Pseudo-Boolean Solving by Incremental Translation to SAT

Panagiotis Manolios
Northeastern University
pete@ccs.neu.edu

Vasilis Papavasileiou
Northeastern University
vpap@ccs.neu.edu

***Abstract*—We revisit pseudo-Boolean Solving via compilation to SAT. We provide an algorithm for solving pseudo-Boolean problems through an incremental translation to SAT that works with any incremental SAT solver as a backend. Experimental evaluation shows that our incremental algorithm solves industrial problems that previous SAT-based approaches do not. We also show that SAT-based algorithms for solving pseudo-Boolean problems should be a part of any portfolio solver.**

I. Introduction

Boolean Satisfiability (SAT) has been the subject of intensive research over the past decade. Many powerful solvers have been developed, and SAT has been successfully applied to problems in a variety of fields, like electronic design automation, hardware verification, AI planning, and others. In many domains, the need for non-propositional constraints like linear inequalities naturally arises. The *Pseudo-Boolean* (PB) formalism accommodates linear constraints over Boolean variables.

Definition 1. *A* Pseudo-Boolean constraint *is a constraint of the form*

$$c_1 p_1 + c_2 p_2 + \cdots + c_n p_n \ \square \ r \tag{1}$$

where $\square$ *is one of the relations* $<$, $\leq$, $=$, $<$, *or* $\geq$, *the* variables p_i *(for* $1 \leq i \leq n$*) can take the values* 0 *and* 1 *and the* coefficients c_i *are integers.*[1] *The integer* r *is the* right-hand side. *We call* $c_i p_i$ *a* term. *A PB problem is a conjunction of PB constraints.*

Several kinds of solvers can deal with PB problems. PB satisfiability (or optimization) is a restriction of *Integer Linear Programming* to 0-1 variables. As a result, ILP solvers can be used. PB constraints can also be thought of as a generalization of clauses. Thus, SAT techniques can be applied, *e.g.*, the DPLL procedure can be modified to handle PB constraints [1, 2]. An alternative is to compile PB constraints to CNF and use an off-the-shelf SAT solver [3, 4, 5].

We revisit PB solving via compilation to SAT. An advantage of this approach is that tools based on it automatically become more competitive as the performance of the underlying SAT solvers they depend on improves. Furthermore, SAT solvers provide flexible, robust, mature, and well-engineered interfaces that have found a plethora of interesting applications. Finally, it is desirable to have a portfolio of complementary solvers that in aggregate provide good performance over a large class of PB problems arising in practice. We show that SAT-based approaches should be an integral part of any such portfolio: while they are not competitive on some PB problems (*e.g.*, those that benefit from ILP-based techniques such as cutting planes), they do very well on other types of problems (*e.g.*, those where the ratio of propositional to arithmetic constraints is high).

We propose an algorithm for PB solving that uses a SAT solver for the efficient exploration of the search space, but at the same time exploits the high-level structure of the PB constraints to simplify the problem and direct the search. We are primarily concerned with industrial problems and show that our algorithm can tackle industrial PB instances that were previously beyond the reach of SAT-based solvers.

The rest of the paper is organized as follows. In section II, we present a class of incremental algorithms for solving PB problems. Our algorithms are parameterized by a method for translating from PB constraints to CNF, and in section III, we analyze several different encoding schemes. We experimentally evaluate our solver in section IV. We review related work in section V, and conclude with section VI.

II. Incremental Solving

Our PB solver, PB-SAT, works by translating constraints to CNF *incrementally*, in stages and performs multiple SAT calls. Knowledge we acquire after each call allows us to simplify the remaining untranslated constraints.

Algorithm 1 shows the basic structure of our solver. We assume the existence of a function TRANSLATE that converts a PB constraint C to an equisatisfiable propositional formula TRANSLATE(C). The algorithm is independent of the specifics of TRANSLATE.

In Algorithm 1, Φ corresponds to the set of clauses we have generated so far. We initialize Φ so that it contains *PB-clauses*: PB constraints of the form $l_1 + l_2 + \ldots + l_n \geq 1$, where the l_i's are literals. Optionally, we can conjoin to Φ the translations of cardinality constraints and other constraints that can be efficiently encoded as clauses. Ψ, initialized to contain the non PB-clauses, is used to record the PB constraints that have not yet been translated to CNF.

[1] We convert the constraints to a normal form with only positive coefficients and no relation other than $\geq$. In this normal form, variables can appear negated.

Algorithm 1 PB Solving by incremental translation to SAT

```
1:  procedure PB-SAT(Ψ)
2:      Φ ← {φ ∈ Ψ : φ is a PB-clause}
3:      Ψ ← {ψ ∈ Ψ : ψ is not a PB-clause}
4:      while true do
5:          A, U ← SAT(Φ)
6:          if A = UNSAT then return UNSAT
7:          Ψ ← SIMPLIFY(Ψ, U)
8:          if A satisfies Ψ then return A
9:          Ψ′ ← {ψ ∈ Ψ : ψ falsified by A}
10:         if Ψ′ = ∅ then Ψ′ ← SELECT(Ψ)
11:         Ψ ← Ψ \ Ψ′
12:         for all ψ ∈ Ψ′ do Φ ← Φ ∧ TRANSLATE(ψ)
```

After initialization, our algorithm enters a loop where it calls the SAT solver on the current set of clauses, Φ (line 5). If Φ is unsatisfiable, the SAT solver returns "*UNSAT*" for A and the input to the algorithm is also unsatisfiable. Otherwise, the SAT solver returns a *partial* satisfying assignment A and the set of known unit literals, U. In line 7, we use U to simplify Ψ, the constraints we have yet to translate. How this is done is explained later. Now, if any full assignment that extends A also satisfies Ψ, then A is a partial assignment that satisfies the input to our algorithm, so we return A. Otherwise, any PB constraints that are false under every full assignment extending A are stored in Ψ'. The idea is to only translate these falsified constraints in the next round, but it may turn out that A does not falsify any of the remaining constraints. In that case, we select a non-empty subset of Ψ to translate. (Note that $\Psi \neq \emptyset$ is an invariant holding right after line 8). Next, in line 11, we update Ψ to reestablish the invariant that it contains the constraints left to translate. We end the loop by translating all the PB constraints in (the non-empty) Ψ'.

Note that we translate all constraints falsified by intermediate partial assignments. However, a lazier version of our algorithm could translate only a subset of the falsified constraints. While there are many ways of deciding what to translate during each iteration, the essence of our approach is to incrementally translate constraints in order to obtain useful information that is used to simplify the remaining constraints.

A. Simplification

The SIMPLIFY function in algorithm 1 uses units discovered by the SAT solver during the incremental queries to simplify the remaining constraints. We explain this with an example.

Example 1. *Given the units x_1 and $\neg x_2$, we simplify the constraint $2x_1 + 2x_2 + x_3 + x_4 \geq 4$ to $x_3 + x_4 \geq 2$, which we further simplify to the units x_3, x_4.*

Notice that we propagate knowledge in both directions: (i) we use units from SAT solving to simplify the PB constraints, and (ii) we learn new units from PB constraint propagation. SIMPLIFY propagates the units we know at the PB level, as described above. This process may modify the constraints and return new units. We give these units to the SAT solver, and perform Boolean Constraint Propagation (BCP). If BCP leads to more units, we repeat. We stop when we reach a fixpoint (we no longer learn anything new).

Our incremental strategy works by considering only a subset of the PB constraints: the ones falsified by intermediate assignments. This lazy approach is very useful in applications like synthesis, where we expect the PB constraints to be satisfiable. For such applications, our approach tends to steer the SAT solver towards satisfying assignments. In addition since we return partial assignments, we can return many solutions simultaneously. If the formula is unsatisfiable, by focusing on the PB constraints that are falsified, we may wind up discovering an unsatisfiable core of PB constraints without encoding all the PB constraints.

B. Discovering More Units

SAT solving and propagation at the PB level as per algorithm 1 may not discover *all possible* units. The reason is that a SAT solver does not discover all the units implied by a propositional formula during the search process. Algorithm 2 offers a practical way to discover more units implied by Φ. The basic idea is as follows: we first find A, a satisfying (partial) assignment for Φ. Now, suppose that literal l is true under A, then l may be a unit (certainly $\neg l$ is not), which we check with the SAT query $\Phi \wedge \neg l$. We can control the time this operation takes by imposing a limit on a resource R, for example the decisions or the propagation steps that the SAT solver performs (call to SAT-LIMITED in line 9).

Relying on a single assignment is not a good idea. Instead, we maintain a set α that contains different assignments for Φ. We only perform queries of the form $\Phi \wedge \neg l$ on variables for which every assignment in α assigns a value (recall assignments are partial) and all these values are equal (condition in line 7). We note that this check and the assignment in line 8 can be implemented efficiently using bit-vectors. If a query on a formula $\Phi \wedge \neg l$ returns an assignment, this assignment also satisfies Φ, so we add it to the set α.

Algorithm 2 Extracting units implied by a formula Φ

```
1:  procedure MORE-UNITS(Φ)
2:      A, U ← SAT(Φ)
3:      if A = UNSAT then return UNSAT
4:      α ← {A}
5:      for all l ∈ U do Φ ← Φ ∧ l
6:      for all variables v s.t. v ∉ U ∧ ¬v ∉ U do
7:          if ∀A1, A2 ∈ α : A1(v) = A2(v) then
8:              l ← POLARITY(A′, v) for some A′ ∈ α
9:              B ← SAT-LIMITED(Φ ∧ ¬l, R)
10:             if B = UNSAT then
11:                 U ← U ∪ {l}
12:                 Φ ← Φ ∧ l
13:             else α ← α ∪ {B}
14:     return PICK(α), U
```

In addition to the units implied by Φ, MORE-UNITS returns a satisfying assignment if there is one. The assignment returned can be any of the assignments in α. Notice that we can simply instantiate SAT in algorithm 1 with MORE-UNITS. In our implementation, we use MORE-UNITS only on the initial formula that contains PB-clauses and cardinality constraints.

C. Optimization

Algorithm 1 can be extended to handle optimization problems. Assume that the problem is minimizing the objective function $f(X)$. Whenever we get a satisfying assignment such that $f(X) = V$, we add the constraint $f(X) < V$ in order to obtain solutions that decrement $f(X)$ by at least 1. We also reset the variable phases to random values, so that the next assignment will not be a small variation of the current assignment. When the problem becomes unsatisfiable, we report the last known V and the corresponding assignment.

Solving optimization problems by decrementing by 1 is naive, but straightforward to implement using an incremental SAT solver. We could have used binary search or some related approach. That would require backtracking, which can be implemented using assertion literals. Our preliminary analysis indicated that using binary search would not have helped us solve more optimization problems in the PB Competition [6], hence we did not implement it. However, as we note in Section IV, the community needs more industrial PB benchmarks.

III. Translation to CNF

In this section, we explain how different encodings of PB constraints affect the behavior of our incremental solver. Encodings of PB constraints into CNF differ (i) in the size of the resulting formulas, and (ii) with regards to the implications preserved between the variables.

The notion of *arc-consistency* captures the desired property of preserving implications: an encoding (say the one generated by the function TRANSLATE) is *arc-consistent* if an assignment that can be propagated on the original constraints can also be propagated on the translated constraints. For a partial assignment A, a PB constraint C and a literal l, if A can be extended to a model of C but $A \cup \{l\}$ cannot, then unit propagation on TRANSLATE(C) and A will produce $\neg l$. Choosing an encoding is a trade-off between (proximity to) arc-consistency and size.

We implemented translations through adders and BDDs, as described in [3]. The encoding through adders is linear, but it does not maintain arc-consistency. It works by synthesizing a network of adders that adds up the terms in the left-hand side, and a circuit comparing the sum to the right-hand side. The encoding for the sum bit of full adders requires ternary XORs, which are known to be problematic for SAT solvers. However, adder-based encodings have the advantage that they lead to small formulas. The benefit of an incremental approach in this case is that we detect implications of the units we learn by performing PB unit propagation, and simplify the problem accordingly. Some of these implications would be lost if we translated everything at once.

In contrast to adders, translation through BDDs is arc-consistent; however, the size of the resulting BDDs is exponential in the worst case. In our examples, some of the original PB constraints are practically impossible to translate through BDDs. Translation becomes possible after we learn units and simplify the problem. In fact, our lazy algorithm sometimes allows us to solve problems without even constructing BDDs for PB constraints we could not directly translate.

Another reason to prefer BDDs is that they can represent conjunctions of constraints. Frequently there are sets of PB constraints with identical sets of variables. We can hash all constraints with the sorted list of variables as the signature, and conjoin the constraints mapped to the same hash value. It is straightforward to adapt the BDD construction algorithm of [3] to build BDDs for conjunctions of constraints. This can lead to more compact encodings. More importantly, we can achieve arc-consistency for the conjunction of constraints.

In general, we can mix encodings and pick the most suitable for each constraint. We use adders only when the BDD is too big, but we could also use sorters. Incremental translation allows us to use BDDs more frequently.

IV. Experimental Evaluation

PB-SAT is implemented in Common Lisp and uses PicoSAT [7] as the backend. In principle we can use any SAT solver that provides incremental functionality. The source code is publicly available.[2] We evaluate PB-SAT with instances arising from industrial design problems, and with instances from the 2010 PB Competition [6]. We used three servers equipped with two 4-core Xeon X5677 (3.47GHz) CPUs each, and 32GB or 96GB of RAM. In Section IV-A, we provide evidence of one of our claimed contributions, *viz.*, that we have improved the state of the art in SAT-based approaches to solving PB problems. In Section IV-B, we provide evidence that SAT-based PB solvers should be part of any portfolio of solvers.

A. Industrial Design Problems

We used our solver with a family of 20 industrial PB instances generated by the CoBaSA design tool [8], where 16 are satisfiable, and 4 unsatisfiable. The instances encode *system assembly problems*: an assignment is a way to assemble system components so that various requirements are met. The basic components in these problems are anywhere from 8 to 22 cabinets that provide resources (including CPU time, memory and networking), about 200 applications that consume resources, and up to 300 memory spaces. Applications and memory spaces have to be mapped to cabinets subject to various constraints, which we are going to briefly describe. See [9] for a detailed description of these problems.

The most important variables in these instances are called *map variables*: $M_{c,p}$ is true iff the resource consumer c (*e.g.*, an application or memory space) is mapped to cabinet p. Each

[2] http://www.ccs.neu.edu/home/vpap/pb-sat.html

	# instances	CPLEX	bsolo	wbo	SAT4J	MS+	PB-SAT	VPS1	VPS2
aardal_1	14	14	14	14	14	14	14	14	14
uclid	50	23	45	44	44	48	49	48	49
tsp	100	90	98	100	100	100	100	100	100
wnqueen	100	97	100	100	100	100	100	100	100
dbst	15	13	15	15	15	15	15	15	15
fpga	57	57	57	39	38	39	39	57	57
armies	12	7	6	6	7	6	8	10	10
pigeon	40	39	21	4	3	3	2	39	39
j{30,60,90,120}	81	66	65	68	68	67	67	68	68
rest	17	10	9	7	9	7	8	13	13
all	486	416	430	397	398	399	402	464	465
average time (sec)	-	135.8	38.0	70.3	67.8	83.1	67.7	-	-

(a) Decision Problems

	# instances	CPLEX	bsolo	wbo	SAT4J	PB-SAT	VPS1	VPS2
feature subscription	20	0	19	10	20	19	20	20
caixa	21	21	21	21	21	21	21	21
j{30,60,90,120}	80	47	52	55	55	55	55	55
area	69	69	25	47	11	22	119	120
logic synthesis	74	71	51	27	24	30	71	71
routing	15	15	15	15	15	15	15	15
primes	156	124	105	105	104	114	127	131
factor	192	192	192	190	192	192	192	192
rest	212	137	100	109	100	72	160	167
all	939	676	580	579	542	540	780	792
average time (sec)	-	30.8	50.2	48.8	25.7	81.3	-	-

(b) Optimization Problems

Fig. 1. Experimental Results: Small Integers, Linear Constraints (timeout after 1800 seconds, 2GB RAM limit)

application j has to reside on exactly one cabinet, so we have cardinality constraints of the form

$$\sum_{p \in P} M_{j,p} = 1,$$

where P is the set of cabinets in the system. We also have resource requirements. For a cabinet p that provides r_p units of the resource r, let C_p be the set of consumers that can be potentially be mapped to p. Each $c \in C_p$ needs r_c units of the resource r. We thus have an inequality

$$\sum_{c \in C_p} r_c M_{c,p} \leq r_p.$$

In addition, we have structural requirements, like co-location or separation of components. These requirements are expressible as propositional constraints. For example, if the applications j_1 and j_2 have to be co-located, for every cabinet p there is a constraint $M_{j_1,p} \iff M_{j_2,p}$. Therefore, the instances contain a balanced mix of propositional and arithmetic constraints.

The ILP and native PB solvers we tried worked very well for this class of problems, unlike existing SAT-based approaches. To understand why, we look at a representative instance in more detail. The best result with MiniSat+ [3] is 91 minutes: CNF generation through sorters takes 80 minutes, and PicoSAT can find a satisfying assignment in 11 minutes. All other translation schemes and different SAT solvers give worse results. For example, we ran MiniSat+ using a BDD encoding for 2 hours, at which point it failed to complete and was using over 80GB of RAM. In contrast, PB-SAT can solve the instance in 32 seconds (21 seconds of PicoSAT time) using BDDs for the translation, with the units extraction mechanism of subsection II-B disabled. The reason is that we learn a significant number of units that allow us to simplify the problem. Constraint propagation reveals 7938 units. Before the 7th and last call to PicoSAT, we know 8240 PB units. Algorithm 2 leads to even more learned units, even with a limit of 10 decisions per query: its execution takes 0.5 seconds, and after its execution we know 8506 PB units. These units improve the running time to 19 seconds. SAT solving accounts for 9 seconds.

The instances have between 14000 and 21000 variables and between 68000 and 93000 constraints. PB-SAT solves all instances, taking 62 seconds on average; MiniSat+ timed out (1800 seconds) for all instances and translation schemes.

B. Pseudo-Boolean Competition Instances

For the sake of completeness, we include experimental results for instances from the PB competition (figure 1). We only provide results for instances with small integers

and linear constraints, because a wide range of solvers is available for these. We compare against bsolo [10], wbo [11], SAT4J [12], CPLEX [13] and MiniSat+. We run the best known configuration of each solver: bsolo with cardinality constraint learning, and the resolution version of SAT4J. In the case of MiniSat+, we generate CNF formulas and run PicoSAT, for a direct comparison with our solver. PB-SAT solves 35 decision instances less than the best solver. The difference can be attributed to hand-crafted instances, some of which contain pigeonhole-like problems (*e.g.*, "pigeon" and "fpga").

The results include two *virtual portfolio solvers*. VPS2 stands for a solver that would run all solvers in parallel and report the best result. VPS1 is VPS2 minus PB-SAT. VPS1 "solves" 30 more decision instances than the best solver, and our solver adds an extra instance to the mix. The combination of PB-SAT and CPLEX [13] solves the same instances as VPS1. The combination of CPLEX and any other solver follows closely (1-3 instances less), while any combination of two without CPLEX solves at most 443 instances. PB-SAT contributes 12 extra optimization instances. According to this analysis, the number of solved instances a solver contributes to a portfolio of solvers is valuable information, due to the diversity of techniques. Translation to SAT is a useful addition.

Interestingly, we do not learn *any* units before the last SAT query for 448 out of the 486 decision instances. These instances either consist entirely of clauses and cardinality constraints, in which case we encode everything at once, or the intermediate formulas do not imply any units. For these problems, our incremental approach obviously does not yield any improvements. These benchmarks problems *are not* characteristic of the industrial problems we have seen, and we encourage the community to contribute industrial PB problems to the PB competition benchmark suite.

V. Related Work

Different encodings of PB constraints into SAT have been proposed. Bailleux et al. [4] describe a variant of the BDD encoding. Een and Sorensson implemented the MiniSat+ PB solver [3], which uses adders, sorters, and BDDs. Bailleux et al. [5] present the first polynomial arc-consistent encoding. Abio et al. [14] revisit BDDs, and provide a polynomial, arc-consistent, BDD-based encoding. In addition, encodings for cardinality constraints (an interesting special case) have been explored (*e.g.*, [15]).

Our algorithm can be viewed from the perspective of lazy SMT [16], as we introduce just enough information for the SAT solver to find a consistent assignment, or prove unsatisfiability. SMT has already been used to tackle PB problems: Cimatti et al. [17] extend the SMT framework with the theory of costs C, and use it to express PB constraints. Our approach differs from SMT in that we actually encode the PB constraints, as opposed to learning a clause that precludes a single theory-inconsistent conjunction of literals.

Our technique also bears resemblance to Abstraction-Refinement, *e.g.*, as applied to the theory of arrays [18]. We abstract the problem by omitting information from the encoding, and then refine the abstraction based on assignments that satisfy the partial encoding but not the PB formula.

VI. Conclusions

We presented an algorithm for pseudo-Boolean solving by incremental translation to SAT, and implemented a solver based on this algorithm. Incrementality allows our solver to use unit literals derived from intermediate SAT queries to simplify pseudo-Boolean constraints. In addition, we learn units from constraint propagation at the pseudo-Boolean level. Experimental evaluation on industrial problems shows that our solver improves the state of the art in SAT-based approaches to pseudo-Boolean problems and that any portfolio solver should include a SAT-based solver.

Acknowledgments

This research is funded in part by NASA Cooperative Agreement NNX08AE37A and NSF proposal CCF-1117184. This article reports on work supported by the Defense Advanced Research Projects Agency under Air Force Research Laboratory (AFRL/Rome) Cooperative Agreement No. FA8750-10-2-0233. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. We would like to thank some of the anonymous reviewers for making helpful suggestions.

References

[1] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," in *DAC*, 2003.
[2] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-boolean SAT solver," *JSAT*, vol. 2, pp. 165–189, 2006.
[3] N. Een and N. Sorensson, "Translating Pseudo-Boolean constraints into SAT," *JSAT*, vol. 2, pp. 1–26, 2006.
[4] O. Bailleux, Y. Boufkhad, and O. Roussel, "A Translation of Pseudo Boolean Constraints to SAT," *JSAT*, vol. 2, pp. 191–200, 2006.
[5] O. Bailleux, Y. Boufkhad, and O. Roussel, "New Encodings of Pseudo-Boolean Constraints into CNF," in *SAT*, 2009.
[6] Vasco Manquinho and Olivier Roussel, "Pseudo-Boolean Competition 2010." See `http://www.cril.univ-artois.fr/PB10/`.
[7] A. Biere, "PicoSAT Essentials," *JSAT*, vol. 4, pp. 75–97, 2008.
[8] P. Manolios, D. Vroon, and G. Subramanian, "Automating component-based system assembly," in *ISSTA*, 2007.
[9] C. Hang, P. Manolios, and V. Papavasileiou, "Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints," in *CAV*, 2011.
[10] V. M. Manquinho and J. Marques-Silva, "On Using Cutting Planes in Pseudo-Boolean Optimization," *JSAT*, vol. 2, pp. 209–219, 2006.
[11] V. Manquinho, J. Marques-Silva, and J. Planes, "Algorithms for Weighted Boolean Optimization," in *SAT*, 2009.
[12] "SAT4J." See `http://www.sat4j.org/`.
[13] "CPLEX." See `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.
[14] I. Abio, R. Nieuwenhuis, A. Oliveras, and E. Rodriguez-Carbonell, "BDDs for Pseudo-Boolean Constraints - Revisited," in *SAT*, 2011.
[15] J. Marques-Silva and I. Lynce, "Towards Robust CNF Encodings of Cardinality Constraints," in *CP*, 2007.
[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *JACM*, vol. 53, no. 6, pp. 937–977, 2006.
[17] A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, and C. Stenico, "Satisfiability Modulo the Theory of Costs: Foundations and Applications," in *TACAS*, 2010.
[18] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *CAV*, 2007.