An extension to Simply for solving Weighted Constraint Satisfaction Problems with Pseudo-Boolean Constraints

Miquel Bofill, Joan Espasa, Miquel Palahí, and Mateu Villaret

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

Abstract: Max-Simply is a high-level programming framework for modelling and solving weighted CSP. Max-Simply can also deal with meta-constraints, that is, constraints on constraints. The technology currently used to solve the generated problem instances is SMT. In this paper we present a variant of Max-Simply which is able to generate not only SMT instances but also pseudo-Boolean instances for certain modellings. Since there are problems that are more naturally encoded using pseudo-Boolean variables, the possibility of generating pseudo-Boolean instances can result in a more efficient and natural fit in some situations. We illustrate the expressiveness of the Max-Simply language by modelling some problems, and provide promising performance results on the corresponding generated pseudo-Boolean instances using state-of-the-art pseudo-Boolean solvers.

Keywords: Modelling languages, Pseudo-Boolean constraints, CSP, Weighted CSP.

1 Introduction

One of the challenges in constraint programming is to develop systems allowing the user to easily specify the problem in a high-level language and, at the same time, being able to efficiently solve it. Various approaches exist for solving constraint satisfaction problems (CSP) specified in a high-level language, such as ad hoc algorithms for some constructs, or the translation to lower level languages. Some years ago, translation was seen as a theoretical possibility but not really feasible. But there have been impressive developments in this area, making this approach not only feasible, but also competitive.

Following this direction, some high-level, solver-independent constraint modelling languages have been developed, such as MiniZinc [NSB+07], ESSENCE [FHJ+08] and `Simply` [BPSV09]. Those languages let the user express most CSPs easily and intuitively. There exist tools for translating from those high-level languages to lower level ones, some of them permitting a great deal of flexibility to the user. For example, the MiniZinc system lets the user specify which constraints wants to leave untranslated, so that an underlying solver or an ad hoc algorithm can deal with them in a better and more efficient way.

`Simply` was developed as a declarative programming system for easy modelling and solving of CSPs. Essentially, the system translates CSP instances (written in its own language) into satisfiability modulo theories (SMT) instances, which are then fed into an SMT solver. SMT instances generalize Boolean formulas by allowing the use of predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. For example, a

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 141-155, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

formula can contain clauses like $p \vee q \vee (x+2 \leqslant y) \vee (x > y+z)$, where p and q are Boolean variables and x, y and z are integer variables. `Max-Simply` [ABP+11] is an extension to `Simply`, allowing to model and solve weighted CSPs.

There exist many problems that are easily or naturally encoded using pseudo-Boolean variables, i.e., integers with a $\{0,1\}$ domain. For those kind of problems, very efficient pseudo-Boolean solvers exist. Here we propose a variant of `Max-Simply`, where we add the possibility to translate most of its high-level language constructs into low-level pseudo-Boolean constructs, allowing the user to solve his modelled weighted CSP problems using an even wider spectrum of solvers and technologies. This extension also provides support for some constraints on constraints, so called meta-constraints.

The rest of the paper is structured as follows. In Section 2, weighted CSP and pseudo-Boolean constraints are presented, including the pseudo-Boolean optimization problem and the weighted Boolean optimization problem. In Section 3, the language of `Simply` is briefly described, empathizing its features and how the translation to pseudo-Boolean constraints works. In Section 4, results of the experiments with two different problems are shown and analyzed. In Section 5, conclusions are drawn from the results so far and some future work is proposed.

2 Preliminaries

2.1 Weighted CSP

A *constraint satisfaction problem (CSP)* instance is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a set of domains containing the values the variables may take, and C is a set of constraints. Each constraint is defined as a relation over a subset of variables. A constraint or relation may be represented *intensionally*: in terms of an expression that defines the relationship that must hold amongst the assignments to the variables it constrains, or it may be represented *extensionally*: as explicitly enumerating the allowed assignments or the disallowed assignments. An *assignment* for a CSP instance $\langle X, D, C \rangle$ is a mapping that assigns to every variable in X an element that belongs to its domain, and a *solution* is an assignment that satisfies the constraints.

A *weighted CSP (WCSP)* is defined as a CSP, where each constraint has attached a *weight*, or falsification *cost* (typically a positive number), expressing the penalty for violating the constraint. An optimal solution to a WCSP instance is a complete assignment in which the sum of the weights of the violated constraints is minimal. We call *hard* those constraints whose associated cost is infinity, *soft* otherwise. For further formal definitions the reader can refer to [LS04].

2.2 Pseudo-Boolean constraints

A pseudo-Boolean constraint (PBC for short) is an inequality on a linear combination of 0-1 (Boolean) variables. PBCs take the form:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \oplus r \tag{1}$$

where $a_1, \ldots, a_n$ and r are integer constants, $x_1, \ldots, x_n$ are Boolean variables, and $\oplus$ is a relational operator in $\{<, >, \leqslant, \geqslant, =\}$.

So a system of PBCs is a system of linear inequalities of 0-1 variables, a paradigm extensively studied in Operations Research. PBCs lie on the border between satisfiability testing, constraint programming, and integer programming. With their expressivity, PBCs can be used to compactly describe many discrete electronic design automation problems, network layout problems, and much more.

The yearly pseudo-Boolean competition [RM12] works as a place to compare the efficiency of the current pseudo-Boolean solvers. The competition is divided in two branches, one for pseudo-Boolean optimization and one for weighted Boolean optimization. Each branch has its own file format. For solving pseudo-Boolean optimization problems, there are three big families of approaches: integer programming, pseudo-Boolean resolution and translation into MaxSAT. For further information, the reader can refer to [RM09].

2.2.1 Pseudo-Boolean optimization

Pseudo-Boolean optimization (PBO) is a natural extension of the Boolean satisfiability problem (SAT). A PBO instance is a set of PBC as defined in Equation 1, plus a linear objective function taking a similar form:

$$min: \ a_1x_1 + a_2x_2 + ... + a_nx_n \tag{2}$$

where $a_1,...,a_n$ are integer coefficients and $x_1,...,x_n$ are Boolean variables. The objective is to find a suitable assignment to the problem variables such that all the constraints are satisfied and the value of the objective function is minimized.

The first approaches into solving PBO instances were the generalization of the most effective techniques used in SAT solving, such as conflict-based learning, and conflict-directed backtracking [MM02, CK05]. The basic idea is, when a solution is found, to add a constraint such as only a lower value solution can be accepted. The solver finishes when a lower value solution cannot be found, thus proving its optimality.

Another common approach is the use a branch and bound algorithm, where the lower bounding procedure tries to estimate the value of the objective function. Direct translations from PBC to SAT have also been used effectively, especially in problem sets where the translation does not grow much larger than the original encoding.

The standard file format used in the pseudo-Boolean competition [RM12] is `opb`. This is a simple format, which is accepted by all the solvers. As a reference, a trivial example of this format can be seen in Figure 1. The first line of the file defines the number of variables and constraints in the problem instance.

```
* #variable= 2 #constraint= 1
**************************************
min: +1 x1 +1 x2;
+1 x1 +6 x2 >= -1;
```

Figure 1: `opb` example.

2.2.2 Weighted Boolean optimization

Weighted Boolean optimization (WBO) is another way of expressing a pseudo-Boolean optimization problem. This format aggregates and extends PBO and MaxSAT. It is composed of two types of constraints: hard and soft. Each soft constraint has an integer associated to it, representing its falsification cost. Given a WBO instance, an optimal solution is to find an assignment to the variables that satisfies all the hard constraints and minimizes the total cost of the unsatisfied soft constraints. Generalizations of MaxSAT and PBO algorithms are the first approaches that have been proposed to solve this kind of problems [MMP09].

The standard file format used in the pseudo-Boolean competition [RM12] for WBO instances is `wbo`. This is a simple format, accepted by most of the solvers. As a reference, a trivial example of this format can be seen in Figure 2. The first line of the file defines the number of variables and constraints, the number of soft constraints, the minimum and maximum cost of a single constraint and the total sum. The second line is defined as the sum of the costs of all soft constraints plus one.

```
* #variable= 3 #constraint= 4 #soft= 2 mincost= 2 maxcost= 3 sumcost= 5
soft: 6
+1 x1 +1 x2 +1 x3 >= 2;
+2 x1 +1 x2 +1 x3 >= 2;
[2] +1 x1 +1 x2 >= 1;
[3] +1 x1 +1 x3 >= 1;
```

Figure 2: `wbo` example.

3 Simply: extensions and variants

`Simply`[1] [BPSV09] is a programming system with a declarative language (also called `Simply`) for easy modelling and solving of CSPs, which uses an SMT solver as its core solving engine. Essentially, the system consists of a compiler from the `Simply` language into SMT, and a recovery module for translating the output generated by the SMT solver to the original problem format. Currently, `Simply` is integrated with the state-of-the-art SMT solver Yices [DM06], using its built-in API. However, it can be easily adapted to work with other SMT solvers, either using external files or any other API. Since most SMT solvers can only deal with decision problems, in the case of optimization `Simply` repeatedly calls the decision procedure performing binary search on the value of the objective function (treated as a constraint).

Recently, `Simply` has been extended to deal with weighted CSPs as well as with meta-constraints, in a new system called `Max-Simply` [ABP+11]. In this extension, the user can choose from the command line whether to solve the weighted CSP instance at hand (i) through successive calls to the decision procedure doing binary search on the corresponding objective function, or (ii) reformulating it into a weighted SMT instance and directly solving it with Yices (which has built-in support for weighted SMT), or (iii) applying the WPM1 algorithm [ABL09] on the generated weighted SMT instance.

[1] The tool is available from http://ima.udg.edu/recerca/LAP/simply/

In this paper we present a variant of `Max-Simply` which uses a pseudo-Boolean solver (PB solver for short) as its core solving engine. Although the pseudo-Boolean language is less expressive than that of SMT, solving CSP instances with a PB solver greatly simplifies the solving approach for optimization problems, due to the fact that PB solvers directly support optimization of objective functions. In the case of dealing with weighted CSPs, the architecture of the system can remain exactly the same as in `Max-Simply`, because most PB solvers support both optimization functions and weighted expressions. Consequently, the weighted instances can be directly reformulated either into PBO or WBO.

3.1 The language

The language of `Simply` is similar to other high-level modelling languages such as those of ESSENCE and MiniZinc. As it is usual, the model and the data of the problem instance can be stored in two different files. The decision variables in `Simply` can be finite domain integer variables and Boolean variables. For the pseudo-Boolean translation, only integer variables with the binary domain $\{0,1\}$ are currently considered.

Problems can be modelled by posting basic constraints, and also global and meta-constraints, that are decomposed into basic ones during the translation. The posting can be made either directly or by using generators such as `Forall` and `Exists`, and the `If-Then-Else` filter. The language also has the declarative facility of list comprehensions, allowing the user to generate parametrized lists of elements or constraints.

The basic constraints consist of arithmetic relational expressions built up with the usual operators $\leq$, $\geq$, $<$, $>$ and $=$ (only linear expressions are supported). These can be combined with other Boolean expressions using the Boolean operators `Or`, `And`, `Xor`, `Iff` and `Implies`, as for example in `b Implies 3*a[1]+4*a[2]+5*a[3] <= 8`.

The most representative global constraints supported by `Simply` are the following:

- `Sum(`l`,`n`)`. The sum of the elements in list l is n.
- `AtLeast(`l`,`v`,`n`)`. The number of occurrences of value v in list l is at least n.
- `AtMost(`l`,`v`,`n`)`. The number of occurrences of value v in list l is at most n.

In all the constraints, the elements of the lists, as well as the values v and n, must be linear integer expressions. In the present variant with pseudo-Boolean variables, the elements of the lists are restricted to expressions of the form $a*x$, where a is an integer constant and x is a pseudo-Boolean variable. Moreover, the last two global constraints have been modified in the following way (with the second argument removed):

- `AtLeast(`$[a_1*x_1,\ldots,a_n*x_n]$`,`n`)` if and only if $\sum_{i=1}^{n} a_i*x_i \geq n$.
- `AtMost(`$[a_1*x_1,\ldots,a_n*x_n]$`,`n`)` if and only if $\sum_{i=1}^{n} a_i*x_i \leq n$.

Example 1 *If we want to constraint the sum of the elements in a list of three elements to be equal to* 10*, we can post the basic constraint* `list[1] + list[2] + list[3] = 10;` *However, if we don't know, a priori, the length of the list because it is an integer constant value* `n` *of the instance data, we can post the following global constraint, where we use a list comprehension to create a list of unknown length:* `Sum([ list[i] | i in [1..n] ], 10);`

The generator

$$\texttt{Forall}(id\ \texttt{in}\ list)\ \{constraints\}$$

posts a conjunction of the generated inner constraints (either basic or global), while the generator

$$\texttt{Exists}(id\ \texttt{in}\ list)\ \{constraints\}$$

posts a disjunction. The language also has the filter

$$\texttt{If}(formula)\ \texttt{Then}\ \{constraints_1\}\ \texttt{Else}\ \{constraints_2\};$$

where the condition *formula* is evaluated in compilation time. If it evaluates to true then the $constraints_1$ are posted; otherwise the $constraints_2$ are posted.

Example 2 *The following code with two generators and one filter*

```
% Every element of the list must be between 0 and 10
Forall(i in [1..n]) {
  list[i] >= 0; list[i] =< 10;
};
% One of the elements ocurring in even positions
% of the list must be greater than or equal to 7
Exists(i in [1..n]) {
  If (n Mod 2 == 0) Then {
    list[i] >= 7;
  };
};
```

results into the following constraints, provided that the value of `n` *is known to be* `5` *at compile time:*

```
list[1] >= 0; list[1] =< 10;
list[2] >= 0; list[2] =< 10;
list[3] >= 0; list[3] =< 10;
list[4] >= 0; list[4] =< 10;
list[5] >= 0; list[5] =< 10;
(list[2] >= 7 Or list[4] >= 7);
```

The language also allows the use of list comprehensions as a `Forall` generator, i.e., list comprehensions can be used to post basic constraints.

Example 3 shows a `Simply` model of a toy instance of the Nurse scheduling problem (NSP).

Example 3 *Consider a simple instance of the NSP (details on the NSP are given in Section 4.3) with two shifts per day and two available nurses. Each shift must be covered with exactly one nurse. We also want to satisfy the preferences of the nurses. Say nurses 1 and 2 want to work both days on shift 1. This particular instance can be modelled with* `Simply` *as follows:*

```
Problem:nsp
   Data
         int nurses = 2; int days = 2; int shifts = 2;
   Domains
         Dom pseudo = [0..1];
   Variables
         IntVar nd[nurses, days, shifts] :: pseudo;
   Constraints
         % Exactly one nurse per day and shift
```

```
Forall(d in [1..days], st in [1..shifts]) {
   Sum([nd[n, d, st] | n in [1..nurses]], 1); };

% Exactly one shift per nurse and day
Forall(d in [1..days], n in [1..nurses]) {
   Sum([nd[n, d, st] | st in [1..shifts]], 1); };

% Preferences for nurse 1
nd[1,1,1] = 1; % work day 1 shift 1
nd[1,2,1] = 1; % work day 2 shift 1

% Preferences for nurse 2
nd[2,1,1] = 1; % work day 1 shift 1
nd[2,2,1] = 1; % work day 2 shift 1
```

We can identify four sections in the `Simply` *model: data, domains, variables and constraints. The data section allows us to define a particular instance of the problem. In the variables section we declare the decision variables of the problem, in this case a tridimensional array of pseudo-Boolean variables* `nd`*, where the size of the first dimension corresponds to the number of nurses, the size of the second dimension to the number of days and the size of the third dimension to the number of shifts. This array states whether each nurse works in each shift of each day. In the constraints section, we first introduce the cover constraints: for every day, we have to ensure that every shift is covered by the required number of nurses (one in this case). To post this constraint we generate the list of nurses working each day* `d` *and shift* `st` *with a list comprehension* `[nd[n, d, st] | n in [1..nurses]]`*, and we use the global constraint* `Sum` *to require the number of nurses working that day and shift to be exactly one. Similarly we post a second constraint for the number of shifts worked by a nurse in one day. Finally, we add the preference constraints of each nurse for each day, specifying which shift they prefer. As we can see this toy instance is unsatisfiable, since all nurses want to work the same shift.*

3.1.1 Soft constraints and meta-constraints

As we have explained at the beginning of this section, `Max-Simply` can deal with weighted CSPs, so the language implements *soft* constraints. For instance, in order to relax the nurse preference constraints of the previous example we only need to turn those hard constraints into soft constraints, by adding their weights as follows:

```
#A: nd[1,1,1] = 1 @{1};  #B: nd[1,2,1] = 1 @{1};
#C: nd[2,1,1] = 1 @{1};  #D: nd[2,2,1] = 1 @{1};
```

where the `@{1}` are the corresponding weights, and where `A`, `B`, `C` and `D` are labels that may be used later on to refer to the corresponding soft constraints (for example, in a meta-constraint).

We can go further at the specification level by allowing the user to express more complex preferences. In [PRB00], a set of constraints on soft constraints, called *meta-constraints*, is introduced. Meta-constraints can be very helpful in the modelling process, as they allow the user to abstract to a higher level. With them we can limit the degree of violation of soft constraints, state certain homogeneity in the amount of violation of disjoint groups of constraints, etc. In [ABP+11] all these meta-constraints are implemented by translation into SMT. Unfortunately, with pseudo-Booleans this is much harder since we do not have integer decision variables.

Therefore, in the present work we have only implemented two meta-constraints, which allow to bound the amount of violation (i.e., the aggregated cost of the unsatisfied constraints) of a list of soft constraints:

- `maxCost(`ll`,`n`)` / `minCost(`ll`,`n`)`. Constraints the maximum/minimum aggregated cost of the unsatisfied constraints of each list of soft constraints in ll to be n.

We remark that ll is a list of lists of soft constraint labels. For instance, following the previous example we could constraint the amount of violation of nurse preferences to be 1. Notice that, this way, we would obtain more fair solutions.

```
MetaConstraints
  maxCost([[A,B],[C,D]],1);
```

3.2 Translation into pseudo-Booleans

For obtaining a pseudo-Boolean instance from a `Simply` model, the first two steps of the translation are the same as for generating an SMT instance: first, the meta-constraints are reformulated into a conjunction of constraints; next, the generators and the list comprehensions are unfolded, so that a conjunction of basic or global of constraints is obtained.

The first difference is with respect to linear arithmetic expressions, which are straightforwardly reformulated into pseudo-Boolean constraints. The translation of Boolean combinations (`Or`, `Implies`, ...) of basic constraints requires reification. This technique is also required to deal with weights when translating to the `pbo` format, and is described in the next subsection.

The translation of the `Sum`, `AtLeast` and `AtMost` global constraints is straightforward from its definition in Subsection 3.1. The translation of the meta-constraint `maxCost(`$[l_1,\ldots,l_m]$`,`n`)` results into a conjunction of `AtMost` global constraints, one for each list l_i of soft constraint labels. Each labelled soft constraint is reified into an auxiliary pseudo-Boolean variable, which is multiplied by the cost of the soft constraint, and those are the arguments given to the `AtMost` global constraints. The `minCost` meta-constraint is translated similarly but using the `AtLeast` global constraint. At the current stage of development, meta-constraints are still not implemented for the `wbo` format.

3.3 Reification

As explained in Section 2, pseudo-Boolean constraints are inequalities on linear combinations of 0-1 variables. Several types of constraints cannot be directly translated to pseudo-Boolean form, for instance the ones containing logical operators (such as `Or`, `Implies`, ...). To be able to express these kind of constraints we use reification, which means to reflect the constraint satisfaction into a 0-1 variable. Then, e.g., `a Or b` can be translated into $i_a + i_b \geq 1$, where i_a and i_b are the 0-1 variables corresponding to the Boolean variables `a` and `b`, respectively.

In addition, since soft constraints are not directly supported by the `pbo` format we use reification to manage their violation costs. To do that, each soft constraint is reified and a reverse reification variable is created using the equation: $reif\_var + reif\_var_{rev} = 1$ (note that in this reverse reification variable we are reflecting the falsification of the constraint). Then the reverse reification variable multiplied by the violation cost is added to the minimization function.

Reification is done according to the BigM method, as described in [Wil99]. In Example 4 we give an example.

Example 4 *Assume we want to reify a (possibly labelled) constraint* $\sum_j a_j x_j \leqslant b$*, being* δ *the corresponding reification variable. The relation between the constraint and* δ *can be modelled as follows:*

- $\delta = 1 \rightarrow \sum_j a_j x_j \leqslant b$ *is translated into* $\sum_j a_j x_j + M\delta \leqslant M + b$*, where* M *is an upper bound for the expression* $\sum_j a_j x_j - b$*.*
- $\delta = 0 \rightarrow \sum_j a_j x_j \not\leqslant b$*, i.e.,* $\delta = 0 \rightarrow \sum_j a_j x_j > b$*. Since none of the two standard pseudo-Boolean formats supports the* $>$ *operator, we need to rewrite the expression* $\sum_j a_j x_j > b$ *as* $\sum_j a_j x_j \geqslant b + \varepsilon$*, where* ε *can be taken as 1 as we are dealing with integers. The final resulting constraint is then* $\sum_j a_j x_j - (m - \varepsilon)\delta \geqslant b + \varepsilon$*, where* m *is a lower bound for the expression* $\sum_j a_j x_j - b$*.*

In the case that the reified constraint is soft, since pseudo-Boolean problems are minimization problems, it can be seen that the implication with $\delta = 0$ is unnecessary. However, our experimental results have shown it beneficial.

4 Experiments

Two WCSP have been chosen for testing the performance of the new system along its development: Soft BACP, a variant of the Balanced Academic Curriculum Problem, and the well known Nurse Scheduling Problem. We have chosen this two problems since they can be naturally encoded in a suitable pseudo-Boolean form. All experiments have been performed on a 2.8GHz 12GB Intel Core i7 system running GNU/Linux with kernel 2.6.35-32 (64 bits).

4.1 Solvers

The solvers used in the experiments, classified by families, are the following.

- Integer Programming based.

  SCIP (scip-2.1.1.linux.x86) - A mixed integer programming solver.

- Pseudo-Boolean resolution based.

  Sat4j (sat4j-pb-v20110329) - A Boolean satisfaction and optimization library for Java. With PBC, it uses a cutting planes approach or pseudo-Boolean resolution [Sat12].

  Clasp (clasp-2.0.6-st-x86-linux) - A conflict-driven answer set solver.

  Bsolo (beta version 3.2 - 23/05/2010) - A solver with cardinality constraint learning.

- Translation to MaxSAT or SMT based.

  PWBO (pwbo2.0) - Also known as the parallel wbo solver. Since the other considered solvers are single-threaded, we have not used its parallel feature. When pbwo is run with one thread it uses a unsat-based core algorithm [MML11].

x	c_1	c_2	c_3
p_1	1	0	0
p_2	0	1	0
p_3	0	0	1

x_aux	c_1	c_2	c_3
p_1	0	0	0
p_2	1	0	0
p_3	1	1	0

Table 1: SBACP prerequisites example.

PB/CT (pbct0.1.2) - Translates the problem into SAT modulo the theory of costs and uses the Open-SMT solver.

4.2 Soft BACP

The first problem we have considered is a relaxed version of the Balanced Academic Curriculum Problem (BACP, prob030 at http://www.csplib.org) where the prerequisite constraints are turned into soft constraints. Note that in this case it is possible to violate a prerequisite constraint between two courses but then, in order to reduce the pedagogic impact of the violation, we introduce a new hard constraint, the corequisite constraint, enforcing both courses to be assigned to the same period. We call this problem the *Soft Balanced Academic Curriculum Problem* (SBACP). The goal of the SBACP is to assign a period to every course, minimizing the total amount of prerequisite constraint violations, and satisfying the constraints on the number of credits and courses per period, and the corequisite constraints.

In Figure 3, we propose a modelling of the SBACP using `Max-Simply`. In order to obtain instances of the SBACP, we have over-constrained the BACP instances from the MiniZinc [NSB+07] repository, by reducing to four the number of periods, and proportionally adapting the bounds on the workload and number of courses of each period. With this reduction on the number of periods, we have been able to turn into unsatisfiable all these instances, hence there are no solutions with zero cost. In the model, `prerequisites[i,1]` denotes a course which is a prerequisite of the course denoted by `prerequisites[i,2]` for each `i`.[2]

In Table 1, we give a small example to illustrate the meaning of the two matrices of variables `x` and `x_aux` used in the model. We consider an instance with three courses and three periods. Matrix `x` shows in what period a course is made. Matrix `x_aux` shows when a course is already done, that is, it is filled with ones for the periods following the one in which the course is done; this also can be seen as the accumulated version of matrix `x`. This duality lets expressing prerequisites easily. Note that the first period row in matrix `x_aux` always contains zeros.

In Figure 4, we show how with the `maxCost` meta-constraint we can restrict the violations of prerequisites for each course to a number `n`. Note that, previously, we must associate the label `pre[i]` to the prerequisite constraints.

In Table 2, we show the performance results for the SBACP. We have tested the pseudo-Boolean version of `Max-Simply` with the different back-ends described in Section 4.1, along with the SMT version of `Max-Simply` using Yices, which is able to solve weighted SMT instances. The first group of rows reflect the execution times without the `maxCost` meta-constraint, while the second and third groups show the execution times with the meta-constraints `maxCost([...],1)` and `maxCost([...],2)`, respectively. Although not shown in the

[2] It is worth noting that in some constraints the $\geq$ operator could be replaced by the $=$ operator, but using $\geq$ results in a simpler formula when reifying; we have observed that this speeds up execution.

```
Problem:sbacp
  Data
    int n_courses; int n_periods; int n_prerequisites;
    int load_per_period_lb;       int load_per_period_ub;
    int courses_per_period_lb;    int courses_per_period_ub;
    int loads[n_courses];         int prerequisites[n_prerequisites, 2];

  Domains
    Dom pseudo = [0..1];

  Variables
    IntVar x[n_periods, n_courses]::pseudo;
    IntVar x_aux[n_periods, n_courses]::pseudo;

  Constraints
    % each course is done only once
    Forall(c in [1..n_courses]) { Sum([x[p,c] | p in [1..n_periods]], 1); };

    Forall(p in [1..n_periods]) {
    % for each period the number of courses is within bounds
      AtLeast([x[p,c] | c in [1..n_courses]], courses_per_period_lb);
      AtMost ([x[p,c] | c in [1..n_courses]], courses_per_period_ub);
    % for each period the course load is within bounds
      AtLeast([ loads[c] * x[p,c] | c in [1..n_courses]], load_per_period_lb);
      AtMost ([ loads[c] * x[p,c] | c in [1..n_courses]], load_per_period_ub); };

    % Prerequisites and Corequisites
    % if course c is done previously to period p then x_aux[p,c] = 1
    Forall(c in [1..n_courses]) {
      Forall(p in [1..n_periods]) {
        Sum([x[p2,c] | p2 in [1..p-1]], x_aux[p,c]); }; };

    % a course being a (soft) prerequisite of another cannot be done after the other
    Forall(i in [1..n_prerequisites]) {
      Forall(p in [1..n_periods-1]) {
        x_aux[p+1, prerequisites[i,1]] >= x[p, prerequisites[i,2]]; }; };

    % prerequisite violation has cost 1
    Forall(i in [1..n_prerequisites]) {
      Forall(p in [1..n_periods]) {
        (x_aux[p, prerequisites[i,1]] >= x[p, prerequisites[i,2]])@ {1}; };
    };
```

Figure 3: `Max-Simply` model for the SBACP.

```
    ...
    % prerequisite violation has cost 1
    Forall(i in [1..n_prerequisites]) {
      #pre[i]: Forall(p in [1..n_periods]) {
                (x_aux[p, prerequisites[i,1]] >= x[p, prerequisites[i,2]])@ {1}; }; };

    MetaConstraints
      maxCost([ [ pre[np] | np in [1..n_prerequisites], prerequisites[np,2] = c ]
       | c in [1..n_courses] ], n);
```

Figure 4: Meta-constraint for the SBACP problem.

		Max-Simply (PB)							Max-Simply (SMT)
		pwbo		Sat4j	clasp	bsolo	PB/CT	SCIP	Yices
		PBO	WBO						
without	Total	31.49	48.19	44.48	23.12	70.86	53.80	33.25	125.96
	Mean	1.50	2.41	1.59	0.83	2.53	2.15	1.19	4.50
	Median	0.05	0.04	0.99	0.32	0.91	1.14	1.10	1.61
	# t.o.	7	8	0	0	0	3	0	0
maxC 1	Total	5.37	*	14.37	0.64	0.85	8.00	6.60	8.29
	Mean	0.19	*	0.51	0.02	0.03	0.29	0.24	0.30
	Median	0.04	*	0.50	0.02	0.02	0.23	0.16	0.28
	# t.o.	0	*	0	0	0	0	0	0
maxC 2	Total	88.84	*	35.05	9.21	19.27	56.72	28.34	47.34
	Mean	6.35	*	1.25	0.33	0.69	2.03	1.01	1.69
	Median	0.07	*	0.98	0.29	0.35	1.39	0.72	1.24
	# t.o.	14	*	0	0	0	0	0	0

Table 2: SBACP solving times with a timeout of 60s. *: for now, meta-constrains are only available for the `pbo` format.

table, it is worth noting that in the case of using the constraint `maxCost([...],1)`, the problem becomes quite more constrained, resulting into 14 unsatisfiable instances out of the 28 instances. However, 9 of the 14 solutions of the remaining instances are improved in terms of fairness, since we have reduced the difference between the less violated course and the most violated course. Despite the fairness improvement, the mean cost of the solutions is penalized with an increment of 2.56 points. In the case of `maxCost([...],2)`, the problem becomes only slightly more constrained, transforming only one of the 28 instances in unsatisfiable. In this case, the number of improved instances, in terms of fairness, is only 2, with an increment in the mean cost of 1 point in the first case and of 2 points in the second case.

4.3 Nurse Scheduling Problem

The Nurse Scheduling Problem (NSP) is a classical constraint programming problem [WHFP95]. Each nurse has his/her preferences on which shifts wants to work. Conventionally a nurse can work in three shifts: day shift, night shift and late night shift. There is a required minimum of personnel for each shift (the shift cover). The problem is described as finding a schedule that both respects the hard constraints and maximizes the nurse satisfaction by fulfilling their wishes. In Figure 5 we propose a modelling of the NSP using `Max-Simply`, where we have the shift covers and the number of working days as hard constraints, and the nurse preferences as soft constraints. Notice that here, contrarily to the case of the SBACP, we have constraints with parametrized weights.

We have taken the instances from the NSPLib. The NSPLib is a repository of thousands of NSP instances, grouped in different sets and generated using different complexity indicators: size of the problem (number of nurses, days or shift types), shifts coverage (distributions over the number of needed nurses) and nurse preferences (distributions of the preferences over the shifts and days). Details can be found in [VM07]. In order to reduce the number of instances to work with, we have focused on a particular set: the N25 set, which contains 7920 instances. The N25 set has the following settings: 25 nurses, 7 days, 4 shift types (where the 4th is the free shift), a certain number of nurses required for every shift of each day and a value between 1 and 4

	Max-Simply (PB)			Max-Simply (SMT)
	pwbo		SCIP	Yices
	PBO	WBO		
# solved	5095	5064	7290	4088
# t.o.	2195	2226	0	3202
Mean	1.63	1.77	0.10	4.49
Median	0.38	0.68	0.09	1.52

Table 3: Results on the NSP with soft constraints on nurse preferences.

(from most desirable to less desirable) for each nurse, shift and day. The nurses are required to work exactly 5 days.

```
Problem:nsp
  Data
    int n_nurses; int n_days; int n_shift_types; int min_turns; int max_turns;
    int covers[n_days, n_shift_types]; int prefs[n_nurses, n_days, n_shift_types];

  Domains
    Dom pseudo = [0..1];

  Variables
    IntVar x[n_nurses, n_days, n_shift_types]::pseudo;

  Constraints
    % each nurse can only work one shift per day
    Forall(n in [1..n_nurses], d in [1..n_days]) {
      Sum([x[n,d,s] | s in [1..n_shift_types]], 1); };

    % minimum covers
    Forall(d in [1..n_days], s in [1..n_shift_types-1]) {
      AtLeast([x[n,d,s] | n in [1..n_nurses]], covers[d,s]); };

    % the number of free days is within bounds
    Forall(n in [1..n_nurses]) {
      If (min_turns = max_turns) Then {
        Sum([x[n,d,n_shift_types] | d in [1..n_days]], n_days - min_turns);
      } Else {
        AtLeast([x[n,d,n_shift_types] | d in [1..n_days]], n_days - max_turns);
        AtMost ([x[n,d,n_shift_types] | d in [1..n_days]], n_days - min_turns); }; };

    % penalize each failed preference
    Forall(n in [1..n_nurses], d in [1..n_days], s in [1..n_shift_types]) {
      (x[n,d,s] = 0) @ {prefs[n,d,s]}; };
```

Figure 5: Max-Simply model for the NSP.

Table 3 shows the summarized results for all the 7290 instances of the N25 set of the NSP. The solvers shown are the pseudo-Boolean version of Max-Simply with pwbo2.0 and SCIP, along with the SMT version of Max-Simply with Yices, as the rest of solvers have not been able to solve any instance within the time limit of 60 seconds. In these executions, SCIP excels, being able to solve all instances with a mean time which is an order of magnitude lower than the one of the second fastest solver.

5 Conclusions and future work

We have presented a work in progress variant of `Max-Simply` that takes advantage of the latest developments and improvements in the pseudo-Boolean solvers, giving the user a greater range of options when solving its constraint problems. Experiments presented here have shown that the pseudo-Boolean approximation can sometimes outperform SMT approximations. Even inside the pseudo-Boolean solvers ecosystem, depending on the solving strategies the results can vary greatly. As problems are easily encoded with a high-level language, the new system can also be seen as a tool to generate pseudo-Boolean benchmarks. It can allow developers and researchers to compare their different solvers and approaches. As for the future work, there are many interesting paths to follow:

- In problems where most decision variables are pseudo-Boolean and only a few are integer, it would be interesting to allow `Simply` to encode those integer variables using pseudo-Boolean variables. To this purpose, we should study the efficiency of different encodings.
- At the moment there are two meta-constraints implemented: `minCost` and `maxCost`. New meta-constraints should be introduced, enriching the language. This could be easier if integer variables were allowed.
- Another more ambitious point of interest should be developing and integrating a pseudo-Boolean theory solver in an SMT solver, in order to take advantage of both a DPLL search algorithm and specialized algorithms for the pseudo-Booleans.
- For now, the code generation phase is a bit slow, as no premature optimizations were made during the development phase of the tool. Now that we have obtained some promising results, code generation speed can and should be improved.

Acknowledgements: We thank Josep Suy for his helpful comments and suggestions.

Bibliography

[ABL09] C. Ansótegui, M. Bonet, J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *SAT 2009*. LNCS 5584, pp. 427–440. Springer, 2009.

[ABP+11] C. Ansótegui, M. Bofill, M. Palahi, J. Suy, M. Villaret. A Proposal for Solving Weighted CSPs with SMT. In *ModRef 2011*. Proceedings of the Tenth International Workshop on Constraint Modelling and Reformulation, pp. 5–19. 2011.

[BPSV09] M. Bofill, M. Palahi, J. Suy, M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *ModRef 2009*. Proceedings of the Eigth International Workshop on Constraint Modelling and Reformulation, pp. 30–44. 2009.

[CK05] D. Chai, A. Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(3):305–317, 2005.

[DM06] B. Dutertre, L. de Moura. The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[FHJ+08] A. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints* 13(3):268–306, 2008.

[LS04] J. Larrosa, T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence* 159(1):1–26, 2004.

[MM02] V. Manquinho, J. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21(5):505–516, 2002.

[MML11] R. Martins, V. Manquinho, I. Lynce. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *ICTAI 2011*. Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, pp. 313–320. 2011.

[MMP09] V. Manquinho, J. Marques-Silva, J. Planes. Algorithms for Weighted Boolean Optimization. In *SAT 2009*. LNCS, pp. 495–508. Springer, 2009.

[NSB+07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP 2007*. LNCS 4741, pp. 529–543. 2007.

[PRB00] T. Petit, J. C. Regin, C. Bessiere. Meta-constraints on violations for over constrained problems. In *ICTAI 2000*. Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence, pp. 358–365. 2000.

[RM09] O. Roussel, V. Manquinho. Pseudo-Boolean and cardinality constraints. *Handbook of Satisfiability* 185:695–733, 2009.

[RM12] O. Roussel, V. Manquinho. Pseudo-Boolean Competition. http://www.cril.univ-artois.fr/PB11/, 2011 (accessed 02/05/2012).

[Sat12] SAT4J Pseudo-Boolean Competition implementation. http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf, 2009 (accessed 02/05/2012).

[VM07] M. Vanhoucke, B. Maenhout. NSPLib - a nurse scheduling problem library: a tool to evaluate (meta-) heuristic procedures. In *Operational research for health policy: making better decisions*. Proceedings of the 31st annual meeting of the working group on operations research applied to health services, pp. 151–165. 2007.

[WHFP95] G. Weil, K. Heus, P. Francois, M. Poujade. Constraint programming for nurse scheduling. *Engineering in Medicine and Biology Magazine, IEEE* 14(4):417–422, 1995.

[Wil99] H. P. Williams. *Model Building in Mathematical Programming*. Wiley, 4th edition, 1999.