PySAT Documentation

Release 0.1.4.dev9

Alexey Ignatiev, Joao Marques-Silva, Antonio Morgado

May 17, 2019

CONTENTS

1 API documentation **3**

1.1 Core PySAT modules . . . 3

1.1.1 Cardinality encodings (`pysat.card`) . . . 3

1.1.2 Boolean formula manipulation (`pysat.formula`) . . . 8

1.1.3 Pseudo-Boolean encodings (`pysat.pb`) . . . 23

1.1.4 SAT solvers' API (`pysat.solvers`) . . . 25

1.2 Supplementary `examples` package . . . 35

1.2.1 Fu&Malik MaxSAT algorithm (`pysat.examples.fm`) . . . 35

1.2.2 Hard formula generator (`pysat.examples.genhard`) . . . 37

1.2.3 Minimum/minimal hitting set solver (`pysat.examples.hitman`) . . . 40

1.2.4 LBX-like MCS enumerator (`pysat.examples.lbx`) . . . 43

1.2.5 LSU algorithm for MaxSAT (`pysat.examples.lsu`) . . . 46

1.2.6 CLD-like MCS enumerator (`pysat.examples.mcsls`) . . . 49

1.2.7 A deletion-based MUS extractor (`pysat.examples.musx`) . . . 52

1.2.8 RC2 MaxSAT solver (`pysat.examples.rc2`) . . . 54

Python Module Index **63**

Index **65**

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 representation supported by the MiniCard solver.

[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

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*, *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`, 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.
- **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`.

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*, *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*, *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]]
```

(continues on next page)

(continued from previous page)

```
>>> 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]]
>>> 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
```

(continues on next page)

(continued from previous page)

```
[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')
>>> 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
```

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):
```

(continues on next page)

(continued from previous page)

```
    for comb in itertools.combinations(pigeons, 2):
        cnf.append([-pigeon(i, j) for i in comb])
```

Module details

class pysat.formula.**CNF**(*from_file=None, from_fp=None, from_string=None, from_clauses=[], comment_lead=['c']*)

Class for manipulating CNF formulas. It can be used for creating formulas, reading them from a file, or writing them to a file. The `comment_lead` parameter can be helpful when one needs to parse specific comment lines starting not with character `c` but with another character or a string.

Parameters

- **from_file** (*str*) – a DIMACS CNF filename to read from
- **from_fp** (*file_pointer*) – a file pointer to read from
- **from_string** (*str*) – a string storing a CNF formula
- **from_clauses** (*list(list(int))*) – a list of clauses to bootstrap the formula with
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

append(*clause*)

Add one more clause to CNF formula. This method additionally updates the number of variables, i.e. variable `self.nv`, used in the formula.

Parameters **clause** (*list(int)*) – a new clause to add.

```
>>> from pysat.formula import CNF
>>> cnf = CNF(from_clauses=[[-1, 2], [3]])
>>> cnf.append([-3, 4])
>>> print cnf.clauses
[[-1, 2], [3], [-3, 4]]
```

copy()

This method can be used for creating a copy of a CNF object. It creates another object of the *CNF* class and makes use of the *deepcopy* functionality to copy the clauses.

Returns an object of class *CNF*.

Example:

```
>>> cnf1 = CNF(from_clauses=[[-1, 2], [1]])
>>> cnf2 = cnf1.copy()
>>> print cnf2.clauses
[[-1, 2], [1]]
>>> print cnf2.nv
2
```

extend(*clauses*)

Add several clauses to CNF formula. The clauses should be given in the form of list. For every clause in the list, method *append()* is invoked.

Parameters **clauses** (*list(list(int))*) – a list of new clauses to add.

Example:

```
>>> from pysat.formula import CNF
>>> cnf = CNF(from_clauses=[[-1, 2], [3]])
>>> cnf.extend([[-3, 4], [5, 6]])
>>> print cnf.clauses
[[-1, 2], [3], [-3, 4], [5, 6]]
```

from_clauses (*clauses*)

This methods copies a list of clauses into a CNF object.

Parameters **clauses** (*list(list(int))*) – a list of clauses.

Example:

```
>>> from pysat.formula import CNF
>>> cnf = CNF(from_clauses=[[-1, 2], [1, -2], [5]])
>>> print cnf.clauses
[[-1, 2], [1, -2], [5]]
>>> print cnf.nv
5
```

from_file (*fname, comment_lead=['c'], compressed_with='use_ext'*)

Read a CNF formula from a file in the DIMACS format. A file name is expected as an argument. A default argument is `comment_lead` for parsing comment lines. A given file can be compressed by either gzip, bzip2, or lzma.

Parameters

- **fname** (*str*) – name of a file to parse.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines
- **compressed_with** (*str*) – file compression algorithm

Note that the `compressed_with` parameter can be `None` (i.e. the file is uncompressed), `'gzip'`, `'bzip2'`, `'lzma'`, or `'use_ext'`. The latter value indicates that compression type should be automatically determined based on the file extension. Using `'lzma'` in Python 2 requires the `backports.lzma` package to be additionally installed.

Usage example:

```
>>> from pysat.formula import CNF
>>> cnf1 = CNF()
>>> cnf1.from_file('some-file.cnf.gz', compressed_with='gzip')
>>>
>>> cnf2 = CNF(from_file='another-file.cnf')
```

from_fp (*file_pointer, comment_lead=['c']*)

Read a CNF formula from a file pointer. A file pointer should be specified as an argument. The only default argument is `comment_lead`, which can be used for parsing specific comment lines.

Parameters

- **file_pointer** (*file pointer*) – a file pointer to read the formula from.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

Usage example:

```
>>> with open('some-file.cnf', 'r') as fp:
...     cnf1 = CNF()
...     cnf1.from_fp(fp)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> with open('another-file.cnf', 'r') as fp:
...     cnf2 = CNF(from_fp=fp)
```

from_string (*string, comment_lead=['c']*)

Read a CNF formula from a string. The string should be specified as an argument and should be in the DIMACS CNF format. The only default argument is `comment_lead`, which can be used for parsing specific comment lines.

Parameters

- **string** (*str*) – a string containing the formula in DIMACS.
- **comment_lead** (*list (str)*) – a list of characters leading comment lines

Example:

```
>>> from pysat.formula import CNF
>>> cnf1 = CNF()
>>> cnf1.from_string('p cnf 2 2\n-1 2 0\n1 -2 0')
>>> print cnf1.clauses
[[-1, 2], [1, -2]]
>>>
>>> cnf2 = CNF(from_string='p cnf 3 3\n-1 2 0\n-2 3 0\n-3 0\n')
>>> print cnf2.clauses
[[-1, 2], [-2, 3], [-3]]
>>> print cnf2.nv
3
```

negate (*topv=None*)

Given a CNF formula $\mathcal{F}$, this method creates a CNF formula $\neg\mathcal{F}$. The negation of the formula is encoded to CNF with the use of *auxiliary* Tseitin variables[1]. A new CNF formula is returned keeping all the newly introduced variables that can be accessed through the `auxvars` variable.

Note that the negation of each clause is encoded with one auxiliary variable if it is not unit size. Otherwise, no auxiliary variable is introduced.

Parameters **topv** (*int*) – top variable identifier if any.

Returns an object of class *CNF*.

```
>>> from pysat.formula import CNF
>>> pos = CNF(from_clauses=[[-1, 2], [3]])
>>> neg = pos.negate()
>>> print neg.clauses
[[1, -4], [-2, -4], [-1, 2, 4], [4, -3]]
>>> print neg.auxvars
[4, -3]
```

to_file (*fname, comments=None, compress_with='use_ext'*)

The method is for saving a CNF formula into a file in the DIMACS CNF format. A file name is expected as an argument. Additionally, supplementary comment lines can be specified in the `comments` parameter. Also, a file can be compressed using either gzip, bzip2, or lzma (xz).

Parameters

- **fname** (*str*) – a file name where to store the formula.

[1] G. S. Tseitin. *On the complexity of derivations in the propositional calculus*. Studies in Mathematics and Mathematical Logic, Part II. pp. 115–125, 1968

- **comments** (*list(str)*) – additional comments to put in the file.
- **compress_with** (*str*) – file compression algorithm

Note that the `compress_with` parameter can be `None` (i.e. the file is uncompressed), `'gzip'`, `'bzip2'`, `'lzma'`, or `'use_ext'`. The latter value indicates that compression type should be automatically determined based on the file extension. Using `'lzma'` in Python 2 requires the `backports.lzma` package to be additionally installed.

Example:

```
>>> from pysat.formula import CNF
>>> cnf = CNF()
...
>>> # the formula is filled with a bunch of clauses
>>> cnf.to_file('some-file-name.cnf')  # writing to a file
```

to_fp(*file_pointer*, *comments=None*)

The method can be used to save a CNF formula into a file pointer. The file pointer is expected as an argument. Additionally, supplementary comment lines can be specified in the `comments` parameter.

Parameters

- **fname** (*str*) – a file name where to store the formula.
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

```
>>> from pysat.formula import CNF
>>> cnf = CNF()
...
>>> # the formula is filled with a bunch of clauses
>>> with open('some-file.cnf', 'w') as fp:
...     cnf.to_fp(fp)  # writing to the file pointer
```

weighted()

This method creates a weighted copy of the internal formula. As a result, an object of class *WCNF* is returned. Every clause of the CNF formula is *soft* in the new WCNF formula and its weight is equal to `1`. The set of hard clauses of the formula is empty.

Returns an object of class *WCNF*.

Example:

```
>>> from pysat.formula import CNF
>>> cnf = CNF(from_clauses=[[-1, 2], [3, 4]])
>>>
>>> wcnf = cnf.weighted()
>>> print wcnf.hard
[]
>>> print wcnf.soft
[[-1, 2], [3, 4]]
>>> print wcnf.wght
[1, 1]
```

class pysat.formula.**CNFPlus**(*from_file=None*, *from_fp=None*, *from_string=None*, *comment_lead=['c']*)

CNF formulas augmented with *native* cardinality constraints.

This class inherits most of the functionality of the *CNF* class. The only difference between the two is that *CNFPlus* supports *native* cardinality constraints of MiniCard.

The parser of input DIMACS files of *CNFPlus* assumes the syntax of AtMostK and AtLeastK constraints defined in the description of MiniCard:

```
c Example: Two cardinality constraints followed by a clause
p cnf+ 7 3
1 -2 3 5 -7 <= 3
4 5 6 -7 >= 2
3 5 7 0
```

Each AtLeastK constraint is translated into an AtMostK constraint in the standard way: $\sum_{i=1}^{n} x_i \geq k \leftrightarrow \sum_{i=1}^{n} \neg x_i \leq (n-k)$. Internally, AtMostK constraints are stored in variable `atmosts`, each being a pair `(lits, k)`, where `lits` is a list of literals in the sum and `k` is the upper bound.

Example:

```
>>> from pysat.formula import CNFPlus
>>> cnf = CNFPlus(from_string='p cnf+ 7 3\n1 -2 3 5 -7 <= 3\n4 5 6 -7 >= 2\n 3 5␣
↪7 0\n')
>>> print cnf.clauses
[[3, 5, 7]]
>>> print cnf.atmosts
[[[1, -2, 3, 5, -7], 3], [[-4, -5, -6, 7], 2]]
>>> print cnf.nv
7
```

For details on the functionality, see *CNF*.

append(*clause*, *is_atmost=False*)

Add a single clause or a single AtMostK constraint to CNF+ formula. This method additionally updates the number of variables, i.e. variable `self.nv`, used in the formula.

If the clause is an AtMostK constraint, this should be set with the use of the additional default argument `is_atmost`, which is set to `False` by default.

Parameters

- **clause** (*list(int)*) – a new clause to add.
- **is_atmost** (*bool*) – if `True`, the clause is AtMostK.

```
>>> from pysat.formula import CNFPlus
>>> cnf = CNFPlus()
>>> cnf.append([-3, 4])
>>> cnf.append([[1, 2, 3], 1], is_atmost=True)
>>> print cnf.clauses
[[-3, 4]]
>>> print cnf.atmosts
[[1, 2, 3], 1]
```

from_fp(*file_pointer, comment_lead=['c']*)

Read a CNF+ formula from a file pointer. A file pointer should be specified as an argument. The only default argument is `comment_lead`, which can be used for parsing specific comment lines.

Parameters

- **file_pointer** (*file pointer*) – a file pointer to read the formula from.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

Usage example:

```
>>> with open('some-file.cnf+', 'r') as fp:
...     cnf1 = CNFPlus()
...     cnf1.from_fp(fp)
>>>
>>> with open('another-file.cnf+', 'r') as fp:
...     cnf2 = CNFPlus(from_fp=fp)
```

to_fp(*file_pointer*, *comments=None*)

The method can be used to save a CNF+ formula into a file pointer. The file pointer is expected as an argument. Additionally, supplementary comment lines can be specified in the `comments` parameter.

Parameters

- **fname** (*str*) – a file name where to store the formula.
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

```
>>> from pysat.formula import CNFPlus
>>> cnf = CNFPlus()
...
>>> # the formula is filled with a bunch of clauses
>>> with open('some-file.cnf+', 'w') as fp:
...     cnf.to_fp(fp)  # writing to the file pointer
```

class pysat.formula.**IDPool**(*start_from=1*, *occupied=[]*)

A simple manager of variable IDs. It can be used as a pool of integers assigning an ID to any object. Identifiers are to start from `1` by default. The list of occupied intervals is empty be default. If necessary the top variable ID can be accessed directly using the `top` variable.

Parameters

- **start_from** (*int*) – the smallest ID to assign.
- **occupied** (*list(list(int))*) – a list of occupied intervals.

id(*obj*)

The method is to be used to assign an integer variable ID for a given new object. If the object already has an ID, no new ID is created and the old one is returned instead.

An object can be anything. In some cases it is convenient to use string variable names.

Parameters **obj** – an object to assign an ID to.

Return type int.

Example:

```
>>> from pysat.formula import IDPool
>>> vpool = IDPool(occupied=[[12, 18], [3, 10]])
>>>
>>> # creating 5 unique variables for the following strings
>>> for i in range(5):
...     print vpool.id('v{0}'.format(i + 1))
1
2
11
19
20
```

In some cases, it makes sense to create an external function for accessing IDPool, e.g.:

```
>>> # continuing the previous example
>>> var = lambda i: vpool.id('var{0}'.format(i))
>>> var(5)
20
>>> var('hello_world!')
21
```

obj(*vid*)
The method can be used to map back a given variable identifier to the original object labeled by the identifier.

Parameters **vid** (*int*) – variable identifier.

Returns an object corresponding to the given identifier.

Example:

```
>>> vpool.obj(21)
'hello_world!'
```

occupy(*start*, *stop*)
Mark a given interval as occupied so that the manager could skip the values from `start` to `stop` (**inclusive**).

Parameters

- **start** (*int*) – beginning of the interval.
- **stop** (*int*) – end of the interval.

restart(*start_from=1*, *occupied=[]*)
Restart the manager from scratch. The arguments replicate those of the constructor of *IDPool*.

class pysat.formula.**WCNF**(*from_file=None, from_fp=None, from_string=None, comment_lead=['c']*)
Class for manipulating partial (weighted) CNF formulas. It can be used for creating formulas, reading them from a file, or writing them to a file. The `comment_lead` parameter can be helpful when one needs to parse specific comment lines starting not with character `c` but with another character or a string.

Parameters

- **from_file** (*str*) – a DIMACS CNF filename to read from
- **from_fp** (*file_pointer*) – a file pointer to read from
- **from_string** (*str*) – a string storing a CNF formula
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

append(*clause*, *weight=None*)
Add one more clause to WCNF formula. This method additionally updates the number of variables, i.e. variable `self.nv`, used in the formula.

The clause can be hard or soft depending on the `weight` argument. If no weight is set, the clause is considered to be hard.

Parameters

- **clause** (*list(int)*) – a new clause to add.
- **weight** (*integer or None*) – integer weight of the clause.

```
>>> from pysat.formula import WCNF
>>> cnf = WCNF()
>>> cnf.append([-1, 2])
>>> cnf.append([1], weight=10)
>>> cnf.append([-2], weight=20)
>>> print cnf.hard
[[-1, 2]]
>>> print cnf.soft
[[1], [-2]]
>>> print cnf.wght
[10, 20]
```

copy()

This method can be used for creating a copy of a WCNF object. It creates another object of the *WCNF* class and makes use of the *deepcopy* functionality to copy both hard and soft clauses.

Returns an object of class *WCNF*.

Example:

```
>>> cnf1 = WCNF()
>>> cnf1.append([-1, 2])
>>> cnf1.append([1], weight=10)
>>>
>>> cnf2 = cnf1.copy()
>>> print cnf2.hard
[[-1, 2]]
>>> print cnf2.soft
[[1]]
>>> print cnf2.wght
[10]
>>> print cnf2.nv
2
```

extend(*clauses*, *weights=None*)

Add several clauses to WCNF formula. The clauses should be given in the form of list. For every clause in the list, method *append()* is invoked.

The clauses can be hard or soft depending on the `weights` argument. If no weights are set, the clauses are considered to be hard.

Parameters

- **clauses** (*list(list(int))*) – a list of new clauses to add.
- **weights** (*list(int)*) – a list of integer weights.

Example:

```
>>> from pysat.formula import WCNF
>>> cnf = WCNF()
>>> cnf.extend([[-3, 4], [5, 6]])
>>> cnf.extend([[3], [-4], [-5], [-6]], weights=[1, 5, 3, 4])
>>> print cnf.hard
[[-3, 4], [5, 6]]
>>> print cnf.soft
[[3], [-4], [-5], [-6]]
>>> print cnf.wght
[1, 5, 3, 4]
```

from_file (*fname, comment_lead=['c'], compressed_with='use_ext'*)

Read a WCNF formula from a file in the DIMACS format. A file name is expected as an argument. A default argument is `comment_lead` for parsing comment lines. A given file can be compressed by either gzip, bzip2, or lzma.

Parameters

- **fname** (*str*) – name of a file to parse.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines
- **compressed_with** (*str*) – file compression algorithm

Note that the `compressed_with` parameter can be `None` (i.e. the file is uncompressed), `'gzip'`, `'bzip2'`, `'lzma'`, or `'use_ext'`. The latter value indicates that compression type should be automatically determined based on the file extension. Using `'lzma'` in Python 2 requires the `backports.lzma` package to be additionally installed.

Usage example:

```
>>> from pysat.formula import WCNF
>>> cnf1 = WCNF()
>>> cnf1.from_file('some-file.wcnf.bz2', compressed_with='bzip2')
>>>
>>> cnf2 = WCNF(from_file='another-file.wcnf')
```

from_fp (*file_pointer, comment_lead=['c']*)

Read a WCNF formula from a file pointer. A file pointer should be specified as an argument. The only default argument is `comment_lead`, which can be used for parsing specific comment lines.

Parameters

- **file_pointer** (*file pointer*) – a file pointer to read the formula from.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

Usage example:

```
>>> with open('some-file.cnf', 'r') as fp:
...     cnf1 = WCNF()
...     cnf1.from_fp(fp)
>>>
>>> with open('another-file.cnf', 'r') as fp:
...     cnf2 = WCNF(from_fp=fp)
```

from_string (*string, comment_lead=['c']*)

Read a WCNF formula from a string. The string should be specified as an argument and should be in the DIMACS CNF format. The only default argument is `comment_lead`, which can be used for parsing specific comment lines.

Parameters

- **string** (*str*) – a string containing the formula in DIMACS.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

Example:

```
>>> from pysat.formula import WCNF
>>> cnf1 = WCNF()
>>> cnf1.from_string('p wcnf 2 2 2\n 2 -1 2 0\n1 1 -2 0')
>>> print cnf1.hard
```

(continues on next page)

(continued from previous page)

```
[[-1, 2]]
>>> print cnf1.soft
[[1, 2]]
>>>
>>> cnf2 = WCNF(from_string='p wcnf 3 3 2\n2 -1 2 0\n2 -2 3 0\n1 -3 0\n')
>>> print cnf2.hard
[[-1, 2], [-2, 3]]
>>> print cnf2.soft
[[-3]]
>>> print cnf2.nv
3
```

to_file (*fname*, *comments=None*, *compress_with='use_ext'*)

The method is for saving a WCNF formula into a file in the DIMACS CNF format. A file name is expected as an argument. Additionally, supplementary comment lines can be specified in the `comments` parameter. Also, a file can be compressed using either gzip, bzip2, or lzma (xz).

Parameters

- **fname** (*str*) – a file name where to store the formula.
- **comments** (*list(str)*) – additional comments to put in the file.
- **compress_with** (*str*) – file compression algorithm

Note that the `compress_with` parameter can be `None` (i.e. the file is uncompressed), `'gzip'`, `'bzip2'`, `'lzma'`, or `'use_ext'`. The latter value indicates that compression type should be automatically determined based on the file extension. Using `'lzma'` in Python 2 requires the `backports.lzma` package to be additionally installed.

Example:

```
>>> from pysat.formula import WCNF
>>> wcnf = WCNF()
...
>>> # the formula is filled with a bunch of clauses
>>> wcnf.to_file('some-file-name.wcnf')  # writing to a file
```

to_fp (*file_pointer*, *comments=None*)

The method can be used to save a WCNF formula into a file pointer. The file pointer is expected as an argument. Additionally, supplementary comment lines can be specified in the `comments` parameter.

Parameters

- **fname** (*str*) – a file name where to store the formula.
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

```
>>> from pysat.formula import WCNF
>>> wcnf = WCNF()
...
>>> # the formula is filled with a bunch of clauses
>>> with open('some-file.wcnf', 'w') as fp:
...     wcnf.to_fp(fp)  # writing to the file pointer
```

unweighed()

This method creates a *plain* (unweighted) copy of the internal formula. As a result, an object of class *CNF*

is returned. Every clause (both hard or soft) of the WCNF formula is copied to the `clauses` variable of the resulting plain formula, i.e. all weights are discarded.

Returns an object of class *CNF*.

Example:

```
>>> from pysat.formula import WCNF
>>> wcnf = WCNF()
>>> wcnf.extend([[-3, 4], [5, 6]])
>>> wcnf.extend([[3], [-4], [-5], [-6]], weights=[1, 5, 3, 4])
>>>
>>> cnf = wcnf.unweighted()
>>> print cnf.clauses
[[-3, 4], [5, 6], [3], [-4], [-5], [-6]]
```

class pysat.formula.**WCNFPlus**(*from_file=None, from_fp=None, from_string=None, comment_lead=['c']*)

WCNF formulas augmented with *native* cardinality constraints.

This class inherits most of the functionality of the *WCNF* class. The only difference between the two is that *WCNFPlus* supports *native* cardinality constraints of MiniCard.

The parser of input DIMACS files of *WCNFPlus* assumes the syntax of AtMostK and AtLeastK constraints following the one defined for *CNFPlus* in the description of MiniCard:

```
c Example: Two (hard) cardinality constraints followed by a soft clause
p wcnf+ 7 3 10
10 1 -2 3 5 -7 <= 3
10 4 5 6 -7 >= 2
5 3 5 7 0
```

Note that every cardinality constraint is assumed to be hard, i.e. soft cardinality constraints are currently *not supported*.

Each AtLeastK constraint is translated into an AtMostK constraint in the standard way: $\sum_{i=1}^{n} x_i \geq k \leftrightarrow \sum_{i=1}^{n} \neg x_i \leq (n-k)$. Internally, AtMostK constraints are stored in variable `atms`, each being a pair `(lits, k)`, where `lits` is a list of literals in the sum and `k` is the upper bound.

Example:

```
>>> from pysat.formula import WCNFPlus
>>> cnf = WCNFPlus(from_string='p wcnf+ 7 3 10\n10 1 -2 3 5 -7 <= 3\n10 4 5 6 -7 >
↪= 2\n5 3 5 7 0\n')
>>> print cnf.soft
[[3, 5, 7]]
>>> print cnf.wght
[5]
>>> print cnf.hard
[]
>>> print cnf.atms
[[[1, -2, 3, 5, -7], 3], [[-4, -5, -6, 7], 2]]
>>> print cnf.nv
7
```

For details on the functionality, see *WCNF*.

append(*clause*, *weight=None*, *is_atmost=False*)

Add a single clause or a single AtMostK constraint to WCNF+ formula. This method additionally updates the number of variables, i.e. variable `self.nv`, used in the formula.

If the clause is an AtMostK constraint, this should be set with the use of the additional default argument `is_atmost`, which is set to `False` by default.

If `is_atmost` is set to `False`, the clause can be either hard or soft depending on the `weight` argument. If no weight is specified, the clause is considered hard. Otherwise, the clause is soft.

Parameters

- **clause** (*list(int)*) – a new clause to add.
- **weight** (*integer or None*) – an integer weight of the clause.
- **is_atmost** (*bool*) – if `True`, the clause is AtMostK.

```
>>> from pysat.formula import WCNFPlus
>>> cnf = WCNFPlus()
>>> cnf.append([-3, 4])
>>> cnf.append([[1, 2, 3], 1], is_atmost=True)
>>> cnf.append([-1, -2], weight=35)
>>> print cnf.hard
[[-3, 4]]
>>> print cnf.atms
[[1, 2, 3], 1]
>>> print cnf.soft
[[-1, -2]]
>>> print cnf.wght
[35]
```

from_fp (*file_pointer, comment_lead=['c']*)

Read a WCNF+ formula from a file pointer. A file pointer should be specified as an argument. The only default argument is `comment_lead`, which can be used for parsing specific comment lines.

Parameters

- **file_pointer** (*file pointer*) – a file pointer to read the formula from.
- **comment_lead** (*list(str)*) – a list of characters leading comment lines

Usage example:

```
>>> with open('some-file.wcnf+', 'r') as fp:
...     cnf1 = WCNFPlus()
...     cnf1.from_fp(fp)
>>>
>>> with open('another-file.wcnf+', 'r') as fp:
...     cnf2 = WCNFPlus(from_fp=fp)
```

to_fp (*file_pointer, comments=None*)

The method can be used to save a WCNF+ formula into a file pointer. The file pointer is expected as an argument. Additionally, supplementary comment lines can be specified in the `comments` parameter.

Parameters

- **fname** (*str*) – a file name where to store the formula.
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

```
>>> from pysat.formula import WCNFPlus
>>> cnf = WCNFPlus()
...
```

(continues on next page)

(continued from previous page)

```
>>> # the formula is filled with a bunch of clauses
>>> with open('some-file.wcnf+', 'w') as fp:
...     cnf.to_fp(fp)  # writing to the file pointer
```

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

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: $(\sum_{i=1}^{n} a_i \cdot x_i) \circ k$, 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 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`.

[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

[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

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*, *encoding=0*)
A synonym for *PBEnc.geq()*.

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

classmethod equals(*lits*, *weights=None*, *bound=1*, *top_id=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*, *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*, *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.
- **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.
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:

- 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:

[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

```
>>> 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

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 `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 *'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 `Lingeling`, `Glucose3`, `Glucose4`, `MapleChrono`, `MapleCM` and `Maplesat` can have additional default arguments. One such argument supported by `Glucose3` and `Glucose4` but also by `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 `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.

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`.

[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

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
```

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.

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
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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. `Lingeling` does 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.

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

Usage example:

```
>>> from pysat.solvers import Glucose3
>>>
>>> g = Glucose3(bootstrap_with=[[1, 2]])
```

(continues on next page)

(continued from previous page)

```
>>> # 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. `Lingeling` does 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).

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
...     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.

```
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
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
```

[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

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.

Continued on next page

Table 6 – continued from previous page

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

[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

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

class examples.genhard.**CB**(*size*, *exhaustive=False*, *topv=0*, *verb=False*)

Mutilated chessboard principle (CB). Given an integer n, the principle states that it is impossible to cover a chessboard of size $2n \cdot 2n$ by domino tiles if two diagonally opposite corners of the chessboard are removed.

Note that the chessboard has $4n^2 - 2$ cells. Introduce a Boolean variable x_{ij} for $i, j \in [4n^2 - 2]$ s.t. cells i and j are adjacent (no variables are introduced for pairs of non-adjacent cells). CB formulas comprise clauses (1) $(\neg x_{ji} \vee \neg x_{ki})$ for every $i, j \neq k$ meaning that no more than one adjacent cell can be paired with the current one; and (2) $(\vee_{j \in \text{Adj}(i)} x_{ij})$ $\forall i$ enforcing that every cell i should be paired with at least one adjacent cell.

Clearly, since the two diagonal corners are removed, the formula is unsatisfiable. Also note the following. Assuming that the number of black cells is larger than the number of the white ones, CB formulas are unsatisfiable even if only a half of the formula is present, e.g. when AtMost1 constraints are formulated only for the white cells while the AtLeast1 constraints are formulated only for the black cells. Depending on the value of parameter `exhaustive` the encoder applies the *complete* or *partial* formulation of the problem.

Mutilated chessboard principle is known to be hard for resolution[3].

Parameters

- **size** (*int*) – problem size (n)
- **exhaustive** (*bool*) – encode the problem exhaustively
- **topv** (*int*) – current top variable identifier
- **verb** (*bool*) – defines whether or not the encoder is verbose

Returns object of class *pysat.formula.CNF*.

class examples.genhard.**GT**(*size*, *topv=0*, *verb=False*)

Generator of ordering (or *greater than*, GT) principle formulas. Given an integer parameter n, the principle states that any partial order on the set $\{1, 2, \ldots, n\}$ must have a maximal element.

Assume variable x_{ij}, for $i, j \in [n], i \neq j$, denotes the fact that $i \succ j$. Clauses $(\neg x_{ij} \vee \neg x_{ji})$ and $(\neg x_{ij} \vee \neg x_{jk} \vee x_{ik})$ ensure that the relation $\succ$ is anti-symmetric and transitive. As a result, $\succ$ is a partial order on $[n]$. The additional requirement that each element i has a successor in $[n] \setminus \{i\}$ represented a clause $(\vee_{j \neq i} x_{ji})$ makes the formula unsatisfiable.

GT formulas were originally conjectured[2] to be hard for resolution. However,[5] proved the existence of a polynomial size resolution refutation for GT formulas.

Parameters

- **size** (*int*) – number of elements (n)
- **topv** (*int*) – current top variable identifier

[5] Gunnar Stålmarck. *Short Resolution Proofs for a Sequence of Tricky Formulas*. Acta Informatica. 33(3). 1996. pp. 277-280

- **verb** (*bool*) – defines whether or not the encoder is verbose

Returns object of class *pysat.formula.CNF*.

class examples.genhard.**PAR**(*size*, *topv=0*, *verb=False*)

Generator of the parity principle (PAR) formulas. Given an integer parameter n, the principle states that no graph on $2n + 1$ nodes consists of a complete perfect matching.

The encoding of the parity principle uses $\binom{2n+1}{2}$ variables $x_{ij}, i \neq j$. If variable x_{ij} is *true*, then there is an edge between nodes i and j. The formula consists of the following clauses: $(\vee_{j \neq i} x_{ij})$ for every $i \in [2n + 1]$, and $(\neg x_{ij} \vee \neg x_{kj})$ for all distinct $i, j, k \in [2n + 1]$.

The parity principle is known to be hard for resolution[4].

Parameters

- **size** (*int*) – problem size (n)
- **topv** (*int*) – current top variable identifier
- **verb** (*bool*) – defines whether or not the encoder is verbose

Returns object of class *pysat.formula.CNF*.

class examples.genhard.**PHP**(*nof_holes*, *kval=1*, *topv=0*, *verb=False*)

Generator of k pigeonhole principle (k-PHP) formulas. Given integer parameters m and k, the k pigeonhole principle states that if $k \cdot m + 1$ pigeons are distributes by m holes, then at least one hole contains more than k pigeons.

Note that if k is 1, the principle degenerates to the formulation of the original pigeonhole principle stating that $m + 1$ pigeons cannot be distributed by m holes.

Assume that a Boolean variable x_{ij} encodes that pigeon i resides in hole j. Then a PHP formula can be seen as a conjunction: $\bigwedge_{i=1}^{k \cdot m+1} \mathsf{AtLeast1}(x_{i1}, \ldots, x_{im}) \wedge \bigwedge_{j=1}^{m} \mathsf{AtMost}k(x_{1j}, \ldots, x_{k \cdot m+1,j})$. Here each AtLeast1 constraint forces every pigeon to be placed into at least one hole while each AtMost$k$ constraint allows the corresponding hole to have at most k pigeons. The overall PHP formulas are unsatisfiable.

PHP formulas are well-known[6] to be hard for resolution.

Parameters

- **nof_holes** (*int*) – number of holes (n)
- **kval** (*int*) – multiplier k
- **topv** (*int*) – current top variable identifier
- **verb** (*bool*) – defines whether or not the encoder is verbose

Returns object of class *pysat.formula.CNF*.

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

List of classes

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

[6] Armin Haken. *The Intractability of Resolution*. Theor. Comput. Sci. 39. 1985. pp. 297-308

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.

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]
```

(continues on next page)

[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

(continued from previous page)

```
[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
...
...     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

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

- **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.
LBXPlus	LBX-like algorithm extended for *WCNFPlus* formulas (using Minicard).

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 heuris-

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

tics 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.

```
>>> 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`).

[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

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
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

class examples.lsu.**LSU**(*formula*, *solver='g4'*, *verbose=0*)

Linear SAT-UNSAT algorithm for MaxSAT[1]. The algorithm can be seen as a series of satisfiability oracle calls refining an upper bound on the MaxSAT cost, followed by one unsatisfiability call, which stops the algorithm. The implementation encodes the sum of all selector literals using the *iterative totalizer encoding*[2]. At every iteration, the upper bound on the cost is reduced and enforced by adding the corresponding unit size clause to the working formula. No clauses are removed during the execution of the algorithm. As a result, the SAT oracle is used incrementally.

> **Warning:** At this point, *LSU* supports only **unweighted** problems.

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

The constructor receives an input *WCNF* formula, a name of the SAT solver to use (see *SolverNames* for details), and an integer verbosity level.

Parameters

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

_assert_lt(*cost*)

The method enforces an upper bound on the cost of the MaxSAT solution. This is done by encoding the sum of all soft clause selectors with the use the iterative totalizer encoding, i.e. *ITotalizer*. Note that the sum is created once, at the beginning. Each of the following calls to this method only enforces the upper bound on the created sum by adding the corresponding unit size clause. Each such clause is added on the fly with no restart of the underlying SAT oracle.

Parameters **cost** (*int*) – the cost of the next MaxSAT solution is enforced to be *lower* than this current cost

_get_model_cost(*formula*, *model*)

Given a WCNF formula and a model, the method computes the MaxSAT cost of the model, i.e. the sum of weights of soft clauses that are unsatisfied by the model.

Parameters

- **formula** (*WCNF*) – an input MaxSAT formula
- **model** (*list(int)*) – a satisfying assignment

Return type int

_init(*formula*)

SAT oracle initialization. The method creates a new SAT oracle and feeds it with the formula's hard clauses. Afterwards, all soft clauses of the formula are augmented with selector literals and also added to the solver. The list of all introduced selectors is stored in variable `self.sels`.

Parameters **formula** (`WCNF`) – input MaxSAT formula

delete()

Explicit destructor of the internal SAT oracle and the *ITotalizer* object.

get_model()

This method returns a model obtained during a prior satisfiability oracle call made in *solve()*.

Return type list(int)

oracle_time()

Method for calculating and reporting the total SAT solving time.

solve()

Computes a solution to the MaxSAT problem. The method implements the LSU/LSUS algorithm, i.e. it represents a loop, each iteration of which calls a SAT oracle on the working MaxSAT formula and refines the upper bound on the MaxSAT cost until the formula becomes unsatisfiable.

Returns `True` if the hard part of the MaxSAT formula is satisfiable, i.e. if there is a MaxSAT solution, and `False` otherwise.

Return type bool

class examples.lsu.**LSUPlus**(*formula*, *verbose=0*)

LSU-like algorithm extended for *WCNFPlus* formulas (using `Minicard`).

Parameters

- **formula** (*WCNFPlus*) – input MaxSAT formula in WCNF+ format
- **verbose** (*int*) – verbosity level

_assert_lt(*cost*)

Overrides _assert_lt of *LSU* in order to use Minicard's native support for cardinality constraints

Parameters **cost** (*int*) – the cost of the next MaxSAT solution is enforced to be *lower* than this current cost

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

List of classes

MCSls	Algorithm BLS for computing MCSes, augmented with "clause D" calls.
MCSlsPlus	Extension of the algorithm for *WCNFPlus* formulas (using Minicard).

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

$ 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
>>>
```

(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)

```
>>> 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 *MCSls* 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 *MCSls*, 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 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

$ 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
```

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

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.

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.

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

1.2.8 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 Evaluation 2018 where, surprisingly, it was ranked first in two complete categories: *unweighted* and *weighted*. A brief solver description can be found in[3].

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)[4] and stratification[5] 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
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')
>>>
```

(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] 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
[5] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, Jordi Levy. *Improving WPM2 for (Weighted) Partial MaxSAT*. CP 2013. pp. 117-132

(continued from previous page)

```
>>> 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

class examples.rc2.**RC2**(*formula*, *solver='g3'*, *adapt=False*, *exhaust=False*, *incr=False*, *minz=False*, *trim=0*, *verbose=0*)

Implementation of the basic RC2 algorithm. Given a (weighted) (partial) CNF formula, i.e. formula in the *WCNF* format, this class can be used to compute a given number of MaxSAT solutions for the input formula. *RC2* roughly follows the implementation of algorithm OLLITI[12] of MSCG and applies a few heuristics on top of it. These include

- *unsatisfiable core exhaustion* (see method *exhaust_core()*),
- *unsatisfiable core reduction* (see method *minimize_core()*),
- *intrinsic AtMost1 constraints* (see method *adapt_am1()*).

RC2 can use any SAT solver available in PySAT. The default SAT solver to use is `g3` (see *SolverNames*). Additionally, if Glucose is chosen, the `incr` parameter controls whether to use the incremental mode of Glucose[6] (turned off by default). Boolean parameters `adapt`, `exhaust`, and `minz` control whether or to apply detection and adaptation of intrinsic AtMost1 constraints, core exhaustion, and core reduction. Unsatisfiable cores can be trimmed if the `trim` parameter is set to a non-zero integer. Finally, verbosity level can be set using the `verbose` parameter.

Parameters

- **formula** (*WCNF*) – (weighted) (partial) CNF formula
- **solver** (*str*) – SAT oracle name
- **adapt** (*bool*) – detect and adapt intrinsic AtMost1 constraints
- **exhaust** (*bool*) – do core exhaustion
- **incr** (*bool*) – use incremental mode of Glucose
- **minz** (*bool*) – do heuristic core reduction
- **trim** (*int*) – do core trimming at most this number of times
- **verbose** (*int*) – verbosity level

_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()*).

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

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

adapt_am1()

Detect and adapt intrinsic AtMost1 constraints. Assume there is a subset of soft clauses $\mathcal{S}' \subseteq \mathcal{S}$ s.t. $\sum_{c \in \mathcal{S}'} c \leq 1$, i.e. at most one of the clauses of $\mathcal{S}'$ can be satisfied.

Each AtMost1 relationship between the soft clauses can be detected in the following way. The method traverses all soft clauses of the formula one by one, sets one respective selector literal to true and checks whether some other soft clauses are forced to be false. This is checked by testing if selectors for other soft clauses are unit-propagated to be false. Note that this method for detection of AtMost1 constraints is *incomplete*, because in general unit propagation does not suffice to test whether or not $\mathcal{F} \wedge l_i \models \neg l_j$.

Each intrinsic AtMost1 constraint detected this way is handled by calling *process_am1()*.

add_clause(*clause*, *weight=None*)

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 *RC2*, adding clauses may be helpful when *enumerating* MaxSAT solutions 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 integer parameter `weight` can be set to meaning the the clause being added is soft having the corresponding weight (note that parameter `weight` is set to `None` by default meaning that the clause is hard).

Parameters

- **clause** (*iterable(int)*) – a clause to add
- **weight** (*int*) – weight of the clause (if any)

```
>>> from pysat.examples.rc2 import RC2
>>> from pysat.formula import WCNF
>>>
>>> wcnf = WCNF()
>>> wcnf.append([-1, -2])  # adding hard clauses
>>> wcnf.append([-1, -3])
>>>
>>> wcnf.append([1], weight=1)  # adding soft clauses
>>> wcnf.append([2], weight=1)
>>> wcnf.append([3], weight=1)
>>>
>>> with RC2(wcnf) as rc2:
...     rc2.compute()  # solving the MaxSAT problem
[-1, 2, 3]
...     print rc2.cost
1
...     rc2.add_clause([-2, -3])  # adding one more hard clause
...     rc2.compute()  # computing another model
[-1, -2, 3]
...     print rc2.cost
2
```

compute()

This method can be used for computing one MaxSAT solution, i.e. for computing an assignment satisfying all hard clauses of the input formula and maximizing the sum of weights of satisfied soft clauses. It is a wrapper for the internal *compute_()* method, which does the job, followed by the model extraction.

Note that the method returns `None` if no MaxSAT model exists. The method can be called multiple times, each being followed by blocking the last model. This way one can enumerate top-k MaxSAT solutions (this can also be done by calling *enumerate()*).

Returns a MaxSAT model

Return type list(int)

```
>>> from pysat.examples.rc2 import RC2
>>> from pysat.formula import WCNF
>>>
>>> rc2 = RC2(WCNF())  # passing an empty WCNF() formula
>>> rc2.add_clause([-1, -2])
>>> rc2.add_clause([-1, -3])
>>> rc2.add_clause([-2, -3])
>>>
>>> rc2.add_clause([1], weight=1)
>>> rc2.add_clause([2], weight=1)
>>> rc2.add_clause([3], weight=1)
>>>
>>> model = rc2.compute()
>>> print model
[-1, -2, 3]
>>> print rc2.cost
2
>>> rc2.delete()
```

compute_()

Main core-guided loop, which iteratively calls a SAT oracle, extracts a new unsatisfiable core and processes it. The loop finishes as soon as a satisfiable formula is obtained. If specified in the command line, the method additionally calls *adapt_am1()* to detect and adapt intrinsic AtMost1 constraints before executing the loop.

Return type bool

create_sum(*bound=1*)

Create a totalizer object encoding a cardinality constraint on the new list of relaxation literals obtained in *process_sels()* and *process_sums()*. The clauses encoding the sum of the relaxation literals are added to the SAT oracle. The sum of the totalizer object is encoded up to the value of the input parameter `bound`, which is set to `1` by default.

Parameters **bound** (*int*) – right-hand side for the sum to be created

Return type *ITotalizer*

Note that if Minicard is used as a SAT oracle, native cardinality constraints are used instead of *ITotalizer*.

delete()

Explicit destructor of the internal SAT oracle and all the totalizer objects creating during the solving process.

enumerate()

Enumerate top MaxSAT solutions (from best to worst). The method works as a generator, which iteratively calls *compute()* to compute a MaxSAT model, blocks it internally and returns it.

Returns a MaxSAT model

Return type list(int)

```
>>> from pysat.examples.rc2 import RC2
>>> from pysat.formula import WCNF
>>>
>>> rc2 = RC2(WCNF())  # passing an empty WCNF() formula
>>> rc2.add_clause([-1, -2])  # adding clauses "on the fly"
>>> rc2.add_clause([-1, -3])
>>> rc2.add_clause([-2, -3])
>>>
>>> rc2.add_clause([1], weight=1)
>>> rc2.add_clause([2], weight=1)
>>> rc2.add_clause([3], weight=1)
>>>
>>> for model in rc2.enumerate():
...     print model, rc2.cost
[-1, -2, 3] 2
[1, -2, -3] 2
[-1, 2, -3] 2
[-1, -2, -3] 3
>>> rc2.delete()
```

exhaust_core(*tobj*)

Exhaust core by increasing its bound as much as possible. Core exhaustion was originally referred to as *cover optimization* in[5].

Given a totalizer object `tobj` representing a sum of some *relaxation* variables $r \in R$ that augment soft clauses $\mathcal{C}_r$, the idea is to increase the right-hand side of the sum (which is equal to 1 by default) as much as possible, reaching a value k s.t. formula $\mathcal{H} \wedge \mathcal{C}_r \wedge (\sum_{r \in R} r \leq k)$ is still unsatisfiable while increasing it further makes the formula satisfiable (here $\mathcal{H}$ denotes the hard part of the formula).

The rationale is that calling an oracle incrementally on a series of slightly modified formulas focusing only on the recently computed unsatisfiable core and disregarding the rest of the formula may be practically effective.

filter_assumps()

Filter out unnecessary selectors and sums from the list of assumption literals. The corresponding values are also removed from the dictionaries of bounds and weights.

Note that assumptions marked as garbage are collected in the core processing methods, i.e. in *process_core()*, *process_sels()*, and *process_sums()*.

get_core()

Extract unsatisfiable core. The result of the procedure is stored in variable `self.core`. If necessary, core trimming and also heuristic core reduction is applied depending on the command-line options. A *minimum weight* of the core is computed and stored in `self.minw`. Finally, the core is divided into two parts:

1. clause selectors (`self.core_sels`),
2. sum assumptions (`self.core_sums`).

init(*formula*, *incr=False*)

Initialize the internal SAT oracle. The oracle is used incrementally and so it is initialized only once when constructing an object of class *RC2*. Given an input *WCNF* formula, the method bootstraps the oracle with its hard clauses. It also augments the soft clauses with "fresh" selectors and adds them to the oracle afterwards.

Optional input parameter `incr` (`False` by default) regulates whether or not Glucose's incremental mode[6] is turned on.

Parameters

- **formula** (*WCNF*) – input formula
- **incr** (*bool*) – apply incremental mode of Glucose

minimize_core()
Reduce a previously extracted core and compute an over-approximation of an MUS. This is done using the simple deletion-based MUS extraction algorithm.

The idea is to try to deactivate soft clauses of the unsatisfiable core one by one while checking if the remaining soft clauses together with the hard part of the formula are unsatisfiable. Clauses that are necessary for preserving unsatisfiability comprise an MUS of the input formula (it is contained in the given unsatisfiable core) and are reported as a result of the procedure.

During this core minimization procedure, all SAT calls are dropped after obtaining 1000 conflicts.

oracle_time()
Report the total SAT solving time.

process_am1(*am1*)
Process an AtMost1 relation detected by *adapt_am1()*. Note that given a set of soft clauses $\mathcal{S}'$ at most one of which can be satisfied, one can immediately conclude that the formula has cost at least $|\mathcal{S}'| - 1$ (assuming *unweighed* MaxSAT). Furthermore, it is safe to replace all clauses of $\mathcal{S}'$ with a single soft clause $\sum_{c \in \mathcal{S}'} c$.

Here, input parameter `am1` plays the role of subset $\mathcal{S}'$ mentioned above. The procedure bumps the MaxSAT cost by `self.minw * (len(am1) - 1)`.

All soft clauses involved in `am1` are replaced by a single soft clause, which is a disjunction of the selectors of clauses in `am1`. The weight of the new soft clause is set to `self.minw`.

Parameters **am1** (*list(int)*) – a list of selectors connected by an AtMost1 constraint

process_core()
The method deals with a core found previously in *get_core()*. Clause selectors `self.core_sels` and sum assumptions involved in the core are treated separately of each other. This is handled by calling methods *process_sels()* and *process_sums()*, respectively. Whenever necessary, both methods relax the core literals, which is followed by creating a new totalizer object encoding the sum of the new relaxation variables. The totalizer object can be "exhausted" depending on the option.

process_sels()
Process soft clause selectors participating in a new core. The negation $\neg s$ of each selector literal s participating in the unsatisfiable core is added to the list of relaxation literals, which will be later used to create a new totalizer object in *create_sum()*.

If the weight associated with a selector is equal to the minimal weight of the core, e.g. `self.minw`, the selector is marked as garbage and will be removed in *filter_assumps()*. Otherwise, the clause is split as described in[1].

process_sums()
Process cardinality sums participating in a new core. Whenever necessary, some of the sum assumptions are removed or split (depending on the value of `self.minw`). Deleted sums are marked as garbage and are dealt with in *filter_assumps()*.

In some cases, the process involves updating the right-hand sides of the existing cardinality sums (see the call to *update_sum()*). The overall procedure is detailed in[1].

set_bound(*tobj*, *rhs*)
Given a totalizer sum and its right-hand side to be enforced, the method creates a new sum assumption literal, which will be used in the following SAT oracle calls.

Parameters

- **tobj** (*ITotalizer*) – totalizer sum
- **rhs** (*int*) – right-hand side

trim_core()
This method trims a previously extracted unsatisfiable core at most a given number of times. If a fixed point is reached before that, the method returns.

update_sum(*assump*)
The method is used to increase the bound for a given totalizer sum. The totalizer object is identified by the input parameter `assump`, which is an assumption literal associated with the totalizer object.

The method increases the bound for the totalizer sum, which involves adding the corresponding new clauses to the internal SAT oracle.

The method returns the totalizer object followed by the new bound obtained.

Parameters **assump** (*int*) – assumption literal associated with the sum

Return type *ITotalizer*, int

Note that if Minicard is used as a SAT oracle, native cardinality constraints are used instead of *ITotalizer*.

class examples.rc2.**RC2Stratified**(*formula*, *solver='g3'*, *adapt=False*, *exhaust=False*, *incr=False*, *minz=False*, *trim=0*, *verbose=0*)
RC2 augmented with BLO and stratification techniques. Although class *RC2* can deal with weighted formulas, there are situations when it is necessary to apply additional heuristics to improve the performance of the solver on weighted MaxSAT formulas. This class extends capabilities of *RC2* with two heuristics, namely

1. Boolean lexicographic optimization (BLO)[4]
2. stratification[5]

There is no way to enable only one of them – both heuristics are applied at the same time. Except for the aforementioned additional techniques, every other component of the solver remains as in the base class *RC2*. Therefore, a user is referred to the documentation of *RC2* for details.

activate_clauses(*beg*)
This method is used for activating the clauses that belong to optimization levels up to the newly computed level. It also reactivates previously deactivated clauses (see *process_sels()* and *process_sums()* for details).

compute()
This method solves the MaxSAT problem iteratively. Each optimization level is tackled the standard way, i.e. by calling `compute_()`. A new level is started by calling *next_level()* and finished by calling *finish_level()*. Each new optimization level activates more soft clauses by invoking *activate_clauses()*.

finish_level()
This method does postprocessing of the current optimization level after it is solved. This includes *hardening* some of the soft clauses (depending on their remaining weights) and also garbage collection.

init_wstr()
Compute and initialize optimization levels for BLO and stratification. This method is invoked once, from the constructor of an object of *RC2Stratified*. Given the weights of the soft clauses, the method divides the MaxSAT problem into several optimization levels.

next_level()
Compute the next optimization level (starting from the current one). The procedure represents a loop, each iteration of which checks whether or not one of the conditions holds:

- partial BLO condition

- stratification condition

If any of these holds, the loop stops.

process_am1(*am1*)
Due to the solving process involving multiple optimization levels to be treated individually, new soft clauses for the detected intrinsic AtMost1 constraints should be remembered. The method is a slightly modified version of the base method *RC2.process_am1()* taking care of this.

process_sels()
A redefined version of *RC2.process_sels()*. The only modification affects the clauses whose weight after splitting becomes less than the weight of the current optimization level. Such clauses are deactivated and to be reactivated at a later stage.

process_sums()
A redefined version of *RC2.process_sums()*. The only modification affects the clauses whose weight after splitting becomes less than the weight of the current optimization level. Such clauses are deactivated and to be reactivated at a later stage.

PYTHON MODULE INDEX

e

examples.fm, 35
examples.genhard, 37
examples.hitman, 40
examples.lbx, 43
examples.lsu, 46
examples.mcsls, 49
examples.musx, 52
examples.rc2, 54

p

pysat.card, 3
pysat.formula, 8
pysat.pb, 23
pysat.solvers, 25

INDEX

Symbols

_assert_lt() (*examples.lsu.LSU method*), 48
_assert_lt() (*examples.lsu.LSUPlus method*), 49
_compute() (*examples.fm.FM method*), 36
_compute() (*examples.lbx.LBX method*), 44
_compute() (*examples.mcsls.MCSls method*), 50
_compute() (*examples.musx.MUSX method*), 53
_filter_satisfied() (*examples.lbx.LBX method*), 45
_get_model_cost() (*examples.lsu.LSU method*), 48
_init() (*examples.lsu.LSU method*), 48
_map_extlit() (*examples.lbx.LBX method*), 45
_map_extlit() (*examples.mcsls.MCSls method*), 50
_map_extlit() (*examples.rc2.RC2 method*), 55
_overapprox() (*examples.mcsls.MCSls method*), 51
_satisfied() (*examples.lbx.LBX method*), 45

A

activate_clauses() (*examples.rc2.RC2Stratified method*), 60
adapt_am1() (*examples.rc2.RC2 method*), 56
add_atmost() (*pysat.solvers.Solver method*), 28
add_clause() (*examples.lbx.LBX method*), 45
add_clause() (*examples.mcsls.MCSls method*), 51
add_clause() (*examples.rc2.RC2 method*), 56
add_clause() (*pysat.solvers.Solver method*), 28
append() (*pysat.formula.CNF method*), 11
append() (*pysat.formula.CNFPlus method*), 15
append() (*pysat.formula.WCNF method*), 17
append() (*pysat.formula.WCNFPlus method*), 21
append_formula() (*pysat.solvers.Solver method*), 29
atleast() (*pysat.card.CardEnc class method*), 4
atleast() (*pysat.pb.PBEnc class method*), 24
atmost() (*pysat.card.CardEnc class method*), 5
atmost() (*pysat.pb.PBEnc class method*), 24

B

block() (*examples.hitman.Hitman method*), 43
block() (*examples.lbx.LBX method*), 46
block() (*examples.mcsls.MCSls method*), 51

C

CardEnc (*class in pysat.card*), 4
CB (*class in examples.genhard*), 39
CNF (*class in pysat.formula*), 11
CNFPlus (*class in pysat.formula*), 14
compute() (*examples.fm.FM method*), 36
compute() (*examples.lbx.LBX method*), 46
compute() (*examples.mcsls.MCSls method*), 51
compute() (*examples.musx.MUSX method*), 53
compute() (*examples.rc2.RC2 method*), 56
compute() (*examples.rc2.RC2Stratified method*), 60
compute_() (*examples.rc2.RC2 method*), 57
conf_budget() (*pysat.solvers.Solver method*), 29
copy() (*pysat.formula.CNF method*), 11
copy() (*pysat.formula.WCNF method*), 18
create_sum() (*examples.rc2.RC2 method*), 57

D

delete() (*examples.fm.FM method*), 36
delete() (*examples.hitman.Hitman method*), 43
delete() (*examples.lbx.LBX method*), 46
delete() (*examples.lsu.LSU method*), 48
delete() (*examples.mcsls.MCSls method*), 51
delete() (*examples.musx.MUSX method*), 53
delete() (*examples.rc2.RC2 method*), 57
delete() (*pysat.card.ITotalizer method*), 6
delete() (*pysat.solvers.Solver method*), 29
do_cld_check() (*examples.lbx.LBX method*), 46
do_cld_check() (*examples.mcsls.MCSls method*), 51

E

EncType (*class in pysat.card*), 5
EncType (*class in pysat.pb*), 23
enum_models() (*pysat.solvers.Solver method*), 29
enumerate() (*examples.hitman.Hitman method*), 43
enumerate() (*examples.lbx.LBX method*), 46
enumerate() (*examples.mcsls.MCSls method*), 51
enumerate() (*examples.rc2.RC2 method*), 57
equals() (*pysat.card.CardEnc class method*), 5
equals() (*pysat.pb.PBEnc class method*), 24
examples.fm (*module*), 35
examples.genhard (*module*), 37

examples.hitman (*module*), 40
examples.lbx (*module*), 43
examples.lsu (*module*), 46
examples.mcsls (*module*), 49
examples.musx (*module*), 52
examples.rc2 (*module*), 54
exhaust_core() (*examples.rc2.RC2 method*), 58
extend() (*pysat.card.ITotalizer method*), 6
extend() (*pysat.formula.CNF method*), 11
extend() (*pysat.formula.WCNF method*), 18

F

filter_assumps() (*examples.rc2.RC2 method*), 58
finish_level() (*examples.rc2.RC2Stratified method*), 60
FM (*class in examples.fm*), 36
from_clauses() (*pysat.formula.CNF method*), 12
from_file() (*pysat.formula.CNF method*), 12
from_file() (*pysat.formula.WCNF method*), 18
from_fp() (*pysat.formula.CNF method*), 12
from_fp() (*pysat.formula.CNFPlus method*), 15
from_fp() (*pysat.formula.WCNF method*), 19
from_fp() (*pysat.formula.WCNFPlus method*), 22
from_string() (*pysat.formula.CNF method*), 13
from_string() (*pysat.formula.WCNF method*), 19

G

geq() (*pysat.pb.PBEnc class method*), 24
get() (*examples.hitman.Hitman method*), 43
get_core() (*examples.rc2.RC2 method*), 58
get_core() (*pysat.solvers.Solver method*), 30
get_model() (*examples.lsu.LSU method*), 48
get_model() (*pysat.solvers.Solver method*), 30
get_proof() (*pysat.solvers.Solver method*), 30
get_status() (*pysat.solvers.Solver method*), 31
GT (*class in examples.genhard*), 39

H

hit() (*examples.hitman.Hitman method*), 43
Hitman (*class in examples.hitman*), 42

I

id() (*pysat.formula.IDPool method*), 16
IDPool (*class in pysat.formula*), 16
increase() (*pysat.card.ITotalizer method*), 6
init() (*examples.fm.FM method*), 36
init() (*examples.hitman.Hitman method*), 43
init() (*examples.rc2.RC2 method*), 58
init_wstr() (*examples.rc2.RC2Stratified method*), 60
ITotalizer (*class in pysat.card*), 5

L

LBX (*class in examples.lbx*), 44
leq() (*pysat.pb.PBEnc class method*), 24
LSU (*class in examples.lsu*), 47
LSUPlus (*class in examples.lsu*), 48

M

MCSls (*class in examples.mcsls*), 50
merge_with() (*pysat.card.ITotalizer method*), 7
minimize_core() (*examples.rc2.RC2 method*), 59
MUSX (*class in examples.musx*), 53

N

negate() (*pysat.formula.CNF method*), 13
new() (*pysat.card.ITotalizer method*), 8
new() (*pysat.solvers.Solver method*), 31
next_level() (*examples.rc2.RC2Stratified method*), 60
nof_clauses() (*pysat.solvers.Solver method*), 31
nof_vars() (*pysat.solvers.Solver method*), 31
NoSuchEncodingError, 8, 24
NoSuchSolverError, 27

O

obj() (*pysat.formula.IDPool method*), 17
occupy() (*pysat.formula.IDPool method*), 17
oracle_time() (*examples.fm.FM method*), 37
oracle_time() (*examples.lbx.LBX method*), 46
oracle_time() (*examples.lsu.LSU method*), 48
oracle_time() (*examples.mcsls.MCSls method*), 52
oracle_time() (*examples.musx.MUSX method*), 53
oracle_time() (*examples.rc2.RC2 method*), 59

P

PAR (*class in examples.genhard*), 40
PBEnc (*class in pysat.pb*), 24
PHP (*class in examples.genhard*), 40
process_am1() (*examples.rc2.RC2 method*), 59
process_am1() (*examples.rc2.RC2Stratified method*), 61
process_core() (*examples.rc2.RC2 method*), 59
process_sels() (*examples.rc2.RC2 method*), 59
process_sels() (*examples.rc2.RC2Stratified method*), 61
process_sums() (*examples.rc2.RC2 method*), 59
process_sums() (*examples.rc2.RC2Stratified method*), 61
prop_budget() (*pysat.solvers.Solver method*), 31
propagate() (*pysat.solvers.Solver method*), 32
pysat.card (*module*), 3
pysat.formula (*module*), 8
pysat.pb (*module*), 23
pysat.solvers (*module*), 25

R

RC2 (*class in examples.rc2*), 55

RC2Stratified (*class in examples.rc2*), 60
reinit() (*examples.fm.FM method*), 37
relax_core() (*examples.fm.FM method*), 37
remove_unit_core() (*examples.fm.FM method*), 37
restart() (*pysat.formula.IDPool method*), 17

S

set_bound() (*examples.rc2.RC2 method*), 59
set_phases() (*pysat.solvers.Solver method*), 32
solve() (*examples.lsu.LSU method*), 48
solve() (*pysat.solvers.Solver method*), 33
solve_limited() (*pysat.solvers.Solver method*), 33
Solver (*class in pysat.solvers*), 27
SolverNames (*class in pysat.solvers*), 34
split_core() (*examples.fm.FM method*), 37

T

time() (*pysat.solvers.Solver method*), 34
time_accum() (*pysat.solvers.Solver method*), 34
to_file() (*pysat.formula.CNF method*), 13
to_file() (*pysat.formula.WCNF method*), 20
to_fp() (*pysat.formula.CNF method*), 14
to_fp() (*pysat.formula.CNFPlus method*), 16
to_fp() (*pysat.formula.WCNF method*), 20
to_fp() (*pysat.formula.WCNFPlus method*), 22
treat_core() (*examples.fm.FM method*), 37
trim_core() (*examples.rc2.RC2 method*), 60

U

unweighed() (*pysat.formula.WCNF method*), 20
update_sum() (*examples.rc2.RC2 method*), 60

W

WCNF (*class in pysat.formula*), 17
WCNFPlus (*class in pysat.formula*), 21
weighted() (*pysat.formula.CNF method*), 14
with_traceback() (*pysat.card.NoSuchEncodingError method*), 8
with_traceback() (*pysat.pb.NoSuchEncodingError method*), 24
with_traceback() (*pysat.solvers.NoSuchSolverError method*), 27