Tools for modeling and solving search problems *

Deborah East [a] Mikhail Iakhiaev [b]
Artur Mikitiuk [c] Mirosław Truszczyński [d]

[a] *Department of Computer Science, Texas State University - San Marcos, San Marcos, TX 78666, USA.*

[b] *Department of Computer Science, University of Texas at Austin, Austin, TX 78712-0233, USA.*

[c] *Department of Computer Science, University of Texas at Tyler, Tyler, TX 75799, USA.*

[d] *Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA*

In this paper, we describe a language PS^{pb} to model search problems that are specified in terms of *boolean combinations* of pseudo-boolean constraints. We then describe software tools that allow one to use SAT and SAT(PB) solvers to compute solutions to instances of search problems represented in the language PS^{pb}.

Keywords: SAT, pseudo-boolean constraints, grounder

1. Introduction

Recent research demonstrated that programs computing models of theories in propositional languages, or *SAT solvers*, can be used to find solutions to a broad class of search problems. Due to advances in the performance of SAT solvers, that approach is now becoming practical for an ever expanding range of applications. Despite this computational potential of SAT solvers, the tools to support and facilitate their use are lagging behind. They are *ad hoc* and problem specific. Typically, to use a SAT solver to compute solutions of a search problem Π, a programmer develops a *specialized* program P_Π that generates, for each instance of Π, a corresponding instance of the SAT problem. That approach makes it difficult to reason about problem constraints as they are "hard-wired" in the program P_Π. It also hinders the use of SAT solvers as *general-purpose* computational mechanism. Different search problems (even equivalent but different representations of the same problem) require an associated specialized "translator" program.

Our objective is to provide support for a more general and systematic approach to solving search problems by SAT solvers consisting of the following three main steps:

1. **Modeling**: First, a programmer represents constraints of a search problem Π as a theory P_Π in a high-level constraint language $\mathcal{L}$, and constructs a description D_I of a specific instance I of the problem Π.
2. **Compiling**: Next, a specialized program compiles the pair (D_I, P_Π) into a theory $T_{\Pi,I}$ in some propositional target logic $\mathcal{L}_{tgt}$ so that solutions to problem Π for an instance I correspond to models of $T_{\Pi,I}$ and can be recovered from them quickly. The compiling program depends only of $\mathcal{L}$ and $\mathcal{L}_{tgt}$ and not on individual search problems.
3. **Solving**: Finally, a *solver* for the logic $\mathcal{L}_{tgt}$, that is, a program computing models of theories in $\mathcal{L}_{tgt}$, finds a model of $T_{\Pi,I}$ (and so, also a solution to Π for I), or determines that no models (solutions) exist.

In this paper we are concerned with the first two steps of that process. Our objective is to design and implement tools that will support the use of *existing* solvers (and their extensions) in the third one.

For the modeling language $\mathcal{L}$ in step 1, we propose extension of the language $PS+$ [9]. Theories in $PS+$ consist of formulas that generalize propositional schemata (clauses of predicate logic). Our language extends $PS+$ with the ability to model explicitly *pseudo-boolean constraints*. We re-

*This is an extended version of a paper presented at the Third International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, Toronto, Canada, September 2004.

AI Communications
ISSN 0921-7126, IOS Press. All rights reserved

fer to it as the language of *propositional schemata with pseudo-boolean constraints* and denote it with PS^{pb}.

As $PS+$, the language PS^{pb} separates the specification of problem constraints from the description of particular data instances. Users model constraints as clauses in the language PS^{pb} and represent problem instances as collections of ground atoms in the language PS^{pb}.

Given a theory consisting of a program (clauses) and data (ground atoms), we interpret PS^{pb} clauses as *propositional schemata*. They represent sets of their propositional instantiations with respect to all constants explicitly mentioned in the data component of the theory. These instantiations are formulas in the *propositional logic with pseudo-boolean constraints*, PL^{pb}, which is a direct extension of propositional logic with propositional pseudo-boolean constraints. In this work, we use the logic PL^{pb} as the logic $\mathcal{L}_{tgt}$ in step 2.

We consider models of the ground instantiation of a PS^{pb} theory T (models in the logic PL^{pb}) to be models of T. Consequently, models of a PS^{pb} theory T can be computed by grounding T and then by finding models of the ground instantiation of T. This latter task can be accomplished by off-the-shelf SAT solvers and solvers of pseudo-boolean constraints (sometimes after some additional simple transformation of the ground theory).

The logic PL^{pb} extends the logic of pseudo-boolean constraints and, consequently, also the standard propositional logic. Thus, if the original PS^{pb} theory contains *no* pseudo-boolean constraints, grounding it yields a propositional theory and off-the-shelf SAT solvers can be used in step 3. If the original PS^{pb} theory contains no boolean combinations of pseudo-boolean constraints (each pseudo-boolean constraint forms a unit clause), grounding it generates a collection of propositional pseudo-boolean constraints that can be solved by SAT(PB) solvers such as *PBS* [1], *Pueblo* [18] and *minisat+* [10] (we refer to [16] for more references). For general PL^{pb} theories there are simple transformations that convert them into one of the forms discussed above.

In the paper, we briefly discuss the language PS^{pb}, which serves as a programming front-end, a modeling language, and the logic PL^{pb}, a target logic for ground PS^{pb} theories, a bridge to a solver. The main objective of this paper is to describe a *grounder* program, *psgrnd*, that automates the task of grounding. Our implementation of *psgrnd* extends the scope and improves the performance of an earlier prototype [9]. It is available at `http://www.cs.uky.edu/psgrnd/`.

We believe that using solvers dealing directly with boolean combinations of pseudo-boolean constraints may be more effective than rewriting a ground theory to eliminate such combinations (which usually makes the theory larger) and using SAT or SAT(PB) solvers on the rewritten theory.

To further facilitate the use of existing and future solver programs, we designed a DIMACS-like default output format for *psgrnd* and scripts to translate it, whenever appropriate, into the DIMACS format [17] and to input formats of several SAT(PB) solvers. Our main objective in designing the *psgrnd* output format is to establish it as the standard input format for future PL^{pb} solvers. Adopting it will make the use of the PS^{pb} modeling language and the *psgrnd* grounding program more straightforward and direct.

2. Logic PL^{pb}

We start with the description of the logic PL^{pb} as it makes it easier later to introduce the logic PS^{pb} as a generalization of PL^{pb} to the first-order language. The logic PL^{pb} extends the logic introduced in [9], in which only cardinality constraints (called there cardinality atoms) were allowed. All basic ideas behind the logic proposed in [9] lift literally to our setting and we follow closely the presentation given there.

A *pseudo-boolean constraint* (*pb-constraint*, for short) is an expression of the form

$$A = l\{p_1 = w_1, \ldots, p_k = w_k\}u,$$

where $p_1, \ldots, p_k$ are atoms from some fixed set of atoms At, $w_1, \ldots, w_k$ are integer *weights* associated with atoms $p_1, \ldots, p_k$, respectively, and l and u, where $l \leq u$, are integers called the *bounds*. One of the bounds, but not both, may be missing. Informally, this constraint is true if and only if the sum of weights w_i for atoms p_i which are true is comprised between l and u. If all weights are equal to 1, we often drop them from the notation and write A as $l\{p_1, \ldots, p_k\}u$ (in this case, we refer to these pb-constraints as *cardinality* constraints).

Our notation for pb-constraints follows the one used in *lparse* [19]. A different notation is used in

[2], where a pb-constraint is seen as an integer-programming constraint $l \leq \sum_{i=1}^{k} p_i w_i \leq u$, with p_is regarded as integer variables with the domain $\{0, 1\}$ rather than propositional variables [3, 2]. Pseudo-boolean constraints generalize propositional clauses and often make modeling of application problems more direct. Consequently, the problem of computing assignments satisfying sets of pseudo-boolean constraints has received much attention in the SAT community and resulted in several effective SAT(PB) solvers [16].

Pb-constraints can be combined into more complex constraints. A *pb-clause* is an expression of the form

$$C = \quad A_1, \ldots, A_s \rightarrow B_1 | \ldots | B_t,$$

where all A_i and B_i are (propositional) atoms or pb-constraints[1]. We note that we write ',' and '|' for the conjunction and the disjunction operators, respectively. Since natural language constraints are typically given as implications, we also use the "implication" notation for clauses in our approach. Similar convention is used in logic programming.

A set of atoms $M \subseteq At$ *satisfies* a pb-constraint $A = l\{p_1 = w_1, \ldots, p_k = w_k\}u$, denoted by $M \models A$, if

$$l \leq \sum_{\{i : p_i \in M\}} w_i \leq u,$$

with an obvious extension to the case when one of l and u is missing. A set of atoms M *satisfies* a pb-clause C, written as $M \models C$, if M satisfies at least one atom or pb-constraint B_j or does not satisfy at least one atom or pb-constraint A_i.

Since the logic PL^{pb} extends the logic of pseudo-boolean constraints, it also extends the clausal propositional logic. But, there is a more direct relationship. Indeed, every CNF clause

$$\neg a_1 \vee \ldots \vee \neg a_m \vee b_1 \vee \ldots \vee b_n$$

has an equivalent representation in the logic PL^{pb} as an implication (we recall that in the logic PL^{pb} ',' and '|' stand for '$\wedge$' and '$\vee$', respectively)

$$a_1, \ldots, a_m \rightarrow b_1 | \ldots | b_n.$$

In other words, the clausal propositional logic is simply a fragment of the logic PL^{pb}.

To illustrate the use of the logic PL^{pb} we will consider the *dominating-set* problem. Let $G = (V, E)$ be an undirected graph. A set $X \subseteq V$ is a *dominating set* in G if every vertex of G is in X or is adjacent to a vertex in X. Given an undirected graph G, a weight function w assigning integers to vertices, and an integer k, the problem is to find a dominating set X such that the total weight of vertices in X is at most k. To model the problem, we define a PL^{pb} theory $D(G, w, k)$ as follows:

$$\{p_v = w(v) : v \in V\}k$$
$$p_v \mid 1\{p_w : \{v, w\} \in E\}, \text{ for every } v \in V.$$

Since all the atoms in the pb-constraint in the second rule have weights equal to 1, according to the convention we mentioned earlier, we dropped weights from the notation. It is clear that a set of vertices X is a dominating set with the total weight at most k if and only if the set of atoms $\{p_v : v \in X\}$ is a model of $D(G, w, k)$. This example shows one of the advantages of the logic PL^{pb} over formalisms which do not allow boolean combinations of pb-constraints. There is a *direct* mapping of the constraint defining a dominating set to a clause of the logic PL^{pb}.

3. Language PS^{pb}

The main question we deal with in this paper is how to specify theories such as $D(G, w, k)$ so that problem specifications are described concisely by means of finite programs that are independent of particular data instances. The language we propose extends the language $PS+$ [9] by direct ways to model general pb-constraints (only cardinality constraints were addressed in [9]).

Understanding the language PS^{pb} and, especially, its semantics is essential for the design of the grounder. Thus, we will now outline main ideas behind it. We follow closely the way in which the language $PS+$ was described in [9] and refer to that paper for more details.

3.1. Language $PS+$

Let us consider the following clauses in the language of predicate logic (to stay consistent with the notation used earlier, we write ',' for '$\wedge$' and '|' for '$\vee$'; $\perp$ stands for a contradictory formula):

[1] Most current SAT(PB) solvers do not accept such complex constraints. They require that each pseudo-boolean constraint represents a "unit" clause. Two exceptions are *aspps* [9,8] (which, however, accepts only clauses built of cardinality constraints) and *wsatcc* [14,15].

$r(X)|g(X)|b(X)$
$edge(X,Y), r(X), r(Y) \rightarrow \bot$
$edge(X,Y), g(X), g(Y) \rightarrow \bot$
$edge(X,Y), b(X), b(Y) \rightarrow \bot$

These clauses can be viewed as a specification of the graph 3-coloring problem. Indeed, given a set of ground atoms

$$D_G = \{vtx(x) : x \in V\} \cup \{edge(x,y) : \{x,y\} \in E\},$$

specifying a graph, and some typing information stating that X and Y can only be substituted with constants in the extension of the relation symbol vtx, these clauses offer a concise notation for the following collection of their ground instantiations

$r(x)|g(x)|b(x)$, for every $x \in V$
$edge(x,y), r(x), r(y) \rightarrow \bot$, for every $x, y \in V$
$edge(x,y), g(x), g(y) \rightarrow \bot$, for every $x, y \in V$
$edge(x,y), b(x), b(y) \rightarrow \bot$, for every $x, y \in V$.

Under an additional assumption that the truth values of the atoms of the form $vtx(x)$ and $edge(x,y)$ are fully determined by the set D_G (those in D_G are true and those not in D_G are false), these clauses can be rewritten as

$r(x)|g(x)|b(x)$, for every $x \in V$
$r(x), r(y) \rightarrow \bot$, for every x and y such that $edge(x,y) \in D_G$
$g(x), g(y) \rightarrow \bot$, for every x and y such that $edge(x,y) \in D_G$
$b(x), b(y) \rightarrow \bot$, for every x and y such that $edge(x,y) \in D_G$.

This is a version of a familiar propositional encoding of the problem of 3-coloring of the graph G. It has the property that there is a one-to-one correspondence between models of that theory and 3-colorings of G.

In [9], we described a formalism to model search problems in a way generalizing the graph 3-coloring example. The language of that formalism is essentially a fragment of the standard language of first-order logic with the *signature* (R_d, R_p, C, V), where R_d and R_p are disjoint sets of relation symbols, and C and V are sets of constant and variable symbols, respectively[2]. However, we distinguish two types of relation symbols depending on whether they belong to R_d or R_p. We use relation symbols in R_d to represent data instances of search problems and call them *data* relation symbols. We call relation symbols in R_p *program* relation symbols. In the graph-coloring example, $R_d = \{vtx, edge\}$ and $R_p = \{r, b, g\}$.

The definitions of *terms*, *atoms*, and *ground* terms and atoms are standard. A clause in the language is an expression of the form

$$C = \quad A_1, \ldots, A_s \rightarrow B_1| \ldots |B_t,$$

where all A_i and B_i are atoms. As before, we write clauses as implications rather than disjunctions. We recall that we use ',' and '|' in place of '$\wedge$' and '$\vee$'.

Given a search problem, we model its particular computational instance by a *data-program* pair (D, P), where D is a set of ground atoms built of data relation symbols and P is a *program*, that is, a set of clauses specifying problem constraints. Programs also contain *typing* declarations. The statements with the keyword *pred* define program relation symbols, specify their arities and the types of the arguments. The statements with the keyword *var* specify types of variables that appear in the program. The types are given by unary data relation symbols. All typing declarations have global scope in a given program. For the graph-coloring example the typing declarations are of the form:

pred $r(vtx)$
pred $g(vtx)$
pred $b(vtx)$
var vtx X, Y

Let (D, P) be a data-program pair. Typing specifies for each variable its domain (as the extension in D of the data predicate defining its type), which in turn, for every clause in P determines its set of ground instances. Since we assume, as in the graph-coloring example above, that the extensions of data relation symbols are fully specified by the input data instance D, we simplify them away from these ground instances. The union of all such ground instances of all clauses in P is a propositional theory, whose models provide the semantics for the data-program pair (D, P).

To sum up, the formalism proposed in [9] is a language for modeling constraints of search problems as programs, that is, sets of declarations and clauses. The data-program pair, consisting of a

[2]The language also contains *predefined* relation symbols ==, <=, <, >= and > for the equality and arithmetic comparisons, and *predefined* function symbols such as +, −, ∗ and / to represent arithmetic operations. For these symbols, we *always* assume their standard interpretation. Consequently, we drop them from the signature.

program and a specification of a particular data instance as a set of ground atoms, represents a propositional theory — a collection of ground instantiations of clauses in the program with respect to constants specified in the data. Models of this propositional theory correspond to problem solutions.

3.2. Syntax of PS^{pb}

We will now describe the language PS^{pb} that extends the formalism from [9] by means to model *arbitrary* pb-constraints and their combinations. A grammar providing a precise definition of the syntax is available at `http://www.cs.uky.edu/psgrnd/`.

The signature of the language of the logic PS^{pb} is (R_d, R_p, C, V, W), where R_d, R_p, C and V are as before, and where W is a set of weight-function symbols. The only terms in the language are arithmetic expressions built of constant and variable symbols in $C \cup V$. Clauses in the logic PS^{pb} are built of pb-constraints, which we view as constraints on *sets* of atoms. Thus, a set of atoms is a basic concept in our language.

A *weighted-set definition* is an expression S of the form $p(t) = w(t')[L] : d_1(s_1) : \ldots : d_m(s_m)$, where p is a program relation symbol, w is a weight-function symbol, L is a list of variables, d_i, $1 \leq i \leq m$, are data or predefined relation symbols, and t, t' and s_i, $1 \leq i \leq m$, are tuples of terms such that all variables appearing in t' appear also in t. We call the expression $d_1(s_1) : \ldots : d_m(s_m)$ the *condition* of S. Intuitively, S stands for the set of all expressions $p(t) = w(t')$, for which all conditions $d_i(s_i)$, $1 \leq i \leq m$, hold. All variables in L must also appear in t. It is possible for L to be empty. In such case, we omit the list from the notation altogether. It is also possible that $m = 0$. In such case we omit the symbol ':'. If the weight function is a constant function equal everywhere to 1, we omit it from the notation and write $p(t)[L] : d_1(s_1) : \ldots : d_m(s_m)$.

Each weighted-set definition S is a "template" for sets of weighted atoms. Variables appearing in S that are not in the list L appearing in S are *free*. Grounding them (replacing with constants) yields different instances of the template S. Variables that appear in the list L of S are *bound* in S. Grounding bound variables in an instance of S, yields elements of the set defined by that instance. We formalize these intuitions below, when we formally define the notion of grounding.

A *first-order pb-constraint* (or pb-constraint, if there is no ambiguity) is any expression

$$l\{S_1; \ldots; S_k\}u,$$

where l and u are terms and $S_1, \ldots, S_k$ are *weighted-set definitions*. Intuitively, the meaning of a predicate pb-constraint $l\{S_1; \ldots; S_k\}u$ is that the total weight of all atoms that are true in the union of the sets specified by the set definitions $S_1, \ldots, S_k$ is at least l and no more than u (we will shortly make this intuition precise). A *(first-order) pb-clause* is an expression of the form

$$C = \quad A_1, \ldots, A_s \to B_1 | \ldots | B_t,$$

where all A_i and B_i are (first-order) atoms or pb-constraints.

We represent an instance of a search problem as a set D of ground atoms specifying the extensions of data relation symbols in R_d. The set D also contains expressions of the form $w(c_1, \ldots, c_k) = u$, where w is a k-ary weight-function symbol in W, $c_1, \ldots, c_k$ are constants that occur in ground atoms listed in D, and u is an integer. These expressions define a weight function w (we assume that argument tuples not listed explicitly do not belong to the domain of w). We call such sets D *data sets*.

We represent the constraints specifying the search problem itself by a collection of pb-clauses, and typing and declaration statements. The latter have the same format as that we introduced earlier when discussing the graph-coloring example. We call such collections *programs*.

We call a pair (D, P), where D is a data set and P is a program, a *data-program pair*.

3.3. Semantics of PS^{pb}

We will now define *models* of data-program pairs. The definition is based on the interpretation of clauses as *propositional schemata*, that is, as shorthands for sets of *ground* propositional clauses.

Let (D, P) be a data-program pair. First, we consider a weighted-set definition S of the form $p(t) = w(t')[L] : d_1(s_1) : \ldots : d_m(s_m)$ appearing in a pb-clause of a data-program pair (D, P). Let ϑ be a ground substitution whose domain contains all free variables in S and does not contain any variables that are bound in S (we note that the sets of free and bound variables are disjoint). By $S\vartheta$ we denote the set of expressions of the form $p(t\vartheta\vartheta') = v$, where

1. ϑ' is a ground substitution with the domain consisting of all variables that are bound in S such that for every i, $1 \leq i \leq m$, $d_i(s_i\vartheta\vartheta')$ holds (we note that term tuples $s_i\vartheta\vartheta'$ are ground and, since data relation symbols are fully specified by a data-program pair, this latter condition can be verified efficiently)
2. v is an integer to which the weight expression $w(t'\vartheta\vartheta')$ evaluates. That is, we evaluate the term tuple $t'\vartheta\vartheta'$, which is ground (performing arithmetic operations, if necessary) and look up in D the value of the function w for the resulting tuple of the ground arguments (we recall that weight functions are fully defined in the data component D). The whole expression is undefined if w is undefined for the term $t'\vartheta\vartheta'$.

To specify the meaning of a pb-clause C occurring in the program P of the data-program pair (D, P), we ground C and replace it with a set of propositional pb-clauses. Let us consider a pb-constraint $A = l\{S_1; \ldots; S_k\}u$ appearing in C. We start by renaming all bound variables by new unique names different from any other variable name in the clause (the renaming does not change the meaning of any of the set definitions in A). In this way, the sets of bound and free variables in C are disjoint. Let ϑ be a ground substitution whose domain contains all free variables and none of the bound ones. We define $A\vartheta$ as follows:

1. $A\vartheta = \perp$, if $l\vartheta$ or $u\vartheta$ are not integers
2. $A\vartheta = \perp$, if for some $1 \leq i \leq k$, $S_i\vartheta$ is undefined
3. $A\vartheta = \perp$ if for some $1 \leq i < j \leq k$ and $w \neq w'$, there are expressions $a = w$ and $a = w'$, in $S_i\vartheta$ and $S_j\vartheta$, respectively.
4. $A\vartheta = l\vartheta\{S_1\vartheta \cup \ldots \cup S_k\vartheta\}u\vartheta$, otherwise. In this case, $l\vartheta$ and $u\vartheta$ are integer constants, and $S_1\vartheta \cup \ldots \cup S_k\vartheta$ is a set of ground expressions of the form $a = w$.

It is clear that $A\vartheta$ is a propositional pb-constraint. Applying ϑ to all atoms in C produces a propositional pb-clause $C\vartheta$. We now define $grnd(D, P)$ to consist of all propositional pb-clauses of the form $C\vartheta$, where C is a pb-clause in P and ϑ is a ground substitution that contains in its domain all free variables in C and none of C's bound variables. We define a set M of ground atoms to be a *model* of (D, P) if it is a model of $grnd(D, P)$.

Given a mapping assigning to an instance I of a search problem Π a data set D_I, we say that a program P *solves* Π if for every instance I, solutions to Π for I correspond to models of the data-program pair (D_I, P).

4. Psgrnd

Models of a data-program pair (D, P) are models of a PL^{pb} theory $grnd(D, P)$. Consequently, they can be computed by solvers for the logic PL^{pb}. To facilitate use of such solvers in computing solutions to search problems represented in the language PS^{pb} as data-program pairs, we implemented a program *psgrnd*. Given a data-program pair (D, P), *psgrnd* outputs its grounding $grnd(D, P)$[3].

4.1. Description

This section describes the program *psgrnd*. Our implementation is a major enhancement of a prototype program described in [9]. It is based on a formal grammar for the language of the logic PS^{pb}. Both the grammar and the *psgrnd* program are available at `http://www.cs.uky.edu/psgrnd/`. To generate the parser source code, we processed this grammar by the Bison utility [12]. We wrote the code of *psgrnd* in C++ and compiled it both under UNIX and Windows environment. We used gcc 3.3 compiler for UNIX and Microsoft Visual Studio 6.0 compiler for Windows XP. The main improvements with respect to the earlier version of the grounder program include the capability to process weight constraints (when all weights are positive) and an option to execute the complete one-atom lookahead (we will explain this concept later) to reduce the size of the propositional theory produced by the grounder.

The output from the grounder program when executed on a data-program pair (D, P) is a set of ground instantiations of pb-clauses in P computed with respect to data specification in D. It also includes the set of atoms whose logical value was determined by the grounder and those that

[3] There are some minor differences between the grounding as we described it and what we implemented in *psgrnd*. Namely, in the cases (1) - (3) of the definition of $A\vartheta$, the grounder produces an error message and terminates with failure.

the grounder determined to be irrelevant (they may assume any logical value in any model of the pair (D, P)). Some of the most important options of the program include output in human-readable form, output in the DIMACS and PBS format (for input data-program pairs without pb-constraints and for input data-program pairs with all pb-constraints forming unit clauses, respectively), disabling lookahead, and disabling propagation.

This implementation of the grounder is often much faster than the previous implementation due to carefully designed memory management module. Moreover, information about data predicates is reorganized before parsing clauses by replacing linked lists of possible values with sorted arrays. This allows us to use binary search during parsing and grounding clauses. The benefits of using binary search exceed additional cost associated with sorting data.

Furthermore, when generating ground instances of clauses of PS^{pb} programs, we ground variables in the order of minimal cost. This eliminates early those variable instantiations which do not lead to valid ground clauses.

Another improvement in efficiency comes from storing names and other multi-character symbols appearing in user input in a symbol table and representing every name in data structures by its corresponding number. As a result, we can often replace inefficient string comparisons with comparisons between two integers.

Previous versions of the program used a binary search tree to store names of ground atoms during grounding process. The current version uses a balanced binary tree data structure for this purpose. In many applications, the number of ground atoms is large (measured in millions). We believe that the use of a balanced binary tree is needed to maintain the set of ground atoms and to search it efficiently.

During the grounding process, when a new clause, say r, is added to the collection of ground clauses and r contains only one atom, *psgrnd* triggers *unit propagation*. It follows a general format of unit propagation for propositional CNF theories [6], modified to handle the case of pb-constraints. When there are no more truth values to propagate and no contradictory pb-clause (empty antecedent and empty consequent) was derived, the unit propagation process terminates. If a contradictory clause was derived, the theory is inconsistent, the whole grounding process terminates and a single contradictory clause is returned.

When grounding is complete, *psgrnd* has an option to perform a complete one-atom lookahead (called also failed literal rule in literature on SAT solvers [13]). For every ground atom with an undetermined truth value, the grounder assigns to this atom a truth value (first true, then false) and tentatively propagates it. If this tentative propagation results in an inconsistency, the atom must have the opposite truth value and permanent propagation is later performed on this atom with its assigned value. If a tentative propagation terminates without a conflict, the program restores the theory to its previous state. A complete one-atom lookahead is costly (runs in time $O(n^2)$, where n is the number of ground atoms) but sometimes results in a much smaller theory.

4.2. Experimental results

We will now briefly discuss the performance of *psgrnd*. The key question is how fast it can process data-program pairs leading to large ground theories with millions of atoms and clauses. We tested *psgrnd* (version of June 4, 2004, used with the lookahead option) on data-program pairs (D_n, P), where P is a program consisting of two simple clauses, each with four variables over the same domain, and D_n specifies this single domain as the integer range $(1..n)$.

```
pred p(num,num,num,num).
pred q(num,num,num,num).
var num A,B,C,D.

p(A,B,C,D)|q(A,B,C,D).
p(A,B,C,D),q(A,B,C,D)-> .
```

The ground theory for a data-program pair (D_n, P) consists of $2n^4$ ground atoms and $2n^4$ ground clauses. Thus, even for small values of n, the ground theory is large, which poses a challenge for grounding programs (for instance, $grnd(D_{50}, P)$ has over 12 million atoms and 12 million clauses). We compared *psgrnd* with its earlier and restricted version [9], which we call here *old-psgrnd* (version of June 4, 2003). We also compared *psgrnd* with *lparse* (version 1.0.13), a state-of-the-art program for grounding DATALOG$^\neg$ programs. For that comparison, we replaced the program P with an equivalent DATALOG$^\neg$ program of a similar structure to P.

```
num(1..n).

p(A,B,C,D) :- num(A;B;C;D), not q(A,B,C,D).
q(A,B,C,D) :- num(A;B;C;D), not p(A,B,C,D).
```

The results for $n = 10$, 20, 30, 40, and 50 are shown in Table 1.

Table 1
Hand-made program, large ground theories, CPU time in seconds

grounder	*psgrnd*	*old-psgrnd*	*lparse*
$n = 10$	0.09	23.68	0.15
$n = 20$	1.8	509.30	2.51
$n = 30$	10.59	2995.87	12.87
$n = 40$	36.43	-	40.48
$n = 50$	94.76	-	102.88

The results indicate that *psgrnd* is at least two orders of magnitude faster than its earlier version, which timed out (the limit was set at 3600 sec of CPU time). It is also slightly faster than *lparse*. We conducted these experiments on a machine with 3.2GHz Intel Pentium processor, 1Gb memory and running Slackware 9.0 Linux kernel 2.4.25.

We also experimented with programs encoding more typical search problems. We present here results for the graph coloring problem for certain simplex graphs with $5n$ vertices and $11n-4$ edges, for $n = 5000$, 10000, 15000 and 20000 (the ground theory in this last case has 300000 atoms and 1059998 clauses). For *psgrnd* we used the following program

```
pred r(vtx).
pred g(vtx).
pred b(vtx).
var vtx X,Y.

r(X) | g(X) | b(X).
edge(X,Y), r(X), r(Y) -> .
edge(X,Y), g(X), g(Y) -> .
edge(X,Y), b(X), b(Y) -> .
```

The equivalent program for *lparse* is

```
r(X) :- vtx(X), not g(X), not b(X).
b(X) :- vtx(X), not r(X), not g(X).
g(X) :- vtx(X), not r(X), not b(X).

:- edge(X,Y), r(X), r(Y).
:- edge(X,Y), g(X), g(Y).
:- edge(X,Y), b(X), b(Y).
```

The results in Table 2 show that *psgrnd* significantly outperforms *old-psgrnd* and is faster than *lparse* (the latter program causes segmentation fault when run for $n = 20000$).

Table 2
Graph-coloring problem

grounder	*psgrnd*	*old-psgrnd*	*lparse*
$n = 5000$	1.14	> 1000	1.84
$n = 10000$	2.28	> 1000	3.74
$n = 15000$	3.43	> 1000	5.70
$n = 20000$	4.51	> 1000	seg fault

4.3. Program usage

The program *psgrnd* is invoked from a UNIX command line in the following way

$$psgrnd\ [-d\ dataFileList]\ -r\ ruleFile$$
$$[-c\ constantList]\ [output]\ [flags]$$

The list *dataFileList* is optional and consists of one or more data files:

$$dataFile1\ dataFile2\ \ldots\ dataFileN$$

Altogether they specify the "data" component of the problem description. The file *ruleFile* contains the "program" component of the problem description.

The *output* is of the form

$$[-o\ outputForSolver]$$
$$[-m\ humanReadableOutput]$$

The file *outputForSolver* contains the file that can be processed by a solver and the file *humanReadableOutput* contains the same output but in human-readable form. If *-o* option is not specified, the default output file is *out.aspps*. If *-m* option is not specified, no human-readable output is generated.

An optional *constantList* is a list of value assignments for symbolic constants appearing in data files or in a rule file. Assuming the names of constants are $n1, \ldots, nM$ and the corresponding values are $v1, \ldots, vM$, the *constantList* has the following form:

$$n1 = v1\ \ n2 = v2\ \ldots\ nM = vM$$

Flags $-d$, $-r$, and $-c$ may be dropped, but in this case they should be dropped altogether and the data files should always precede the rule file. *Psgrnd* has several options specifying output formats and processing instructions. We refer to `http://www.cs.uky.edu/psgrnd/` for details.

The program *psgrnd* can currently produce its output in several formats. The default format is designed for the case of *general* theories in the logic PL^{pb}. It is accepted by solvers *aspps* [9,8] (if all pb-constraints in the ground theory are cardinality constraints) and *wsatcc* [14,15].

To support the use of existing SAT solvers and SAT(PB) solvers, we developed simple scripts translating the *psgrnd* format into the DIMACS format [17] (if the ground theory contains no pb-constraints) and into input formats of SAT(PB) solvers [16] (if all clauses containing pb-constraints are unit clauses). The scripts generating DIMACS and PBS [1] formats are already integrated as options of *psgrnd* ($-dimacs$ or $-D$, and $-pbs$ or $-P$, respectively). The other scripts will be integrated in the future.

We will now briefly describe the default output format for *psgrnd*. Our objectives are to establish that format as the standard input format for solvers for the logic PL^{pb} and to support the use of PS^{pb} as a programming front-end for existing and future SAT(PB) solvers.

The default output format for *psgrnd* has the following properties. The output file starts with a header line

$p\ \ n\_of\_prop\_atoms\ \ n\_of\_pb\_constr\ \ n\_of\_rules$

The next n_of_rules lines represent clauses of the ground program. Propositional atoms are represented by positive integers. Pseudo-boolean constraints are represented in the form

$$\texttt{[}\ l\ u\ a_1\ \ = w_1\ \ldots\ a_k\ \ = w_k\ \texttt{]}$$

where l and u are integers representing the lower and the upper bound, respectively, $a_1, \ldots, a_k$ are integers representing propositional atoms, and $w_1, \ldots, w_k$ are integer positive weights assigned to these atoms. Cardinality constraints (special case of pseudo-boolean constraints where all weights are equal to 1) are represented as expressions

$$\{\ l\ u\ a_1\ \ldots\ a_k\ \}$$

with the meaning of l, u, and $a_1, \ldots, a_k$ as above. When the lower bound is missing, the number of 0 is output in its place. When the upper bound is missing, the sum of weights of the propositional atoms in the pb-constraint is output in its place. Clauses of the ground program contain first a set of atoms and pb-constraints in the body (possibly empty), next a comma and, finally, a set of atoms and pb-constraints (possibly empty) in the head. Within sets of atoms and pb-constraints in the body and in the head, propositional atoms are followed by cardinality constraints, and then by other pb-constraints.

After lines representing clauses, there are lines containing description of propositional atoms in the form

$$c\ atom\_number\ atom\_name$$

Atoms appearing in the rules are assigned positive numbers starting with 1. Atoms determined during grounding to be true are assigned number 0. Atoms that the grounder determined to be irrelevant (they may assume any logical value in any model) are assigned number -1. In general, if a line starts with a 'c' character, it is considered a comment, not a part of the ground program.

5. Examples

We will now consider several search problems and describe PS^{pb} programs that solve them. All relevant data relation symbols and weight function symbols appear in typing statements and in clauses of programs solving search problems. Therefore, we will only describe programs and omit a detailed discussion of the data representation schemata.

We start with the dominating set problem, which we discussed earlier. The program solving the problem consists of the following statements and clauses:

```
pred in(vtx).
var vtx X,Y.

{in(X)=w(X)[X]}k.
in(X) | 1{in(Y)[Y]: edge(X,Y)}.
```

The first clause captures the constraint that the sum of weights of selected vertices is at most k. The second clause represents the constraint defining a dominating set: every vertex belongs to the set or at least one of its neighbors does[4]. This program (given a data set) grounds to the PL^{pb} theory we described in Section 2. We note that following the grammar of the syntax of the language PS^{pb} accepted by *psgrnd*, we complete each declaration and each clause with a period '.'. We also note that the constant k appearing in the first clause and

[4]We assume here that every edge $\{x, y\}$ of an input graph is represented both as $edge(x, y)$ and $edge(y, x)$.

specifying the bound on the total weight of a dominating set needs to be specified at the command line when calling *psgrnd*.

The next problem we discuss is the $n \times n$ magic square problem. To solve it we can use the following PS^{pb} program.

```
pred in(index,index,entry).
var index I,J.
var entry K.

1 {in(I,J,K)[K]} 1.
1 {in(I,J,K)[I,J]} 1.
n*(n*n+1)/2 {in(I,J,K)=w(K)[J,K]} n*(n*n+1)/2.
n*(n*n+1)/2 {in(I,J,K)=w(K)[I,K]} n*(n*n+1)/2.
n*(n*n+1)/2 {in(I,I,K)=w(K)[I,K]} n*(n*n+1)/2.
n*(n*n+1)/2 {in(I,n+1-I,K)=w(K)[I,K]} n*(n*n+1)/2.
```

The first two clauses are unit cardinality constraints that ensure that (1) there is exactly one value K (from the range $(1..n^2)$ defined in the data set) for every position (I, J) in the array, and that (2) every value K is placed in some position (I, J). The remaining four clauses describe the basic problem constraints that each row, column and two main diagonals have entries that sum up to the same value $n(n^2 + 1)/2$. As before, the constant n needs to be specified at the command line, when calling *psgrnd*.

Next, we consider the Schur problem: given integers k and n, find an assignment of $1, \ldots, n$ into k bins so that each bin is sum-free (if x and y are in a bin, $x + y$ is not). We can solve that problem with the following program.

```
pred in(num,bin).
var num M,N.
var bin B.

1{in(M,B)[B]}.
in(M,B), in(N,B), in(M+N,B) -> .
```

The first clause captures the requirement that each integer M (in the range specified in the data set), is assigned to some bin B. The second clause describes the Schur constraint.

Data-program pairs with these three programs ground to theories in the logic PL^{pb}. In the last two cases, these theories consist of pseudo-boolean constraints forming only unit clauses and SAT(PB) solvers can be used to compute their models (solutions to the corresponding instances of the search problems). Since PS^{pb} is a modeling language, the same constraint can be sometimes modeled in several ways. For example, PS^{pb} allows us to write the first clause as

```
in(M,B)[B].
```

The meaning of this clause is similar to the one used before. The difference is that it grounds to a collection of *propositional clauses* of the form $in(m, 1)| \ldots |in(m, k)$ rather than to propositional pb-constraints of the form $1\{in(m, 1), \ldots , in(m, k)\}$. Clearly, both types of ground expressions have the same semantics. However, now the corresponding data-program pairs ground to propositional CNF theories and *standard* SAT solvers can be used for computing solutions.

To better illustrate the capabilities of the language PS^{pb}, our final example concerns a more complex problem. Namely, we will present two encodings of the *15-puzzle problem*. In this problem integers $0, 1, \ldots , 15$ are placed in a 4×4 array, each number appearing precisely once. A move consists of swapping 0 with the content of a neighboring cell (neighboring cells must be in the same row or in the same column; sharing a "corner" is not enough). The goal is to find a sequence of moves that result in a configuration arranged in the row-major order ($0, 1, 2, 3$ in the first row, $4, 5, 6, 7$ in the second row, etc.). The first encoding does not use pb-constraints or cardinality constraints.

```
% Declare program predicates
pred in(time,pos,pos,entry).
pred move(time,pos,pos).

% Declare variables
var time T.
var pos U,V,X,Y.
var entry A,B.

% For each time T, define an assignment of numbers
% 0, 1, ..., 15 to 16 locations in the array

% Choose at least one number for entry (X,Y)
in(T,X,Y,A)[A].
% No entry (X,Y) contains two different numbers
in(T,X,Y,A), in(T,X,Y,B) -> A==B.
% Each number A appears in at least one entry
in(T,X,Y,A)[X,Y].

% At time T=0, computed assignment must coincide
% with the initial assignment (note: in the
% presence of other constraints, the second
% constraint is redundant)
in(0,X,Y,A) -> in0(X,Y,A).
in0(X,Y,A) -> in(0,X,Y,A).

% For each time T<t, select exactly one location
% neighboring with 0 for the swap

% Select at least one location for the swap
T<t -> move(T,X,Y)[X,Y].
% Select at most one location for the swap
T<t, move(T,X,Y), move(T,U,V) -> X==U.
T<t, move(T,X,Y), move(T,U,V) -> Y==V.
```

```
% The location selected must neighbor
% the location with 0
in(T,X,Y,0), move(T,U,V) ->
        abs(X-U)+abs(Y-V)==1.

% Do not undo the last move
in(T,X,Y,0), in(T+2,X,Y,0) ->.

% Assignment at time T+1 must correspond to the
% assignment at time T modulo swapped values

% Number in entry (X,Y) at time T+1 was in this
% entry at time T, or there was 0 at time T, or
% that entry was chosen for the swap
T<t, in(T+1,X,Y,A) ->
        in(T,X,Y,A) | in(T,X,Y,0) | move(T,X,Y).
% If at time T we chose (X,Y) to move 0,
% 0 is in that location at time T+1
T<t, move(T,X,Y) -> in(T+1,X,Y,0).

% Specify the goal configuration
% 0 1 2 3
% 4 5 6 7
% 8 9 10 11
% 12 13 14 15
in(t,1,1,0). in(t,1,2,1). in(t,1,3,2). in(t,1,4,3).
in(t,2,1,4). in(t,2,2,5). in(t,2,3,6). in(t,2,4,7).
in(t,3,1,8). in(t,3,2,9). in(t,3,3,10).in(t,3,4,11).
in(t,4,1,12).in(t,4,2,13).in(t,4,3,14).in(t,4,4,15).
```

For this program and a set of data representing the initial configuration, *psgrnd* can produce output in the DIMACS format and the output theory can be later processed by a SAT solver.

The second encoding of the 15-puzzle problem uses cardinality constraints. There are two differences only. They concern the way in which we model constraints defining an assignment of numbers in the array and determining the location for the swap. We only present encoding of these constraints below.

```
% For each time T, define an assignment of numbers
% 0, 1, ..., 15 to 16 locations in the array

% Each location contains exactly one number
1{in(T,X,Y,A)[A]}1.
% Each number is in exactly one locattion
1{in(T,X,Y,A)[X,Y]}1.

% For each time T<t, select exactly one location
% neighboring with 0 for the swap
T<t -> 1{move(T,X,Y)[X,Y]}1.
in(T,X,Y,0), move(T,U,V) ->
        abs(X-U)+abs(Y-V)==1.
```

For this theory, *psgrnd* cannot produce output in the DIMACS format. However, the output theory can be later processed by SAT(PB) solvers such as PBS.

We run *psgrnd* for both encodings of the 15-puzzle problem using the following set of data.

```
time(0..t).
pos(1..4).
entry(0..15).

in0(1,1,1).  in0(1,2,5).  in0(1,3,2).  in0(1,4,7).
in0(2,1,8).  in0(2,2,4).  in0(2,3,3).  in0(2,4,11).
in0(3,1,9).  in0(3,2,13). in0(3,3,15). in0(3,4,0).
in0(4,1,12). in0(4,2,10). in0(4,3,6).  in0(4,4,14).
```

Since this initial configuration requires 19 moves to reach the goal, we specified the constant t in the command line as 19. For the first encoding, we obtained a ground theory in the DIMACS format with 45103 rules. The size of the output file was 834739 bytes. For the second encoding, we obtained a theory in the PBS format. The output consisted of two files. One of them had 3133 rules in the DIMACS format and the file size was 107246 bytes. The second file had 125096 bytes and contained 956 pb-constraints. Thus, using pb-constraints can significantly reduce the size of the output theory both in terms of the number of ground rules and disk space used.

6. Conclusions, related and future work

We defined a language PS^{pb} for modeling search problems specified by *boolean combinations* of pseudo-boolean constraints. Based on a formal grammar of PS^{pb}, we designed and implemented a program *psgrnd* that converts specifications of search problems in the language PS^{pb} into theories in the logic PL^{pb}. The theories generated by *psgrnd* can be output in formats accepted by current SAT solvers for CNF theories and by solvers for more general PL^{pb} theories (at present, under some syntactic restrictions). In this way, the *psgrnd* program facilitates the use of SAT and SAT(PB) solvers as a general computational mechanism for finding solutions to search problems modeled in the language PS^{pb}.

Currently *psgrnd* allows only positive weights. There is a script translating theories with negative weights into a form accepted by *psgrnd*. This script will be in the future integrated with our grounder to make it more convenient for the user.

Other researchers also studied the problem of modeling propositional and pseudo-boolean constraints. The closest to our work are the language

ESO (the existential fragment of the second-order logic) [5,4] and the language *QPROP*, extending the language of propositional logic with finite quantification [11]. Each of these languages is a restricted version of our language in that they do not admit pseudo-boolean constraints. A study of generalization of techniques for SAT to cardinality and pseudo-boolean constraints was recently published in [7].

As we mentioned at the beginning, representing search problems in the language PS^{pb} makes it possible to reason about them. Two important problems stemming from that possibility are: (1) to develop automated techniques to rewrite problem specifications to improve the performance of solvers on the corresponding ground theories produced by *psgrnd*, and (2) to design a class of solvers whose heuristics could be customized to a particular ground theory based on properties of its high level specification in the language PS^{pb}.

Acknowledgments

This work was supported by the NSF grants IIS-0097278 and IIS-0325063 and by Faculty Research Grant Program at UT Tyler.

The authors would like to thank the reviewers for their helpful suggestions.

References

[1] F.A. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pages 346 – 353, 2002.

[2] P. Barth. A Davis-Putnam based elimination algorithm for linear pseudo-boolean optimization. Technical report, Max-Planck-Institut für Informatik, 1995. MPI-I-95-2-003.

[3] B. Benhamou, L. Sais, and P. Siegel. Two proof procedures for a cardinality based language in propositional calculus. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS-1994)*, volume 775 of *LNCS*, pages 71–82. Springer, 1994.

[4] M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. In Alan M. Frisch, editor, *Proceedings of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 33–47, 2003. (In conjunction with CP-2003).

[5] M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. In *Proceedings of the European Symposium On Programming (ESOP-2001)*, volume 2028 of *LNAI*, pages 387–401. Springer, 2001.

[6] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[7] H.E. Dixon, M.L. Ginsberg, and A.J. Parkes. Generalizing Boolean Satisfiability I: Background and Survey of Existing Work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.

[8] D. East and M. Truszczyński. *Aspps solver*, version of 06/04/2003 (2003) `http://cs.engr.uky.edu/ai/aspps/`.

[9] D. East and M. Truszczyński. Predicate-calculus based logics for modeling and solving search problems. *ACM Transactions on Computational Logic*, 2006. To appear, available at `http://www.acm.org/tocl/accepted.html`.

[10] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2:1-26, 2006.

[11] M.L. Ginsberg and A.J. Parkes. Satisfiability algorithms and finite quantification. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning, (KR-2000)*, pages 690–701. Morgan Kaufmann, 2000.

[12] GNU Project - Free Software Foundation. Bison, 2004. `http://www.gnu.org/software/bison/bison.html`.

[13] C.M. Li and M. Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371. Morgan Kaufmann, 1997.

[14] L. Liu and M. Truszczyński. Local-search techniques in propositional logic extended with cardinality atoms. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP-2003*, volume 2833 of *LNCS*, pages 495–509. Springer, 2003.

[15] L. Liu and M. Truszczyński. *Wsatcc solver*, version of 7/29/2005 (2005) `http://cs.engr.uky.edu/ai/wsatcc/`.

[16] V. Manquinho and O. Roussel. The first evaluation of pseudo-boolean solvers (PB'05). *Journal on Satisfiability, Boolean Modeling and Computation* 2:103-143, 2006.

[17] Satisfiability Suggested Format Last revision: May 8, 1993 `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.tex`.

[18] H. Sheini and K. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 2:165-189, 2006.

[19] T. Syrjänen. *lparse*, a procedure for grounding domain restricted logic programs. `http://www.tcs.hut.fi/Software/smodels/lparse/`, 1999.