Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

ELSEVIER

Artificial Intelligence 146 (2003) 43–75

Artificial Intelligence

www.elsevier.com/locate/artint

# Hybrid backtracking bounded by tree-decomposition of constraint networks

Philippe Jégou, Cyril Terrioux [*]

*Laboratoire des Sciences de l'Information et des Systèmes, LSIS-CNRS, Université d'Aix-Marseille 3,*
*Av. Escadrille Normandie-Niemen, 13397 Marseille Cedex 20, France*

Received 23 January 2002

**Abstract**

We propose a framework for solving CSPs based both on backtracking techniques and on the notion of tree-decomposition of the constraint networks. This mixed approach permits us to define a new framework for the enumeration, which we expect that it will benefit from the advantages of two approaches: a practical efficiency of enumerative algorithms and a warranty of a limited time complexity by an approximation of the tree-width of the constraint networks. Finally, experimental results allow us to show the advantages of this approach.
© 2003 Published by Elsevier Science B.V.

*Keywords:* Constraint networks; Time-space; Hybrid algorithms; Tree-decomposition; Empirical evaluation

## 1. Introduction

The CSP formalism (Constraint Satisfaction Problem) offers a powerful framework for representing and solving efficiently many problems. Formulating a problem as a CSP consists in defining a set $X$ of variables $x_1, x_2, \dots, x_n$, which must be assigned in their respective finite domain $D_i$, by satisfying a set $C$ of constraints which express restrictions between the different possible assignments. A solution is an assignment of every variable which satisfies all constraints. Many academic or real problems can be formulated in this framework. This formal framework allows the expression of NP-complete problems.

[*] Corresponding author.
*E-mail addresses:* philippe.jegou@univ.u-3mrs.fr (P. Jégou), cyril.terrioux@univ.u-3mrs.fr (C. Terrioux).

0004-3702/03/$ – see front matter © 2003 Published by Elsevier Science B.V.
doi:10.1016/S0004-3702(02)00400-9

The usual method for solving CSPs is based on backtracking search, which, in order to be efficient, must use both filtering techniques and heuristics for choosing the next variable or value. This approach, often efficient in practice, has an exponential theoretical complexity in $\mathrm{O}(m.d^n)$ where $n$ and $m$ are respectively the number of variables and the number of constraints of the treated instance, while $d$ is the maximum size of domains.

Several works have been developed, in order to provide bounds of the theoretical complexity according to particular features of the instance, like for example the acyclicity of a constraint network [14,16]. The best known bounds of complexity are given by the "tree-width" of a CSP, i.e., a parameter associated with the graph which represents the interactions between variables via the constraints. Different methods are proposed like the *Tree-Clustering* [15] (see [19] for a survey about these methods and their theoretical comparison). Tree-Clustering is based on the notion of tree-decomposition of the graph. It aims to represent any constraint network by covering the constraints by cliques, whose arrangement is a tree. The new structure must be equivalent in terms of set of solutions. The best decomposition leads to a time complexity in $\mathrm{O}(n.d^{w+1})$, where $w$ is the tree-width of the network [29]. Depending on the instances, the effective gain may be significant with respect to enumerative approaches. Yet, the space complexity, which is not considered for the backtracking because it is generally linear, may make such an approach absolutely ineffective in practice. It can be reduced to $\mathrm{O}(n.s.d^s)$ where $s$ is the maximum size of minimal separators of the network [13].

The purpose of this contribution is to propose an alternative way which aims to benefit from backtracking for its practical efficiency while giving bounds of complexity which will be ones provided by structural approaches. The main idea of our approach is that backtracking search will be guided, for the choice of variables, by the structure of the network's tree-decomposition. The order imposed to enumeration will allow to exploit two notions. The first one is the notion of "structural nogood". It is a nogood in the classical sense of the term (i.e., a partial assignment of the set of variables which cannot be extended to a solution [34]), but we only find it thanks to structural properties. It will be used for pruning the tree search by cuts which permit not to explore inconsistent subtrees. The second notion is one of "structural good". A good is a partial assignment which has at least a consistent extension on a well-identified part of the problem. A good will be detected by structural criteria. The pruning induced by goods is used to cut branches of the search tree in order to avoid exploring consistent subtrees. In some respects, exploiting goods leads to realize a "forward-jump" in the search tree, by analogy with the classical and reverse terminology of backjumping. Note that related notions of goods and nogoods based on structural properties have been introduced in [5] but these notions are formally different.

The exploitation of the structure through the notions of structural goods and nogoods is at the root of our scheme of enumerative resolution. We will explain how this approach can guarantee a theoretical time bound, which is at most $\mathrm{O}(n.d^{w+1})$ if we get an optimal tree-decomposition of the network, while limiting the space complexity to $\mathrm{O}(n.s.d^s)$. The given bounds are in the worst case; so we will show that our approach is always more efficient than Tree-Clustering because our method requires less time and less space. Experimental results will confirm these features.

In Section 2, we remember the main notations and results about CSPs as well as the notions of graph theory exploited in tree-decomposition methods. Section 3 presents the method and justifies its validity. In Section 4, we then provide comparative theoretical results and time and space complexities. Section 5 presents some experimental results, Section 6 recalls some related works, and we finally give some perspectives which are offered by our approach in Section 7.

## 2. Preliminaries

### 2.1. Notations

Formally, a *Constraint Satisfaction Problem* is defined by a quadruplet $\mathcal{P} = (X, D, C, R)$ with $X = \{x_1, x_2, \ldots, x_n\}$ a finite set of variables and $D = \{D_1, D_2, \ldots, D_n\}$ a finite set of domains such that $D_i$ is the finite set of values which the variable $x_i$ can take. $C = \{C_1, C_2, \ldots, C_m\}$ is a finite set of constraints such that a constraint $C_i$ is defined by a set of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_{j_i}}\}$ and $R = \{R_1, R_2, \ldots, R_m\}$ is a finite set of relations over the domains of variables of each constraint, i.e., a relation is associated with each $C_i$ such that $R_i \subseteq D_{i_1} \times \cdots \times D_{i_{j_i}}$. The relation $R_i$ defines the allowed assignments of variables, i.e., the assignments which satisfy the constraint $C_i$.

Given such a quadruplet, different queries can be formulated, like the decision problem which concerns the existence of an assignment of variables satisfying all the constraints, i.e., does a function $f : X \to \bigcup_{i=1}^{n} D_i$ exist such that $\forall i,\ 1 \leqslant i \leqslant m$, $(f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_{j_i}})) \in R_i$. If such a function exists, then $f$ is a solution of $\mathcal{P}$. The CSP problem is NP-complete.

Afterwards, we call *binary CSP* every instance of CSP whose arity of constraints is two. For binary CSPs (every constraint involves a pair of variables), the mathematical object corresponding to the constraint network is a graph $(X, C)$, whose vertices and edges are labeled respectively by the domains and the relations; it is called the *constraint graph*. For $n$-ary CSPs (the constraints have any arity), the mathematical object is an hypergraph, the *constraint hypergraph*. In this paper, we restrict the study to binary CSPs, without loss of generality, in order to simplify the explanations.

### 2.2. Tree-decomposition of CSPs

The significant works about CSPs can be divided in three trends, which are not incompatible: the techniques of simplification by filtering, the optimization of backtracking algorithms, and the decomposition methods based on the exploitation of polynomial classes.

The basic method of resolution, generally called *Chronological Backtracking*, assigns to each variable a value of its domain, by checking the consistency of the current instantiation—compatibility of the new assignment with the previous ones—and by going back as far as possible in the search tree if an inconsistency occurs, or by extending it otherwise. This approach leads to a combinatorial explosion. Its worst-case time complexity is $\mathrm{O}(m.d^n)$ while its space complexity can be bounded to $\mathrm{O}(n)$. In order to lessen the impact of the theoretical and practical inefficiency of such an approach, many different techniques were developed. For example, simplify the treated instance by filtering, before or during the resolution. Either, analyze the reasons of failures in order to prevent these failures reproducing (constraint learning [12], nogood recording [34]) as well as jumping back as far as possible in the search tree (backjumping [17], dependency directed backtracking [33], conflict-directed backjumping [28], Dynamic Backtracking [18]). Jointly, many heuristics were proposed with a view to guide the algorithms for the choices of variables and values to assign first. To date, there is neither algorithm, nor heuristic which are always better than other ones, because the particular features of instances can favour one method or another one. Note that if we consider static variables (and/or values) ordering, a formal comparison between backtracking algorithms can be partially established (see [23]). [11] partly extends these results to dynamic orderings.

The only guarantee which can exist in terms of theoretical complexity before solving a problem are offered by decomposition methods. They proceed by isolating the parts intrinsically exponential—that is to say untractable in polynomial theoretical time—to induce a second step which guarantees a polynomial time of resolution. These methods generally exploit topological properties of the constraint graph and are based on the notion of tree-decomposition of graphs [29], as defined below.

**Definition 1** (*tree-decomposition* [29]). Let $G = (X, E)$ be a graph.

A tree-decomposition of $G$ is a pair $(\mathcal{C}, \mathcal{T})$ with $\mathcal{T} = (I, F)$ a tree and $\mathcal{C} = \{\mathcal{C}_i\colon i \in I\}$ a family of subsets of $X$, such that each $\mathcal{C}_i$ is a node of $\mathcal{T}$ and verifies:

(1) $\bigcup_{i\in I} \mathcal{C}_i = X$,
(2) for all edge $\{x, y\} \in E$, there exists $i \in I$ with $\{x, y\} \subset \mathcal{C}_i$, and
(3) for all $i, j, k \in I$, if $k$ is in a path from $i$ to $j$ in $\mathcal{T}$, then $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$.

The width of a tree-decomposition $(\mathcal{C}, \mathcal{T})$ is equal to $\max_{i\in I} |\mathcal{C}_i| - 1$. The tree-width of the graph $G$ is the minimal width over all the tree-decompositions of $G$.

Note that for the reader who is not familiar with these notions, the definition of a tree $\mathcal{T} = (I, F)$ refers to a set of edges $F$ which is required to satisfy the part (3) of Definition 1. Even if the complexity of the problem of finding tree-decomposition is NP-Hard [1], many works have been developed in this direction [3], which often exploit equivalent definitions of this notion, including one based on an algorithmic approach related to *triangulated* graphs (see [20] for an introduction to triangulated graphs). The link between triangulated graphs and tree-decomposition is obvious. Indeed, given a triangulated graph, the set of maximal cliques $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}$ of $(X, E)$ corresponds to the family of subsets associated with a tree-decomposition. As any graph $G = (X, E)$ is not necessarily triangulated, a tree-decomposition can be approximated by triangulating $G$. We call *triangulation* the addition to $G$ of a set $E'$ of edges such that $G' = (X, E \cup E')$ has no cycle of length at least 4 without a chord (i.e., an edge joining two non-consecutive vertices in the cycle). The width of a triangulation $G'$ of graph $G$ is equal to the maximal size of cliques minus one in the resulting graph $G'$. The tree-width of $G$ is then equal to the minimal width over all triangulations.

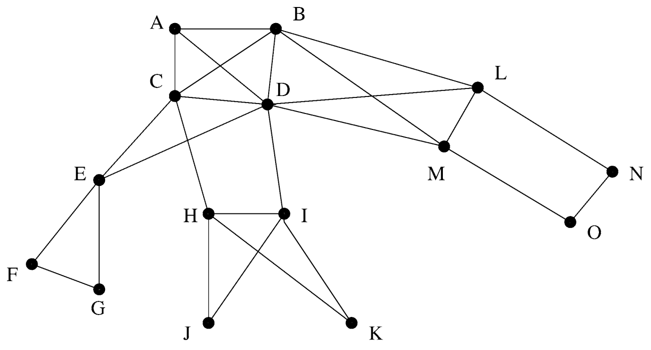

Fig. 1. A constraint graph on 15 variables.

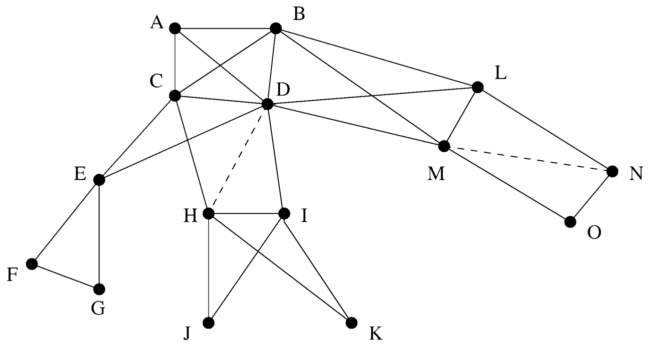

Fig. 2. The constraint graph given in Fig. 1 after its triangulation (dashed lines).

The graph in Fig. 1 is not triangulated. In Fig. 2, a possible triangulation of this graph is provided where the maximum size of cliques is four (see Fig. 3). This is an optimal triangulation, so, the tree-width of this graph is three. In Fig. 4, a tree whose nodes correspond to maximal cliques of the triangulated graph is a possible tree-decomposition for the graph of Fig. 1. So, we get $\mathcal{C}_1 = \{A, B, C, D\}$, $\mathcal{C}_2 = \{C, D, E\}$, $\mathcal{C}_3 = \{E, F, G\}$, $\mathcal{C}_4 = \{C, D, H\}$, $\mathcal{C}_5 = \{D, H, I\}$, $\mathcal{C}_6 = \{H, I, J\}$, $\mathcal{C}_7 = \{H, J, K\}$, $\mathcal{C}_8 = \{B, D, L, M\}$, $\mathcal{C}_9 = \{L, M, N\}$ and $\mathcal{C}_{10} = \{M, N, O\}$.

The CSP decomposition method called *Tree-Clustering*, proposed by Dechter and Pearl [15] is based on these notions (see also [13] for a more recent description); it proceeds by four steps:

1. Triangulate the constraint graph.
2. Find maximal cliques (each clique corresponds to a subproblem).
3. Solve every subproblem induced by the maximal cliques.
4. Solve the new acyclic $n$-ary CSP.

The guiding idea of this method is to provide a systematic scheme, which, from any CSP, produces an equivalent $n$-ary CSP by a covering of the set of constraints in order to

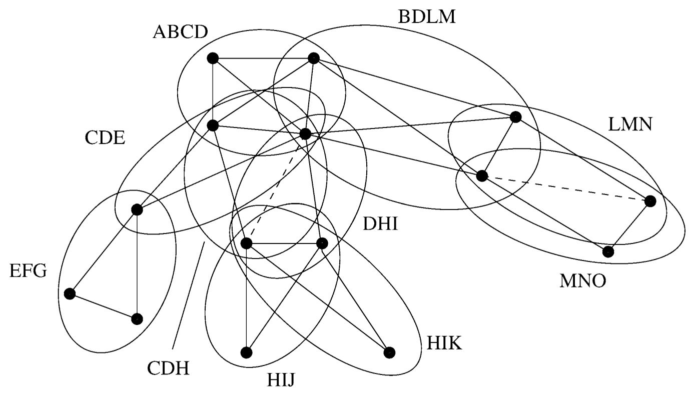

Fig. 3. The acyclic hypergraph induced by maximal cliques of the triangulated graph given in Fig. 2.

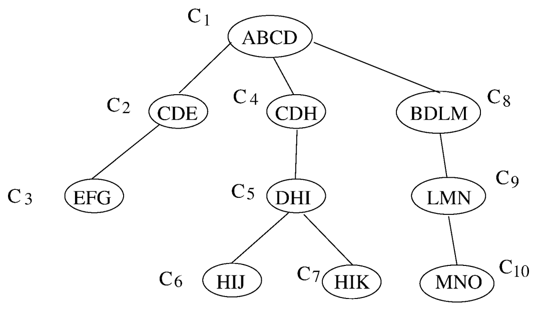

Fig. 4. The tree-decomposition of the triangulated constraint graph given in Fig. 2.

build an acyclic constraint hypergraph. Such a CSP can be solved in polynomial time with respect to the size of the induced $n$-ary CSP.

This method is generally presented [15] using an approximation of the optimal triangulation (some comments about triangulations are given in Section 5). Phases 1 and 2 are feasible in polynomial time, more precisely, in $O(n + m')$ with $m'$ the number of edges of the graph after the triangulation ($m \leqslant m' < n^2$). Moreover, note that the tree associated to the acyclic hypergraph can be computed in linear time, given the maximal cliques. Step 3 is feasible in $O(m.d^{w^+ +1})$ with $w^+$ the size minus one of the biggest produced clique ($w^+ + 1 \leqslant n$). The last step has the same complexity. The space complexity, which is bound to the storage of solutions of subproblems, can be reduced to $O(n.s.d^s)$ with $s$ the maximal size of minimal separators, which equals the size of the biggest intersection between subproblems ($s \leqslant w^+$). Finally, note that for every decomposition which induces a value $w^+$, we have $w \leqslant w^+$ with $w$ the tree-width of the initial constraint graph.

Figs. 1–3 can be considered as an illustration of this method. In Fig. 1, we see a constraint graph. After step 1, the triangulation adds two edges (the dashed lines). A covering of this graph by maximal cliques defines an acyclic hypergraph. Each maximal clique defines a subproblem.

Although theoretically interesting, all the practical interest of this method is not proved yet, even if it is clear that, for some classes of CSP, it can provide an useful approach [13]. One reason of the lack of efficiency of Tree-Clustering is due to the heaviness of the approach, and specially the required space. In the case where all the solutions are searched, it may be useful. In the other hand, if we check the consistency or if we search only one solution, we will prefer to use an enumerative algorithm such as Forward Checking (denoted FC [21], Real Full Look-Ahead (denoted RFLA [27]) or Maintaining Arc-Consistency (denoted MAC [31]), due to the space costs of Tree-Clustering, and to its practical efficiency.

In the next section, we show how the reference to such a structural decomposition allows to establish a search procedure based on enumeration while keeping the complexity bounds given above.

## 3. The BTD method

### 3.1. Presentation

The BTD method (for Backtracking with Tree-Decomposition) proceeds by an enumerative search guided by a static pre-established partial order induced by a tree-decomposition of the constraint-network. So, the first step of BTD consists in computing a tree-decomposition or an approximation of a tree-decomposition. The obtained partial order allows to exploit some structural properties of the graph, during the search, in order to prune some branches of the search tree. Hence, what distinguishes BTD from other techniques concerns the following points:

- the variable instantiation order is induced by a tree-decomposition of the constraint graph,
- some parts of the search space would not be visited again as soon as their consistency is known (notion of *structural good*),
- some parts of the search space would not be visited again if it is known that the current instantiation leads to a failure (notion of *structural nogood*).

Note that this method is called BTD for Backtracking with Tree-Decomposition, but we will see latter that the enumerative search can be implemented with the basic Backtracking, or FC, or MAC (and more sophisticated algorithms).

### 3.2. Theoretical foundations

Let $\mathcal{P} = (X, D, C, R)$ be an instance where $(X, C)$ is a graph, with $\mathcal{A} = (\mathcal{C}, \mathcal{T})$ a tree-decomposition (or an approximation) where $\mathcal{T} = (I, F)$ is a tree. We suppose that the elements of $\mathcal{C} = \{\mathcal{C}_i\colon\ i \in I\}$ are indexed with respect to the notion of *compatible numeration*:

**Definition 2.** A numeration on $\mathcal{C}$ compatible with a prefix numeration of $\mathcal{T} = (I, F)$ with $\mathcal{C}_1$ the root is called *compatible numeration* $N_{\mathcal{C}}$.

Note that the example of tree-decomposition given in Fig. 4 is a compatible numeration on $\mathcal{C}$. We note $Desc(\mathcal{C}_j)$ the set of variables belonging to the union of the descendants $\mathcal{C}_k$ of $\mathcal{C}_j$ in the tree rooted in $\mathcal{C}_j$, $\mathcal{C}_j$ included. For example, $Desc(\mathcal{C}_4) = \mathcal{C}_4 \cup \mathcal{C}_5 \cup \mathcal{C}_6 \cup \mathcal{C}_7 = \{C, D, H, I, J, K\}$. Note that the numeration $N_{\mathcal{C}}$ defines a partial variable ordering that permits to get an enumeration order of the variables of $\mathcal{P}$:

**Definition 3.** An order $\preceq_X$ of variables of $X$ such that $\forall x \in \mathcal{C}_i,\ \forall y \in \mathcal{C}_j$, with $i < j$, $x \preceq_X y$ is a compatible enumeration order.

For example, the alphabetical order $A, B, \ldots, N, O$ is a compatible enumeration order. The tree-decomposition with the numeration $N_C$ permits to clarify some relations in the constraint graph.

**Theorem 1.** *Let $\mathcal{C}_j$ be a son of $\mathcal{C}_i$ (so $i < j$). There does not exist an edge $\{x, y\}$ in the graph $(X, C)$ where $x \in (\bigcup_{k=1}^{j-1} \mathcal{C}_k) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ and $y \in Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$.*

**Proof.** By construction, $\mathcal{C}_i \cap \mathcal{C}_j$ is clearly a separator of the graph which disconnects $(\bigcup_{k=1}^{j-1} \mathcal{C}_k) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$ and $Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)$. □

For example, let $i = 1$, $j = 4$, and $\mathcal{C}_4$ be a son of $\mathcal{C}_1$. There is no edge in $G$ between $(\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3) \backslash (\mathcal{C}_1 \cap \mathcal{C}_4) = \{A, B, C, D, E, F, G\} \backslash \{C, D\} = \{A, B, E, F, G\}$ and $Desc(\mathcal{C}_4) \backslash (\mathcal{C}_1 \cap \mathcal{C}_4) = \{C, D, H, I, J, K\} \backslash \{C, D\} = \{H, I, J, K\}$.

In terms of CSP, there is no constraint joining these two subsets of variables and therefore these two subproblems. Consequently, the compatibility relations between instantiations pass only through the separator $\mathcal{C}_i \cap \mathcal{C}_j$.

The BTD method is based on compatible enumeration order and this first theorem. Let us consider a consistent instantiation $\mathcal{A}$ of variables of $\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_i \cup \mathcal{C}_{i+1} \cup \cdots \cup \mathcal{C}_{j-1}$, with $\mathcal{C}_j$ a son of $\mathcal{C}_i$. Due to the definition of compatible orders, the enumeration continues with the variables of the lineage $Desc(\mathcal{C}_j)$ except ones which belong to $\mathcal{C}_i \cap \mathcal{C}_j$. Then two cases arise depending on whether a consistent extension of the current instantiation on $Desc(\mathcal{C}_j)$ exists or not:

- *There is no consistent extension*. In such a case, the reason of the inconsistency can only be the unsatisfaction of some constraints which join two variables of $Desc(\mathcal{C}_j)$ or (not exclusive or) a variable of this set and a variable which precedes it in the order, so which belongs to $\mathcal{C}_i \cap \mathcal{C}_j$ (see Theorem 1). In both case, if a new consistent assignment $\mathcal{A}'$ such that $\mathcal{A}'$ and $\mathcal{A}$ are equal on $\mathcal{C}_i \cap \mathcal{C}_j$ is tried, its extension on $Desc(\mathcal{C}_j)$ will lead to the same failure, independently of what precedes. In fact, the instantiation restricted to $\mathcal{C}_i \cap \mathcal{C}_j$ may be considered as a *nogood* in the usual sense of the term, although, here, it is found by structural criteria. This nogood can be recorded and exploited during next searches.

- *There exists a consistent extension*. By a similar reasoning to previous one, we can prove that every instantiation which is the same on $\mathcal{C}_i \cap \mathcal{C}_j$ will lead to a success on $Desc(\mathcal{C}_j)$, because it is independent of what precedes. This assignment can be now considered as a *good* in the sense that on a part of the problem, $Desc(\mathcal{C}_j)$, this assignment has a consistent extension. Like nogoods, goods may be recorded and used during further searches, allowing to jump in the search tree (*forward-jumping*), what leads to continue the enumeration with the variables located after ones of $Desc(\mathcal{C}_j)$ in the compatible enumeration order.

The closest works of our approach are ones of Bayardo and Miranker in [4] whose study is limited to the resolution of binary CSPs whose constraint graph is a tree. Our approach can be considered as a generalization of their work since their goods and nogoods instantiate variables while our goods and nogoods instantiate sets of variables (separators). In [5], Bayardo and Miranker propose another generalization of goods and nogoods which is not based on separators but on sets of ancestors in an ordered constraint graph. Formally, their work is different though their use of goods and nogoods during search is similar to ours (see Section 6 for more details).

Now, we formally introduce goods and nogoods based on separators.

**Definition 4.** Given $\mathcal{C}_i$ and $\mathcal{C}_j$ one of its sons, a **good** (respectively **nogood**) of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$, noted $g(\mathcal{C}_i/\mathcal{C}_j)$ (respectively $ng(\mathcal{C}_i/\mathcal{C}_j)$), is a consistent assignment $\mathcal{A}$ of $\mathcal{C}_i \cap \mathcal{C}_j$ such that there exists (respectively does not exist) a consistent extension of $\mathcal{A}$ on $Desc(\mathcal{C}_j)$.

The following Lemma 1 and its corollary show that the interactions between a subproblem rooted in $\mathcal{C}_j$ and the remaining of the CSP pass through the intersection between $\mathcal{C}_j$ and its father $\mathcal{C}_i$. These properties are at the origin of the cuttings (for the nogoods) and the jumps (for the goods) which will be realized in the tree search.

**Lemma 1.** *Given $\mathcal{C}_i$ and $\mathcal{C}_j$ one of its sons, given $Y \subset X$ such that $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, every consistent instantiation $\mathcal{B}$ of $Desc(\mathcal{C}_j)$ is compatible with every consistent instantiation $\mathcal{A}$ of $Y$ iff $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$.*

**Proof.** According to Theorem 1 and by construction, the only constraints joining the variables of $Y$ to the variables of $Desc(\mathcal{C}_j)$ are the constraints which involve the variables common to $Desc(\mathcal{C}_j)$ and to $Y$, i.e., $\mathcal{C}_i \cap \mathcal{C}_j$. It results that $\mathcal{A}$ and $\mathcal{B}$ are compatible iff each common variable has the same value in $\mathcal{A}$ and $\mathcal{B}$ (i.e., $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$). □

It ensues the following corollary:

**Corollary 1.** *Given $\mathcal{C}_i$ and $\mathcal{C}_j$ one of its sons, every consistent instantiation $\mathcal{B}$ of $Desc(\mathcal{C}_j)$ is compatible with every consistent instantiation $\mathcal{A}$ of $(X \backslash Desc(\mathcal{C}_j)) \cup \mathcal{C}_i$ iff $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$.*

We then formalize the exploitation of goods:

**Lemma 2** (jump by the goods). *Given $\mathcal{C}_i$ and $\mathcal{C}_j$ one of its sons, given $Y \subset X$ such that $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, for all $g(\mathcal{C}_i/\mathcal{C}_j)$, every consistent instantiation $\mathcal{A}$ of $Y$ such that $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g(\mathcal{C}_i/\mathcal{C}_j)$ has a consistent extension on $Desc(\mathcal{C}_j)$.*

**Proof.** Let $\mathcal{A}$ be a consistent instantiation such that $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g(\mathcal{C}_i/\mathcal{C}_j)$. According to the definition of goods, there exists an instantiation $\mathcal{B}$ on $Desc(\mathcal{C}_j)$ such that $\mathcal{B}$ is consistent and $\mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j] = g(\mathcal{C}_i/\mathcal{C}_j)$. As $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = g(\mathcal{C}_i/\mathcal{C}_j) = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$, $\mathcal{A}$ and $\mathcal{B}$ are compatible (according to Lemma 1). Therefore, $\mathcal{B}$ is a consistent extension of $\mathcal{A}$ on $Desc(\mathcal{C}_j)$. □

Thus, if a partial instantiation $\mathcal{A}$ is such that $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ is a good of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$, then it is not necessary to extend the search on $Desc(\mathcal{C}_j)$. So the enumeration goes on with the variables of the first $\mathcal{C}_k$ located out of $Desc(\mathcal{C}_j)$, for instance the next brother of $\mathcal{C}_j$, if there exists one.

**Lemma 3** (cutting by the nogoods). *Given $\mathcal{C}_i$ and $\mathcal{C}_j$ one of its sons, given $Y \subset X$ such that $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$, for all $ng(\mathcal{C}_i/\mathcal{C}_j)$, there is no assignment $\mathcal{A}$ of $Y$ such that $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = ng(\mathcal{C}_i/\mathcal{C}_j)$ and such that $\mathcal{A}$ has a consistent extension on $Desc(\mathcal{C}_j)$.*

**Proof.** According to the definition of a nogood, there is no extension of $ng(\mathcal{C}_i/\mathcal{C}_j)$ on $Desc(\mathcal{C}_j)$. As $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = ng(\mathcal{C}_i/\mathcal{C}_j)$, $\mathcal{A}$ cannot be extended on $Desc(\mathcal{C}_j)$. □

## *3.3. The basic algorithm*

The method obtained from these notions can be implemented in several ways according to whether a filtering is associated or not with the enumeration. However, the mechanisms will be similar. The BTD method explores the search space by using a compatible order $\preceq_X$, which begins with the variables of $\mathcal{C}_1$. Inside $\mathcal{C}_i$, the enumeration works in classical way. On the other hand, when all the variables are assigned by satisfying all the involved constraints, we then get a consistent instantiation $\mathcal{A}$ of variables of $\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_i$. The search must go on with the variables of the first son $\mathcal{C}_{i+1}$ of $\mathcal{C}_i$ if there exists one. More generally, let us consider the case of one son $\mathcal{C}_j$ of $\mathcal{C}_i$. We check if $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ is a good or a nogood and we take appropriate action:

- In the case of a nogood, we change the current instantiation on $\mathcal{C}_i$.
- In the case of a good, a "forward-jump" happens in order to continue the enumeration with the first variable located after those of $Desc(\mathcal{C}_j)$. Fig. 5 illustrates the case of a forward-jump, assuming that $\mathcal{A}[\mathcal{C}_4 \cap \mathcal{C}_5] = \mathcal{A}[\{D, H\}]$ is a good. We show in part (a) the jump in a compatible enumeration order, and in part (b), where the search goes on in the structure of the instance.
- In the other cases, i.e., $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ is neither a good nor a nogood, $\mathcal{A}$ must be extended in consistent way on the variables of $Desc(\mathcal{C}_j)$. If so, $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ is recorded as a good; on the contrary, if $\mathcal{A}$ cannot be extended in consistent way, the nogood $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ is recorded.

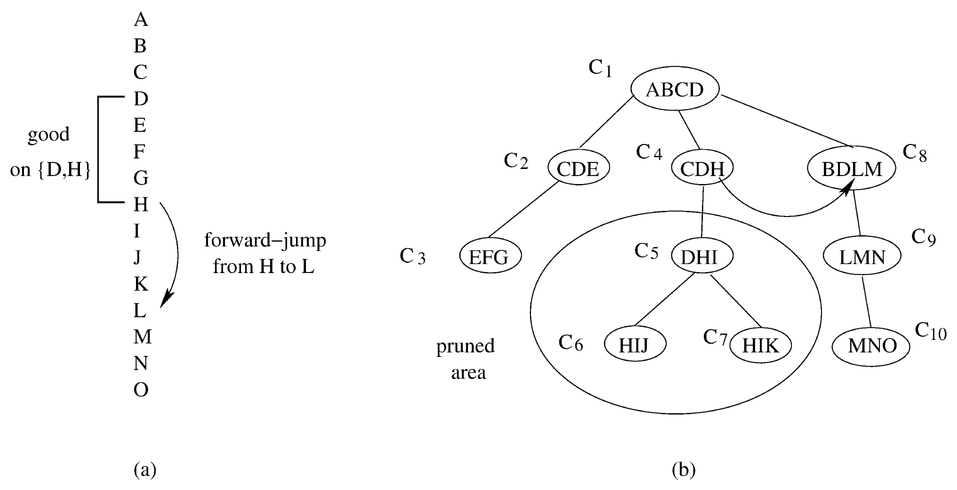

Fig. 5. Example of a forward-jump with a good $\mathcal{A}[\mathcal{C}_4 \cap \mathcal{C}_5]$ on $\{D, H\}$. In (a), we show the jump along the enumeration order, while in (b) we see the jump in the structure of the problem.

Fig. 6 describes the BTD algorithm restricted to the consistency check: it returns **True** if the consistent instantiation $\mathcal{A}$ can be extended to a consistent instantiation on $V_{\mathcal{C}_i}$ and on all the descents of $\mathcal{C}_i$; **False** otherwise. $V_{\mathcal{C}_i}$ represents the set of unassigned variables of $\mathcal{C}_i$ and $G$ and $N$ respectively the set of recorded goods and of nogoods. This algorithm is run after having computed a tree-decomposition (or an approximation) of the constraint graph.

**Theorem 2.** *BTD is sound, complete and terminates.*

**Proof.** This algorithm is proved by induction, exploiting properties of structural goods and nogoods. The induction is made on the number of variables appearing in the lineage of $\mathcal{C}_i$ except the already assigned variables of $\mathcal{C}_i$. This set of variables is denoted

$$VAR(\mathcal{C}_i, V_{\mathcal{C}_i}) = V_{\mathcal{C}_i} \cup \left( \bigcup_{\mathcal{C}_j \in Sons(\mathcal{C}_i)} \big(Desc(\mathcal{C}_j) \backslash (\mathcal{C}_i \cap \mathcal{C}_j)\big) \right).$$

$VAR(\mathcal{C}_i, V_{\mathcal{C}_i})$ is then the set of variables to assign to know whether $\mathcal{A}$ can be extended to a consistent assignment on $V_{\mathcal{C}_i}$ and its lineage.

To prove BTD, we must prove the property $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ defined as:

“BTD($\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}$) returns true if the consistent assignment $\mathcal{A}$ can be extended to a consistent assignment on $V_{\mathcal{C}_i}$ and the lineage of $\mathcal{C}_i$; otherwise, BTD returns false”.

Consider $P(\mathcal{A}, \emptyset)$:

If $VAR(\mathcal{C}_i, V_{\mathcal{C}_i}) = \emptyset$, then $V_{\mathcal{C}_i} = \emptyset$ and $Sons(\mathcal{C}_i) = \emptyset$. Since $\mathcal{A}$ is a consistent assignment, $\mathcal{A}$ can be extended to a consistent assignment on $V_{\mathcal{C}_i}$ and on the lineage of $\mathcal{C}_i$. Therefore $P(\mathcal{C}_i, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ is true.

Induction step: $P(\mathcal{A}, S)$ with $S \neq \emptyset$. Suppose that $\forall S' \subset S$, $P(\mathcal{A}, S')$ holds.

```
1.  BTD($\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}$)
2.  If $V_{\mathcal{C}_i} = \emptyset$
3.  Then
4.      If $Sons(\mathcal{C}_i) = \emptyset$ Then Return True
5.      Else
6.          $Consistency \leftarrow$ True
7.          $F \leftarrow Sons(\mathcal{C}_i)$
8.          While $F \neq \emptyset$ and $Consistency$ Do
9.              Choose $\mathcal{C}_j$ in $F$
10              $F \leftarrow F \backslash \{\mathcal{C}_j\}$
11.             If $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ is a good of $\mathcal{C}_i/\mathcal{C}_j$ in $G$ Then $Consistency \leftarrow$ True
12.             Else
13.                 If $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ is a nogood of $\mathcal{C}_i/\mathcal{C}_j$ in $N$ Then $Consistency \leftarrow$ False
14.                 Else
15.                     $Consistency \leftarrow BTD(\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i))$
16.                     If $Consistency$
17.                     Then Record the good $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ of $\mathcal{C}_i/\mathcal{C}_j$ in $G$
18.                     Else Record the nogood $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ of $\mathcal{C}_i/\mathcal{C}_j$ in $N$
19.                 EndIf
20.             EndIf
21.         EndWhile
22.         Return $Consistency$
23.     EndIf
24. Else
25.     Choose $x \in V_{\mathcal{C}_i}$
26.     $d_x \leftarrow D_x$
27.     $Consistency \leftarrow$ False
28.     While $d_x \neq \emptyset$ and $\neg Consistency$ Do
29.         Choose $v$ in $d_x$
30.         $d_x \leftarrow d_x \backslash \{v\}$
31.         If $\nexists c \in C$ such that $c$ is not satisfied by $\mathcal{A} \cup \{x \leftarrow v\}$
32.         Then $Consistency \leftarrow BTD(\mathcal{A} \cup \{x \leftarrow v\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\})$
33.         EndIf
34.     EndWhile
35.     Return $Consistency$
36. EndIf
```

Fig. 6. The BTD algorithm.

– If $V_{\mathcal{C}_i} \neq \emptyset$:
During the **While** loop (lines 28–34) the assertion: “there is no value $v$ of $x$ already checked such that $\mathcal{A}$ extended by that value leads to a consistent assignment for $V_{\mathcal{C}_i}$ and the lineage of $\mathcal{C}_i$” is true.
If BTD is called (line 32), $\mathcal{A} \cup \{x \leftarrow v\}$ is then consistent (since no constraint is violated) and $VAR(\mathcal{C}_i, V_{\mathcal{C}_i \backslash \{x\}}) \subset VAR(\mathcal{C}_i, V_{\mathcal{C}_i})$. According to the induction hypothesis, the assignment $\mathcal{A}$ has been extended if $BTD(\mathcal{A} \cup \{x \leftarrow v\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\})$ is true. In that case, $BTD(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i})$ returns true and $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ is satisfied.

After the loop (line 35), all the possible values have been tried without consistent extension of $\mathcal{A}$. Therefore, $BTD(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i})$ returns false and $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ is satisfied.

– If $V_{\mathcal{C}_i} = \emptyset$:
During the **While** loop (lines 8–21) the assertion: “for each son $\mathcal{C}_f$ already checked, $\mathcal{A}$ can be extended to a consistent assignment on $Desc(\mathcal{C}_f)$” holds.
We show that this assertion is true at the end of the loop.
Let $\mathcal{C}_j$ be a son of $\mathcal{C}_i$ to be examined.
  + If $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ is a good of $\mathcal{C}_i/\mathcal{C}_j$, by Lemma 2, we know that $\mathcal{A}$ can be extended on $Desc(\mathcal{C}_j)$. Therefore, the assertion is true at the end of the loop.
  + If $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ is a nogood of $\mathcal{C}_i/\mathcal{C}_j$, by Lemma 3, we know that $\mathcal{A}$ cannot be extended on $Desc(\mathcal{C}_j)$. The loop is then finished.
  + If $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ is neither a good, nor a nogood, then, BTD is called with $\mathcal{A}$ which is a consistent assignment and $VAR(\mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i)) \subset VAR(\mathcal{C}_i, \emptyset)$. So, according to the induction hypothesis, BTD$(\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i))$ returns true if $\mathcal{A}$ admits a consistent assignment on $Desc(\mathcal{C}_j)$, and then the assertion is verified. Otherwise, the loop is stopped.
  After the loop (line 22), BTD$(\mathcal{A}, \mathcal{C}_i, \emptyset)$ returns true if $\mathcal{A}$ has been consistently extended on every son, and returns false otherwise.
  Therefore, $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$ is satisfied. Note that the memorization of goods and nogoods is justified by their definition.

To summarize, since BTD satisfies $P(\mathcal{A}, VAR(\mathcal{C}_i, V_{\mathcal{C}_i}))$, in particular BTD satisfies the property $P(\emptyset, VAR(\mathcal{C}_1, \mathcal{C}_1))$ for the first call, and then BTD is sound, complete and terminates. □

### *3.4. Extensions of BTD*

We now discuss extensions of the BTD algorithm presented in the previous section. It is based on Chronological Backtracking. It is well known that this algorithm is not efficient in practice. So, its natural extensions which generally exploit lookahead techniques like arc-consistency or forward-checking must be integrated to the BTD approach.

Thus, we introduce two extensions based on filterings:

- **FC-BTD** which is BTD using the classical filtering used in Forward-Checking [21].
- **MAC-BTD** which is BTD using an arc-consistency filtering [31].

These extensions are straightforward if the used filtering does not modify the structure of the constraint network. Indeed, a more powerful filtering like path-consistency [26] [25] applied during search is not possible because new edges can be added to the constraint network, modifying its structural properties with consequences on the properties of BTD. So that for FC-BTD, the correctness of the extension is trivial, for MAC-BTD this extension is straightforward but we consider it must be established by the next property:

**Theorem 3.** *Let $\mathcal{C}_j$ be a son of $\mathcal{C}_i$ and let $\mathcal{A}$ be a consistent assignment on $\bigcup_{k=1}^{j-1} \mathcal{C}_k$. Assume that the arc-consistent closure of the CSP $\mathcal{P}$ after the assignment $\mathcal{A}$ (denoted $AC(\mathcal{P}, \mathcal{A})$) has no empty domains. If $g$ is a good of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$ in $\mathcal{P}$ such that $g = \mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$, then $g$ is a good in $AC(\mathcal{P}, \mathcal{A})$.*

**Proof.** Let $\mathcal{B}$ be a consistent assignment on $Desc(\mathcal{C}_i)$ associated to the good $g$. That is $\mathcal{B}$ is a solution of the subproblem of $\mathcal{P}$ induced by the variables occurring in $Desc(\mathcal{C}_i)$. Therefore, we get $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$. By definition, $\mathcal{B}$ satisfies all the constraints belonging to $Desc(\mathcal{C}_j)$. Moreover, all the values in $\mathcal{A}$ are compatible with all the values in $\mathcal{B}$. Indeed, the constraints between $\mathcal{A}$ and $\mathcal{B}$ associate pairs of variables $\{x_i, x_j\}$ such that $x_i \in \mathcal{C}_i$ and $x_j \in \mathcal{C}_j$. Then, three cases exist:

(1) $x_j \in \mathcal{C}_i$. Therefore, since $\mathcal{A}$ is a consistent assignment, $\mathcal{A}$ satisfies the constraints occurring in $\mathcal{C}_i$ especially $\{x_i, x_j\}$.
(2) $x_i \in \mathcal{C}_j$. Therefore, since $\mathcal{B}$ is a consistent assignment, $\mathcal{B}$ satisfies the constraints occurring in $\mathcal{C}_j$ especially $\{x_i, x_j\}$.
(3) $x_i, x_j \in \mathcal{C}_i \cap \mathcal{C}_j$ which is a particular case of the upper cases.

Therefore, the assignment defined by the assignment $\mathcal{A}$ extended by $\mathcal{B}$, that is $\mathcal{A} \cup \mathcal{B}$, is a solution of the subproblem defined by the variables appearing in $\mathcal{A}$ or in $Desc(\mathcal{C}_j)$ since all the constraints are satisfied. Thus, $\mathcal{A} \cup \mathcal{B}$ is a consistent assignment, and then the values in $\mathcal{B}$ necessarily appear in $AC(\mathcal{P}, \mathcal{A})$. □

Another way to improve backtracking search consists in using a non-chronological backtracking like *Backjumping* classically denoted **BJ** [17]. Backjumping allows us to define three immediate extensions of BTD:

- **BTD-BJ** which is BTD using Backjumping.
- **FC-BTD-BJ** which is BTD using the classical filtering used in Forward-Checking and Backjumping.
- **MAC-BTD-BJ** which is BTD using an arc-consistency filtering and Backjumping.

BTD-BJ (respectively FC-BTD-BJ and MAC-BTD-BJ) is similar to BTD (respectively FC-BTD and MAC-BTD) with an additional phase of backjump. This phase of backjump is achieved when BTD comes back to the cluster $\mathcal{C}_i$ after a failure during the search for an extension of the current instantiation over the descent of $\mathcal{C}_i$ rooted in a son $\mathcal{C}_j$ of $\mathcal{C}_i$. It consists in coming back to the deepest variable which belongs both to $\mathcal{C}_i$ and $\mathcal{C}_j$.

Finally, note that the BTD algorithm only builds a consistent instantiation which can be extended to a solution of the treated instance, if one exists. Indeed, some variables are unassigned due to the jumps realized thanks to goods. Nonetheless, it is easy to extend the produced assignment to a solution of the problem by using a backtracking search and by checking the recorded goods and nogoods as new constraints. Note that this extension does not change anything to the complexity bounds provided in the next section.

## 4. Time and space complexities

In this section, we first assess the time and space complexities of the BTD algorithm. Then, we compare BTD with the Chronological Backtracking and the Tree-Clustering. Note that these results also hold if we consider more sophisticated backtracking search as FC or MAC. Let us assume that a tree-decomposition or its approximation has been computed.

We begin by evaluating the space complexity of BTD:

**Theorem 4.** *BTD has a space complexity in* $\mathrm{O}(n.s.d^s)$ *where* $s$ *is the size of the largest intersection* $\mathcal{C}_i \cap \mathcal{C}_j$ *with* $\mathcal{C}_j$ *son of* $\mathcal{C}_i$.

**Proof.** BTD only records the goods and the nogoods. Goods and nogoods are instantiations on the intersections $\mathcal{C}_i \cap \mathcal{C}_j$ with $\mathcal{C}_j$ son of $\mathcal{C}_i$. Therefore, if $s$ is the size of the largest of these intersections, BTD has a space complexity in $\mathrm{O}(n.s.d^s)$ because the number of intersections $\mathcal{C}_i \cap \mathcal{C}_j$ is bounded by $n$ while the number of goods and nogoods associated to one intersection is bounded by $d^s$ and the size of a good or a nogood is at most $s$. □

Next, we calculate the time complexity of BTD.

**Theorem 5.** *BTD has a time complexity in* $\mathrm{O}(n.s^2.m.\log(d^s).d^{w^+ +1})$ *with* $w^+ + 1$ *the size of the largest* $\mathcal{C}_i$ *and* $s$ *the size of the largest intersection* $\mathcal{C}_i \cap \mathcal{C}_j$ *with* $\mathcal{C}_j$ *son of* $\mathcal{C}_i$.

**Proof.** Assume that we want to extend an instantiation on $\mathcal{C}_j$. There exist two cases:

– Either $\mathcal{C}_j = \mathcal{C}_1$, and then find the consistent instantiations on $\mathcal{C}_j$ has a worst-case time complexity in $\mathrm{O}(m.d^{|\mathcal{C}_j|})$. Note that $m$ is due to the number of constraints to check to ensure consistency.

– Or $\mathcal{C}_j$ is a son of $\mathcal{C}_i$. Let $\mathcal{A}$ be a consistent assignment on $Y$ ($Y \subset X$ such that $Desc(\mathcal{C}_j) \cap Y = \mathcal{C}_i \cap \mathcal{C}_j$).
  Find the consistent extensions of $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ on $\mathcal{C}_j$ has a worst-case time complexity in $\mathrm{O}(m.d^{|\mathcal{C}_j \backslash \mathcal{C}_j \cap \mathcal{C}_i|})$.
  BTD searches the extension of $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$ once and only once (thanks to recorded goods and nogoods). As there exist at most $d^{|\mathcal{C}_i \cap \mathcal{C}_j|}$ assignments $\mathcal{A}[\mathcal{C}_j \cap \mathcal{C}_i]$, the worst-case time complexity of finding the extension on $\mathcal{C}_j$ is in $\mathrm{O}(d^{|\mathcal{C}_j|})$.

Therefore, if $w^+ + 1$ is the size of the largest $\mathcal{C}_i$, the search of an extension by BTD has a complexity in $\mathrm{O}(n.m.d^{w^+ +1})$, to which must be added the cost of managing and exploiting goods and nogoods. As this cost is zero for $\mathcal{C}_1$, we focus on the case where $\mathcal{C}_j$ is a son of $\mathcal{C}_i$. The comparison between $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ and a recorded good (or nogood) requires $\mathrm{O}(|\mathcal{C}_i \cap \mathcal{C}_j|)$ steps. The addition or the search of a good (or a nogood) is in $\mathrm{O}(|\mathcal{C}_i \cap \mathcal{C}_j| \log(d^{|\mathcal{C}_i \cap \mathcal{C}_j|}))$. So the management and the exploitation of goods and nogoods have a complexity in

$O(d^{|\mathcal{C}_i|}|\mathcal{C}_i \cap \mathcal{C}_j|\log(d^{|\mathcal{C}_i \cap \mathcal{C}_j|}))$, given $\mathcal{C}_i$ and one of its sons $\mathcal{C}_j$. Therefore, on the overall search, it has a cost in $O(n.s.m.d^{w^+ +1}\log(d^s))$.

Thus, the time complexity of BTD is $O(n.m.d^{w^+ +1} + n.s.m.d^{w^+ +1}\log(d^s))$, i.e., a complexity in $O(n.s^2.m.\log(d^s).d^{w^+ +1})$. □

The time and space complexities of BTD are comparable to ones of Tree-Clustering. We now show that BTD develops fewer nodes (or as many nodes in the worst case) than Chronological Backtracking (denoted BT) and than Tree-Clustering (denoted TC). In order to do these comparisons, we consider that BT and TC use the same variables/values order as BTD and TC must exploit the same tree-decomposition as BTD. Using compatible orders allows to compare easily BT with BTD. Nevertheless, it is clear that a compatible order is not necessarily a good variable order for BT. A more general comparison between BTD and BT (FC and MAC too), requires to study different orders. So, this analysis should be extended in the future to consider different orders. We first compare BTD and BT:

**Theorem 6.** *Given a compatible order, BTD develops at most as many nodes as BT.*

**Proof.** Using goods and nogoods permits BTD to avoid some redundancies in the tree search. So BTD develops at most as many nodes as BT. □

Like BT, BTD stops as soon as the problem's consistency is found. In the other hand, TC builds every consistent assignment on $\mathcal{C}_i$, for each $\mathcal{C}_i$. Furthermore, when BTD does not develop a consistent instantiation on $\mathcal{C}_i$, it ensues a saving in number of nodes on all the descent of $\mathcal{C}_i$.

And so, the next theorem shows the gain in nodes of BTD with respect to TC:

**Theorem 7.** *Given a compatible order, BTD develops at most as many nodes as TC, which uses BT for solving each* $\mathcal{C}_i$.

**Proof.** BTD and TC develop in the worst case the same number of nodes for $\mathcal{C}_1$. For all other $\mathcal{C}_j$ ($j \neq 1$), TC searches systematically all consistent assignments on $\mathcal{C}_j$, whereas BTD only builds the consistent instantiations on $\mathcal{C}_j$ which are compatible with the current instantiation on $\mathcal{C}_i$, the father of $\mathcal{C}_j$. Thus, BTD develops at most as many nodes as TC. □

Finally, to conclude this section, note that if we put FC or MAC instead of BT, Theorem 6 still holds. Moreover, for time complexity, we get the theoretical complexity time by multiplying the cost by a factor due to the cost of one filtering, in the same spirit as the complexity analysis proposed in [24].

## 5. Experimental results

The following experiments are carried out with a view to assessing the interest of a method like BTD. The first experiments concern networks whose tree-width is not necessarily small. For them, we hope that BTD is as efficient as any classical enumerative algorithms. The second experiments work on structured CSPs: we hope that BTD will exploit efficiently topological properties of the network when these properties are related to tree-decomposition, that is CSP with small tree-width. Finally, we assess the behaviour of our method on some real-world instances.

### *5.1. About implementation*

#### *5.1.1. The implemented algorithms*

We implement different versions of BTD. The first version, noted FC-BTD, corresponds to a simple implementation of the BTD algorithm based on the Forward-Checking algorithm. The second version, noted FC-BTD-BJ, is FC-BTD with the additional phase of backjump (see Section 3.4 for more details). The last two versions, noted respectively FC-BTD$^-$ and FC-BTD-BJ$^-$, respectively correspond to FC-BTD and FC-BTD-BJ without the recording of the goods and nogoods. We need these versions to assess the contribution of goods and nogoods. In other words, these versions correspond to Forward Checking where the choice of the next variable to instantiate is partly guided by a compatible enumeration order of BTD. Likewise, we define the MAC based versions of BTD.

We implement several algorithms in order to compare them with the different versions of BTD. We use FC [21], Forward-Checking with Conflict-directed BackJumping (denoted FC-CBJ [28]), and MAC [31]. For MAC, arc-consistency is achieved thanks to the AC-2001 algorithm [8], which has an optimal worst-case time complexity.

For the purpose of comparing the number of developed nodes and the space requirements of BTD and Tree-Clustering, we implement a partial version of Tree-Clustering. By partial version, we mean that we only compute all solutions of each cluster. We do not solve the acyclic CSP obtained from the previous computation because this step presents no interest for our comparisons. We note TC-FC our partial implementation of Tree-Clustering based on the Forward-Checking algorithm. Of course, BTD and TC-FC exploit the same tree-decomposition (or the same approximation). Finally, note that we only assess the required memory for TC-FC without recording any partial instantiation because we would need too much space.

#### *5.1.2. Heuristic for choosing the next variable to instantiate*

For choosing the next variable to instantiate, all the algorithms in this study use the heuristic *dom*/*deg* [7]. This heuristic is one of the best heuristics for ordering variables. According to this heuristic, the next variable to instantiate is the variable $x_i$ which minimizes the ratio $|D_i|/|\Gamma_i|$ with $D_i$ the current domain of $x_i$ and $\Gamma_i$ the set of the neighbours of $x_i$. We select the next variable:

– among all the unassigned variables of the problem for FC, FC-CBJ or MAC,
– among all the unassigned variables of the current cluster for the different versions of BTD.

Note that the different versions of FC-BTD (respectively MAC-BTD) use exactly the same variable ordering.

### *5.1.3. Approximation of a tree-decomposition by triangulation*

As the problem of finding a tree-decomposition is NP-Hard, we only use an approximation of a tree-decomposition by triangulating the constraint graph.

We try several algorithms for triangulating the constraint graph among the *LEX-M* algorithm [30], the *LB-TRIANG* algorithm [2] and the *Fill-in Computation* algorithm [35]. The first two algorithms produce a minimal triangulation (a triangulation $E'$ of a graph $G = (V, E)$ is minimal if there is no triangulation $E''$ such that $E'' \subset E'$). They have a time complexity in $O(nm)$ with $n$ the number of vertices and $m$ one of edges of the graph, whereas the time complexity of the *Fill-in Computation* algorithm is linear in $O(n + m')$ ($m'$ is the number of edges of the triangulated graph). The experimentations on classical random problems show that the *LEX-M* algorithm provides the best results for BTD. So, for all the following results, we use the *LEX-M* algorithm to compute a triangulation.

From this triangulation, if we compute an approximation of a tree-decomposition, we obtain that cliques and separators have on average a reasonable size, that is to say, the time and the memory needed by BTD are feasible in practice. On the contrary, the largest separator size may be too important, that is to say BTD may request too much memory. So, to prevent this problem, we propose to limit the size of separators by a given parameter $s_{\max}$, like in [13]. This trade-off is made to the detriment of the size of clusters and so of the time. First we compute normally the clique-tree. Then, we traverse the tree in breadth first search. If the son $\mathcal{C}_j$ has an intersection with its parent $\mathcal{C}_i$ whose size is less than $s_{\max}$, the son and its parent remain unchanged. Else, we merge the parent $\mathcal{C}_i$ and its son $\mathcal{C}_j$. The obtained cluster replaces $\mathcal{C}_i$ in the tree (so we call this cluster $\mathcal{C}_i$). Furthermore, the sons of $\mathcal{C}_j$ become the sons of $\mathcal{C}_i$. Finally, note that these modifications do not change the size of the intersection between $\mathcal{C}_i$ and the brothers of $\mathcal{C}_j$.

For the provided results, we limit the separator size to 5. For this size, the separator size is neither too small, nor too large.

## *5.2. The experimental protocol*

The following experimentations are realized on a Linux-based PC with an Intel Pentium III 550 MHz processor and 256 Mb of memory. We set a one hour time limit for determining whether a problem is consistent or not. Beyond one hour, the search is stopped and the problem's consistency is said unknown. The given run-time includes the time of the preliminary treatments (like computing an approximation of a tree-decomposition).

We work on random binary CSPs generated according to two models and on real-world instances.

### *5.2.1. Classical random CSPs*

In order to produce classical random instances, we use the random generator written by D. Frost, C. Bessière, R. Dechter and J.-C. Régin. This generator[1] takes 4 parameters $n$, $d$, $m$ and $T$. It builds a CSP of class $(n, d, m, T)$ with $n$ variables, each having a domain of size $d$, and $m$ binary constraints ($0 \leqslant m \leqslant n(n-1)/2$) in which $T$ tuples are forbidden

[1] Downloadable at http://www.lirmm.fr/∼bessiere/generator.html.

$(0 \leqslant T \leqslant d^2)$. Among the CSPs produced by this generator, we keep only those whose constraint graph is connected.

The listed results are the averages of results obtained on 100 problems per class. We experiment on random instances with 50 variables and domains of size 15 and whose constraint graph has a density between 10% and 30%. We also test some problems with a larger domain from the class $(50, 25, 123, 439)$. Considered classes are close to the satisfiability's threshold.

### 5.2.2. *Structured random CSPs*

We define a new binary CSPs random generator, which produces instances with a structured constraint graph. The constraint graph is triangulated. This property allows us to exactly know the tree-width of the constraint network, and then to know the theoretical complexity bound. This generator takes 5 parameters $n$, $d$, $r_{\max}$, $T$ and $s_{\max}$. It builds a binary CSP of the class $(n, d, r_{\max}, T, s_{\max})$ with $n$ variables which have domains of size $d$ and whose constraint graph has the following properties:

– each vertex $v$ belongs at least to a maximal clique with a size greater than 1,
– the cliques have a size at most $r_{\max}$,
– the intersection between two cliques has a size at most $s_{\max}$,
– the cliques form a clique-tree and then the graph is triangulated.

To build such a problem, we first choose a set of $r_{\max}$ variables to form the root clique. Then, while there are remaining variables, we proceed like this:

(1) choose randomly a parent clique $\mathcal{C}_i$,
(2) choose randomly a size of the intersection between $\mathcal{C}_i$ and its son $\mathcal{C}_j$ (the size is bounded by 1 and $s_{\max}$),
(3) choose randomly a size of the clique $\mathcal{C}_j$ (the size is at least 3 and bounded by the size of the intersection plus 1 and $r_{\max}$),
(4) choose randomly the variables of $\mathcal{C}_i$ which belong to the separator.

We associate to each constraint a relation in which $T$ tuples are forbidden $(0 \leqslant T \leqslant d^2)$. An important drawback of this generator is that the number of constraints depends on the produced problem. For each class $(n, d, r_{\max}, T, s_{\max})$, we solve 100 problems and present the average of obtained results. The given results correspond to problems of the classes $(50, 25, 15, T, 5)$ with $T$ between 265 and 281. Theses classes are near the satisfiability's threshold.

### 5.2.3. *Real-world instances*

We experiment our algorithm on some real-world instances of the CELAR from the FullRLFAP archive.[2] These instances correspond to radio link frequency assignment problems. For more details, they are described in [10]. Note that solving the problems

[2] We thank the Centre d'Electronique de l'Armement (France).

SCEN#01 and SCEN#08 requires a special adaptation of our implementation of BTD because these problems have a constraint graph with several connected components.

## 5.3. Experimental results for classical random CSPs

### 5.3.1. Comparisons of the different versions of BTD

Before comparing BTD to some classical algorithms like FC or MAC, we study the behaviour of our algorithm. First, we assess the contribution of backjumping by counting the number of nodes developed by FC-BTD which are not visited by FC-BTD-BJ. We observe there is no gain for most classes and a slight one for classes $(50, 15, 123, 141)$ or $(50, 25, 123, 439)$ (the classes we use are given in Table 1). But, even if there is a gain, it is insignificant. As a good or a nogood is recorded each time BTD comes back from a cluster to its parent, we can say, according to the little number of recorded goods and nogoods, that FC-BTD and FC-BTD-BJ rarely visit the descendants of the root cluster. Therefore, the phase of backjumping is seldom used, which explains that FC-BTD and FC-BTD-BJ obtain similar or equal results for classical random problems.

Then, we measure the contribution of goods and nogoods by counting the number of nodes developed by FC-BTD-BJ$^-$ which are not visited by FC-BTD-BJ. Like the previous comparison, there is no gain or a slight one. Indeed only a few goods or nogoods are used by FC-BTD-BJ to prune the search because of the little number of recorded goods and nogoods. And so FC-BTD-BJ$^-$ and FC-BTD-BJ present similar results. For information, we obtain similar results with MAC-BTD. As the various versions of BTD based on FC (respectively on MAC) obtain similar results, for the following comparisons on classical random problems, we only present the results of FC-BTD-BJ (respectively MAC-BTD-BJ).

### 5.3.2. Comparisons between FC-BTD-BJ and FC and between MAC-BTD-BJ and MAC

Table 1 presents the number of nodes and of constraint checks and the run-time for FC and FC-BTD-BJ. We observe that FC-BTD-BJ and FC are comparable. And even, for some classes, FC-BTD-BJ improves the results of FC, by developing fewer nodes and realizing fewer constraint checks than FC.

Similar results are obtained with MAC and MAC-BTD-BJ, as shown in Table 2.

Table 1
(Classical random CSPs.) Number of nodes, and number of constraint checks and run-time (in milliseconds) for FC and FC-BTD-BJ

| Class | FC | | | FC-BTD-BJ | | |
|---|---|---|---|---|---|---|
| | # nodes | # checks | time | # nodes | #checks | time |
| $(50, 15, 123, 141)$ | 15,884 | 458,342 | 250 | 19,417 | 541,178 | 263 |
| $(50, 15, 184, 112)$ | 223,588 | 7,346,620 | 3,775 | 229,901 | 7,521,911 | 3,490 |
| $(50, 15, 245, 93)$ | 1,742,077 | 64,695,274 | 31,613 | 1,690,389 | 62,741,411 | 28,045 |
| $(50, 15, 306, 78)$ | 6,695,576 | 275,447,261 | 130,334 | 6,516,523 | 268,222,843 | 122,202 |
| $(50, 15, 368, 68)$ | 19,899,917 | 865,863,076 | 410,365 | 20,202,681 | 880,491,613 | 374,439 |
| $(50, 25, 123, 439)$ | 148,793 | 5,968,598 | 3,164 | 183,304 | 7,106,934 | 3,416 |

Table 2
(Classical random CSPs.) Number of nodes, and number of constraint checks and run-time (in milliseconds) for MAC and MAC-BTD-BJ

| Class | MAC | | | MAC-BTD-BJ | | |
|---|---|---|---|---|---|---|
| | # nodes | # checks | time | # nodes | # checks | time |
| (50, 15, 123, 141) | 433 | 211,854 | 158 | 426 | 212,751 | 160 |
| (50, 15, 184, 112) | 10,570 | 4,749,549 | 4,366 | 10,589 | 4,767,163 | 4,468 |
| (50, 15, 245, 93) | 115,272 | 53,354,043 | 55,693 | 111,641 | 51,618,005 | 52,203 |
| (50, 15, 306, 78) | 577,928 | 263,294,873 | 293,339 | 560,541 | 255,317,033 | 279,650 |
| (50, 15, 368, 68) | 2,024,325 | 936,053,949 | 1,082,427 | 2,053,352 | 948,798,297 | 1,101,599 |
| (50, 25, 123, 439) | 2,912 | 2,600,557 | 1,767 | 2,703 | 2,476,033 | 1,674 |

Table 3
(Classical random CSPs.) Number of nodes, and number of constraint checks and run-time (in milliseconds) for FC-CBJ

| Class | FC-CBJ | | | FC-BTD-BJ | | |
|---|---|---|---|---|---|---|
| | # nodes | # checks | time | # nodes | # checks | time |
| (50, 15, 123, 141) | 13,820 | 407,967 | 285 | 19,417 | 541,178 | 263 |
| (50, 15, 184, 112) | 214,314 | 7,089,277 | 4,657 | 229,901 | 7,521,911 | 3,490 |
| (50, 15, 245, 93) | 1,707,839 | 63,628,692 | 39,310 | 1,690,389 | 62,741,411 | 28,045 |
| (50, 15, 306, 78) | 6,612,237 | 272,582,414 | 160,745 | 6,516,523 | 268,222,843 | 122,202 |
| (50, 15, 368, 68) | 19,722,533 | 859,100,282 | 504,513 | 20,202,681 | 880,491,613 | 374,439 |
| (50, 25, 123, 439) | 127,093 | 5,208,464 | 3,613 | 183,304 | 7,106,934 | 3,416 |

### *5.3.3. Comparisons between FC-BTD-BJ and FC-CBJ*

As FC-BTD-BJ exploits backjumping and “forwardjumping”, we compare our algorithm with a classical backjumping algorithm, namely FC-CBJ. Table 3 provides the number of nodes, of constraint checks and the run-time for FC-CBJ. We observe that FC-CBJ often develops fewer nodes than FC-BTD-BJ. However, if we consider the run-time, we note that FC-BTD-BJ is faster than FC-CBJ for all the classes. A partial explanation of such a result is the cost of the computation of the conflicts which is too expensive compared to the number of saved nodes.

### *5.3.4. Comparisons between BTD and Tree-Clustering*

We compare the space requirements for FC-BTD-BJ and our partial version of Tree-Clustering. In order to measure the memory requirement, we count one unit per assigned value contained in the recorded partial instantiation. For example, recording a good about five variables requires five units. Table 4 presents the memory required by FC-BTD-BJ (for recording goods and nogoods), the memory required by TC-FC (for recording consistent instantiations respectively on separators and on clusters), the number of developed nodes and the run-time (in milliseconds) for TC-FC. We observe that TC-FC requires significantly more memory than FC-BTD-BJ, because FC-BTD-BJ records only a part of the goods which TC-FC memorizes. Note that for some classes like (50, 25, 123, 439), TC-FC requires too much memory in practice. Furthermore, TC-FC develops significantly

Table 4
(Classical random CSPs.) Comparison between FC-BTD-BJ and Tree-Clustering based on FC

| Class | FC-BTD-BJ | TC-FC | | | |
|---|---|---|---|---|---|
| | memory | memory | | # nodes | time |
| | | separator | cluster | | |
| (50, 15, 123, 141) | 24.7 | 219,402 | 406,212,164 | 155,668,480 | 62,994 |
| (50, 15, 184, 112) | 9.9 | 163,523 | 1,840,482 | 942,758 | 8,752 |
| (50, 15, 245, 93) | 1.3 | 33,217 | 401,269 | 2,438,672 | 38,894 |
| (50, 15, 306, 78) | 0.5 | 11,620 | 199,244 | 12,932,108 | 226,546 |
| (50, 15, 368, 68) | 0.1 | 7,052 | 53,470 | 25,859,906 | 492,491 |
| (50, 25, 123, 439) | 19.2 | 1,560,479 | 375,943,617 | 89,379,304 | 106,367 |

more nodes and is slower than FC-BTD-BJ. So it seems difficult to use Tree-Clustering in practice.

### 5.3.5. *Summary*

FC-BTD and MAC-BTD obtain results which are comparable with ones of FC (or FC-CBJ) and MAC respectively. It seems difficult to use Tree-Clustering in practice, due to the required space.

## 5.4. *Experimental results with structured random CSPs*

### 5.4.1. *Comparisons of the different versions of BTD*

Like for classical problems, before making a comparison between BTD and classical algorithms like FC, FC-CBJ or MAC, we study the behaviour of our algorithm. First, with a view to comparing FC-BTD and FC-BTD-BJ, we assess the contribution of the backjumping by counting the number of nodes developed by FC-BTD which are not visited by FC-BTD-BJ. Fig. 7 presents the number of nodes developed by FC-BTD and FC-BTD-BJ. We note on this figure that FC-BTD-BJ develops significantly fewer nodes than FC-BTD. The economy in term of number of nodes varies between 8% and 26%. However, using the backjumping has a cost. Indeed, according to Fig. 8 (which reports the run-time for FC-BTD and FC-BTD-BJ), we observe that the gain in time is slightly less important than one in nodes. It is bounded by 5% and 19%.

In order to assess the contribution of goods and nogoods, we count the number of nodes developed by FC-BTD-BJ$^-$ which are not visited by FC-BTD-BJ. According to Fig. 9, it turns out that FC-BTD-BJ always develops fewer nodes than FC-BTD-BJ$^-$ and the gain is very important in some cases, namely near the satisfiability's threshold. The two algorithms differ only in recording and using goods and nogoods. It ensues that the gain in nodes is obtained thanks to the use of goods and nogoods. This gain leads to an economy in time, as shown in Fig. 10 (which presents the run-time for FC-BTD-BJ and FC-BTD-BJ$^-$).

Similar experimentations are realized with FC-BTD and FC-BTD$^-$. First, it results from these experimentations that FC-BTD$^-$ is unable to solve some instances in one hour. Table 5 gives their number. Therefore, in order to compare FC-BTD and FC-BTD$^-$, we take into account the problems solved by FC-BTD$^-$. Fig. 11 shows the number of nodes developed by FC-BTD and FC-BTD$^-$.

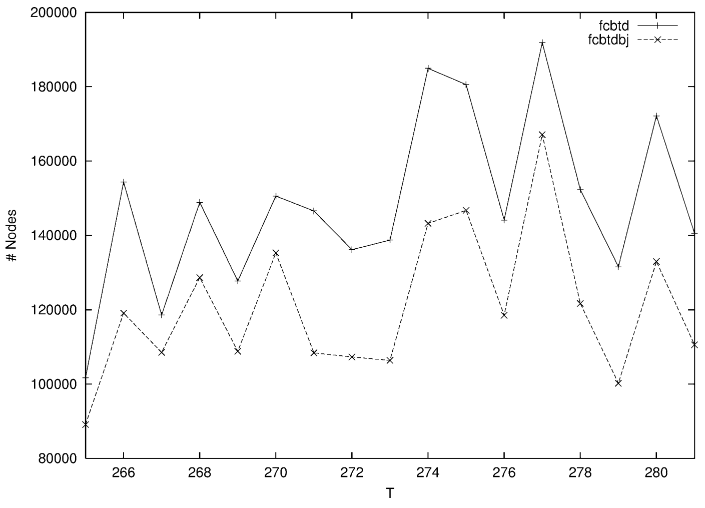

Fig. 7. (Structured random CSPs $(50, 25, 15, T, 5)$.) Number of nodes developed by FC-BTD and FC-BTD-BJ.

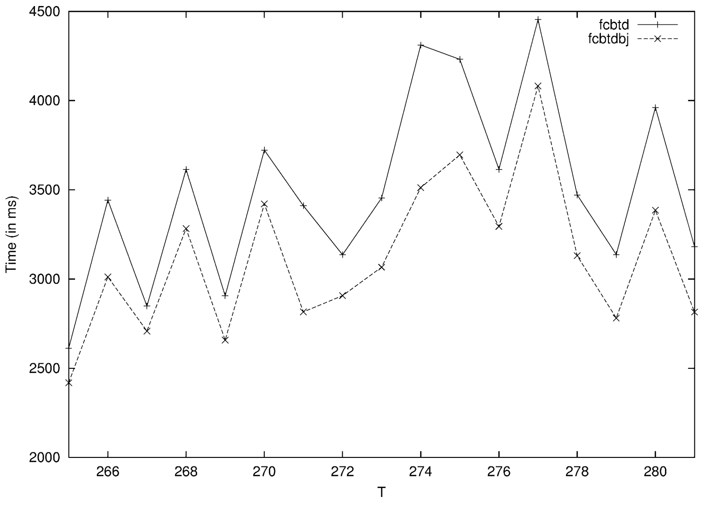

Fig. 8. (Structured random CSPs $(50, 25, 15, T, 5)$.) Run-time (in milliseconds) for FC-BTD and FC-BTD-BJ.

Then, we observe that FC-BTD develops fewer nodes than FC-BTD$^-$, thanks to the use of goods and nogoods. Furthermore, the difference between FC-BTD and FC-BTD$^-$ is more important than one between FC-BTD-BJ and FC-BTD-BJ$^-$. This gap highlights a lot of redundancies in the search tree developed by FC-BTD$^-$, which underlines all the more the contribution of goods and nogoods and/or of the phase of backjumping (because FC-BTD-BJ$^-$ is not so penalized as FC-BTD$^-$).

According to the previous results, we focus our study on FC-BTD-BJ for the next comparisons.

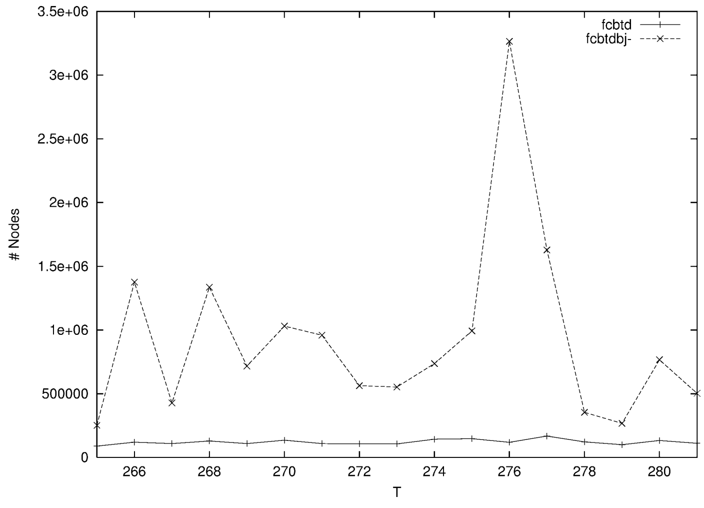

Fig. 9. (Structured random CSPs $(50, 25, 15, T, 5)$.) Number of nodes developed by FC-BTD-BJ and FC-BTD-BJ$^{-}$.

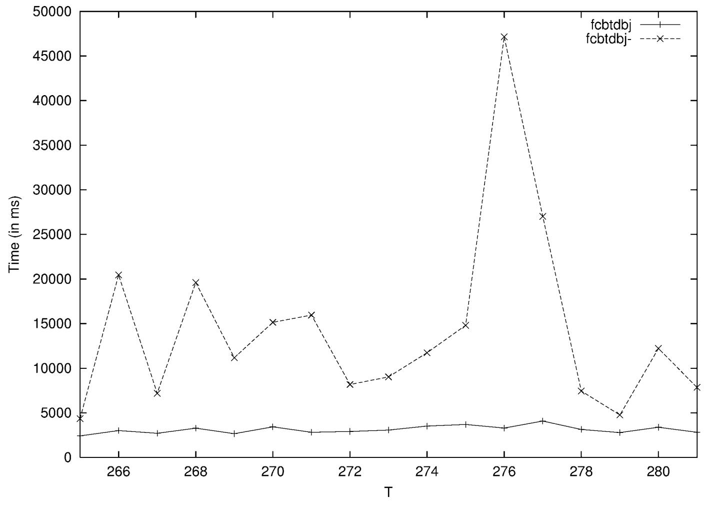

Fig. 10. (Structured random CSPs $(50, 25, 15, T, 5)$.) Run-time (in milliseconds) for FC-BTD-BJ and FC-BTD-BJ$^{-}$.

### *5.4.2. Comparisons between FC-BTD-BJ and FC and between MAC-BTD-BJ and MAC*

FC and MAC are unable to solve some problems in one hour. Hence, in order to compare FC (respectively MAC) and FC-BTD-BJ (respectively MAC-BTD-BJ) we consider only the instances which FC (respectively MAC) can solve in one hour. Table 5 gives the number of problems solved by FC (respectively MAC). Note that FC-BTD-BJ and MAC-BTD-BJ solve all the considered instances.

Table 5
(Structured random CSPs $(50, 25, 15, T, 5)$.) Number of consistent (C), inconsistent (I) and unknown (U) problems

| $T$ | FC-BTD | | FC-BTD$^-$ | | | FC | | | MAC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | C | I | C | I | U | C | I | U | C | I | U |
| 265 | 70 | 30 | 70 | 30 | 0 | 67 | 30 | 3 | 67 | 30 | 3 |
| 266 | 61 | 39 | 60 | 38 | 2 | 56 | 39 | 5 | 55 | 35 | 10 |
| 267 | 63 | 37 | 62 | 36 | 2 | 61 | 36 | 3 | 60 | 36 | 4 |
| 268 | 57 | 43 | 56 | 42 | 2 | 55 | 42 | 3 | 54 | 42 | 4 |
| 269 | 63 | 37 | 62 | 37 | 1 | 58 | 35 | 7 | 54 | 35 | 11 |
| 270 | 60 | 40 | 60 | 39 | 1 | 53 | 40 | 7 | 53 | 39 | 8 |
| 271 | 53 | 47 | 51 | 46 | 3 | 46 | 47 | 7 | 43 | 47 | 10 |
| 272 | 51 | 49 | 49 | 48 | 3 | 47 | 49 | 4 | 44 | 49 | 7 |
| 273 | 51 | 49 | 50 | 46 | 4 | 45 | 45 | 10 | 44 | 45 | 11 |
| 274 | 39 | 61 | 38 | 60 | 2 | 34 | 61 | 5 | 32 | 60 | 8 |
| 275 | 37 | 63 | 35 | 62 | 3 | 32 | 60 | 8 | 31 | 58 | 11 |
| 276 | 29 | 71 | 26 | 70 | 4 | 27 | 71 | 2 | 24 | 71 | 5 |
| 277 | 39 | 61 | 36 | 57 | 7 | 34 | 61 | 5 | 33 | 59 | 8 |
| 278 | 35 | 65 | 33 | 65 | 2 | 26 | 64 | 10 | 25 | 64 | 11 |
| 279 | 41 | 59 | 39 | 57 | 4 | 35 | 57 | 8 | 33 | 56 | 11 |
| 280 | 24 | 76 | 24 | 72 | 4 | 21 | 75 | 4 | 20 | 75 | 5 |
| 281 | 27 | 73 | 26 | 72 | 2 | 25 | 72 | 3 | 24 | 72 | 4 |

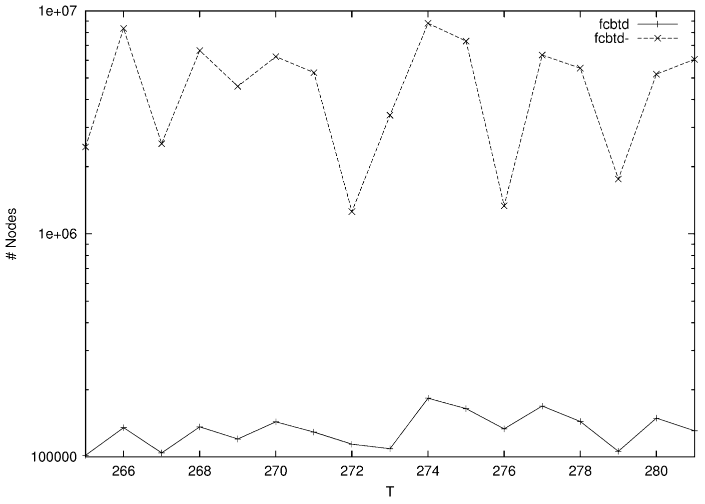

Fig. 11. (Structured random CSPs $(50, 25, 15, T, 5)$.) Number of nodes developed by FC-BTD and FC-BTD$^-$ (with a log scale).

Fig. 12 presents the run-time for FC, FC-BTD-BJ and FC-BTD-BJ$^-$. We note that FC-BTD-BJ is significantly faster than FC. Indeed, the ratio of the run-time for FC over one for FC-BTD-BJ is between 7 and 24. We save time not only thanks to the goods and nogoods, but also thanks to the backjumping. Indeed, the contribution of the backjumping is proved by the run-time for FC-BTD-BJ$^-$, which is better than one of FC in most cases.

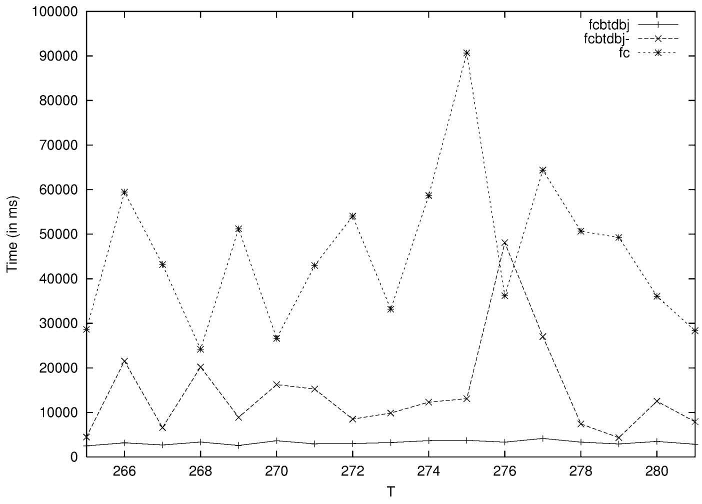

Fig. 12. (Structured random CSPs $(50, 25, 15, T, 5)$.) Run-time (in milliseconds) for FC-BTD-BJ, FC-BTD-BJ$^-$ and FC.

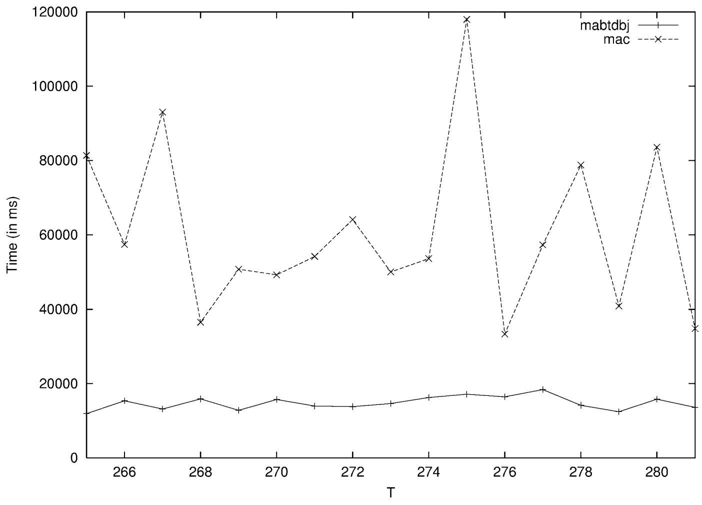

Fig. 13. (Structured random CSPs $(50, 25, 15, T, 5)$.) Run-time (in milliseconds) for MAC and MAC-BTD-BJ.

We obtain similar results when we compare MAC and MAC-BTD-BJ, as is shown by Fig. 13. MAC-BTD-BJ is between 2 and 7 times as fast as MAC.

### *5.4.3. Comparisons between FC-BTD-BJ and FC-CBJ*

As FC-BTD-BJ uses backjumping and "forwardjumping", we have to compare FC-BTD-BJ with an algorithm which exploits backjumping like FC-CBJ. Figs. 14 and 15 present the number of nodes and the run-time for FC-CBJ and FC-BTD-BJ. About the number of nodes, neither FC-CBJ nor FC-BTD-BJ is always better than the other one.

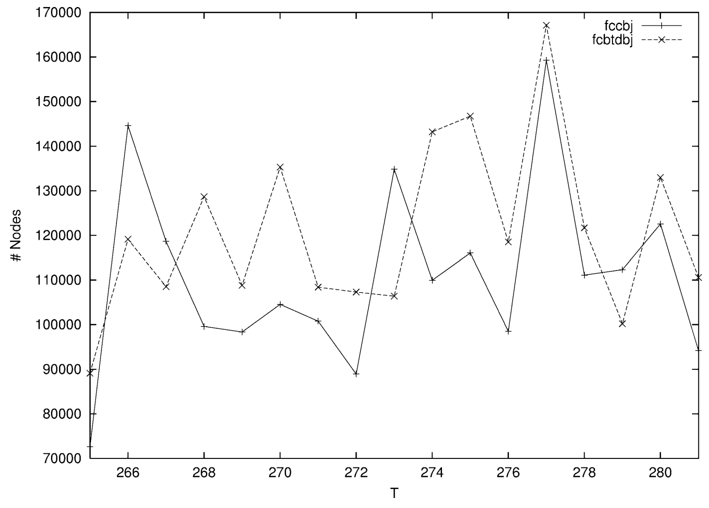

Fig. 14. (Structured random CSPs $(50, 25, 15, T, 5)$.) Number of nodes developed by FC-CBJ and FC-BTD-BJ.

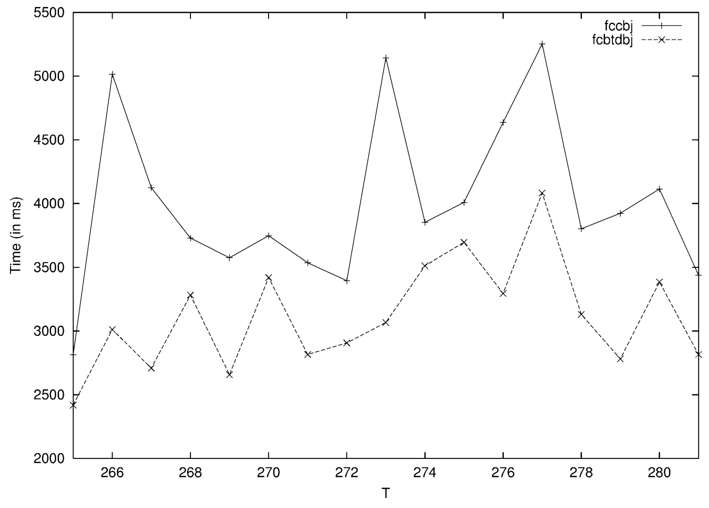

Fig. 15. (Structured random CSPs $(50, 25, 15, T, 5)$.) Run-time (in milliseconds) for FC-CBJ and FC-BTD-BJ.

Nonetheless, FC-BTD-BJ is always faster than FC-CBJ. This difference in time is mostly explained by the cost of the computation of conflicts in FC-CBJ which is too important in comparison with the gain obtained thanks to backjumping.

### *5.4.4. Comparisons between BTD and Tree-Clustering*

Like for classical random problems, we compare the space requirements for FC-BTD-BJ and our partial version of Tree-Clustering. Table 4 shows the memory requirement of FC-BTD-BJ (for recording good ands nogoods), the memory requirement of TC-FC (for recording consistent instantiations respectively on separators and on clusters), the

Table 6
(Structured random CSPs $(50, 25, 15, T, 5)$.) Memory requirements for FC-BTD-BJ and Tree-Clustering based on FC

| $T$ | FC-BTD-BJ | TC-FC | | | |
|---|---|---|---|---|---|
| | memory | memory | | # nodes | time |
| | | separator | cluster | | |
| 265 | 3,599 | 563,376 | 16,269,923 | 4,329,007 | 14,718 |
| 266 | 4,054 | 544,683 | 15,741,988 | 4,081,041 | 13,270 |
| 267 | 3,471 | 391,140 | 13,536,010 | 3,630,053 | 12,486 |
| 268 | 4,174 | 500,454 | 14,331,723 | 3,779,950 | 12,439 |
| 269 | 3,218 | 426,517 | 12,862,473 | 3,477,518 | 11,743 |
| 270 | 5,567 | 457,120 | 13,071,460 | 3,505,728 | 11,652 |
| 271 | 5,005 | 413,560 | 13,010,768 | 3,451,125 | 11,170 |
| 272 | 4,273 | 453,395 | 11,745,237 | 3,192,403 | 10,770 |
| 273 | 5,476 | 401,098 | 11,476,495 | 3,083,502 | 10,286 |
| 274 | 9,008 | 444,808 | 10,237,218 | 2,805,207 | 9,750 |
| 275 | 5,289 | 393,569 | 9,107,353 | 2,575,782 | 9,324 |
| 276 | 5,134 | 342,977 | 8,301,716 | 2,385,563 | 8,921 |
| 277 | 8,408 | 379,848 | 9,794,940 | 2,712,032 | 9,246 |
| 278 | 5,910 | 350,243 | 8,589,484 | 2,384,354 | 8,370 |
| 279 | 6,734 | 416,477 | 8,265,270 | 2,307,709 | 7,917 |
| 280 | 8,304 | 319,735 | 7,267,237 | 2,066,851 | 7,398 |
| 281 | 5,637 | 247,736 | 6,232,299 | 1,790,943 | 6,554 |

number of developed nodes and the run-time (in milliseconds) for TC-FC. We observe that FC-BTD-BJ outperforms TC-FC by requiring significantly less memory. Furthermore, it develops fewer nodes and is faster than TC-FC. So, the use of Tree-Clustering seems difficult in practice.

### *5.4.5. Summary*

Among the different versions of FC-BTD (respectively MAC-BTD), the best one is FC-BTD-BJ (respectively MAC-BTD-BJ). FC-BTD-BJ and MAC-BTD-BJ are significantly faster than FC and MAC respectively. Note that FC and MAC are unable to solve some instances. FC-BTD-BJ is faster than FC-CBJ although they develop comparable number of nodes. FC-BTD-BJ requires fewer memory is faster than TC-FC.

## *5.5. Real-world instances*

Table 7 presents the results obtained for some instances of the CELAR from the FullRLFAP archive. In several cases, MAC-BTD-BJ realizes either fewer constraint checks than MAC or as many as MAC, except for the SCEN#02 instance for which MAC-BTD-BJ does a few additional checks. About the run-time, MAC-BTD-BJ and MAC are comparable, except for the SCEN#05 instance. For this instance, MAC-BTD-BJ is significantly faster than MAC thanks to its reduced number of constraint checks.

We do not give any results about TC-FC because TC-FC is unable to find all solutions of the root cluster for all problems except the obviously inconsistent ones.

Table 7
(Real-world Instances.) Number of constraint checks and run-time (in milliseconds) of MAC and MAC-BTD-BJ for some instances of the FullRLFAP archive

| Instance | MAC | | MAC-BTD-BJ | |
|---|---|---|---|---|
| | # checks | time | # checks | time |
| SCEN#01 | 1,857,660 | 610 | 1,855,040 | 790 |
| SCEN#02 | 427,104 | 120 | 427,306 | 150 |
| SCEN#03 | 947,199 | 300 | 930,909 | 400 |
| SCEN#04 | 246,034 | 90 | 246,013 | 120 |
| SCEN#05 | 9,220,866 | 15,380 | 1,190,682 | 210 |
| SCEN#06 | 691,367 | 90 | 691,367 | 80 |
| SCEN#07 | 1,123,856 | 110 | 1,123,856 | 110 |
| SCEN#08 | 2,346,455 | 240 | 2,346,455 | 230 |
| SCEN#09 | 84 | 10 | 84 | 10 |
| SCEN#10 | 84 | 10 | 84 | 10 |
| SCEN#11 | 22,520,823 | 25,520 | 22,513,770 | 25,230 |

### 5.6. *Summary about experimental results*

In this section, we have presented experiments on three kinds of CSPs benchmarks:

- Classical random CSPs.
- Structural random CSPs.
- Real-world instances.

For the first class, BTD, that is FC-BTD or MAC-BTD, obtains similar results than FC or MAC. So, the exploitation of the structure does not slow down the efficiency of search. For structured random CSPs, we have observed a significant improvement of the search in using FC-BTD (respectively MAC-BTD) with respect to FC (respectively MAC). We also have observed that FC-CBJ develops as many nodes as FC-BTD, but FC-BTD is faster. Finally, on real-world instances, BTD obtains either better results than classical algorithms, or comparable ones.

For these different kinds of benchmarks, we have observed that Tree-Clustering cannot be run for two reasons. On the one hand, its practical time complexity is too high. On the other hand, the required space is really prohibitive, making this method untractable while this criterion does not constitute a problem for BTD.

To conclude, BTD seems to be an approach which can exploit structural features of CSPs, without the drawbacks of other structural decomposition methods related to space complexity.

## 6. Related works

We can classify related works in three principal trends:

- Backtracking exploiting structural goods and nogoods as in Bayardo and Miranker [4, 5].
- Tree-Clustering [15] and its theoretical improvements [19].
- Hybrid approaches trying compromise between Tree-Clustering (or adaptive consistency [15]) and Backtracking [13,24].

As indicated in the presentation of BTD (see Section 3.2), the closest works are ones of Bayardo and Miranker in [4] and in [5]. Note that our approach can be considered as a natural generalization of [4] since their study is limited to acyclic binary CSPs (trees). With respect to [5], while the exploitation of goods and nogoods is similar to ours, our notions of goods and nogoods are formally different. In [5], a good (or a nogood) is defined with respect to a variable $x_i$ and to an ordering on vertices. A good (or a nogood) is an assignment of a set of variables which precede $x_i$ in the ordering and are connected to at least one variable belonging to the descendants of $x_i$ in the tree-decomposition. This definition is thus formally different from ours. For example, if we consider a triangulated constraint graph, and $x_i \in \mathcal{C}_j$, the last variable in $\mathcal{C}_j$, then a good (or a nogood) will be an assignment of $\mathcal{C}_j \backslash \{x_i\}$. Then, the space requirement of Learning-Tree-Solve (the algorithm of [5]) will be $\mathrm{O}(n.d^{w^+ + 1})$ ($w^+ + 1$ is the size of the largest $\mathcal{C}_j$) while the space requirement of BTD is limited to $\mathrm{O}(n.d^s)$ with $s$ the size of the largest separator. The time complexity of Learning-Tree-Solve is $\mathrm{O}(\exp(w^+ + 1))$ like BTD. Note that these comments do not constitute an analysis but present some elements for a comparison that indicate the formal difference between these methods.

Finally, the practical interest of Learning-Tree-Solve is not presented in [5]. Moreover, in [6], Bayardo and Pehoushek recall the practical advantages on exploiting nogoods for consistency checking. Nevertheless they have also evoked the difficulty to implement efficiently this notion of goods which is not realized neither in [5] nor in [6].

The work of Baget and Tognetti [9] can be considered as a similar approach. Indeed, in their method, clusters are defined by biconnected components, and then goods and nogoods (they do not use these expressions) are limited to the assignment of one variable, the one which separates biconnected components. The time complexity of their method is then $\mathrm{O}(n.d^k)$ with $k$ the maximum size of biconnected components. In this case, $w^+ + 1 \leqslant k$. If we consider the constraint graph in Fig. 1, we get two biconnected components, $\{E, F, G\}$ and $\{A, B, C, D, H, I, J, K, L, M, N, O\}$, and then, $k = 12$ while $w^+ = 3$. Nevertheless, Baget and Tognetti indicated a few ways to improve their approach exploiting a generalization to $k$-connected components. Note that no experimental result is presented in [9].

BTD is principally based on tree-decomposition. So, works which have been developed like Tree-Clustering and its improvements are interesting for our purpose. In [19], an improvement of Tree-Clustering is presented while a theoretical comparison between decomposition methods is given. These results may indicate ways for (theoretical) improvements of BTD but we are not sure of their practical effects.

BTD can be considered as an hybrid approach realizing a tradeoff between practical time and space complexity. In [13], Dechter and El Fattah present a time-space tradeoff scheme. This scheme allows them to propose a spectrum of algorithms such that tree-clustering and cycle-cutset conditioning (linear for space complexity) are two extremes in this spectrum. Another interesting idea in their work is the possibility to modify the size of separators to minimize space. We have exploited this idea in Section 5 to minimize the size of separators. Finally, note that their experimental results are limited to the valuation of structural parameters ($w^+$ and $s$) on real-world structured instances (combinatorial circuits), and then no result on the efficiency in solving these instances is presented.

In [24], Larrosa proposes an hybrid method based on Adaptive Consistency [15] and on Backtracking (or FC or MAC). Adaptive Consistency (AdCons) relies on the general scheme of variable elimination which replace sets of variables by new constraints which summarize the effects of eliminated variables. AdCons has the same bounds as Tree-Clustering for time and space complexities. So, exponential space complexity limits severely the algorithm usefulness. The idea of Larrosa consists in limiting the size of the new constraints produced by AdCons to a parameter $k$. If larger arity constraints should be produced, then it switches to search (BT, FC, MAC, ...). This hybrid approach allows to bound the required space to $\mathrm{O}(d^k)$ but the time complexity is now $\mathrm{O}(\exp(z(k) + k + 1))$. Here $z(k)$ is a structural parameter induced by $k$ and the width of the constraint graph such that $z(k) + k < n$. Note that for sparse constraint graphs (6 per cent), and limited values of $k$ ($k = 2$), the author obtains interesting results on random CSPs.

## 7. Summary and conclusion

The CSP formalism offers a powerful framework for representing and solving efficiently many problems. Generally, CSPs are solved applying tree search algorithms which use optimizations of backtracking and then obtain good experimental results. However, since CSP is a NP-complete problem, there are no better bound for theoretical time complexity than the size of the search space, which is exponential. On the contrary, methods which offer better bounds for time complexity—which are generally based on tree-decomposition of CSPs—have not proved yet their practical efficiency. This paper presents a framework—BTD—for solving CSPs. BTD is based both on backtracking techniques and on the notion of tree-decomposition of the constraint network.

We have shown that BTD inherits the advantages of the two other approaches: the practical efficiency of backtracking algorithms, and a warranty of limited time/space complexity. In Section 4, we have proved that the theoretical time and space complexities of BTD are similar to Tree-Clustering's ones, namely a time complexity in $\mathrm{O}(n.s^2.m.\log(d^s).d^{w^+ +1})$ and a space complexity in $\mathrm{O}(n.s.d^s)$. Moreover, experiments allow us to show that:

- BTD is as efficient as classical algorithms on classical random problems, in some cases, it is even better,
- on structured random problems, BTD presents a significant gain thanks to the exploitation of goods and nogoods,
- on real-world instances, BTD obtains either better results than classical algorithms, or comparable ones,
- about required space, BTD can be used in practice, unlike Tree-Clustering which is too expensive in memory.

Among the potential extensions of this method, the first one concerns the generalization to $n$-ary CSPs, which should not raise much difficulty, because it is immediately obtained by construction. A more promising extension is related to optimization tasks. In fact, if we consider, for instance, the valued CSP framework [32], methods like Russian Dolls Search [36] or the dynamic programming approach [22] are among the most efficient ones. These methods record and exploit some informations they explicit during the search. Now, if we exploit a method like BTD which limits the number of recorded informations, we can expect significant gains in practice. Finally, the theoretical comparison between BTD and BT (respectively FC-BTD vs FC and MAC-BTD vs MAC) should be extended in the future to consider different orders.

## References

[1] S. Arnborg, D. Corneil, A. Proskuroswki, Complexity of finding embedding in a $k$-tree, SIAM J. Discrete Math. 8 (1987) 277–284.

[2] A. Berry, A wide-range efficient algorithm for minimal triangulation, in: Proceedings of SODA'99 SIAM Conference, 1999.

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

[4] R.J. Bayardo, D.P. Miranker, An optimal backtrack algorithm for tree-structured constraint satisfaction problems, Artificial Intelligence 71 (1994) 159–181.

[5] R.J. Bayardo, D.P. Miranker, A complexity analysis of space-bounded learning algorithms for the constraints satisfaction problem, in: Proceedings of 13th National Conference on Artificial Intelligence, Portland, OR, 1996, pp. 298–304.

[6] R. Bayardo, J. Pehoushek, Counting models using connected components, in: Proceedings of AAAI 2000, Austin, TX, 2000, pp. 157–162.

[7] C. Bessière, J.-C. Régin, MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems, in: Proceedings of CP'96, 1996, pp. 61–75.

[8] C. Bessière, J.-C. Régin, Refining the basic constraint propagation algorithm, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence, Seattle, WA, 2001, pp. 309–315.

[9] J.-F. Baget, Y. Tognetti, Backtracking through biconnected components of a constraint graph, in: Proceedings of the 17th International Joint Conference on Artificial Intelligence, Seattle, WA, 2001, pp. 291–296.

[10] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, J.P. Warners, Radio link frequency assignment, Constraints 4 (1999) 79–89.

[11] X. Chen, P. van Beek, Conflict-directed backjumping revisited, J. Artificial Intelligence Res. 14 (2001) 53–81.

[12] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition, Artificial Intelligence 41 (1990) 273–312.

[13] R. Dechter, Y. El Fattah, Topological parameters for time-space tradeoff, Artificial Intelligence 125 (2001) 93–118.

[14] R. Dechter, J. Pearl, The cycle-cutset method for improving search performance in AI applications, in: Proceedings of the Third IEEE on Artificial Intelligence Applications, 1987, pp. 224–230.

[15] R. Dechter, J. Pearl, Tree-clustering for constraint networks, Artificial Intelligence 38 (1989) 353–366.

[16] E. Freuder, A sufficient condition for backtrack-free search, J. ACM 29 (1982) 24–32.

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

[18] M. Ginsberg, Dynamic backtracking, J. Artificial Intelligence Res. 1 (1993) 25–46.

[19] G. Gottlob, N. Leone, F. Scarcello, A comparison of structural CSP decomposition methods, Artificial Intelligence 124 (2000) 282–343.

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

[21] R. Haralick, G. Elliot, Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence 14 (1980) 263–313.
[22] A. Koster, Frequency assignment—models and algorithms, PhD Thesis, University of Maastricht, November 1999.
[23] G. Kondrak, P. van Beek, A theorical evaluation of selected backtracking algorithms, Artificial Intelligence 89 (1997) 365–387.
[24] J. Larrosa, Boosting search with variable elimination, in: Proceedings of the 6th CP, 2000.
[25] A. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1977) 99–118.
[26] U. Montanari, Networks of constraints: Fundamental properties and applications to picture processing, Artificial Intelligence 7 (1974) 95–132.
[27] B. Nadel, Tree search and arc consistency in constraint-satisfaction algorithms, in: Search in Artificial Intelligence, Springer, Berlin, 1988, pp. 287–342.
[28] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Comput. Intelligence 9 (1993) 268–299.
[29] N. Robertson, P.D. Seymour, Graph minors II: Algorithmic aspects of tree-width, Algorithms 7 (1986) 309–322.
[30] D. Rose, R. Tarjan, G. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput. 5 (1976) 266–283.
[31] D. Sabin, E. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: Proceedings of 11th ECAI, 1994, pp. 125–129.
[32] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: Hard and easy problems, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, Quebec, 1995, pp. 631–637.
[33] R. Stallman, G. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, Artificial Intelligence 9 (1977) 135–196.
[34] T. Schiex, G. Verfaillie, Nogood recording for static and dynamic constraint satisfaction problems, Internat. J. Artificial Intelligence Tools 3 (2) (1994) 187–207.
[35] R. Tarjan, M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Comput. 13 (3) (1984) 566–579.
[36] G. Verfaillie, M. Lemaître, T. Schiex, Russian doll search for solving constraint optimization problems, in: Proceedings of the 13th AAAI, Portland, OR, 1996, pp. 181–187.