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}
```

a variable involved in the failure (according to J). Finally, when the current domain of x is wiped-out or a backjump is triggered, *Nogood_Recording*$(\mathcal{A}, x, V, J)$ records new nogoods containing x (if any) and J is returned.

Theorem 3 *SBBT is sound, complete and terminates.*

Proof: The scheme SBBT is based on BT which is sound, complete and terminates. So, we are going to prove that these properties of BT are not endangered by the pruning thanks to (no)goods and the backjumping of SBBT. A good is recorded when a subproblem induced by a SP_{k,S_i} is fully assigned in the current assignment $\mathcal{A}$. So $\mathcal{A}[S_i]$ has a consistent extension on CC_{k,S_i}. Thus $\mathcal{A}[S_i]$ is a structural good related to the subproblem SP_{k,S_i}. For any assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, we know that $\mathcal{B}$ can be extended consistently on CC_{k,S_i} (theorem 2). So, we can safely continue the search on $V \backslash SP_{k,S_i}$. Regarding the nogood recording, we know that if some variables in CC_{k,S_i} are assigned before all the variables in S_i we cannot record the assignment on S_i as a nogood in case it cannot be extended consistently in CC_{k,S_i}. This is due to the fact that SBBT does not try all the possible assignments on CC_{k,S_i} when it backtracks in S_i. So a nogood is recorded when a separator S_i is fully assigned before any variable in a subproblem induced by a CC_{k,S_i} in the current assignment $\mathcal{A}$ and the reasons we fail in extending $\mathcal{A}$ on CC_{k,S_i} are in the subproblem induced by SP_{k,S_i}. So $\mathcal{A}[S_i]$ cannot be extended consistently on CC_{k,S_i}: $\mathcal{A}[S_i]$ is a structural nogood. For another assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, we know that $\mathcal{B}$ cannot be extended consistently on CC_{k,S_i} (theorem 2). So, we can backtrack because the current assignment cannot lead to a solution. Finally, when SBBT fails to extend consistently the current assignment with the variable x, it backjumps to the last assigned variable in J, the set (or superset) of variables involved in the failure. Since the reasons of this failure are in J, backtracking everywhere else will lead to the same failure. Since the additional prunings does not endanger the properties of BT, SBBT is sound, complete and terminates. □

4 Complexity analysis

The complexity of SBBT depends on the set of separators and the procedures it contains. For instance, BT can be obtained from SBBT by using empty *Good_Recording* and *Nogood_Recording* procedures and a naive *Failure* function returning $X'_{\mathcal{A}}$. A chronological backtrack can lead to encounter several times the same failures. In SBBT, these redundancies can be avoided by defining and backtracking in a set containing the variables causing actually the failures (*Backjump* structure (lines 14-15) and *Failure*). This returned set can be computed in different ways (e.g. formulae of CBJ [8] or GBJ [9]). Furthermore, the set of separators and *Check_Good_Nogood*, *Good_Cancel*, *Good_Recording* and *Nogood_Recording* also reduce the size of the search space by using some structural and semantic properties of the problem. Some parts of the search space will be pruned as soon as their (in)consistency is known. Overall, the variable ordering heuristic (function $Heuristic_{var}$) is extremely important for the efficiency of the

algorithms. Its freedom degree can be bounded more or less to derive benefit from the structure of the problem or the efficiency of dynamic heuristics. It is possible to make several combinations of these techniques in order to define new algorithms and to capture in a very easy way well known ones like BTD [6], BCC [10, 11], pseudo-tree search [12], Tree-solve and Learning Tree-solve [13], AND/OR Search Tree and AND/OR Search Graph [14]. In the following, we will present these methods and the way they can be captured by SBBT.

4.1 Separator set based on a tree-decomposition

BTD (for Backtracking with Tree-Decomposition) relies on a tree-decomposition of the constraint graph. Let $G = (X, C)$ be a graph, a *tree-decomposition* [15] of G is a pair $(E, \mathcal{T})$ where $\mathcal{T} = (I, F)$ is a tree with nodes I and edges F and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of $\mathcal{T}$ and verifies: (i) $\cup_{i \in I} E_i = X$, (ii) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$, and (iii) for all $i, j, k \in I$, if k is in a path from i to j in $\mathcal{T}$, then $E_i \cap E_j \subseteq E_k$. The width of a tree-decomposition $(E, \mathcal{T})$ is equal to $max_{i \in I}|E_i| - 1$. The *tree-width* w of G is the minimal width over all the tree-decompositions of G. In figure 1(b), we have a possible tree-decomposition of the graph in figure 1(a). BTD assigns the variables w.r.t. an order induced by the considered tree-decomposition of the constraint graph. Moreover, some parts of the search space will not be visited again as soon as their (in)consistency is known. This is possible by using the notion of *structural (no)good*. A good (resp. nogood) is a consistent partial assignment on a set of variables (a separator) that can (resp. cannot) be consistently extended on the part of the CSP located after the separator. The variable order is computed as follows. Let Y be a set of assigned variables, $x_i \in E_i$, if $x_i \in Y$, then $\forall E_j \in E$, such that $i \geq j$ $\forall x_j \in E_j$, $x_j \in Y$. So, BTD assigns a variable $x_i \in E_i$ iff all the variables in clusters preceding E_i in the cluster order are assigned first. Its time complexity is $O(exp(w + 1))$.

SBBT can capture BTD, by using as a directed separator set, the set of cluster intersections in the given tree-decomposition directed from the connected overcomponent containing the root cluster and enforcing the $Heuristic_{var}$ to choose the variable in the same order BTD does. Given a numeration on clusters s.t. E_1 is the root cluster, $Heuristic_{1,var}$ chooses as the next variable to assign $x_i \in E_i$ s.t. all the variables in clusters E_j with $j \leq i$ are already assigned or assignable by a good. So, SBBT records at least the same structural (no)goods BTD does. That allows to guarantee the same time complexity bound.

Theorem 4 *The time complexity of SBBT with the configuration described above is* $O(exp(w + 1))$.

$Heuristic_{2,var}$ is similar to $Heuristic_{1,var}$, but it is allowed to choose the next variable in a whole branch of the tree-decomposition (a branch is path from the root cluster to a leaf). We can consider that the clusters in a same branch

are grouped in a single cluster. And we run $Heuristic_{1,var}$ on this new tree-decomposition whose width is $h-1$, where h is the maximum number of variables in a branch of the basic tree-decomposition.

Theorem 5 *The time complexity of SBBT with the configuration described above is* $O(exp(h))$.

$Heuristic_{3,var}$ is similar to $Heuristic_{1,var}$, but we can choose the next variable among $w + k + 1$ variables in a path included in a branch of the tree-decomposition where k is a constant to parameterize [16].

Theorem 6 *The time complexity of SBBT with the the configuration described above is* $O(exp(2(w + k + 1) - s^{-}))$, *with* s^{-} *the minimum size of the cluster intersections.*

4.2 Separator set based on biconnected components

Regarding BCC (for Biconnected Component Backtracking), it relies on the biconnected components of the constraint graph. A *biconnected component* (or *bicomponent*) of a graph G is a maximum subgraph of G which is not disconnected by the removal of any vertex. The graph of bicomponents, obtained by representing each bicomponent as a node, then adding an edge between two components if they share a vertex, is a tree (we suppose that the constraint graph is connected) called the BCC tree of G. In figure 1(c), we have a possible BCC tree of the graph in figure 1(a). BCC is based on this tree whose nodes are *naturally* ordered s.t. the children are greater than their parent. Both DFS and BFS traversals result in a natural ordering. BCC assigns the variables of the problem w.r.t. a static *BCC-compatible* order (compatible with the natural ordering of its BCC tree): the variables in V_i are assigned before those in V_j if V_i and V_j are bicomponents s.t. $i < j$. Given a BCC-compatible ordering, the accessor of a bicomponent is its smallest variable. This variable order allows to avoid some redundancies. In fact, some values of the accessors of the bicomponents are marked if it is known that they can be extended consistently on a subset of the next variables according to the order. So the next time these same values are assigned to those variables, a forward-jump is performed to the unvisited part of the problem. If a value of an accessor cannot be consistently extended on a subset of the next variables according to the order, it is removed from the problem. Moreover, if a failure occurs, BCC backjumps to the last assigned variable whose value could explain this failure. Its time complexity is $O(exp(k))$ with k the size of the largest bicomponent.

SBBT can also capture the BCC method, by using as a directed separator set the set of bicomponent intersections of the given BCC tree, the same set of variables causing the failures in *Failure* and a BCC-compatible variable ordering for $Heuristic_{var}$ ($Heuristic_{BCC,var}$). So, SBBT records at least the values marked (resp. removed) by BCC as structural goods (resp. nogoods). Besides, SBBT performs the same backjumping when a failure occurs as BCC does. That allows to guarantee the same time complexity bound.

Theorem 7 *The time complexity of SBBT with the the configuration described above is $O(exp(k))$ with k the size of the largest bicomponent.*

4.3 Separator set based on a hinge decomposition

The hinge decomposition is based on the notion of hinge set [17]. Let $G = (X, C)$ be a connected graph, $C' \subseteq C$ containing at least two edges, $CC_1, \ldots, CC_m$ the connected components induced by C' of $G' = (X, C - C')$. C' is hinge if for all $i = 1, \ldots, m$, there is an edge $c_i \in C'$ such that $CC_i \cap var(C') \subseteq c_i$ with $var(C')$ the set of variables linked by the edges in C'. A hinge is minimal if it does not contain any other hinge. A hinge decomposition of G is a tree $\mathcal{T}$ that verifies: (i) the nodes of $\mathcal{T}$ are minimal hinges of G, (ii) each edge in C is at least in one node of $\mathcal{T}$, (iii) two neighbouring nodes A and B of $\mathcal{T}$ share exactly one edge $c_i \in C$, $c_i = A \cap B$, (iv) the variables in the intersection between two nodes of the tree $\mathcal{T}$ are in each node in the path linking these two nodes. The Hinge width (denoted w_H) of a constraint graph G is equal to the maximum size of the nodes in a hinge decomposition: it is an invariant of G named cyclicity degree. Indeed, for a given hinge decomposition, the nodes of the tree are minimal hinges. They define connected components CC_i separated from the rest of the problem by an unique edge c_i. These c_i can be considered as separators. In the framework of binary CSPs, a hinge decomposition can be seen as a tree-decomposition by replacing the set of edges in each node of the tree by the set of variables linked by these edges. Thus, the intersections between the nodes of the tree are separators of the constraint graph. So, SBBT can use the structure derived from a hinge decomposition of the constraint graph in the same way it does with a tree-decomposition. The intersections between nodes of the tree form the directed separator set like previously for the BTD method. It is also possible to use the heuristic $Heuristic_{1,var}$ defined for BTD. The complexity of SBBT is given by the following theorem.

Theorem 8 *The time complexity of SBBT with the configuration described above is $O(exp(w_H))$.*

4.4 Separator set based on a pseudo-tree or on a rooted-tree arrangement

The Pseudo-Tree Search method (PTS [12]) uses the notion of pseudo-tree (figure 1(d)) which allows to take in account the structure of the problem: as soon as some parts of the problem become independent during the solving, they are solved independently. A pseudo-tree $\mathcal{T} = (X, C')$ of $G = (X, C)$, is a directed rooted tree such that each edge in C which is not included in C' links a vertex in X with one of its ancestors in $\mathcal{T}$. The variables are assigned w.r.t. an order induced by $\mathcal{T}$: the solving begins at the root and the subproblems rooted on the sons of the current variable are solved recursively and independently.

The Tree-Solve method [13] is very close to PTS and relies on the notion of rooted-tree arrangement [18]. A rooted-tree arrangement (figure 1(d)) of a graph

$G = (X, C)$, is a directed rooted tree $\mathcal{T} = (X, C')$ such that two neighbouring vertices of G are in a same branch of $\mathcal{T}$ (which is a path from the root to a leaf of the tree). Tree-Solve proceeds in the same way PTS does on a rooted-tree arrangement of the constraint graph.

The AND/OR Search Tree method [14] relies on the computation of an AND/OR search space based on a pseudo-tree of the constraint graph. The independences between the produced subproblems allow to reduce exponentially the size of the search space. Let $\mathcal{T} = (X, C')$ be a pseudo-tree of $G = (X, C)$, the AND/OR search tree related to $\mathcal{T}$, $S_{\mathcal{T}}(\mathcal{P})$ has alternating levels of AND and OR nodes. The OR nodes x_i are variables while the AND nodes $< x_i, v_i >$ (or v_i) correspond to the values assigned to variables in their domain. The root of the AND/OR tree is the node OR given by the root of $\mathcal{T}$. An OR node x_i has a child AND node $< x_i, v_i >$ iff $< x_i, v_i >$ is consistent with the partial assignment defined on the path from the root of the tree to the node x_i. An AND node $< x_i, v_i >$ has a child OR node x_j iff x_j is a son of x_i in the pseudo-tree. A solution of $\mathcal{P}$ is a subtree of the AND/OR search tree containing its root and that verifies: if it contains an OR node then it also contains at least one of its children, if it contains an AND node then it contains all its children and all its leaves are consistent. The AND/OR Search Tree solving method is based on the computing of a pseudo-tree of the constraint graph and the construction of the related AND/OR search tree. Thus, a depth first search to find a solution subtree is sufficient to solve the problem.

SBBT captures PTS, Tree-solve and AND/OR Search Tree by using a variable ordering heuristic induced by a pseudo-tree (PTS and AND/OR Search Tree) or a rooted-tree arrangement (Tree-Solve) of the CSP constraint graph. Besides, the procedures *Good_Recording* and *Nogood_Recording* are defined empty and the function *Failure* returns $X_{\mathcal{A}'}$. The set of separators can be chosen freely.

Theorem 9 *The time complexity of SBBT with the configuration described above is $O(exp(h))$ with h the depth of the pseudo-tree or the rooted-tree arrangement.*

The Tree-Solve and AND/OR Search Tree methods can be improved by recording informations which allow to avoid many redundancies and so to reduce the size of the search space. The notion of parent-separators defined in [14] for a pseudo-tree is quasi-similar to one of definition set of a subproblem for a rooted-tree arrangement [13]. These two notions define a separator set of the constraint graph. For a node x_i in $\mathcal{T}$, a pseudo-tree or a rooted-tree arrangement, the parent-separators set of x_i contains x_i and its ancestors in $\mathcal{T}$ which are neighbours in G of its descendants in $\mathcal{T}$ while the definition set of x_i includes only these ancestors. Identical assignments on a separator lead to the solving of the same subproblem. To avoid these redundancies, it is possible to record these assignments ((no)goods: Learning Tree-solve). For an AND/OR search tree, it has been proved in [14] that two nodes with the same parent-separators set root identical subproblems if the assignments on the variables of the parent-separators set are the same. So it is possible to merge these nodes, this operation leads to a fix-point named minimal context AND/OR search graph (AND/OR Search Graph).

While the basic Tree-Solve and AND/OR Search Tree methods have a linear space complexity, these versions have an exponential space complexity in the induced width w^* of the pseudo-tree or rooted-tree arrangement. Let $G = (X, C)$ be a graph and $\mathcal{T} = (X, C')$ a pseudo-tree or a rooted-tree arrangement of G, the induced width of $\mathcal{T}$ is the width of $G = (X, C \cup C')$. For a given order on nodes of the tree, the width of a node is the number of its neighbours preceding it in the order. The width of the order is the maximum width over all nodes. The width of a graph is the minimum width over all possible orders.

SBBT captures the Learning Tree-Solve and the AND/OR Search Graph methods by using as directed separator set, the set of subproblem definition sets induced by the rooted-tree arrangement (Learning Tree-Solve) or the set of parents-separators induced by the pseudo-tree (AND/OR Search Graph) and a variable ordering induced by a prefix numeration on the tree for $Heuristic_{var}$. This time, the procedures *Good_Recording* and *Nogood_Recording* and the function *Failure* must be defined in the usual way. The (no)goods recorded on the separators are the same recorded by the Learning Tree-Solve method. They constitute as well the set of merged nodes in the minimal context graph of the AND/OR Search Graph method.

Theorem 10 *The time complexity of SBBT with the configuration described above is* $O(exp(w^*))$.

4.5 General case

We see that SBBT can easily capture several well known methods. Furthermore, it defines a family of new methods like the possible implementation proposed in section 3. This new scheme allows to compute directly a set of separators and so to ensure some suitable properties on it (e.g. the separator size or number, or the connected component size). Since a set of separators defines a family of tree-decompositions, it gives a more general structure. It is also easier to compute a structure with suitable properties than for tree-decompositions on which an additional work must often be performed to obtain these properties. The SBBT scheme also uses backjumping techniques, the notions of structural (no)goods to reduce the size of the search space by avoiding many redundancies. Besides, the $Heuristic_{var}$ has a significant impact on the number of (no)goods recorded. Unlike methods like BTD or BCC which limit the possible $Heuristic_{var}$, SBBT gives a total freedom in this choice and continues to exploit (no)goods. Yet, we have no guarantee on the number of structural (no)goods recorded by SBBT. So, it is not possible to preserve good theoretical time complexity bounds that depend on the redundancies avoided by using the (no)goods. In practice, it may be possible to record a considerable number of (no)goods, but theoretically, we have the same time complexity as BT.

Theorem 11 *In the general case, the time complexity of SBBT is* $O(exp(n))$.

The space complexity of SBBT only depends on the separator set, since all the informations recorded are assignments on the separators. The number of

(no)goods recorded on a separator S_i is bounded by $d^{|S_i|}$. Thus, the memory space required is bounded by the maximum number of (no)goods that can be recorded on the separators.

Theorem 12 *Let s be the maximum size of the separators, the space complexity of SBBT is $O(n.s.exp(s))$.*

We show that the time complexity bound of SBBT depends on the separator set, the variable ordering heuristic and on the functions and procedures used. Furthermore, according to the choices, we have seen that SBBT is able to capture in different ways several well known methods exploiting the problem structure.

5 Related works

The generic framework we propose in this paper allows us to cover a large spectrum of algorithms according to the choice made for the separator set and the included procedures and functions. This spectrum includes algorithms from structural methods (e.g. BTD, BCC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree, AND/OR Search Graph) to backtracking ones like BT. Moreover, whereas the SBBT presentation is based on BT, we can safely exploit filtering techniques which do not modify the constraint graph. For instance, SBBT can rely on FC or MAC. Yet, a filtering like path-consistency cannot be used since it may add some constraints and so some separators may not remain separators of the new constraint graph.

We show as well that SBBT can easily capture GBJ [9] and CBJ [8] by defining the function *Failure* in the right way. Regarding learning algorithms, SBBT turns to be related to the Nogood Recording algorithm (NR [19]). In fact, the structural nogoods of SBBT are a special case of classical nogoods exploited in NR. They mostly differ in their justifications. For structural nogoods, the justifications rely on the separators and the induced subproblems (i.e. on the structure of the constraint graph) instead of the encountered conflicts for classical nogoods.

Finally, the spectrum covered by SBBT includes structural methods. For instance, SBBT captures PTS and AND/OR Search Tree if the variable ordering is induced by a pseudo-tree of the constraint graph, Tree-Solve if it is induced by a rooted-tree arrangement. In case the set of separators is computed from a tree-decomposition of the constraint graph, SBBT is equivalent to BTD. If this set is based on biconnected components of the constraint graph, it is equivalent to BCC. Likewise, our generic framework captures the Learning Tree-solve method if the set of separators is computed from a rooted-tree arrangement and the AND/OR Search Graph method in case the separator set is computed from a pseudo-tree. Nevertheless, while the most of structural methods exploit static variable orders, SBBT does not suffer from this drawback. It results that the time complexity and the ability to record nogoods depend on the degree of freedom of the used variable ordering. Indeed, nogoods are only recorded when this recording is safe, what may decrease the number of recorded nogoods w.r.t.

structural methods which exploits static variable order like BTD or BCC. Note that the recording of goods is independent of the variable order.

6 Conclusion and future works

In this paper, we have described a generic framework called SBBT which exploits semantic and topological properties of the constraint network to produce (no)goods. In particular, SBBT exploits a separator set of the constraint graph. It can be modulated to exploit heuristics, filtering, classical nogood recording, topological (no)good recording, and topological complexity bounds inherited from graph decompositions like tree-decompositions. By so doing, the spectrum of algorithms described by SBBT includes algorithms from structural methods (e.g. BTD, BCC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree, AND/OR Search Graph) to backtracking ones like BT. Hence, the time complexity varies between $O(exp(w+1))$ and $O(exp(n))$ for a constraint graph having a tree-width w and n variables. The space complexity is $O(n.s.exp(s))$ with s the size of the largest separator.

Even if the time complexity of SBBT depends on the used separator set and variable ordering heuristic, SBBT does not require any particular feature for the separators. In other words, any set of separators may be exploited in SBBT. Yet, if the separator set relies on some topological properties of the constraint graph (e.g. a tree-decomposition or bicomponents), we can obtain a more powerful algorithm with a better time complexity bound. As no condition is required on the separator set, we may easily derive hybrid algorithms exploiting different topological features according to the considered part of the constraint graph. For instance, on one part of the problem, the separators can be computed from a tree-decomposition and on the other from bicomponents.

Furthermore, the exploited variable ordering heuristic has also an influence on the ability to record nogoods. The more free the heuristic is, the less structural nogoods are recorded. As the recorded nogoods allow SBBT to avoid some redundancies, their recording and use may have a significant impact on the solving efficiency. Likewise, it is well known that variable ordering heuristics play a central role in the efficiency of solving methods. So, from a practical viewpoint, it could be interesting to exploit some trade-off between the freedom of the variable ordering heuristic and the ability of recording structural nogoods. Our generic framework is powerful enough to capture such trade-offs.

Concerning the future works, the influence of the variable heuristic on the ability to record safe nogoods must be reduced. A solution could rely on the exploitation of some techniques from Dynamic Backtracking [20]. Then, we must compare SBBT to other structural or backtracking methods. Regarding the separator set, in this article, SBBT is presented by using a static separator set which is computed before the solving. So a promising extension consists in computing it dynamically. Finally, it could be useful to extend this work to optimization problems modeled as soft constraints [21].

References

1. R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
2. D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI*, pages 125–129, 1994.
3. R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
4. G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124:343–282, 2000.
5. G. Gottlob, M. Hutle, and F. Wotawa. Combining hypertree, bicomp and hinge decomposition. In *Proc. of ECAI*, pages 161–165, 2002.
6. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
7. R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.
8. P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
9. R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1990.
10. J.-F. Baget and Y. Tognetti. Backtracking Throught Biconnected Components of a Constraint Graph. In *Proc. of IJCAI*, pages 291–296, 2001.
11. E. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32:755–761, 1985.
12. E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proc. of IJCAI*, pages 1076–1078, 1985.
13. 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.
14. R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171:73–106, 2007.
15. N. Robertson and P.D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *Algorithms*, 7:309–322, 1986.
16. 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.
17. M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.
18. F. Gavril. Some NP-complete Problems on Graphs. In *Proc. of the Conference on Information Sciences and Systems*, pages 91–95, 1977.
19. T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *IJAIT*, 3(2):187–207, 1994.
20. M. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
21. 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.