PyPBLib Documentation

Release 0.2

Logic Optimization Group

Sep 14, 2018

CONTENTS:

CHAPTER

ONE

PYTHON PSEUDO-BOOLEAN LIBRARY

This library offers a comprehensive set of utilites to work with pseudo-boolean formulas within Python.

This module provides bindings to the PBLib – A C++ Toolkit for Encoding Pseudo-Boolean Constraints into CNF.

Note:

You can build the pypblib via:

```
python3 setup.py build
```

You can install the pypblib via:

```
python3 setup.py install
```

1.1 Overview

1.1.1 Pseudo-Boolean Constraint

PBLib provides classes that allows us to encode pseudo-Boolean constraints.
The first to know is the WeightedLit class.

Class WeightedLit

The *WeightedLit* class is used to represent the literals of a pseudo-Boolean constraint, including their weight.
If we take the following pseudo-Boolean constraint as an example:

2 x1 + 3 x2 + 1 x3 <= 3

We can represent their literals as follows:

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 3)
>>> wl3 = pblib.WeightedLit(3, 1)
>>> print(wl1, wl2, wl3)
(1, 2) (2, 3) (3, 1)
```

An operator (inequality) is also required to represent the constraint.

Comparator

The Comparator represents the inequality symbols. It is a set of constants. You can treat the comparator as it were an enum: { LEQ, GEQ, BOTH }

- LEQ (less-equal)
- GEQ (greater-equal)
- BOTH (less-equal & greater-equal)

Now we have everything we need to instantiate a constraint.

Class PBConstraint

The *PBConstraint* class is used to represent a pseudo-Boolean constraint. Receives a list of WeightedLit, a comparator and a bound.
Following the previous example we can generate a constraint as follows:

```
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], pblib.LEQ, 3)
>>> constr
+2x1+3x2+x3<=3
```

However, depending on the comparator, two limits may be necessary. When using the BOTH comparator, a single limit is equivalent to strict equal. Therefore, in case working with a constraint such as the following,

1<=+2x1+3x2+x3<=3

it is necessary to use the BOTH comparator and two bounds. The Pblib also provides a constructor for the PBConstraint in these cases. This takes a list of WeightedLit, a comparator, the less-equal bound and the greater-equal bound.

```
>>> constr_2 = pblib.PBConstraint([wl1, wl2, wl3], pblib.BOTH,
                                  3, 1)
>>> constr_2
1<=+2x1+3x2+x3<=3
```

class PBConfig

An intance of the *PBConfig* class contains the configuration for all options in the PBLib.
For more information about the default configuration and all configuration options, you can see all details *here*.

class Pb2cnf

The class Pb2cnf is the main interface to encode a pseudo-boolean constraint.
The Pb2cnf constructor takes a PBConfig as a parameter. However, if PBConfig is omitted the default configuration will be used.

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

encode_at_most_k (literals: [int], k: long, formula: [], first_free_variable: int) -> int

encode_at_least_k (literals: [int], k: long, formula: [], first_free_variable: int) -> int

encode_leq (weights: [long], literals: [int], k: long, formula: [], first_free_variable: int) -> int

encode_geq (weights: [long], literals: [int], k: long, formula: [], first_free_variable: int) -> int

encode_both (weights: [long], literals [int], leq: long, geq: long, formula: [], first_free_variable: int) -> int

But for a more sophisticated interface you should use, besides the PBConstraint, a VectorClauseDatabase and an AuxVarManager.

If your encode has only one constraint, you can use one of the methods listed above. However, if you have several constraints, you will need to use the following method:

encode (constr: PBConstraint, formula: VectorClauseDatabase, aux_var_manager: AuxVarManager)

class AuxVarManager

The constructor of *AuxVarManager* class takes an integer as parameter.
The auxiliary variable manager, returns fresh variable to the encoder. Therefore it must be initialized with the first fresh variable.

class VectorClauseDatabase

The constructor of *VectorClauseDatabase* class takes an *PBConfig* as parameter.
The clause database is a container of clauses. Every clause that is added is saved into a vector of clauses.

If we take the followings pseudo-Boolean constraints as example (this is a dummy instance; only for demonstration purposes):

1 x1 + 1 x2 + 2 x3 <= 2
1 x4 + 2 x5 - 1 x6 >= 1

The following is a encoding full example:

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> constr_1 = pblib.PBConstraint([wl1, wl2, wl3], pblib.LEQ, 2)
>>> wl4 = pblib.WeightedLit(4, 1)
>>> wl5 = pblib.WeightedLit(5, 2)
>>> wl6 = pblib.WeightedLit(6, -1)
>>> constr_2 = pblib.PBConstraint([wl4, wl5, wl6], pblib.GEQ, 1)
>>> config = pblib.PBConfig()
>>> pb2 = pblib.Pb2cnf()
>>> aux_var = pblib.AuxVarManager(7)
>>> formula = pblib.VectorClauseDatabase(config)
>>> pb2.encode(constr_1, formula, aux_var)
>>> formula
Num. Causes: 12
==============================
7 0
-7 8 0
-2 8 0
-7 -2 9 0
-1 8 0
-7 -1 9 0
-2 -1 9 0
-7 -2 -1 10 0
-3 11 0
-9 11 0
-3 -9 12 0
-12 0
==============================
```

```
>>> pb2.encode(constr_2, formula, aux_var)
>>> formula
Num. Causes: 24
==============================
```

(continues on next page)

(continued from previous page)

```
7 0
-7 8 0
-2 8 0
-7 -2 9 0
-1 8 0
-7 -1 9 0
-2 -1 9 0
-7 -2 -1 10 0
-3 11 0
-9 11 0
-3 -9 12 0
-12 0
13 0
-13 14 0
-6 14 0
-13 -6 15 0
4 14 0
-13 4 15 0
-6 4 15 0
-13 -6 4 16 0
5 17 0
-15 17 0
5 -15 18 0
-18 0
==============================
```

1.1.2 Incremental Pseudo-Boolean Constraint

In this part we will assume that you already know some of the basic PBLib classes that are used to generate the encoding of a pseudo-Boolean constraint. These classes, *WeightedLit* , Comparator , *PBConfig*, *Pb2cnf* , *AuxVarManager* and *VectorClauseDatabase* , are also used to encoding incremental pseudo-boolean constraints. Hence, if you are not familiar with these classes, go to *Pseudo-Boolean Constraint* before continuing to read this part.

Class IncPBConstraint

The IncPBConstraint behaves very much like PBConstraint. The main difference is that it allows, when the coding is already generated, to make it more restrictive. To do this, instead of using the encode(...) method of Pb2cnf class, we will use this other method:

encode_inc_initial (inc_constr: IncPBConstraint, formula: VectorClauseDatabase, aux_var_manager: AuxVarManager)

When we already have the encoding, we can increase the restriction using the methods provided by the IncPBConstraint class, depending on the case:

encode_new_geq (geq_bound: long, formula: VectorClauseDatabase, aux_var_manager: AuxVarManager)

encode_new_leq (leq_bound: long, formula: VectorClauseDatabase, aux_var_manager: AuxVarManager)

Let's see an example:

1 x1 + 1 x2 + 1 x3 <= 2

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, 1)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3], pblib.LEQ, 2)
>>> config = pblib.PBConfig()
>>> pb2 = pblib.Pb2cnf()
>>> aux_var = pblib.AuxVarManager(4)
>>> formula = pblib.VectorClauseDatabase(config)
>>> pb2.encode_inc_initial(i_constr, formula, aux_var)
>>> formula
Num. Causes: 8
==============================
-3 4 0
-2 4 0
-3 -2 5 0
-1 4 0
-3 -1 5 0
-2 -1 5 0
-3 -2 -1 6 0
-6 0
==============================
>>>
>>> i_constr.encode_new_leq(1, formula, aux_var)
>>> formula
Num. Causes: 9
==============================
-3 4 0
-2 4 0
-3 -2 5 0
-1 4 0
-3 -1 5 0
-2 -1 5 0
-3 -2 -1 6 0
-6 0
-5 0
==============================
```

1.1.3 Example, from OPB to CNF file

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import re

from pypblib import pblib


#
################################################################################

PBLIB_OPS = {'<=': pblib.LEQ, '>=': pblib.GEQ, '=': pblib.BOTH}

HEADER_RE = re.compile(r'(.*)\s*#variable=\s*(\d+)\s*#constraint=\s*(\d+)\s*')
TERM_RE = re.compile(r'([+-]?\d)\s+x(\d+)')
IND_TERM_RE = re.compile(r'([>=|<=|=]+)\s+([+-]?\d+)')


#
################################################################################

def transform_pb_to_cnf(pb_formula_path: str):
    config = pblib.PBConfig()
    aux = pblib.AuxVarManager(1)
    pb2cnf = pblib.Pb2cnf(config)
    formula = pblib.VectorClauseDatabase(config)

    with open(pb_formula_path, 'r') as f:
        m = HEADER_RE.match(f.readline())
        aux.reset_aux_var_to(int(m.group(2)) + 1)

        reader = (l for l in map(str.strip, f) if l and l[0] != '*')
        for line in reader:
            wl = [pblib.WeightedLit(int(l), int(w))
                  for w, l in TERM_RE.findall(line)]
            op, ind_term = IND_TERM_RE.search(line).groups()

            constraint = pblib.PBConstraint(wl, PBLIB_OPS[op], int(ind_term))
            pb2cnf.encode(constraint, formula, aux)
    return formula, aux


#
################################################################################

if __name__ == '__main__':
    f, aux = transform_pb_to_cnf(sys.argv[1])
    f.print_formula("test.cnf", aux.get_biggest_returned_auxvar())
```

1.1.4 Example, from OPB to WCNF file

```
import sys
import re
from pypblib import pblib

#
################################################################################

DEFAULT_FILE_NAME = "test_wcnf.cnf"
op = {'<=': pblib.LEQ, '>=': pblib.GEQ, '=': pblib.BOTH }
_sum_soft_weights = 1

#
################################################################################

def read_opb(path):

    config = pblib.PBConfig()
    aux = pblib.AuxVarManager(1)
    hard = pblib.VectorClauseDatabase(config)
    soft = []
    pb2 = pblib.Pb2cnf(config)

    f = open(path,'r')
    for line in f:

        if len(line) == 0: continue
        if '#variable=' in line:
            aux.reset_aux_var_to(int(re.findall(r"\d+", line)[0]) + 1)
        if line[0] == '*': continue
        if 'min:' in line:
            soft = re.findall(r"[-\d]+", line)
            continue

        # This is linear equation
        lq = re.findall(r"[-\d|>=|<=|=]+", line)
        wl = [pblib.WeightedLit(int(lq[i+1]), int(lq[i])) for i,e in enumerate(lq[:-
↪2]) if(i%2 == 0)]

        pb2.encode(pblib.PBConstraint(wl, op[lq[-2]], int(lq[-1])), hard, aux)

    f.close()

    return aux, soft, hard
#
################################################################################

def write_cnf(file_path, aux, soft, hard):

    with open(file_path, 'w') as f:
        f.write("p wcnf " + str(aux.get_biggest_returned_auxvar()) + " " + str(hard.
↪get_num_clauses() + int(len(soft)/2)) \
                + " " + str(sum(int(e) for i,e in enumerate(soft) if(i%2 != 0)) + 1) +
↪'\n')
        for i, e in enumerate(soft):
            if(i%2 == 0):
```

(continues on next page)

(continued from previous page)

```
                global _sum_soft_weights
                _sum_soft_weights += abs(int(soft[i]))
                if(int(soft[i]) < 0):
                    f.write(str(abs(int(soft[i]))) + " " + str(int(soft[i+1])) + " 0\n
↪")
                else:
                    f.write(str(abs(int(soft[i]))) + " " + str(-int(soft[i+1])) + "␣
↪0\n")

        v_form = hard.get_clauses()

        for c in v_form:
            tmp = ""
            for i in c:
                tmp += str(i);
            f.write(str(_sum_soft_weights) + " " + tmp + " 0\n")

    f.close()
#
##############################################################################


if __name__ == '__main__':

    file_path = DEFAULT_FILE_NAME
    if len(sys.argv) > 2:
        file_path = argv[2]

    aux, soft, hard = read_opb(sys.argv[1])
    write_cnf(file_path, aux, soft, hard)
```

1.2 Index

1.2.1 Class WeightedLit

Constructor

class WeightedLit (*literal: [int], weight: [long]*)

Creates an object to represents a literal and its associated weight in a Pseudo-Boolean formula.

```
>>> import pypblib.pblib
>>> wl = pypblib.pblib.WeightedLit(2, 3)
>>> print(wl)
(2, 3)
```

Method summary

Return Type	Method
bool	*comp_variable_asc* (wl1: WeightedLit, wl2: WeightedLit)
bool	*comp_variable_des* (wl1: WeightedLit, wl2: WeightedLit)
bool	*comp_variable_des_var* (wl1: WeightedLit, wl2: WeightedLit)

Method details

comp_variable_asc (*wl1: WeightedLit*, *wl2: WeightedLit*) → bool

Returns true if the weight of first WeightedLit is bigger than the weight of the second. Returns false otherwise.

```
>>> from pypblib.pblib import WeightedLit
>>> wl1 = WeightedLit(1, 2)
>>> wl2 = WeightedLit(2, 1)
>>> wl1.comp_variable_asc(wl1, wl2)
False
>>> wl1.comp_variable_asc(wl2, wl1)
True
```

comp_varialbe_des (*wl1: WeightedLit*, *wl2: WeightedLit*) → bool

Returns true if the weight of first WeightedLit is smaller than the weight of the second. Returns false otherwise.

```
>>> from pypblib.pblib import WeightedLit
>>> wl1 = WeightedLit(1, 2)
>>> wl2 = WeightedLit(2, 1)
>>> wl1.comp_variable_des(wl1, wl2)
True
>>> wl1.comp_variable_des(wl2, wl1)
False
```

comp_variable_des_var (*wl1: WeightedLit*, *wl2: WeightedLit*) → bool

Returns true if absolute value of literal's first WeightedLit is smaller than absolute value of the literal's second. Returns false otherwise.

```
>>> from pypblib.pblib import WeightedLit
>>> wl1 = WeightedLit(1, 2)
>>> wl2 = WeightedLit(-2, 2)
>>> wl1.comp_variable_des_var(wl1, wl2)
False
>>> wl1.comp_variable_des_var(wl2, wl1)
True
```

1.2.2 Class PBConstraint

Constructor

PBConstraint(weightedlits: [*WeightedLits*], comparator, bound: int)

Creates an object to represents a Pseudo-Boolean constraint.

```
>>> import pypblib.pblib
>>> wl_list = [pypblib.pblib.WeightedLit(1,3),
               pypblib.pblib.WeightedLit(2,1),
               pypblib.pblib.WeightedLit(3,5)]
>>> constr = pypblib.pblib.PBConstraint(wl_list, pypblib.pblib.GEQ, 1)
>>> constr
+3x1+x2+5x3>=1
```

```
>>> from pypblib import pblib
>>> w1 = pblib.WeightedLit(1, 2)
>>> w2 = pblib.WeightedLit(2, 1)
>>> w3 = pblib.WeightedLit(3, -2)
>>> constr = pblib.PBConstraint([w1, w2, w3], pblib.LEQ, 1)
>>> constr
+2x1+x2-2x3<=1
```

PBConstraint(weightedlits: [*WeightedLits*], comparator, bound_geq: int, bound_leq: int)

Creates an object to represents a Pseudo-Boolean constraint, in case pblib.BOTH comparator and bound less-equal and greater-equal are different.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> comp = pblib.BOTH
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], comp, 2, 0)
>>> constr
0<=+2x1+x2-1x3<=2
```

Methods summary

Return Type	Method
void	*add_conditional(cond: int)*
void	*add_conditionals(conds: [int])*
[int]	*get_conditionals()*
void	*clear_conditionals()*
comparator	*get_comparator()*
long	*get_leq()*
long	*get_geq()*
long	*get_min_sum()*
long	*get_max_sum()*
int	*get_n()*
[*WeightedLit*]	*get_weighted_literals()*
PBConstraint	*get_geq_constraint()*
PBConstraint	*get_leq_constraint()*
void	*set_comparator* (comparator)
void	*set_leq(leq_bound: long)*
void	*set_geq(geq_bound: long)*

Methods details

add_conditional (*cond: int*)

Adds a conditional to the constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], comp, 2, 0)
>>> constr.add_conditional(3)
```

add_conditionals (*conds: [int]*)

Adds a list of conditionals to the constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], comp, 2, 0)
>>> constr.add_conditionals([1, 2])
```

get_conditionals () → [int]

Returns a list of conditionals.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> constr.add_conditionals([1, 2])
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], comp, 2, 0)
>>> cond = constr.get_conditionals()
>>> cond
[1, 2]
```

clear_conditional ()

Removes all conditionals in a constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], comp, 2, 0)
>>> constr.add_conditionals([1, 2])
>>> cond = constr.get_conditionals()
>>> cond
[1, 2]
>>> cond = constr.clear_conditionals()
>>> cond
>>>
```

get_comparator () → comparator

Returns an int to represents the comparator (GEQ = 0, LEQ = 1 and BOTH = 2).

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> constr = pblib.PBConstraint([wl1, wl2], pblib.GEQ, 1)
>>> comp = constr.get_comparator()
>>> print(comp)
0
```

get_leq() → long

Returns the less-equal value bound of the constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> constr = pblib.PBConstraint([wl1, wl2], pblib.LEQ, 1)
>>> n = constr.get_leq()
>>> print(n)
1
```

get_geq() → long

Returns the greater-equal value bound of the constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> constr = pblib.PBConstraint([wl1, wl2], pblib.GEQ, 0)
>>> n = constr.get_geq()
>>> print(n)
0
```

get_min_sum() → long

Returns the minimum sum of literals' weights.

```
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> wls = [wl1, wl2, wl3]
>>> constr = pblib.PBConstraint(wls, pblib.LEQ, 4)
>>> s = constr.get_min_sum()
>>> print("Min = ", s)
>>> Min = -1
```

get_max_sum() → long

Returns the maximum sum of literals' weights.

```
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> wls = [wl1, wl2, wl3]
>>> constr = pblib.PBConstraint(wls, pblib.LEQ, 4)
>>> s = constr.get_min_sum()
>>> print("Max = ", s)
>>> Max = 2
```

get_n() → int

Returns the number of weigted literals in the constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> constr = pblib.PBConstraint([wl1, wl2], pblib.GEQ, 1)
>>> n = constr.get_n()
>>> print(n)
2
>>> wl3 = pblib.WeightedLit(3, 2)
>>> constr2 = pblib.PBConstraint([wl1, wl2, wl3], pblib.GEQ, 1)
>>> n2 = constr2.get_n()
>>> print(n2)
3
```

get_weighted_literals() → [WeightedLiterals]

Returns a copy of the vector of weighted literals which PBContraint contains.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], pblib.LEQ, 2)
>>> w_lits = constr.get_weighted_literals()
>>> print(w_lits)
[WeightedLit(lit=1, weight=2), WeightedLit(lit=2, weight=2),␣
↪WeightedLit(lit=3, weight=2)]
>>> constr_2 = pblib.PBConstraint(w_lits, pblib.GEQ, 4)
>>> constr_2
+2x1+2x2+2x3>=4
```

get_geq_constraint() → PBConstraint

Takes a pseudo boolean constraint and returns a new PBConstraint changing the comparator to a greater-equal. However, the bound does not change. If the greater-equal bound is equal to zero (because the previous restriction had not been assigned any value), the bound will remain zero.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], pblib.LEQ, 2)
>>> constr
+2x1+2x2+2x3<=2
>>> constr_2 = constr.get_geq_constraint()
>>> constr_2
+2x1+2x2+2x3>=0
```

get_leq_constraint() → PBConstraint

Takes a pseudo boolean constraint and returns a new PBConstraint changing the comparator to a less-equal. However, the bound does not change. If the less-equal bound is equal to zero (because the previous restriction had not been assigned any value), the bound will remain zero.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, 2)
```

(continues on next page)

(continued from previous page)

```
>>> constr = pblib.PBConstraint([wl1, wl2, wl3], pblib.GEQ, 4)
>>> constr
+2x1+2x2+2x3>=4
>>> constr_2 = constr.get_leq_constraint()
>>> constr_2
+2x1+2x2+2x3<=0
```

set_comparator(*comparator*)

Sets the constraint's comparator to specified value.

```
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> wls = [wl1, wl2, wl3]
>>> constr = pblib.PBConstraint(wls, pblib.LEQ, 4)
>>> print(constr.get_comparator())
1
>>> constr.set_comparator(pblib.GEQ)
>>> print(constr.get_comparator())
0
```

set_leq(*leq_bound: long*)

Sets the less-equal bound to specified value.

```
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> wls = [wl1, wl2, wl3]
>>> constr = pblib.PBConstraint(wls, pblib.LEQ, 2)
>>> print(constr.get_leq())
2
>>> constr.set_leq(1)
>>> print(constr.get_leq())
1
```

set_geq(*geq_bound: long*)

Sets the greater-equal bound to specified value.

```
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> wls = [wl1, wl2, wl3]
>>> constr = pblib.PBConstraint(wls, pblib.GEQ, 1)
>>> print(constr.get_geq())
1
>> constr.set_geq(0)
>> print(constr.get_geq())
0
```

1.2.3 Class IncPBConstraint

Constructor

IncPBConstraint(weightedlits: [*WeightedLits*], comparator, bound: int)

Creates an object to represents a Pseudo-Boolean incremental costraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2,-1)
>>> wl3 = pblib.WeightedLit(3, 3)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.LEQ, 3)
>>> i_constr
+2x1-1x2+3x3<=3
```

IncPBConstraint(weightedlits: [*WeightedLits*], comparator, bound_geq: int, bound_leq: int)

Creates an object to represents a Pseudo-Boolean incremental costraint, in case pblib.BOTH comparator and bound less-equals and greater-equal are different.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2,-1)
>>> wl3 = pblib.WeightedLit(3, 3)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.BOTH, 3, -1)
>>> i_constr
-1<=+2x1-1x2+3x3<=3
```

Methods summary

Return Type	Method
void	*add_conditional(cond: int)*
void	*add_conditionals(conds: [int])*
[int]	*get_conditionals()*
void	*clear_conditionals()*
comparator	*get_comparator()*
long	*get_leq()*
long	*get_geq()*
void	*set_comparator* (comparator)
void	*encode_new_geq(geq: long, VectorClauseDatabase: formula, AuxVarManager: aux)*
void	*encode_new_leq(leq: long, VectorClauseDatabase: formula, AuxVarManager: aux)*
PBConstraint	*get_non_inc_constraint()*
IncPBConstraint	*get_geq_inc_constraint()*
IncPBConstraint	*get_leq_inc_constraint()*
int	*get_n()*

Methods details

add_conditional (*cond: int*)

Adds a conditional to the incremental constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1,-3)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.LEQ, 1)
>>> i_constr
-3x1+x2+2x3<=1
>>> i_constr.add_conditional(3)
```

add_conditionals(*conds: [int]*)
Adds a list of conditionals to the incremental constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1,-3)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(2, 2)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.LEQ, 1)
>>> i_constr.add_conditionals([2, 3])
```

get_conditionals() → [int]
Returns a list of incremental conditionals.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 3)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.LEQ, 2)
>>> i_constr.add_conditionals([1, 3])
>>> i_constr.get_conditionals()
[1, 3]
```

clear_conditionals()
Removes all conditionals in a incremental constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 3)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.LEQ, 2)
>>> i_constr.add_conditionals([1, 3])
>>> i_constr.get_conditionals()
[1, 3]
>>> i_constr.clear_conditionals()
>>> cond = i_constr.get_conditionals()
[]
```

get_comparator() → comparator

Returns an int to represents the comparator (GEQ = 0, LEQ = 1 and BOTH = 2).

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
```

(continues on next page)

(continued from previous page)

```
>>> i_constr = pblib.IncPBConstraint([wl1, wl2],
                                     pblib.GEQ, 1)
>>> comp = i_constr.get_comparator()
>>> print(comp)
0
```

get_leq() → long

Returns the less-equal value bound of the incremental constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> i_constr = pblib.PBConstraint([wl1, wl2],
                                  pblib.LEQ, 1)
>>> n = i_constr.get_leq()
>>> print(n)
1
```

get_geq() → long

Returns the greater-equal value bound of the incremental constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> i_constr = pblib.PBConstraint([wl1, wl2],
                                  pblib.GEQ, 0)
>>> n = i_constr.get_geq()
>>> print(n)
0
```

set_comparator(*comparator*)

Sets the incremental constraint's comparator to specified value.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 1)
>>> wl2 = pblib.WeightedLit(2, 1)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> wls = [wl1, wl2, wl3]
>>> i_constr = pblib.IncPBConstraint(wls,
                                  pblib.LEQ, 4)
>>> print(i_constr.get_comparator())
1
>>> i_constr.set_comparator(pblib.GEQ)
>>> print(i_constr.get_comparator())
0
```

encode_new_geq(*geq_bound: long*, *VectorClauseDatabase: formula*, *AuxVarManager: aux_var*)

After the initial encoding of an incremental pseudo-boolean constraint, it is possible to encode a new (tighter) bounds with this method. Encodes an incremental constraint with a new greater-equal bound.

Example:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pypblib import pblib

#
################################################################################

def test_formula():
    wl1 = pblib.WeightedLit(1, 1)
    wl2 = pblib.WeightedLit(2, 1)
    wl3 = pblib.WeightedLit(3, 1)
    wl4 = pblib.WeightedLit(4, -1)
    wl5 = pblib.WeightedLit(5, -1)
    wls = [wl1, wl2, wl3, wl4, wl5]
    config = pblib.PBConfig()
    aux_var = pblib.AuxVarManager(6)
    formula = pblib.VectorClauseDatabase(config)
    geq = 1
    config.set_PB_Encoder(pblib.PB_SORTINGNETWORKS)

    i_const = pblib.IncPBConstraint(wls, pblib.GEQ, 0)
    pb2 = pblib.Pb2cnf(config)

    pb2.encode_inc_initial(i_const, formula, aux_var)
    formula.print_formula("test1.cnf", aux_var.get_biggest_returned_auxvar())

    i_const.encode_new_geq(geq, formula, aux_var)
    formula.print_formula("test2.cnf", aux_var.get_biggest_returned_auxvar())


#
################################################################################

test_formula()
```

encode_new_leq(*leq_bound: long*, *VectorClauseDatabase: formula*, *AuxVarManager: aux_var*)

After the initial encoding of an incremental pseudo-boolean constraint, it is possible to encode a new (tighter) bounds with this method. Encodes an incremental constraint with a new leq bound.

Example:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pypblib import pblib

#
################################################################################

def test_formula():
    wl1 = pblib.WeightedLit(1, 1)
    wl2 = pblib.WeightedLit(2, 1)
    wl3 = pblib.WeightedLit(3, 1)
    wl4 = pblib.WeightedLit(4, 1)
    wl5 = pblib.WeightedLit(5, 1)
```

(continues on next page)

(continued from previous page)

```
    wls = [wl1, wl2, wl3, wl4, wl5]
    config = pblib.PBConfig()
    aux_var = pblib.AuxVarManager(6)
    formula = pblib.VectorClauseDatabase(config)
    leq = 3
    config.set_PB_Encoder(pblib.PB_SORTINGNETWORKS)

    i_const = pblib.IncPBConstraint(wls, pblib.LEQ, 4)
    pb2 = pblib.Pb2cnf(config)

    pb2.encode_inc_initial(i_const, formula, aux_var)
    formula.print_formula("test1.cnf", aux_var.get_biggest_returned_auxvar())

    i_const.encode_new_leq(leq, formula, aux_var)
    formula.print_formula("test2.cnf", aux_var.get_biggest_returned_auxvar())


    #
    ##############################################################################

test_formula()
```

get_non_inc_constraint -> PBConstraint

Returns a new PBConstraint with the same atributes (weighted literals, comparator and bounds) of the IncPBConstraint which takes as a parameter.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3], pblib.LEQ, 2)
>>> i_constr
+2x1+2x2+2x3<=2
>>> type(i_constr)
<class 'pypblib.pblib.IncPBConstraint'>
>>> constr = i_constr.get_non_inc_constraint()
>>> constr
+2x1+2x2+2x3<=2
>>> type(constr)
<class 'pypblib.pblib.PBConstraint'>
```

get_geq_inc_constraint() → IncPBConstraint

Takes an incremental pseudo boolean constraint and returns a new IncPBConstraint changing the comparator to a greater-equal. However, the bound does not change. If the greater-equal bound is equal to zero (because the previous incremental restriction had not been assigned any value), the bound will remain zero.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3], pblib.LEQ, 2)
>>> i_constr
+2x1+2x2+2x3<=2
>>> i_constr2 = i_constr.get_geq_inc_constraint()
```

(continues on next page)

(continued from previous page)

```
>>> i_constr2
+2x1+2x2+2x3>=0
```

get_leq_inc_constraint() → IncPBConstraint

Takes an incremental pseudo boolean constraint and returns a new IncPBConstraint changing the comparator to a less-equal. However, the bound does not change. If the less-equal bound is equal to zero (because the previous incremental restriction had not been assigned any value), the bound will remain zero.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, 2)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3], pblib.GEQ, 4)
>>> i_constr
+2x1+2x2+2x3>=4
>>> i_constr2 = i_constr.get_leq_inc_constraint()
>>> i_constr2
+2x1+2x2+2x3<=0
```

get_n() → int

Returns the number of weigted literals in the incremental constraint.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 3)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> i_constr = pblib.IncPBConstraint([wl1, wl2, wl3],
                                     pblib.LEQ, 2)
>>> i_constr.get_n()
3
```

1.2.4 Class PBConfig

Constructor

PBConfig()

An instance of a PBConfig class contains the configuration for all options in PBLib. The following is its default setting:

```
>>> from pypblib import pblib
>>> c = pblib.PBConfig()
>>> c
PBConfig:
========
PB_Encoder = BEST
AMK_Encoder = BEST
AMO_Encoder = BEST
BIMANDER_M_IS = N_HALF
bimander = 3
k product min lit cnt for splitting = 10
K product k = 2
Commander encoding k = 3
```

(continues on next page)

(continued from previous page)

```
Max clause per constraint = 1000000
Use formula cache = FALSE
Print used encoding = FALSE
Check for duplicate literals = FALSE
Use gac Binary Merge = FALSE
Binary merge no support for single bits = TRUE
Use recursive BDD test = FALSE
Use real robdds = TRUE
Use watch dog encoding in Binary Merger = FALSE
```

Methods summary

Return_Type	Method
void	*set_PB_Encoder* (PB_Encoder)
void	*set_AMK_Encoder* (AMK_Encoder)
void	*set_AMO_Encoder* (AMO_Encoder)
void	*set_Bimander* (Bimander_M_IS)
void	*set_Bimander_m* (int: m)
void	*set_k_product_minimum_lit_count_for_splitting(min_lits:int)*
void	*set_k_product_k(k: int)*
void	*set_commander_encoding_k(k: int)*
void	*set_max_clause_per_constraint(max: int)*
void	*set_use_formula_cache(use: bool)*
void	*check_for_dup_literals(check: bool)*
void	*use_gac_binary_merge(use: bool)*
void	*binary_merge_no_support_for_single_bits(support: bool)*
void	*use_recursive_BDD_test(use: bool)*
void	*use_real_robdds(use: bool)*
void	*use_watch_dog_encoding_in_binary_merger(use: bool)*

Methods details

set_PB_Encoder(*pb_encoder*)

Sets the PB_Encoder configuration. By default: pypblib.pblib.PB_BEST

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_PB_Encoder(pblib.PB_BDD)
```

set_AMK_Encoder(*amk_encoder*)

Sets the AMK_Encoder configuration. By default: pypblib.pblib.AMK_BEST

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_AMK_Encoder(pblib.AMK_CARD)
```

set_AMO_Encoder(*amo_encoder*)

Sets the AMO_Encoder configuration. By default: pypblib.pblib.AMO_BEST

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_AMO_Encoder(pblib.AMO_NESTED)
```

set_Bimander(*bimander_encode*)

Sets the BIMANDER_M_IS configuration. By default: pypblib.pblib.N_HALF

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_Bimander(pblib.FIXED)
```

set_bimander_m(*m: int*)

Sets bimander m. Defalut value is 3.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_bimander_m(5)
```

set_k_product_minimum_lit_count_for_splitting(*nim_lits: int*)

Sets k-product minimum literals for splitting. Default value is 10.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_k_product_minimum_lit_count_for_splitting(12)
```

set_k_product_k(*k: int*)

Sets the k value for k-product. Default value is 2.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_kproduct_k(4)
```

set_commander_encoding_k(*k: int*)

Sets the k value of commander encoding. Default value is 3.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_commander_encoding_k(5)
```

set_max_clause_per_constraint(*max: int*)

Sets the maximum number of clauses per constraint. Default value is 1000000.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_max_clause_per_constraint(2000000)
```

set_use_formula_cache(*use: bool*)

Sets the use of formula cache. False by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.set_use_formula_cache(True)
```

check_for_dup_literals (*check: bool*)

Sets the check for duplicated literals. False by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.check_for_dup_literals(True)
```

use_gac_binary_merge (*use: bool*)

Sets the use of GAC (generalized arc-consistent). False by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.use_gac_binary_merge(True)
```

binary_merge_no_support_for_single_bits (*support: bool*)

True by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.binary_merge_no_support_for_single_bits(False)
```

use_recursive_BDD_test (*use: bool*)

False by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.use_recursive_BDD_test(True)
```

use_real_robdds (*use: bool*)

True by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.use_real_robdds(False)
```

use_watch_dog_encoding_in_binary_merger (*use: bool*)

False by default.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> config.use_watch_dog_encoding_in_binary_merger(True)
```

1.2.5 Class AuxVarManager

Constructor

AuxVarManager(free_var: int)

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

```
>>> from pypblib import pblib
>>> aux_var = pblib.AuxVarManager(5)
```

Methods summary

Return Type	Methods
int	*get_variable()*
void	*free_variable(var: int)*
void	*free_variables(vars: [int])*
void	*free_range_variables(var_start: int, var_end: int)*
int	*get_biggest_returned_auxvar()()*
void	*reset_aux_var_to(first_free_var: int)*

Methods details

get_variable() → int

Returns the first free variable if all variables between 1 and the biggest variable are been used. Otherwise returns an unused variable between 1 and the biggest used variable. From that moment the returned variable is considered as used.

Warning: As you can see in the next example, in the case of unused variables between 1 and the biggest variable, the returned variable could be random.

```
>>> from pypblib import pblib
>>> aux = pblib.AuxVarManager(10)
>>> aux.get_variable()
10
>>> aux.get_variable()
11
>>> aux.free_variables([1, 3, 5, 7, 9])
>>> aux.get_variable()
9
>>> aux.get_variable()
1
>>> aux.get_variable()
3
```

free_variable(*var: int*)

Frees an used variable to refill the gap later on.

```
>>> from pypblib import pblib
>>> aux = pblib.AuxVarManager(9)
>>> aux.free_variable(3)
>>> aux.get_variable()
3
```

free_variables(*vars: [int]*)

Frees the used variables specified in the list.

```
>>> from pypblib import pblib
>>> aux = pblib.AuxVarManager(10)
>>> aux.free_variables([1, 3, 5, 7, 9])
>>> aux.get_variable()
9
>>> aux.get_variable()
1
```

free_range_variables(*var_start: int*, *var_end: int*)

Frees the range of used variables between the specified start and end. Both limits are included.

```
>>> from pypblib import pblib
>>> aux = pblib.AuxVarManager(10)
>>> aux.free_range_variables(3, 5)
>>> aux.get_variable()
5
>>> aux.get_variable()
4
>>> aux.get_variable()
3
>>> aux.get_variable()
10
```

get_biggest_returned_auxvar() → int

Returns the biggest used variable. Hence every variable between this (including) number and *free_var* (probably) occurs in some clause database.

```
>>> from pypblib import pblib
>>> aux_var = pblib.AuxVarManager(5)
>>> print(aux_var.get_biggest_returned_auxvar())
4
```

reset_aux_var_to(*first_free_var: int*)

Resets the first free variable with the specified integer.

```
>>> from pypblib import pblib
>>> aux = pblib.AuxVarManager(10)
>>> aux.get_biggest_returned_auxvar()
9
>>> aux.reset_aux_var_to(20)
>>> aux.get_biggest_returned_auxvar()
19
```

1.2.6 Class VectorClauseDatabase

Its a default implementation of a Clause Database, in PBLib. Every constraint is saved into a vector of clauses.

Constructor

VectorClauseDatabase(config: *PBConfig*)

Constructs a vector clause database with the specified configuration. Takes one parameters: a *PBConfig*.

```
>>> from pypblib.pblib import PBConfig, VectorClauseDatabase
>>> config = PBConfig()
>>> formula = VectorClauseDatabase(config)
>>> formula
Num. Causes: 0
==============================
==============================
```

VectorClauseDatabase(config: *PBConfig*, clauses: [[int]])

Constructs a vector clause database with the specified configuration and save the recieved clauses. Takes two parameters: a *PBConfig* and a list of integer's lists. Every list represents a clause in a previous formula.

```
>>> from pypblib import pblib
>>> formula = pblib.VectorClauseDatabase(pblib.PBConfig(),
                        [[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> formula
Num. Causes: 3
==============================
1 2 3 0
4 5 6 0
7 8 9 0
==============================
```

Methods summary

Return Type	Method
int	*get_num_clauses()*
void	*print_formula()*
void	*print_formula(fname: string, nclauses: int)*
[[int]]	*get_clauses()*
void	*clear_database()*
void	*reset_internal_unsat_state()*

Methods details

get_num_clauses() → int

Returns the number of clauses in database.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> cl = [[1, -2, 3], [4, -5, 6], [2, -4,]]
>>> formula = pblib.VectorClauseDatabase(config, cl)
>>> formula.get_num_clauses()
3
```

get_clauses() → [[int]]

Returns a list of integer's lists to represents the clauses in VectorClauseDatabase.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> formula = pblib.VectorClauseDatabase(config,
              [[1, -2, 3], [4, -5, 6], [7, -8, 9]])
>>> clauses = formula.get_clauses()
>>> print(clauses)
[[1, -2, 3], [4, -5, 6], [7, -8, 9]]
```

print_formula()

Prints the codification into standard output.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> cl = [[1, -2, 3], [4, -5, 6], [2, -4,]]
>>> formula = pblib.VectorClauseDatabase(config, cl)
>>> formula.print_formula()
 1 -2 3 0
 4 -5 6 0
 2 -4 0
```

print_formula(*fname: string*, *nclauses: int*)

Prints the codification into a file with specified name.

clear_database()

Resets the clause database.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> cl = [[1, -2, 3], [4, -5, 6], [2, -4,]]
>>> formula = pblib.VectorClauseDatabase(config, cl)
>>> formula.print_formula()
 1 -2 3 0
 4 -5 6 0
 2 -4 0
>>> formula.clear_database()
>>> formula.print_formula()
>>>
```

reset_internal_unsat_state()

Sets the internal unsat state to false.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> cl = [[1, -2, 3], [4, -5, 6], [2, -4,]]
>>> formula = pblib.VectorClauseDatabase(config, cl)
>>> formula.reset_internal_unsat_state()
```

1.2.7 Class Pb2cnf

The class Pb2cnf is the main interface to encode a PB constraint.

Constructor

Pb2cnf(config: *PBConfig*, stat: Statistics)

Creates a Pb2cnf with the specified configuration & statistics.

```
>>> from pypblib import pblib
>>> conf = pblib.PBConfig()
>>> stat = pblib.Statistics()
>>> pb2 = pblib.Pb2cnf(conf, stat)
```

Pb2cnf(config: *PBConfig*)

Creates a Pb2cnf with the specified configuration.

```
>>> from pypblib import pblib
>>> c = pblib.PBConfig()
>>> c
>>> pb2 = pblib.Pb2cnf(c)
```

Pb2cnf()

If the constructor does not receive any parameters, it will be created with default configuration.

```
>>> from pypblib import pblib
>>> pb2 = pblib.Pb2cnf()
```

Methods sumary

Return Type	Method
int	*encode_at_most_k* (literals: [int], k: long, formula: [], first_free_var: int)
int	*encode_at_least_k* (literals: [int], k: long, formula: [], first_free_var: int)
int	*encode_leq* (weights: [long], literals: [int], k: long, formula: [], first_free_var: int)
int	*encode_geq* (weights: [long], literals: [int], k: long, formula: [], first_free_var: int)
int	*encode_both* (weights: [long], literals [int], leq: long, geq: long, formula: [], first_free_var: int)
void	***encode*** **(constr_list: [*** *PBConstraint* **], formula:** *VectorClauseDatabase*, aux_var_manager *AuxVarManager*)
void	***encode_inc_initial*** **(i_const_list: [*** *IncPBConstraint* **], form:** *VectorClauseDatabase*, aux_var_man *AuxVarManager*)

Methods details

encode_at_most_k (*literals: [int], k: long, formula: [], first_free_variable: int*) → int

For a simple interface 'at most k'. The return value is the last used variable. Hence the next integer is the first free variable. At the end, the formula will contains the codification.

```
>>> from pypblib import pblib
>>> literals = [1, 2, 3]
>>> k = 1
>>> first_free_var = 4
>>> formula = []
>>> config = pblib.PBConfig()
>>> max_var = pb2.encode_at_most_k(literals, k, formula,
                                   first_free_var)
>>> print(max_var)
3
>>> print(formula)
[[-3, -2], [-3, -1], [-2, -1]]
```

encode_at_least_k (*literals: [int], k: long, formula: [], first_free_variable: int*) → int

For a simple interface 'at least k'. The return value is the last used variable. Hence the next integer is the first free variable. At the end, the formula will contains the codification.

```
>>> from pypblib.pblib import PBConfig, Pb2cnf
>>> formula = []
>>> pb2 = Pb2cnf(PBConfig())
>>> max_var = pb2.encode_at_least_k([1, 2, 3, 4], 2,
                                    formula, 5)
```

(continues on next page)

(continued from previous page)

```
>>>print(max_var)
10
>>> for clause in formula:
...         for var in clause:
...                 print(var, end=" ")
...         print("0")
...
5 0
1 -6 0
2 -6 0
1 2 -7 0
7 -8 0
6 3 -8 0
7 3 -9 0
9 -10 0
8 4 -10 0
10 0
```

encode_leq(*weights: [long], literals: [int], k: long, formula: [], first_free_variable: int*) → int

For a simple interface and LEQ (less-equals) comparator. The return value is the last used variable. Hence the next integer is the first free variable. At the end, the formula will contains the codification.

```
>>> from pypblib.pblib import Pb2cnf, PBConfig
>>> pb2 = Pb2cnf(PBConfig())
>>> formula = []
>>> weights = [2, 3, 1, 2]
>>> literals = [1, 2, 3, 4]
>>> first_free_var = 5
>>> leq = 3
>>> max_var = pb2.encode_leq(weights, literals, leq,
                             formula, first_free_var)
>>> print(max_var)
9
>>> print(formula)
[[5], [-3, -6], [-1, -6], [6, -7], [-4, -7], [-1, -4, -8],
 [8, -9], [7, -2, -9], [9]]
```

encode_geq(*weights: [long], literals: [int], k: long, formula: [], first_free_variable: int*) → int

For a simple interface and GEQ (greater-equals) comparator. The return value is the last used variable. Hence the next integer is the first free variable. At the end, the formula will contains the codification.

```
>>> from pypblib.pblib import Pb2cnf, PBConfig
>>> pb2 = Pb2cnf(PBConfig())
>>> formula = []
>>> max_var = pb2.encode_geq([2, 1, 2, 4], [1, 2, 3, 4],
                             3, formula, 5)
>>> print(max_var)
9
>>> print(formula)
[[5], [2, -6], [1, -6], [2, 1, -7], [7, -8], [6, 3, -8],
 [8, 4, -9], [9]]
```

encode_both(*weights: [long], literals [int], leq: long, geq: long, formula: [], first_free_variable: int*) → int

For a simple interface, LEQ (less-equals) and GEQ (greater-equals) comparators. The return value is the

last used variable. Hence the next integer is the first free variable. At the end, the formula will contains the codification.

```
>>> from pypblib import pblib
>>> config = pblib.PBConfig()
>>> pb2 = pblib.Pb2cnf(config)
>>> formula = []
>>> leq = 3
>>> geq = 1
>>> weights = [2, 3, 1, 1]
>>> literals = [1, 2, 3, 4]
>>> first_free_var = 5
>>> max_var = pb2.encode_both(weights, literals, leq, geq,
                              formula, first_free_var)
>>> print(max_var)
10
>>> for clause in formula:
...         for var in clause:
...                 print(var, end=" ")
...         print("0")
...
4 3 2 1 0
5 0
-3 -6 0
-4 -6 0
6 -7 0
-1 -7 0
-3 -4 -8 0
8 -1 -9 0
9 -10 0
7 -2 -10 0
10 0
```

encode(*constr: PBConstraint*, *formula: VectorClauseDatabase*, *aux_var_manager: AuxVarManager*)

For this sophisticated interface you should use, besides the *PBConstraint*, a *VectorClauseDatabase* and a *AuxVarManager* to encode a pseudo boolean constriant. At the end, the formula will contains the codification.

```
>>> from pypblib import pblib
>>> wl_List = [pblib.WeightedLit(1,3),
               pblib.WeightedLit(2,1),
               pblib.WeightedLit(3,5)]
>>> comparator = pblib.GEQ
>>> bound = 1
>>> aux_var = pblib.AuxVarManager(4)
>>> constr = pblib.PBConstraint(wl_List, comparator, bound)
>>> config = pblib.PBConfig()
>>> formula = pblib.VectorClauseDatabase(config)
>>> pb2cnf = pblib.Pb2cnf(config)
>>> pb2cnf.encode(constr, formula, aux_var)
>>> print(formula)
4 0
2 1 -5 0
5 3 -6 0
6 0
>>> formula.print_formula("encode_test.cnf",
                          aux_var.get_biggest_returned_auxvar())
```

encode_inc_initial(*inc_constr: IncPBConstraint*, *formula: VectorClauseDatabase*, *aux_var_manager: AuxVarManager*)

For incremental pseudo boolean constraint. You should use, besides the *IncPBConstraint*, a *VectorClauseDatabase* and a *AuxVarManager* to encode. At the end, the formula will contains the codification.

```
>>> from pypblib import pblib
>>> wl1 = pblib.WeightedLit(1, 2)
>>> wl2 = pblib.WeightedLit(2, 2)
>>> wl3 = pblib.WeightedLit(3, -1)
>>> i_contr = pblib.IncPBConstraint([wl1, wl2, wl3], pblib.LEQ, 3)
>>> aux_var = pblib.AuxVarManager(4)
>>> config = pblib.PBConfig()
>>> formula = pblib.VectorClauseDatabase(config)
>>> pb2 = pblib.Pb2cnf()
>>> pb2.encode_inc_initial(i_contr, formula, aux_var)
```

PYTHON MODULE INDEX

p

INDEX