PBLib – A C++ Toolkit for Encoding Pseudo-Boolean Constraints into CNF

Peter Steinke
peter.steinke@tu-dresden.de

February 23, 2015

Abstract
Many different encodings for PB constraints into conjunctive normal form (CNF) have been proposed in the past. The PBLib project starts to collect and implement these encodings to be able to encode PB constraints in a very simple, but effective way. The aim is not only to generate as few clauses as possible, but also using encodings that maintain generalized arc consistency by unit propagation, to speedup the run time of the SAT solver, solving the formula.

A major issue of the implementation is a high flexibility for the user. Consequently it is not required to bring a PB constraint into a certain normal form. The PBLib automatically normalizes the constraints and decides which encoder provides the most effective translation.

The user can also define constraints with two comparators (less equal and greater equal) and each PB constraint can be encoded in an incremental way: After an initial encoding it is possible to add a tighter bound with only a few additional clauses. This mechanism allows the user to develop SAT-based solvers for optimization problems with incremental strengthening and to keep the learned clauses for incremental SAT solver calls.

1 Implemented Encodings

Table 1 shows the encodings are currently implemented in the PBLib. The label todo denotes encodings that are planned for the (near) future.

2 Overview

The overview in Figure 1 shows the sequence of encoding a PB constraint. A PB constraint is given to the PB2CNF class, where the constraint is simplified and normalized by the PreEncoder. Next PB2CNF selects a suitable encoder for the simplified constraint. The clauses generated by the encoder are added to a clauses database and auxiliary variables are provided by an instance of the AuxVarManager class.

Figure 2 shows the encoding process of an incremental constraint. After this initial encoding the user can encode a tighter bound with the incremental data (without calling PB2CNF again), using the information that the initial bound is already in the clause database.
At most one  At most K  PB
sequential[11]**  BDD[7, 4]**
pairwaise  binary merge[9]
nested  sequential weight counter[5]

Table 1: Implemented encodings

* equivalent to BDD, latter and regular encoding
** equivalent to sequential counter

PB constraint \[\xrightarrow{\text{PB2CNF}}\] PreEncoder \[\xrightarrow{\text{Simple PB Constraint}}\]
ClauseDatabase \[\xleftarrow{\text{Encoder}}\] auxVarManager

Figure 1: Encoding a PB constraint to CNF

Inc. PB constraint \[\xrightarrow{\text{PB2CNF}}\] PreEncoder \[\xrightarrow{\text{Simple PB Constraint}}\]
ClauseDatabase \[\xleftarrow{\text{Encoder}}\] auxVarManager
IncrementalData

Figure 2: Encoding an incremental PB constraint to CNF
3 Using the PBLib

3.1 Including the PBLib

The following includes are needed for the PBLib:

```c
#include "pb2cnf.h"  // encoding interface
```

Optional includes are:

```c
#include "VectorClauseDatabase.h"  // basic clause database
#include "PBParser.h"  // parser for opb files
#include "SATSolverClauseDatabase.h"  // a clause database that adds all clauses directly to a SAT solver (minisat)
```

The project has to be linked against the PBLib (libpblib.a or libpblib.so). Note that PBLib uses the C++11 standard.

3.2 PB Constraints

For an instance of a PB constraint a vector of weighted literals is used:

```c
WeightedLit (int32_t literal, int64_t weight);
```

In addition to the weighed literals a comparator is needed. This could be less equal, greater equal or a combination of both.

```c
enum Comparator { LEQ, GEQ, BOTH };
```

Depending on the comparator one or two bounds are needed:

```c
PBConstraint (vector< WeightedLit > const & literals, Comparator comparator, int64_t bound);
```

```c
PBConstraint (vector< WeightedLit > const & literals, Comparator comparator, int64_t less_eq, int64_t greater_eq);
```

// in case comparator == BOTH and less_eq != greater_eq

3.3 Incremental PB Constraints

In addition to the PB constraints presented above, the incremental version allows an incremental encoding of a sequence of tighter bounds. Note that an incremental PB constraint is not a subclass of PB constraint (in the sense of object-oriented programming).

After the initial encoding of an incremental PB constraint `incPbConstraint`, it is possible to encode new (tighter) bounds with the methods:

```c
incPbConstraint.encodeNewGeq (int64_t newGeq, ClauseDatabase & formula, AuxVarManager & auxVars);
```

```c
incPbConstraint.encodeNewLeq (int64_t newLeq, ClauseDatabase & formula, AuxVarManager & auxVars);
```

The clause database `formula` has to contain the initial encoding of `incPbConstraint`. 
3.4 Clause Database

The clause database is a container class for clauses – the CNF formula. Every implementation of this class provides the following methods:

addClause (vector<int32_t> const & clause);

The PBLib contains one default implementation of this class: VectorClauseDatabase. Every clause that is added is saved into a vector of clauses.

3.5 Auxiliary Variable Manager

The Auxiliary Variable Manager returns fresh variables to the encoder. Therefore it is initialized with the first fresh variable. Hence it is assumed that all variables in the original constraints are smaller than this first fresh variable.

AuxVarManager (int32_t first_free_variable);

Every getVariable call returns the successor variable of the last returned variable. With the method int32_t getBiggestReturnedAuxVar() the biggest returned variable can be obtained. Hence every variable between this (including) number and first_free_variable (probably) occurs in some clause database.

3.6 Encoding PB constraints into CNF

The class PB2CNF is the main interface to encode an (incremental) PB constraint.

PB2CNF(PBConfig & config);

Where config is a configuration object explained in the next subsection. If config is omitted the default configuration will be used.

For a simple interface you can now use one of the following methods:

int32_t encodeLeq
(  
  vector<int32_t> & weights,  
  vector<int32_t> & literals,  
  int64_t leq,  
  vector<vector<int32_t>> & formula,  
  int32_t firstAuxiliaryVariable  
);  
int32_t encodeGeq(...);
int32_t encodeBoth(...);

Where the return values is the last used auxiliary variable. Hence the next integer value is the next free variable.

For a more sophisticated interface you should use the ClauseDatabase and the AuxVarManager. Besides the constraint, a clause database and an auxiliary variable manager is needed to encode a PB constraint with the encode method:

encode(PBConstraint const & pbconstraint, ClauseDatabase & formula, AuxVarManager & auxVars);

encodeIncInitial(IncPBConstraint & incPbconstraint, ClauseDatabase & formula, AuxVarManager & auxVars);
3.7 Configuration

An instance of the PBConfig class contains the configuration for all options in the PBLib. Since PBConfig is a shared pointer you have to initialize it in the following way:

```cpp
PBConfig config = make_shared<PBConfigClass>();
```

The following options (with the given default values) are currently available:

```cpp
PB2CNF_PB_Encoder pb_encoder = PB_ENCODER::BEST;
PB2CNF_AMK_Encoder amk_encoder = AMK_ENCODER::BEST;
PB2CNF_AMO_Encoder amo_encoder = AMO_ENCODER::BEST;
BIMANDER_M_IS bimander_m_is = BIMANDER_M_IS::N_HALF;
int bimander_m = 3;
int k_product_minimum_lit_count_for_splitting = 10;
int k_product_k = 2;
int commander_encoding_k = 3;
int64_t MAX_CLAUSES_PER_CONSTRAINT = 1000000;
bool use_formula_cache = false;
bool print_used_encodings = false;
bool check_for_dup_literals = false;
bool use_real_robdds = true;
bool use_watch_dog_encoding_in_binary_merger = false;
```

```cpp
enum PB2CNF_AMO_Encoder {BEST, NESTED, BDD,
BIMANDER, COMMANDER,
KPRODUCT, BINARY, PAIRWISE};
enum PB2CNF_AMK_Encoder {BEST, BDD, CARD};
enum PB2CNF_PB_Encoder {BEST, BDD, SWC, SORTINGNETWORKS,
ADDER, BINARY_MERGE};
enum BIMANDER_M_IS {N_HALF, N_SQRT, FIXED};
```

Note that if the maximum number of clauses per constraint is (approximately) exceeded, the adder encoding is used as a fallback.

4 PB encoder

The PBLib also includes a PBEncoder which takes an opb input file and translates it into CNF using the PBLib:

```bash
usage ./pbencoder inputfile
```

5 Example PB Solver

The PBLib source code contains also a folder BasicPBSolver with the implementations of an example PB Solver. It uses minisat 2.2 as a back-end SAT solver.

```bash
usage ./pbsolver inputfile
```
6 PB Fuzzer

Included is also a PB fuzzer. The program generates a random PB constraint that is solves it with different configuration within the PBLib.

This program helps to find bugs in new or customized implementations of the PBLib.

usage: ./fuzzer

References


