Decomposition and good recording for solving Max-CSPs

Jégou Philippe and **Terrioux Cyril** [1]

Abstract. [22] presents a new method called BTD for solving Valued CSPs and so Max-CSPs. This method based both on enumerative techniques and the tree-decomposition notion provides better theoretical time complexity bounds than classical enumerative methods and aims to benefit of the practical efficiency of enumerative methods thanks to the structural goods which are recorded and exploited during the search. However, [22] does not provide any experimental result and it does not discuss the way of finding an optimal solution from the optimal cost (because BTD only computes the cost of the best assignment). Providing an optimal solution is an important task for a solver, especially when we consider real-world instances. So, in this paper, we first raise these two questions. Then we explain how a solution can be efficiently computed and we provide experimental results which emphasize the practical interest of BTD.

1 INTRODUCTION

Many various problems, like boolean formulae satisfiability, configuration, graph coloring, planning, ..., can be expressed as a Constraint Satisfaction Problem (CSP). A CSP is defined by a set of variables (each one having a finite domain) and a set of constraints. Each constraint forbids some combinations of values for a subset of variables. Solving a CSP requires to assign a value to each variable such that the assignment satisfies all constraints. Determining whether a CSP has a solution is a NP-complete task. When we consider real-world problems, they involve two kinds of constraints: hard constraints which express some physical properties and soft constraints which express notions like possibility or preference. The first ones must be satisfied whereas the second ones can be violated. Unfortunately, representing these problems in the CSP formalism (where each constraint must be satisfied) often produces over-constrained problems which do not have any solution. However, even if there is no perfect solution, we can be interested by finding an assignment which optimizes a certain criterion on the constraint satisfaction. Hence, recently, many extensions of the CSP framework have been proposed (e.g. [6, 2, 21]).

In this paper, we focus our study on the Max-CSP problem [6]. Solving a Max-CSP instance requires to find an assignment which maximizes the number of satisfied constraints. Many algorithms have been defined in the past years for solving this problem. On the one hand, they exploit enumerative techniques like Branch and Bound (BB) or the arc-consistency notion [13, 10, 14, 4]. On the other hand, some other methods are based on the dynamic programming approach [23, 8, 15, 16, 17, 11]. Some of them exploit the problem structure like [8, 15, 12, 11]. These different approaches have been provided interesting results in some different cases. In [22], an hybrid method, called BTD, is presented for solving the Valued CSP problem [21] which is a generalization of the Max-CSP problem. This method is based both on enumerative techniques and on the tree-decomposition notion. It aims to benefit from the practical efficiency of enumerative methods while providing better theoretical time complexity bounds than enumerative methods. From [22], two important questions are raised. The first one is how we can compute an optimal solution from the optimal cost (because BTD only computes the cost of the best assignment, and not the assignment itself). Providing an optimal solution is one of the most important tasks for a solver, especially when we consider real-world instances. The second raised question deals with the practical efficiency of this method. BTD presents a good behaviour on classical CSPs [7]. In contrast, its behaviour on the Max-CSP problem is unknown and must be assessed. This article tries to answer these two important questions.

The paper is organized as follows. Section 2 introduces the basic notions about CSPs and Max-CSPs. Section 3 is devoted to the BTD method. Then, section 4 explains how we can compute an optimal solution. Finally, we present some empirical results in section 5, before concluding and giving some ideas of future works in section 6.

2 BASIC NOTIONS

A *constraint satisfaction problem* (CSP) is defined by a tuple (X, D, C, R). X is a set $\{x_1, \ldots, x_n\}$ of n variables. Each variable x_i takes its values in the finite domain d_{x_i} from D. Variables are subject to constraints from C. Each constraint c is defined as a set $\{x_{c_1}, \ldots, x_{c_k}\}$ of variables. A relation r_c (from R) is associated with each constraint c such that r_c represents the set of allowed tuples over $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Note that we can also define constraints by using functions or predicates for instance. Given $Y \subseteq X$ such that $Y = \{x_1, \ldots, x_k\}$, an *assignment* of variables from Y is a tuple $\mathcal{A} = (v_1, \ldots, v_k)$ from $d_{x_1} \times \cdots \times d_{x_k}$. A constraint c is said *satisfied* by $\mathcal{A}$ if $c \subseteq Y, (v_1, \ldots, v_k)[c] \in r_c$, *violated* otherwise. We note the assignment $(v_1, \ldots, v_k)$ in the more meaningful form $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$. In this paper, without lost of generality, we only consider binary constraints (i.e. constraints which involve two variables). So, the structure of a CSP can be represented by the graph (X, C), called the *constraint graph*, whose vertices are the variables of X and for which there an edge between two vertices if the corresponding variables share a constraint. Given an instance, the CSP problem consists in determining whether there is an assignment of each variable which satisfies each constraint. This problem is NP-Complete. Unfortunately, representing real-world instances as a CSP may produce over-constrained instances which do not have any solution. In such cases, as there is no perfect solution, we can be interested by finding an assignment which optimizes a certain criterion on the constraint satisfaction. Hence, in the recent years, many extensions of the CSP framework have been proposed (e.g. [6, 2, 21]).

In this paper, we focus our study on the Max-CSP problem [6]. Solving a Max-CSP instance requires to find an assignment which

[1] LSIS, Université d'Aix-Marseille III, Marseille, France. Email: {philippe.jegou,cyril.terrioux}@lsis.org

maximizes the number of satisfied constraints. In other words, we want to minimize the number of violated constraints. The number of constraints violated by an assignment is called the cost of this assignment. Many complete algorithms have been recently developed for solving Max-CSPs. They are often based on enumerative techniques or on dynamic programming approaches. Enumerative methods exploit a lower bound, which underestimates the cost of the best complete extension of the current assignment, and an upper bound which is generally the cost of the best known assignment. Then, if the lower bound does not exceed the upper one, they extend the current assignment by assigning a new variable. Otherwise, they backtrack and try to assign a new value to the last assigned variable. If all the values have been tried, they backtrack again. The efficiency of enumerative methods mostly depends on the quality of the lower and upper bounds. The greater the lower bound (respectively the smaller the upper bound) is, the less nodes are visited and constraint checks performed. The basic enumerative method is Branch and Bound (BB). It simply uses the cost of the current assignment as lower bound. Then, many improvements have been proposed from the classical CSP framework. For instance, the lower bound can be improved by using prospective techniques like Forward-Checking (FC [6]) or the arc-consistency notion [13, 10, 14, 4]. On the other hand, some other methods are based on the dynamic programming approach [23, 8, 15, 16, 17, 12, 11]. These methods divide the problem into different subproblems. Then each subproblem is solved and some informations are recorded during each resolution. These informations are exploited for solving a bigger subproblem, and so on until the whole problem is solved. In particular, they can be used for computing good lower or upper bounds like in Russian dolls search (RDS [23]) and its variants [15, 16, 12, 17]. Some of these methods exploit the problem structure like [8, 15, 12, 11]. From a practical viewpoint, the enumerative methods which use arc-consistency obtain goods results when the instances to solve have a limited size. However, they seem have some difficulties in solving larger instances like the CELAR real-world instances [3]. On the other hand, dynamic programming methods may seem perform many redundant searches or visit some useless parts of the search space. Nonetheless, in practice, they can obtain interesting results. For instance, RDS [23] and the Koster's structural method [8] succeed in solving the SCEN-06 instance of the CELAR (which is one of the hardest instances).

3 THE BTD METHOD

In [22] a new method is proposed for solving Valued CSPs [21] and so Max-CSP. This method called BTD (for Backtracking with Tree-Decomposition) is an enumerative method which is guided by a tree-decomposition of the constraint graph. A *tree-decomposition* [18] of a graph $G = (X, E)$ is a pair $(\mathcal{C}, \mathcal{T})$ with $\mathcal{T} = (I, F)$ a tree and $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ a family of subsets of X, such that each cluster $\mathcal{C}_i$ is a node of $\mathcal{T}$ and verifies: (1) $\cup_{i \in I} \mathcal{C}_i = X$, (2) for each edge $\{x, y\} \in E$, there exists $i \in I$ with $\{x, y\} \subseteq \mathcal{C}_i$, (3) for all $i, j, k \in I$, if k is on 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 G is the minimal width over all the tree-decompositions of G. Note that finding an optimal tree-decomposition is a NP-Hard problem [1]. However, we can easily compute a good tree-decomposition by using the notion of *triangulated graphs*. Figure 1(b) presents a possible tree-decomposition for the graph of figure 1(a). So, we get $\mathcal{C}_1 = \{x_1, x_2, x_3\}$, $\mathcal{C}_2 = \{x_2, x_3, x_4, x_5\}$, $\mathcal{C}_3 = \{x_4, x_5, x_6\}$ and $\mathcal{C}_4 = \{x_3, x_7, x_8\}$, and the tree-width is 3. In the following, from a tree-decomposition, we consider a rooted tree (I, F) where $\mathcal{C}_1$ is the root

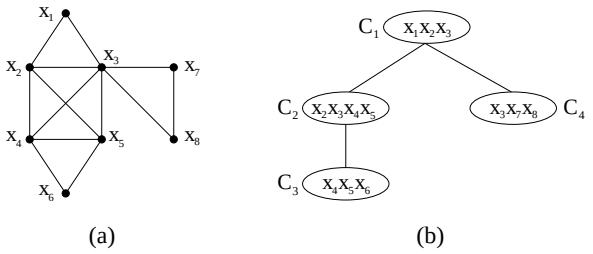

Figure 1. (a) A constraint graph on 8 variables. (b) A tree-decomposition of this constraint graph.

and we note $Desc(\mathcal{C}_j)$ the set of variables which belong to $\mathcal{C}_j$ or to any descendant $\mathcal{C}_k$ of $\mathcal{C}_j$ in the tree rooted in $\mathcal{C}_j$. For instance, $Desc(\mathcal{C}_2) = \mathcal{C}_2 \cup \mathcal{C}_3 = \{x_2, x_3, x_4, x_5, x_6\}$.

The first step of BTD consists in computing a tree-decomposition of the constraint graph. The computed tree-decomposition induces a partial variable ordering which allows BTD to exploit some structural properties of the graph and so to prune some parts of the search tree. In fact, variables are assigned according to a depth-first traversal of the rooted tree. In other words, we first assign the variables of the root cluster $\mathcal{C}_1$, then we assign the variables of $\mathcal{C}_2$, then $\mathcal{C}_3$'s ones, and so on. For example, $x_1, x_2, \ldots, x_8$ is a possible variable ordering. Furthermore, the tree-decomposition and the variable ordering allow BTD to divide the problem $\mathcal{P}$ into many subproblems. Given two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ (with $\mathcal{C}_j$ a $\mathcal{C}_i$'s son), the subproblem rooted in $\mathcal{C}_j$ depends on the current assignment $\mathcal{A}$ on $\mathcal{C}_i \cap \mathcal{C}_j$. It is denoted $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Its variable set is equal to $Desc(\mathcal{C}_j)$. The domain of each variable which belongs to $\mathcal{C}_i \cap \mathcal{C}_j$ is restricted to its value in $\mathcal{A}$. Regarding the constraint set, it contains the constraints which involve at least one variable which exclusively appears in $\mathcal{C}_j$ or in a descendant of $\mathcal{C}_j$. For instance, let us consider the CSP whose constraint graph is provided in figure 1(a). We assume that each domain is $\{1, 2, 3\}$ and each constraint $c_{ij} = \{x_i, x_j\}$ means $x_i \neq x_j$. Given $\mathcal{A} = (x_2 \leftarrow 2, x_3 \leftarrow 2)$, the variable set of $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ is $Desc(\mathcal{C}_2)$, (with $d_{x_2} = d_{x_3} = \{2\}$ and $d_{x_4} = d_{x_5} = d_{x_6} = \{1, 2, 3\}$) and its constraint set is $\{c_{24}, c_{25}, c_{34}, c_{35}, c_{45}, c_{46}, c_{56}\}$. Note that the constraint c_{23} does not belong to its constraint set because x_2 and x_3 appear both in $\mathcal{C}_1$. Remark that the definition of subproblems defines a partition of the constraint set. Such a partition ensures that BTD takes into account each constraint only once and so that it safely computes the cost of any assignment. Finally, the tree-decomposition notion permits to define the *valued structural good* notion (by analogy with the nogood notion). A structural valued good of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$ (with $\mathcal{C}_j$ a $\mathcal{C}_i$'s son) is a pair $(\mathcal{A}, v)$ with $\mathcal{A}$ the current assignment on $\mathcal{C}_i \cap \mathcal{C}_j$ and v the optimal cost of the subproblem $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. For instance, if we consider the assignment $\mathcal{A} = (x_2 \leftarrow 2, x_3 \leftarrow 2)$ on $\mathcal{C}_1 \cap \mathcal{C}_2$, we obtain the good $(\mathcal{A}, 0)$ since the best assignment on $Desc(\mathcal{C}_2)$ is $(x_2 \leftarrow 2, x_3 \leftarrow 2, x_4 \leftarrow 1, x_5 \leftarrow 3, x_6 \leftarrow 2)$ (which violates no constraint because c_{23} does not belong to $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$).

Figure 2 describes the BTD algorithm based on BB. It explores the search space according to the variable ordering induced by the tree-decomposition. So, it begins with the variables of the root cluster $\mathcal{C}_1$. Inside a cluster $\mathcal{C}_i$, it proceeds classically like BB by assigning a value to a variable, by maintaining and comparing upper and lower bounds and by backtracking if the lower bound exceeds the upper bound. The bounds in BTD are similar to BB's ones but they only take into account the constraints of the subproblem $\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}$ (with

```
BTD(A, C_i, V_{C_i}, l_{C_i}, α_{C_i})
1.  If V_{C_i} = ∅
2.  Then
3.     F ← Sons(C_i)
4.     While F ≠ ∅ and l_{C_i} < α_{C_i} Do
5.        Choose C_j in F
6.        F ← F\{C_j}
7.        If (A[C_i ∩ C_j], v) is a good of C_i/C_j in G Then l_{C_i} ← l_{C_i} + v
8.        Else
9.           v ← BTD(A, C_j, C_j\(C_j ∩ C_i), 0, α_{C_1})
10.          l_{C_i} ← l_{C_i} + v
11.          Record the good (A[C_i ∩ C_j], v) of C_i/C_j in G
12.       EndIf
13.    EndWhile
14.    Return l_{C_i}
15. Else
16.    Choose x ∈ V_{C_i}
17.    d ← d_x
18.    While d ≠ ∅ and l_{C_i} < α_{C_i} Do
19.       Choose a in d
20.       d ← d\{a}
21.       l_a ← |{c = {x, y} ∈ C | y ∉ V_{C_i} and A ∪ {x ← a} violates c}|
22.       If l_{C_i} + l_a < α_{C_i}
23.       Then α_{C_i} ← min(α_{C_i}, BTD(A ∪ {x ← a}, C_i, V_{C_i}\{x},
                                          l_{C_i} + l_a, α_{C_i}))
24.       EndIf
25.    EndWhile
26.    Return α_{C_i}
27. EndIf
```

Figure 2. The BTD algorithm.

$\mathcal{C}_{p(i)}$ the $\mathcal{C}_i$'s father and $\mathcal{A}$ the assignment on $\mathcal{C}_i \cap \mathcal{C}_{p(i)}$). The lower bound corresponds to the cost of the current assignment on $Desc(\mathcal{C}_i)$ while the upper one is defined by the cost of the best known solution for the subproblem $\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}$. When every variable in $\mathcal{C}_i$ is assigned, if the lower bound is less than the upper bound, BTD keeps on the search with the first son of $\mathcal{C}_i$ (if there is one). More generally, let us consider a son $\mathcal{C}_j$ of $\mathcal{C}_i$. Given the current assignment $\mathcal{A}$ on $\mathcal{C}_i$, BTD checks whether the assignment $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ corresponds to a valued structural good. If so, BTD adds its associated cost v to the lower bound. Otherwise it extends $\mathcal{A}$ on $Desc(\mathcal{C}_j)$ in order to compute the optimal cost v of the subproblem $\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}$. Then, it adds v to the lower bound and it records the valued good $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], v)$. If, after having proceeded the son $\mathcal{C}_j$, the lower bound does not exceed the upper bound, BTD keeps on the search with the next son of $\mathcal{C}_i$. Finally, if a failure occurs, BTD tries to modify the current assignment on $\mathcal{C}_i$.

In figure 2, given an assignment $\mathcal{A}$ and a cluster $\mathcal{C}_i$, BTD looks for the best assignment $\mathcal{B}$ on $Desc(\mathcal{C}_i)$ such that $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$ and the cost of $\mathcal{B}$ is less than $\alpha_{\mathcal{C}_i}$. $V_{\mathcal{C}_i}$ denotes the set of unassigned variables in $\mathcal{C}_i$, $l_{\mathcal{C}_i}$ the lower bound and $\alpha_{\mathcal{C}_i}$ the upper bound with respect to the subproblem $\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}],\mathcal{C}_{p(i)}/\mathcal{C}_i}$. If BTD finds such an assignment, it returns its cost, otherwise it returns a cost greater than (or equal to) $\alpha_{\mathcal{C}_i}$. The first call is BTD($\emptyset, \mathcal{C}_1, \mathcal{C}_1, 0, +\infty$).

Finally, BTD has a space complexity in $O(n.s.d^s)$ and a time complexity in $O(n.s^2.m.\log(d).d^{w+1})$ with $w+1$ the size of the largest $\mathcal{C}_k$ and s the size of the largest intersection $\mathcal{C}_i \cap \mathcal{C}_j$ with $\mathcal{C}_j$ a son of $\mathcal{C}_i$ [22]. These complexities assume that a tree-decomposition has been computed (structural parameters w and s are related to this decomposition).

4 HOW TO COMPUTE A SOLUTION?

BTD only provides the optimal cost α of the instance we want to solve. It does not compute an optimal solution of this instance. Indeed, when BTD exploits a good of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$, it does not assign again the variables of $Desc(\mathcal{C}_j) - (\mathcal{C}_i \cap \mathcal{C}_j)$. What is called in [7, 22] a *forward-jump* (by analogy with the backjump notion). For instance, after having assigned the variable x_3 in $\mathcal{C}_1$, if BTD exploits a good on $\mathcal{C}_1 \cap \mathcal{C}_2$, then, it checks for a good on $\mathcal{C}_1 \cap \mathcal{C}_4$ without exploring again $Desc(\mathcal{C}_2)$. Hence, as many variables may be unassigned, BTD cannot provide a solution of the problem we want to solve. It can only look for its optimal cost. Even if computing the optimal cost may be an important task, the main task in the Max-CSP framework is to provide an assignment which minimizes the number of violated constraints. What raises a fundamental question for BTD: how can we compute an optimal solution from the optimal cost provided by BTD? More generally, this question is often raised for algorithms like BTD which make a trade-off between time and space. As an example, the adaptation of Tree-Clustering proposed in [5] with a limited space-complexity suffers from the same drawback since it only records informations on each separator and then it cannot produce a solution in a backtrack free-manner.

In this section, we explain how we can build a solution from the optimal cost α. A basic way consists in using any enumerative algorithm for looking for an assignment with a cost α. But such a way is clearly inefficient and has a time-complexity worse than BTD's one. By so doing, we do not benefit from the tree-decomposition or from the goods which BTD has recorded during the search. So we can build a solution thanks to a method derived from BTD which would exploit the goods previously recorded. For instance, given a cluster $\mathcal{C}_i$, we can look for an assignment $\mathcal{A}$ on $\mathcal{C}_i$ such that for each son $\mathcal{C}_j$ of $\mathcal{C}_i$, $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ is a good. This method has a time complexity similar to BTD's one. However, it is clear that, in practice, it performs fewer nodes and constraint checks than BTD (except in the case where there is a single cluster). This method is better than the first, but it still seems too expensive because BTD may record a lot of goods. So for efficiency reasons, we must restrict the number of goods which are liable to be exploited for guiding the search for a solution. The ideal case would be to keep a single good per intersection $\mathcal{C}_i \cap \mathcal{C}_j$. In fact, this ideal case can be reached if we memorize some additional informations when we record or use a good.

Keeping a single good per intersection means that for each intersection, we keep the good which participates in an optimal solution. The main difficulty comes from the forward-jumps which may occur during the search. Indeed, when BTD uses a good of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$, it does not visit again the subproblem rooted in $\mathcal{C}_j$. So it does not check the goods of $\mathcal{C}_j$ with respect to any son of $\mathcal{C}_j$. For instance, by using a good on $\mathcal{C}_1 \cap \mathcal{C}_2$, BTD does not check the goods of $\mathcal{C}_2$ with respect to $\mathcal{C}_3$. Therefore, when BTD records a new good g of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$, it must also memorize, for each son $\mathcal{C}_k$ of $\mathcal{C}_j$, the good on $\mathcal{C}_j \cap \mathcal{C}_k$ which is exploited for building the current good g. By applying recursively this concept, we keep exactly one good per intersection. Thanks to a suitable data structure, these additional recordings do not change the space and time complexities of BTD.

Then, for computing a solution, we first assign the variables which appear in at least one intersection $\mathcal{C}_i \cap \mathcal{C}_j$ with the value they have in the corresponding good. We note $\mathcal{U}_{\mathcal{C}_i}$ the set of unassigned variables of $\mathcal{C}_i$. Clearly, we have $\mathcal{U}_{\mathcal{C}_i} = \mathcal{C}_i - (\mathcal{C}_{p(i)} \cup \bigcup_{\mathcal{C}_j \in Sons(\mathcal{C}_i)} \mathcal{C}_j)$. For each cluster $\mathcal{C}_i$, we consider the subproblem defined by the subgraph $(\mathcal{C}_i, C_{\mathcal{C}_i})$ of the constraint graph (X, C) with $C_{\mathcal{C}_i} = C \cap (\mathcal{C}_i \times (\mathcal{C}_i \backslash \mathcal{C}_{p(i)}))$. For each subproblem, only the variables of $\mathcal{U}_{\mathcal{C}_i}$ must be assigned. Moreover, we can solve independently each subproblem since each intersection $\mathcal{C}_i \cap \mathcal{C}_j$ is a separator of the graph (X, C). Then, for each subproblem $\mathcal{C}_i$, the initial lower bound (if we use BB) is defined by $|\{c = \{x, y\} \in C_{\mathcal{C}_i} | \exists \mathcal{C}_j \in Sons(\mathcal{C}_i), x \in$

$\mathcal{C}_i$ *and* $y \in \mathcal{C}_i \cap \mathcal{C}_j$ *and* $\mathcal{A}$ *violates* $c\}| + \sum_{\mathcal{C}_j \in Sons(\mathcal{C}_i)} \alpha_{\mathcal{C}_j}$ where $\mathcal{A} = \bigcup_{\mathcal{C}_j \in Sons(\mathcal{C}_i)} \mathcal{A}_{\mathcal{C}_j}$ and $(\mathcal{A}_{\mathcal{C}_j}, \alpha_{\mathcal{C}_j})$ is the good of $\mathcal{C}_i$ with respect to $\mathcal{C}_j$. Straightforwardly, the two terms of the previous sum involve different constraints. Regarding the upper bound, it is simply $\alpha_{\mathcal{C}_i}$. The time-complexity of this method is, in the worst case, $O(nmd^k)$ where k is the size of the largest set $\mathcal{U}_{\mathcal{C}_i}$. Of course, finding an optimal solution still requires an enumeration, but our method limits this enumeration to its strict minimum.

Finally, if there is a single cluster, obviously, we have $k = n$ and providing an optimal solution requires an enumeration on n variables. To avoid such a redundant work, we add to BTD the ability to record the best known assignment for the variables of the root cluster. This trade-off slightly changes the space-complexity ($O(n+n.s.d^s)$ instead of $O(n.s.d^s)$) while saving many redundant works.

5 EXPERIMENTAL RESULTS

The second main question raised by BTD deals with its practical efficiency. Thanks to theoretical results and some intuitive ideas, we can think that BTD is efficient on some instances (in particular if they have a good structure) but no experimental result is presented in [22]. Indeed, first, the time-complexity bound of BTD is clearly better than one of enumerative methods since $w + 1 \leq n$. Then, by recording and using goods, BTD solves each subproblem only once, which allows BTD to save time and constraint checks. In contrast, BTD uses local lower and upper bounds, what may limit its pruning capacity. Consequently, some experimentations are required in order to really assess the practical interest of BTD.

This section provides empirical results on random and real-world instances. In both cases, the experimentations are realized on a linux-based PC with an Intel Pentium IV 2.4 GHz and 512Mb of memory. For random instances, we limit to half an hour the time spent for solving a given instance. So, sometimes, some instances may be unsolved. For these instances, we consider that the running time is half an hour. For each class of random instances, we solve 50 instances. The presented results are then the averages of results obtained for each instance. For both random and real-world instances, a tree-decomposition is computed by triangulating the constraint graph (thanks to the algorithm proposed in [19]) and by searching the maximal cliques of the triangulated constraint graph. From this tree-decomposition, we produce a tree-decomposition whose parameter s does not exceed 5 for random instances and 10 for real-world ones (see [7] for more details about this computation), which limits the memory requirements of BTD. For efficiency reasons, BTD is based on FC (instead of BB), what does not change any previous theoretical results. Inside each cluster, the variable heuristic for BTD is dom/deg which first chooses the variable x_i which minimizes the ratio $|d_{x_i}|/|\Gamma_{x_i}|$ with d_{x_i} the current domain of x_i and Γ_{x_i} its neighbour set. We do not use a particular value heuristic.

5.1 Random instances

We first assess the behaviour of BTD on classical random instances. In the classical CSP framework, BTD solves classical random instances as efficiently as the best classical enumerative algorithms [7], even if these instances do not present a priori good structural properties. Unfortunately, in the Max-CSP framework, in many cases, BTD can perform worse than algorithms like FC or FC-MRDAC. Indeed, as these instances do not have good structural properties, the clusters are often under-constrained and so BTD spends a lot of time to enumerate all possible solutions (because BTD exploits local bounds).

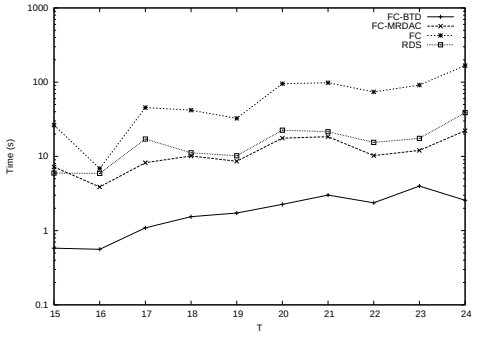

Figure 3. Mean run-time in seconds (with a log scale) for FC-BTD, FC-MRDAC, FC and RDS on class (30,5,10,T,5).

Then, we study the practical interest of BTD on structured random instances, for which one can expect that BTD provides better results than classical algorithms thanks to the exploitation of the structure. For these experiments, we use the model of structured random instances proposed in [7]. We consider several classes $(n, d, r_{max}, T, s_{max})$. An instance of the class $(n, d, r_{max}, T, s_{max})$ has n variables (each one having a domain of size d). Its constraint graph is a clique-tree such that the size of the largest clique is r_{max} and the size of the largest intersection is at most s_{max}. T denotes the tightness of each constraint. We compare BTD with FC, RDS [23] and FC-MRDAC [13]. RDS and FC use dom/deg as a variable heuristic (in a static way for RDS) and no particular value heuristic. For FC-MRDAC, we use the implementation provided by J. Larrosa [9].

Table 1. Mean run-time in seconds (respectively number of unsolved instances) for FC-BTD, FC-MRDAC, FC and RDS on random structured instances.

Classe	FC-BTD	FC-MRDAC	FC	RDS
(30,10,10,78,5)	18.9	280.6 (1)	124.2	154.9 (1)
(40,5,10,15,5)	2.7	144.6 (1)	149.3	152.8 (0)
(40,10,10,55,5)	7.6	318.5 (3)	77.6	503.1 (8)
(40,5,15,9,5)	15.5	160.3 (1)	64.2	109.8 (0)

We can first observe that, for the class of structured random instances presented in figure 3, BTD outperforms the three classical methods for any tightness. In effect, BTD solves these instances between 7 and 14 times faster than FC-MRDAC. This gain is obtained thanks to the exploitation of structural valued goods. Goods allow BTD to avoid visiting some redundant parts of the search. So, BTD achieves less constraint checks than each of the three algorithms. On the average, only a few hundred goods are produced, but each good is used up to 8,000 times. In order to confirm these results, we then compare BTD, FC-MRDAC, FC and RDS on four other classes of structured random instances (see table 1). We first note that, for some classes, FC-MRDAC or RDS are unable to solve every instance while BTD solves each of them in only a few seconds. Then we observe that BTD is significantly more efficient than FC-MRDAC, FC and RDS on these structured instances. It fully benefits from the structure. Indeed, like for the first class, only a few goods are recorded but their use allows BTD to prune a lot of branches and to achieve less constraint checks. Therefore, BTD makes a good trade-off between time and space since this recording does not require much memory.

Finally, we observe that the method proposed in section 4 for com-

puting an optimal solution requires at most a thousand additional constraint checks, which is insignificant with respect to the millions of constraint checks performed by BTD to compute the optimal cost.

5.2 Real-world instances

We experiment BTD 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, see [3]). Some of them can be easily expressed as binary Max-CSPs. We focus our study on the SUBCELAR class which contains five subproblems produced from the SCEN-06 instance (one of the hardest instances in the archive). We exploit the simplification proposed by T. Schiex [20]. It consists in removing the hard equality constraints and dividing by two the number of variables. By so doing, we obtain a smaller constraint graph and so a better tree-decomposition. For example, the smallest instance ($SUBCELAR_0$) has 16 variables and 57 constraints while the largest ($SUBCELAR_4$) has 22 variables and 131 constraints. Domains contain 36 or 44 values.

As shown in table 2, FC-BTD succeeds in solving all the SUBCELAR instances. These results are mostly due to the exploitation of structural goods. Indeed, on the average, BTD exploits each good between 9 and 261 times, which allows it to save many redundant works. For information, for these instances, computing an optimal solution from the optimal cost requires at most 3,270 constraint checks, which is insignificant with respect to the millions of constraint checks achieved for computing the optimal cost.

Comparing our results with previous ones (e.g. [13, 8, 12, 17, 11]) is quite difficult because the computer architectures are conceptually different and experimental protocols differ. For instance, [13] solves the SUBCELAR instances by using the optimal cost as initial upper bound while BTD does not exploit any initial upper bound. Nevertheless, for information, FC-MRDAC solves $SUBCELAR_2$ in about 23,000 s on a Sun Sparc 2. [11] takes also advantage of the problem structure to provide theoretical time and space complexity bounds but the experimental results are not convincing. In [17], an improved version of RDS obtains, on a Pentium IV 1.8 GHz based PC, either better results or worse ones than BTD's ones. Comparing BTD and the Koster's method [8], the best known method for solving CELAR instances, is not easy because this method exploits many pretreatments which reduce the problem size (and so the size of the constraint graph). Furthermore, we have not studied yet the influence of some structural parameters (like w or s) on the behaviour of BTD. So, by adding to BTD some pretreatments like Koster's ones or thanks to a better choice for some parameters, we can expect to improve the practical efficiency of BTD. In practice, these improvements are needed for solving larger and harder instances like SCEN-06.

Table 2. Results obtained by FC-BTD on SUBCELAR instances

Instance	Time (s)	# goods	# good uses (thousands)	# good checks (millions)
$SUBCELAR_0$	2.5	34,170	306	1.57
$SUBCELAR_1$	308	80,375	1,336	6.48
$SUBCELAR_2$	405	96,980	996	15.64
$SUBCELAR_3$	1,883	515,735	19,661	162.70
$SUBCELAR_4$	122,933	403,282	105,386	844.44

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

6 CONCLUSION AND FUTURE WORKS

In this paper, we have raised two questions about the BTD method. The first one concerns the construction of an optimal solution from the optimal cost provided by BTD. The second one deals with the practical efficiency of BTD. Then we have proposed an efficient method for computing such an optimal solution. Finally, we have shown the practical interest of BTD for solving instances with good structural properties. Indeed, BTD clearly outperforms FC-MRDAC, FC and RDS on random structured instances and it succeeds in solving all the SUBCELAR instances.

Regarding the future works, BTD can be improved by taking into account the constraints between unassigned variables (for instance by using arc-consistency [13, 4]). Then, studying the influence of some structural parameters on the behaviour of BTD can help us to optimize some choices about these parameters, which would allow BTD to obtain better results. Finally, we can add to BTD some preatreatments like Koster's ones [8].

REFERENCES

[1] S. Arnborg, D. Corneil, and A. Proskurowski, 'Complexity of finding embeddings in a k-tree', *SIAM Journal of Discrete Mathematics*, **8**, 277–284, (1987).

[2] S. Bistarelli, U. Montanari, and F. Rossi, 'Constraint solving over semirings', in *Proc. of IJCAI*, pp. 624–630, (1995).

[3] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, 'Radio Link Frequency Assignment', *Constraints*, **4**, 79–89, (1999).

[4] M. Cooper and T. Schiex, 'Arc consistency for soft constraints', *Artificial Intelligence*, **154**, 199–227, (2004).

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

[6] E. Freuder and R. Wallace, 'Partial constraint satisfaction', *Artificial Intelligence*, **58**, 21–70, (1992).

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

[8] A. Koster, *Frequency Assignment - Models and Algorithms*, Ph.D. dissertation, University of Maastricht, November 1999.

[9] J. Larrosa. http://www.lsi.upc.es/ larrosa/pfc-mrdac.

[10] J. Larrosa, 'On arc and node consistency', in *Proc. of AAAI*, pp. 48–53, (2002).

[11] J. Larrosa and R. Dechter, 'Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems', *Constraints*, **8**(3), 303–326, (2003).

[12] J. Larrosa, P. Meseguer, and M. Sánchez, 'Pseudo-Tree Search with Soft Constraints', in *Proc. of ECAI*, pp. 131–135, (2002).

[13] J. Larrosa, P. Meseguer, and T. Schiex, 'Maintaining reversible DAC for Max-CSP', *Artificial Intelligence*, **107**(1), 149–163, (1999).

[14] J. Larrosa and T. Schiex, 'In the quest of the best form of local consistency for Weighted CSP', in *Proc. of IJCAI*, pp. 239–244, (2003).

[15] P. Meseguer and M. Sánchez, 'Tree-based Russian Doll Search', in *Proc. of CP Workshop on soft constraint*, (2000).

[16] P. Meseguer and M. Sánchez, 'Specializing Russian Doll Search', in *Proc. of CP*, pp. 464–478, (2001).

[17] P. Meseguer, M. Sánchez, and G. Verfaillie, 'Opportunistic Specialization in Russian Doll Search', in *Proc. of CP*, pp. 264–279, (2002).

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

[19] D. Rose, R. Tarjan, and G. Lueker, 'Algorithmic Aspects of Vertex Elimination on Graphs', *SIAM Journal on computing*, **5**, 266–283, (1976).

[20] T. Schiex. http://www.inra.fr/bia/t/schiex/doc/celare.html.

[21] T. Schiex, H. Fargier, and G. Verfaillie, 'Valued Constraint Satisfaction Problems: hard and easy problems', in *Proc. of IJCAI*, pp. 631–637, (1995).

[22] C. Terrioux and P. Jégou, 'Bounded backtracking for the valued constraint satisfaction problems', in *Proc. of CP*, pp. 709–723, (2003).

[23] G. Verfaillie, M. Lemaître, and T. Schiex, 'Russian Doll Search for Solving Constraint Optimization Problems', in *Proc. of AAAI*, pp. 181–187, (1996).