Boosting Weighted CSP Resolution with Shared 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 new approach for solving Weighted Constraint Satisfaction Problems (WCSP). The method is based on encoding the violation cost of soft constraints as a pseudo-Boolean objective function, and successively calling a decision procedure bounding the maximum allowable cost. The novelty of our approach consists in building a Binary Decision Diagram (BDD) for the objective function, using state-of-the-art generalized arc-consistent SAT encodings for it. Moreover, with our approach we maximize the reuse of the BDDs for the objective function between successive calls to the decision procedure by creating a shared BDD. The method has been incorporated into the WCSP solving system WSimply, based on reformulation into SMT, with preliminary encouraging results.

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 usually 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 kind 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 *hard*. Then, the objective is to find an assignment which satisfies all hard constraints and minimizes the aggregated cost of the violated soft constraints [21]. These problems are known as Weighted CSP (WCSP) [20] or, alternatively, as Cost Function Networks (CFN) [14].

* Partially supported by the Spanish Ministry of Science and Innovation through the projects TIN2012-33042, and by the Universitat de Girona under grant BR2010.

`WSimply` [4,6] is a language and system for solving intensionally represented WCSPs by reformulation into Satisfiability Modulo Theories (SMT) [8], 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 [5,10]. 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.

`WSimply` benefits from the expressiveness of the SMT language and the performance of current SMT solvers. However, SMT solvers are decision procedures, and they scarcely support optimization. A few solvers support (weighted partial) MaxSMT [16,15,13], and there is a recent attempt to introduce optimization into SMT by means of a theory of costs [12]. 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 [7].

In this paper we extend the WCSP solving capability of `WSimply` by introducing a new optimization approach, based on representing the objective function (generated from the violation cost of the soft constraints) as a BDD [2]. This allows us to encode the objective function as a pure propositional formula, following the generalized arc-consistent encodings proposed in [1]. This way, 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. An interesting aspect of our approach is the reutilization of BDDs in successive calls to the decision procedure. Although changing the bounds of the objective function implies building a new BDD, some parts can be easily reused. We create a Shared BDD [19], 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 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 encouraging preliminary results on the Soft Balanced Academic Curriculum Problem introduced in [4,6] and on a WCSP version of the Maximal Density Still Life Problem.

The paper is structured as follows. In Section 2 we introduce the required background on Weighted Constraint Satisfaction Problems (WCSP). In Sec-

tion 3 we introduce Pseudo-Boolean constraints and (Reduced Ordered) Binary Decision Diagrams (ROBDD). In Section 4 we present our method for solving WCSPs using shared ROBDDs. In Section 5 we study the performance of the new method and we compare it with the previous ones implemented in `WSimply`. Finally, in Section 6 we conclude and propose future work.

2 Solving WCSPs with SMT

The expert reader can skip the first two subsections and directly go to the solving algorithms described in Subsection 2.3.

2.1 CSPs, 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. Each constraint $C_i = \langle S_i, R_i\rangle$ is defined as a relation R_i over a subset of variables $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ (called the *constraint scope*) which specifies the allowable combinations of values for that subset. Usually, the relation R_i is represented *intensionally* as a condition that the assignments to the variables must satisfy (e.g., $x_1 < x_2$). An *assignment* v for a CSP instance $\langle X, D, C\rangle$ is a mapping that assigns to every variable $x_i \in X$ an element $v(x_i) \in d(x_i)$. An assignment v *satisfies* a constraint $\langle\{x_{i_1}, \ldots, x_{i_k}\}, R_i\rangle$ in C if and only if $\langle v(x_{i_1}), \ldots, v(x_{i_k})\rangle \in R_i$. A *solution* to a CSP instance is an assignment to its variables that satisfies all the constraints. The Constraint Satisfaction Problem for a CSP instance consists in finding a solution for that instance.

A *weighted CSP (WCSP)* instance is a triple $\langle X, D, C\rangle$, where X and D are variables and domains, respectively, as in a CSP. A constraint C_i is now defined as a pair $\langle S_i, f_i\rangle$, where $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ is the constraint scope and $f_i : d(x_{i_1}) \times \cdots \times d(x_{i_k}) \to \mathbb{N} \cup \{\infty\}$ is a *cost (weight) function* that maps tuples to its associated cost (a natural number or infinity). Forbidden tuples receive infinite cost. The cost of a constraint C_i induced by an assignment v in which the variables of $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ take values $b_{i_1}, \ldots, b_{i_k}$ is $f_i(b_{i_1}, \ldots, b_{i_k})$. A *solution* to a WCSP instance is an assignment to its variables which makes the sum of the costs of the constraints minimal. The Weighted Constraint Satisfaction Problem for a WCSP instance consists in finding a solution for that instance.

In this paper we assume to deal with weighted constraints $(c, w(c))$, where c is a constraint as defined for a CSP and $w(c)$ is the cost corresponding to its falsification, which can be a natural number or infinity. Note that this corresponds to an intensional formulation of the definition of *cost function* given above. We call those constraints whose associated cost is infinity *hard*, if otherwise *soft*. In this setting, the cost of a variable assignment corresponds to the sum of all weights of the constraints that are violated under the assignment.

A *Constraint Optimization Problem (COP)* instance consists of an optimization variable O matched to an objective function to be minimized (or 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 (or maximizes) the value of the optimization variable O.

2.2 SMT and Weighted SMT

A *Satisfiability Modulo Theories (SMT)* instance 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. Predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory [8]. 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 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 WCSPs

Now we describe the reformulations and solving procedures of `WSimply` that are relevant for the present work. We remark that in this paper we assume that costs of soft constraints are constant natural numbers.[3]

The input of `WSimply` is a WCSP instance written in the `WSimply` language (we refer the reader to [9,6] for details). This is reformulated (according to a command line option indicating the solving method) either as a COP or as a WSMT instance:

– When reformulating a WCSP instance into a COP instance, each soft constraint C_i with weight w_i is replaced by the following constraints:

$$C_i \rightarrow (o_i = 0) \tag{1}$$
$$\neg C_i \rightarrow (o_i = 1) \tag{2}$$

[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] In `WSimply` costs can be defined by a linear integer arithmetic expression.

where o_i is a fresh (pseudo-Boolean) variable reifying the violation of C_i. Secondly, the objective function is defined by introducing a fresh integer variable O to be the aggregation of violation costs, with the constraint:

$$O = \sum_{i=1}^{m} w_i * o_i \tag{3}$$

The variable O is the variable to be minimized in the resulting COP. Original hard constraints are kept without modification.

– The reformulation of a WCSP instance into a WSMT instance is trivial: each soft constraint C_i with weight w_i is replaced by the WSMT clause (C'_i, w_i) where C'_i is the translation of C_i into SMT as described in [9] (recall that our method is based on translation of CSPs into SAT modulo linear integer arithmetic). Finally, hard constraints are replaced by their equivalent hard SMT clauses.

Once the WCSP instance has been reformulated, the system can solve it by means of one of three methods, called `yices`, `core` and `dico` respectively. The two first methods allow to solve WSMT instances, while the last one allows to solve COP instances:

– 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 [6], of the core based WPM1 algorithm [7] from the MaxSAT field.
– In the `dico` method, the system first translates the constraints of the COP into SMT formulae, and then incrementally calls an 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. Note that bounding the objective function of a COP which encodes a WCSP instance, where the weights of soft constraints are natural numbers, results into a pseudo-Boolean constraint (see Section 3).

3 SAT Encodings of Pseudo-Boolean Constraints using BDDs

Pseudo-Boolean (PB) constraints [11] 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 $\{<, >, \leq, \geq, =\}$. For our purposes we assume that $\#$ is $\leq$ and that the a_i and K are positive. Under these assumptions, these constraints are monotonic (decreasing) Boolean

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

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.

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 and Figure 2 illustrate, respectively, a BDD and a ROBDD for the same PB constraint.

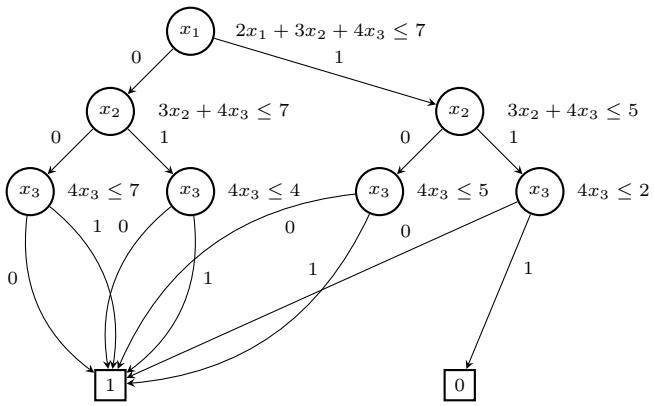

Fig. 1. BDD for $2x_1 + 3x_2 + 4x_3 \leq 7$.

There exist several BDD-based approaches for reformulating PB constraints into SAT clauses [17]. We focus on the recent work of [1], that proposes a simple and efficient algorithm to construct ROBDDs and a corresponding Generalized Arc Consistent (GAC) SAT encoding for monotonic Boolean functions.

A key point of that ROBDD construction algorithm is the reuse of BDDs, which is sustained by the so-called concept of *PB intervals*. Therefore, first of all we define the concept of PB interval. Let C be a constraint 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).

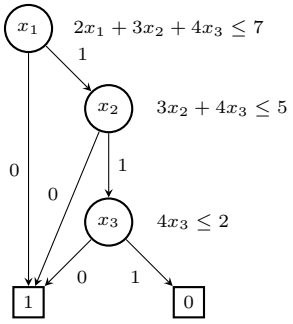

Fig. 2. ROBDD for $2x_1 + 3x_2 + 4x_3 \leq 7$.

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 $2x_1 + 3x_2 + 4x_3 \leq 8$. Since each node in a BDD represents a PB constraint, we can naturally overload the notion of interval and refer also to intervals of nodes.

The algorithm of [1] is a dynamic, bottom up BDD construction algorithm, which keeps the BDD intervals corresponding to the already visited nodes in *layers*: for a given variable ordering of a PB constraint, say $x_1, x_2, \ldots, x_n$, a list of layers $\mathcal{L} = L_1, \ldots, L_{n+1}$ is created. A layer L_i is a set of pairs of the form $([\beta, \gamma], \mathcal{B})$, where $\mathcal{B}$ is the ROBDD of the constraint $a_i x_i + \cdots + a_n x_n \leq K$ for every $K \in \{\beta..\gamma\}$. 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 $a_i x_i + \cdots + a_n x_n \leq K$ 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. It is important to recall here that ROBDDs are unique for a given function and variable ordering.

The algorithm first initializes each layer L_i, with i in $1..n+1$, with the pairs $((-\infty, -1], \mathbf{0})$ and $([\sum_{j=i}^{n} a_j, \infty), \mathbf{1})$. The ROBDD construction procedure has the following parameters: a PB constraint $a_i x_i + \cdots + a_n x_n \leq K$, the list of layers $\mathcal{L}$ and an index i denoting the current layer.

For instance, following the example of $2x_1 + 3x_2 + 4x_3 \leq 7$, the algorithm initializes $\mathcal{L}$ with:

$$L_1 = \{((-\infty, -1], \mathbf{0}), ([9, \infty), \mathbf{1})\}$$
$$L_2 = \{((-\infty, -1], \mathbf{0}), ([7, \infty), \mathbf{1})\}$$
$$L_3 = \{((-\infty, -1], \mathbf{0}), ([4, \infty), \mathbf{1})\}$$
$$L_4 = \{((-\infty, -1], \mathbf{0}), ([0, \infty), \mathbf{1})\}$$

Then, the construction procedure is called with $2x_1 + 3x_2 + 4x_3 \leq 7$, the list of layers $\mathcal{L}$, and index 1.

The first step of the construction procedure consists in searching in layer L_i if there exists a ROBDD $\mathcal{B}$ for the current K, i.e., if a pair $([\beta, \gamma], \mathcal{B})$ with $K \in \{\beta..\gamma\}$ exists in L_i. If so, the existing pair is returned, otherwise the procedure is recursively called for the two descendants increasing the index layer to $i+1$ and updating the K of the true child to $K - a_i$. Once the two descendants' ROBDD

are returned a new pair for the layer L_i is created. If the two returned ROBDDs are different, the $\mathcal{B}$ of the new pair will be a new ROBDD created from them, otherwise $\mathcal{B}$ will be the returned ROBDD of the children.

Following our example, the algorithm recursively calls the construction procedure for the two descendants with the parameters:

- True child: $3x_2 + 4x_3 \leq 5$, $\mathcal{L}$ and 2
- False child: $3x_2 + 4x_3 \leq 7$, $\mathcal{L}$ and 2

Since the two returned ROBDDs are different, the algorithm creates a new ROBDD $\mathcal{B}_1$ with pair $[7, 8]$, which will be inserted into L_1, resulting in:

$$\big\{((-\infty, -1], \mathbf{0}), ([7, 8], \mathcal{B}_1), ([9, \infty), \mathbf{1})\big\}$$

With the ROBDD constructed, we only have to encode it to SAT. As usual, the encoding introduces an auxiliary variable for every node. Let v be a node with selector variable x and auxiliary variable n. Let f be the variable of its false child and t be the variable of its true child. We only need to add two clauses per node:

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

and a unit clause with the variable of the 1-terminal and another one with the negation of the 0-terminal. Finally, we only have to add a unit clause forcing the variable of the root node to be *true*. This encoding is GAC.

Additional details of the BDD construction or the SAT encoding can be found in [1].

4 Solving WCSPs using Shared ROBDDs

As we have seen in Subsection 2.3, we can use a PB constraint to encode the objective function of a COP (possibly resulting from a WCSP). Moreover, we can use a ROBDD to represent this PB constraint and then encode it into SAT, as a GAC formula.

Our WCSP solving method consists in reformulating a WCSP into a COP, written in the SMT language, and solving the optimization problem by iteratively calling an SMT solver with the problem instance together with successively tighter bounds for the objective function. This is accomplished by adding the SAT encoding of the (tighter) PB constraints representing the objective function.

The fact of constructing ROBDDs to represent the PB constraints, using the same variable ordering and the same coefficients (we only change the K), may lead to have isomorphic subgraphs between ROBDDs. When multiple BDDs have isomorphic subgraphs they can be joined into a single *Shared BDD (SBDD)*, that is, a BDD with several roots [19].

The key point of our solving method is to construct a SBDD, in our case a Shared ROBDD (SROBDD), using the ROBDD construction method presented in Section 3. Note that when the ROBDD construction procedure finishes, in $\mathcal{L}$ we will have all the ROBDD nodes, with their corresponding intervals. Using

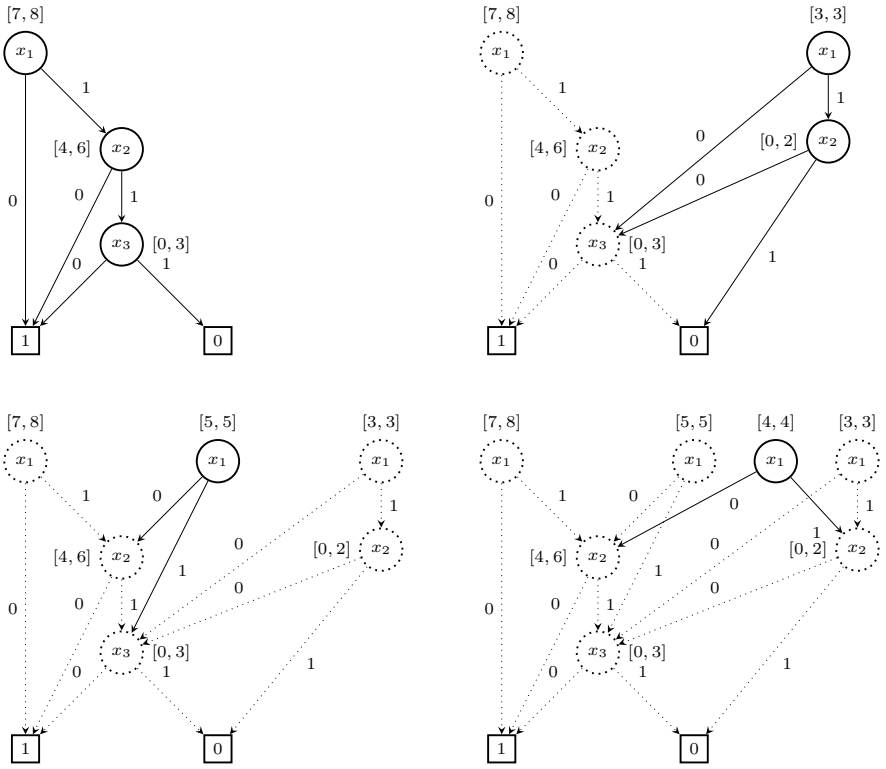

Fig. 3. ROBDDs and intervals for $2x_1 + 3x_2 + 4x_3 \leq 7$ (top left), $2x_1 + 3x_2 + 4x_3 \leq 3$ (top right), $2x_1 + 3x_2 + 4x_3 \leq 5$ (bottom left) and $2x_1 + 3x_2 + 4x_3 \leq 4$ (bottom right), illustrating the reuse of previous ROBDDs.

these pairs during the creation of all new constraints to bound the objective function, it is possible to create a SROBDD.

Figure 3 shows the evolution of the SROBDD representing the bounding constraint $2x_1 + 3x_2 + 4x_3 \leq K$, for $K = 7$ (sat), $K = 3$ (unsat), $K = 5$ (sat) and $K = 4$ (sat).

4.1 Optimization algorithm

Algorithm 1 describes our new WCSP solving method. First of all, we reify the soft constraints, we create a formula φ with the reified soft constraints together with the hard constraints, and check the satisfiability of φ. The function SMT_algorithm returns a tuple with the satisfiability (st) of φ and, if satisfiable, a model (M) of φ. If φ is unsatisfiable then the problem does not have any solution. 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.

Algorithm 1 Binary Search Algorithm using Shared ROBDD

Input: $\varphi_s = \{(C_1, w_1), \ldots, (C_n, w_m)\}$, $\varphi_h = \{C_{m+1}, \ldots, C_{m+m'})\}$
Output: 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) ← construct_ROBDD(φ_s, K, L)
   root ← 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
```

Before starting a binary search procedure, we initialize the list of layers $\mathcal{L}$ using the objective function, i.e., the weights of the soft constraints φ_s, as we have explained Section 3.

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 method described in Section 3, with the set of soft clauses φ_s, K and $\mathcal{L}$, being this last an input/output parameter. This way, $\mathcal{L}$ will contain the SROBDD 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 SAT clauses from $\mathcal{B}$, as explained in Section 3, but only for the new nodes. Then the procedure inserts these clauses into φ and returns the auxiliary variable associated to the root node of $\mathcal{B}$. This variable is inserted into φ as a unit clause to effectively force that the objective function has to be less or equal than K.

At this point we call the SMT solver to check the satisfiability of φ. If φ is satisfiable we can keep all the learned clauses. Otherwise, we only have 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 is not less or equal than K, i.e., it has to be greater than K.

Finally, we update the bounds of the search using the interval $[\beta, \gamma]$ of $\mathcal{B}$:

- If φ is unsatisfiable: $lb = \gamma$
- If φ is satisfiable: $ub = min(\beta, sum(\{w_i \mid (C_i, w_i) \in \varphi_s, C_i = \mathit{false}\}))$

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, we can compute more refined bounds.

Since we have the invariant that the lower bound always corresponds to an unsatisfiable case, while the upper bound corresponds to a satisfiable case, when $ub = lb + 1$ we are done.

5 Benchmarking

In this section we compare the performance of our new solving method to that of the methods already existing in `WSimply`. In particular, we use the following two problems (and some variants of them) for benchmarking:

- A softened version of the well-known Balanced Academic Curriculum Problem (BACP), the so-called Soft BACP (SBACP) [4,6]. In the BACP, a number of courses have to be scheduled in a limited number of periods, balancing students' load, and satisfying some prerequisite constraints between courses. In the SBACP the number of periods is reduced until all instances become unsatisfiable due to the prerequisites chain, and then the prerequisite constraints are considered to be soft. We use the five variants of the SBACP presented in [4,6] for this performance study.
- The Maximal Density Still Life Problem (Still Life) is to place the maximum number of live cells in a given board, so that the board configuration is stable under the Conway's Game of Life. We have translated this COP into a WCSP written in `WSimply`. We have considered the three harder variants of the problem found in the benchmarks folder of the MiniZinc distribution.[5] This COP is well-suited for our purposes because it has a PB optimization function.

Experiments have been run on an Intel® Core™ i5 CPU@2.66GHz, with 3GB of RAM, under 32-bit openSUSE 11.2 (kernel 2.6.31). We use `WSimply` with the API of the Yices 1.0.33 [16] SMT solver, with a cutoff of 600 seconds per run. By calling Yices through its API, we are able to keep learned clauses from previous calls that are still valid.

Table 4 shows the aggregated time per problem variant and solving method. We consider the solving methods `yices`, `core` and `dico`, already described in Subsection 2.3, plus the new method using SROBDDs, with two different variable

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

orderings: `sbdd≤` where variables in the BDD are ordered from small (root) to big (leaves) coefficients, and `sbdd≥` where variables are ordered from big (root) to small (leaves) coefficients.

	#	`dico`	`yices`	`core`	`sbdd≤`	`sbdd≥`
sbacp	28	95.95	58.94	394.48 (15 t.o.)	12.79	
sbacp_h1	28	5.68	13.28	544.22 (12 t.o.)	5.99	
sbacp_h2	28	32.01	35.05	1128.93 (13 t.o.)	10.68	
sbacp_h2_ml2	28	344.95	153.40	114.29 (19 t.o.)	126.35	94.30
sbacp_h2_ml3	28	866.63	741.48	59.42 (24 t.o.)	145.76	258.32
still life	10	160.87 (1 t.o.)	382.76 (1 unk.)	0.87 (4 t.o.)	269.64	
still life free	10	553.95 (2 t.o.)	553.33 (3 unk.)	46.86 (3 t.o.)	126.28	
still life no border	10	296.77 (1 t.o.)	407.96 (1 unk.)	0.57 (4 t.o.)	220.74	

Fig. 4. Aggregated times in seconds for the solved instances of the 5 variants of the SBACP and the 3 variants of Still Life. The column # indicates the number of instances per set. The indication (n t.o.) refers to the number of unsolved instances, with a cutoff of 600s per instance. The indication (n unk.) refers to the number of instances for which the solver returned *unknown*.

In the first three variants of the SBACP and in the Still Life sets, the `sbdd≤` and `sbdd≥` methods are the same because the coefficients are 1 for all variables of PB constraints (with an average of 67 variables per PB constraint in the SBACP and of 30 in Still Life).

In the last two variants of the SBACP the coefficients are different. In the *sbacp_h2_ml2* set, PB constraints have an average of 312 variables, with coefficients 1 and 246, while in the *sbacp_h2_ml3* set, PB constraints have an average of 332 variables, with coefficients 1, 21 and 5166. In both cases the coefficients are stratified, for example, in the *sbacp_h2_ml2* set, PB constraints have an average of 245 variables with coefficient 1 and 67 variables with coefficient 246.

In these two latter sets of instances of the SBACP, the two distinct variable orderings for constructing the BDDs result into alternated best performance between `sbdd≤` and `sbdd≥`. In any case, our new solving method with SROBDDs is clearly the best for all sets of instances (except for *sbacp_h1*), the improvement being even clearer in the hardest sets, *sbacp_h2_ml3* and *still life free*. It is worthy to notice that in *still life free*, `dico` gives two time outs while `yices` three unknowns (recall that the Yices MaxSMT solver is non exact and incomplete).

Finally, the `core` method has really bad performance for these kind of problems. This is probably due to the bad quality of the cores found during the solving process.

We also provide particular comparisons between the `sbdd≤` and `dico` methods, and between the `sbdd≤` and `yices` methods. Figure 5 shows the comparison for all the 140 (28×5) instances of the SBACP variants, where `sbdd≤` is able to solve 104 instances in less time than `yices` (left), and 100 instances in less time than `dico` (right).

Figure 6 shows the comparison for all the 30 (10×3) instances of the Still Life variants, where the `sbdd`$\leq$ method is able to 19 instances in less time than `yices`, and `yices` cannot find a solution in 5 instances (left), and the `sbdd`$\leq$ method is able to solve 23 instances in less time than `dico` (right).

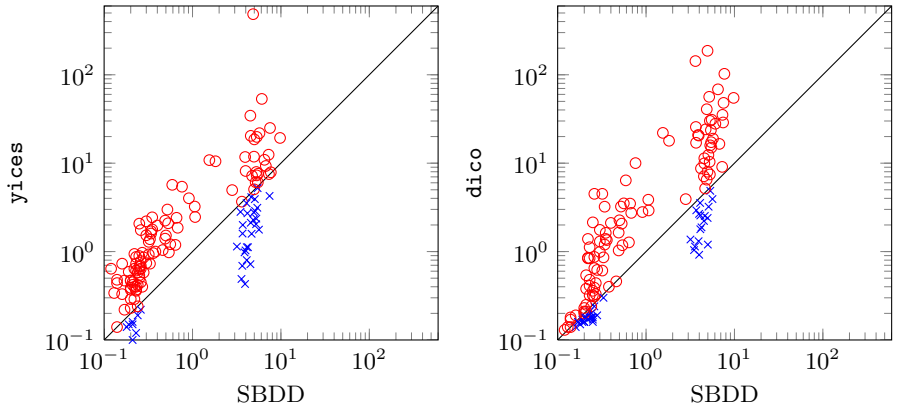

Fig. 5. Scatter plot of the solving times (in seconds) of the `sbdd`$\leq$ and `yices` methods (left), and of the `sbdd`$\leq$ and `dico` methods (right), for the 140 (28×5) instances of the SBACP. The `sbdd`$\leq$ method solves 104 instances in less time than `yices`, and 100 in less time than `dico`.

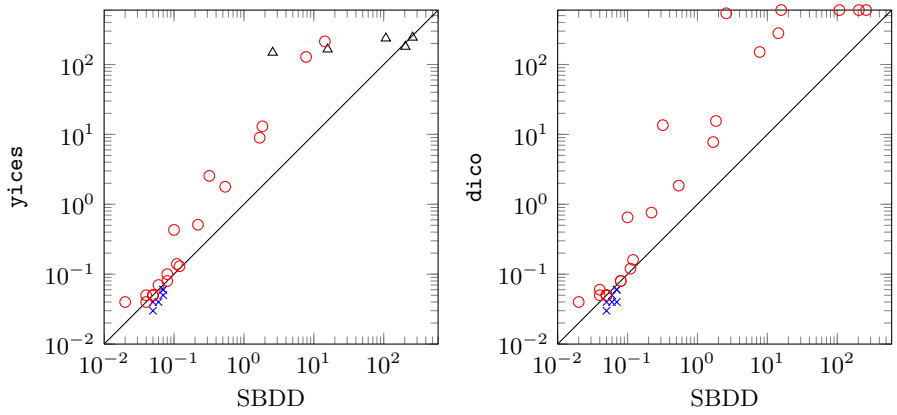

Fig. 6. Scatter plot of the solving times (in seconds) of the `sbdd`$\leq$ and `yices` methods (left), and of the `sbdd`$\leq$ and `dico` methods (right) for the 30 (10×3) instances of Still Life. The `sbdd`$\leq$ method solves 19 instances in less time than `yices`, which moreover cannot find a solution for 5 instances (marked with a triangle). The `sbdd`$\leq$ method solves solves 23 instances in less time than `dico`.

We have tried to figure out where is the gain in efficiency when using BDDs to deal with the objective function. Therefore, we have analyzed the time spent per iteration due to SROBDD construction, clauses assertion and satisfiability checking in the `sbdd`$\leq$ method, which in general is the best solving method.

Since Still Life instances have in average 30 variables in the PB constraint representing the bound of the objective function, all of them with coefficient 1, and we do binary search when seeking for the optimum, the number of iterations of the solving algorithm is at most 6 for these sets. A similar behavior appears for the three first sets of instances of the SBACP. For this reason, we only illustrate the time analysis for the *sbacp_h2_ml2* and *sbacp_h2_ml3* sets, in which the variables of the PB constraint have bigger coefficients, resulting in an average of 13 and 16 iteration steps respectively. Moreover, to see how useful is the usage of shared BDDs we have also computed the average percentage of node reuse in the construction of the new SROBDD per iteration.

Figure 7 has two plots showing this analysis for the *sbacp_h2_ml2* set (top) and the *sbacp_h2_ml3* set (bottom). As we can see, the majority of the time is spent in clauses assertion (this is the difference between the total time and the others). However, between iterations 5 and 9, in both sets, node reuse raises from 15% to almost 90% and therefore, the clauses assertion time strongly decreases. Naturally, at the same iterations we can appreciate a decrease on the total solving time, which in the last five iterations turns to be (nearly) just the time to check the satisfiability of the instance.

The node reuse behavior is similar for the rest of the instances of the SBACP and of Still Life, where the coefficients of the variables are always 1, making the BDDs very reusable. The only exception with respect to reuse is in the `sbdd`$\geq$ method, which has a high node reuse percentage in the *sbacp_h2_ml2* set (being fairly better than the `sbdd`$\leq$ method on the same set) and a low node reuse percentage in the *sbacp_h2_ml3* set. This can be appreciated in Table 4, where `sbdd`$\leq$ and `sbdd`$\geq$ swap their performance. We want to remark that, for the *sbacp_h2_ml3* set, the size of the first constructed ROBDD is very similar for both the `sbdd`$\leq$ and `sbdd`$\geq$ methods, with an average of 10395 nodes and an average of 11411 nodes, respectively. But the final SROBDD has in average 66128 nodes for `sbdd`$\leq$ and of 146791 nodes for `sbdd`$\geq$. Clearly, an important aspect to study in the future is how to find a good variable ordering for the objective function in order to get a highly reusable SROBDD.

6 Conclusions and future work

We have presented a new WCSP solving method, implemented in the `WSimply` system, based on using SROBDDs to generate SAT clauses representing the objective function. Although this is a preliminary work, we have shown that the new method clearly outperforms the previous `WSimply` solving methods on the two tested problems. In addition, we have shown how to boost the ROBDD generation for objective functions taking advantage of previously generated ROBDDs, more precisely constructing a SROBDD.

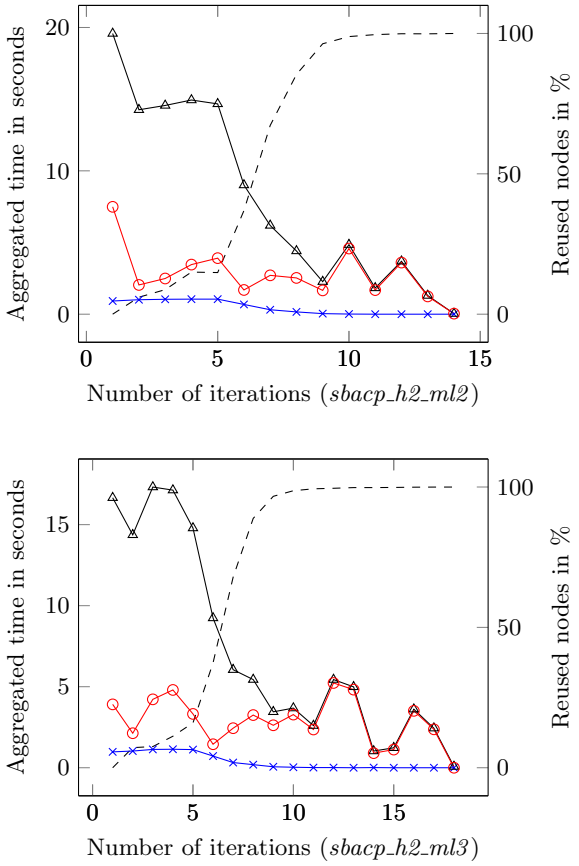

Fig. 7. Average of the percentage of SROBDD reused nodes per iteration (dashed line), aggregated SROBDD construction time (blue cross line), aggregated SROBDD construction time plus aggregated check time (red circle line) and aggregated total time (black triangles line) for the *sbacp_h2_ml2* set (top) and the *sbacp_h2_ml3* set (bottom).

As future work we want to study more deeply the efficiency of our method on other weighted constraint satisfaction problems and compare it with state-of-the-art WCSP solvers like *toulbar2* [3]. Also, as pointed out, an important 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 would like to check the efficiency of our new method on weighted MaxSAT and MaxSMT instances.

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. D. Allouche, S. de Givry, and T. Schiex. Toulbar2, an open source exact cost function network solver. Technical report, INRIA, 2010.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. M. Bofill, M. Palahí, J. Suy, and M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *Proceedings of the Eighth International Workshop on Constraint Modelling and Reformulation (ModRef 2009)*, pages 30–44, 2009.
10. M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.
11. E. Boros and P. L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.
12. 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.
13. 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.
14. 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.
15. 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.

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. 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.
20. J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-Consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.
21. 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.