PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF

Tobias Philipp and Peter Steinke(✉)

Knowledge Representation and Reasoning Group,
Technische Universität Dresden, 01062 Dresden, Germany
peter.steinke@tu-dresden.de

Abstract. PBLib is an easy-to-use and efficient library, written in C++, for translating pseudo-Boolean (PB) constraints into CNF. We have implemented fifteen different encodings of PB constraints. Our aim is to use efficient encodings, in terms of formula size and whether unit propagation maintains generalized arc consistency. Moreover, PBLib normalizes PB constraints and automatically uses a suitable encoder for the translation. We also support incremental strengthening for optimization problems, where the tighter bound is realized with few additional clauses, as well as conditions for PB constraints.

1 Introduction

Many applications such as hardware verification and model checking benefit from the impressive developments in the area of SAT solving by translating high level descriptions into propositional formulas in conjunctive normal form (CNF) [24,30]. Pseudo-Boolean (PB) constraints are expressions of the form $\sum_{i=1}^{n} w_i \cdot x_i \lhd k$ and require that the weighted sum over the literals x_i is $\lhd$-related with k. They frequently occur in scheduling, planning, and translations of problems from languages like CSP, ASP or integer programming. Moreover, optimization problems like MaxSAT, minimal unsatisfiable core extraction, maximal satisfying subformulas, and PB optimization itself, rely on good translations from PB constraints into CNF [4,5,18,21,22]. However, there is no straightforward translation into CNF [6,11,15,16,25].

In this paper, we present *PBLib*, an easy-to-use and efficient library, written in C++, and distributed under the MIT license[1]. The library contains *fifteen different encodings* for PB constraints, which differ in the number of clauses, auxiliary variables and further properties. For instance, *generalized arc consistency* (GAC), a notion developed in the area of constraint programming [25], allows to cut off the search space as soon as possible. Therefore, maintaining generalized arc consistency by unit propagation is an important property of encodings. A weaker property than GAC is that *unit propagation detects inconsistent assignments*. The size of an encoding is another performance criteria, since SAT solvers often perform better when the number of clauses is small [8,19,29]. Additionally,

[1] available at http://tools.computational-logic.org/content/pblib.php

M. Heule and S. Weaver (Eds.): SAT 2015, LNCS 9340, pp. 9–16, 2015.
DOI: 10.1007/978-3-319-24318-4_2

PBLib performs *constraint normalization*, and supports *incremental strengthening* as well as *conditionals*. Experiments have shown that PBLib performs better than minisat+ [10].

The rest of this paper is structured as follows: We formally introduce the concept of encodings and generalized arc consistency in Sect. 2. Afterwards, we describe the concepts in PBLib and present code examples in Sect. 3. In Sect. 4, we give an overview of the tools included in PBLib. Then, we evaluate different encodings and compare PBLib with minisat+ in Sect. 5. Finally, we conclude in Sect. 6.

2 Pseudo-Boolean Constraints and Encodings

We assume that the reader is familiar with the concepts in propositional logic. *Pseudo-Boolean (PB) constraints* are expressions of the form $\sum_{i=1}^{n} w_i \cdot x_i \lhd k$, where x_i are literals, $w_i \in \mathbb{Z}$ are the *associated weights* for the literals x_i for every $i \in \{1, \ldots, n\}$, $k \in \mathbb{Z}$, and $\lhd \in \{=, \leq, \geq\}$ is the *comparator*. A *cardinality constraint* is a PB constraint, where all weights are equal to 1. Depending on the comparator, we call a cardinality constraint an *at-most-one, at-least-one* or *exactly-one* constraint, if $k = 1$.

Formally, the formula F *encodes* the original formula G iff *1)* F entails G, and *2)* for every model I of the formula G there is a model I' of F such that $I(x) = I'(x)$ for every variable x occurring in G. The first condition states that every model of the encoding is a model of the original formula. The second condition states that every model of the original formula can be transformed to a model of the encoding by modifying the interpretation of the auxiliary variables.

We consider the following two structural properties: generalized arc consistency (GAC) and inconsistency detection. Both are important inference rules in constraint programming and can significantly reduce the search space [25]. As in [23], we describe the notions of GAC and inconsistency detection in terms of the entailment relation. An *assignment* J is a consistent set of literals. We say that J is *consistent* w.r.t. a constraint C iff the formula $(\bigwedge_{x \in J} x) \wedge C$ is satisfiable. Otherwise, J is *inconsistent* w.r.t. C. An encoding *detects inconsistencies by unit propagation* if unit propagation in the encoding derives the empty clause, if the assignment J is inconsistent w.r.t. C. Informally, an assignment is GAC, if the assignment contains all entailed literals. Formally, a consistent assignment J is *GAC w.r.t. the constraint* C iff for every variable y occurring in C, $y \in J$ whenever $(\bigwedge_{x \in J} x) \wedge C \models y$, and $\neg y \in J$, whenever $(\bigwedge_{x \in J} x) \wedge C \models \neg y$. *Unit propagation maintains GAC*, if unit propagation transforms a consistent assignment to generalized arc consistent assignment.

3 Description of the PBLib

Table 1 presents the encodings offered by PBLib. Unit propagation in the offered encodings, except sorting and adder networks, detects inconsistent assignment and maintains generalized arc consistency. We also offer a variant of the watchdog

Table 1. A catalog of encodings offered by PBLib, categorized into different fragments of PB constraints.

at-most-one	cardinality	pseudo-Boolean
sequential counter[1] [27]	BDD[2] [10,14]	BDD [10,14]
bimander [13]	cardinality networks [1]	adder networks [10]
commander [17]	adder networks [10]	watchdog [23]
k-product [7]		sorting networks [10]
binary [2]		binary merge [20]
pairwise		sequential weight counter[3] [12]
nested		

[1] similar to BDD, latter and regular encoding,[2] similar to sequential counter
[3] similar to BDD but useful for incremental encoding

and binary merge encoding for which unit propagation detects only inconsistent assignments, with the advantage of fewer clauses.

3.1 Components of the PBLib

PB constraints. In the PBLib, a PB constraint $\sum_{i=1}^{n} w_i \cdot x_i \lhd k$ is specified with a list of weighted literals, a comparator and an integer k, where every 64 bit integer is accepted as weight and as k. The comparator can be either less equal, greater equal, or a combination of both. Hence it is possible to specify a single constraint like $\sum_{i=1}^{n} w_i \cdot x_i \leq k_1 \wedge \sum_{i=1}^{n} w_i \cdot x_i \geq k_2$. Note that GAC and inconsistency detection refers to single PB constraint using $\leq$ or $\geq$ as comparator.

PreEncoder. The *PreEncoder* normalizes PB constraints such that the following holds: 1. $n > 0$, 2. $1 \leq w_i \leq k$ for every $i \in \{1, \ldots, n\}$, 3. no literal in a constraint occurs twice, and 4. the comparator is either less equal or both: less equal and greater equal. Moreover, it detects trivial constraints such as units and tautologies, directly encodes them, and applies some simplifications such as removing unnecessary comparators.

ClauseDatabase. As container for the clauses in a formula a *ClauseDatabase* is used. The PBLib contains different instances of ClauseDatabases such as a *VectorClauseDatabase* that stores each clause in a vector, and a *SATSolverClauseDatabase* that stores each clause in a minisat-like [9] SAT solver. The ClauseDatabase is a simple interface, requiring only an implementation for the addition method for single clauses. This makes it easy to integrate PBLib in projects. Moreover, every ClauseDatabase can process minisat+ like Boolean circuits [10] by translating them into clauses.

AuxVarManager. For handling auxiliary variables, PBLib uses an auxiliary variable manager, called *AuxVarManager*. Initialized with a fresh variable, AuxVarManager returns the next free variable upon request. It is possible to reset already used auxiliary variables as well as marking individual variables as fresh variables. Hence the AuxVarManager helps to keep the set of used variables tight.

Encoder. The PBLib contains 15 different encoders, where each produces different clause sets. Some encoders are only applicable for specific subsets of PB constraint, e.g. at-most-one or cardinality constraints. In the framework of the PBLib, it is easy to extend the set of encoders with new encodings.

IncrementalData. It is required to use the class *IncPBConstraint* to represent PB constraints that supports incrementally strengthening. After the initial encoding of such a constraint, the *IncPBConstraint* stores *IncrementalData* internally that allows to restrict the constraint with a tighter bound. This allows the implementation of an easy to handle SAT-based linear optimization algorithm.

Conditionals. PB and incremental PB constraints can be augmented with *conditions*, i.e. finite conjunctions of literals. This is achieved by adding the complementary literals to all activation clauses of the encoding. For example, we can express the following constraint with a single constraint in PBLib:

$$(x_5 \wedge \overline{x_6}) \rightarrow (-3 \leq -7x_1 + 5\overline{x_2} + 9\overline{x_3} - 3\overline{x_{10}} + 7x_{10} \leq 8)$$

PB2CNF. The *PB2CNF* class handles constructed PB constraints: It normalizes the constraint, classifies it, and chooses a suitable encoding depending on the kind and size of the constraint. Produced clauses are stored in the given ClauseDatabase and auxiliary variables are managed by the AuxVarManager.

3.2 Example

We demonstrate how to encode the constraint $3\overline{x_1} - 2\overline{x_2} + 7x_3 \geq -4$ in the following example. First, we reserve space for two vectors containing the decision literals and their associated weights, and for the resulting formula, which is a vector of vectors of literals. Moreover, we specify the first free variable in `firstAuxVar`. Finally, we call the method `encodeGeq` that encodes the constraint and stores the result in `formula`.

```
#include "PB2CNF.h"
int main() {
  PBLib::PB2CNF pb2cnf;
  vector< int64_t > weights = {3, -2, 7};
  vector< int32_t > literals = {-1, -2, 3};
  vector< vector< int32_t > > formula;
  int32_t firstAuxVar = 4;
  int64_t k = -4;
  pb2cnf.encodeGeq(weights, literals, k, formula, firstAuxVar);
}
```

You can also add a less equal and a greater equal comparator, as well as incremental constraints. For the latter one, we need the generic formula container ClauseDatabase and an instance of AuxVarManager. Moreover, we have to use the configurations class of the PBLib. In the following example, the constraint $-5 \leq -7x_1 + 5\overline{x_2} + 9\overline{x_3} - 3\overline{x_{10}} + 7x_{10} \leq 100$ is encoded:

```
using namespace PBLib;
PBConfig config = make_shared< PBConfigClass >();
VectorClauseDatabase formula(config);
PB2CNF pb2cnf(config);
AuxVarManager auxvars(11);
vector< WeightedLit > literals =
  {WeightedLit(1, -7), WeightedLit(-2, 5), WeightedLit(-3, 9),
   WeightedLit(-10, -3), WeightedLit(10, 7)};
IncPBConstraint constraint(literals, BOTH, 100, -5);
pb2cnf.encodeIncInital(constraint, formula, auxvars);
```

We can increase the bounds:

```
constraint.encodeNewGeq(3, formula, auxvars);
constraint.encodeNewLeq(8, formula, auxvars);
```

The constraint $-3 \leq -7x_1 + 5\overline{x_2} + 9\overline{x_3} - 3\overline{x_{10}} + 7x_{10} \leq 8$ is encoded with the code above in combination with the formula encoded with encodeIncInital.

4 Included Tools

The PBLib includes the following programs: *pbencoder*, *pbsolver* and a *fuzzer*. *pbencoder* takes as input a list of PB constraints in the OPB format [26] and encodes them into CNF. The result is printed on the standard output. *pbsolver* solves a OPB instance by translating the PB constraints and afterwards solving the resulting CNF formula with a back-end SAT solver such as *minisat 2.2* [28]. For optimization instances, pbsolver iteratively encodes upper bounds until the optimum is reached. The program *fuzzer* randomly generates PB constraints, and afterwards encode them with different configurations. This program helps to find bugs in new or customized implementations.

5 Empirical Evaluation and Related Work

We evaluated *pbsolver* on all *new* instances in the PB competition 2012, in total 2782 instances[2]. Note that the most recent PB competition was held in the year 2012. Besides various encodings inside the PBLib, we compared the performance of *minisat+* [10] on this benchmark. The evaluation was performed on a PC cluster with Intel E5-2690 CPUs (2.90 GHz) and 2 GB RAM.

minisat+ follows the same approach for solving PB constraints: It translates them into CNF and applies an iterative solving strategy for optimization problems. In contrast to PBLib, minisat+ uses only three encodings: BDDs, sorting networks and adder networks. BDDs in minisat+ are encoded with three clauses per BDD node instead of only two as in the PBLib and presented in [14]. For a fair comparison, we used minisat 2.2 as back-end SAT solver in pbsolver and in minisat+. Figure 1 shows the results of the evaluation. Adder and sorting

[2] available at http://www.cril.univ-artois.fr/PB12/

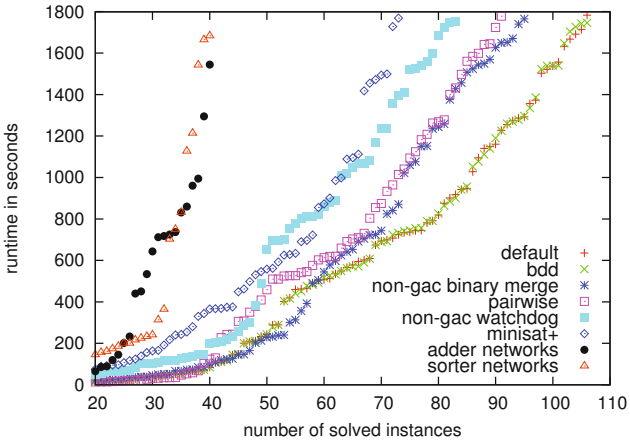

Fig. 1. A comparison of different encodings in the PBLib and minisat+ using 2782 new instances in the PB competition 2012

networks have the least number of clauses, but the worst runtime, because in both encodings unit propagation neither detects inconsistent assignments nor maintains generalized arc consistency. We observe that the plot for the default configuration and the use of BDDs for general PB constraint are nearly the same. This can be explained since PBLib decides for the BDD encoding due to the low number of clauses. In contrast to minisat+, PBLib in pbsolver solves significantly more instances in the timeout of 1800 seconds. This has two reasons: First, PBLib uses a better BDD encoding. Second, minisat+ does not distinguish between at-most-one, at-most-k and PB constraint. Therefore, all constraints, but clauses, are handled in the same way. Instead, PBLib detects these special cases and chooses a more appropriate encoding.

The Java library Boolvar/PB [3] is also related to the PBLib: It uses basically the same encodings as minisat+, but sorting networks have been replaced by the watchdog encoding.

6 Conclusion

In this paper, we presented PBLib, an efficient and easy-to-use library for encoding PB constraints into clause sets. It normalizes PB constraints before encoding them, and can automatically choose between fifteen different encodings that vary in size and propagation properties. Moreover, PBLib supports incremental strengthening and conditional PB constraints. Experiments have shown that our library outperforms minisat+ in the recent benchmark of the PB evaluation. It is distributed under the MIT license.

In future, we plan to implement more encodings, and to increase confidence in the tools by mechanically verifying them.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A parametric approach for smaller and better encodings of cardinality constraints. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 80–96. Springer, Heidelberg (2013)
2. Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving non-Boolean satisfiability problems with stochastic local search: A comparison of encodings. Journal of Automated Reasoning **35** (2005)
3. Bailleux, O.: Boolvar/PB 1.0 (2012). http://boolvar.sourceforge.net/
4. Belov, A., Janota, M., Lynce, I., Marques-Silva, J.: On computing minimal equivalent subformulas. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 158–174. Springer, Heidelberg (2012)
5. Boros, E., Hammer, P.L.: Pseudo-Boolean optimization. Discrete Applied Mathematics **123**(1–3), 155–225 (2002)
6. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12), 92–103 (2011)
7. Chen, J.: A new SAT encoding of the at-most-one constraint. In: ModRef 2010 (2010)
8. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
10. Een, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**, 1–26 (2006)
11. Großmann, P., Hölldobler, S., Manthey, N., Nachtigall, K., Opitz, J., Steinke, P.: Solving periodic event scheduling problems with SAT. In: Jiang, H., Ding, W., Ali, M., Wu, X. (eds.) IEA/AIE 2012. LNCS, vol. 7345, pp. 166–175. Springer, Heidelberg (2012)
12. Hölldobler, S., Manthey, N., Steinke, P.: A compact encoding of pseudo-boolean constraints into SAT. In: Glimm, B., Krüger, A. (eds.) KI 2012. LNCS, vol. 7526, pp. 107–118. Springer, Heidelberg (2012)
13. Hölldobler, S., Nguyen, V.H.: On SAT-encodings of the at-most-one constraint. In: ModRef 2013 (2013)
14. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: BDDs for pseudo-boolean constraints – revisited. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 61–75. Springer, Heidelberg (2011)
15. Kautz, H., Selman, B.: Planning as satisfiability. In: Neumann, B. (ed.) ECAI 1992, pp. 359–363. John Wiley & Sons Inc., New York (1992)
16. Kautz, H., Selman, B.: Pushing the envelope: planning, propositional logic, and stochastic search. In: AAAI 1996, pp. 1194–1201. MIT Press (1996)
17. Klieber, W., Kwon, G.: Efficient CNF encoding for selecting 1 from n objects. In: CFV 2007 (2007)
18. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1), 1–33 (2008)
19. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC. LNCS, vol. 7857, pp. 102–117. Springer, Heidelberg (2013)

20. Manthey, N., Philipp, T., Steinke, P.: A more compact translation of pseudo-boolean constraints into CNF such that generalized arc consistency is maintained. In: Lutz, C., Thielscher, M. (eds.) KI 2014. LNCS, vol. 8736, pp. 123–134. Springer, Heidelberg (2014)
21. Manthey, N., Steinke, P.: npSolver - a SAT based solver for optimization problems. In: POS 2012 (2012)
22. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Bloem, R., Sharygina, N. (eds.) FMCAD 2010, pp. 121–128 (2010)
23. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-boolean constraints into CNF. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 181–194. Springer, Heidelberg (2009)
24. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2**(3), 293–304 (1986)
25. Rossi, F., Beek, P.V., Walsh, T.: Handbook of Constraint Programming. Elsevier Science Inc., New York (2006)
26. Roussel, O., Manquinho, V.: Input/output format and solver requirements for the competitions of pseudo-Boolean solvers (2012)
27. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
28. Sorensson, N., Een, N.: MiniSat - a SAT solver with conflict-clause minimization. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS. Springer, Heidelberg (2005)
29. Subbarayan, S., Pradhan, D.K.: NiVER: non-increasing variable elimination resolution for preprocessing SAT instances. In: H. Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 276–291. Springer, Heidelberg (2005)
30. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning. Symbolic Computation, pp. 466–483. Springer, Heidelberg (1983)