PySAT Documentation

Release 0.1.5.dev12

Alexey Ignatiev, Joao Marques-Silva, Antonio Morgado

Apr 12, 2020

CONTENTS

This site covers the usage and API documentation of the PySAT toolkit. For the basic information on what PySAT is, please, see the main project website.

CHAPTER

ONE

API DOCUMENTATION

The PySAT toolkit has four core modules: *card*, *formula*, *pb* and *solvers*. The three of them (*card*, *pb* and *solvers*) are Python wrappers for the code originally implemented in the C/C++ languages while the *formula* module is a *pure* Python module. Version *0.1.4.dev0* of PySAT brings a new module called `pb`, which is a wrapper for the basic functionality of a third-party library PyPBLib developed by the Logic Optimization Group of the University of Lleida.

1.1 Core PySAT modules

1.1.1 Cardinality encodings (`pysat.card`)

List of classes

EncType	This class represents a C-like `enum` type for choosing the cardinality encoding to use.
CardEnc	This abstract class is responsible for the creation of cardinality constraints encoded to a CNF formula.
ITotalizer	This class implements the iterative totalizer encoding[11].

Module description

This module provides access to various *cardinality constraint*[1] encodings to formulas in conjunctive normal form (CNF). These include pairwise[2], bitwise[2], ladder/regular[3,4], sequential counters[5], sorting[6] and cardinality networks[7], totalizer[8], modulo totalizer[9], and modulo totalizer for k-cardinality[10], as well as a *native* cardinality constraint repre-

[11] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, Inês Lynce. *Incremental Cardinality Constraints for MaxSAT*. CP 2014. pp. 531-548

[1] Olivier Roussel, Vasco M. Manquinho. *Pseudo-Boolean and Cardinality Constraints*. Handbook of Satisfiability. 2009. pp. 695-733

[2] Steven David Prestwich. *CNF Encodings*. Handbook of Satisfiability. 2009. pp. 75-97

[3] Carlos Ansótegui, Felip Manyà. *Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables*. SAT (Selected Papers) 2004. pp. 1-15

[4] Ian P. Gent, Peter Nightingale. *A New Encoding of Alldifferent Into SAT*. In International workshop on modelling and reformulating constraint satisfaction problems 2004. pp. 95-110

[5] Carsten Sinz. *Towards an Optimal CNF Encoding of Boolean Cardinality Constraints*. CP 2005. pp. 827-831

[6] Kenneth E. Batcher. *Sorting Networks and Their Applications*. AFIPS Spring Joint Computing Conference 1968. pp. 307-314

[7] Roberto Asin, Robert Nieuwenhuis, Albert Oliveras, Enric Rodriguez-Carbonell. *Cardinality Networks and Their Applications*. SAT 2009. pp. 167-180

[8] Olivier Bailleux, Yacine Boufkhad. *Efficient CNF Encoding of Boolean Cardinality Constraints*. CP 2003. pp. 108-122

[9] Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, Hiroshi Fujita. *Modulo Based CNF Encoding of Cardinality Constraints and Its Application to MaxSAT Solvers*. ICTAI 2013. pp. 9-17

[10] António Morgado, Alexey Ignatiev, Joao Marques-Silva. *MSCG: Robust Core-Guided MaxSAT Solving*. System Description. JSAT 2015. vol. 9, pp. 129-134

sentation supported by the MiniCard solver.

A cardinality constraint is a constraint of the form: $\sum_{i=1}^{n} x_i \leq k$. Cardinality constraints are ubiquitous in practical problem formulations. Note that the implementation of the pairwise, bitwise, and ladder encodings can only deal with AtMost1 constraints, e.g. $\sum_{i=1}^{n} x_i \leq 1$.

Access to all cardinality encodings can be made through the main class of this module, which is `CardEnc`.

Additionally, to the standard cardinality encodings that are basically "static" CNF formulas, the module is designed to able to construct *incremental* cardinality encodings, i.e. those that can be incrementally extended at a later stage. At this point only the *iterative totalizer*[11] encoding is supported. Iterative totalizer can be accessed with the use of the `ITotalizer` class.

Module details

class pysat.card.**CardEnc**

This abstract class is responsible for the creation of cardinality constraints encoded to a CNF formula. The class has three *class methods* for creating AtMostK, AtLeastK, and EqualsK constraints. Given a list of literals, an integer bound and an encoding type, each of these methods returns an object of class `pysat.formula.CNFPlus` representing the resulting CNF formula.

Since the class is abstract, there is no need to create an object of it. Instead, the methods should be called directly as class methods, e.g. `CardEnc.atmost(lits, bound)` or `CardEnc.equals(lits, bound)`. An example usage is the following:

```
>>> from pysat.card import *
>>> cnf = CardEnc.atmost(lits=[1, 2, 3], encoding=EncType.pairwise)
>>> print(cnf.clauses)
[[-1, -2], [-1, -3], [-2, -3]]
>>> cnf = CardEnc.equals(lits=[1, 2, 3], encoding=EncType.pairwise)
>>> print(cnf.clauses)
[[1, 2, 3], [-1, -2], [-1, -3], [-2, -3]]
```

classmethod atleast (*lits*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=1*)

This method can be used for creating a CNF encoding of an AtLeastK constraint, i.e. of $\sum_{i=1}^{n} x_i \geq k$. The method takes 1 mandatory argument `lits` and 3 default arguments can be specified: `bound`, `top_id`, `vpool`, and `encoding`.

Parameters

- **lits** (*iterable(int)*) – a list of literals in the sum.
- **bound** (*int*) – the value of bound k.
- **top_id** (*integer or None*) – top variable identifier used so far.
- **vpool** (IDPool) – variable pool for counting the number of variables.
- **encoding** (*integer*) – identifier of the encoding to use.

Parameter `top_id` serves to increase integer identifiers of auxiliary variables introduced during the encoding process. This is helpful when augmenting an existing CNF formula with the new cardinality encoding to make sure there is no collision between identifiers of the variables. If specified, the identifiers of the first auxiliary variable will be `top_id+1`.

Instead of `top_id`, one may want to use a pool of variable identifiers `vpool`, which is automatically updated during the method call. In many circumstances, this is more convenient than using `top_id`. Also note that parameters `top_id` and `vpool` **cannot** be specified *simultaneusly*.

The default value of `encoding` is `Enctype.seqcounter`.

The method *translates* the AtLeast constraint into an AtMost constraint by *negating* the literals of `lits`, creating a new bound $n - k$ and invoking `CardEnc.atmost()` with the modified list of literals and the new bound.

Raises **`CardEnc.NoSuchEncodingError`** – if encoding does not exist.

Return type a `CNFPlus` object where the new clauses (or the new native atmost constraint) are stored.

classmethod atmost (*lits*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=1*)

This method can be used for creating a CNF encoding of an AtMostK constraint, i.e. of $\sum_{i=1}^{n} x_i \leq k$. The method shares the arguments and the return type with method `CardEnc.atleast()`. Please, see it for details.

classmethod equals (*lits*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=1*)

This method can be used for creating a CNF encoding of an EqualsK constraint, i.e. of $\sum_{i=1}^{n} x_i = k$. The method makes consecutive calls of both `CardEnc.atleast()` and `CardEnc.atmost()`. It shares the arguments and the return type with method `CardEnc.atleast()`. Please, see it for details.

class pysat.card.**EncType**

This class represents a C-like `enum` type for choosing the cardinality encoding to use. The values denoting the encodings are:

```
pairwise    = 0
seqcounter  = 1
sortnetwrk  = 2
cardnetwrk  = 3
bitwise     = 4
ladder      = 5
totalizer   = 6
mtotalizer  = 7
kmtotalizer = 8
native      = 9
```

The desired encoding can be selected either directly by its integer identifier, e.g. `2`, or by its alphabetical name, e.g. `EncType.sortnetwrk`.

Note that while most of the encodings are produced as a list of clauses, the "native" encoding of MiniCard is managed as one clause. Given an AtMostK constraint $\sum_{i=1}^{n} x_i \leq k$, the native encoding represents it as a pair `[lits, k]`, where `lits` is a list of size `n` containing literals in the sum.

class pysat.card.**ITotalizer** (*lits=[]*, *ubound=1*, *top_id=None*)

This class implements the iterative totalizer encoding[11]. Note that `ITotalizer` can be used only for creating AtMostK constraints. In contrast to class `EncType`, this class is not abstract and its objects once created can be reused several times. The idea is that a *totalizer tree* can be extended, or the bound can be increased, as well as two totalizer trees can be merged into one.

The constructor of the class object takes 3 default arguments.

Parameters

- **lits** (*iterable(int)*) – a list of literals to sum.
- **ubound** (*int*) – the largest potential bound to use.
- **top_id** (*integer or None*) – top variable identifier used so far.

The encoding of the current tree can be accessed with the use of `CNF` variable stored as `self.cnf`. Potential bounds **are not** imposed by default but can be added as unit clauses in the final CNF formula. The bounds are stored in the list of Boolean variables as `self.rhs`. A concrete bound k can be enforced by considering a unit clause `-self.rhs[k]`. **Note** that `-self.rhs[0]` enforces all literals of the sum to be *false*.

An *ITotalizer* object should be deleted if it is not needed anymore.

Possible usage of the class is shown below:

```
>>> from pysat.card import ITotalizer
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.cnf.clauses)
[[-2, 4], [-1, 4], [-1, -2, 5], [-4, 6], [-5, 7], [-3, 6], [-3, -4, 7]]
>>> print(t.rhs)
[6, 7]
>>> t.delete()
```

Alternatively, an object can be created using the `with` keyword. In this case, the object is deleted automatically:

```
>>> from pysat.card import ITotalizer
>>> with ITotalizer(lits=[1, 2, 3], ubound=1) as t:
...     print(t.cnf.clauses)
[[-2, 4], [-1, 4], [-1, -2, 5], [-4, 6], [-5, 7], [-3, 6], [-3, -4, 7]]
...     print(t.rhs)
[6, 7]
```

delete()

Destroys a previously constructed *ITotalizer* object. Internal variables `self.cnf` and `self.rhs` get cleaned.

extend(*lits=[]*, *ubound=None*, *top_id=None*)

Extends the list of literals in the sum and (if needed) increases a potential upper bound that can be imposed on the complete list of literals in the sum of an existing *ITotalizer* object to a new value.

Parameters

- **lits** (*iterable(int)*) – additional literals to be included in the sum.
- **ubound** (*int*) – a new upper bound.
- **top_id** (*integer or None*) – a new top variable identifier.

The top identifier `top_id` applied only if it is greater than the one used in `self`.

This method creates additional clauses encoding the existing totalizer tree augmented with new literals in the sum and updating the upper bound. As a result, it appends the new clauses to the list of clauses of `CNF` `self.cnf`. The number of newly created clauses is stored in variable `self.nof_new`.

Also, if the upper bound is updated, a list of bounds `self.rhs` gets increased and its length becomes `ubound+1`. Otherwise, it is updated with new values.

The method can be used in the following way:

```
>>> from pysat.card import ITotalizer
>>> t = ITotalizer(lits=[1, 2], ubound=1)
>>> print(t.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4]]
>>> print(t.rhs)
[3, 4]
>>>
>>> t.extend(lits=[5], ubound=2)
>>> print(t.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4], [-5, 6], [-3, 6], [-4, 7], [-3, -5, 7], [-4, -
↪5, 8]]
>>> print(t.cnf.clauses[-t.nof_new:])
[[-5, 6], [-3, 6], [-4, 7], [-3, -5, 7], [-4, -5, 8]]
```

(continues on next page)

(continued from previous page)

```
>>> print(t.rhs)
[6, 7, 8]
>>> t.delete()
```

increase (*ubound=1*, *top_id=None*)

Increases a potential upper bound that can be imposed on the literals in the sum of an existing *ITotalizer* object to a new value.

Parameters

- **ubound** (*int*) – a new upper bound.
- **top_id** (*integer or None*) – a new top variable identifier.

The top identifier `top_id` applied only if it is greater than the one used in `self`.

This method creates additional clauses encoding the existing totalizer tree up to the new upper bound given and appends them to the list of clauses of `CNF` `self.cnf`. The number of newly created clauses is stored in variable `self.nof_new`.

Also, a list of bounds `self.rhs` gets increased and its length becomes `ubound+1`.

The method can be used in the following way:

```
>>> from pysat.card import ITotalizer
>>> t = ITotalizer(lits=[1, 2, 3], ubound=1)
>>> print(t.cnf.clauses)
[[-2, 4], [-1, 4], [-1, -2, 5], [-4, 6], [-5, 7], [-3, 6], [-3, -4, 7]]
>>> print(t.rhs)
[6, 7]
>>>
>>> t.increase(ubound=2)
>>> print(t.cnf.clauses)
[[-2, 4], [-1, 4], [-1, -2, 5], [-4, 6], [-5, 7], [-3, 6], [-3, -4, 7], [-3, -
↪5, 8]]
>>> print(t.cnf.clauses[-t.nof_new:])
[[-3, -5, 8]]
>>> print(t.rhs)
[6, 7, 8]
>>> t.delete()
```

merge_with (*another*, *ubound=None*, *top_id=None*)

This method merges a tree of the current *ITotalizer* object, with a tree of another object and (if needed) increases a potential upper bound that can be imposed on the complete list of literals in the sum of an existing *ITotalizer* object to a new value.

Parameters

- **another** (*ITotalizer*) – another totalizer to merge with.
- **ubound** (*int*) – a new upper bound.
- **top_id** (*integer or None*) – a new top variable identifier.

The top identifier `top_id` applied only if it is greater than the one used in `self`.

This method creates additional clauses encoding the existing totalizer tree merged with another totalizer tree into *one* sum and updating the upper bound. As a result, it appends the new clauses to the list of clauses of `CNF` `self.cnf`. The number of newly created clauses is stored in variable `self.nof_new`.

Also, if the upper bound is updated, a list of bounds `self.rhs` gets increased and its length becomes `ubound+1`. Otherwise, it is updated with new values.

The method can be used in the following way:

```
>>> from pysat.card import ITotalizer
>>> with ITotalizer(lits=[1, 2], ubound=1) as t1:
...     print(t1.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4]]
...     print(t1.rhs)
[3, 4]
...
...     t2 = ITotalizer(lits=[5, 6], ubound=1)
...     print(t1.cnf.clauses)
[[-6, 7], [-5, 7], [-5, -6, 8]]
...     print(t1.rhs)
[7, 8]
...
...     t1.merge_with(t2)
...     print(t1.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4], [-6, 7], [-5, 7], [-5, -6, 8], [-7, 9], [-8,␣
↪10], [-3, 9], [-4, 10], [-3, -7, 10]]
...     print(t1.cnf.clauses[-t1.nof_new:])
[[-6, 7], [-5, 7], [-5, -6, 8], [-7, 9], [-8, 10], [-3, 9], [-4, 10], [-3, -7,
↪ 10]]
...     print(t1.rhs)
[9, 10]
...
...     t2.delete()
```

new (*lits=[], ubound=1, top_id=None*)

The actual constructor of *ITotalizer*. Invoked from `self.__init__()`. Creates an object of *ITotalizer* given a list of literals in the sum, the largest potential bound to consider, as well as the top variable identifier used so far. See the description of *ITotalizer* for details.

exception `pysat.card.`**NoSuchEncodingError**

This exception is raised when creating an unknown an AtMostk, AtLeastK, or EqualK constraint encoding.

with_traceback ()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

1.1.2 Boolean formula manipulation (`pysat.formula`)

List of classes

`IDPool`	A simple manager of variable IDs.
`CNF`	Class for manipulating CNF formulas.
`CNFPlus`	CNF formulas augmented with *native* cardinality constraints.
`WCNF`	Class for manipulating partial (weighted) CNF formulas.
`WCNFPlus`	WCNF formulas augmented with *native* cardinality constraints.

Module description

This module is designed to facilitate fast and easy PySAT-development by providing a simple way to manipulate formulas in PySAT. Although only clausal formulas are supported at this point, future releases of PySAT are expected to implement data structures and methods to manipulate arbitrary Boolean formulas. The module implements the `CNF` class, which represents a formula in conjunctive normal form (CNF).

Recall that a CNF formula is conventionally seen as a set of clauses, each being a set of literals. A literal is a Boolean variable or its negation. In PySAT, a Boolean variable and a literal should be specified as an integer. For instance, a Boolean variable x_{25} is represented as integer `25`. A literal $\neg x_{10}$ should be specified as `-10`. Moreover, a clause $(\neg x_2 \vee x_{19} \vee x_{46})$ should be specified as `[-2, 19, 46]` in PySAT. *Unit size clauses* are to be specified as unit size lists as well, e.g. a clause (x_3) is a list `[3]`.

CNF formulas can be created as an object of class `CNF`. For instance, the following piece of code creates a CNF formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$.

```
>>> from pysat.formula import CNF
>>> cnf = CNF()
>>> cnf.append([-1, 2])
>>> cnf.append([-2, 3])
```

The clauses of a formula can be accessed through the `clauses` variable of class `CNF`, which is a list of lists of integers:

```
>>> print(cnf.clauses)
[[-1, 2], [-2 ,3]]
```

The number of variables in a CNF formula, i.e. the *largest variable identifier*, can be obtained using the `nv` variable, e.g.

```
>>> print(cnf.nv)
3
```

Class `CNF` has a few methods to read and write a CNF formula into a file or a string. The formula is read/written in the standard DIMACS CNF format. A clause in the DIMACS format is a string containing space-separated integer literals followed by `0`. For instance, a clause $(\neg x_2 \vee x_{19} \vee x_{46})$ is written as `-2 19 46 0` in DIMACS. The clauses in DIMACS should be preceded by a *preamble*, which is a line `p cnf nof_variables nof_clauses`, where `nof_variables` and `nof_clauses` are integers. A preamble line for formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ would be `p cnf 3 2`. The complete DIMACS file describing the formula looks this:

```
p cnf 3 2
-1 2 0
-2 3 0
```

Reading and writing formulas in DIMACS can be done with PySAT in the following way:

```
>>> from pysat.formula import CNF
>>> f1 = CNF(from_file='some-file-name.cnf')  # reading from file
>>> f1.to_file('another-file-name.cnf')  # writing to a file
>>>
>>> with open('some-file-name.cnf', 'r+') as fp:
...     f2 = CNF(from_fp=fp)  # reading from a file pointer
...
...     fp.seek(0)
...     f2.to_fp(fp)  # writing to a file pointer
>>>
>>> f3 = CNF(from_string='p cnf 3 3\n-1 2 0\n-2 3 0\n-3 0\n')
```

(continues on next page)

(continued from previous page)

```
>>> print(f3.clauses)
[[-1, 2], [-2, 3], [-3]]
>>> print(f3.nv)
3
```

Besides plain CNF formulas, the `pysat.formula` module implements an additional class for dealing with *partial* and *weighted partial* CNF formulas, i.e. WCNF formulas. A WCNF formula is a conjunction of two sets of clauses: *hard* clauses and *soft* clauses, i.e. $\mathcal{F} = \mathcal{H} \wedge \mathcal{S}$. Soft clauses of a WCNF are labeled with integer *weights*, i.e. a soft clause of $\mathcal{S}$ is a pair (c_i, w_i). In partial (unweighted) formulas, all soft clauses have weight 1.

WCNF can be of help when solving optimization problems using the SAT technology. A typical example of where a WCNF formula can be used is maximum satisfiability (MaxSAT), which given a WCNF formula $\mathcal{F} = \mathcal{H} \wedge \mathcal{S}$ targets satisfying all its hard clauses $\mathcal{H}$ and maximizing the sum of weights of satisfied soft clauses, i.e. maximizing the value of $\sum_{c_i \in \mathcal{S}} w_i \cdot c_i$.

An object of class `WCNF` has two variables to access the hard and soft clauses of the corresponding formula: `hard` and `soft`. The weights of soft clauses are stored in variable `wght`.

```
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF()
>>> wcnf.append([-1, -2])
>>> wcnf.append([1], weight=1)
>>> wcnf.append([2], weight=3)  # the formula becomes unsatisfiable
>>>
>>> print(wcnf.hard)
[[-1, -2]]
>>> print(wcnf.soft)
[[1], [2]]
>>> print(wcnf.wght)
[1, 3]
```

A properly constructed WCNF formula must have a *top weight*, which should be equal to $1 + \sum_{c_i \in \mathcal{S}} w_i$. Top weight of a formula can be accessed through variable `topw`.

```
>>> wcnf.topw = sum(wcnf.wght) + 1  # (1 + 3) + 1
>>> print(wcnf.topw)
5
```

Note: Although it is not aligned with the WCNF format description, starting with the 0.1.5.dev8 release, PySAT is able to deal with WCNF formulas having not only integer clause weights but also weights represented as *floating point numbers*. Moreover, *negative weights* are also supported.

Additionally to classes `CNF` and `WCNF`, the module provides the extended classes `CNFPlus` and `WCNFPlus`. The only difference between `?CNF` and `?CNFPlus` is the support for *native* cardinality constraints provided by the MiniCard solver (see `pysat.card` for details). The corresponding variable in objects of `CNFPlus` (`WCNFPlus`, resp.) responsible for storing the AtMostK constraints is `atmosts` (`atms`, resp.). **Note** that at this point, AtMostK constraints in `WCNF` can be *hard* only.

Besides the implementations of CNF and WCNF formulas in PySAT, the `pysat.formula` module also provides a way to manage variable identifiers. This can be done with the use of the `IDPool` manager. With the use of the `CNF` and `WCNF` classes as well as with the `IDPool` variable manager, it is pretty easy to develop practical problem encoders into SAT or MaxSAT/MinSAT. As an example, a PHP formula encoder is shown below (the implementation can also be found in `examples.genhard.PHP`).

```
from pysat.formula import CNF
cnf = CNF()  # we will store the formula here

# nof_holes is given

# initializing the pool of variable ids
vpool = IDPool(start_from=1)
pigeon = lambda i, j: vpool.id('pigeon{0}@{1}'.format(i, j))

# placing all pigeons into holes
for i in range(1, nof_holes + 2):
    cnf.append([pigeon(i, j) for j in range(1, nof_holes + 1)])

# there cannot be more than 1 pigeon in a hole
pigeons = range(1, nof_holes + 2)
for j in range(1, nof_holes + 1):
    for comb in itertools.combinations(pigeons, 2):
        cnf.append([-pigeon(i, j) for i in comb])
```

Module details

1.1.3 Pseudo-Boolean encodings (`pysat.pb`)

List of classes

EncType	This class represents a C-like `enum` type for choosing the pseudo-Boolean encoding to use.
PBEnc	Abstract class responsible for the creation of pseudo-Boolean constraints encoded to a CNF formula.

Module description

Note: Functionality of this module is available only if the *PyPBLib* package is installed, e.g. from PyPI:

```
$ pip install pypblib
```

This module provides access to the basic functionality of the PyPBLib library developed by the Logic Optimization Group of the University of Lleida. PyPBLib provides a user with an extensive Python API to the well-known PBLib library[1]. Note the PyPBLib has a number of additional features that cannot be accessed through PySAT *at this point*. (One concrete example is a range of cardinality encodings, which clash with the internal `pysat.card` module.) If a user needs some functionality of PyPBLib missing in this module, he/she may apply PyPBLib as a standalone library, when working with PySAT.

A *pseudo-Boolean constraint* is a constraint of the form: $\left(\sum_{i=1}^{n} a_i \cdot x_i\right) \circ k$, where $a_i \in \mathbb{N}$, $x_i \in \{y_i, \neg y_i\}$, $y_i \in \mathbb{B}$, and $\circ \in \{\leq, =, \geq\}$. Pseudo-Boolean constraints arise in a number of important practical applications. Thus, several *encodings* of pseudo-Boolean constraints into CNF formulas are known[2]. The list of pseudo-Boolean encodings

[1] Tobias Philipp, Peter Steinke. *PBLib - A Library for Encoding Pseudo-Boolean Constraints into CNF*. SAT 2015. pp. 9-16

[2] Olivier Roussel, Vasco M. Manquinho. *Pseudo-Boolean and Cardinality Constraints*. Handbook of Satisfiability. 2009. pp. 695-733

supported by this module include BDD[3][4], sequential weight counters[5], sorting networks[3], adder networks[3], and binary merge[6]. Access to all cardinality encodings can be made through the main class of this module, which is *PBEnc*.

Module details

class pysat.pb.**EncType**

This class represents a C-like `enum` type for choosing the pseudo-Boolean encoding to use. The values denoting the encodings are:

```
best       = 0
bdd        = 1
seqcounter = 2
sortnetwrk = 3
adder      = 4
binmerge   = 5
```

The desired encoding can be selected either directly by its integer identifier, e.g. `2`, or by its alphabetical name, e.g. `EncType.seqcounter`.

All the encodings are produced and returned as a list of clauses in the `pysat.formula.CNF` format.

Note that the encoding type can be set to `best`, in which case the encoder selects one of the other encodings from the list (in most cases, this invokes the `bdd` encoder).

exception pysat.pb.**NoSuchEncodingError**

This exception is raised when creating an unknown LEQ, GEQ, or Equals constraint encoding.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pysat.pb.**PBEnc**

Abstract class responsible for the creation of pseudo-Boolean constraints encoded to a CNF formula. The class has three main *class methods* for creating LEQ, GEQ, and Equals constraints. Given (1) either a list of weighted literals or a list of unweighted literals followed by a list of weights, (2) an integer bound and an encoding type, each of these methods returns an object of class `pysat.formula.CNF` representing the resulting CNF formula.

Since the class is abstract, there is no need to create an object of it. Instead, the methods should be called directly as class methods, e.g. `PBEnc.atmost(wlits, bound)` or `PBEnc.equals(lits, weights, bound)`. An example usage is the following:

```
>>> from pysat.pb import *
>>> cnf = PBEnc.atmost(lits=[1, 2, 3], weights=[1, 2, 3], bound=3)
>>> print(cnf.clauses)
[[4], [-1, -5], [-2, -5], [5, -3, -6], [6]]
>>> cnf = PBEnc.equals(lits=[1, 2, 3], weights=[1, 2, 3], bound=3,␣
↪encoding=EncType.bdd)
>>> print(cnf.clauses)
[[4], [-5, -2], [-5, 2, -1], [-5, -1], [-6, 1], [-7, -2, 6], [-7, 2], [-7, 6], [-
↪8, -3, 5], [-8, 3, 7], [-8, 5, 7], [8]]
```

classmethod atleast(*lits*, *weights=None*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=0*)

A synonym for *PBEnc.geq()*.

[3] Niklas Eén, Niklas Sörensson. *Translating Pseudo-Boolean Constraints into SAT*. JSAT. vol. 2(1-4). 2006. pp. 1-26

[4] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell. *BDDs for Pseudo-Boolean Constraints - Revisited*. SAT. 2011. pp. 61-75

[5] Steffen Hölldobler, Norbert Manthey, Peter Steinke. *A Compact Encoding of Pseudo-Boolean Constraints into SAT*. KI. 2012. pp. 107-118

[6] Norbert Manthey, Tobias Philipp, Peter Steinke. *A More Compact Translation of Pseudo-Boolean Constraints into CNF Such That Generalized Arc Consistency Is Maintained*. KI. 2014. pp. 123-134

classmethod atmost (*lits*, *weights=None*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=0*)
A synonim for `PBEnc.leq()`.

classmethod equals (*lits*, *weights=None*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=0*)
This method can be used for creating a CNF encoding of a weighted EqualsK constraint, i.e. of $\sum_{i=1}^{n} a_i \cdot x_i = k$. The method shares the arguments and the return type with method `PBEnc.leq()`. Please, see it for details.

classmethod geq (*lits*, *weights=None*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=0*)
This method can be used for creating a CNF encoding of a GEQ (weighted AtLeastK) constraint, i.e. of $\sum_{i=1}^{n} a_i \cdot x_i \geq k$. The method shares the arguments and the return type with method `PBEnc.leq()`. Please, see it for details.

classmethod leq (*lits*, *weights=None*, *bound=1*, *top_id=None*, *vpool=None*, *encoding=0*)
This method can be used for creating a CNF encoding of a LEQ (weighted AtMostK) constraint, i.e. of $\sum_{i=1}^{n} a_i \cdot x_i \leq k$. The resulting set of clauses is returned as an object of class `formula.CNF`.

The input list of literals can contain either integers or pairs `(l, w)`, where `l` is an integer literal and `w` is an integer weight. The latter can be done only if no `weights` are specified separately. The type of encoding to use can be specified using the `encoding` parameter. By default, it is set to `EncType.best`, i.e. it is up to the PBLib encoder to choose the encoding type.

Parameters

- **lits** (*iterable(int)*) – a list of literals in the sum.
- **weights** (*iterable(int)*) – a list of weights
- **bound** (*int*) – the value of bound k.
- **top_id** (*integer or None*) – top variable identifier used so far.
- **vpool** (`IDPool`) – variable pool for counting the number of variables.
- **encoding** (*integer*) – identifier of the encoding to use.

Return type `pysat.formula.CNF`

1.1.4 SAT solvers' API (pysat.solvers)

List of classes

`SolverNames`	This class serves to determine the solver requested by a user given a string name.
`Solver`	Main class for creating and manipulating a SAT solver.
`Cadical`	CaDiCaL SAT solver.
`Glucose3`	Glucose 3 SAT solver.
`Glucose4`	Glucose 4.1 SAT solver.
`Lingeling`	Lingeling SAT solver.
`MapleChrono`	MapleLCMDistChronoBT SAT solver.
`MapleCM`	MapleCM SAT solver.
`Maplesat`	MapleCOMSPS_LRB SAT solver.
`Minicard`	Minicard SAT solver.
`Minisat22`	MiniSat 2.2 SAT solver.
`MinisatGH`	MiniSat SAT solver (version from github).

Module description

This module provides *incremental* access to a few modern SAT solvers. The solvers supported by PySAT are:

- CaDiCaL (rel-1.0.3)
- Glucose (3.0)
- Glucose (4.1)
- Lingeling (bbc-9230380-160707)
- MapleLCMDistChronoBT (SAT competition 2018 version)
- MapleCM (SAT competition 2018 version)
- Maplesat (MapleCOMSPS_LRB)
- Minicard (1.2)
- Minisat (2.2 release)
- Minisat (GitHub version)

All solvers can be accessed through a unified MiniSat-like[1] incremental[2] interface described below.

The module provides direct access to all supported solvers using the corresponding classes `Glucose3`, `Glucose4`, `Lingeling`, `MapleChrono`, `MapleCM`, `Maplesat`, `Minicard`, `Minisat22`, and `MinisatGH`. However, the solvers can also be accessed through the common base class `Solver` using the solver `name` argument. For example, both of the following pieces of code create a copy of the `Glucose3` solver:

```
>>> from pysat.solvers import Glucose3, Solver
>>>
>>> g = Glucose3()
>>> g.delete()
>>>
>>> s = Solver(name='g3')
>>> s.delete()
```

The `pysat.solvers` module is designed to create and manipulate SAT solvers as *oracles*, i.e. it does not give access to solvers' internal parameters such as variable polarities or activities. PySAT provides a user with the following basic SAT solving functionality:

- creating and deleting solver objects
- adding individual clauses and formulas to solver objects
- making SAT calls with or without assumptions
- propagating a given set of assumption literals
- setting preferred polarities for a (sub)set of variables
- extracting a model of a satisfiable input formula
- enumerating models of an input formula
- extracting an unsatisfiable core of an unsatisfiable formula
- extracting a DRUP proof logged by the solver

[1] Niklas Eén, Niklas Sörensson. *An Extensible SAT-solver*. SAT 2003. pp. 502-518

[2] Niklas Eén, Niklas Sörensson. *Temporal induction by incremental SAT solving*. Electr. Notes Theor. Comput. Sci. 89(4). 2003. pp. 543-560

PySAT supports both non-incremental and incremental SAT solving. Incrementality can be achieved with the use of the MiniSat-like *assumption-based* interface[2]. It can be helpful if multiple calls to a SAT solver are needed for the same formula using different sets of "assumptions", e.g. when doing consecutive SAT calls for formula $\mathcal{F} \wedge (a_{i_1} \wedge \ldots \wedge a_{i_1+j_1})$ and $\mathcal{F} \wedge (a_{i_2} \wedge \ldots \wedge a_{i_2+j_2})$, where every a_{l_k} is an assumption literal.

There are several advantages of using assumptions: (1) it enables one to *keep and reuse* the clauses learnt during previous SAT calls at a later stage and (2) assumptions can be easily used to extract an *unsatisfiable core* of the formula. A drawback of assumption-based SAT solving is that the clauses learnt are longer (they typically contain many assumption literals), which makes the SAT calls harder.

In PySAT, assumptions should be provided as a list of literals given to the `solve()` method:

```
>>> from pysat.solvers import Solver
>>> s = Solver()
>>>
... # assume that solver s is fed with a formula
>>>
>>> s.solve()  # a simple SAT call
True
>>>
>>> s.solve(assumptions=[1, -2, 3])  # a SAT call with assumption literals
False
>>> s.get_core()  # extracting an unsatisfiable core
[3, 1]
```

In order to shorten the description of the module, the classes providing direct access to the individual solvers, i.e. classes `Cadical`, `Glucose3`, `Glucose4`, `Lingeling`, `MapleChrono`, `MapleCM`, `Maplesat`, `Minicard`, `Minisat22`, and `MinisatGH`, are **omitted**. They replicate the interface of the base class *`Solver`* and, thus, can be used the same exact way.

Module details

exception `pysat.solvers.`**`NoSuchSolverError`**

This exception is raised when creating a new SAT solver whose name does not match any name in *`SolverNames`*. The list of *known* solvers includes the names *'cadical'*, *'glucose3'*, *'glucose4'*, *'lingeling'*, *'maplechrono'*, *'maplecm'*, *'maplesat'*, *'minicard'*, *'minisat22'*, and *'minisatgh'*.

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `pysat.solvers.`**`Solver`**`(`*name='m22'*, *bootstrap_with=None*, *use_timer=False*, ***kwargs*`)`

Main class for creating and manipulating a SAT solver. Any available SAT solver can be accessed as an object of this class and so *`Solver`* can be seen as a wrapper for all supported solvers.

The constructor of *`Solver`* has only one mandatory argument `name`, while all the others are default. This means that explicit solver constructors, e.g. `Glucose3` or `MinisatGH` etc., have only default arguments.

Parameters

- **`name`** (*`str`*) – solver's name (see *`SolverNames`*).
- **`bootstrap_with`** (*`iterable(iterable(int))`*) – a list of clauses for solver initialization.
- **`use_timer`** (*`bool`*) – whether or not to measure SAT solving time.

The `bootstrap_with` argument is useful when there is an input CNF formula to feed the solver with. The argument expects a list of clauses, each clause being a list of literals, i.e. a list of integers.

If set to `True`, the `use_timer` parameter will force the solver to accumulate the time spent by all SAT calls made with this solver but also to keep time of the last SAT call.

Once created and used, a solver must be deleted with the `delete()` method. Alternatively, if created using the `with` statement, deletion is done automatically when the end of the `with` block is reached.

Given the above, a couple of examples of solver creation are the following:

```
>>> from pysat.solvers import Solver, Minisat22
>>>
>>> s = Solver(name='g4')
>>> s.add_clause([-1, 2])
>>> s.add_clause([-1, -2])
>>> s.solve()
True
>>> print(s.get_model())
[-1, -2]
>>> s.delete()
>>>
>>> with Minisat22(bootstrap_with=[[-1, 2], [-1, -2]]) as m:
...     m.solve()
True
...     print(m.get_model())
[-1, -2]
```

Note that while all explicit solver classes necessarily have default arguments `bootstrap_with` and `use_timer`, solvers `Cadical`, `Lingeling`, `Glucose3`, `Glucose4`, `MapleChrono`, `MapleCM` and `Maplesat` can have additional default arguments. One such argument supported by `Glucose3` and `Glucose4` but also by `Cadical`, `Lingeling`, `MapleChrono`, `MapleCM`, and `Maplesat` is DRUP proof logging. This can be enabled by setting the `with_proof` argument to `True` (`False` by default):

```
>>> from pysat.solvers import Lingeling
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(nof_holes=2)  # pigeonhole principle for 3 pigeons
>>>
>>> with Lingeling(bootstrap_with=cnf.clauses, with_proof=True) as l:
...     l.solve()
False
...     l.get_proof()
['-5 0', '6 0', '-2 0', '-4 0', '1 0', '3 0', '0']
```

Additionally and in contrast to `Cadical` and `Lingeling`, both `Glucose3` and `Glucose4` have one more default argument `incr` (`False` by default), which enables incrementality features introduced in Glucose3[3]. To summarize, the additional arguments of Glucose are:

Parameters

- **`incr`** (*bool*) – enable the incrementality features of Glucose3[3].
- **`with_proof`** (*bool*) – enable proof logging in the DRUP format.

`add_atmost` (*lits*, *k*, *no_return=True*)

This method is responsible for adding a new *native* AtMostK (see *`pysat.card`*) constraint into `Minicard`.

Note that none of the other solvers supports native AtMostK constraints.

[3] Gilles Audemard, Jean-Marie Lagniez, Laurent Simon. *Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction*. SAT 2013. pp. 309-317

An AtMostK constraint is $\sum_{i=1}^{n} x_i \leq k$. A native AtMostK constraint should be given as a pair `lits` and `k`, where `lits` is a list of literals in the sum.

Parameters

- **lits** (*iterable(int)*) – a list of literals.
- **k** (*int*) – upper bound on the number of satisfied literals
- **no_return** (*bool*) – check solver's internal formula and return the result, if set to `False`.

Return type bool if `no_return` is set to `False`.

A usage example is the following:

```
>>> s = Solver(name='mc', bootstrap_with=[[1], [2], [3]])
>>> s.add_atmost(lits=[1, 2, 3], k=2, no_return=False)
False
>>> # the AtMostK constraint is in conflict with initial unit clauses
```

add_clause (*clause*, *no_return=True*)

This method is used to add a single clause to the solver. An optional argument `no_return` controls whether or not to check the formula's satisfiability after adding the new clause.

Parameters

- **clause** (*iterable(int)*) – an iterable over literals.
- **no_return** (*bool*) – check solver's internal formula and return the result, if set to `False`.

Return type bool if `no_return` is set to `False`.

Note that a clause can be either a `list` of integers or another iterable type over integers, e.g. `tuple` or `set` among others.

A usage example is the following:

```
>>> s = Solver(bootstrap_with=[[-1, 2], [-1, -2]])
>>> s.add_clause([1], no_return=False)
False
```

append_formula (*formula*, *no_return=True*)

This method can be used to add a given list of clauses into the solver.

Parameters

- **formula** (*iterable(iterable(int))*) – a list of clauses.
- **no_return** (*bool*) – check solver's internal formula and return the result, if set to `False`.

The `no_return` argument is set to `True` by default.

Return type bool if `no_return` is set to `False`.

```
>>> cnf = CNF()
... # assume the formula contains clauses
>>> s = Solver()
>>> s.append_formula(cnf.clauses, no_return=False)
True
```

clear_interrupt()
Clears a previous interrupt. If a limited SAT call was interrupted using the *interrupt()* method, this method **must be called** before calling the SAT solver again.

conf_budget(*budget=- 1*)
Set limit (i.e. the upper bound) on the number of conflicts in the next limited SAT call (see *solve_limited()*). The limit value is given as a `budget` variable and is an integer greater than `0`. If the budget is set to `0` or `-1`, the upper bound on the number of conflicts is disabled.

Parameters budget (*int*) – the upper bound on the number of conflicts.

Example:

```
>>> from pysat.solvers import MinisatGH
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(nof_holes=20)  # PHP20 is too hard for a SAT solver
>>> m = MinisatGH(bootstrap_with=cnf.clauses)
>>>
>>> m.conf_budget(2000)  # getting at most 2000 conflicts
>>> print(m.solve_limited())  # making a limited oracle call
None
>>> m.delete()
```

delete()
Solver destructor, which must be called explicitly if the solver is to be removed. This is not needed inside an `with` block.

enum_models(*assumptions=[]*)
This method can be used to enumerate models of a CNF formula. It can be used as a standard Python iterator. The method can be used without arguments but also with an argument `assumptions`, which is a list of literals to "assume".

Parameters assumptions (*iterable(int)*) – a list of assumption literals.

Return type list(int)

Example:

```
>>> with Solver(bootstrap_with=[[-1, 2], [-2, 3]]) as s:
...     for m in s.enum_models():
...         print(m)
[-1, -2, -3]
[-1, -2, 3]
[-1, 2, 3]
[1, 2, 3]
>>>
>>> with Solver(bootstrap_with=[[-1, 2], [-2, 3]]) as s:
...     for m in s.enum_models(assumptions=[1]):
...         print(m)
[1, 2, 3]
```

get_core()
This method is to be used for extracting an unsatisfiable core in the form of a subset of a given set of assumption literals, which are responsible for unsatisfiability of the formula. This can be done only if the previous SAT call returned `False` (*UNSAT*). Otherwise, `None` is returned.

Return type list(int) or `None`.

Usage example:

```
>>> from pysat.solvers import Minisat22
>>> m = Minisat22()
>>> m.add_clause([-1, 2])
>>> m.add_clause([-2, 3])
>>> m.add_clause([-3, 4])
>>> m.solve(assumptions=[1, 2, 3, -4])
False
>>> print(m.get_core())  # literals 2 and 3 are not in the core
[-4, 1]
>>> m.delete()
```

get_model()

The method is to be used for extracting a satisfying assignment for a CNF formula given to the solver. A model is provided if a previous SAT call returned `True`. Otherwise, `None` is reported.

Return type list(int) or `None`.

Example:

```
>>> from pysat.solvers import Solver
>>> s = Solver()
>>> s.add_clause([-1, 2])
>>> s.add_clause([-1, -2])
>>> s.add_clause([1, -2])
>>> s.solve()
True
>>> print(s.get_model())
[-1, -2]
>>> s.delete()
```

get_proof()

A DRUP proof can be extracted using this method if the solver was set up to provide a proof. Otherwise, the method returns `None`.

Return type list(str) or `None`.

Example:

```
>>> from pysat.solvers import Solver
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(nof_holes=3)
>>> with Solver(name='g4', with_proof=True) as g:
...     g.append_formula(cnf.clauses)
...     g.solve()
False
...     print(g.get_proof())
['-8 4 1 0', '-10 0', '-2 0', '-4 0', '-8 0', '-6 0', '0']
```

get_status()

The result of a previous SAT call is stored in an internal variable and can be later obtained using this method.

Return type Boolean or `None`.

`None` is returned if a previous SAT call was interrupted.

interrupt()

Interrupt the execution of the current *limited* SAT call (see `solve_limited()`). Can be used to enforce time limits using timer objects. The interrupt must be cleared before performing another SAT call (see `clear_interrupt()`).

Behaviour is **undefined** if used to interrupt a *non-limited* SAT call (see `solve()`).

Example:

```
>>> from pysat.solvers import MinisatGH
>>> from pysat.examples.genhard import PHP
>>> from threading import Timer
>>>
>>> cnf = PHP(nof_holes=20)  # PHP20 is too hard for a SAT solver
>>> m = MinisatGH(bootstrap_with=cnf.clauses)
>>>
>>> def interrupt(s):
>>>     s.interrupt()
>>>
>>> timer = Timer(10, interrupt, [m])
>>> timer.start()
>>>
>>> print(m.solve_limited())
None
>>> m.delete()
```

new(*name='m22'*, *bootstrap_with=None*, *use_timer=False*, ***kwargs*)
The actual solver constructor invoked from `__init__()`. Chooses the solver to run, based on its name. See `Solver` for the parameters description.

Raises `NoSuchSolverError` – if there is no solver matching the given name.

nof_clauses()
This method returns the number of clauses currently appearing in the formula given to the solver.

Return type int.

Example:

```
>>> s = Solver(bootstrap_with=[[-1, 2], [-2, 3]])
>>> s.nof_clauses()
2
```

nof_vars()
This method returns the number of variables currently appearing in the formula given to the solver.

Return type int.

Example:

```
>>> s = Solver(bootstrap_with=[[-1, 2], [-2, 3]])
>>> s.nof_vars()
3
```

prop_budget(*budget=- 1*)
Set limit (i.e. the upper bound) on the number of propagations in the next limited SAT call (see `solve_limited()`). The limit value is given as a `budget` variable and is an integer greater than `0`. If the budget is set to `0` or `-1`, the upper bound on the number of conflicts is disabled.

Parameters budget (`int`) – the upper bound on the number of propagations.

Example:

```
>>> from pysat.solvers import MinisatGH
>>> from pysat.examples.genhard import Parity
>>>
>>> cnf = Parity(size=10)  # too hard for a SAT solver
>>> m = MinisatGH(bootstrap_with=cnf.clauses)
>>>
>>> m.prop_budget(100000)  # doing at most 100000 propagations
>>> print(m.solve_limited())  # making a limited oracle call
None
>>> m.delete()
```

propagate (*assumptions=[]*, *phase_saving=0*)

The method takes a list of assumption literals and does unit propagation of each of these literals consecutively. A Boolean status is returned followed by a list of assigned (assumed and also propagated) literals. The status is `True` if no conflict arised during propagation. Otherwise, the status is `False`. Additionally, a user may specify an optional argument `phase_saving` (`0` by default) to enable MiniSat-like phase saving.

Note that only MiniSat-like solvers support this functionality (e.g. `Cadical` and `Lingeling` do not support it).

Parameters

- **assumptions** (*iterable(int)*) – a list of assumption literals.
- **phase_saving** (*int*) – enable phase saving (can be `0`, `1`, and `2`).

Return type tuple(bool, list(int))

Usage example:

```
>>> from pysat.solvers import Glucose3
>>> from pysat.card import *
>>>
>>> cnf = CardEnc.atmost(lits=range(1, 6), bound=1, encoding=EncType.pairwise)
>>> g = Glucose3(bootstrap_with=cnf.clauses)
>>>
>>> g.propagate(assumptions=[1])
(True, [1, -2, -3, -4, -5])
>>>
>>> g.add_clause([2])
>>> g.propagate(assumptions=[1])
(False, [])
>>>
>>> g.delete()
```

set_phases (*literals=[]*)

The method takes a list of literals as an argument and sets *phases* (or MiniSat-like *polarities*) of the corresponding variables respecting the literals. For example, if a given list of literals is `[1, -513]`, the solver will try to set variable x_1 to true while setting x_{513} to false.

Note that once these preferences are specified, `MinisatGH` and `Lingeling` will always respect them when branching on these variables. However, solvers `Glucose3`, `Glucose4`, `MapleChrono`, `MapleCM`, `Maplesat`, `Minisat22`, and `Minicard` can redefine the preferences in any of the following SAT calls due to the phase saving heuristic.

Also **note** that `Cadical` does not support this functionality.

Parameters **literals** (*iterable(int)*) – a list of literals.

Usage example:

```
>>> from pysat.solvers import Glucose3
>>>
>>> g = Glucose3(bootstrap_with=[[1, 2]])
>>> # the formula has 3 models: [-1, 2], [1, -2], [1, 2]
>>>
>>> g.set_phases(literals=[1, 2])
>>> g.solve()
True
>>> g.get_model()
[1, 2]
>>>
>>> g.delete()
```

solve(*assumptions=[]*)

This method is used to check satisfiability of a CNF formula given to the solver (see methods `add_clause()` and `append_formula()`). Unless interrupted with SIGINT, the method returns either `True` or `False`.

Incremental SAT calls can be made with the use of assumption literals. (**Note** that the `assumptions` argument is optional and disabled by default.)

Parameters **assumptions** (`iterable(int)`) – a list of assumption literals.

Return type Boolean or `None`.

Example:

```
>>> from pysat.solvers import Solver
>>> s = Solver(bootstrap_with=[[-1, 2], [-2, 3])
>>> s.solve()
True
>>> s.solve(assumptions=[1, -3])
False
>>> s.delete()
```

solve_limited(*assumptions=[]*)

This method is used to check satisfiability of a CNF formula given to the solver (see methods `add_clause()` and `append_formula()`), taking into account the upper bounds on the *number of conflicts* (see `conf_budget()`) and the *number of propagations* (see `prop_budget()`). If the number of conflicts or propagations is set to be larger than 0 then the following SAT call done with `solve_limited()` will not exceed these values, i.e. it will be *incomplete*. Otherwise, such a call will be identical to `solve()`.

As soon as the given upper bound on the number of conflicts or propagations is reached, the SAT call is dropped returning `None`, i.e. *unknown*. `None` can also be returned if the call is interrupted by SIGINT. Otherwise, the method returns `True` or `False`.

Note that only MiniSat-like solvers support this functionality (e.g. `Cadical` and `Lingeling` do not support it).

Incremental SAT calls can be made with the use of assumption literals. (**Note** that the `assumptions` argument is optional and disabled by default.)

Parameters **assumptions** (`iterable(int)`) – a list of assumption literals.

Return type Boolean or `None`.

Doing limited SAT calls can be of help if it is known that *complete* SAT calls are too expensive. For instance, it can be useful when minimizing unsatisfiable cores in MaxSAT (see `pysat.examples.RC2.minimize_core()` also shown below).

Also and besides supporting deterministic interruption based on `conf_budget()` and/or `prop_budget()`, limited SAT calls support *deterministic* and *non-deterministic* interruption from inside a Python script. See the `interrupt()` and `clear_interrupt()` methods for details.

Usage example:

```
... # assume that a SAT oracle is set up to contain an unsatisfiable
... # formula, and its core is stored in variable "core"
oracle.conf_budget(1000)  # getting at most 1000 conflicts be call

i = 0
while i < len(core):
    to_test = core[:i] + core[(i + 1):]

    # doing a limited call
    if oracle.solve_limited(assumptions=to_test) == False:
        core = to_test
    else:  # True or *unknown*
        i += 1
```

time()

Get the time spent when doing the last SAT call. **Note** that the time is measured only if the `use_timer` argument was previously set to `True` when creating the solver (see `Solver` for details).

Return type float.

Example usage:

```
>>> from pysat.solvers import Solver
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(nof_holes=10)
>>> with Solver(bootstrap_with=cnf.clauses, use_timer=True) as s:
...     print(s.solve())
False
...     print('{0:.2f}s'.format(s.time()))
150.16s
```

time_accum()

Get the time spent for doing all SAT calls accumulated. **Note** that the time is measured only if the `use_timer` argument was previously set to `True` when creating the solver (see `Solver` for details).

Return type float.

Example usage:

```
>>> from pysat.solvers import Solver
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(nof_holes=10)
>>> with Solver(bootstrap_with=cnf.clauses, use_timer=True) as s:
...     print(s.solve(assumptions=[1]))
False
...     print('{0:.2f}s'.format(s.time()))
1.76s
...     print(s.solve(assumptions=[-1]))
False
...     print('{0:.2f}s'.format(s.time()))
113.58s
```

(continues on next page)

(continued from previous page)

```
...     print('{0:.2f}s'.format(s.time_accum()))
115.34s
```

class pysat.solvers.**SolverNames**

This class serves to determine the solver requested by a user given a string name. This allows for using several possible names for specifying a solver.

```
cadical     = ('cd', 'cdl', 'cadical')
glucose3    = ('g3', 'g30', 'glucose3', 'glucose30')
glucose4    = ('g4', 'g41', 'glucose4', 'glucose41')
lingeling   = ('lgl', 'lingeling')
maplechrono = ('mcb', 'chrono', 'maplechrono')
maplecm     = ('mcm', 'maplecm')
maplesat    = ('mpl', 'maple', 'maplesat')
minicard    = ('mc', 'mcard', 'minicard')
minisat22   = ('m22', 'msat22', 'minisat22')
minisatgh   = ('mgh', 'msat-gh', 'minisat-gh')
```

As a result, in order to select Glucose3, a user can specify the solver's name: either `'g3'`, `'g30'`, `'glucose3'`, or `'glucose30'`. *Note that the capitalized versions of these names are also allowed.*

1.2 Supplementary `examples` package

1.2.1 Fu&Malik MaxSAT algorithm (`pysat.examples.fm`)

List of classes

FM	A non-incremental implementation of the FM (Fu&Malik, or WMSU1) algorithm.

Module description

This module implements a variant of the seminal core-guided MaxSAT algorithm originally proposed by[1] and then improved and modified further in[2][3][4][5]. Namely, the implementation follows the WMSU1 variant[5] of the algorithm extended to the case of *weighted partial* formulas.

The implementation can be used as an executable (the list of available command-line options can be shown using `fm.py -h`) in the following way:

```
$ xzcat formula.wcnf.xz
p wcnf 3 6 4
1 1 0
1 2 0
1 3 0
4 -1 -2 0
```

(continues on next page)

[1] Zhaohui Fu, Sharad Malik. *On Solving the Partial MAX-SAT Problem*. SAT 2006. pp. 252-265

[2] Joao Marques-Silva, Jordi Planes. *On Using Unsatisfiability for Solving Maximum Satisfiability*. CoRR abs/0712.1097. 2007

[3] Joao Marques-Silva, Vasco M. Manquinho. *Towards More Effective Unsatisfiability-Based Maximum Satisfiability Algorithms*. SAT 2008. pp. 225-230

[4] Carlos Ansótegui, Maria Luisa Bonet, Jordi Levy. *Solving (Weighted) Partial MaxSAT through Satisfiability Testing*. SAT 2009. pp. 427-440

[5] Vasco M. Manquinho, Joao Marques Silva, Jordi Planes. *Algorithms for Weighted Boolean Optimization*. SAT 2009. pp. 495-508

(continued from previous page)

```
4 -1 -3 0
4 -2 -3 0

$ fm.py -c cardn -s glucose3 -vv formula.wcnf.xz
c cost: 1; core sz: 2
c cost: 2; core sz: 3
s OPTIMUM FOUND
o 2
v -1 -2 3 0
c oracle time: 0.0001
```

Alternatively, the algorithm can be accessed and invoked through the standard `import` interface of Python, e.g.

```
>>> from pysat.examples.fm import FM
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF(from_file='formula.wcnf.xz')
>>>
>>> fm = FM(wcnf, verbose=0)
>>> fm.compute()  # set of hard clauses should be satisfiable
True
>>> print(fm.cost) # cost of MaxSAT solution should be 2
>>> 2
>>> print(fm.model)
[-1, -2, 3]
```

Module details

class `examples.fm.`**FM**(*formula*, *enc=0*, *solver='m22'*, *verbose=1*)

A non-incremental implementation of the FM (Fu&Malik, or WMSU1) algorithm. The algorithm (see details in[5]) is *core-guided*, i.e. it solves maximum satisfiability with a series of unsatisfiability oracle calls, each producing an unsatisfiable core. The clauses involved in an unsatisfiable core are *relaxed* and a new AtMost1 constraint on the corresponding *relaxation variables* is added to the formula. The process gets a bit more sophisticated in the case of weighted formulas because of the *clause weight splitting* technique.

The constructor of *FM* objects receives a target `WCNF` MaxSAT formula, an identifier of the cardinality encoding to use, a SAT solver name, and a verbosity level. Note that the algorithm uses the `pairwise` (see *card.EncType*) cardinality encoding by default, while the default SAT solver is MiniSat22 (referred to as `'m22'`, see *SolverNames* for details). The default verbosity level is `1`.

Parameters

- **formula** (`WCNF`) – input MaxSAT formula
- **enc** (*int*) – cardinality encoding to use
- **solver** (*str*) – name of SAT solver
- **verbose** (*int*) – verbosity level

_compute()

This method implements WMSU1 algorithm. The method is essentially a loop, which at each iteration calls the SAT oracle to decide whether the working formula is satisfiable. If it is, the method derives a model (stored in variable `self.model`) and returns. Otherwise, a new unsatisfiable core of the formula is extracted and processed (see *treat_core()*), and the algorithm proceeds.

compute()

Compute a MaxSAT solution. First, the method checks whether or not the set of hard clauses is satisfiable.

If not, the method returns `False`. Otherwise, add soft clauses to the oracle and call the MaxSAT algorithm (see `_compute()`).

Note that the soft clauses are added to the oracles after being augmented with additional *selector* literals. The selectors literals are then used as *assumptions* when calling the SAT oracle and are needed for extracting unsatisfiable cores.

delete()

Explicit destructor of the internal SAT oracle.

init(*with_soft=True*)

The method for the SAT oracle initialization. Since the oracle is is used non-incrementally, it is reinitialized at every iteration of the MaxSAT algorithm (see `reinit()`). An input parameter `with_soft` (`False` by default) regulates whether or not the formula's soft clauses are copied to the oracle.

Parameters **with_soft** (*bool*) – copy formula's soft clauses to the oracle or not

oracle_time()

Method for calculating and reporting the total SAT solving time.

reinit()

This method calls `delete()` and `init()` to reinitialize the internal SAT oracle. This is done at every iteration of the MaxSAT algorithm.

relax_core()

Relax and bound the core.

After unsatisfiable core splitting, this method is called. If the core contains only one clause, i.e. this clause cannot be satisfied together with the hard clauses of the formula, the formula gets augmented with the negation of the clause (see `remove_unit_core()`).

Otherwise (if the core contains more than one clause), every clause c of the core is *relaxed*. This means a new *relaxation literal* is added to the clause, i.e. $c \leftarrow c \vee r$, where r is a fresh (unused) relaxation variable. After the clauses get relaxed, a new cardinality encoding is added to the formula enforcing the sum of the new relaxation variables to be not greater than 1, $\sum_{c \in \phi} r \leq 1$, where ϕ denotes the unsatisfiable core.

remove_unit_core()

If an unsatisfiable core contains only one clause c, this method is invoked to add a bunch of new unit size hard clauses. As a result, the SAT oracle gets unit clauses $(\neg l)$ for all literals l in clause c.

split_core(*minw*)

Split clauses in the core whenever necessary.

Given a list of soft clauses in an unsatisfiable core, the method is used for splitting clauses whose weights are greater than the minimum weight of the core, i.e. the `minw` value computed in `treat_core()`. Each clause $(c \vee \neg s, w)$, s.t. $w > minw$ and s is its selector literal, is split into clauses (1) clause $(c \vee \neg s, minw)$ and (2) a residual clause $(c \vee \neg s', w - minw)$. Note that the residual clause has a fresh selector literal s' different from s.

Parameters **minw** (*int*) – minimum weight of the core

treat_core()

Now that the previous SAT call returned UNSAT, a new unsatisfiable core should be extracted and relaxed. Core extraction is done through a call to the `pysat.solvers.Solver.get_core()` method, which returns a subset of the selector literals deemed responsible for unsatisfiability.

After the core is extracted, its *minimum weight* `minw` is computed, i.e. it is the minimum weight among the weights of all soft clauses involved in the core (see[5]). Note that the cost of the MaxSAT solution is incremented by `minw`.

Clauses that have weight larger than `minw` are split (see `split_core()`). Afterwards, all clauses of the unsatisfiable core are relaxed (see `relax_core()`).

1.2.2 Hard formula generator (`pysat.examples.genhard`)

List of classes

`CB`	Mutilated chessboard principle (CB).
`GT`	Generator of ordering (or *greater than*, GT) principle formulas.
`PAR`	Generator of the parity principle (PAR) formulas.
`PHP`	Generator of k pigeonhole principle (k-PHP) formulas.

Module description

This module is designed to provide a few examples illustrating how PySAT can be used for encoding practical problems into CNF formulas. These include combinatorial principles that are widely studied from the propositional proof complexity perspective. Namely, encodings for the following principles are implemented: *pigeonhole principle* (`PHP`)[1], *ordering (greater-than) principle* (`GT`)[2], *mutilated chessboard principle* (`CB`)[3], and *parity principle* (`PAR`)[4].

The module can be used as an executable (the list of available command-line options can be shown using `genhard.py -h`) in the following way

```
$ genhard.py -t php -n 3 -v
c PHP formula for 4 pigeons and 3 holes
c (pigeon, hole) pair: (1, 1); bool var: 1
c (pigeon, hole) pair: (1, 2); bool var: 2
c (pigeon, hole) pair: (1, 3); bool var: 3
c (pigeon, hole) pair: (2, 1); bool var: 4
c (pigeon, hole) pair: (2, 2); bool var: 5
c (pigeon, hole) pair: (2, 3); bool var: 6
c (pigeon, hole) pair: (3, 1); bool var: 7
c (pigeon, hole) pair: (3, 2); bool var: 8
c (pigeon, hole) pair: (3, 3); bool var: 9
c (pigeon, hole) pair: (4, 1); bool var: 10
c (pigeon, hole) pair: (4, 2); bool var: 11
c (pigeon, hole) pair: (4, 3); bool var: 12
p cnf 12 22
1 2 3 0
4 5 6 0
7 8 9 0
10 11 12 0
-1 -4 0
-1 -7 0
-1 -10 0
-4 -7 0
-4 -10 0
-7 -10 0
-2 -5 0
-2 -8 0
-2 -11 0
-5 -8 0
-5 -11 0
```

(continues on next page)

[1] Stephen A. Cook, Robert A. Reckhow. *The Relative Efficiency of Propositional Proof Systems*. J. Symb. Log. 44(1). 1979. pp. 36-50

[2] Balakrishnan Krishnamurthy. *Short Proofs for Tricky Formulas*. Acta Informatica 22(3). 1985. pp. 253-275

[3] Michael Alekhnovich. *Mutilated Chessboard Problem Is Exponentially Hard For Resolution*. Theor. Comput. Sci. 310(1-3). 2004. pp. 513-525

[4] Miklós Ajtai. *Parity And The Pigeonhole Principle*. Feasible Mathematics. 1990. pp. 1–24

(continued from previous page)

```
-8 -11 0
-3 -6 0
-3 -9 0
-3 -12 0
-6 -9 0
-6 -12 0
-9 -12 0
```

Alternatively, each of the considered problem encoders can be accessed with the use of the standard `import` interface of Python, e.g.

```
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(3)
>>> print(cnf.nv, len(cnf.clauses))
12 22
```

Given this example, observe that classes `PHP`, `GT`, `CB`, and `PAR` inherit from class `pysat.formula.CNF` and, thus, their corresponding clauses can accessed through variable `.clauses`.

Module details

1.2.3 Minimum/minimal hitting set solver (`pysat.examples.hitman`)

List of classes

`Hitman`	A cardinality-/subset-minimal hitting set enumerator.

Module description

A SAT-based implementation of an implicit minimal hitting set[1] enumerator. The implementation is capable of computing/enumerating cardinality- and subset-minimal hitting sets of a given set of sets. Cardinality-minimal hitting set enumeration can be seen as ordered (sorted by size) subset-minimal hitting enumeration.

The minimal hitting set problem is trivially formulated as a MaxSAT formula in WCNF, as follows. Assume $E = \{e_1, \ldots, e_n\}$ to be a universe of elements. Also assume there are k sets to hit: $s_i = \{e_{i,1}, \ldots, e_{i,j_i}\}$ s.t. $e_{i,l} \in E$. Every set $s_i = \{e_{i,1}, \ldots, e_{i,j_i}\}$ is translated into a hard clause $(e_{i,1} \vee \ldots \vee e_{i,j_i})$. This results in the set of hard clauses having size k. The set of soft clauses comprises unit clauses of the form $(\neg e_j)$ s.t. $e_j \in E$, each having weight 1.

Taking into account this problem formulation as MaxSAT, ordered hitting enumeration is done with the use of the state-of-the-art MaxSAT solver called `RC2`[2][3][4] while unordered hitting set enumeration is done through the *minimal correction subset* (MCS) enumeration, e.g. using the *`LBX`*-[5] or *`MCSls`*-like[6] MCS enumerators.

`Hitman` supports hitting set enumeration in the *implicit* manner, i.e. when sets to hit can be added on the fly as well as hitting sets can be blocked on demand.

[1] Erick Moreno-Centeno, Richard M. Karp. *The Implicit Hitting Set Approach to Solve Combinatorial Optimization Problems with an Application to Multigenome Alignment*. Operations Research 61(2). 2013. pp. 453-468

[2] António Morgado, Carmine Dodaro, Joao Marques-Silva. *Core-Guided MaxSAT with Soft Cardinality Constraints*. CP 2014. pp. 564-573

[3] António Morgado, Alexey Ignatiev, Joao Marques-Silva. *MSCG: Robust Core-Guided MaxSAT Solving*. JSAT 9. 2014. pp. 129-134

[4] Alexey Ignatiev, António Morgado, Joao Marques-Silva. *RC2: a Python-based MaxSAT Solver*. MaxSAT Evaluation 2018. p. 22

[5] Carlos Mencía, Alessandro Previti, Joao Marques-Silva. *Literal-Based MCS Extraction*. IJCAI. 2015. pp. 1973-1979

[6] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, Anton Belov. *On Computing Minimal Correction Subsets*. IJCAI. 2013. pp. 615-622

An example usage of *Hitman* through the Python `import` interface is shown below. Here we target unordered subset-minimal hitting set enumeration.

```
>>> from pysat.examples.hitman import Hitman
>>>
>>> h = Hitman(solver='m22', htype='lbx')
>>> # adding sets to hit
>>> h.hit([1, 2, 3])
>>> h.hit([1, 4])
>>> h.hit([5, 6, 7])
>>>
>>> h.get()
[1, 5]
>>>
>>> h.block([1, 5])
>>>
>>> h.get()
[2, 4, 5]
>>>
>>> h.delete()
```

Enumerating cardinality-minimal hitting sets can be done as follows:

```
>>> from pysat.examples.hitman import Hitman
>>>
>>> sets = [[1, 2, 3], [1, 4], [5, 6, 7]]
>>> with Hitman(bootstrap_with=sets, htype='sorted') as hitman:
...     for hs in hitman.enumerate():
...         print(hs)
...
[1, 5]
[1, 6]
[1, 7]
[3, 4, 7]
[2, 4, 7]
[3, 4, 6]
[3, 4, 5]
[2, 4, 6]
[2, 4, 5]
```

Finally, implicit hitting set enumeration can be used in practical problem solving. As an example, let us show the basic flow of a MaxHS-like[7] algorithm for MaxSAT:

```
>>> from pysat.examples.hitman import Hitman
>>> from pysat.solvers import Solver
>>>
>>> hitman = Hitman(htype='sorted')
>>> oracle = Solver()
>>>
>>> # here we assume that the SAT oracle
>>> # is initialized with a MaxSAT formula,
>>> # whose soft clauses are extended with
>>> # selector literals stored in "sels"
>>> while True:
...     hs = hitman.get()  # hitting the set of unsatisfiable cores
...     ts = set(sels).difference(set(hs))  # soft clauses to try
```

(continues on next page)

[7] Jessica Davies, Fahiem Bacchus. *Solving MAXSAT by Solving a Sequence of Simpler SAT Instances*. CP 2011. pp. 225-239

(continued from previous page)

```
...
...     if oracle.solve(assumptions=ts):
...         print('s OPTIMUM FOUND')
...         print('o', len(hs))
...         break
...     else:
...         core = oracle.get_core()
...         hitman.hit(core)
```

Module details

class examples.hitman.**Hitman**(*bootstrap_with=[], solver='g3', htype='sorted'*)

A cardinality-/subset-minimal hitting set enumerator. The enumerator can be set up to use either a MaxSAT solver `RC2` or an MCS enumerator (either *LBX* or *MCSls*). In the former case, the hitting sets enumerated are ordered by size (smallest size hitting sets are computed first), i.e. *sorted*. In the latter case, subset-minimal hitting are enumerated in an arbitrary order, i.e. *unsorted*.

This is handled with the use of parameter `htype`, which is set to be `'sorted'` by default. The MaxSAT-based enumerator can be chosen by setting `htype` to one of the following values: `'maxsat'`, `'mxsat'`, or `'rc2'`. Alternatively, by setting it to `'mcs'` or `'lbx'`, a user can enforce using the *LBX* MCS enumerator. If `htype` is set to `'mcsls'`, the *MCSls* enumerator is used.

In either case, an underlying problem solver can use a SAT oracle specified as an input parameter `solver`. The default SAT solver is Glucose3 (specified as `g3`, see *SolverNames* for details).

Objects of class *Hitman* can be bootstrapped with an iterable of iterables, e.g. a list of lists. This is handled using the `bootstrap_with` parameter. Each set to hit can comprise elements of any type, e.g. integers, strings or objects of any Python class, as well as their combinations. The bootstrapping phase is done in *init()*.

Parameters

- **bootstrap_with** (*iterable(iterable(obj))*) – input set of sets to hit
- **solver** (*str*) – name of SAT solver
- **htype** (*str*) – enumerator type

block(*to_block*)

The method serves for imposing a constraint forbidding the hitting set solver to compute a given hitting set. Each set to block is encoded as a hard clause in the MaxSAT problem formulation, which is then added to the underlying oracle.

Parameters **to_block** (*iterable(obj)*) – a set to block

delete()

Explicit destructor of the internal hitting set oracle.

enumerate()

The method can be used as a simple iterator computing and blocking the hitting sets on the fly. It essentially calls *get()* followed by *block()*. Each hitting set is reported as a list of objects in the original problem domain, i.e. it is mapped back from the solutions over Boolean variables computed by the underlying oracle.

Return type list(obj)

get()

This method computes and returns a hitting set. The hitting set is obtained using the underlying oracle operating the MaxSAT problem formulation. The computed solution is mapped back to objects of the problem domain.

Return type list(obj)

hit (*to_hit*)
This method adds a new set to hit to the hitting set solver. This is done by translating the input iterable of objects into a list of Boolean variables in the MaxSAT problem formulation.

Parameters **to_hit** (*iterable(obj)*) – a new set to hit

init (*bootstrap_with*)
This method serves for initializing the hitting set solver with a given list of sets to hit. Concretely, the hitting set problem is encoded into partial MaxSAT as outlined above, which is then fed either to a MaxSAT solver or an MCS enumerator.

Parameters **bootstrap_with** (*iterable(iterable(obj))*) – input set of sets to hit

1.2.4 LBX-like MCS enumerator (`pysat.examples.lbx`)

List of classes

LBX	LBX-like algorithm for computing MCSes.

Module description

This module implements a prototype of the LBX algorithm for the computation of a *minimal correction subset* (MCS) and/or MCS enumeration. The LBX abbreviation stands for *literal-based MCS extraction* algorithm, which was proposed in[1]. Note that this prototype does not follow the original low-level implementation of the corresponding MCS extractor available online (compared to our prototype, the low-level implementation has a number of additional heuristics used). However, it implements the LBX algorithm for partial MaxSAT formulas, as described in[1].

The implementation can be used as an executable (the list of available command-line options can be shown using `lbx.py -h`) in the following way:

```
$ xzcat formula.wcnf.xz
p wcnf 3 6 4
1 1 0
1 2 0
1 3 0
4 -1 -2 0
4 -1 -3 0
4 -2 -3 0

$ lbx.py -d -e all -s glucose3 -vv formula.wcnf.xz
c MCS: 1 3 0
c cost: 2
c MCS: 2 3 0
c cost: 2
c MCS: 1 2 0
c cost: 2
c oracle time: 0.0002
```

Alternatively, the algorithm can be accessed and invoked through the standard `import` interface of Python, e.g.

[1] Carlos Mencia, Alessandro Previti, Joao Marques-Silva. *Literal-Based MCS Extraction*. IJCAI 2015. pp. 1973-1979

```
>>> from pysat.examples.lbx import LBX
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF(from_file='formula.wcnf.xz')
>>>
>>> lbx = LBX(wcnf, use_cld=True, solver_name='g3')
>>> for mcs in lbx.enumerate():
...     lbx.block(mcs)
...     print(mcs)
[1, 3]
[2, 3]
[1, 2]
```

Module details

class `examples.lbx.`**`LBX`**(*formula*, *use_cld=False*, *solver_name='m22'*, *use_timer=False*)

LBX-like algorithm for computing MCSes. Given an unsatisfiable partial CNF formula, i.e. formula in the `WCNF` format, this class can be used to compute a given number of MCSes of the formula. The implementation follows the LBX algorithm description in[1]. It can use any SAT solver available in PySAT. Additionally, the "clause D" heuristic can be used when enumerating MCSes.

The default SAT solver to use is `m22` (see *`SolverNames`*). The "clause D" heuristic is disabled by default, i.e. `use_cld` is set to `False`. Internal SAT solver's timer is also disabled by default, i.e. `use_timer` is `False`.

Parameters

- **`formula`** (`WCNF`) – unsatisfiable partial CNF formula
- **`use_cld`** (*`bool`*) – whether or not to use "clause D"
- **`solver_name`** (*`str`*) – SAT oracle name
- **`use_timer`** (*`bool`*) – whether or not to use SAT solver's timer

`_compute`()

The main method of the class, which computes an MCS given its over-approximation. The over-approximation is defined by a model for the hard part of the formula obtained in *`compute()`*.

The method is essentially a simple loop going over all literals unsatisfied by the previous model, i.e. the literals of `self.setd` and checking which literals can be satisfied. This process can be seen a refinement of the over-approximation of the MCS. The algorithm follows the pseudo-code of the LBX algorithm presented in[1].

Additionally, if *`LBX`* was constructed with the requirement to make "clause D" calls, the method calls *`do_cld_check()`* at every iteration of the loop using the literals of `self.setd` not yet checked, as the contents of "clause D".

`_filter_satisfied`(*update_setd=False*)

This method extracts a model provided by the previous call to a SAT oracle and iterates over all soft clauses checking if each of is satisfied by the model. Satisfied clauses are marked accordingly while the literals of the unsatisfied clauses are kept in a list called `setd`, which is then used to refine the correction set (see *`_compute()`*, and *`do_cld_check()`*).

Optional Boolean parameter `update_setd` enforces the method to update variable `self.setd`. If this parameter is set to `False`, the method only updates the list of satisfied clauses, which is an under-approximation of a *maximal satisfiable subset* (MSS).

Parameters **`update_setd`** (*`bool`*) – whether or not to update setd

_map_extlit(*l*)
Map an external variable to an internal one if necessary.

This method is used when new clauses are added to the formula incrementally, which may result in introducing new variables clashing with the previously used *clause selectors*. The method makes sure no clash occurs, i.e. it maps the original variables used in the new problem clauses to the newly introduced auxiliary variables (see `add_clause()`).

Given an integer literal, a fresh literal is returned. The returned integer has the same sign as the input literal.

Parameters **l** (*int*) – literal to map

Return type int

_satisfied(*cl*, *model*)
Given a clause (as an iterable of integers) and an assignment (as a list of integers), this method checks whether or not the assignment satisfies the clause. This is done by a simple clause traversal. The method is invoked from `_filter_satisfied()`.

Parameters

- **cl** (*iterable(int)*) – a clause to check
- **model** (*list(int)*) – an assignment

Return type bool

add_clause(*clause*, *soft=False*)
The method for adding a new hard of soft clause to the problem formula. Although the input formula is to be specified as an argument of the constructor of `LBX`, adding clauses may be helpful when *enumerating* MCSes of the formula. This way, the clauses are added incrementally, i.e. *on the fly*.

The clause to add can be any iterable over integer literals. The additional Boolean parameter `soft` can be set to `True` meaning the the clause being added is soft (note that parameter `soft` is set to `False` by default).

Parameters

- **clause** (*iterable(int)*) – a clause to add
- **soft** (*bool*) – whether or not the clause is soft

block(*mcs*)
Block a (previously computed) MCS. The MCS should be given as an iterable of integers. Note that this method is not automatically invoked from `enumerate()` because a user may want to block some of the MCSes conditionally depending on the needs. For example, one may want to compute disjoint MCSes only in which case this standard blocking is not appropriate.

Parameters **mcs** (*iterable(int)*) – an MCS to block

compute()
Compute and return one solution. This method checks whether the hard part of the formula is satisfiable, i.e. an MCS can be extracted. If the formula is satisfiable, the model computed by the SAT call is used as an *over-approximation* of the MCS in the method `_compute()` invoked here, which implements the LBX algorithm.

An MCS is reported as a list of integers, each representing a soft clause index (the smallest index is `1`).

Return type list(int)

delete()
Explicit destructor of the internal SAT oracle.

do_cld_check(*cld*)
Do the "clause D" check. This method receives a list of literals, which serves a "clause D"[2], and checks whether the formula conjoined with D is satisfiable.

If clause D cannot be satisfied together with the formula, then negations of all of its literals are backbones of the formula and the LBX algorithm can stop. Otherwise, the literals satisfied by the new model refine the MCS further.

Every time the method is called, a new fresh selector variable s is introduced, which augments the current clause D. The SAT oracle then checks if clause $(D \vee \neg s)$ can be satisfied together with the internal formula. The D clause is then disabled by adding a hard clause $(\neg s)$.

Parameters **cld** (*list(int)*) – clause D to check

enumerate()
This method iterates through MCSes enumerating them until the formula has no more MCSes. The method iteratively invokes `compute()`. Note that the method does not block the MCSes computed - this should be explicitly done by a user.

oracle_time()
Report the total SAT solving time.

1.2.5 LSU algorithm for MaxSAT (`pysat.examples.lsu`)

List of classes

`LSU`	Linear SAT-UNSAT algorithm for MaxSAT[1].
`LSUPlus`	LSU-like algorithm extended for `WCNFPlus` formulas (using `Minicard`).

Module description

The module implements a prototype of the known *LSU/LSUS*, e.g. *linear (search) SAT-UNSAT*, algorithm for MaxSAT, e.g. see[1]. The implementation is improved by the use of the *iterative totalizer encoding*[2]. The encoding is used in an incremental fashion, i.e. it is created once and reused as many times as the number of iterations the algorithm makes.

The implementation can be used as an executable (the list of available command-line options can be shown using `lsu.py -h`) in the following way:

```
$ xzcat formula.wcnf.xz
p wcnf 3 6 4
1 1 0
1 2 0
1 3 0
4 -1 -2 0
4 -1 -3 0
4 -2 -3 0

$ lsu.py -s glucose3 -m -v formula.wcnf.xz
```

(continues on next page)

[2] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, Anton Belov. *On Computing Minimal Correction Subsets*. IJCAI 2013. pp. 615-622

[1] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, Joao Marques-Silva. *Iterative and core-guided MaxSAT solving: A survey and assessment*. Constraints 18(4). 2013. pp. 478-534

[2] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, Inês Lynce. *Incremental Cardinality Constraints for MaxSAT*. CP 2014. pp. 531-548

(continued from previous page)

```
c formula: 3 vars, 3 hard, 3 soft
o 2
s OPTIMUM FOUND
v -1 -2 3 0
c oracle time: 0.0000
```

Alternatively, the algorithm can be accessed and invoked through the standard `import` interface of Python, e.g.

```
>>> from pysat.examples.lsu import LSU
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF(from_file='formula.wcnf.xz')
>>>
>>> lsu = LSU(wcnf, verbose=0)
>>> lsu.solve()  # set of hard clauses should be satisfiable
True
>>> print(lsu.cost) # cost of MaxSAT solution should be 2
>>> 2
>>> print(lsu.model)
[-1, -2, 3]
```

Module details

1.2.6 CLD-like MCS enumerator (`pysat.examples.mcsls`)

List of classes

MCSls	Algorithm BLS for computing MCSes, augmented with "clause *D*" calls.

Module description

This module implements a prototype of a BLS- and CLD-like algorithm for the computation of a *minimal correction subset* (MCS) and/or MCS enumeration. More concretely, the implementation follows the *basic linear search* (BLS) for MCS exctraction augmented with *clause D* (CLD) oracle calls. As a result, the algorithm is not an implementation of the BLS or CLD algorithms as described in[1] but a mixture of both. Note that the corresponding original low-level implementations of both can be found online.

The implementation can be used as an executable (the list of available command-line options can be shown using `mcsls.py -h`) in the following way:

```
$ xzcat formula.wcnf.xz
p wcnf 3 6 4
1 1 0
1 2 0
1 3 0
4 -1 -2 0
4 -1 -3 0
4 -2 -3 0
```

(continues on next page)

[1] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, Anton Belov. *On Computing Minimal Correction Subsets*. IJCAI 2013. pp. 615-622

(continued from previous page)

```
$ mcsls.py -d -e all -s glucose3 -vv formula.wcnf.xz
c MCS: 1 3 0
c cost: 2
c MCS: 2 3 0
c cost: 2
c MCS: 1 2 0
c cost: 2
c oracle time: 0.0002
```

Alternatively, the algorithm can be accessed and invoked through the standard `import` interface of Python, e.g.

```
>>> from pysat.examples.mcsls import MCSls
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF(from_file='formula.wcnf.xz')
>>>
>>> mcsls = MCSls(wcnf, use_cld=True, solver_name='g3')
>>> for mcs in mcsls.enumerate():
...     mcsls.block(mcs)
...     print(mcs)
[1, 3]
[2, 3]
[1, 2]
```

Module details

class `examples.mcsls.`**`MCSls`**(*formula*, *use_cld=False*, *solver_name='m22'*, *use_timer=False*)

Algorithm BLS for computing MCSes, augmented with "clause D" calls. Given an unsatisfiable partial CNF formula, i.e. formula in the `WCNF` format, this class can be used to compute a given number of MCSes of the formula. The implementation follows the description of the basic linear search (BLS) algorithm description in[1]. It can use any SAT solver available in PySAT. Additionally, the "clause D" heuristic can be used when enumerating MCSes.

The default SAT solver to use is `m22` (see *`SolverNames`*). The "clause D" heuristic is disabled by default, i.e. `use_cld` is set to `False`. Internal SAT solver's timer is also disabled by default, i.e. `use_timer` is `False`.

Parameters

- **`formula`** (`WCNF`) – unsatisfiable partial CNF formula
- **`use_cld`** (*bool*) – whether or not to use "clause D"
- **`solver_name`** (*str*) – SAT oracle name
- **`use_timer`** (*bool*) – whether or not to use SAT solver's timer

`_compute`()

The main method of the class, which computes an MCS given its over-approximation. The over-approximation is defined by a model for the hard part of the formula obtained in *`_overapprox()`* (the corresponding oracle is made in *`compute()`*).

The method is essentially a simple loop going over all literals unsatisfied by the previous model, i.e. the literals of `self.setd` and checking which literals can be satisfied. This process can be seen a refinement of the over-approximation of the MCS. The algorithm follows the pseudo-code of the BLS algorithm presented in[1].

Additionally, if `MCS1s` was constructed with the requirement to make "clause D" calls, the method calls `do_cld_check()` at every iteration of the loop using the literals of `self.setd` not yet checked, as the contents of "clause D".

`_map_extlit`(*l*)

Map an external variable to an internal one if necessary.

This method is used when new clauses are added to the formula incrementally, which may result in introducing new variables clashing with the previously used *clause selectors*. The method makes sure no clash occurs, i.e. it maps the original variables used in the new problem clauses to the newly introduced auxiliary variables (see `add_clause()`).

Given an integer literal, a fresh literal is returned. The returned integer has the same sign as the input literal.

Parameters **`l`** (*`int`*) – literal to map

Return type int

`_overapprox`()

The method extracts a model corresponding to an over-approximation of an MCS, i.e. it is the model of the hard part of the formula (the corresponding oracle call is made in `compute()`).

Here, the set of selectors is divided into two parts: `self.ss_assumps`, which is an under-approximation of an MSS (maximal satisfiable subset) and `self.setd`, which is an over-approximation of the target MCS. Both will be further refined in `_compute()`.

`add_clause`(*clause*, *soft=False*)

The method for adding a new hard of soft clause to the problem formula. Although the input formula is to be specified as an argument of the constructor of `MCS1s`, adding clauses may be helpful when *enumerating* MCSes of the formula. This way, the clauses are added incrementally, i.e. *on the fly*.

The clause to add can be any iterable over integer literals. The additional Boolean parameter `soft` can be set to `True` meaning the the clause being added is soft (note that parameter `soft` is set to `False` by default).

Parameters

- **`clause`** (*`iterable(int)`*) – a clause to add
- **`soft`** (*`bool`*) – whether or not the clause is soft

`block`(*mcs*)

Block a (previously computed) MCS. The MCS should be given as an iterable of integers. Note that this method is not automatically invoked from `enumerate()` because a user may want to block some of the MCSes conditionally depending on the needs. For example, one may want to compute disjoint MCSes only in which case this standard blocking is not appropriate.

Parameters **`mcs`** (*`iterable(int)`*) – an MCS to block

`compute`()

Compute and return one solution. This method checks whether the hard part of the formula is satisfiable, i.e. an MCS can be extracted. If the formula is satisfiable, the model computed by the SAT call is used as an *over-approximation* of the MCS in the method `_compute()` invoked here, which implements the BLS algorithm augmented with CLD oracle calls.

An MCS is reported as a list of integers, each representing a soft clause index (the smallest index is `1`).

Return type list(int)

`delete`()

Explicit destructor of the internal SAT oracle.

do_cld_check (*cld*)

Do the "clause D" check. This method receives a list of literals, which serves a "clause D"[1], and checks whether the formula conjoined with D is satisfiable.

If clause D cannot be satisfied together with the formula, then negations of all of its literals are backbones of the formula and the MCSls algorithm can stop. Otherwise, the literals satisfied by the new model refine the MCS further.

Every time the method is called, a new fresh selector variable s is introduced, which augments the current clause D. The SAT oracle then checks if clause $(D \vee \neg s)$ can be satisfied together with the internal formula. The D clause is then disabled by adding a hard clause $(\neg s)$.

Parameters **cld** (*list (int)*) – clause D to check

enumerate ()

This method iterates through MCSes enumerating them until the formula has no more MCSes. The method iteratively invokes `compute()`. Note that the method does not block the MCSes computed - this should be explicitly done by a user.

oracle_time ()

Report the total SAT solving time.

1.2.7 An iterative model enumerator (`pysat.examples.models`)

List of classes

enumerate_models	Enumeration procedure.

Module description

The module implements a simple iterative enumeration of a given number of models of `CNF` or `CNFPlus` formula. In the latter case, only `Minicard` can be used as a SAT solver. The module aims at illustrating how one can work with model computation and enumeration.

The implementation facilitates the simplest use of a SAT oracle from the *command line*. If one deals with the enumeration task from a Python script, it is more convenient to exploit the internal model enumeration of the `pysat.solvers` module. Concretely, see `pysat.solvers.Solver.enum_models()`.

```
$ cat formula.cnf
p cnf 4 4
-1 2 0
-1 3 0
-2 4 0
3 -4 0

$ models.py -e all -s glucose3 formula.cnf
v -1 -2 +3 -4 0
v +1 +2 -3 +4 0
c nof models: 2
c accum time: 0.00s
c mean  time: 0.00s
```

Module details

`examples.models.`**`enumerate_models`**(*formula*, *to_enum*, *solver*)

Enumeration procedure. It represents a loop iterating over satisfying assignment for a given formula until either all or a given number of them is enumerated.

Parameters

- **formula** (`CNFPlus`) – input WCNF formula
- **to_enum** (*int or 'all'*) – number of models to compute
- **solver** (*str*) – name of SAT solver

1.2.8 A deletion-based MUS extractor (`pysat.examples.musx`)

List of classes

MUSX	MUS eXtractor using the deletion-based algorithm.

Module description

This module implements a deletion-based algorithm[1] for extracting a *minimal unsatisfiable subset* (*MUS*) of a given (unsafistiable) CNF formula. This simplistic implementation can deal with *plain* and *partial* CNF formulas, e.g. formulas in the DIMACS CNF and WCNF formats.

The following extraction procedure is implemented:

```
# oracle: SAT solver (initialized)
# assump: full set of assumptions

i = 0

while i < len(assump):
    to_test = assump[:i] + assump[(i + 1):]
    if oracle.solve(assumptions=to_test):
        i += 1
    else:
        assump = to_test

return assump
```

The implementation can be used as an executable (the list of available command-line options can be shown using `musx.py -h`) in the following way:

```
$ cat formula.wcnf
p wcnf 3 6 4
1 1 0
1 2 0
1 3 0
4 -1 -2 0
4 -1 -3 0
4 -2 -3 0
```

(continues on next page)

[1] Joao Marques-Silva. *Minimal Unsatisfiability: Models, Algorithms and Applications*. ISMVL 2010. pp. 9-14

(continued from previous page)

```
$ musx.py -s glucose3 -vv formula.wcnf
c MUS approx: 1 2 0
c testing clid: 0 -> sat (keeping 0)
c testing clid: 1 -> sat (keeping 1)
c nof soft: 3
c MUS size: 2
v 1 2 0
c oracle time: 0.0001
```

Alternatively, the algorithm can be accessed and invoked through the standard `import` interface of Python, e.g.

```
>>> from pysat.examples.musx import MUSX
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF(from_file='formula.wcnf')
>>>
>>> musx = MUSX(wcnf, verbosity=0)
>>> musx.compute()  # compute a minimally unsatisfiable set of clauses
[1, 2]
```

Note that the implementation is able to compute only one MUS (MUS enumeration is not supported).

Module details

class `examples.musx.`**MUSX**(*formula*, *solver='m22'*, *verbosity=1*)

MUS eXtractor using the deletion-based algorithm. The algorithm is described in[1] (also see the module description above). Essentially, the algorithm can be seen as an iterative process, which tries to remove one soft clause at a time and check whether the remaining set of soft clauses is still unsatisfiable together with the hard clauses.

The constructor of *MUSX* objects receives a target `WCNF` formula, a SAT solver name, and a verbosity level. Note that the default SAT solver is MiniSat22 (referred to as `'m22'`, see *SolverNames* for details). The default verbosity level is `1`.

Parameters

- **formula** (`WCNF`) – input WCNF formula
- **solver** (*str*) – name of SAT solver
- **verbosity** (*int*) – verbosity level

_compute(*approx*)

Deletion-based MUS extraction. Given an over-approximation of an MUS, i.e. an unsatisfiable core previously returned by a SAT oracle, the method represents a loop, which at each iteration removes a clause from the core and checks whether the remaining clauses of the approximation are unsatisfiable together with the hard clauses.

Soft clauses are (de)activated using the standard MiniSat-like assumptions interface[2]. Each soft clause c is augmented with a selector literal s, e.g. $(c) \leftarrow (c \vee \neg s)$. As a result, clause c can be activated by assuming literal s. The over-approximation provided as an input is specified as a list of selector literals for clauses in the unsatisfiable core.

Parameters **approx** (*list(int)*) – an over-approximation of an MUS

Note that the method does not return. Instead, after its execution, the input over-approximation is refined and contains an MUS.

[2] Niklas Eén, Niklas Sörensson. *Temporal induction by incremental SAT solving*. Electr. Notes Theor. Comput. Sci. 89(4). 2003. pp. 543-560

compute()
This is the main method of the *MUSX* class. It computes a set of soft clauses belonging to an MUS of the input formula. First, the method checks whether the formula is satisfiable. If it is, nothing else is done. Otherwise, an *unsatisfiable core* of the formula is extracted, which is later used as an over-approximation of an MUS refined in *_compute()*.

delete()
Explicit destructor of the internal SAT oracle.

oracle_time()
Method for calculating and reporting the total SAT solving time.

1.2.9 RC2 MaxSAT solver (`pysat.examples.rc2`)

List of classes

`RC2`	Implementation of the basic RC2 algorithm.
`RC2Stratified`	RC2 augmented with BLO and stratification techniques.

Module description

An implementation of the RC2 algorithm for solving maximum satisfiability. RC2 stands for *relaxable cardinality constraints* (alternatively, *soft cardinality constraints*) and represents an improved version of the OLLITI algorithm, which was described in[1] and[2] and originally implemented in the MSCG MaxSAT solver.

Initially, this solver was supposed to serve as an example of a possible PySAT usage illustrating how a state-of-the-art MaxSAT algorithm could be implemented in Python and still be efficient. It participated in the MaxSAT Evaluations 2018 and 2019 where, surprisingly, it was ranked first in two complete categories: *unweighted* and *weighted*. A brief solver description can be found in[3]. A more detailed solver description can be found in[4].

The file implements two classes: `RC2` and `RC2Stratified`. The former class is the basic implementation of the algorithm, which can be applied to a MaxSAT formula in the `WCNF` format. The latter class additionally implements Boolean lexicographic optimization (BLO)[5] and stratification[6] on top of `RC2`.

The implementation can be used as an executable (the list of available command-line options can be shown using `rc2.py -h`) in the following way:

```
$ xzcat formula.wcnf.xz
p wcnf 3 6 4
1 1 0
1 2 0
1 3 0
4 -1 -2 0
4 -1 -3 0
4 -2 -3 0

$ rc2.py -vv formula.wcnf.xz
```

(continues on next page)

[1] António Morgado, Carmine Dodaro, Joao Marques-Silva. *Core-Guided MaxSAT with Soft Cardinality Constraints*. CP 2014. pp. 564-573
[2] António Morgado, Alexey Ignatiev, Joao Marques-Silva. *MSCG: Robust Core-Guided MaxSAT Solving*. JSAT 9. 2014. pp. 129-134
[3] Alexey Ignatiev, António Morgado, Joao Marques-Silva. *RC2: A Python-based MaxSAT Solver*. MaxSAT Evaluation 2018. p. 22
[4] Alexey Ignatiev, António Morgado, Joao Marques-Silva. *RC2: An Efficient MaxSAT Solver*. MaxSAT Evaluation 2018. JSAT 11. 2019. pp. 53-64
[5] Joao Marques-Silva, Josep Argelich, Ana Graça, Inês Lynce. *Boolean lexicographic optimization: algorithms & applications*. Ann. Math. Artif. Intell. 62(3-4). 2011. pp. 317-343
[6] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, Jordi Levy. *Improving WPM2 for (Weighted) Partial MaxSAT*. CP 2013. pp. 117-132

(continued from previous page)

```
c formula: 3 vars, 3 hard, 3 soft
c cost: 1; core sz: 2; soft sz: 2
c cost: 2; core sz: 2; soft sz: 1
s OPTIMUM FOUND
o 2
v -1 -2 3
c oracle time: 0.0001
```

Alternatively, the algorithm can be accessed and invoked through the standard `import` interface of Python, e.g.

```
>>> from pysat.examples.rc2 import RC2
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF(from_file='formula.wcnf.xz')
>>>
>>> with RC2(wcnf) as rc2:
...     for m in rc2.enumerate():
...         print('model {0} has cost {1}'.format(m, rc2.cost))
model [-1, -2, 3] has cost 2
model [1, -2, -3] has cost 2
model [-1, 2, -3] has cost 2
model [-1, -2, -3] has cost 3
```

As can be seen in the example above, the solver can be instructed either to compute one MaxSAT solution of an input formula, or to enumerate a given number (or *all*) of its top MaxSAT solutions.

Module details

PYTHON MODULE INDEX

e

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W