Solving Intensional Weighted CSPs by Incremental Optimization with BDDs

Miquel Bofill*, Miquel Palahí*†, Josep Suy*, and Mateu Villaret*

Departament d'Informàtica, Matemàtica Aplicada i Estadística
Universitat de Girona, Spain
{mbofill,mpalahi,suy,villaret}@imae.udg.edu

Abstract. We present a method for solving weighted Constraint Satisfaction Problems, based on translation into a Constraint Optimization Problem and iterative calls to an SMT solver, with successively tighter bounds of the objective function. The novelty of the method herewith described lies in representing the bound constraint as a shared Binary Decision Diagram, which in turn is translated into SAT. This offers two benefits: first, BDDs built for previous bounds can be used to build the BDDs for new (tighter) bounds, considerably reducing the BDD construction time; second, as a by-product, many clauses asserted to the solver in previous iterations can be reused.
The reported experimentation on the `WSimply` system shows that this technique has better performance in general than other methods implemented in the system. Moreover, with the new technique `WSimply` outperforms some state-of-the-art solvers in most of the studied instances.

1 Introduction

A Constraint Satisfaction Problem (CSP) is a decision problem where the goal is to determine whether an assignment of values to a set of variables exists which satisfies a given set of constraints. It is common to find CSPs where, additionally to determine if there exists a solution for the problem, the possible solution has to minimize or maximize some objective function. These kinds of CSP are known as Constraint Optimization Problems (COP).

Occasionally, some real-world CSP instances have no solution. In such situations, we can relax the CSP by allowing the violation of a subset of the constraints, and try to maximize the number of satisfied constraints. This CSP variant is known as Maximum CSP (MaxCSP) [18]. Furthermore, there can exist preferences over which constraints to violate. A convenient way of expressing these preferences is by giving a weight to each constraint, denoting its violation cost. The constraints that can be violated (the ones with a non-infinite weight) are usually called *soft*, while those constraints that must be satisfied are called

*Supported by the Spanish Ministry of Science and Innovation (project TIN2012-33042).

†Supported by UdG grant (BR 2010).

hard. Then, the objective is to find an assignment which satisfies all hard constraints and minimizes the aggregated cost of the violated soft constraints [23]. These problems are known as Weighted CSP (WCSP) [21] or, alternatively, as Cost Function Networks (CFN) [13].

`WSimply` [3,5] is a pioneering language and system for solving intensionally represented WCSPs by reformulation into Satisfiability Modulo Theories (SMT) [7], namely, into SAT modulo Linear Integer Arithmetic (LIA). An SMT formula can be seen as a generalization of propositional Boolean formula, where some predicates have predefined interpretations from background theories, and any satisfying assignment has to be compatible with those theories. Leveraging the advances made in SAT solvers in the last decade, SMT solvers have proved to be competitive with classical decision methods in many areas, and in particular in CSP solving [4,9]. Most modern SMT solvers integrate a SAT solver with decision procedures (theory solvers) for sets of literals belonging to each theory. For example, variations of the simplex method are used for dealing with LIA predicates. This way, one can hopefully get the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms for the theory reasoning. As shown in [9], SMT outperforms other methods on instances with a significant Boolean component, i.e., instances with Boolean decision variables and disjunctions of (arithmetic) constraints.

`WSimply` benefits from the expressiveness of the SMT language and the performance of current SMT solvers. However, SMT solvers are decision procedures, and they rarely support optimization. A few solvers support (weighted partial) MaxSMT [16,14,12], and there is a recent attempt to introduce optimization into SMT by means of a theory of costs [11]. In `WSimply`, optimization is implemented by means of successive calls to the decision procedure in several (user choosable) ways: performing sequential or binary search, or using algorithms based on unsatisfiable cores like WPM1 [6].

In this paper we extend the WCSP solving capability of `WSimply` by introducing a new optimization method, based on representing the bound constraint on the objective function (generated from the violation cost of soft constraints) as a BDD [2]. This allows us to encode the objective as a pure propositional formula, following the generalized arc-consistent encoding proposed in [1]. This way, on the one hand we increase the Boolean component of the instances and, on the other hand, we tighten the link between optimization and the logical structure of the problem, with the hope of benefiting from crucial capabilities of the underlying solver, such as conflict driven learning [22]. Since the BDD for the objective can be really big, and so the number of clauses to represent it, an interesting aspect of our approach is the reutilization of BDDs in successive calls to the decision procedure. Although changing the bound of the objective function implies building a new BDD, some parts can be easily reused. We create a Shared BDD [20], also known in the literature as Multi-Rooted BDD, keeping all the generated BDDs. This allows not only to improve the performance of the

BDD construction algorithm, but also to keep a number of learned clauses from the solver, as we reuse the clauses representing the previous BDDs.

Since the size of BDDs strongly depends on the number of variables involved, and we use them to represent bound constraints on objective functions encoding the violation cost of soft constraints, our method is especially well suited for WCSP instances involving a *small* number of soft constraints. We provide an experimentation section where we show that the new BDD-based solving method outperforms previous `WSimply` solving methods in the majority of the problems. Moreover, we also provide comparisons with other state-of-the-art optimization and WCSP solvers, showing that our method is, in general, the most robust one.

The paper is structured as follows. In Section 2 we introduce the required background. In Section 3 we introduce Pseudo-Boolean constraints and (Reduced Ordered) Binary Decision Diagrams (ROBDD). In Section 4 we present our incremental method of solving WCSPs using shared ROBDDs. In Section 5 we study the performance of the new method and we compare it with other methods implemented in `WSimply` and other state-of-the-art systems. In the same section, we report on some experiments showing how the use of BDDs to represent the objective, instead of using a linear equation, has a positive impact in the learning of the SMT solver. Finally, in Section 6 we conclude and propose some future work.

2 Preliminaries

In this section we recall some basic definitions related to our research. First, we introduce the kind of WCSPs we are interested to solve. Second, we briefly explain what is SMT. Finally, we briefly review how the `WSimply` system faces the optimization process by iterative calls to an SMT solver.

2.1 WCSPs and COPs

A *Constraint Satisfaction Problem (CSP)* instance is defined as a triple $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{d(x_1), \ldots, d(x_n)\}$ is a set of domains containing the values the variables may take, and $C = \{C_1, \ldots, C_m\}$ is a set of constraints defining relations between subsets of variables. In this paper we assume to deal with intensional Weighted CSPs, where weighted constraints have the form $(c, w(c))$, being c a constraint as defined for a CSP and $w(c)$ the cost corresponding to its falsification, which can be a natural number or infinity. We call those constraints whose associated cost is infinity *hard*, if otherwise *soft*. A solution to a WCSP is an assignment that satisfies all hard constraints and minimizes the sum of falsified soft-constraints costs.

A *Constraint Optimization Problem (COP)* instance consists of an optimization variable O, matched to an objective function to be minimized (maximized) subject to the constraints of a CSP instance $\langle X, D, C\rangle$, where $O \in X$. A solution to a COP instance is a solution to the CSP instance that minimizes (maximizes) the value of the optimization variable O. A WCSP can be seen as a COP where

the objective function to minimize is $\sum_{i=1}^{m} w_i * o_i$, being w_i the cost of the soft-constraint i of the WCSP and o_i a pseudo-Boolean variable representing if the soft-constraint i is violated.

2.2 SMT and weighted SMT

A *Satisfiability Modulo Theories (SMT)* formula is a generalization of a Boolean formula in which some propositional variables have been replaced by predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. For example, a formula can contain clauses like $p \vee q \vee (x + 2 \leq y) \vee (x > y + z)$, where p and q are Boolean variables and x, y and z are integer variables. A *solution* to an SMT instance is an assignment that satisfies the formula, provided that predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory [7]. As in the CSP case, we can extend SMT to *weighted SMT (WSMT)* as follows.

A *weighted SMT clause* is a pair (C, w), where C is an SMT clause[1] and w is a natural number or infinity (indicating the penalty for violating C). A *weighted SMT formula* is a multiset of weighted SMT clauses

$$\{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$$

where the first m clauses are soft and the last m' clauses are hard. The optimal cost of a weighted SMT formula is the minimal cost of all its assignments. An optimal assignment is an assignment with optimal cost. The *WSMT problem*[2] for a WSMT formula is the problem of finding an optimal assignment for that formula.

2.3 Solving WCSP with (weighted) SMT

`WSimply` [5] is a tool with its own language for WCSP and COP modelling that solves the corresponding instances translating them into (weighted) SMT. First of all, the hard-constraints of the instance are translated into an SMT formula. Second, depending on the user-specified way to solve the instance, the soft-constraints are reformulated either into an objective function or into weighted SMT formulas.[3] Finally, the generated (weighted) SMT formula is solved using one of three solving methods: `yices`, `core` or `dico`, where the two former are used to solve WCSP instances, while the latter is used to solve COP instances. In order to give the reader some idea of these methods we add a short description of them:

[1] In fact these can be general SMT formulas, not necessarily disjunctions of literals.

[2] In the literature the weighted SMT problem is also referred to as weighted MaxSMT, same as in the SAT formalism. We prefer to talk about WSMT because it is closer to WCSP.

[3] Note that in case it is wanted to solve a COP instance using WSMT, the objective function can be easily translated into WSMT clauses.

– The `yices` method uses an algorithm that performs a sequence of satisfiability checks until the optimum is found. It is the default Yices [16] algorithm for solving WSMT (`WSimply` is built on top of Yices). This algorithm is not exact since Yices defines a maximum number of iterations for the search.[4] We are not aware of any document describing the procedures used there.
– The `core` method is an implementation, introduced in [5], of the core based WPM1 algorithm [6] for MaxSAT.
– In the `dico` method, the system first translates the constraints of the COP into SMT formulae, and then iteratively calls the SMT solver, bounding the optimization variable O by adding the unit clause $O \leq K$, where K is an integer constant determined by the system using binary search.
It is worth noting that in the case we are bounding the objective function of a COP encoding a WCSP instance, this results in a pseudo-Boolean constraint (see Subsection 2.1).

For deeper details about `WSimply` and its solving techniques we refer the reader to [5].

3 Binary Decision Diagrams

A typical data structure to represent Boolean functions is a *Binary Decision Diagram (BDD)*, which consists of a rooted, directed, acyclic graph, where each non-terminal (decision) node corresponds to a Boolean variable x and has two child nodes with edges representing a *true* and a *false* assignment to x, respectively. We talk about the *true child* (resp. *false child*) to refer to the child node linked by the *true* (resp. *false*) edge. Terminal nodes are called 0-terminal and 1-terminal, representing the truth value of the formula for the assignment leading to them. A BDD is called *ordered* if different variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if the following two rules have been applied to its graph until fixpoint:

– Merge any isomorphic subgraphs.
– Eliminate any node whose two children are isomorphic.

A *Reduced Ordered Binary Decision Diagram (ROBDD)* is canonical (unique) for a particular function and variable order. Figure 1 shows an example of a ROBDD.

An interesting property of ordered BDDs is that when multiple ordered BDDs contain isomorphic subgraphs, they can be joined in a single *shared BDD (SBDD)* [20], providing a more compact representation of the Boolean functions.

3.1 SAT encodings of pseudo-Boolean constraints using BDDs

Pseudo-Boolean (PB) constraints [10] are constraints of the form $a_1x_1 + \cdots + a_nx_n \ \# \ K$, where the a_i and K are integer coefficients, the x_i are pseudo-Boolean (0/1) variables, and the relation operator # belongs to $\{<, >, \leq, \geq, =\}$.

[4] `http://yices.csl.sri.com/language.shtml`

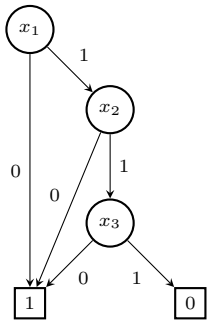

Fig. 1. ROBDD for the Boolean function $\overline{x_1} + x_1 \cdot \overline{x_2} + x_1 \cdot x_2 \cdot \overline{x_3}$.

For our purposes we will assume that # is $\leq$ and that the a_i and K are positive. Under these assumptions, these constraints are monotonic (decreasing) Boolean functions $C : \{0,1\}^n \rightarrow \{0,1\}$, i.e., any solution for C remains a solution after flipping input values from 1 to 0.

It is quite common to use BDDs to represent PB-constraints. For example, the ROBDD of Figure 1 also corresponds to the PB constraint $2x_1 + 3x_2 + 4x_3 \leq 7$.

An interval of a PB constraint C is the set of values of K for which C has identical solutions. More formally, given a constraint C of the form $a_1x_1 + \cdots + a_nx_n \ \leq \ K$, the *interval of* C is the set of all integers M such that the constraint $a_1x_1 + \cdots + a_nx_n \ \leq \ M$, seen as a Boolean function, is equivalent to C (i.e., that the corresponding Boolean functions have the same truth table). For instance, the interval of $2x_1 + 3x_2 + 4x_3 \leq 7$ is $[7, 8]$ since, as no combination of coefficients adds to 8, we have that the constraint $2x_1 + 3x_2 + 4x_3 \leq 7$ is equivalent to $2x_1 + 3x_2 + 4x_3 \leq 8$.

There exist several BDD-based approaches for reformulating PB constraints into propositional clauses [17]. We focus on the recent work of [1], that proposes a simple and efficient algorithm to construct ROBDDs for monotonic Boolean functions, and a corresponding generalized arc-consistent SAT encoding. The algorithm proposed in [1] is a dynamic, bottom up BDD construction algorithm, which runs in polynomial time with respect to the ROBDD size and the number of variables. The key point is that it keeps the intervals of the PB constraints built for the already visited nodes: for a given variable ordering, say $x_1, x_2, \ldots, x_n$, a list of *layers* $\mathcal{L} = L_1, \ldots, L_{n+1}$ is maintained, where each layer L_i is a set of pairs of the form $([\beta, \gamma], \mathcal{B})$, being $\mathcal{B}$ the ROBDD of the constraint $a_ix_i + \cdots + a_nx_n \ \leq \ K$, for every K in the interval $[\beta, \gamma]$.

These intervals are used to detect if some needed ROBDD has already been constructed. That is, if for some node at level i, the ROBDD for the constraint $a_ix_i + \cdots + a_nx_n \ \leq \ K$ is needed for a given K, and K belongs to some interval already computed in layer L_i, then the same ROBDD can be used for this node. Otherwise, a new ROBDD is constructed and a new pair (the ROBDD and its respective interval) is added to the layer. It is important to recall here that ROBDDs are unique for a given function and variable ordering.

The encoding of the ROBDD to SAT that we borrow from [1] is generalized arc-consistent, and works as follows. For each node with a selector variable x we create, a new auxiliary variable n, which represents the state[5] of the node, and two clauses:

$$\bar{f} \rightarrow \bar{n} \qquad \bar{t} \wedge x \rightarrow \bar{n}$$

being f the state variable of its false child and t the state variable of its true child. Finally, we add a unit clause with the state variable of the 1-terminal node, another clause with the negation of the variable of the 0-terminal node. To make the encoding generalized arc-consistent, a unit clause forcing the state variable of the root node to be *true* must be added.

We refer the reader to [1] for additional details on the BDD construction algorithm and the SAT encoding.

4 Solving WCSPs by Incremental Optimization using Shared ROBDDs

The WCSP solving method presented here consists in reformulating the WCSP into a COP, and solving the resulting optimization problem by iteratively calling an SMT solver with the problem instance, together with successively tighter bounds for the objective function.

The novelty of the method lies in the way the objective function is treated. Inspired by the idea of intervals in the BDD construction algorithm of [1], our aim is to represent the pseudo-Boolean objective function resulting from the WCSP (see Subsection 2.1) as a BDD, and take profit of BDD reuse in successive iterations of the optimization process. That is, instead of creating a (reduced ordered) BDD from scratch at every iteration, we build a shared BDD. Since the PB constraint encoded at each iteration is almost the same, with only the bound constant K changing, this will hopefully lead to high node reuse.

We claim that using shared BDDs has two important benefits. The first one, fairly evident, is that node reuse considerably reduces the BDD construction time. The second one, which is not so evident, is that, as a by-product, we will be reusing many literals and clauses resulting from the SAT encoding of BDDs from previous iterations (in addition to clauses learned at previous steps). In Section 5 we present some experiments to support these claims.

4.1 Incremental optimization algorithm

Algorithm 1 describes our WCSP solving method. The input of the algorithm is a WCSP instance divided into a set φ_s of soft constraints and a set φ_h of hard constraints, and its output is the optimal cost of $\varphi_s \cup \varphi_h$ if φ_h is satisfiable, and UNSAT otherwise.

[5] That is, if the PB constraint corresponding to (the interval of) the node is satisfied or not.

Algorithm 1 Solving a WCSP by incremental optimization using a shared ROBDD

Input: $\varphi_s = \{(C_1, w_1), \ldots, (C_m, w_m)\}$, $\varphi_h = \{C_{m+1}, \ldots, C_{m+m'})\}$
Output: Optimal cost of $\varphi_s \cup \varphi_h$ or *UNSAT*

```
φ ← φ_h ∪ reif_soft(φ_s)
(st, M) ← SMT_algorithm(φ)
if st = UNSAT then
  return UNSAT
else
  ub ← sum({w_i | (C_i, w_i) ∈ φ_s and C_i is falsified in M})
end if
lb ← −1
L ← init_layers(φ_s)
while ub > lb + 1 do
  K ← ⌊(ub + lb)/2⌋
  ([β, γ], B) ← ROBDD(φ_s, K, L)
  (root, φ) ← BDD2SAT(B, φ)
  φ ← φ ∪ {root}
  (st, M) ← SMT_algorithm(φ)
  if st = UNSAT then
    lb ← γ
    φ ← (φ \ {root}) ∪ {¬root}
  else
    ub ← min(β, sum({w_i | (C_i, w_i) ∈ φ_s and C_i is falsified in M}))
  end if
end while
return ub
```

The first step is to reify the soft constraints (**reif_soft**), and create a formula φ with the reified soft constraints together with the hard constraints. By reifying a soft constraint C_i we mean adding a hard constraint $x_i \leftrightarrow \overline{C_i}$, where x_i is a new pseudo-Boolean variable. Note that in fact we are reifying the negation of C_i, since the cost w_i is associated to its violation (see Subsection 2.1).

Then, the satisfiability of φ is checked by an SMT solver. **SMT_algorithm** returns a tuple with the satisfiability (st) of φ and, if satisfiable, an assignment (model) M of φ. Note that if φ is unsatisfiable, this implies that φ_h is unsatisfiable, and hence we return UNSAT. Otherwise, we use the solution found to compute an upper bound ub of the objective function by aggregating the weights of the violated soft constraints, and set the lower bound lb to -1.

Before starting a binary search procedure to determine the optimal cost, in $\mathcal{L} \leftarrow$ **init_layers**(φ_s) we initialize each layer i of $\mathcal{L}$ (for i in $1..m+1$) with the two pairs of intervals and (trivial) ROBDDs $\{((-\infty, -1], \mathbf{0}), ([\sum_{j=i}^{m} w_j, \infty), \mathbf{1})\}$, meaning that the PB constraint $w_i x_i + \cdots + w_m x_m \leq K$ is trivially false for $K \in (-\infty, -1]$ and trivially true for $K \in [\sum_{j=i}^{m} w_j, \infty)$, where x_i,...,x_m denote the reification variables of the soft constraints C_i,...,C_m as described above (see Subsection 3.1 for the definition of layer and interval).

In the first step of the `while` statement, we determine a new tentative bound K for the objective function. Then, we call the **ROBDD** construction algorithm of [1] (briefly described in Subsection 3.1) with the set of soft clauses φ_s, the new bound K and the list of layers $\mathcal{L}$, being this last an input/output parameter. This way, $\mathcal{L}$ will contain the shared ROBDD with all the computed ROBDDs, and may be used in the following iterations of the search, significantly reducing the construction time and avoiding the addition of repeated clauses. This procedure returns the ROBDD $\mathcal{B}$ representing the objective function for the specific K in the current iteration.

In the next step we call the **BDD2SAT** procedure, which generates the propositional clauses from $\mathcal{B}$, as explained in Subsection 3.1, but only for the new nodes. The procedure inserts these clauses into the formula φ, and the new formula is returned together with the auxiliary variable *root* associated to the root node of $\mathcal{B}$. This variable is inserted into φ as a unit clause to effectively force the objective function to be less or equal than K.

At this point we call the SMT solver to check the satisfiability of the new φ. We remark that, since we are using the SMT solver through its API, we only need to feed it with the new clauses. If φ is satisfiable we can keep all the learned clauses. Otherwise, we need to remove the unit clause for the root node. This way, we will (only) remove the learned clauses related to this unit clause. In addition, we add a unit clause with the negation of the root node variable, stating that the objective function value must be greater than K.

Finally, we update either the lower or upper bound according to the interval $[\beta, \gamma]$ of the ROBDD $\mathcal{B}$ and the computed assignment M for φ: if φ is unsatisfiable, then the lower bound is set to γ; otherwise, the upper bound is set to $min(\beta, sum(\{w_i \,|\, (C_i, w_i) \in \varphi_s \text{ and } C_i \text{ is falsified in } M\}))$. From the invariant that the lower bound always corresponds to an unsatisfiable case, while the upper bound corresponds to a satisfiable case, when $ub = lb + 1$ this value corresponds to the optimum.

Note that, thanks to the intervals, in fact we are checking the satisfiability of the PB constraints for several values at the same time and hence, sometimes, this can allow us to obtain better lower/upper bound updates.

Example 1. Figure 2 and Figure 3 show, respectively, the evolution of the shared ROBDD and the propositional clauses added to the SMT formula for the objective $2x_1 + 3x_2 + 4x_3 \leq K$, with successive values $K = 7$, $K = 3$, $K = 5$ and $K = 4$.

5 Benchmarking

In this section we first compare the performance of the presented solving method with that of other methods implemented in `WSimply`. Second, we compare the performance of `WSimply`, using this new method, with that of several state-of-the-art CSP, WCSP and ILP solvers. Third, we study the benefits, such as learning, obtained from using a shared BDD for the objective instead of a linear

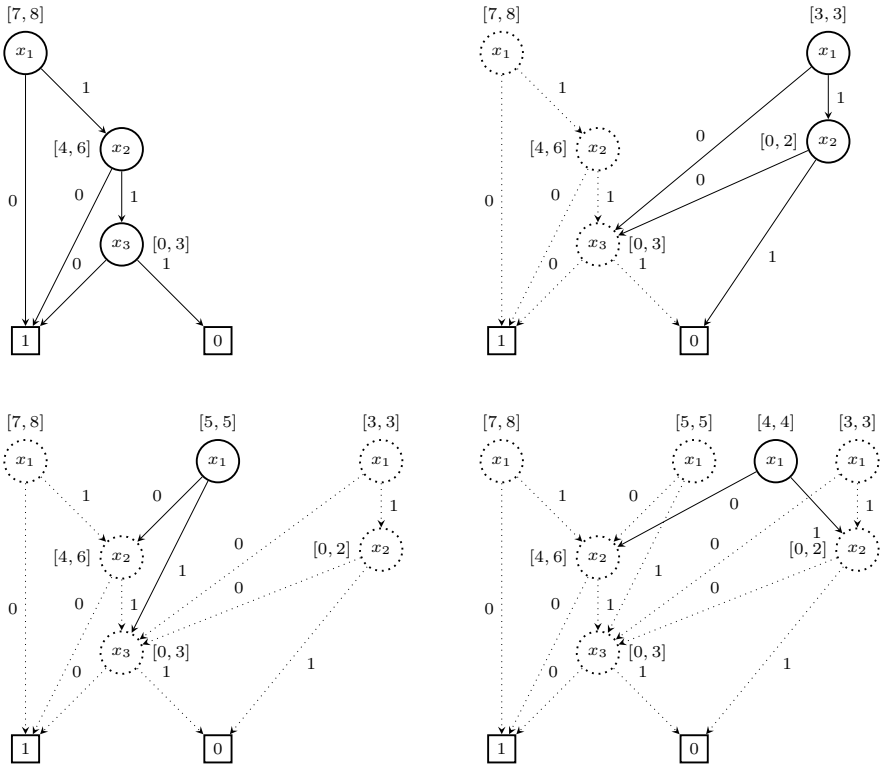

Fig. 2. Shared ROBDDs and intervals for objective $2x_1+3x_2+4x_3 \leq K$, with successive values $K = 7$ (top left), $K = 3$ (top right), $K = 5$ (bottom left) and $K = 4$ (bottom right), illustrating the reuse of previous ROBDDs.

$K = 7:\ \varphi_1 = \varphi \cup \{\overline{t} \to \overline{n_{1,1}}, \overline{n_{2,1}} \wedge x_1 \to \overline{n_{1,1}}, \overline{t} \to \overline{n_{2,1}}, \overline{n_{3,1}} \wedge x_2 \to \overline{n_{2,1}}, \overline{t} \to \overline{n_{3,1}},$
$\overline{f} \wedge x_3 \to \overline{n_{3,1}}, t, \overline{f}\} \cup \{n_{1,1}\}$

$K = 3:\ \varphi_2 = \varphi_1 \cup \{\overline{n_{3,1}} \to \overline{n_{1,2}}, \overline{n_{2,2}} \wedge x_1 \to \overline{n_{1,2}}, \overline{n_{3,1}} \to \overline{n_{2,2}}, \overline{f} \wedge x_2 \to \overline{n_{2,2}}\} \cup \{n_{1,2}\}$

$K = 5:\ \varphi_3 = (\varphi_2 \setminus \{n_{1,2}\}) \cup \{\overline{n_{1,2}}\} \cup \{\overline{n_{2,1}} \to \overline{n_{1,3}}, \overline{n_{3,1}} \wedge x_1 \to \overline{n_{1,3}}\} \cup \{n_{1,3}\}$
(supposing that φ_2 has been found to be unsatisfiable in iteration 2)

$K = 4:\ \varphi_4 = \varphi_3 \cup \{\overline{n_{2,1}} \to \overline{n_{1,4}}, \overline{n_{2,2}} \wedge x_1 \to \overline{n_{1,4}}\} \cup \{n_{1,4}\}$

Fig. 3. Formula update (clauses added and removed), for objective $2x_1+3x_2+4x_3 \leq K$ with successive values $K = 7$, $K = 3$, $K = 5$ and $K = 4$, according to the shared ROBDDs of Figure 2. φ_j denotes the SMT formula at hand when checking satisfiability in iteration number j of Algorithm 1. Each atom $n_{i,j}$ is the state variable of the node with selector variable x_i, that is added in iteration j, and t and f are the variables of the 1-terminal and 0-terminal nodes, respectively.

Table 1. Average number of variables, and weights, of the soft constraints in each pack of instances. Weights are denoted as $(min - max)$ for ranges and as $(val_1, val_2, \dots)$ for packs with a small number of distinct values in weights.

Problem	#Variables	Weights	Problem	#Variables	Weights
sbacp	67	1	*talent*	80	(4,5,10,20,40)
sbacp_h1	67	1			
sbacp_h2	67	1	*auction*	138	∼(100-1000)
sbacp_h2_ml2	312	(1,246)			
sbacp_h2_ml3	332	(1,21,5166)	*spot5*	385	(1-5,1000,2000)
s.l.	30	1	*celar*	1482	(1,10,100,1000)
s.l. free	30	1			(1,100,10000,100000)
s.l. no border	30	1			(1,2,3,4)

equation, and the amount of BDD node reutilization between successive iterations in the optimization process.

For the comparison we use (variants of) six problems:[6] five variants of the Soft BACP (SBACP) [5] (a softened version of the well-known Balanced Academic Curriculum Problem), three variants of the Still Life Problem and one variant of the Talent Scheduling Problem from the MiniZinc distribution[7] (all of them reformulated as a WCSP) and three classical WCSPs: CELAR, SPOT5 and Combinatorial Auctions.

Since the size of the generated BDDs strongly depends on the number of variables and the number of distinct coefficients in the objective function, we briefly describe these features for the studied problems in Table 1.

The experiments were run on a cluster of Intel® Xeon™CPU@3.1GHz machines, with 8GB of RAM, under 64-bit CentOS release 6.3 (kernel 2.6.32). `WSimply` was run on top of the Yices 1.0.33 [16] SMT solver. It is worth noting that by calling Yices through its API, we are able to keep learned clauses from previous calls that are still valid.

5.1 `WSimply` solving methods comparison

Table 2 shows the aggregated time (sum of the times to produce an optimal solution and prove its optimality, for all instances) for the solved instances. We consider the `yices`, `core` and `dico` solving methods described in Subsection 2.1, plus the new method `sbdd`$\leq$ using shared ROBDDs, where variables in the BDD are ordered from small (root) to big (leaves) coefficients. We also studied the performance of `sbdd`$\geq$, where variables are ordered from big (root) to small (leaves) coefficients, but the performance of `sbdd`$\leq$ was slightly better.

The new solving method with shared ROBDDs is clearly the best in the majority of problems. The performance of the `sbdd`$\leq$ method is better on instances with a small number of distinct coefficients (especially when all coefficients are 1), namely, in the *sbacp* and *still life* problems. In these cases, the BDDs are relatively small, having the bigger ones only thousands of nodes. On the other

[6] All instances available in `http://ima.udg.edu/Recerca/lap/simply`

[7] `http://www.minizinc.org`

Table 2. Aggregated time in seconds for the solved instances in each set, with a cutoff of 600s per instance. The column # indicates the number of instances per set. The indication (n^1) refers to the number of instances for which the solver ran out of time, (n^2) refers to the number of instances for which the solver returned *unknown*, and (n^3) refers to the number of instances for which the solver ran out of memory.

Problem	#	`dico`	`yices`	`core`	`sbdd`$\leq$
sbacp	28	70.53	40.17	1342.16 (13^1)	**9.83**
sbacp_h1	28	**6.68**	12.02	363.78 (12^1)	6.93
sbacp_h2	28	26.04	26.65	748.81 (13^1)	**9.97**
sbacp_h2_ml2	28	280.68	109.42	75.82 (19^1)	**78.36**
sbacp_h2_ml3	28	804.74	516.51	547.55 (23^1)	**92.37**
s.l.	10	118.05 (1^1)	102.97 (1^2)	3.35 (4^1)	**191.56**
s.l. free	10	434.05 (2^1)	2.69 (3^2)	386.91 (3^1)	**98.75**
s.l. no border	10	187.94 (1^1)	166.91 (1^2)	81.24 (3^1)	**105.30**
talent	11	289.90	60.72	94.82 (8^1)	**38.25**
auction	170	9084,03 (93^1)	**1083.44 (85^2)**	0 (80^1 90^3)	12028.50 (78^1 11^3)
celar	16	582,64 (14^1)	750.32 (10^1 2^2)	94.82 (8^1 6^3)	**1417.19 (6^1)**
spot5	20	205,2 (18^1)	9.32 (5^1 13^2)	94.82 (14^1 3^3)	**1990.59 (7^1)**

side, the timeouts occurring in the *celar* and *spot5* problems are mostly in the instances with higher number of variables in the objective function (e.g., from approximately 1200 to 5300 variables in *celar*), generating BDDs of hundreds of thousands nodes (a similar situation occurs in the *auction* instances running out of time, where there are not so many variables but a lot of distinct variable coefficients). In spite of this, `sbdd`$\leq$ is still the best method on the *celar* and *spot5* problems.

It is worth to notice that there are some *unknowns* in the `yices` method, due to the fact that the Yices MaxSMT solver is non exact and incomplete. The `core` method has really bad performance on the crafted instances considered in this experiment, probably due to the bad quality of the unsatisfiable cores found during the solving process.

5.2 SBDD-based versus state-of-the-art CSP and WCSP solvers

For the sake of completeness we tested the performance of the MATHSAT5-MAX and LL_{WPM} MaxSMT solvers of [12] on the weighted SMT instances generated by `WSimply` in the previous experiment. However we do not report these results, since they are comparable to those of the `WSimply core` method.

The second part of our experiments consists in reformulating the WCSPs into (MiniZinc) COPs, and compare the performance of `WSimply` using the `sbdd`$\leq$ method with that of IBM ILOG CPLEX 12.6 and some state-of-the-art CSP and WCSP solvers, namely Toulbar2 0.9.5.0, G12-CPX 1.6.0 and Opturion 1.0.2, all of them capable of solving MiniZinc instances. We used Numberjack [19] as a platform for solving MiniZinc COPs through the Toulbar2 API. We also used the Toulbar2 binary to solve the original WCSP instances of *auction*, *celar* and *spot5*, indicated in Table 3 as *auction (wcsp)*, *celar (wcsp)* and *spot5 (wcsp)*,

Table 3. Aggregated time in seconds for the solved instances in each set, with a cutoff of 600s per instance. The column # indicates the number of instances per set. The indication (n^1) refers to the number of instances for which the solver ran out of time, (n^3) refers to the number of instances for which the solver ran out of memory, and (n^4) refers to the number of instances for which the solver returned another type of error.

Problem	#	**TB2**	**G12-CPX**	**Opturion**	`sbdd`$\leq$	**CPLEX**
sbacp	28	662.58 (24^1)	76.31	83.47	**9.83**	36.81
sbacp_h1	28	707.78	**4.00**	13.25	6.93	21.78
sbacp_h2	28	483.98	19.53	30.51	**9.97**	24.89
sbacp h2_ml2	28	3849.38 (6^1)	166.69	91.55	78.36	**64.98**
sbacp h2_ml3	28	2408.66 (12^1)	336.25	99.77	92.37	**72.95**
s.l.	10	166.08 (2^1)	232.1 (1^1)	43.81 (2^1)	**191.56**	234.71
s.l. free	10	69.85 (3^1)	267.13 (2^1)	394.28 (2^1)	98.75	**29.91**
s.l. no border	10	122.05 (2^1)	354.41 (1^1)	102.29 (2^1)	105.3	**68.37**
talent	11	2.38 (9^4)	1.81 (8^1)	2.07 (8^1)	**38.25**	1269.13 (2^1)
auction	170	888.99	7288.73 (95^1)	7053.43 (100^1)	12028.5 (78^1 11^3)	**220.12**
auction (wcsp)		5038.49 (3^1)				
celar	16	0 (16^1)	0 (16^4)	0 (16^1)	1417.19 (6^1)	0 (16^1)
celar (wcsp)		**311.23 (4^1)**				
spot5	20	114.66 (8^1 8^4)	0 (20^1)	0 (20^1)	**1990.59 (7^1)**	297.59 (7^1 10^4)
spot5 (wcsp)		76.28 (16^1)				

respectively. In order to test the performance of CPLEX on the considered problems, we used `WSimply` to translate the instances to pseudo-Boolean constraints as in [8]. For the experiments with Opturion we used a slightly different computer (Intel® Core™CPU@2.8GHz, with 12GB of RAM, under 64-bit Ubuntu 12.04.3, kernel 3.2.0) due to some library dependence problems.

Table 3 shows the results of this second experiment. We can observe that, in general, `sbdd`$\leq$ is the most robust method, considering the number of instances solved and their aggregated solving time. The *sbacp* and *still life* problems seem to be reasonably well suited for SMT solvers (in particular, the latter consists of disjunctions of arithmetic constraints). We highlight the *talent scheduling* problem, where `sbdd`$\leq$ clearly outperforms the other solvers, being the only solver capable to solve all the instances. In fact, this is probably the best suited problem for SMT solvers, as it consists of binary clauses of *difference logic* constraints. Unfortunately, in most of the instances of this problem, Toulbar2 reported an error saying that the model decomposition was too big. In the *auction* problem, CPLEX is by far the best solver, solving all 170 instances in only 220.12 seconds. This is clearly one of the worst kind of problems for SMT solvers, as it simply consists of a conjunction of arithmetic (0/1) constraints, i.e., it has a trivial Boolean structure and all deductions need to be performed by the theory solver. For *celar*, only `sbdd`$\leq$ and Toulbar2 (in the WCSP version) were able to solve some instances (10 and 12 out of 16, respectively). This problem also has a balanced combination of arithmetic constraints and clauses. G12-CPX reported an

error (not finding the `int_plus` constraint), and Toulbar2 (in the COP version), Opturion and CPLEX ran out of time on all instances. Finally, for *spot5*, `sbdd`$\leq$ was able to solve some instances (13 out of 20), Toulbar2 only 4, CPLEX only 3, and Opturion and G12-CPX ran out of time on all instances. This problem is also well suited for SMT solvers since it basically consists of clauses with equality constraint literals.

We remark that we also tested the G12-Lazy, G12-FD and Gecode 4.2.1 solvers, but since they presented slightly worst performance than G12-CPX we have not included them in Table 3.

5.3 SBDD incrementality

In this section we study the benefits of using a shared BDD for the objective instead of a linear equation, in particular, the effect that this has on the learning capability of the SMT solver. Note that with this technique we are increasing the Boolean component of the formula at hand and, hence, we should expect some improvement in learning. Finally, we quantify the amount of BDD node reutilization through iterations.

We compare the performance of four solving approaches: using either SBDDs or arithmetic expressions to bound the objective function, with and without using the learning capabilities of the solver. From now on we will denote by SBDD+L, SBDD-L, LIA+L and LIA-L these four possibilities, where LIA stands for linear integer arithmetic expressions and +L/-L indicates the use or not of learning. Note that `WSimply` uses Yices as a back-end solver, and it does not provide statistics about learned clauses. Therefore, we used the Yices commands *push* and *pop* to define backtrack points and to backtrack to that points, respectively, in order to force the solver to forget learned clauses.

Table 4 summarizes the solving times for the four options. They do not include neither BDD construction, clause generation nor assertion times. Since solving times are very sensitive to the bounds on the objective function, first of all we solved all the instances with the SBDD+L approach and stored the bounds at each iteration. Then these bounds were passed to the other three methods to obtain their solving times under the similar conditions.

If we first compare SBDD-L and LIA-L we can appreciate that using BDDs considerably reduces the number of timeouts in *still life*, *auction*, *celar* and *spot5*, and the solving time, almost 5 times in *sbacp* and 10 times in *talent scheduling*.

Furthermore, comparing SBDD+L and SBDD-L, we can easily appreciate that learning reduces even more the solving times. In *sbacp* the number of timeouts is reduced to 0 and the sum of solving times is reduced almost 4 times; in *talent scheduling* and *still life* the solving time is reduced almost to the half; and in *celar* and *spot5* the number of timeouts is reduced in 2 and 3 instances respectively. In *auction*, the SBDD+L method has the drawback of increasing the number of memory outs.

If we compare SBDD-L and LIA-L with respect to SBDD+L and LIA+L, we can see that the improvement in terms of number of timeouts and solving time is higher in the SBDD approach than in the LIA approach.

Table 4. Aggregated time in seconds for the solved instances of each problem, with a cutoff of 600s per instance, and indication of the number of unsolved instances due to time out (TO) and memory out (MO).

		SBDD+L		**SBDD-L**		**LIA+L**		**LIA-L**	
Problem	# Inst.	Solving	# TO/MO	Solving	# TO/MO	Solving	# TO	Solving	# TO
sbacp	140	58.03	0	221.30	1	853.56	0	924.83	2
s.l.	30	393.22	0	715.17	0	795.62	4	895.28	4
talent	11	35.36	0	55.57	0	363.43	0	472.09	0
auction	170	9631.19	73/12	5673.22	75/7	8540.76	92	9919.79	90
celar	16	1199.99	6	1299.99	8	10.94	15	8.29	15
spot5	20	1675.90	7	943.96	10	165.48	18	59.99	18

Finally, to quantify the contribution of the shared BDD to the amount of BDD node reutilization, we computed the percentage of reused nodes at each iteration of the optimization process. Our experiments shown an average 50% of node reuse when the solving process was about the 40.73% of the search, and a 80% of node reuse when it was about the 48.72% of the search. This is especially relevant because the BDDs attained hundreds of thousands of nodes.

6 Conclusions and Future Work

We have presented a new WCSP solving method, implemented in the `WSimply` system, based on using shared ROBDDs to generate propositional clauses representing the objective. We think that it opens a promising research line, taking into account that the presented method clearly outperforms not only the previously implemented solving methods in `WSimply`, but also some state-of-the art solvers, on several problems. We have also shown how to boost the generation of ROBDDs for objective functions using previously generated ROBDDs, more precisely constructing a shared ROBDD.

As future work we want to study more deeply the efficiency of our method on other weighted CSPs. Also, as a well-known challenge, a crucial aspect to study is how to find a good variable ordering for the objective function. Although the problem of finding the optimal variable ordering in order to generate a minimal BDD is known to be NP-hard, we are interested in finding a variable ordering that maximizes the node reuse through iterations. Another aspect that could be interesting to explore is to extend the new method to deal with objective functions with (finite domain) integer variables, using Multi-valued Decision Diagrams (MDDs) to represent them. Finally, we also want to test if making visible the intermediate literals of the arithmetic representation of the objective function could benefit the `dico` solving method.

Acknowledgments. The authors would like to thank Simon de Givry for providing the CELAR, SPOT5 and Combinatorial Auctions WCSP instances, and Roberto Sebastiani for providing the MATHSAT5-MAX and LL_{WPM} MaxSMT solvers for the experiments. The authors also thank the anonymous referees for suggestions to improve the paper.

References

1. I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A New Look at BDDs for Pseudo-Boolean Constraints. *Journal of Artificial Intelligence Research (JAIR)*, 45:443–480, 2012.
2. S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.
3. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. A Proposal for Solving Weighted CSPs with SMT. In *Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011)*, pages 5–19, 2011.
4. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. Satisfiability Modulo Theories: an Efficient Approach for the Resource-Constrained Project Scheduling Problem. In *Proceedings of the 9th Symposium on Abstraction, Reformulation and Approximation (SARA 2011)*, pages 2–9, 2011.
5. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving weighted CSPs with meta-constraints by reformulation into Satisfiability Modulo Theories. *Constraints*, 18(2):236–268, 2013.
6. C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *LNCS*, pages 427–440. Springer, 2009.
7. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
8. M. Bofill, J. Espasa, M. Palahí, and M. Villaret. An extension to Simply for solving Weighted Constraint Satisfaction Problems with Pseudo-Boolean Constraints. In *XII Spanish Conference on Programming and Computer Languages (PROLE 2012)*, pages 141–155, Almería, Spain, September 2012.
9. M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.
10. E. Boros and P. L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.
11. A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *Proceeding of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, volume 6015 of *LNCS*, pages 99–113. Springer, 2010.
12. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. A Modular Approach to MaxSAT Modulo Theories. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT 2013)*, volume 7962 of *LNCS*, pages 150–165. Springer, 2013.
13. S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 84–89, 2005.
14. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

15. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification, CAV'06*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
16. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf, August 2006.
17. N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):1–26, 2006.
18. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21 – 70, 1992.
19. E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint Programming and Combinatorial Optimisation in Numberjack. In *Proceedings of the 7th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CPAIOR 2010)*, volume 6140 of *LNCS*, pages 181–185. Springer, 2010.
20. S. ichi Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation (DAC 1990)*, pages 52–57, 1990.
21. J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-Consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.
22. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. *SAT Handbook*, pages 131–154, 2009.
23. P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 9. Elsevier, 2006.