PySAT Documentation

Release 0.1.3.dev14

Alexey Ignatiev, Joao Marques-Silva, Antonio Morgado

Aug 17, 2018

CONTENTS

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

CHAPTER

ONE

API DOCUMENTATION

The PySAT toolkit has three core modules: `card`, `formula`, and `solvers`. The two of them (`card` and `solvers`) are Python wrappers for the code originally implemented in the C/C++ languages while the `formula` module is a *pure* Python module.

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.

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

[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

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** (*list(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 ***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** (*list (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** (*list(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.

class pysat.card.**KAMTotalizer**(*lits=[]*, *top_id=None*, *approx=True*)
KAM totalizer.

atmost(*bound=1*, *top_id=None*)
Increase a possible upper bound (right-hand side) in an existing totalizer object.

delete()
Destroy a kamto object.

new(*lits=[]*, *top_id=None*, *approx=True*)
Create a new kamto object.

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.

Continued on next page

Table 2 – continued from previous page

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
```

(continues on next page)

(continued from previous page)

```
...
...     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):
    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']*)

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.

Parameters

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

Usage example:

```
>>> from pysat.formula import CNF
>>> cnf1 = CNF()
>>> cnf1.from_file('some-file.cnf')
>>>
>>> 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)
>>>
>>> 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*)

The method is for saving a CNF formula into a file in the DIMACS CNF format. A file name is expected

[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

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
>>> 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']*)
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.

Parameters

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

Usage example:

```
>>> from pysat.formula import WCNF
>>> cnf1 = WCNF()
>>> cnf1.from_file('some-file.wcnf')
>>>
>>> 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
[[-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
```

(continues on next page)

(continued from previous page)

```
[[-3]]
>>> print cnf2.nv
3
```

to_file (*fname*, *comments=None*)

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.

Parameters

- **fname** (*str*) – a file name where to store the formula.
- **comments** (*list of 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
>>> 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 of 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()
```

(continues on next page)

(continued from previous page)

```
>>> 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()
...
>>> # 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 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.
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)
- 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`, `Minicard`, `Minisat22`, and `MinisatGH`. However, the solvers can also be accessed through the common base class *Solver* using the solver `name` argument. For example, both of the following pieces of code create a copy of the `Glucose3` solver:

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

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

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

[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

- 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`, `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'*, *'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** (*`list(list(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`, and `Glucose4` can have additional default arguments. One such argument supported by `Glucose3` and `Glucose4` but also by `Lingeling` 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.

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

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

Parameters

- **lits** (*list (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** (*list (int)*) – a list of 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(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** (*list (list (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** (*list(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
```

(continues on next page)

(continued from previous page)

```
[-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
>>>
>>> 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** (*list(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)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> 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`, `Minisat22`, and `Minicard` can redefine the preferences in any of the following SAT calls due to the phase saving heuristic.

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

Usage example:

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

solve (*assumptions=[]*)

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

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

Parameters **assumptions** (*list(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** (*list(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')
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 (not yet documented)

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

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

Algorithm FM - FU & Malik - MSU1.

compute()

Compute and return a solution.

delete()

Explicit destructor.

init(*with_soft=True*)

Initialize the SAT solver.

oracle_time()

Report the total SAT solving time.

reinit()

Delete and create a new SAT solver.

relax_core()
Relax and bound the core.

remove_unit_core()
Remove a clause responsible for a unit core.

split_core(*minw*)
Split clauses in the core whenever necessary.

treat_core()
Found core in main loop, deal with it.

examples.fm.**parse_options**()
Parses command-line option

examples.fm.**usage**()
Prints usage message.

1.2.2 Hard formula generator (pysat.examples.genhard)

class examples.genhard.**CB**(*size*, *exhaustive=False*, *topv=0*, *verb=False*)
Mutilated chessboard principle for the chessboard of 2size x 2size.

class examples.genhard.**GT**(*size*, *topv=0*, *verb=False*)
GT (greater than) principle formula for a set of elements of a given size.

class examples.genhard.**PHP**(*nof_holes*, *kval=1*, *topv=0*, *verb=False*)
Pigeonhole principle formula for (kval * nof_holes + 1) pigeons and nof_holes holes.

class examples.genhard.**Parity**(*size*, *topv=0*, *verb=False*)
Parity principle formula.

examples.genhard.**parse_options**()
Parses command-line options:

examples.genhard.**usage**()
Prints usage message.

1.2.3 LBX-like MCS enumerator (pysat.examples.lbx)

class examples.lbx.**LBX**(*formula*, *use_cld=False*, *solver_name='m22'*, *use_timer=False*)
LBX-like algorithm for computing MCSes.

block(*mcs*)
Block a (previously computed) MCS.

compute()
Compute and return one solution.

delete()
Explicit destructor.

do_cld_check(*cld*)
Do clause D check.

enumerate()
Enumerate all MCSes and report them one by one.

oracle_time()
Report the total SAT solving time.

class examples.lbx.**LBXPlus**(*formula*, *use_cld=False*, *use_timer=False*)
Algorithm LBX for CNF+/WCNF+ formulas.

block(*mcs*)
Block a (previously computed) MCS.

compute()
Compute and return one solution.

delete()
Explicit destructor.

do_cld_check(*cld*)
Do clause D check.

enumerate()
Enumerate all MCSes and report them one by one.

oracle_time()
Report the total SAT solving time.

examples.lbx.**parse_options**()
Parses command-line options.

examples.lbx.**usage**()
Prints help message.

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

class examples.mcsls.**MCSls**(*formula*, *use_cld=False*, *solver_name='m22'*, *use_timer=False*)
Algorithm LS of MCSls augmented with D calls.

block(*mcs*)
Block a (previously computed) MCS.

compute()
Compute and return one solution.

delete()
Explicit destructor.

do_cld_check(*cld*)
Do clause D check.

enumerate()
Enumerate all MCSes and report them one by one.

oracle_time()
Report the total SAT solving time.

class examples.mcsls.**MCSlsPlus**(*formula*, *use_cld=False*, *use_timer=False*)
Algorithm LS of MCSls for CNF+/WCNF+ formulas.

block(*mcs*)
Block a (previously computed) MCS.

compute()
Compute and return one solution.

delete()
Explicit destructor.

do_cld_check(*cld*)
Do clause D check.

enumerate()
Enumerate all MCSes and report them one by one.

oracle_time()
Report the total SAT solving time.

examples.mcsls.**parse_options**()
Parses command-line options.

examples.mcsls.**usage**()
Prints help message.

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

class examples.musx.**MUSX**(*formula*, *solver='m22'*, *verbosity=1*)
MUS eXctractor using the deletion based algorithm.

compute()
Compute and return a solution.

delete()
Explicit destructor.

oracle_time()
Report the total SAT solving time.

examples.musx.**parse_options**()
Parses command-line option

examples.musx.**usage**()
Prints usage message.

1.2.6 RC2 MaxSAT solver (pysat.examples.rc2)

class examples.rc2.**RC2**(*formula*, *solver='g3'*, *adapt=False*, *exhaust=False*, *incr=False*, *minz=False*, *trim=0*, *verbose=0*)
MaxSAT algorithm based on relaxable cardinality constraints (RC2/OLL).

adapt_am1()
Try to detect atmost1 constraints involving soft literals.

compute()
Compute and return a solution.

compute_()
Compute a MaxSAT solution with RC2.

create_sum(*bound=1*)
Create a totalizer object encoding a new cardinality constraint. For Minicard, native atmostb constraints is used instead.

delete()
Explicit destructor.

enumerate()
Enumerate MaxSAT solutions (from best to worst).

exhaust_core(*tobj*)
Exhaust core by increasing its bound as much as possible.

filter_assumps()
Filter out both unnecessary selectors and sums.

get_core()
Extract unsatisfiable core.

init(*formula*, *incr=False*)
Initialize the SAT solver.

minimize_core()
Try to minimize a core and compute an approximation of an MUS. Simple deletion-based MUS extraction.

oracle_time()
Report the total SAT solving time.

process_am1(*am1*)
Process an atmost1 relation detected (treat as a core).

process_core()
Deal with a core found in the main loop.

process_sels()
Process soft clause selectors participating in a new core.

process_sums()
Process cardinality sums participating in a new core.

set_bound(*tobj*, *rhs*)
Set a bound for a given totalizer object.

trim_core()
Trim unsatisfiable core at most a given number of times.

update_sum(*assump*)
Increase the bound for a given totalizer object.

class examples.rc2.**RC2Stratified**(*formula*, *solver='g3'*, *adapt=False*, *exhaust=False*, *incr=False*, *minz=False*, *trim=0*, *verbose=0*)
Stratified version of RC2 exploiting Boolean lexicographic optimization and stratification.

activate_clauses(*beg*)
Add more soft clauses to the problem.

adapt_am1()
Try to detect atmost1 constraints involving soft literals.

compute()
Exploit Boolean lexicographic optimization when solving.

compute_()
Compute a MaxSAT solution with RC2.

create_sum(*bound=1*)
Create a totalizer object encoding a new cardinality constraint. For Minicard, native atmostb constraints is used instead.

delete()
Explicit destructor.

enumerate()
Enumerate MaxSAT solutions (from best to worst).

exhaust_core(*tobj*)
Exhaust core by increasing its bound as much as possible.

filter_assumps()
Filter out both unnecessary selectors and sums.

finish_level()
Postprocess the current optimization level: harden clauses depending on their remaining weights.

get_core()
Extract unsatisfiable core.

init(*formula*, *incr=False*)
Initialize the SAT solver.

init_wstr()
Compute and initialize optimization levels for BLO.

minimize_core()
Try to minimize a core and compute an approximation of an MUS. Simple deletion-based MUS extraction.

next_level()
Get next weight to use in BLO.

oracle_time()
Report the total SAT solving time.

process_am1(*am1*)
Process an atmost1 relation detected (treat as a core).

process_core()
Deal with a core found in the main loop.

process_sels()
Process soft clause selectors participating in a new core.

process_sums()
Process cardinality sums participating in a new core.

set_bound(*tobj*, *rhs*)
Set a bound for a given totalizer object.

trim_core()
Trim unsatisfiable core at most a given number of times.

update_sum(*assump*)
Increase the bound for a given totalizer object.

examples.rc2.**parse_options**()
Parses command-line option

examples.rc2.**usage**()
Prints usage message.

PYTHON MODULE INDEX

e

p

INDEX

G

I

K

L

M

N

O

P

R

S

T

U

W