# 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 SAT solvers’ API (pysat.solvers) ......................... 23
  1.2 Supplementary examples package (partially documented) .... 32
    1.2.1 Fu&Malik MaxSAT algorithm (pysat.examples.fm) ...... 32
    1.2.2 Hard formula generator (pysat.examples.genhard) ....... 35
    1.2.3 Minimum/minimal hitting set solver (pysat.examples.hitman) .... 38
    1.2.4 LBX-like MCS enumerator (pysat.examples.lbx) ........... 41
    1.2.5 LSU algorithm for MaxSAT (pysat.examples.lsu) .......... 42
    1.2.6 CLD-like MCS enumerator (pysat.examples.mcsls) ......... 42
    1.2.7 A deletion-based MUS extractor (pysat.examples.musx) ... 43
    1.2.8 RC2 MaxSAT solver (pysat.examples.rc2) .................. 45

Python Module Index .................................................. 49

Index .............................................................................. 51
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.
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. |

Module description

This module provides access to various cardinality constraint encodings to formulas in conjunctive normal form (CNF). These include pairwise, bitwise, ladder/regular, sequential counters, sorting, and cardinality networks, totalizer, modulo totalizer, and modulo totalizer for $k$-cardinality, 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

---

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
9 Toru Ogawa, Yangyang Liu, Ryozi 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\(^\text{11}\) 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:

```python
>>> 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* 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:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pairwise</td>
<td>0</td>
</tr>
<tr>
<td>seqcounter</td>
<td>1</td>
</tr>
<tr>
<td>sortnetwrk</td>
<td>2</td>
</tr>
<tr>
<td>cardnetwrk</td>
<td>3</td>
</tr>
<tr>
<td>bitwise</td>
<td>4</td>
</tr>
<tr>
<td>ladder</td>
<td>5</td>
</tr>
<tr>
<td>totalizer</td>
<td>6</td>
</tr>
<tr>
<td>mtotalizer</td>
<td>7</td>
</tr>
<tr>
<td>kmtotalizer</td>
<td>8</td>
</tr>
<tr>
<td>native</td>
<td>9</td>
</tr>
</tbody>
</table>

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 \([\text{lits}, k]\), where \text{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\(^1\). 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:

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

```python
>>> from pysat.card import ITotalizer
>>> with ITotalizer(lits=[1, 2, 3], ubound=1) as t:
...   print t.cnf.clauses
...   print t.rhs
...   print t.delete()

[[-2, 4], [-1, 4], [-1, -2, 5], [-4, 6], [-5, 7], [-3, 6], [-3, -4, 7]]
[6, 7]
```

delete()
Destroy 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()` – additional literals to be included in the sum.
- `ubound()` – a new upper bound.
- `top_id()` – 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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)
... 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.

1.1.2 Boolean formula manipulation (pysat.formula)

List of classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDPool</td>
<td>A simple manager of variable IDs.</td>
</tr>
<tr>
<td>CNF</td>
<td>Class for manipulating CNF formulas.</td>
</tr>
<tr>
<td>CNFPlus</td>
<td>CNF formulas augmented with native cardinality</td>
</tr>
<tr>
<td></td>
<td>constraints.</td>
</tr>
<tr>
<td>WCNF</td>
<td>Class for manipulating partial (weighted) CNF formulas.</td>
</tr>
<tr>
<td>WCNFPlus</td>
<td>WCNF formulas augmented with native cardinality</td>
</tr>
<tr>
<td></td>
<td>constraints.</td>
</tr>
</tbody>
</table>

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 $(\neg x_2 \lor x_{19} \lor x_{46})$ 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 \((\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3)\).

```python
>>> 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:

```python
>>> 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.

```python
>>> 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 \((\neg x_2 \lor x_19 \lor x_46)\) 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 \((\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3)\) 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:

```python
>>> 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
-1 2 0
-2 3 0
-3 0
')
>>> 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. \(\mathcal{F} = \mathcal{H} \land \mathcal{S}\). 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 \(\mathcal{F} = \mathcal{H} \land \mathcal{S}\) 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`.

```python
>>> 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 $1 + \sum_{c_i \in S} w_i$. Top weight of a formula can be accessed through variable `topw`.

```python
>>> 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`).

```python
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(1, 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.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 method copies a list of clauses into a CNF object.

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

Example:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:**

```python
>>> from pysat.formula import CNF
>>> cnf1 = CNF()
>>> cnf1.from_string(='p cnf 2 2

-1 2 0

1 -2 0')
>>> print cnf1.clauses
[[-1, 2], [1, -2]]
>>> cnf2 = CNF(from_string='p cnf 3 3

-1 2 0

-2 3 0

-3 0')
>>> 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. 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**.

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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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.CNFPus` (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 `CNFPus` supports native cardinality constraints of MiniCard.

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

```plaintext
p cnf+ 7 3
1 -2 3 5 -7 <= 3
```

(continues on next page)
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:

```python
>>> from pysat.formula import CNFPlus
>>> cnf = CNFPlus(from_string='p cnf+ 7 3
1 -2 3 5 -7 <= 3
4 5 6 -7 >= 2
3 5 7 0
7
')
>>> 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.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 by 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:

```python
>>> 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:**

```python
>>> 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.
```python
>>> 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:**

```python
>>> 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:**

```python
>>> 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 auto-
matically 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
2 -1 2 0
1 1 -2 0')
>>> print cnf1.hard
```

(continues on next page)
[[[-1, 2]]
>>> print cnf1.soft
[[1, 2]]
>>> cnf2 = WCNF(from_string='p wcnf 3 3 2
2 -1 2 0
2 -2 3 0
1 -3 0')
>>> 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:

```python
>>> 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:

```python
>>> 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 \textit{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:

```python
>>> 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:

```python
>>> from pysat.formula import WCNFPlus
>>> cnf = WCNFPlus(from_string='p wcnf+ 7 3 10
10 1 -2 3 5 -7 <= 3
10 4 5 6 -7 >\n5 3 5 7 0')

>>> 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.*

```python
>>> 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:

```python
>>> 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:

```python
>>> from pysat.formula import WCNFPlus
>>> cnf = WCNFPlus()
... (continues on next page)```
```python
>>> # 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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolverNames</td>
<td>This class serves to determine the solver requested by a user given a string name.</td>
</tr>
<tr>
<td>Solver</td>
<td>Main class for creating and manipulating a SAT solver.</td>
</tr>
<tr>
<td>Glucose3</td>
<td>Glucose 3 SAT solver.</td>
</tr>
<tr>
<td>Glucose4</td>
<td>Glucose 4.1 SAT solver.</td>
</tr>
<tr>
<td>Lingeling</td>
<td>Lingeling SAT solver.</td>
</tr>
<tr>
<td>Minicard</td>
<td>Minicard SAT solver.</td>
</tr>
<tr>
<td>Minisat22</td>
<td>MiniSat 2.2 SAT solver.</td>
</tr>
<tr>
<td>MinisatGH</td>
<td>MiniSat SAT solver (version from github).</td>
</tr>
</tbody>
</table>

### 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-like¹ incremental² 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:

```python
>>> 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:

---


---

### 1.1. Core PySAT modules
• 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
• 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\textsuperscript{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 $\mathcal{F} \land (a_{i1} \land \ldots \land a_{i1+j1})$ and $\mathcal{F} \land (a_{i2} \land \ldots \land a_{i2+j2})$, where every $a_{ik}$ 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:

```python
>>> 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’.

**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:

```python
>>> 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):

```python
>>> 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.
• **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:

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

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

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

**Parameters**

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

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

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

A usage example is the following:

```python
>>> 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.

---

26 Chapter 1. API documentation
Return type: bool if no_return is set to False.

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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 arises 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:

```python
>>> 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, 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:

```python
>>> from pysat.solvers import Glucose3

>>> g = Glucose3(bootstrap_with=[[1, 2]])

>>> # the formula has 3 models: [-1, 2], [1, -2], [1, 2]

>>> g.set_phases(literals=[1, 2])

>>> g.solve()
True

>>> g.get_model()
[1, 2]

>>> g.delete()
```

**solve** *(assumptions=[])*

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

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

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

**Return type** Boolean or None.

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

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

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

Note that only MiniSat-like solvers support this functionality (e.g. 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

(continues on next page)
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:

```python
>>> 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

>>> 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.

```python
glucose3 = ('g3', 'g30', 'glucose3', 'glucose30')
glucose4 = ('g4', 'g41', 'glucose4', 'glucose41')
ingeling = ('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 (partially documented)

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 \textit{weighted partial} formulas.

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

```bash
$ xzcat formula.wcnf.xz
p wcnf 3 6 4
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
```

Alternatively, the algorithm can be accessed and invoked through the standard \texttt{import} interface of Python, e.g.

```python
>>> 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

```python
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

\(^1\) Zhaohui Fu, Sharad Malik. \textit{On Solving the Partial MAX-SAT Problem}. SAT 2006. pp. 252-265

1.2. Supplementary examples package (partially documented)
is core-guided, i.e. it solves maximum satisfiability with a series of unsatisfiability oracle calls, each produc-
ing 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 \texttt{FM} objects receives a target \texttt{WCNF} MaxSAT formula, an identifier of the cardinality en-
coding to use, a SAT solver name, and a verbosity level. Note that the algorithm uses the pairwise (see
\texttt{EncType}) cardinality encoding by default, while the default SAT solver is MiniSat22 (referred to as 'm22',
see \texttt{SolverNames} for details). The default verbosity level is 1.

**Parameters**

- \texttt{formula (WCNF)} – input MaxSAT formula
- \texttt{enc (int)} – cardinality encoding to use
- \texttt{solver (str)} – name of SAT solver
- \texttt{verbose (int)} – verbosity level

\texttt{_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 \texttt{self.model}) and returns. Otherwise, a new unsatisfiable core of the formula
is extracted and processed (see \texttt{treat_core()}), and the algorithm proceeds.

\texttt{compute()}

Compute a MaxSAT solution. First, the method checks whether or not the set of hard clauses is satisfiable.
If not, the method returns \texttt{False}. Otherwise, add soft clauses to the oracle and call the MaxSAT algorithm
(see \texttt{_compute()}).

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

\texttt{delete()}

Explicit destructor of the internal SAT oracle.

\texttt{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 \texttt{reinit()}). An input parameter \texttt{with\_soft} (False
by default) regulates whether or not the formula’s soft clauses are copied to the oracle.

**Parameters** \texttt{with\_soft (bool)} – copy formula’s soft clauses to the oracle or not

\texttt{oracle\_time()}

Method for calculating and reporting the total SAT solving time.

\texttt{reinit()}

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

\texttt{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 \texttt{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. \(c \leftarrow c \lor r\), 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, \( \sum_{r \in \phi} r \leq 1 \), where \( \phi \) 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 \( \text{minw} \) value computed in \text{treat_core}(). Each clause \((c \lor \neg s, w)\), s.t. \( w > \text{minw} \) and \( s \) is its selector literal, is split into clauses (1) clause \((c \lor \neg s, \text{minw})\) and (2) a residual clause \((c \lor \neg s', w - \text{minw})\). 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 \text{pysat.solvers.Solver.get_core()} method, which returns a subset of the selector literals deemed responsible for unsatisfiability.

After the core is extracted, its \textit{minimum weight} \( \text{minw} \) 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 \( \text{minw} \).

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

### 1.2.2 Hard formula generator (**pysat.examples.genhard**)

#### List of classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB</td>
<td>Mutilated chessboard principle (CB).</td>
</tr>
<tr>
<td>GT</td>
<td>Generator of ordering (or greater than, GT) principle formulas.</td>
</tr>
<tr>
<td>PAR</td>
<td>Generator of the parity principle (PAR) formulas.</td>
</tr>
<tr>
<td>PHP</td>
<td>Generator of ( k ) pigeonhole principle (( k )-PHP) formulas.</td>
</tr>
</tbody>
</table>

**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: \textit{pigeonhole principle} (\( \text{PHP} \)), \textit{ordering} (\textit{greater-than}) principle (\( \text{GT} \)), \textit{mutilated chessboard principle} (\( \text{CB} \)), and \textit{parity principle} (\( \text{PAR} \)).

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

---

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

```python
>>> 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**

```python
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 \( 2n \times 2n \) 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) \((\neg x_{ji} \lor \neg x_{ki})\) for every \(i, j \neq k\) meaning that no more than one adjacent cell can be paired with the current one; and (2) \((\lor_{j \in \text{Adj}(i)} x_{ij})\) \(\forall i\) 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 \texttt{AtMost1} constraints are formulated only for the white cells while the \texttt{AtLeast1} constraints are formulated only for the black cells. Depending on the value of parameter \texttt{exhaustive} the encoder applies the complete or partial formulation of the problem.

Mutilated chessboard principle is known to be hard for resolution\(^5\).

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

Returns object of class \texttt{pysat.formula.CNF}.

\texttt{class examples.genhard.GT(size, topv=0, verb=False)}

Generator of ordering (or \texttt{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 \((\neg x_{ij} \lor \neg x_{jk})\) and \((\neg x_{ij} \lor \neg x_{jk} \lor x_{ik})\) 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 \((\lor_{j \neq i} x_{ji})\) 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
- \texttt{size (int)} – number of elements \((n)\)
- \texttt{topv (int)} – current top variable identifier
- \texttt{verb (bool)} – defines whether or not the encoder is verbose

Returns object of class \texttt{pysat.formula.CNF}.

\texttt{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 \((\frac{2n+1}{2})\) variables \(x_{ij}, i \neq j\). If variable \(x_{ij}\) is \texttt{true}, then there is an edge between nodes \(i\) and \(j\). The formula consists of the following clauses: \((\lor_{j \neq i} x_{ij})\) for every \(i \in [2n + 1]\), and \((\neg x_{ij} \lor \neg x_{ik})\) for all distinct \(i, j, k \in [2n + 1]\).

The parity principle is known to be hard for resolution\(^4\).

Parameters
- \texttt{size (int)} – problem size \((n)\)
- \texttt{topv (int)} – current top variable identifier
- \texttt{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: \(\bigwedge_{i=1}^{k \cdot m + 1} \text{AtLeast1}(x_{i1}, \ldots, x_{im}) \land \bigwedge_{j=1}^{m} \text{AtMost}k(x_{1j}, \ldots, x_{k \cdot m + 1, j})\). 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

<table>
<thead>
<tr>
<th>Hitman</th>
</tr>
</thead>
</table>

A cardinality-/subset-minimal hitting set enumerator.

Module description

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

The minimal hitting set problem is trivially formulated as a MaxSAT formula in WCNF, as follows. Assume \(E = \{e_1, \ldots, e_n\}\) to be a universe of elements. Also assume there are \(k\) sets to hit: \(s_i = \{e_{i,1}, \ldots, e_{i,k}\}\) s.t. \(e_{i,l} \in E\). Every set \(s_i = \{e_{i,1}, \ldots, e_{i,k}\}\) is translated into a hard clause \((e_{i,1} \lor \ldots \lor e_{i,k})\). 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\(^2\) 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.

```python
>>> 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:

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

Finally, implicit hitting set enumeration can be used in practical problem solving. As an example, let us show the basic flow of a MaxHS-like\(^\text{7}\) algorithm for MaxSAT:

```python
>>> 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"
```

\(^7\) Jessica Davies, Fahiem Bacchus. Solving MAXSAT by Solving a Sequence of Simpler SAT Instances. CP 2011. pp. 225-239

1.2. Supplementary examples package (partially documented)
```python
>>> 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
- **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()*.

**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)

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

LBX-like algorithm for computing MCSes.

add_clause(clause, soft=False)

Add new hard or soft clause (may be needed in MCS enumeration).

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.

add_clause(clause, soft=False)

Add new hard or soft clause (may be needed in MCS enumeration).

block(mcs)

Block a (previously computed) MCS.

compute()

Compute and return one solution.

delete()

Explicit destructor.

1.2. Supplementary examples package (partially documented)
do_cld_check(cld)
    Do clause D check.

do_enum(cld)
    Enumerate all MCSes and report them one by one.

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

oracle_time()
    Report the total SAT solving time.

elements.lbx.parse_options()
    Parses command-line options.

elements.lbx.usage()
    Prints help message.

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

class examples.lsu.LSU(formula, solver='g4', verbose=0)
    Linear Sat-Unsat algorithm for MaxSAT. Only supports unweighted problems for now.

    get_model()
        Returns the internal model.

    oracle_time()
        Report the total SAT solving time.

    solve()
        Computes a solution to the MaxSAT problem. Returns True if a solution exists, False if the hard
        formula is unsatisfiable.

    examples.lsu.parse_formula(file)
        Parse and return MaxSAT formula.

    examples.lsu.parse_options()
        Parses command-line options.

    examples.lsu.print_usage()
        Prints usage message.

1.2.6 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.

    add_clause(clause, soft=False)
        Add new hard or soft clause (may be needed in MCS enumeration).

    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.

callable orc래le_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.

callable add_clause(clause, soft=False)
   Add new hard or soft clause (may be needed in MCS enumeration).

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

callable compute()
   Compute and return one solution.

callable delete()
   Explicit destructor.

callable do_cld_check(cld)
   Do clause D check.

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

callable oracle_time()
   Report the total SAT solving time.

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

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

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 for extracting a minimal unsatisfiable subset (MUS) of a given (unsatisfiable) 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:

```python
# oracle: SAT solver (initialized)
# assump: full set of assumptions
i = 0
while i < len(assump):
    # (continues on next page)
```

---


1.2. Supplementary examples package (partially documented)
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
```

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

```python
>>> 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. \((c) \leftarrow (c \lor \neg s)\). 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.

### 1.2.8 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.

**add_clause** (*clause, weight=None*)
Add a new clause (needed for incremental MaxSAT solving).

**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 \( (aml) \)
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.

add_clause \( (clause, weight=None) \)
Add a new clause (needed for incremental MaxSAT solving).

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.

toolbox.parse_options()
    Parses command-line option

toolbox.usage()
    Prints usage message.
e
examples.fm, 32
examples.genhard, 35
examples.hitman, 38
examples.lbx, 41
examples.lsu, 42
examples.mcsls, 42
examples.musx, 43
examples.rc2, 45

p
pysat.card, 3
pysat.formula, 8
pysat.solvers, 23
Symbols

_compute() (examples.fm.FM method), 34
_compute() (examples.musx.MUSX method), 45

activate_clauses() (examples.rc2.RC2Stratified method), 46
adapt_am1() (examples.rc2.RC2 method), 45
adapt_am1() (examples.rc2.RC2Stratified method), 46
add_atmost() (pysat.solvers.Solver method), 26
add_clause() (examples.lbx.LBX method), 41
add_clause() (examples.lbx.LBXPlus method), 41
add_clause() (examples.mcsls.MCSls method), 42
add_clause() (examples.mcsls.MCSlsPlus method), 43
add_clause() (examples.rc2.RC2 method), 45
add_clause() (examples.rc2.RC2Stratified method), 46
add_clause() (pysat.solvers.Solver method), 26
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), 26
atleast() (pysat.card.CardEnc class method), 4
atmost() (pysat.card.CardEnc class method), 4

block() (examples.hitman.Hitman method), 40
block() (examples.lbx.LBX method), 41
block() (examples.lbx.LBXPlus method), 41
block() (examples.mcsls.MCSls method), 42
block() (examples.mcsls.MCSlsPlus method), 43

CardEnc (class in pysat.card), 4
CB (class in examples.genhard), 36
CNF (class in pysat.formula), 11
CNFPlus (class in pysat.formula), 14
compute() (examples.fm.FM method), 34
compute() (examples.lbx.LBX method), 41
compute() (examples.lbx.LBXPlus method), 41
compute() (examples.mcsls.MCSls method), 42
compute() (examples.mcsls.MCSlsPlus method), 43
compute() (examples.musx.MUSX method), 45
compute() (examples.rc2.RC2 method), 45
compute() (examples.rc2.RC2Stratified method), 46
conf_budget() (pysat.solvers.Solver method), 27
copy() (pysat.formula.CNF method), 11
copy() (pysat.formula.WCNF method), 18
create_sum() (examples.rc2.RC2 method), 45
create_sum() (examples.rc2.RC2Stratified method), 46

delete() (examples.fm.FM method), 34
delete() (examples.hitman.Hitman method), 40
delete() (examples.lbx.LBX method), 41
delete() (examples.lbx.LBXPlus method), 41
delete() (examples.mcsls.MCSls method), 42
delete() (examples.mcsls.MCSlsPlus method), 43
delete() (examples.musx.MUSX method), 45
delete() (examples.rc2.RC2 method), 45
delete() (examples.rc2.RC2Stratified method), 46
delete() (pysat.card.ITotalizer method), 6
do_cld_check() (examples.lbx.LBX method), 41
do_cld_check() (examples.lbx.LBXPlus method), 41
do_cld_check() (examples.mcsls.MCSls method), 42
do_cld_check() (examples.mcsls.MCSlsPlus method), 43

EncType (class in pysat.card), 5
enumerate() (pysat.solvers.Solver method), 27
enumerate() (examples.hitman.Hitman method), 40
enumerate() (examples.lbx.LBX method), 41
enumerate() (examples.lbx.LBXPlus method), 42
enumerate() (examples.mcsls.MCSls method), 42
enumerate() (examples.mcsls.MCSlsPlus method), 43
equals() (pysat.card.CardEnc class method), 4
examples.fm (module), 32
examples.genhard (module), 35
examples.hitman (module), 38
examples.lbx (module), 41
examples.lsu (module), 42
examples.mcsls (module), 42
examples.musx (module), 43
examples.rc2 (module), 45
e exhaust_core() (examples.rc2.RC2 method), 45
e exhaust_core() (examples.rc2.RC2Stratified method), 47
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), 46
filter_assumps() (examples.rc2.RC2Stratified method), 47
finish_level() (examples.rc2.RC2Stratified method), 47
FM (class in examples.fm), 33
from_clauses() (pysat.formula.CNF method), 11
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.WCNF method), 19
from_fp() (pysat.formula.WCNFPlus method), 15
from_string() (pysat.formula.CNF method), 12
from_string() (pysat.formula.WCNF method), 19
G
get() (examples.hitman.Hitman method), 40
get_core() (examples.rc2.RC2 method), 46
get_core() (examples.rc2.RC2Stratified method), 47
get_model() (examples.lsu. LSU method), 42
get_model() (pysat.solvers.Solver method), 28
get_proof() (pysat.solvers.Solver method), 28
get_status() (pysat.solvers.Solver method), 28
GT (class in examples.genhard), 37
H
hit() (examples.hitman.Hitman method), 41
Hitman (class in examples.hitman), 40
I
id() (pysat.formula.IDPool method), 16
IDPool (class in pysat.formula), 16
increase() (pysat.card.ITotalizer method), 6
init() (examples.fm.FM method), 34
init() (examples.hitman.Hitman method), 41
init() (examples.rc2.RC2 method), 46
init() (examples.rc2.RC2Stratified method), 47
init_wstr() (examples.rc2.RC2Stratified method), 47
ITotalizer (class in pysat.card), 5
L
LBX (class in examples.lbx), 41
LBXPlus (class in examples.lbx), 41
LSU (class in examples.lsu), 42
M
MCSls (class in examples.mcsls), 42
MCSlsPlus (class in examples.mcsls), 43
merge_with() (pysat.card.ITotalizer method), 7
minimize_core() (examples.rc2.RC2 method), 46
minimize_core() (examples.rc2.RC2Stratified method), 47
MUSX (class in examples.musx), 44
N
negate() (pysat.formula.CNF method), 13
new() (pysat.card.ITotalizer method), 8
new() (pysat.solvers.Solver method), 28
next_level() (examples.rc2.RC2Stratified method), 47
nof_clauses() (pysat.solvers.Solver method), 29
nof_vars() (pysat.solvers.Solver method), 29
NoSuchEncodingError, 8
NoSuchSolverError, 24
O
obj() (pysat.formula.IDPool method), 17
occupy() (pysat.formula.IDPool method), 17
oracle_time() (examples.fm.FM method), 34
oracle_time() (examples.lbx.LBX method), 41
oracle_time() (examples.lbx.LBXPlus method), 42
oracle_time() (examples.lsu.LSU method), 42
oracle_time() (examples.mcsls.MCSls method), 43
oracle_time() (examples.mcsls.MCSlsPlus method), 43
oracle_time() (examples.musx.MUSX method), 45
oracle_time() (examples.rc2.RC2 method), 46
oracle_time() (examples.rc2.RC2Stratified method), 47
P
PAR (class in examples.genhard), 37
parse_formula() (in module examples.lsu), 42
parse_options() (in module examples.lbx), 42
parse_options() (in module examples.lsu), 42
parse_options() (in module examples.mcsls), 43
parse_options() (in module examples.rc2), 47
PHP (class in examples.genhard), 38
print_usage() (in module examples.lsu), 42
process_am1() (examples.rc2.RC2 method), 46
process_am1() (examples.rc2.RC2Stratified method), 47
process_core() (examples.rc2.RC2 method), 46
process_core() (examples.rc2.RC2Stratified method), 47
process_sels() (examples.rc2.RC2 method), 46
process_sels() (examples.rc2.RC2Stratified method), 47
process_sums() (examples.rc2.RC2 method), 46
process_sums() (examples.rc2.RC2Stratified method), 47
prop_budget() (pysat.solvers.Solver method), 29
propagate() (pysat.solvers.Solver method), 29
pysat.card (module), 3
pysat.formula (module), 8
pysat.solvers (module), 23

R
RC2 (class in examples.rc2), 45
RC2Stratified (class in examples.rc2), 46
reinit() (examples.fm.FM method), 34
relax_core() (examples.fm.FM method), 34
remove_unit_core() (examples.fm.FM method), 35
restart() (pysat.formula.IDPool method), 17

S
set_bound() (examples.rc2.RC2 method), 46
set_bound() (examples.rc2.RC2Stratified method), 47
set_phases() (pysat.solvers.Solver method), 30
solve() (examples.lsu.LSU method), 42
solve() (pysat.solvers.Solver method), 30
solve_limited() (pysat.solvers.Solver method), 31
Solver (class in pysat.solvers), 24
SolverNames (class in pysat.solvers), 32
split_core() (examples.fm.FM method), 35

T
time() (pysat.solvers.Solver method), 31
time_accum() (pysat.solvers.Solver method), 32
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), 35
trim_core() (examples.rc2.RC2 method), 46
trim_core() (examples.rc2.RC2Stratified method), 47

U
unweighed() (pysat.formula.WCNF method), 20
update_sum() (examples.rc2.RC2 method), 46
update_sum() (examples.rc2.RC2Stratified method), 47
usage() (in module examples.lbx), 42
usage() (in module examples.mcsls), 43
usage() (in module examples.rc2), 47

W
WCNF (class in pysat.formula), 17
WCNFPlus (class in pysat.formula), 21
weighted() (pysat.formula.CNF method), 14