Optimizing the space to extend the tractability of (valued) structured CSP

Karim Boutaleb, Philippe Jégou, and Cyril Terrioux

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

Abstract. It was shown that constraint satisfaction problems (CSPs) with a low width can be solved effectively by structural methods. In particular, the BTD method which exploits the concepts of goods and nogoods makes it possible to solve efficiently difficult instances. However, the memory space required for the storage of these (no)goods may make difficult or impossible the resolution of certain problems. We propose here to represent goods and nogoods with Binary Decision Diagrams (BDD). BDDs are data structures which efficiently represent informations in a compact and canonical form. Then, the practical interest of this trade-off which allows to save space memory to the detriment of time is assessed. In particular, we observe that, thanks to BDDs, BTD succeeds in solving some instances which cannot be solved previously due to the required memory space.

1 Introduction

The CSP formalism (Constraint Satisfaction Problem) offers a powerful framework for representing and solving efficiently many problems. Modeling 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, 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 the constraints. Determining if a solution exists is a NP-complete problem.

The usual method for solving CSPs is based on backtracking search. This approach, often efficient in practice, has an exponential theoretical time complexity in $O(e.d^n)$ for an instance having n variables and e constraints and whose largest domain has d values. Several works have been developed, in order to provide better theoretical complexity bounds according to particular features of the instance. The best known complexity bounds are given by the "tree-width" of a CSP (often denoted w). This parameter is related to some topological properties of the constraint graph which represents the interactions between variables via the constraints. It leads to a time complexity in $O(n.d^{w+1})$. Different methods have been proposed to reach this bound like *Tree-Clustering* [1] (see [2] for a survey and a theoretical comparison of these methods). They rely on the notion of tree-decomposition of the constraint graph. They aim to cluster variables such that the cluster arrangement is a tree. Depending on the instances, we can expect a significant gain w.r.t. enumerative approaches. Yet, the space complexity, often linear for enumerative methods, may make such an approach unusable in practice. It can be reduced to $O(n.s.d^s)$ with s the size of the largest minimal separators of the graph [3]. Several works based on this approach have been performed. Most of them only present theoretical results. Except [4, 5], no practical results have been provided. Clearly, this lack of practical results is mostly explained by the greediness of these methods in terms of memory space. Even for implemented methods, the question about the amount of required memory is raised since these methods are not able to solve any instance. For instance, the BTD method [4], whose recorded informations (namely structural goods and nogoods) are memorized in hash tables, requires sometimes more memory than available for solving some instances. A priori, this memory problem is even more raised for optimization problems which can be modeled as Valued CSPs (VCSPs [6]). In fact, solving a VCSP often requires to explore a large part of the space search (the problem is NP-Hard and is usually solved thanks to branch and bound algorithms). Hence, for a structural method like BTD [7], we may expect that a lot of goods are produced and recorded, what may make the method unusable in practice if we do not exploit a relevant data structure for storing the recorded goods.

For the both formalism, a solution to this memory problem may consist in using Binary Decision Diagram (BDD) [8]. BDDs are data structures which efficiently represent informations in a compact and canonical form. They are used in many areas, like circuit design, combinatorial logic, ... This approach was already used besides in solving VCSP with an extension of BTD [9]. However, the presented results did not make it possible to precisely evaluate the interest of this approach, in particular for the case of structured problems. In this article, we empirically study the use of BDDs for a method like BTD. We thus try to better understand their contribution. We note in particular that the profit in space is very significant. It makes it possible to solve problems by avoiding the saturation of the memory. However, the time required for the management of BDDs results in less interesting performances. That leads us to propose orientations of research to optimize the use of BDDs within this framework.

This paper is organized as follows. Section 2 provides the basic notions about (V)CSPs and methods based on the tree-decomposition notion. In section 3, we remind of the BDD framework. Section 4 explains how BDDs are exploited in the BTD method. Then, section 5 provides some experimental results to assess the practical interest of our propositions. In the last section, we conclude and discuss about relevant works.

2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined by a tuple (X, D, C). X is a set $\{x_1, \ldots, x_n\}$ of n variables. Each variable x_i takes its values in a finite domain from D (d denotes the size of the largest domain). The variables are subject to the constraints from C. Given an instance (X, D, C), the CSP problem consists in determining if there is an assignment of each variable which satisfies each constraint. This problem is NP-complete. In this paper, without loss of generality, we only consider binary constraints (i.e. constraints which involve two variables). So, the structure of a CSP can be represented by the graph (X, C), called the *constraint graph*. The vertices of this graph are the variables of X and an edge joins two vertices if the corresponding variables share a constraint.

Tree-Clustering [1] is the reference method for solving CSPs thanks to the structure of its constraint graph. It is based on the notion of tree-decomposition of graphs [10]. Let $G = (X, C)$ be a graph, a *tree-decomposition* of G is a pair $(E, \mathcal{T})$ where $\mathcal{T} = (I, F)$ is a tree with nodes I and edges F and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of $\mathcal{T}$ and verifies:

- $\cup_{i \in I} E_i = X$,
- for each edge $\{x, y\} \in E$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$,
- for all $i, j, k \in I$, if k is in a path from i to j in $\mathcal{T}$, then $E_i \cap E_j \subseteq E_k$.

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

The time complexity of Tree-Clustering is $O(n.d^{w+1})$ while its space complexity can be reduced to $O(n.s.d^s)$ with s the size of the largest minimal separators of the graph [3]. Note that Tree-Clustering does not provide interesting results in practical cases. So, an alternative approach, also based on tree-decomposition of graphs was proposed in [4]. This method is called BTD and seems to provide empirical results among the best ones obtained by structural methods.

The BTD method (for Backtracking with Tree-Decomposition) proceeds by an enumerative search guided by a 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. 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, what distinguishes BTD from other enumerative methods. More precisely, such a variable ordering is produced thanks to a depth-first traversal of the cluster tree. So, BTD begins with the variables of the root cluster E_1. Inside a cluster E_i, it proceeds classically like any backtracking algorithm by assigning a value to a variable, checking constraints and backtracking if a failure occurs. When all the variables of the cluster E_i are assigned, BTD keeps on the search with the first son of E_i (if there is one). More generally, let us consider a son E_j of E_i. Given the current assignment $\mathcal{A}$ on $E_i \cap E_j$, BTD checks whether the assignment $\mathcal{A}$ corresponds to a structural good or nogood. A *structural good* (respectively *nogood*) of E_i with respect to E_j is a consistent assignment $\mathcal{A}$ on $E_i \cap E_j$ such that there exists (respectively does not exist) a consistent extension of $\mathcal{A}$ on $Desc(E_j)$. $Desc(E_j)$ denotes the variables which belong to the descent of the cluster E_i rooted in E_j. If $\mathcal{A}$ corresponds to a good, we already know that the assignment $\mathcal{A}$ can be consistently extended on $Desc(E_j)$ and so BTD does not solve again the subproblem corresponding to $Desc(E_j)$. It keeps on the search with the next cluster according to the considered depth-first traversal of the root cluster (what is called a *forward-jump*, by analogy with backjump). In case $\mathcal{A}$ corresponds to a nogood, we already know that there exists no consistent extension of $\mathcal{A}$ on $Desc(E_j)$. Then BTD does not solve again the subproblem corresponding to $Desc(E_j)$ and a backtrack occurs. Finally, if $\mathcal{A}$ corresponds neither to a good nor to a nogood, BTD solves the subproblem rooted in E_j. If BTD succeeds in extending consistently $\mathcal{A}$ on $Desc(E_j)$, $\mathcal{A}$ is recorded as a new structural good on $E_i \cap E_j$. Otherwise, $\mathcal{A}$ is memorized as a new structural nogood. Note that a structural nogood is a particular kind of nogood justified by structural properties of the constraint network.

For optimization problems, a generalization of the BTD method has been proposed [7]. It proceeds like in the CSP case except that it relies on a branch and bound algorithm (instead of backtracking algorithm) and that it records valued goods (instead of goods and nogoods). In fact, the valued goods correspond to an extension of both goods and nogoods. A *valued structural good* of E_i with respect to E_j (with E_j a son of E_i) is a pair $(\mathcal{A}, v)$ with $\mathcal{A}$ an assignment on $E_i \cap E_j$ and v the optimal cost of the subproblem induced by $\mathcal{A}$ and rooted in E_j. Its computation is close to one of a structural nogood since finding an optimal assignment requires to explore exhaustively the corresponding search subspace. In contrast, regarding its exploitation, depending on its associated cost, it can lead to a backtrack (like nogoods) or to a forward-jump (like goods).

Thanks to the recording and the exploitations of (no)goods which allow it to prune some redundant parts of the search space, BTD offers an interesting theoretical time complexity bound in $O(n.d^{w+1})$ while classical enumerative algorithms have a time complexity in $O(e.d^n)$ ($w+1 \leq n$). Unfortunately, the space complexity, generally linear for classical enumerative algorithms, is in $O(n.s.d^s)$, what is the main drawback of structural methods like BTD. Due to the amount of required memory, few structural methods have been implemented and used successfully. The experimental results about BTD given in [4, 11] have been obtained by using an hash table for each separator. However, this solution does not allow to solve any problem. In some cases, the amount of available memory is not sufficient for solving some problems. Hence, the use of BDDs for recording goods and nogoods may allow us to reduce the amount of required memory.

3 ROBDDs for partial assignments

In the framework of BDDs, Reduced Ordered Binary Decision Diagrams (ROBDD [8, 12]) are commonly exploited. ROBDDs aim to represent boolean functions under the shape of oriented graphs without circuit. The OBDDs offer a powerful setting for solving boolean equation systems or for the treatment of various operations on boolean functions. More generally, they make it possible to represent sets in a concise way, such as for example of the sets of assignments. Their principles and mechanisms are described in details in [13, 8, 14, 12, 15] which are giving some building optimization. We recall in this sections, their principles and mechanisms of construction.

Given a boolean formula F and X its set of variables, we consider a total order $(x_1, \dots x_n)$ on X. The decision tree associated to F is a labeled path to nodes representing all interpretations of F. Internals nodes are labeled by elements of X, while leaves or *terminal nodes* are labeled by 0 or 1. These labels are noted $var(s)$ for each node s compatible with the order on X: a node of the i^{th} level in the graph is labeled $var(s) = x_i$, the root is labeled x_1. The internal nodes s possess two children corresponding to the interpretations of $var(s)$: the *left child* $lc(s)$ ($var(s)$ is interpreted to 0) and the *right child* $rc(s)$ ($var(s)$ is interpreted to 1). One calls vertices $(s, lc(s))$ and $(s, rc(s))$ respectively the left vertex and the right vertex. Thus, every *maximal path* joining the root to a leaf is equivalent to an interpretation; it is a model if the label of the leaf is 1 (*positive maximal path*) and an counter-model if the label is 0.

The OBDD representing a boolean function F corresponds to one concise expression of the decision tree of F. It is a directed graph without circuit but can possess cycles.

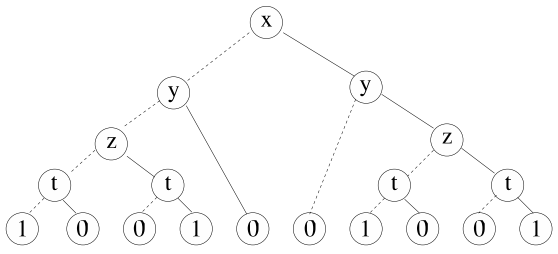

Fig. 1. Function and decision tree for the formula $(x \Leftrightarrow y) \wedge (z \Leftrightarrow t)$ according to the order (x, y, z, t) [8]. The left edges are in dotted lines, the right edges in solid lines.

The OBDD is the smallest graph which satisfies the following properties:

- it contains at most two terminal nodes: one labeled 1 and the other 0 ; if the represented function is a tautology (or a function with no model), the graph is reduced to an unique node labeled 1 (or 0).
- for any internal node s, $var(s) < var(lc(s))$ and $var(s) < var(rc(s))$, but if $var(s) = x_i$, we do not have necessarily neither $var(lc(s)) = x_{i+1}$, nor $var(rc(s)) = x_{i+1}$, nor $var(lc(s)) = var(rc(s))$.

Figure 1 gives an example of an OBDD. Every maximal path of the OBDD corresponds to a partial instantiation, restricted to variable labels of nodes of the path. If the label of the last node is 1 (or 0), all the extensions of this interpretation are models (or counter-models) of the represented function. Conversely, to any interpretation corresponds an unique maximal path in the OBDD. We will note I_c the interpretation associated to the maximal path c. The consistency check of a function F is achieved by verifying if the OBDD is reduced to the terminal node 0. To verify if an interpretation is a model, it is sufficient to browse the OBDD from the root while achieving the branchings corresponding to the interpretation. The time complexity is linear in the number of variables. A model can be obtained by searching a positive maximal path. Its complexity is $O(|B_F|)$ where $|B_F|$ is the size of the OBDD associated to F.

The size of a OBDD can be significantly reduced using other reductions in order to obtain a ROBDD, that is a Reduced OBDD. The reduction of the graph associated to a formula F relies on an elimination of redundant nodes. This reduction does not modify the satisfiability of the formula coded, but allows to reduce considerably the OBDD size compared to the decision tree. The elimination of redundancy in the graph representing the coded function is defined by this three transformations:

1. duplicated external node elimination: all external nodes labeled 0 (or 1) are merged in only one node labeled 0 (or 1).
2. duplicated internal node elimination: if two internal nodes u and v are such that $var(u) = var(v)$, $lc(u) = lc(v)$ and $rc(u) = rc(v)$, then these nodes are merged.
3. redundant internal node elimination: an internal node u verifying $lc(u) = rc(u)$ is eliminated, the retractable incident edge of u being directed towards $lc(u)$.

The graph representing a boolean function is *reduced* if it contains no internal node u such that $lc(u) = rc(u)$, and if it does not contain two distinct internal nodes u and v such that the sub-graphs rooted by u and v are isomorphic (i.e. they represent a same function). A ROBDD is a reduced graph representing a boolean function. The reduced graphs possess some properties [8]:

– For every reduced graph, for every node u of this graph, the sub-graph rooted by u is a reduced graph.
– Given a boolean function F and an order on the variables of F, there is an unique (up to isomorphism) reduced graph representing this function ; it is the ROBDD representing F. Any other graph representing F contains more nodes.

The ROBDD reduction depends on the variable ordering. The order impact on the size of the ROBDD can be significant [12]. For example, for boolean functions representing the addition of integer numbers, the size of the ROBDD can grow linear to exponential. Furthermore, there are some pathological cases, as boolean functions representing the multiplication of integer numbers, for some order, the size of the ROBDD is exponential. Figure 2 provides two examples of reduction according to two different orders for the formula considered in Figure 1.

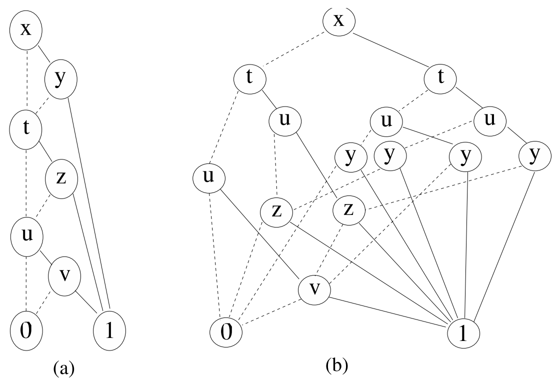

Fig. 2. Influence of the order in BDD size.

In order to build the ROBDD associated to a function F writing itself by $f < op > g$ where $< op >$ is an boolean operator, one has to compose the sub-graphs B_f and B_g associated to f and g. The time complexity is in $O(|B_f|.|B_g|)$ where $|B_f|$ and $|B_g|$ denote respectively the number of ROBDDs nodes for B_f and B_g. Especially, if F is a function having a variable x in its scope, the computation of the ROBDD coding the restriction of F to x, $F < and > x$ (case where $x = 1$) will be linear in the size of the ROBDD.

There exist several extensions of BDD. Each extension depends on its application area. We can cite, for example, FDD, ADD, BED, MTBDD, BMD, KMDD, BGD,... In our approach, we exploit the MDD [16] and ADD [17] extensions, respectively for solving CSPs and VCSPs. These two extensions represent discrete functions whose input variables are binary for ADD and multi-valued for MDD in the form of rooted, directed, ordered acyclic graphs. Each internal node corresponds to a binary variable for ADD and multi-valued variable for MDD and each leaf node represents one value of the function. Each internal node has d edges such that each edge corresponds to one of the d possible values for a variable.

4 Good and nogood represented by BDD

Solving a (V)CSP instance thanks to the BTD method often requires to record a large amount of informations (namely (no)goods). The (no)goods allow to save significant time but consume a great quantity of memory. In fact, currently, for the empirical results presented in [4, 11, 18], (no)goods are vectors of values memorized in hash tables. When we use the hash tables, we often memorized redundant informations. Indeed, in most cases, as shown in the next paragraphs, partial instantiations can appear several times in the assignments. So, we clearly see the interest to use a compact and effective structure which makes it possible to reduce the size of the recorded data. Our choice is related to two extensions of BDD to finite domains.

The adequate versions of BDD to our problems are Multi-valued Decision Diagrams (MDD [16]) for CSP and Algebraic Decision Diagrams (ADD [17]) for VCSP . They make it possible to represent canonically a set of finite domains. We exploit the package extracted from *VIS*[1]. This package has been developed at the Colorado University. It also uses the *CUDD* package[2]. We note that, in most of the applications for efficiency questions, the multi-valued variable is built with sets of ROBDDs in the internal structures. Then, each one is decomposed in a set of binary variables. For example, in figure 3, we represent $x \in \{0, 1, 2\}$ by two binary variables. More generally, we decompose x in $log_2(max_{a \in D_x}(a))$ binary variables (D_x denotes the value domain of x). In this manner, the set of the values taken by a multi-valued variable is built on a micro-structure ROBDD. Of course, there exist packages that implement directly MDDs without passing by ROBDD structure [19]. Unfortunately they suffer in general from the problem of optimization [20].

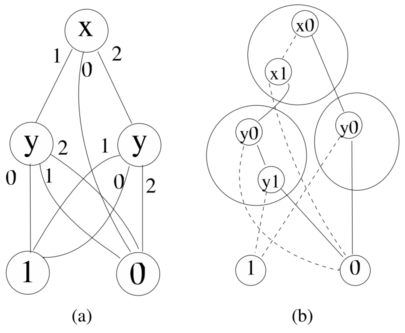

Fig. 3. Mapping from a MDD to a ROBDD [21].

In order to obtain good results, the variables are ordered according to the variable ordering induced by the tree-decomposition. This order is static. Indeed, the results with dynamic orders for all binary variables which optimize the required memory space do not show a great usefulness: the saved amount of memory space with a dynamic order does not exceed 15% with respect to a static order, while we may spend 40% of additional time.

[1] Verification Interacting with Synthesis. http://vlsi.colorado.edu/˜vis
[2] Colorado University Decision Diagrams: http://vlsi.colorado.edu/˜fabio

5 Experimental results

Before presenting the empirical results, we first describe the experimental protocol.

Experimental protocol Applying a structural method on an instance generally assumes that this instance presents some particular topological features. So, our study is first performed on instances having a structure which can be exploited by structural methods like Tree-Clustering or BTD. In practice, the two current ways of recording (no)goods are compared here on random partial structured CSPs and on real-world VCSP instances in order to point up the best one w.r.t. the (V)CSP solving runtime and the required amount of memory space. For building a random partial structured instance of a class (n, d, w, t, s, ns, p), the first step consists in producing randomly a structured CSP according to the model described in [4]. This structured instance consists of n variables having d values in their domain. Its constraint graph is a clique tree with ns cliques whose size is at most w and whose separator size does not exceed s. Each constraint forbids t tuples. Then, the second step removes randomly p% edges from the structured instance. The experimentations are performed on a Linux-based PC with a Pentium IV 2.8GHz and 512MB of memory. For each considered random class, the presented results are the average on 50 instances. We limit the runtime to 30 minutes. Above, the solver is stopped and the involved instance is considered as unsolved. In the following tables, the symbol > denotes that at least one instance cannot be solved within 30 minutes and so the mean runtime is greater than the provided value. The letter M means that at least one instance cannot be solved because it requires more than 512MB of memory. For valued CSP, we experiment on some real-world instances, namely radio-link frequency assignment problems from the FullRLFAP archive (for more details, see [22]). Of course, for these instances, the runtime is not limited.

In [18], a study was performed on triangulation algorithms to find out the best way to compute a good tree-decomposition w.r.t. CSP solving. As MCS [23] obtains the best results, we use it to compute tree-decompositions in this study.

Given a tree-decomposition, for CSP instances, we choose as root cluster the cluster which minimizes the ratio of the expected number of partial solutions of the cluster over its size. Likewise, for each cluster, its sons are ordered according this increasing ratio. For VCSP instances, we choose as root cluster the largest one and the sons are ordered according to the increasing size of the intersection with their parent cluster. Inside a cluster, for both CSP and VCSP instances, the unassigned variables are ordered thanks to the *dom/deg* heuristic. This heuristic chooses as next variable the variable x which minimizes the ratio number of the remaining values for x over the degree of x in the constraint graph.

Experimental results In this part, we compare two versions of the BTD method. These two versions differ in the way they store the (no)goods. On the one hand, the (no)goods are stored in several hash tables (one per separator). It

Table 1. Number of recorded value in hash tables, number of binary nodes in the MDDs, required memory space in MB for hash tables and MDDs, ratio hash table size in MB / MDD size in MB for Consistent and Inconsistent instances and runtime in seconds for a separator size limited to 5. For the class (250,20,20,99,10,25,0.1), one instance cannot be solved within the time limit by the version based on MDDs. For this class, the reported MDD size and the ratio correspond to the mean over the 49 solved instances.

Instances (n, d, w, t, s, ns, pr)	Memory Space						Time	
	Size				Ratio			
	Hash		MDD		C	I	Hash	MDD
	# values	MB	# nodes	MB				
(150,25,15,215,5,15,0.1)	16,168	6.75	6,795	0.11	91.6	43.9	2.30	2.69
(150,25,15,237,5,15,0.2)	22,799	7.64	7,652	0.12	106.9	48.1	1.79	2.38
(150,25,15,257,5,15,0.3)	29,448	9.46	7,412	0.18	137.9	49.6	1.01	1.80
(150,25,15,285,5,15,0.4)	5,418	13.12	3,764	0.06	259.1	177.7	0.40	0.52
(250,20,20,107,5,20,0.1)	47,836	9.11	8,558	0.14	160.8	43.7	10.39	11.70
(250,20,20,117,5,20,0.2)	59,392	10.33	9,516	0.15	200.1	40.5	8.52	10.46
(250,20,20,129,5,20,0.3)	48,135	11.63	5,408	0.09	321.9	74.5	5.82	7.91
(250,20,20,146,5,20,0.4)	90,180	14.83	8,250	0.13	252.9	73.8	3.81	6.03
(250,20,20,99,10,25,0.1)	1,554,308	25.22	100,696	1.61	27.0	12.2	58.21	>82.36
(250,25,15,211,5,25,0.1)	70,326	11.37	18,968	0.31	7.6	23.3	7.12	9.49
(250,25,15,230,5,25,0.2)	72,645	12.78	19,472	0.32	81.8	26.3	4.13	6.30
(250,25,15,253,5,25,0.3)	85,627	15.79	13,713	0.22	240.2	43.1	4.00	6.30
(250,25,15,280,5,25,0.4)	60,960	21.42	17,041	0.27	221.2	45.9	1.61	3.27

is the initial version of BTD [4, 11]. On the other hand, in the version proposed in this paper, these informations are recorded in several MDDs for CSP and ADDs for VCSP (one per separator). This comparison only focuses on the runtime and the required amount of memory space. In particular, we do not need to consider other data like the number of visited nodes or the number of performed constraint checks. Indeed, the two versions exactly obtain the same results if they exploit the same heuristics for choosing the root cluster, the next son cluster or the next variable to visit. Regarding the required amount of memory space, for the version based on hash tables, we assess it by counting the total number of recorded values. For instance, we count 3 for a good which involves 3 variables. For the version based on BDD extension, we count the total number of nodes required in binary decomposition. For instances, the MDD of figure 3 is represented by 5 binary nodes.

Tables 1 and 2 provide the results obtained on random partial structured CSPs for a limited separator size and an unlimited one. One of the main interests of the restriction of the separator size consists in limiting the amount of required memory space. Indeed, with smaller separators, the size of goods and nogoods and their potential number decrease. Without such a limitation, the BTD version based on hash tables turns sometimes to be unable to solve some instances by lack of memory space.

Table 2. Number of recorded value in hash tables, number of binary nodes in the MDDs, required memory space in MB for hash tables and MDDs, ratio hash table size in MB / MDD size in MB for Consistent and Inconsistent instances and runtime in seconds for an unlimited separator size.

Instances	Memory Space						Time	
	Size				Ratio			
(n, d, w, t, s, ns, pr)	Hash		MDD		C	I	Hash	MDD
	# values	MB	# nodes	MB				
(150,25,15,215,5,15,0.1)	188,510	16.04	90,042	1.44	16.7	8.6	2.57	8.25
(150,25,15,237,5,15,0.2)	340,683	20.58	123,594	1.98	20.1	8.1	2.70	12.65
(150,25,15,257,5,15,0.3)	252,311	23.60	86,961	1.39	35.3	10.1	1.55	8.71
(150,25,15,285,5,15,0.4)	M	-	55,573	0.89	-	-	M	3.44
(250,20,20,107,5,20,0.1)	1,898,500	31.82	317,018	5.07	15.9	4.8	18.17	43.30
(250,20,20,117,5,20,0.2)	2,614,225	41.70	274,648	4.39	17.1	6.5	13.67	37.91
(250,20,20,129,5,20,0.3)	2,731,434	46.86	363,931	5.82	16.6	9.3	12.54	61.26
(250,20,20,146,5,20,0.4)	463,786	39.54	150,649	2.41	17.1	15.8	2.37	15.64
(250,20,20,99,10,25,0.1)	M	-	960,291	15.36	-	-	M	139.00
(250,25,15,211,5,25,0.1)	317,138	26.26	202,155	3.23	21.4	5.3	6.04	18.07
(250,25,15,230,5,25,0.2)	2,235,933	45.53	356,295	5.70	28.4	9.2	24.90	33.04
(250,25,15,253,5,25,0.3)	M	-	236,044	3.77	-	-	M	30.84
(250,25,15,280,5,25,0.4)	3,173,339	64.89	452,737	7.24	30.3	6.4	12.81	81.27

Table 1 highlights the great performances of the version based on MDDs in terms of memory space. MDDs consume at least 15 times less memory space as hash tables. For some classes of instances, this rate can be greater than 50. Such a result is not surprising because, on the one hand, initially, an empty hash table requires more memory than an empty MDD, and, on the other hand, the recorded goods and nogoods often share values. According to table 1, the rate appears to be more important for consistent problems. Indeed, this gain is mostly explained by the weak number of recorded goods and nogoods for such instances. As few (no)goods are recorded, their storage in MDDs or in hash tables requires little memory space. However, only a tiny part of the memory used for the storage of the hash tables is devoted to the storage of the values of these (no)goods and the remaining part turns to be very expensive. Hence, we observe an important rate like one we could obtain by comparing an empty hash table and an empty MDD. In contrast, for inconsistent problems, the main part of the memory used for the hash tables is devoted to the storage of the (no)goods. The rate is then greater than 12. For such instances, BTD visits fully the search tree and so it produces and records more goods and nogoods. Moreover, the more goods and nogoods are recorded, the more chance of sharing values is increased. So, clearly, the representation is often more compact for inconsistent instances than for consistent ones, what is not the case when we record goods and nogoods in hash tables.

Regarding the runtime presented in Table 1, we observe an inverse behaviour but the rate is less important. The version based on MDDs is at most twice as slow than one based on hash tables due to the cost of the main operations. For hash tables, the memorization of a new good or nogood can be achieved in linear time (w.r.t. the size of the considered good or nogood) while checking if a good or a nogood is present in the hash table requires a time close to linear as soon as the goods and nogoods are fairly distributed in the hash table. For MDDs, the addition or the check can be performed in $O(a * log_2(a))$ where a is the size of the considered good or nogood. The $log_2(a)$ factor comes from the decomposition of the multi-valued variables in binary variables. Hence, a direct representation as a MDD (i.e. without a mapping to BDD) would be more interesting here. However, if, by so doing, we save a $log_2(a)$ factor for the runtime, we consume more memory with the same factor. We note that the two versions solve all the instances, except one instance of the class (250,20,20,99,10,25,0.1) for the version based on MDDs. This instance cannot be solved within the time limit.

Given the promising results obtained thanks to MDDs in terms of required memory space, we assess the behaviour of the two versions for an unlimited separator size. By so doing, the size and the number of goods and nogoods increase and so we can expect a greater benefit from MDDs. Table 2 presents the observed results. Like previously, the version based on MDDs outperforms one based on hash tables w.r.t. the required memory space while it spends more time for solving the instances. This additional time corresponds again to the cost of managing goods and nogoods in the MDDs. Nonetheless, unlike for a limited separator size, the version based on hash tables does not succeed in solving all the instances. The amount of required memory space prevents from solving several instances. Note that the compactness of the MDD representation allows to solve these same instances.

Finally, we assess the interest of this approach for VCSPs on some real-world instances. We observe similar trends to ones obtained for CSPs. The exploitation of ADDs allows us to save a great amount of memory space. Indeed, the version based on ADDs consumes at least 4 times less memory space. Regarding the runtime, like previously, the version based on hash tables is faster. However, we can note that, except for the SUBCELAR0 instance, the difference of runtime between the two compared version is significantly reduced (only about 5%).

To sum up, the compactness of recorded informations allows to reduce the amount of required memory space but it requires some additional runtime. Hence, unlike the results about the memory space, the runtime obtained by using MDDs is not competitive enough with respect to one of the initial version of BTD based on hash tables. As explained above, this additional cost results from the construction and the management of MDDs mapped to BDDs. For VCSP, the version based on ADDs presents sometimes competitive runtime while reducing significantly the required space memory. However, in spite of the non-competitive runtimes, the BTD version based on MDDs/ADDs remains interesting. Indeed, it is often possible to spend more time for solving an instance whereas we cannot consume more memory than available and, unfortunately, we cannot foresee the amount of needed memory space.

Table 3. Number of recorded value in hash tables, number of binary nodes in the ADDs, required memory space in MB for hash tables and ADDs and runtime in seconds for an unlimited separator size.

Instances	Memory Space				Time	
	Hash		ADD		Hash	ADD
(n, d, w, t, s, ns, pr)	#values	MB	#nodes	MB		
SUBCELAR0	176,741	2.35	12,680	0.20	374.7	735.0
SUBCELAR1	482,694	3.10	32,960	0.53	827.0	844.8
SUBCELAR2	1,105,558	6.80	93,431	1.49	766.9	803.3
SUBCELAR3	2,696,358	15.81	241,126	3.86	7,493.9	7,889.5
SUBCELAR4	3,090,263	18.65	264,480	4.23	12,923.1	13,185.0

6 Conclusion and discussion

In this article, we have studied the resolution of structured CSPs and VCSPs. In particular, we have been interested in the BTD method [4, 7] whose efficiency results from the exploitation of structural (no)goods learnt and recorded during the search. Whereas, in its initial version, BTD represented (no)goods in extension with hash tables, we have studied here from a practical viewpoint the interest which can present a memorization of these informations in a compact structure like BDDs (MDDs and ADDs precisely).

A similar work has already been performed by [9] with an extension of BTD for solving VCSPs. However, this work does not make it possible to determine the real interest of the use of the ADDs, in particular for the case of structured (V)CSPs. Here, we present a study which aims to better assess this interest with respect to the saved amount of memory, but also the runtime.

Concerning the structured CSP, we have observed a very significant profit in terms of required memory space. Indeed, several problems which could not be solved by BTD with the hash tables are now manageable. More generally, one observes a systematic profit for space on all the problems. Regarding the runtime, we have observed a degradation of the efficiency. Indeed, the time devoted to the management of BDDs, in particular for the addition of (no)goods, slows down significantly the effectiveness of the approach. This report leads us to continue this work while trying to better manage space. In particular, we should propose an approach which would improve significantly the runtime. For VCSP, we have observed a slightly different trend. Indeed, the BTD version based on ADDs has sometimes presented competitive runtime on real-world instances while reducing significantly the required space memory.

On the level of the other prospects to this work, we will keep on evaluating this approach on real problems for both CSPs and VCSPs. Nonetheless, it still seems more interesting to us and more promising to focus our study on optimization problems like Valued CSP rather than decision ones. That is possible by exploiting ADDs [17] like proposed in [9] or used in this study. However, such an extension could also pass by the design of a new kind of BDDs better adapted to the resolution of optimization problems.

Acknowledgments This work is supported by a ”programme blanc” ANR grant (STAL-DEC-OPT project).

References

1. R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
2. G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124:343–282, 2000.
3. R. Dechter and Y. El Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125:93–118, 2001.
4. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
5. G. Gottlob, M. Hutle, and F. Wotawa. Combining hypertree, bicomp and hinge decomposition. In *Proc. European Conference on Artificial Intelligence*, pages 161–165, 2002.
6. T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 631–637, 1995.
7. C. Terrioux and P. Jgou. Bounded backtracking for the valued constraint satisfaction problems. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP-2003)*, pages 709–723, 2003.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
9. M. Sachenbacher and B. C. Williams. Bounded Search and Symbolic Inference for Constraint Optimization. In *Proceedings of IJCAI*, pages 286–291, 2005.
10. N. Robertson and P.D. Seymour. Graph minors II: Algorithmic aspects of tree-width. *Algorithms*, 7:309–322, 1986.
11. P. Jégou and C. Terrioux. Decomposition and good recording for solving Max-CSPs. In *Proceedings of ECAI*, pages 196–200, 2004.
12. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
13. Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.
14. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 40–45, 1990.
15. J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 205–210. IEEE Computer Society Press, 1988.
16. Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *ICCAD*, pages 92–95, 1990.
17. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 188–191, 1993.
18. P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proc. of the 11th International*

Conference on Principles and Practice of Constraint Programming, pages 777–781, 2005.

19. D. Miller and R. Drechsler. Implementing a multiple-valued decision diagram package. In *ISMVL '98: Proceedings of the The 28th International Symposium on Multiple-Valued Logic*, page 52, Washington, DC, USA, 1998. IEEE Computer Society.
20. Frank Schmiedle, Wolfgang Günther, and Rolf Drechsler. Dynamic re-encoding during mdd minimization. In *ISMVL*, pages 239–244, 2000.
21. T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4(1-2):9–62, 1998.
22. C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4:79–89, 1999.
23. R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13 (3):566–579, 1984.