(No)good Recording and ROBDDs for Solving Structured (V)CSPs

Karim Boutaleb Philippe Jégou
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 efficiently by structural methods. However, these methods often present an important drawback: they generally require a large amount of memory space, what makes their use difficult or impossible. For instance, the BTD method solves efficiently difficult instances thanks to the recording of goods and nogoods. As this recording may require an exponential memory size, the exploitation of a compact data structure is crucial. In this paper, we propose to store (no)goods in Binary Decision Diagrams (BDD). BDDs are data structures which efficiently represent informations in a compact and canonical form. Finally, we assess the practical interest of this tradeoff which allows to save space memory and consequently to solve problems that cannot be solved without BDDs.

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 $x_1, x_2, \ldots x_n$, 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.

CSPs are usually solved by 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 $O(n.d^{w+1})$. This bound is reached by several structural methods like *Tree-Clustering* [9] (see [11] for a survey and a theoretical comparison of these methods). These methods 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 [8]. Such a space complexity is an important drawback which surely explains why most of performed works remains theoretical. Indeed, except [13, 10], no practical results have been provided. Even for implemented methods, the amount of required memory may be problematic. For instance, the BTD method (for Backtracking with Tree-Decomposition [13]), which records structural goods and nogoods in hash tables, requires sometimes more memory than available for solving some instances. A priori, this memory problem is even more delicate for optimization problems modeled as Valued CSPs (VCSPs [19]). In fact, solving a VCSP often requires to explore a larger part of the search space (the problem is NP-Hard and is usually solved thanks to branch and bound algorithms). Hence, a structural method like BTD [23] may produce and record a lot of goods, what may make the method unusable in practice if we do not exploit a relevant data structure for storing the recorded goods.

For both CSP and VCSP formalisms, the use of a compact data structure for storing (no)goods is crucial. A solution may consist in using Binary Decision Diagram (BDD)

[4]. BDDs are data structures which efficiently represent informations in a compact and canonical form. They are used in many areas, (e.g. circuit design or combinatorial logic). This approach was already used previously in solving VCSP with an extension of BTD [18]. However, the presented results did not provide any information about the interest of this approach, in particular for 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 a very significant profit in space which 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 runtime. 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 and we explain how BDDs are exploited in BTD. Then, section 4 provides some empirical results to assess the practical interest of our propositions. In section 5, we conclude and discuss about related works.

2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined by a tuple (X, D, C). X is a set $\{x_1, \dots, x_n\}$ 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 [9] is the basic method for solving CSPs thanks to the structure of its constraint graph. It is based on the notion of tree-decomposition of graphs [17]. 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 corresponds to a node of $\mathcal{T}$ and verifies: (1) $\cup_{i \in I} E_i = X$, (2) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$ and (3) 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 (i.e. intersection between clusters) of the graph [8]. Note that Tree-Clustering does not provide interesting results in practical cases. So, an alternative approach, called BTD and also based on tree-decomposition of graphs, was proposed in [13]. It seems to provide empirical results among the best ones obtained by structural methods.

The BTD method 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 from which a partial variable ordering is induced. This ordering 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 the *separator* $E_i \cap E_j$ such that there exists (resp. 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 [23]. 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 w.r.t. 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).

Recording and exploiting (no)goods allow BTD to prune some redundant parts of the search space and so to offer 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 [13, 14] have been obtained by using an hash table for each separator. However, this solution does not allow to solve any problem since, in some cases, the amount of available memory is not sufficient. Hence, the storage in goods and nogoods in compact data structure like BDDs may reduce the amount of required memory.

3 Goods and nogoods stored in BDDs

3.1 ROBDDs for partial assignments

In the framework of BDDs, Reduced Ordered Binary Decision Diagrams (ROBDD [4, 5]) are commonly exploited. ROBDDs aim to represent boolean functions under the shape of oriented graphs without circuit. ROBDDs 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 the sets of assignments. We recall here their principles and mechanisms of construction (see [1, 4, 3, 5, 15] for more details).

Given a boolean formula F and its set X 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. Internal 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).

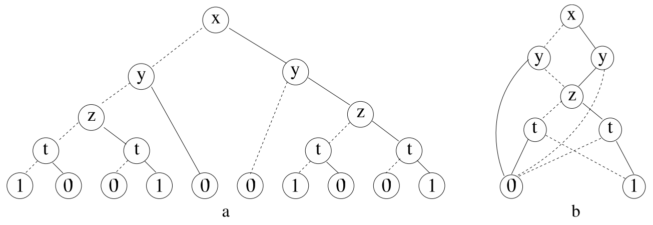

Figure 1. Associated OBDD (a) to the function defined by the formula $(x \Leftrightarrow y) \wedge (z \Leftrightarrow t)$ according to the order (x, y, z, t) [4]. The left edges are in dotted lines, the right edges in solid lines. Corresponding ROBDD (b).

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 a counter-model if the label is 0.

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

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 a 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 (a) 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 a 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 branching 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 Reduced OBDD

(ROBDD). 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 encoded formula, but allows to reduce considerably the OBDD size compared to the decision tree. The elimination of redundancy in the graph representing the function is defined by these three transformations:

1. duplicated terminal node elimination: all terminal 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. Figure 1 (b) gives an example of ROBDD. The reduced graphs possess some properties [4]:

- 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 a 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 [5]. 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.

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 encoding the restriction of F to x, $F < and > x$ (case where $x = 1$) will be linear in the size of the ROBDD.

3.2 (Valued) (No)Goods stored in BDDs

There exist several extensions of BDD (e.g. FDD, ADD, BED, MTBDD, BMD, KMDD or BGD), each one depending on its application area. In our approach, we exploit the Multi-valued Decision Diagram (MDD [21]) and Algebraic Decision Diagram (ADD [2]) 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.

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 significantly time but consume a great quantity of memory. In fact, currently, for the empirical results presented in [13, 14, 12], (no)goods are vectors of values stored in hash tables. Precisely, we use an hash table per separator for storing goods and another one for nogoods. When we use hash tables, we often memorize redundant informations. Indeed, in most cases, several assignments may contain a same subassignment. So, we clearly see the interest to use a compact and efficient 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, namely MDD [21] for CSP and ADD [2] for VCSP. Each hash table is replaced by a MDD or a ADD.

Now, if we consider a VCSP, to memorize a valued good in an ADD, it is enough to represent the value of the corresponding assignment in the terminal node labeled by that value.

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 2, we represent $x \in \{0, 1, 2\}$ by two binary variables. More generally, we decompose x in $log_2(|D_x|)$ binary variables (D_x indicates the field of values of x). In this manner, the set of the values taken by a multi-valued variable is built on a ROBDD. Of course, some packages implement directly MDDs without passing by ROBDD structure [16], but, unfortunately, they generally suffer from the problem of optimization [20].

In order to obtain good results, the variables are or-

[1]Verification Interacting with Synthesis. http://vlsi.colorado.edu/˜vis

[2]Colorado University Decision Diagrams: http://vlsi.colorado.edu/˜fabio

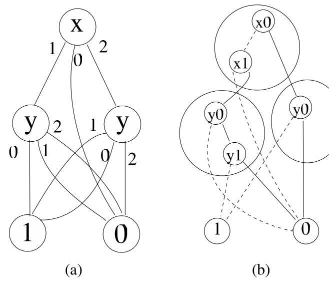

Figure 2. Mapping from a MDD to a ROBDD.

dered 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: we have observed that the saved amount of memory space with a dynamic order does not exceed 15% w.r.t. a static order, while we may spend 40% of additional time.

Finally, the check of a (no)good associated to s variables can be performed in $O(s * log_2(d))$. The $log_2(d)$ factor comes from the decomposition of the multi-valued variables in binary variables. The most important problem here is related to the addition of a new (no)good. Indeed, the time complexity is then O($|MDD|.|NG|$) where $|MDD|$ and $|NG|$ denote respectively the size of the current MDD and the size of the (no)good, precisely $|NG| = s * log_2(d)$.

4 Experimental results

4.1 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, n_c, p), the first step consists in producing randomly a structured CSP according to the model described in [13]. This structured instance consists of n variables having d values in their domain. Its constraint graph is a clique tree with n_c 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 [6]. Of course, for these instances, the runtime is not limited.

We use MCS [22] to compute a tree-decomposition since [12] has pointed out that MCS computes relevant tree-decompositions w.r.t. CSP solving. 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.

4.2 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 is the initial version of BTD [13, 14]. On the other hand, in the version proposed here, (no)goods are recorded in MDDs for CSP and ADDs for VCSP. 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 memory space, we assess it by counting the required memory in MB, and by reporting the number of recorded values in hash tables, and the number of binary nodes in the MDDs.

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 (no)goods and their potential number decrease. Without such a limitation, the BTD version based on hash tables

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, n_c, 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
(250,20,20,146,5,20,0.4)	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
(250,25,15,230,5,25,0.2)	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

turns out sometimes to be unable to solve some instances by lack of memory space.

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 the (no)goods 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 (no)goods 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 (array of pointers) turns out 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 (no)goods. Moreover, the more (no)goods are recorded, the more chance of sharing values is increased. So, clearly, the representation as MDDs is often more compact for inconsistent instances than for consistent ones, what is not the case by using 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 (no)good can be achieved in linear time (w.r.t. the size of the considered (no)good) while checking if a (no)good is present in the hash table requires a time close to linear (w.r.t. the size of the considered (no)good) as soon as the (no)goods are fairly distributed in the hash table. For MDDs, we have seen in section 3 that the addition of a new (no)good is more expensive since we have multiplicative factor related to the size of the MDD.

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 $log_2(d)$ 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 (no)goods 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 (no)goods in the MDDs. Nonetheless, unlike for a limited separator size, the version based on hash tables does not succeed in solving all

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 (n, d, w, t, s, n_c, pr)	Memory Space						Time	
	Size				Ratio			
	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
(250,20,20,146,5,20,0.4)	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
(250,25,15,230,5,25,0.2)	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

the instances due to the expensive amount of required memory space while the compactness of the MDD representation allows to solve them.

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 versions 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 w.r.t. 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 runtime, 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.

5 Conclusion and discussion

In this article, we have studied the solving of structured (V)CSPs. In particular, we have been interested in the BTD method [13, 23] 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 [18] has already been performed with an extension of BTD for solving VCSPs. However, the presented results did not provide any information about the interest of the use of ADD, in particular for structured problems. Here, our study aims to better assess this interest w.r.t. the saved amount of memory, but also the runtime.

Regarding the structured CSPs, 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 in-

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
	#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

stances while reducing significantly the required space.

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 [2] like proposed in [18] 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 solving of optimization problems, or then, by the adaptation of approaches even more effective than BDDs such as for example d-DNNF [7].

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

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[2] 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 *Proc. of ICCAD*, pages 188–191, 1993.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of DAC*, pages 40–45, 1990.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[5] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[6] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4:79–89, 1999.

[7] A. Darwiche. New advances in compiling cnf to decomposable negational normal form. In *Proc. of ECAI*, pages 328–332, 2004.

[8] R. Dechter and Y. E. Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125:93–118, 2001.

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

[10] G. Gottlob, M. Hutle, and F. Wotawa. Combining hypertree, bicomp and hinge decomposition. In *Proc. of ECAI*, pages 161–165, 2002.

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

[12] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proc. of CP*, pages 777–781, 2005.

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

[14] P. Jégou and C. Terrioux. Decomposition and good recording for solving Max-CSPs. In *Proc. of ECAI*, pages 196–200, 2004.

[15] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proc. of DAC*, pages 205–210, 1988.

[16] D. Miller and R. Drechsler. Implementing a multiple-valued decision diagram package. In *Proc. of ISMVL*, page 52, 1998.

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

[18] M. Sachenbacher and B. C. Williams. Bounded Search and Symbolic Inference for Constraint Optimization. In *Proc. of IJCAI*, pages 286–291, 2005.

[19] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: hard and easy problems. In *Proc. of IJCAI*, pages 631–637, 1995.

[20] F. Schmiedle, W. Günther, and R. Drechsler. Dynamic reencoding during mdd minimization. In *Proc. of ISMVL*, pages 239–244, 2000.

[21] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Proc. of ICCAD*, pages 92–95, 1990.

[22] 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.

[23] C. Terrioux and P. Jégou. Bounded backtracking for the valued constraint satisfaction problems. In *Proc. of CP*, pages 709–723, 2003.