Counting Models in Integer Domains

António Morgado[1], Paulo Matos[1], Vasco Manquinho[1], and João Marques-Silva[2]

[1] IST/INESC-ID, Technical University of Lisbon, Portugal
{ajrm, pocm, vmm}@sat.inesc-id.pt
[2] School of Electronics and Computer Science, University of Southampton, UK
jpms@ecs.soton.ac.uk

Abstract. This paper addresses the problem of counting models in integer linear programming (ILP) using Boolean Satisfiability (SAT) techniques, and proposes two approaches to solve this problem. The first approach consists of encoding ILP instances into pseudo-Boolean (PB) instances. Moreover, the paper introduces a model counter for PB constraints, which can be used for counting models in PB as well as in ILP. A second alternative approach consists of encoding instances of ILP into instances of SAT. A two-step procedure is proposed, consisting of first mapping the ILP instance into PB constraints and then encoding the PB constraints into SAT. One key observation is that not all existing PB to SAT encodings can be used for counting models. The paper provides conditions for PB to SAT encodings that can be safely used for model counting, and proves that some of the existing encodings are safe for model counting while others are not. Finally, the paper provides experimental results, comparing the PB and SAT approaches, as well as existing alternative solutions.

1 Introduction

Besides its well-known theoretical relevancy, the problem of counting models of Boolean Satisfiability (SAT) formulas (#SAT) has a large number of key application areas [2,18]. Recent years have seen significant improvements in algorithms for #SAT, which include the utilization of well-known SAT techniques as well as the identification of connected components and component caching, but also variable lifting and blocking clauses [6,12,17,18]. Nevertheless, model counting is also extremely important in non-Boolean domains, including Integer Linear Programming (ILP) [5,11] and Linear Integer Arithmetic (LIA) [7,16]. This paper focus on ILP, but the techniques proposed can be extended to LIA.

Existing algorithms for counting models in ILP [5,11] are extremely sensitive to the number of variables in the problem formulation, being able to solve instances with a very small number of variables. Hence, in many practical applications, existing algorithms are ineffective.

This paper proposes two alternative solutions to counting models in ILP, by considering the utilization of SAT-based techniques. The first approach consists of encoding instances of ILP into instances of pseudo-Boolean (PB) constraints.

A. Biere and C.P. Gomes (Eds.): SAT 2006, LNCS 4121, pp. 410–423, 2006.

Moreover, the paper introduces a model counter for PB constraints, which can be used for counting models in PB as well as in ILP. A second alternative approach consists of encoding instances of ILP into instances of SAT. A two-step procedure is proposed, consisting of first mapping the ILP instance into PB constraints and then encoding the PB constraints into SAT. One key concern is that not all existing PB to SAT encodings can be used for counting models. The paper provides conditions for encodings that can be safely used for model counting, and proves that some of the existing PB to SAT encodings are safe for model counting. Finally, the paper provides experimental results, comparing the PB and the SAT approaches, as well as existing alternative solutions. The results provide interesting insights into the problem of counting models in ILP. First, the PB counter, albeit a preliminary prototype, is competitive with SAT counters, which integrate more sophisticated techniques including the identification of connected components and component caching. Second, the very effective SAT-techniques used in Cachet [18] may not scale for integer domains.

The paper is organized as follows. Section 2 presents the notation used throughout the paper. Afterwards, the paper addresses the encoding of ILP into PB constraints, and describes a model counter for PB formulations. Section 5 details the second approach to model counting in ILP, based on encoding ILP into SAT. This section proves that some existing encodings will yield correct results, whereas others can overestimate the number of integer models. Section 6 compares the two approaches and also evaluates an alternative solution [11]. Section 7 surveys related work, and the paper concludes in Section 8.

2 Definitions

An *Integer Linear Programming* (ILP) problem with n variables and m constraints can be defined as follows [14]:

$$\begin{array}{l} \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \\ x_j, a_{ij}, b_i \in \mathbb{Z} \\ j \in \{1, \ldots, n\}, i \in \{1, \ldots, m\} \end{array} \tag{1}$$

where a_{ij} denote the coefficients of the problem variables x_j in the set of m linear constraints. Note that all ILP problem instances can be rewritten as defined in (1). ILP problem instances are usually defined with a cost function to minimize or maximize. However, for the purpose of counting the number of models, the cost function is irrelevant. Hence, for simplification, we focus solely on the constraints.

An ILP problem instance is said to be a *Linear Pseudo-Boolean* (PB) problem instance (also known as 0-1 ILP) if the variable domain of the ILP variables is Boolean. In this case, all constraints correspond to pseudo-Boolean constraints. A particular type of pseudo-Boolean constraints are propositional clauses and a problem instance where all constraints are propositional clauses is an instance of the *Propositional Satisfiability* (SAT) problem.

In a propositional formula, a literal l_j denotes either a variable x_j or its complement $\bar{x}_j$. If a literal $l_j = x_j$ and x_j is assigned value 1 or $l_j = \bar{x}_j$ and x_j is assigned value 0, then the literal is said to be true. Otherwise, the literal is said to be false.

A propositional clause is a disjunction of literals such as $l_1 \vee l_2 \vee \ldots \vee l_k$ where l_j is a literal representing either x_j or $\bar{x}_j$. We should observe that propositional clauses can also be represented as linear inequalities, e.g. $\sum_{j=1}^{k} l_j \geq 1$. One can obtain a linear inequality as in (1) if we replace literals $\bar{x}_j$ by $1 - x_j$. In the context of SAT we will represent propositional clauses as a disjunction of literals, instead of linear inequalities. However, in the context of ILP or pseudo-Boolean, we use the linear inequalities formalism.

Whenever an assignment to *all* problem variables is found such that all problem constraints become satisfied, we say that a model has been found. However, it may occur that a *partial assignment* (i.e. not all problem variables are assigned) is able to satisfy all problem constraints. In this case, the partial assignment represents a set of models for the problem instance.

We say that an ILP instance defines a *convex polytope* if the number of integer solutions (models) to the ILP constraints is finite. Note that all PB and SAT problem instances define rational convex polytopes since the value of the problem variables is bounded. Hence, the number of solutions is $O(2^n)$ for both PB and SAT instances, where n is the number of problem variables. However, not all ILP instances define convex polytopes. For example, the following ILP has an infinite number of solutions:

$$\begin{array}{l} x_1 - x_2 \leq 10,\ x_1 - x_3 \leq 5 \\ x_1, x_2, x_3 \in \mathbb{Z} \end{array} \tag{2}$$

In section 3 we discuss how to determine if a set of ILP constraints define a convex polytope by finding lower and upper bounds on the value of all problem variables.

3 Encoding ILP into Pseudo-Boolean

This section presents a procedure to encode an Integer Linear Programming (ILP) problem instance into a Linear Pseudo-Boolean (PB) problem instance. The resulting PB instance can then be solved using specific Boolean techniques [1,8]. A key aspect of this encoding is to determine lower and upper bounds on the possible values of the integer valued variables in the ILP. We assume that the ILP instance defines a convex polytope; otherwise the number of integer solutions would be infinite. Hence, every integer variable is guaranteed to have a lower and an upper bound.

Given an ILP instance as presented in section 2, let $lower(x_j)$ and $upper(x_j)$ denote respectively the lower and upper bound on the value of variable x_j in the ILP. If specified in the problem instance, the values of $lower(x_j)$ and $upper(x_j)$ can be determined directly from the instance bounds or by constraints of the type $x_j \geq l$ and $x_j \leq u$.

In general, the lower and upper bounds of an integer valued variable x_j can be determined by solving a linear programming relaxation (LPR) as follows:

$$\begin{aligned} &\text{minimize/maximize } x_j \\ &\text{subject to} \quad \sum_{j=1}^{n} a_{ij}x_j \leq b_i, \\ &\qquad\qquad x_j \in \mathbb{R}, a_{ij}, b_i \in \mathbb{Z} \end{aligned} \tag{3}$$

where we use *minimize* or *maximize* depending on whether we are interested in obtaining a lower or a upper bound, respectively. Note that in this formulation all problem variables are no longer integer and there are known polynomial time algorithms for solving these formulations [14].

Let z_j^m and z_j^M denote respectively the solutions of (3) in the minimization and maximization formulations. By relaxing the variable integer constraints we can obtain a lower and upper bound on the value of x_j, since no integer solution to (1) can be obtained such that $x_j < z_j^m$ or $x_j > z_j^M$. Hence, we have $lower(x_j) = \lceil z_j^m \rceil$ and $upper(x_j) = \lfloor z_j^M \rfloor$, where $\lceil z_j^m \rceil$ denotes the smallest integer value not lower than z_j^m and $\lfloor z_j^M \rfloor$ denotes the largest integer value not higher than z_j^M.

Observe that in order to obtain lower bounds on the problem variables, the well-known replacement of each problem variable x_j with $x_j^{'} - x_j^{''}$, where $x_j^{'} \geq 0$ and $x_j^{''} \geq 0$, cannot be used. For example, suppose we have the following constraints for variable x_1:

$$x_1 \geq -1, x_1 \leq 3 \tag{4}$$

In this formulation, x_1 is clearly bounded. However, if we replace x_1 with $x_1^{'} - x_1^{''}$, we would get:

$$\begin{aligned} &x_1^{'} - x_1^{''} \geq -1,\ x_1^{'} - x_1^{''} \leq 3 \\ &x_1^{'} \geq 0, x_1^{''} \geq 0 \end{aligned} \tag{5}$$

For this new formulation, both variables $x_1^{'}$ and $x_1^{''}$ are not bounded, since we can always find arbitrary large values for $x_1^{'}$ and $x_1^{''}$ such that the constraints are satisfied.

One should also note that if (3) is unbounded for any given problem variable x_j, then the original ILP does not define a convex polytope. Otherwise, if (3) is bounded for all problem variables, then the ILP is a convex polytope and there is a finite number of integer solutions to (1).

Since all integer variables x_j of (1) are limited between $lower(x_j)$ and $upper(x_j)$ in a convex polytope, we can apply a substitution of all variables x_j with $y_j - lower(x_j)$ so that in the new ILP we have all new problem variables y_j bounded between 0 and $upper(y_j)$ where $upper(y_j) = upper(x_j) - lower(x_j)$. Afterwards, we can encode each integer variable y_j as a set of weighted bits as follows:

$$\begin{aligned} &y_j = \sum_{i=0}^{b_j} 2^i y_j^i \\ &y_j^i \in \{0, 1\} \end{aligned} \tag{6}$$

where b_j is the number of bits necessary to represent $upper(y_j)$ and variables y_j^i are Boolean. Additionally, we can also add the following constraints:

$$\sum_{i=0}^{b_j} 2^i y_j^i \leq upper(y_j) \tag{7}$$

As a result of integer variable replacements from (6) and the addition of upper bound constraints from (7), we get a pseudo-Boolean instance that encodes the convex polytope defined by the original ILP.

It is important to ensure that the number of solutions of the PB instance is the same as in the original ILP. Indeed, for this encoding, every unique satisfiable assignment for the ILP instance corresponds to a unique satisfiable assignment for the PB instance, because the integer variables are encoded as a set of weighted bits as it is represented in the computer memory.

4 Model Counting in Pseudo-Boolean Formulations

One way for performing model counting in Pseudo-Boolean (PB) formulations is to implicitly enumerate all possible variable assignments using a backtrack search PB solver. Moreover, current state-of-the-art PB solvers are able to perform *conflict learning* [1,8] and thus prevent entering areas of the search space where no satisfiable assignment exists. This technique has been found particularly useful when the solver has to implicitly visit the complete search space.

It is possible to modify backtrack search PB solvers to count models in PB formulations. Basically, whenever a new solution is found, a propositional clause is added such that it prevents accounting for the same solution later in the search. In the context of model counting, these clauses are known as *blocking clauses* [12,17].

The most straightforward way of generating a new blocking clause is to consider the negation of the search path when a new satisfiable assignment is found. Therefore, if the search path corresponds to the decision assignments $\{x_1 = v_1, x_2 = v_2, \ldots, x_k = v_k\}$, then the blocking clause is defined by:

$$\sum_{j=1}^{k} l_j \geq 1 \tag{8}$$

where $l_j = x_j$ if $x_j = 0$ in the search path or $l_j = \bar{x}_j$ if $x_j = 1$. Observe that this blocking clause prevents the current search path to occur later in the search. Hence, the models corresponding to the partial solution will not be counted twice. Note that a partial assignment of k problem variables that satisfies all problem constraints implicitly represents a set of models. Therefore, whenever a PB solver finds a new solution considering only k variables, it has in fact found 2^{n-k} possible ways of satisfying the problem constraints.

Simplification of Satisfying Partial Assignments. It is well-known that the problem of computing the satisfying partial assignment with the smallest number of specified variables can be formulated as an integer linear program [15]. However, we are just interested in simplifying satisfying partial assignments computed by a PB solver.

Variable lifting denotes a number of techniques used for the elimination of assignments that can be declared redundant [12,17]. A simple variable lifting technique consists of removing from a satisfying partial assignment all variable assignments that are not used to satisfy any constraint. Moreover, these variable assignments cannot also be used in constraints that imply other variable assignments. When using this technique, we can immediately conclude that all implied assignments cannot be removed from the partial assignment since they are necessary to satisfy at least one constraint. Otherwise, these assignments would not be implied. Hence, we only have to check decision assignments.

Suppose we have the following decision assignment $x_1 = x_2 = x_3 = 0$ and that $x_5 = 0$ is an implied assignment. Consider also the following constraints:

$$(x_1 + x_2 + x_3 \leq 1) \land (x_2 + x_3 \leq 1) \land (x_2 + x_4 \leq 1) \land (-x_3 + x_5 \leq 0) \qquad (9)$$

Clearly, the assignment to x_1 is not relevant to satisfy the problem constraints. Note that x_3 cannot be considered irrelevant, since it is necessary to imply the value of x_5. Hence, the resulting blocking clause would be $x_2 + x_3 \geq 1$. Since there are two variables (x_1 and x_4) that are not relevant to satisfy the constraints in this partial assignment, then we conclude that 4 models have been found. In [12,17] other lifting techniques are presented. However, they require a significant computational overhead.

Additional SAT Techniques. The identification of connected components [6] and component caching [18] are among the most effective techniques for model counting instances of SAT. These techniques are not yet integrated in the PB model counter described above, since they will require significant re-implementation effort, and our objective is first to evaluate whether these techniques are effective for ILP and PB model counting.

5 Encoding Pseudo-Boolean Constraints as SAT

Several algorithms exist that are based on modifying a SAT solver in order to deal with pseudo-Boolean constraints [1,8], as well as generalizing other techniques like conflict analysis. A different approach is based on encoding all pseudo-Boolean constraints into propositional clauses and use a SAT solver directly [1,3,4,9,19]. Some of these encodings are polynomial whereas others are exponential in the worst case. The objective of encoding PB constraints into SAT is to take advantage of the powerful techniques of SAT solvers in dealing with propositional clauses. This section defines *counting safety*, and shows that not all encodings are counting safe. Moreover, this section also shows that some encodings are counting safe and so can be used for model counting.

Definition 1 (Counting Safety). *A PB formulation to SAT encoding is counting safe iff the number of models in the PB formulation and in the encoded SAT formulation are the same.*

5.1 Unsafe Encodings

The vast majority of PB to SAT encodings solely aim the discovery of one solution and may introduce auxiliary variables. These variables may lead to double counting of the same solution in pseudo-Boolean. For example, consider the following PB constraints:

$$\begin{aligned} 2x_0 + 4x_1 + 8x_2 + 3y_0 + 6y_1 + 12y_2 \leq 18 \\ -2x_0 - 4x_1 - 8x_2 + 1y_0 + 2y_1 + 4y_2 \geq -10 \end{aligned}$$

If *any* of the encodings proposed in [9] is used with this example, and the resulting CNF formula is given to model counter, e.g. *cachet* [18], the number of models reported will be at least 38. However, the correct number of models for this example is 31. Hence, the encodings proposed in [9] do not satisfy the counting safety property, and cannot be used for model counting. The next section addresses encodings which are counting safe.

5.2 Safe Encodings

Both the well-known Warners PB to SAT encoding [19] as well as the more recent arc-consistency encoding of Bailleux, Boufkhad and Roussel (BBR) [4] can be shown to be counting safe. Due to space constraints, this section addresses solely the BBR encoding; a detailed analysis of Warners encoding is available in [13]. Next, we provide a brief description of the BBR PB to SAT encoding [4].

Consider a pseudo-Boolean constraint ω with the constraint literals l_j sorted according to their coefficients a_j:

$$\begin{aligned} &\omega = \sum_{j=1}^{n} a_j l_j \leq b, \\ &\text{where } 0 < a_1 \leq a_2 \leq \ldots \leq a_n \end{aligned} \tag{10}$$

Let $\omega_{i,k}$ represent the constraint ω considering only the first i literals ($0 \leq i \leq n$) with the right-hand side value k, i.e. $\omega_{i,k} : \sum_{j=1}^{i} a_j l_j \leq k$. Therefore, the original constraint ω corresponds to $\omega_{n,b}$.

In order to generate the CNF encoding for a given constraint ω, we need to introduce new Boolean variables $D_{i,k}$ which represent the satisfaction of constraints $\omega_{i,k}$ obtained from ω. Hence, we have $D_{i,k} = 1$ iff constraint $\omega_{i,k}$ is satisfied. As a result, $D_{n,b} = 1$ represents the satisfaction of the original pseudo-Boolean constraint ω in the CNF encoding.

When building the CNF encoding, variables $D_{i,k}$ are said to be *terminal* if $k \leq 0$ or if $\sum_{j=1}^{i} a_j \leq k$. Otherwise, variables $D_{i,k}$ are said to be *non-terminal*.

The CNF encoding for a pseudo-Boolean constraint ω proceeds as follows:

1. Start with a set of variables containing variables x_j in constraint ω, as well as variable $D_{n,b}$, and an empty set of propositional clauses. Mark all variables x_j.
2. Consider an unmarked variable $D_{i,k}$.
3. If $D_{i,k}$ is a non-terminal variable, add two new variables $D_{i-1,k}$ and $D_{i-1,k-a_i}$ to the set of variables, if they are not already in this set. Moreover, mark selected variable $D_{i,k}$ and add the following propositional clauses:

$$\bar{D}_{i-1,k-a_i} + D_{i,k} \geq 1 \tag{11}$$
$$\bar{D}_{i,k} + D_{i-1,k} \geq 1 \tag{12}$$
$$\bar{D}_{i,k} + \bar{l}_i + D_{i-1,k-a_i} \geq 1 \tag{13}$$
$$\bar{D}_{i-1,k} + l_i + D_{i,k} \geq 1 \tag{14}$$

4. If $D_{i,k}$ is a terminal variable, and if $k \neq 0$, then:

$$D_{i,k} = \begin{cases} 0 \text{ if } k < 0. \text{ Add } \bar{D}_{i,k} \geq 1 \text{ to the clause set.} \\ 1 \text{ if } \sum_{j=1}^{i} a_j \leq k. \text{ Add } D_{i,k} \geq 1 \text{ to the clause set.} \end{cases} \tag{15}$$

   Otherwise, if $k = 0$, then add the following set of clauses:

$$\bar{D}_{i,k} + \bar{l}_j \geq 1, 1 \leq j \leq i \tag{16}$$
$$\sum_{j=1}^{i} l_j + D_{i,k} \geq 1 \tag{17}$$

5. If there are any unmarked variables, go to step 2. Otherwise, the propositional clause set contains the CNF encoding of constraint ω and the procedure terminates.

The following example illustrates how the proposed CNF encoding works:

$$\omega : 2\bar{x}_1 + 3x_2 + 3x_3 \leq 5 \tag{18}$$

Figure 1 presents the new variables created for the CNF encoding. For each non-terminal variable, two new additional variables are created whereas terminal variables are represented as leaf nodes. The full encoding for constraint ω as a set of propositional clauses is as follows:

$$\begin{array}{lll} D_{3,5} \geq 1 & \bar{D}_{1,-1} + D_{2,2} \geq 1 & \bar{D}_{1,-1} \geq 1 \\ \bar{D}_{2,2} + D_{3,5} \geq 1 & \bar{D}_{2,2} + D_{1,2} \geq 1 & D_{1,2} \geq 1 \\ \bar{D}_{3,5} + D_{2,5} \geq 1 & \bar{D}_{2,2} + \bar{x}_2 + D_{1,-1} \geq 1 & \\ \bar{D}_{3,5} + \bar{x}_3 + D_{2,2} \geq 1 & \bar{D}_{1,2} + x_2 + D_{2,2} \geq 1 & \\ \bar{D}_{2,5} + x_3 + D_{3,5} \geq 1 & D_{2,5} \geq 1 & \end{array} \tag{19}$$

One should note that in addition to the propositional clauses added by the encoding procedure, it is also necessary that variable $D_{3,5}$ be assigned value 1.

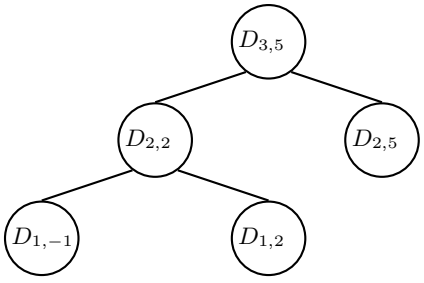

Fig. 1. Additional variables in CNF encoding

This is required since $D_{3,5}$ represents the satisfaction of the original pseudo-Boolean constraint.

Finally, we refer to [4] for details, namely proofs of correction of the encoding and of the maintenance of generalized arc consistency in the resulting CNF encoding, as well as examples of constraints for which this encoding provides exponential increase in the size of the CNF encoding.

Theorem 1. *The BBR arc-consistency encoding [4] is counting safe.*

Proof:
We want to prove that to each model for the PB constraints there is a corresponding unique model for the SAT encoding, and that to each model for the SAT encoding there is a unique corresponding model for the PB constraints.

($\leftarrow$) If we have a model for the SAT encoding then we can only have one model for the PB constraints, which is the model for the SAT encoding restricted to the variables for the PB constraints.

($\rightarrow$) Suppose we have a model for the PB constraints. We know that the encoding is correct [4], so we already know that there exists at least one model for the SAT encoding corresponding to the model for the PB constraints. What we want to show is that this model is unique. We also know that the model in SAT corresponds to the same assignments made to the variables of the PB constraints plus the assignments to the variables introduced by the encoding. In order to show that the model is unique all we have to prove is that these new variables $D_{i,b}$ can only have one possible assignment for satisfying the created instance. We are going to prove this claim by induction on n. Let us first consider a PB constraint $\sum_{j=1}^{n} a_j x_j \leq k$ and the corresponding variable node $D_{n,k}$.

Base: $n = 1$:
There can be two cases, depending on whether $D_{1,k}$ is terminal. If $D_{1,k}$ is terminal we may have $k \neq 0$ or $k = 0$. If $k \neq 0$, due to the definition of a terminal node, either $k < 0$ and the value of $D_{1,k}$ is 0, or $a_1 \leq k$ and the value of $D_{1,k}$ is 1. Otherwise, if $k = 0$, then the encoding adds clauses $(\bar{D}_{1,0} \vee x_1)$, $(D_{1,0} \vee \bar{x}_1)$. Since x_1 has a determined value it implies the unique value of $D_{1,k}$.

We now consider the case when $D_{1,k}$ is not terminal. In this case, the encoding adds the following clauses: $(\bar{D}_{0,k-a_1} \vee D_{1,k})$, $(D_{1,k} \vee D_{0,k})$, $(\bar{D}_{1,k} \vee \bar{x}_1 \vee D_{0,k-a_1})$,

and $(\bar{D}_{0,k} \vee x_1 \vee D_{1,k})$. Since $D_{1,k}$ is not terminal, then if $k > 0$ we have $D_{0,k} = 1$ or if $k < a_1$ we have $D_{0,k-a_1} = 0$. The first and the second clauses added by the encoding are trivially satisfied. By removing the false literals of the third and fourth clauses we get $(\bar{D}_{1,k} \vee \bar{x}_1)$, $(x_1 \vee D_{1,k})$. Since x_1 has a determined value both clauses imply the unique value of $D_{1,k}$.

Step:
Hypothesis: For $i < n$, $D_{i,b}$ has only one possible assignment that satisfies the created instance.

We now consider node $D_{n,k}$. If the node is terminal with $k < 0$, then the encoding adds the clause $(\bar{D}_{n,k})$ and $D_{n,k}$ can only be assigned value false.

If the node is terminal with $k \neq 0$ and $k \geq \sum_{j=1}^{n} a_j$, then the encoding adds the clause $(D_{n,k})$ and $D_{n,k}$ can only be assigned value true.

If the node is terminal with $k = 0$, then the encoding adds the clauses $(\bar{D}_{n,0} \vee \bar{x}_j)$, $1 \leq j \leq n$ and $(x_1 \vee x_2 \vee \ldots \vee x_i \vee D_{n,0})$. Two situations may occur. If all the variables x_j, $1 \leq j \leq n$, are false, then (17) implies the value of $D_{n,0}$ to true. If at least one x_j, $1 \leq j \leq n$ is true, then (16) implies the value of $D_{n,0}$ to false and x_j satisfies (17).

If the node is non-terminal, then we apply the hypothesis to $D_{n-1,k}$ and to $D_{n-1,k-a_{n-1}}$. We get that these variable nodes have a fixed known value. We have four cases depending on the value of the variable nodes:

- If $D_{n-1,k}$ and $D_{n-1,k-a_{n-1}}$ are false, then from (12) $D_{n,k}$ can only be false and the other clauses are all satisfied.
- If $D_{n-1,k}$ is false and $D_{n-1,k-a_{n-1}}$ is true, then from (11) and (12) there is a contradiction, and the encoded instance is unsatisfiable. This situation is acceptable since we cannot have $(\sum_{j=1}^{n-1} a_j x_j \leq k - a_{n-1})$ being satisfiable and at the same time being unable to satisfy the same left hand side of the equation with a higher right hand side $(\sum_{j=1}^{n-1} a_j x_j \leq k)$.
- If $D_{n-1,k}$ is true and $D_{n-1,k-a_{n-1}}$ is false, then we can have two cases:
  x_n **is false** , then from (14) $D_{n,k}$ can only be true and the other clauses are all satisfied;
  x_n **is true** , then from (13) $D_{n,k}$ can only be false and the other clauses are all satisfied.
- If $D_{n-1,k} = 1$ and $D_{n-1,k-a_{n-1}} = 1$, then from (11) $D_{n,k} = 1$ and all the other clauses are satisfied.

Finally, the results holds for any constraint, and so necessarily holds for all constraints in an instance of PB. ■

6 Experimental Results

This section presents experimental results for model counting in integer domains. Instances from different problems are considered. Moreover, different model counting approaches are evaluated. The existing implementation of Barvinok's algorithm, *LattE* [11], is evaluated. A prototype PB model counter, described in Section 4 is evaluated. Finally, SAT model counters are evaluated. In order

to use the SAT model counters, we use the counting safe encodings of Warners [19] as well as the arc-consistency encoding proposed by Bailleux, Boufkhad and Roussel (BBR) [4]. All CPU times presented are for a AMD Athlon 1.9GHz processor with 1GB of physical memory. The time limit for each instance was set to one hour. If the time limit was reached, we provide a partial solution when one is available, i.e. the number of models found when the search was stopped. This is preceded by the sign $\geq$ since the total number of models must be higher than or equal to the ones already found. If all models are found, we provide the total time in seconds. In cases where time limit was reached and the counter did not provide any count of the number of models, **TO** (Time Out) is shown. All processes were run with 900MB of allowed memory. **MO** (Memory Out) is shown for the cases where the counter reached the allowed memory limit. On some instances of Table 1 **MO*** is shown because it was the translator of pseudo-Boolean to SAT that reached the memory limit instead of the counter. Observe that **MO*** can only take place with the BBR encoding.

The experimental results are shown in Table 1, and are organized in three parts, according to the source of the problem instances. The first part of Table 1 presents results for instances of finding the *Frobenius* number plus 1 in knapsack problems [11]. For these instances, *Latte* is the only solver able to count the number of models. The underline numerical problem proved to be very difficult for pseudo-Boolean or SAT counters. Nevertheless, our solver *pb_counter* was able to find partial solutions for some instances, while both *cachet* and *relsat* (in both encodings) were unable to do so.

The second part of Table 1 presents results for ILP benchmarks, generated from well-known graph coloring benchmarks. Since the optimum number of colors is known for these instances, a constraint was added in order to count the number of optimum solutions. For these instances *Latte* performed poorly. In fact, our experience is that *Latte* performs better for numerical problems with very few variables. On the other hand, both *relsat* and *pb_counter* were able to solve some instances, as well as providing partial solutions for most instances. It can also be observed from the number of partial solutions found, that better results were obtained using *relsat* with the BBR encoding. Like *relsat*, *cachet* was also able to solve completely 3 instances (using the BBR encoding), but on all others no solution was found because *cachet* exhausted the available memory.

Finally, the third part of Table 1 presents results for benchmarks which encode the problem of finding the minimum-size prime implicant [15] for several instances from the DIMACS [10] benchmark suite. As in the graph coloring instances, the optimum value for these instances is also known. For each instance, a constraint was added in order to only allow models with the optimum value. Hence, for each instance, the number of models corresponds to number of different minimum-size prime implicants. Observe that *Latte* was unable to solve any of these instances, while the other model counters were able to find the total number of models for most instances. In fact, it was surprising that *pb_counter* was able to outperform the other counters on these instances, since most of the constraints are originally propositional clauses.

Table 1. Results on several benchmark instances

			relsat		*cachet*		
Benchmark	# Models	*Latte*	Warners	BBR	Warners	BBR	*pb_counter*
cuww1	1	0.11	TO	MO	MO	MO	TO
cuww2	1	0.40	TO	MO*	MO	MO*	TO
cuww3	2	0.35	TO	MO*	MO	MO*	TO
cuww4	1	0.55	TO	MO*	MO	MO*	TO
cuww5	1	4.51	TO	MO*	MO	MO*	TO
prob1	8.592e8	22.14	TO	MO*	MO	MO*	TO
prob2	2.047e6	13.51	TO	MO*	MO	MO*	TO
prob3	0	27.01	TO	TO	MO	MO	TO
prob4	6.319e7	19.68	TO	MO*	MO	MO*	TO
prob5	2.178e10	18.65	TO	MO*	MO	MO*	$\geq$ 4.337e4
prob6	2.188e5	96.12	TO	MO*	MO	MO*	$\geq$ 7
prob7	4.198e12	81.20	TO	MO*	MO	MO*	$\geq$ 3.812e4
prob8	6.743e6	257.77	TO	MO*	MO	MO*	TO
prob10	1.024e17	145.83	TO	MO*	MO	MO*	$\geq$ 764
1-FullIns_3	5.069e7	–	$\geq$ 8.448e6	2701.56	MO	67.73	$\geq$ 5.194e6
2-FullIns_3	–	–	$\geq$ 3.061e7	$\geq$ 5.381e8	MO	MO	$\geq$ 1.067e7
2-Insertions_3	–	–	$\geq$ 4.156e7	$\geq$ 4.625e8	MO	MO	$\geq$ 9.006e6
3-FullIns_3	–	–	$\geq$ 5.542e6	TO	MO	MO	$\geq$ 5.217e6
3-Insertions_3	–	–	$\geq$ 6.291e7	$\geq$ 2.527e10	MO	MO	$\geq$ 3.094e7
4-Insertions_3	–	–	$\geq$ 6.753e7	$\geq$ 1.307e12	MO	MO	$\geq$ 1.501e7
games120	–	–	$\geq$ 1.296e4	$\geq$ 1.407e6	MO	MO	$\geq$ 1.194e6
mug100_1	–	–	$\geq$ 2.967e13	$\geq$ 2.725e23	MO	MO	$\geq$ 2.476e7
mug88_1	–	–	$\geq$ 9.834e6	$\geq$ 1.138e23	MO	MO	$\geq$ 1.513e7
myciel3	12480	–	4.75	1.27	0.41	0.26	0.96
myciel4	–	–	$\geq$ 2.995e6	$\geq$ 7.215e7	MO	MO	$\geq$ 5.065e6
myciel5	–	–	$\geq$ 1.150e9	$\geq$ 3.637e11	MO	MO	$\geq$ 1.134e7
queen5_5	240	–	1123.32	151.69	71.31	29.78	4.38
queen6_6	–	–	TO	$\geq$ 2	MO	MO	$\geq$ 1.251e4
queen7_7	–	–	TO	TO	MO	MO	$\geq$ 1.447e3
aim-100-1_6-yes1-2	1	–	0.19	9.04	3.58	2.27	0.02
aim-100-2_0-yes1-3	1	–	0.19	9.59	2.73	2.61	0.03
aim-100-3_4-yes1-4	1	–	0.2	9.95	2.37	4.98	0.04
aim-100-6_0-yes1-1	1	–	0.2	9.76	0.70	1.31	0.05
aim-200-1_6-yes1-3	1	–	0.42	185.29	30.67	20.42	0.04
aim-200-2_0-yes1-4	1	–	0.44	181.01	15.89	41.97	0.07
aim-200-3_4-yes1-1	1	–	0.42	172.86	13.16	MO	0.08
aim-200-6_0-yes1-2	1	–	0.51	169.77	6.67	24.55	0.14
ii8a1	1056	–	17.41	45.77	18.67	7.35	7.25
jnh1	12	–	2.26	1079.66	7.52	8.12	0.53
jnh12	1	–	0.21	1.95	0.97	0.81	0.07
jnh17	35	–	0.42	6.13	1.19	2.49	0.20
jnh7	26	–	1.37	6.09	4.18	1.75	0.26
ssa7552-038	–	–	TO	TO	MO	MO	$\geq$ 1.853e4
ssa7552-158	–	–	$\geq$ 1.319e13	TO	MO	MO	$\geq$ 5.183e4

7 Related Work

Besides the work on model counting and enumeration in SAT [6,12,17,18], there has been work on model counting in non-Boolean domains, including Integer Linear Programming (ILP) [5,11] and Linear Integer Arithmetic (LIA) [7,16].

Existing work on model counting in LIA is described in [7,16]. The work of [7] is based binary decision diagrams and does not scale to large number of variables. The work of [16] enumerates a large number of applications for model counting in LIA. The proposed algorithm is also only suitable for a small number of variables, or when most variables have fixed values.

The most well-known work on model counting in ILP is Barvinok's algorithm [5]. An existing implementation, *LattE* [11], which incorporates a number of improvements, has been extensively used. As the results of Section 6 confirm, Barvinok's algorithm is adequate for instances of ILP with a small number of variables which may have larger domains. Observe that the algorithms for model counting in LIA can also be used for ILP (a special case of LIA) but current solutions can only handle small instances.

8 Conclusions

This paper proposes two alternative approaches for counting models in ILP instances. The first approach is based on encoding ILP into Pseudo-Boolean (PB) and using a PB counter. A PB counter was developed for this purpose. A second approach is based on encoding ILP into SAT, using an intermediate encoding into PB. The paper shows that some PB to SAT encoding may overestimate the number of models, whereas others are shown to yield the correct number of models. As a result, counting models in integer domains can be achieved by encoding ILP constraints into SAT and directly using SAT model counters, thus taking advantage of the techniques already incorporated into SAT counters. Experimental results indicate that the PB counter is competitive with the SAT counters. Moreover, an existing alternative to SAT-based model counters, using Barvinok's algorithm [5,11], provides essentially orthogonal results, being more efficient for problem instances having few variables with large domains, and being inadequate for problem instances having many variables with small domains.

Despite the interesting insights, many challenges still remain. The PB counter is a prototype, aiming to prove the concept of counting models for PB constraints. A more sophisticated algorithm is expected to provide significant gains, for example if connected components are identified for PB constraints. There is also a clear gap between *LattE* (the implementation of Barvinok's algorithm) and the SAT-based solutions. Work on closing this gap is also an interesting challenge. Finally, the utilization of the model counter in instances of linear integer arithmetic [7,16], one of the motivations for this work, will require significantly more optimized ILP counters.

Acknowledgments. This work is partially supported by FCT under research projects POSI/SRI/41926/2001 and POSC/EIA/61852/2004.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *International Conference on Computer-Aided Design*, pages 450–457, November 2002.
2. F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Symposium on Foundations of Computer Science*, pages 340–351, 2003.
3. O. Bailleux and Y. Boufkhad. Full CNF encoding: The counting constraints case. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.
4. O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo Boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, March 2006.
5. A. Barvinok and J. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New Perspectives in Algebraic Combinatorics*, volume 38, pages 91–147. MSRI Publications, Cambridge University Press, 1999.
6. R. J. Bayardo and J. D. Pehoushek. Counting models using connected components. In *National Conference on Artificial Intelligence*, 2000.
7. B. Boigelot and L. Latour. Counting the solutions of presburger equations without enumerating them. *Theoretical Computer Science*, 313(1):17–29, 2004.
8. D. Chai and A. Kuehlmann. A Fast Pseudo-Boolean Constraint Solver. In *Proceedings of the Design Automation Conference*, pages 830–835, 2003.
9. N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2, March 2006.
10. D. S. Johnson and M. A. Trick. Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1994.
11. J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, 2004.
12. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *International Conference on Computer-Aided Verification*, 2002.
13. A. Morgado, P. Matos, V. Manquinho, and J. Marques-Silva. Counting models in integer domains. Technical Report 05/2006, INESC-ID, March 2006.
14. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.
15. C. Pizzuti. Computing Prime Implicants by Integer Programming. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, pages 332–336, November 1996.
16. W. Pugh. Counting solutions to presburger formulas: How and why. In *Conference on Programming Language Design and Implementation*, pages 121–134, June 1994.
17. K. Ravi and F. Somenzi. Minimal satisfying assignments for conjunctive normal formulae. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2004.
18. T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, May 2004.
19. J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.