PySAT Documentation

Release 1.8.dev11

Alexey Ignatiev, Joao Marques-Silva, Antonio Morgado

Apr 18, 2024

CONTENTS

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

CHAPTER

ONE

API DOCUMENTATION

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

A supplementary sixth module `examples` presents a list of scripts, which are supposed to demonstrate how the toolkit can be used for practical problem solving. The module includes a formula generator, several MaxSAT solvers including an award-winning RC2, a few (S)MUS extractors and enumerators as well as MCS enumerators, among other scripts.

Finally, an additional seventh module `allies` brought by version *0.1.8.dev3* is meant to provide access to a number of third-party tools important for practical SAT-based problem solving.

1.1 Core PySAT modules

1.1.1 Cardinality encodings (`pysat.card`)

List of classes

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

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

Module description

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

A cardinality constraint is a constraint of the form: $\sum_{i=1}^{n} x_i \leq k$. Cardinality constraints are ubiquitous in practical problem formulations. Note that the implementation of the pairwise, bitwise, and ladder encodings can only deal with 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*[7] encoding is supported. Iterative totalizer can be accessed with the use of the *ITotalizer* class.

Module details

class pysat.card.**CardEnc**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Parameters

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

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

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

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

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

Raises

- **CardEnc.NoSuchEncodingError** – if encoding does not exist.
- **ValueError** – if bound is meaningless for encoding.

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

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

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

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

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

class pysat.card.**EncType**

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

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

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

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

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

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

The constructor of the class object takes 3 default arguments.

Parameters

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

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

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

Possible usage of the class is shown below:

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

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

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

delete()

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

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

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

Parameters

- **lits** (*iterable(int)*) – additional literals to be included in the sum.

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

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

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

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

The method can be used in the following way:

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

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

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

Parameters

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

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

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

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

The method can be used in the following way:

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

(continues on next page)

(continued from previous page)

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

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

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

Parameters

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

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

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

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

The method can be used in the following way:

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

(continues on next page)

(continued from previous page)

```
...
...     t2.delete()
```

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

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

exception pysat.card.**NoSuchEncodingError**

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

with_traceback()

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

exception pysat.card.**UnsupportedBound**

This exception is raised when creating a pairwise, or bitwise, or ladder encoding of AtMostK, AtLeastK, or EqualsK with the bound different from `1` and `N - 1`.

with_traceback()

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

1.1.2 Boolean formula manipulation (`pysat.formula`)

List of classes

IDPool	A simple manager of variable IDs.
Formula	Abstract formula class.
Atom	Atomic formula, i.e. a variable or constant.
And	Conjunction.
Or	Disjunction.
Neg	Negation.
Implies	Implication.
Equals	Equivalence.
XOr	Exclusive disjunction.
ITE	If-then-else operator.
CNF	Class for manipulating CNF formulas.
CNFPlus	CNF formulas augmented with *native* cardinality constraints.
WCNF	Class for manipulating partial (weighted) CNF formulas.
WCNFPlus	WCNF formulas augmented with *native* cardinality constraints.

Module description

This module is designed to facilitate fast and easy PySAT-development by providing a simple way to manipulate formulas in PySAT. The toolkit implements several facilities to manupulate Boolean formulas. Namely, one can opt for creating arbitrary non-clausal formulas suitable for simple problem encodings requiring no or little knowledge about the process of logical encoding. However, the main and most often used kind of formula in PySAT is represented by the *CNF* class, which can be used to define a formula in conjunctive normal form (CNF).

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
>>>
>>> f3 = CNF(from_string='p cnf 3 3\n-1 2 0\n-2 3 0\n-3 0\n')
>>> print(f3.clauses)
[[-1, 2], [-2, 3], [-3]]
>>> print(f3.nv)
3
```

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

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

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

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

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

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

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

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

Apart from the aforementioned variants of (W)CNF formulas, the module now offers a few additional classes for managing non-clausal Boolean formulas. Namely, a user may create complex formulas using variables (atomic formulas

implemented as objects of class *Atom*), and the following logic connectives: *And*, *Or*, *Neg*, *Implies*, *Equals*, *XOr*, and *ITE*. (All of these classes inherit from the base class *Formula*.) Arbitrary combinations of these logic connectives are allowed. Importantly, the module provides seamless integration of *CNF* and various subclasses of *Formula* with the possibility to clausify these on demand.

```
>>> from pysat.formula import *
>>> from pysat.solvers import Solver

# creating two formulas: CNF and XOr
>>> cnf = CNF(from_clauses=[[-1, 2], [-2, 3]])
>>> xor = Atom(1) ^ Atom(2) ^ Atom(4)

# passing the conjunction of these to the solver
>>> with Solver(bootstrap_with=xor & cnf) as solver:
...     # setting Atom(3) to false results in only one model
...     for model in solver.enum_models(assumptions=Formula.literals([~Atom(3)])):
...         print(Formula.formulas(model, atoms_only=True))  # translating the model back␣
↪to atoms
>>>
[Neg(Atom(1)), Neg(Atom(2)), Neg(Atom(3)), Atom(4)]
```

Note: Combining CNF formulas with non-CNF ones will not necessarily result in the best possible encoding of the complex formula. The on-the-fly encoder may introduce variables that a human would avoid using, e.g. if `cnf1` and `cnf2` are *CNF* formulas then `And(cnf1, cnf2)` will introduce auxiliary variables `v1` and `v2` encoding them, respectively (although it is enough to join their sets of clauses).

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

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

# nof_holes is given

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

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

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

Module details

class pysat.formula.**And**(**args*, ***kwargs*)

Conjunction. Given a list of operands (subformulas) f_i, $i \in \{1, \ldots, n\}, n \in \mathbb{N}$, it creates a formula $\bigwedge_{i=1}^{n} f_i$. The list of operands *of size at least 1* should be passed as arguments to the constructor.

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> conj = And(x, y, z)
```

If an additional Boolean keyword argument `merge` is provided set to `True`, the toolkit will try to flatten the current *And* formula merging its *conjuctive* sub-operands into the list of operands. For example, if `And(And(x, y), z, merge=True)` is called, a new Formula object will be created with two operands: `And(x, y)` and `z`, followed by merging `x` and `y` into the list of root-level `And`. This will result in a formula `And(x, y, z)`. Merging sub-operands is enabled by default if bitwise operations are used to create `And` formulas.

Example:

```
>>> from pysat.formula import *
>>> a1 = And(And(Atom('x'), Atom('y')), Atom('z'))
>>> a2 = And(And(Atom('x'), Atom('y')), Atom('z'), merge=True)
>>> a3 = Atom('x') & Atom('y') & Atom('z')
>>>
>>> repr(a1)
"And[And[Atom('x'), Atom('y')], Atom('z')]"
>>> repr(a2)
"And[Atom('x'), Atom('y'), Atom('z')]"
>>> repr(a2)
"And[Atom('x'), Atom('y'), Atom('z')]"
>>>
>>> id(a1) == id(a2)
False
>>>
>>> id(a2) == id(a3)
True  # formulas a2 and a3 refer to the same object
```

Note: If there are two formulas representing the same fact with and without merging enabled, they technically sit in two distinct objects. Although PySAT tries to avoid it, clausification of these two formulas may result in unique (different) auxiliary variables assigned to such two formulas.

---

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the subformulas and creates a new formula.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = x & y & z
```

(continues on next page)

(continued from previous page)

```
>>>
>>> print(a.simplified(assumptions=[y]))
x & z
>>> print(a.simplified(assumptions=[~y]))
F  # False
```

class pysat.formula.**Atom**(**args*, ***kwargs*)

Atomic formula, i.e. a variable or constant. Although we often refer to negated literals as atomic formulas too, they are techically implemented as `Neg(Atom)`.

To create an atomic formula, a user needs to specify an `object` this formula should signify. When it comes to clausifying the formulas this atom is involved in, the atom receives an auxiliary variable assigned to it as a `name`.

Example:

```
>>> from pysat.formula import *
>>> x = Atom('x')
>>> y = Atom(object='y')
>>> # checking x's name
>>> x.name
>>> # None
>>> # right, that's because the atom is not yet clausified
>>> x.clausify()
>>> x.name
1
```

If a given object is a positive integer (negative integers aren't allowed), the integer itself serves as the atom's name, which is assigned in the constructor, i.e. no call to `clausify()` is required.

Example:

```
>>> from pysat.formula import Atom
>>> x, y = Atom(1), Atom(2)
>>> x.name
1
>>> y.name
2
```

Special atoms are reserved for the Boolean constants `True` and `False`. Namely, `Atom(False)` and `Atom(True)` can be accessed through the constants `PYSAT_FALSE` and `PYSAT_TRUE`, respectively.

Example:

```
>>> from pysat.formula import *
>>>
>>> print(PYSAT_TRUE, repr(PYSAT_TRUE))
T Atom(True)
>>>
>>> print(PYSAT_FALSE, repr(PYSAT_FALSE))
F Atom(False)
```

Note: Constant `Atom(True)` is distinguished from variable `Atom(1)` by checking the type of the object (bool vs int).

simplified(*assumptions=[]*)

Checks if the current literal appears in the list of assumptions provided in argument `assumptions`. If it is, the method returns `PYSAT_TRUE`. If the opposite atom is present in `assumptions`, the method returns `PYSAT_FALSE`. Otherwise, it return `self`.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type PYSAT_TRUE, PYSAT_FALSE, or self

class pysat.formula.**CNF**(**args*, ***kwargs*)

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
- **from_aiger** (`aiger.AIG` (see py-aiger package)) – an AIGER circuit to bootstrap the formula with
- **comment_lead** (*list(str)*) – a list of characters leading comment lines
- **by_ref** (*bool*) – flag to indicate how to copy clauses - by reference or deep-copy

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

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

The additional keyword argument `update_vpool` can be set to `True` if the user wants to update for default static pool of variable identifiers stored in class *Formula*.

Parameters

- **clause** (*list(int)*) – a new clause to add
- **update_vpool** (*bool*) – update or not the static vpool

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

copy()

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

Returns an object of class *CNF*.

Example:

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

(continues on next page)

(continued from previous page)

```
>>> print(cnf2.nv)
2
```

extend(*clauses*, *update_vpool=False*)

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.

The additional keyword argument `update_vpool` can be set to `True` if the user wants to update for default static pool of variable identifiers stored in class *Formula*.

Parameters

- **clauses** (*list(list(int))*) – a list of new clauses to add
- **update_vpool** (*bool*) – update or not the static vpool

Example:

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

from_aiger(*aig*, *vpool=None*)

Create a CNF formula by Tseitin-encoding an input AIGER circuit.

Input circuit is expected to be an object of class `aiger.AIG`. Alternatively, it can be specified as an `aiger.BoolExpr`, or an `*.aag` filename, or an AIGER string to parse. (Classes `aiger.AIG` and `aiger.BoolExpr` are defined in the py-aiger package.)

Parameters

- **aig** (`aiger.AIG` (see py-aiger package)) – an input AIGER circuit
- **vpool** (*IDPool*) – pool of variable identifiers (optional)

Example:

```
>>> import aiger
>>> x, y, z = aiger.atom('x'), aiger.atom('y'), aiger.atom('z')
>>> expr = ~(x | y) & z
>>> print(expr.aig)
aag 5 3 0 1 2
2
4
8
10
6 3 5
10 6 8
i0 y
i1 x
i2 z
o0 6c454aea-c9e1-11e9-bbe3-3af9d34370a9
>>>
>>> from pysat.formula import CNF
>>> cnf = CNF(from_aiger=expr.aig)
```

(continues on next page)

(continued from previous page)

```
>>> print(cnf.nv)
5
>>> print(cnf.clauses)
[[3, 2, 4], [-3, -4], [-2, -4], [-4, -1, 5], [4, -5], [1, -5]]
>>> print(['{0} <-> {1}'.format(v, cnf.vpool.obj(v)) for v in cnf.inps])
['3 <-> y', '2 <-> x', '1 <-> z']
>>> print(['{0} <-> {1}'.format(v, cnf.vpool.obj(v)) for v in cnf.outs])
['5 <-> 6c454aea-c9e1-11e9-bbe3-3af9d34370a9']
```

from_clauses(*clauses*, *by_ref=False*)

This methods copies a list of clauses into a CNF object. The optional keyword argument `by_ref`, which is by default set to `False`, signifies whether the clauses should be deep-copied or copied by reference (by default, deep-copying is applied although it is slower).

Parameters

- **clauses** (*list(list(int))*) – a list of clauses
- **by_ref** (*bool*) – a flag to indicate whether to deep-copy the clauses or copy them by reference

Example:

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

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

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

Parameters

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

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

Usage example:

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

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

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

Parameters

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

Usage example:

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

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

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

Parameters

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

Example:

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

negate(*topv=None*)

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

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

```
>>> from pysat.formula import CNF
>>> pos = CNF(from_clauses=[[-1, 2], [3]])
>>> neg = pos.negate()
>>> print(neg.clauses)
[[1, -4], [-2, -4], [-1, 2, 4], [4, -3]]
>>> print(neg.auxvars)
[4]
>>> print(neg.enclits)  # literals encoding the negation of clauses
[4, -3]
```

simplified(*assumptions=[]*)

As any other Formula type, CNF formulas have this method, although intentionally left unimplemented. Raises a `FormulaError` exception.

to_alien(*file_pointer*, *format='opb'*, *comments=None*)

The method can be used to dump a CNF formula into a file pointer in an alien file format, which at this point can either be LP, OPB, or SMT. The file pointer is expected as an argument. Additionally, the target format 'lp', 'opb', or 'smt' may be specified (equal to 'opb' by default). Finally, supplementary comment lines can be specified in the `comments` parameter.

Parameters

- **file_pointer** (*file pointer*) – a file pointer where to store the formula.
- **format** (*str*) – alien file format to use
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

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

to_dimacs()

Return the current state of the object in DIMACS format.

For example, if 'some-file.cnf' contains:

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

Then you can obtain the DIMACS with:

```
>>> from pysat.formula import CNF
>>> cnf = CNF(from_file='some-file.cnf')
>>> print(cnf.to_dimacs())
c Example
p cnf 3 3
-1 2 0
```

(continues on next page)

(continued from previous page)

```
-2 3 0
-3 0
```

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:

```
>>> 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

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

Example:

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

weighted()

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

Returns an object of class *WCNF*.

Example:

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

class pysat.formula.**CNFPlus**(**args*, ***kwargs*)

CNF formulas augmented with *native* cardinality constraints.

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

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

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

Additionally, *CNFPlus* support pseudo-Boolean constraints, i.e. weighted linear constraints by extending the above format. Basically, a pseudo-Boolean constraint needs to specify all the summands as `weight*literal` with the entire constraint being prepended with character `w` as follows:

```
c Example: One cardinality constraint and one PB constraint followed by a clause
p cnf+ 7 3
1 -2 3 5 -7 <= 3
w 1*4 2*5 1*6 3*-7 >= 2
3 5 7 0
```

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

Example:

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

For details on the functionality, see *CNF*.

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

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

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

Parameters

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

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

copy()

This method can be used for creating a copy of a CNFPlus object. It creates another object of the *CNFPlus* class, call the copy function of CNF class and makes use of the *deepcopy* functionality to copy the atmost constraints.

Returns an object of class *CNFPlus*.

Example:

```
>>> cnf1 = CNFPlus()
>>> cnf1.extend([[-1, 2], [1]])
>>> cnf1.append([[1, 2], 1], is_atmost=True)
>>> cnf2 = cnf1.copy()
>>> print(cnf2.clauses)
[[-1, 2], [1]]
>>> print(cnf2.nv)
2
>>> print(cnf2.atmosts)
[[[1, 2], 1]]
```

extend(*formula*)

Extend the CNF+ formula with more clauses and/or AtMostK constraints. The additional clauses and AtMostK constraints to add should be given in the form of *CNFPlus*. Alternatively, a list of clauses can be added too. For every single clause and AtMostK constraint in the input formula, method *append()* is invoked.

Parameters **formula** (*CNFPlus*) – new constraints to add.

Example:

```
>>> from pysat.formula import CNFPlus
>>> cnf1 = CNFPlus()
>>> cnf1.extend([[-3, 4], [5, 6], [[1, 2, 3], 1]])
>>> print(cnf1.clauses)
[[-3, 4], [5, 6]]
```

(continues on next page)

(continued from previous page)

```
>>> print(cnf1.atmosts)
[[[1, 2, 3], 1]]
>>> cnf2 = CNFPlus()
>>> cnf2.extend(cnf1)
>>> print(cnf1.clauses)
[[-3, 4], [5, 6]]
>>> print(cnf1.atmosts)
[[[1, 2, 3], 1]]
```

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

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

Parameters

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

Usage example:

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

to_alien(*file_pointer*, *format='opb'*, *comments=None*)

The method can be used to dump a CNF+ formula into a file pointer in an alien file format, which at this point can either be LP, OPB, or SMT. The file pointer is expected as an argument. Additionally, the target format 'lp', 'opb', or 'smt' may be specified (equal to 'opb' by default). Finally, supplementary comment lines can be specified in the `comments` parameter.

Note: SMT-LIB2 does not directly support PB constraints. As a result, native cardinality constraints of CNF+ cannot be translated to SMT-LIB2 unless an explicit cardinality encoding is applied. You may want to use Z3's API instead (see its PB interface).

Parameters

- **file_pointer** (*file pointer*) – a file pointer where to store the formula.
- **format** (*str*) – alien file format to use
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

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

to_dimacs()

Return the current state of the object in extended DIMACS format.

For example, if 'some-file.cnf' contains:

```
c Example
p cnf+ 7 3
1 -2 3 5 -7 <= 3
4 5 6 -7 >= 2
3 5 7 0
```

Then you can obtain the DIMACS with:

```
>>> from pysat.formula import CNFPlus
>>> cnf = CNFPlus(from_file='some-file.cnf')
>>> print(cnf.to_dimacs())
c Example
p cnf+ 7 3
3 5 7 0
1 -2 3 5 -7 <= 3
-4 -5 -6 7 <= 2
```

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

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

Example:

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

weighted()

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

Returns an object of class *WCNFPlus*.

Example:

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

(continues on next page)

(continued from previous page)

```
>>> wcnf = cnf.weighted()
>>> print(wcnf.hard)
[]
>>> print(wcnf.soft)
[[-1, 2], [3, 4]]
>>> print(wcnf.wght)
[1, 1]
>>> print(wcnf.atms)
[[[1, 2], 1]]
```

class pysat.formula.**Equals**(**args*, ***kwargs*)

Equivalence. Given a list of operands (subformulas) $f_i, i \in \{1, \ldots, n\}, n \in \mathbb{N}$, it creates a formula $f_1 \leftrightarrow f_2 \leftrightarrow \ldots \leftrightarrow f_n$. The list of operands *of size at least 2* should be passed as arguments to the constructor.

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> equiv = Equals(x, y, z)
```

If an additional Boolean keyword argument `merge` is provided set to `True`, the toolkit will try to flatten the current *Equals* formula merging its *equivalence* sub-operands into the list of operands. For example, if `Equals(Equals(x, y), z, merge=True)` is called, a new Formula object will be created with two operands: `Equals(x, y)` and `z`, followed by merging `x` and `y` into the list of root-level `Equals`. This will result in a formula `Equals(x, y, z)`. Merging sub-operands is enabled by default if bitwise operations are used to create `Equals` formulas.

Example:

```
>>> from pysat.formula import *
>>> a1 = Equals(Equals(Atom('x'), Atom('y')), Atom('z'))
>>> a2 = Equals(Equals(Atom('x'), Atom('y')), Atom('z'), merge=True)
>>> a3 = Atom('x') == Atom('y') == Atom('z')
>>>
>>> print(a1)
(x @ y) @ z
>>> print(a2)
x @ y @ z
>>> print(a3)
x @ y @ z
>>>
>>> id(a1) == id(a2)
False
>>>
>>> id(a2) == id(a3)
True  # formulas a2 and a3 refer to the same object
```

Note: If there are two formulas representing the same fact with and without merging enabled, they technically sit in two distinct objects. Although PySAT tries to avoid it, clausification of these two formulas may result in unique (different) auxiliary variables assigned to such two formulas.

---

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the subformulas and creates a new formula.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = x @ y @ z
>>>
>>> print(a.simplified(assumptions=[y]))
x & z    # x and z must also be True
>>> print(a.simplified(assumptions=[~y]))
~x & ~z  # x and z must also be False
```

class pysat.formula.**Formula**(**args*, ***kwargs*)

Abstract formula class. At the same time, the class is a factory for its children and can be used this way although it is recommended to create objects of the children classes directly. In particular, its children classes include *Atom* (atomic formulas - variables and constants), *Neg* (negations), *And* (conjunctions), *Or* (disjunctions), *Implies* (implications), *Equals* (equalities), *XOr* (exclusive disjunctions), and *ITE* (if-then-else operations).

Due to the need to clausify formulas, an object of any subclass of *Formula* is meant to be represented in memory by a single copy. This is achieved by storing a dictionary of all the known formulas attached to a given *context*. Thus, once a particular context is activated, its dictionary will make sure each formula variable refers to a single representation of the formula object it aims to refer. When it comes to clausifying this formula, the formula is encoded exactly once, despite it may be potentially used multiple times as part of one of more complex formulas.

Example:

```
>>> from pysat.formula import *
>>>
>>> x1, x2 = Atom('x'), Atom('x')
>>> id(x1) == id(x2)
True  # x1 and x2 refer to the same atom
>>> id(x1 & Atom('y')) == id(Atom('y') & x2)
True  # it holds if constructing complex formulas with them as subformulas
```

The class supports multi-context operation. A user may have formulas created and clausified in different context. They can also switch from one context to another and/or cleanup the instances known in some or all contexts.

Example:

```
>>> from pysat.formula import *
>>> f1 = Or(And(...))  # arbitrary formula
>>> # by default, the context is set to 'default'
>>> # another context can be created like this:
>>> Formula.set_context(context='some-other-context')
>>> # the new context knows nothing about formula f1
>>> # ...
>>> # cleaning up context 'some-other-context'
>>> # this deletes all the formulas known in this context
>>> Formula.cleanup(context='some-other-context')
>>> # switching back to 'default'
>>> Formula.set_context(context='default')
```

A user may also want to disable duplicate blocking, which can be achieved by setting the context to `None`.

Boolean constants False and True are represented by the atomic "formulas" `Atom(False)` and `Atom(True)`, respectively. There are two constants storing these values: `PYSAT_FALSE` and `PYSAT_TRUE`.

```
>>> PYSAT_FALSE, PYSAT_TRUE
(Atom(False), Atom(True))
```

static attach_vpool(*vpool*, *context='default'*)

Attach an external *IDPool* to be associated with a given context. This is useful when a user has an already created *IDPool* object and wants to reuse it when clausifying their *Formula* objects.

Parameters

- **vpool** (*IDPool*) – an external variable manager
- **context** (*hashable*) – target context to be the user of the vpool

clausify()

This method applies Tseitin transformation to the formula. Recursively gives all the formulas Boolean names accordingly and uses them in the current logic connective following its semantics. As a result, each subformula stores its clausal representation independently of other subformulas (and independently of the root formula).

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> a.clausify()  # clausifying formula a
>>>
>>> # let's what clauses represent the root logic connective
>>> a.clauses
[[3, 4]]  # 4 corresponds to z while 3 represents the equality x @ y
```

static cleanup(*context=None*)

Clean up either a given context (if specified as different from `None`) or all contexts (otherwise); afterwards, start the "default" context from scratch.

A context is cleaned by destroying all the associated *Formula* objects and all the corresponding variable managers. This may be useful if a user wants to start encoding their problem from scratch.

Note: Once cleaning of a context is done, the objects referring to the context's formulas must not be used. At this point, they are orphaned and can't get re-clausified.

Parameters **context** (None or hashable) – target context

static export_vpool(*active=True*, *context='default'*)

Opposite to *attach_vpool()*, this method returns a variable managed attached to a given context, which may be useful for external use.

Parameters

- **active** (*bool*) – export the currently active context
- **context** (*hashable*) – context using the vpool we are interested in (if `active` is `False`)

Return type *IDPool*

static formulas(*lits*, *atoms_only=True*)

Given a list of integer literal identifiers, this method returns a list of formulas corresponding to these identifiers. Basically, the method can be seen as mapping auxiliary variables naming formulas to the corresponding formulas they name.

If the argument `atoms_only` is set to `True` only, the method will return a subset of formulas, including only atomic formulas (literals). Otherwise, any formula whose name occurs in the input list will be included in the result.

Parameters

- **lits** (*iterable*) – input list of literals
- **atoms_only** (*bool*) – include all known formulas or atomic ones only

Return type list(*Formula*)

Example:

```
>>> from pysat.formula import *
>>> from pysat.solvers import Solver
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) ^ z
>>>
>>> with Solver(bootstrap_with=a) as solver:
...     for model in solver.enum_models():
...         # using method formulas to map the model back to atoms
...         print(Formula.formulas(model, atoms_only=True))
...
[Neg(Atom('x')), Neg(Atom('y')), Neg(Atom('z'))]
[Neg(Atom('x')), Atom('y'), Atom('z')]
[Atom('x'), Atom('y'), Neg(Atom('z'))]
[Atom('x'), Neg(Atom('y')), Atom('z')]
```

static literals(*forms*)

Extract formula names for a given list of formulas and return them as a list of integer identifiers. Essentially, the method is the opposite to *formulas()*.

Parameters **forms** (*iterable*) – list of formulas to map

Return type list(int)

Example:

```
>>> from pysat.solvers import Solver
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) ^ z
>>>
>>> # applying Tseitin transformation to formula a
>>> a.clausify()
>>>
>>> # checking what facts the internal vpool knows
>>> print(Formula.export_vpool().id2obj)
```

(continues on next page)

(continued from previous page)

```
{1: Atom('x'), 2: Atom('y'), 3: Equals[Atom('x'), Atom('y')], 4: Atom('z')}
>>>
>>> # now, mapping two atoms to their integer id representations
>>> Formula.literals(forms=[Atom('x'), ~Atom('z')])
[1, -4]
```

satisfied(*model*)

Given a list of atomic formulas, this method checks whether the current formula is satisfied by assigning these atoms. The method returns `True` if the formula gets satisfied, `False` if it is falsified, and `None` if the answer is unknown.

Parameters **model** (list(*Formula*)) – list of atomic formulas

Return type bool or `None`

Example:

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z
>>>
>>> a.satisfied(model=[z])
True
>>> a.satisfied(model=[~z])
>>> # None, as it is not enough to set ~z to determine satisfiability of a
```

static set_context(*context='default'*)

Set the current context of interest. If set to `None`, no context will be assumed and duplicate checking will be disabled as a result.

Parameters **context** (*hashable*) – new active context

simplified(*assumptions=[]*)

Given a list of assumption atomic formula literals, this method recursively assigns these atoms to the corresponding values followed by formula simplification. As a result, a new formula object is returned.

Parameters **assumptions** (*list*) – atomic formula objects

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>>
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = (x @ y) | z  # a formula over 3 variables: x, y, and z
>>>
>>> a.simplified(assumptions=[z])
Atom(True)
>>>
>>> a.simplified(assumptions=[~z])
Equals[Atom('x'), Atom('y')]
>>>
>>> b = a ^ Atom('p')  # a more complex formula
>>>
```

(continues on next page)

(continued from previous page)

```
>>> b.simplified(assumptions=[x, ~Atom('p')])
Or[Atom('y'), Atom('z')]
```

exception pysat.formula.**FormulaError**

This exception is raised when an formula-related issue occurs.

class pysat.formula.**FormulaType**(*value*)

This class represents a C-like `enum` type for choosing the formula type to use. The values denoting all the formula types are as follows:

```
ATOM = 0
AND = 1
OR = 2
NEG = 3
IMPL = 4
EQ = 5
XOR = 6
ITE = 7
```

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

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

Parameters

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

id(*obj=None*)

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. Note that if the object is not provided, the method will return a new id unassigned to any object.

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

Return type int.

Example:

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

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

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

obj(*vid*)

The method can be used to map back a given variable identifier to the original object labeled by the identifier.

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

Returns an object corresponding to the given identifier.

Example:

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

occupy(*start*, *stop*)

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

Parameters

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

restart(*start_from=1*, *occupied=[]*)

Restart the manager from scratch. The arguments replicate those of the constructor of *IDPool*.

class pysat.formula.**ITE**(**args*, ***kwargs*)

If-then-else operator. Given three operands (subformulas) x, y, and z, it creates a formula $(x \rightarrow y) \wedge (\neg x \rightarrow z)$. The operands should be passed as arguments to the constructor.

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> ite = ITE(x, y, z)
>>>
>>> print(ite)
>>> (x >> y) & (~x >> z)
```

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the subformulas and creates a new formula.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> ite = ITE(x, y, z)
>>>
>>> print(ite.simplified(assumptions=[y]))
```

(continues on next page)

(continued from previous page)

```
x | z
>>> print(ite.simplified(assumptions=[~y]))
~x & z
```

class pysat.formula.**Implies**(**args*, ***kwargs*)

Implication. Given two operands f_1 and f_2, it creates a formula $f_1 \rightarrow f_2$. The operands must be passed to the constructors either as two arguments or two keyword arguments `left` and `right`.

Example:

```
>>> from pysat.formula import *
>>> x, y = Atom('x'), Atom('y')
>>> a = Implies(x, y)
>>> print(a)
x >> y
```

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the left and right subformulas and then creates and returns a new formula with these simplified subformulas.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y')
>>> a = x >> y
>>>
>>> print(a.simplified(assumptions=[y]))
T
>>> print(a.simplified(assumptions=[~y]))
~x
```

class pysat.formula.**Neg**(**args*, ***kwargs*)

Negation. Given a single operand (subformula) f, it creates a formula $\neg f$. The operand must be passed as an argument to the constructor.

Example:

```
>>> from pysat.formula import *
>>> x = Atom('x')
>>> n1 = Neg(x)
>>> n2 = Neg(subformula=x)
>>> print(n1, n2)
~x, ~x
>>> n3 = ~n1
>>> print(n3)
x
```

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the subformula and then creates and returns a new formula with this simplified subformula.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = x & y | z
>>> b = ~a
>>>
>>> print(b.simplified(assumptions=[y]))
~(x | z)
>>> print(b.simplified(assumptions=[~y]))
~z
```

class pysat.formula.**Or**(**args*, ***kwargs*)

Disjunction. Given a list of operands (subformulas) $f_i, i \in \{1, \ldots, n\}, n \in \mathbb{N}$, it creates a formula $\bigvee_{i=1}^{n} f_i$. The list of operands *of size at least 1* should be passed as arguments to the constructor.

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> conj = Or(x, y, z)
```

If an additional Boolean keyword argument `merge` is provided set to `True`, the toolkit will try to flatten the current *Or* formula merging its *conjuctive* sub-operands into the list of operands. For example, if `Or(Or(x, y), z, merge=True)` is called, a new Formula object will be created with two operands: `Or(x, y)` and `z`, followed by merging `x` and `y` into the list of root-level `Or`. This will result in a formula `Or(x, y, z)`. Merging sub-operands is enabled by default if bitwise operations are used to create `Or` formulas.

Example:

```
>>> from pysat.formula import *
>>> a1 = Or(Or(Atom('x'), Atom('y')), Atom('z'))
>>> a2 = Or(Or(Atom('x'), Atom('y')), Atom('z'), merge=True)
>>> a3 = Atom('x') | Atom('y') | Atom('z')
>>>
>>> repr(a1)
"Or[Or[Atom('x'), Atom('y')], Atom('z')]"
>>> repr(a2)
"Or[Atom('x'), Atom('y'), Atom('z')]"
>>> repr(a2)
"Or[Atom('x'), Atom('y'), Atom('z')]"
>>>
>>> id(a1) == id(a2)
False
>>>
>>> id(a2) == id(a3)
True  # formulas a2 and a3 refer to the same object
```

Note: If there are two formulas representing the same fact with and without merging enabled, they technically sit in two distinct objects. Although PySAT tries to avoid it, clausification of these two formulas may result in unique (different) auxiliary variables assigned to such two formulas.

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the subformulas and creates a new formula.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = x | y | z
>>>
>>> print(a.simplified(assumptions=[y]))
T  # True
>>> print(a.simplified(assumptions=[~y]))
x | z
```

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

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

Parameters

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

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

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

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

Parameters

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

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

copy()

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

Returns an object of class *WCNF*.

Example:

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

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

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

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

Parameters

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

Example:

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

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

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

Parameters

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

- **compressed_with** (*str*) – file compression algorithm

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

Usage example:

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

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

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

Parameters

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

Usage example:

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

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

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

Parameters

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

Example:

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

(continues on next page)

(continued from previous page)

```
>>> print(cnf2.soft)
[[-3]]
>>> print(cnf2.nv)
3
```

normalize_negatives(*negatives*)

Iterate over all soft clauses with negative weights and add their negation either as a hard clause or a soft one.

Parameters **negatives** (*list(list(int))*) – soft clauses with their negative weights.

to_alien(*file_pointer*, *format='opb'*, *comments=None*)

The method can be used to dump a WCNF formula into a file pointer in an alien file format, which at this point can either be LP, OPB, or SMT. The file pointer is expected as an argument. Additionally, the target format 'lp', 'opb', or 'smt' may be specified (equal to 'opb' by default). Finally, supplementary comment lines can be specified in the `comments` parameter.

Parameters

- **file_pointer** (*file pointer*) – a file pointer where to store the formula.
- **format** (*str*) – alien file format to use
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

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

to_dimacs()

Return the current state of the object in extended DIMACS format.

For example, if 'some-file.cnf' contains:

```
c Example
p wcnf 2 3 10
1 -1 0
2 -2 0
10 1 2 0
```

Then you can obtain the DIMACS with:

```
>>> from pysat.formula import WCNF
>>> cnf = WCNF(from_file='some-file.cnf')
>>> print(cnf.to_dimacs())
c Example
p wcnf 2 3 10
10 1 2 0
1 -1 0
2 -2 0
```

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

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

Parameters

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

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

Example:

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

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

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

Parameters

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

Example:

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

unweighted()

This method creates a *plain* (unweighted) copy of the internal formula. As a result, an object of class *CNF* is returned. Every clause (both hard or soft) of the WCNF formula is copied to the `clauses` variable of the resulting plain formula, i.e. all weights are discarded.

Returns an object of class *CNF*.

Example:

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

(continues on next page)

(continued from previous page)

```
>>> 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
```

Additionally, *WCNFPlus* support pseudo-Boolean constraints, i.e. weighted linear constraints by extending the above format. Basically, a pseudo-Boolean constraint needs to specify all the summands as `weight*literal` with the entire constraint being prepended with character `w` as follows:

```
c Example: One cardinality constraint and one PB constraint followed by a soft␣
↪clause
p wcnf+ 7 3 10
10 1 -2 3 5 -7 <= 3
10 w 1*4 2*5 1*6 3*-7 >= 2
5 3 5 7 0
```

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

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

Example:

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

For details on the functionality, see *WCNF*.

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

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

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

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

Parameters

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

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

copy()

This method can be used for creating a copy of a WCNFPlus object. It creates another object of the *WCNFPlus* class, call the copy function of WCNF class and makes use of the *deepcopy* functionality to copy the atmost constraints.

Returns an object of class *WCNFPlus*.

Example:

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

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

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

Parameters

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

Usage example:

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

to_alien(*file_pointer*, *format='opb'*, *comments=None*)

The method can be used to dump a WCNF+ formula into a file pointer in an alien file format, which at this point can either be LP, OPB, or SMT. The file pointer is expected as an argument. Additionally, the target format 'lp', 'opb', or 'smt' may be specified (equal to 'opb' by default). Finally, supplementary comment lines can be specified in the `comments` parameter.

Note: SMT-LIB2 does not directly support PB constraints. As a result, native cardinality constraints of CNF+ cannot be translated to SMT-LIB2 unless an explicit cardinality encoding is applied. You may want to use Z3's API instead (see its PB interface).

Parameters

- **file_pointer** (*file pointer*) – a file pointer where to store the formula.
- **format** (*str*) – alien file format to use
- **comments** (*list(str)*) – additional comments to put in the file.

Example:

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

to_dimacs()

Return the current state of the object in extended DIMACS format.

For example, if 'some-file.cnf' contains:

```
c Example
p wcnf+ 7 3 10
10 1 -2 3 5 -7 <= 3
10 4 5 6 -7 >= 2
5 3 5 7 0
```

Then you can obtain the DIMACS with:

```
>>> from pysat.formula import WCNFPlus
>>> cnf = WCNFPlus(from_file='some-file.cnf')
>>> print(cnf.to_dimacs())
c Example
p wcnf+ 7 4 10
10 -1 3 5 0
5 3 5 7 0
10 1 -2 3 5 -7 <= 3
10 -4 -5 -6 7 <= 2
```

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

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

Example:

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

unweighted()

This method creates a *plain* (unweighted) copy of the internal formula. As a result, an object of class *CNFPlus* is returned. Every clause (both hard or soft) of the original WCNFPlus formula is copied to the `clauses` variable of the resulting plain formula, i.e. all weights are discarded.

Note that the cardinality constraints of the original (weighted) formula remain unchanged in the new (plain) formula.

Returns an object of class *CNFPlus*.

Example:

```
>>> from pysat.formula import WCNF
>>> wcnf = WCNFPlus()
>>> wcnf.extend([[-3, 4], [5, 6]])
>>> wcnf.extend([[3], [-4], [-5], [-6]], weights=[1, 5, 3, 4])
>>> wcnf.append([[1, 2, 3], 1], is_atmost=True)
>>>
>>> cnf = wcnf.unweighted()
>>> print(cnf.clauses)
[[-3, 4], [5, 6], [3], [-4], [-5], [-6]]
>>> print(cnf.atmosts)
[[[1, 2, 3], 1]]
```

class pysat.formula.**XOr**(**args*, ***kwargs*)

Exclusive disjunction. Given a list of operands (subformulas) f_i, $i \in \{1, \ldots, n\}, n \in \mathbb{N}$, it creates a formula $f_1 \oplus f_2 \oplus \ldots \oplus f_n$. The list of operands *of size at least 2* should be passed as arguments to the constructor.

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> xor = XOr(x, y, z)
```

If an additional Boolean keyword argument `merge` is provided set to `True`, the toolkit will try to flatten the current *XOr* formula merging its *equivalence* sub-operands into the list of operands. For example, if `XOr(XOr(x, y), z, merge=True)` is called, a new Formula object will be created with two operands: `XOr(x, y)` and `z`, followed by merging `x` and `y` into the list of root-level `XOr`. This will result in a formula `XOr(x, y, z)`. Merging sub-operands is disabled by default if bitwise operations are used to create `XOr` formulas.

Example:

```
>>> from pysat.formula import *
>>> a1 = XOr(XOr(Atom('x'), Atom('y')), Atom('z'))
>>> a2 = XOr(XOr(Atom('x'), Atom('y')), Atom('z'), merge=True)
>>> a3 = Atom('x') ^ Atom('y') ^ Atom('z')
>>>
>>> print(a1)
(x ^ y) ^ z
>>> print(a2)
x ^ y ^ z
>>> print(a3)
(x ^ y) ^ z
>>>
>>> id(a1) == id(a2)
False
>>>
>>> id(a1) == id(a3)
True  # formulas a1 and a3 refer to the same object
```

Note: If there are two formulas representing the same fact with and without merging enabled, they technically sit in two distinct objects. Although PySAT tries to avoid it, clausification of these two formulas may result in unique (different) auxiliary variables assigned to such two formulas.

---

simplified(*assumptions=[]*)

Given a list of assumption literals, recursively simplifies the subformulas and creates a new formula.

Parameters **assumptions** (list(*Formula*)) – atomic assumptions

Return type *Formula*

Example:

```
>>> from pysat.formula import *
>>> x, y, z = Atom('x'), Atom('y'), Atom('z')
>>> a = x ^ y ^ z
>>>
>>> print(a.simplified(assumptions=[y]))
~x ^ z
>>> print(a.simplified(assumptions=[~y]))
x ^ z
```

1.1.3 External engines (pysat.engines)

List of classes

Propagator	An abstract class for creating external user-defined propagators / reasoning engines to be used with solver Cadical195 through the IPASIR-UP interface.
BooleanEngine	A simple *example* Boolean constraint propagator inheriting from the class *Propagator*.
LinearConstraint	A possible implementation of linear constraints over Boolean variables, including cardinality and pseudo-Boolean constraints.
ParityConstraint	A possible implementation of parity constraints.

Module description

This module provides a user with the possibility to define their own propagation engines, i.e. constraint propagators, attachable to a SAT solver. The implementation of this functionality builds on the use of the IPASIR-UP interface[1]. This may come in handy when it is beneficial to reason over non-clausal constraints, for example, in the settings of satisfiability modulo theories (SMT), constraint programming (CP) and lazy clause generation (LCG).

Note: Currently, the only SAT solver available in PySAT supporting the interface is CaDiCaL 1.9.5.

The interface allows a user to attach a single reasoning engine to the solver. This means that if one needs to support multiple kinds of constraints simultaneously, the implementation of the engine may need to be sophisticated enough to make it work.

It is imperative that any propagator a user defines must inherit the interface of the abstract class *Propagator* and defines all the required methods for the correct operation of the engine.

An example propagator is shown in the class *BooleanEngine*. It currently supports two kinds of example constraints: linear (cardinality and pseudo-Boolean) constraints and parity (exclusive OR, XOR) constraints. The engine can run in the *adaptive mode*, i.e. it can enable and disable itself on the fly.

Once an engine is implemented, it should be attached to a solver object by calling connect_propagator() of Cadical195. The propagator will then need to inform the solver what variable it requires to observe.

```
solver = Solver(name='cadical195', bootstrap_with=some_formula)

engine = MyPowerfulEngine(...)
solver.connect_propagator(engine)

# attached propagator wants to observe these variables
for var in range(some_variables):
    solver.observe(var)

...
```

Note: A user is encouraged to examine the source code of *BooleanEngine* in order to see how an external reasoning

[1] Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, Armin Biere. *IPASIR-UP: User Propagators for CDCL*. SAT. 2023. pp. 8:1-8:13

engine can be implemented and attached to CaDiCaL 1.9.5. Also consult the implementation of the corresponding methods of `Cadical195`.

Module details

class pysat.engines.**BooleanEngine**(*bootstrap_with=[]*, *adaptive=True*)

A simple *example* Boolean constraint propagator inheriting from the class *Propagator*. The idea is to exemplify the use of external reasoning engines. The engine should be general enough to support various constraints over Boolean variables.

Note: Note that this is not meant to be a model implementation of an external engine. One can devise a more efficient implementation with the same functionality.

The initialiser of of the class object may be provided with a list of constraints, each being a tuple ('type', constraint), as a value for parameter `bootstrap_with`.

Currently, there are two types of constraints supported (to be specified) in the constraints passed in: `'linear'` and `'parity'` (exclusive OR). The former will be handled as objects of class *LinearConstraint* while the latter will be transformed into objects of `ParityConstraint`.

Here, each type of constraint is meant to have a list of literals stored in variable `.lits`. This is required to set up watched lists properly.

The second keyword argument `adaptive` (set to `True` by default) denotes the fact that the engine should check its own efficiency and disable or enable itself on the fly. This functionality is meant to exemplify how adaptive external engines can be created. A user is referred to the source code of the implementation for the details.

adaptive_constants(*pdecay*, *pbound*, *mdecay*, *mbound*)

Set magic numeric constants used in adaptive mode.

adaptive_update(*satisfied*)

Update adaptive mode: either enable or disable the engine. This depends on the statistics accumulated in the current run and whether or not the previous assignment found by the solver satisfied the constraints.

add_clause()

Extract a new clause to add to the solver if one exists; return an empty clause `[]` otherwise.

add_constraint(*constraint*)

Add a new constraint to the engine and integrate it to the internal structures, i.e. watched lists. Also, return the newly added constraint to the callee.

check_model(*model*)

Check if a given model satisfies all the constraints.

cleanup_watched(*lit*, *garbage*)

Garbage collect holes in the watched list for +lit (and potentially for -lit).

decide() → int

This method allows the propagator to influence the decision process. Namely, it is used when the solver asks the propagator for the next decision literal (if any). If the method returns `0`, the solver will make its own choice.

Return type int

disable()

Notify the solver that the propagator should become inactive as it does not contribute much to the inference process. From now on, it will only be called to check complete models obtained by the solver (see *check_model()*).

enable()

Notify the solver that the propagator is willing to become active from now on.

is_active()

Return engine's status. It is deemed active if the method returns `True` and passive otherwise.

on_assignment(*lit*, *fixed*)

Update the propagator's state given a new assignment.

on_backtrack(*to*)

Cancel all the decisions up to a certain level.

on_new_level()

Keep track of decision level updates.

preprocess()

Run some (naive) preprocessing techniques if available for the types of constraints under considerations. Each type of constraints is handled separately of the rest of constraints.

process_linear()

Process linear constraints. Here we apply simple pairwise summation of constraints. As the number of result constraints is quadratic, we stop the process as soon as we get 100 new constraints. Also, if a result of the sum is longer than each of the summands, the result constraint is ignored.

This is trivial procedure is made to illustrate how constraint processing can be done. It can be made dependent on user-specified parameters, e.g. the number of rounds or a numeric value indicating when a pair of constraints should be added and when they should not be added. For consideration in the future.

process_parity()

Process parity/XOR constraints. Basically, this runs Gaussian elimination and see if anything can be derived from it.

propagate()

Run the propagator given the current assignment.

provide_reason(*lit*)

Return the reason clause for a given literal.

setup_observe(*solver*)

Inform the solver about all the variables the engine is interested in. The solver will mark them as observed by the propagator.

class pysat.engines.**LinearConstraint**(*lits=[]*, *weights=[]*, *bound=1*)

A possible implementation of linear constraints over Boolean variables, including cardinality and pseudo-Boolean constraints. Each such constraint is meant to be in the less-than form, i.e. a user should transform the literals, weights and the right-hand side of the constraint into this form before creating an object of *LinearConstraint*. The class is designed to work with *BooleanEngine*.

The implementation of linear constraint propagation builds on the use of counters. Basically, each time a literal is assigned to a positive value, it is assumed to contribute to the total weight on the left-hand side of the constraint, which is calculated and compared to the right-hand side.

The constructor receives three arguments: `lits`, `weights`, and `bound`. Argument `lits` represents a list of literals on the left-hand side of the constraint while argument `weights` contains either a list of their weights or a dictionary mapping literals to weights. Finally, argument `bound` is the right-hand side of the constraint.

Note that if no weights are provided, each occurrence of a literal is assumed to have weight 1.

Note: All weights are supposed to be non-negative values.

Parameters

- **lits** – list of literals (left-hand side)
- **weights** (*list or dict*) – weights of the literals
- **bound** (*int or float*) – right-hand side of the constraint

abandon_unweighted(*dummy_lit*)

Clear the reason of a given literal.

abandon_weighted(*lit*)

Clear the reason of a given literal.

attach_values(*values*)

Give the constraint access to centralised values exposed from *BooleanEngine*.

explain_failure()

Provide a reason clause for why the previous model falsified the constraint. This will clause will be added to the solver.

falsified_by(*model*)

Check if the constraint is violated by a given assignment. Upon receiving such an input assignment, the method counts the sum of the weights of all satisfied literals and checks if it exceeds the right-hand side.

justify_unweighted(*dummy_lit*)

Provide a reason for a literal propagated by this constraint. In the unweighted case, all the literals propagated by this constraint share the same reason.

justify_weighted(*lit*)

Provide a reason for a literal propagated by this constraint. In the case of weighted constraints, a literal may have a reason different from the other literals propagated by the same constraint.

propagate_unweighted(*lit=None*)

Get all the consequences of a given literal in the unweighted case. The implementation *counts* how many literals on the left-hand side are assigned to true.

propagate_weighted(*lit=None*)

Get all the consequences of a given literal in the weighted case. The implementation counts the weights of all the literals assigned to true and propagates all the other literals (yet unassigned) such that adding their weights to the total sum would exceed the right-hand side of the constraint.

register_watched(*to_watch*)

Add self to the centralised watched literals lists in *BooleanEngine*.

unassign(*lit*)

Unassign a given literal, which is done by decrementing the literal's contribution to the total sum of the weights of assigned literals.

class pysat.engines.**Propagator**

An abstract class for creating external user-defined propagators / reasoning engines to be used with solver `Cadical195` through the IPASIR-UP interface. All user-defined propagators should inherit the interface of this abstract class, i.e. all the below methods need to be properly defined. The interface is as follows:

```
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass      # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass      # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass      # process backtracking to a given level

    def check_model(self, model: List[int]) -> bool:
        pass      # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0  # make a decision and (if any) inform the solver

    def propagate(self) -> List[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> List[int]:
        pass      # explain why a given literal was propagated

    def add_clause(self) -> List[int]:
        return [] # add an(y) external clause to the solver
```

add_clause() → List[int]

The method is called by the solver to add an external clause if there is any. The clause can be arbitrary but if it is root-satisfied or tautological, the solver will ignore it without learning it.

Root-falsified literals are eagerly removed from the clause. Falsified clauses trigger conflict analysis, propagating clauses trigger propagation. Unit clauses always (unless root-satisfied, see above) trigger backtracking to level 0.

An empty clause (or root falsified clause, see above) makes the formula unsatisfiable and stops the search immediately.

Return type iterable(int)

check_model(*model: List[int]*) → bool

The method is used for checking if a given (complete) truth assignment satisfies the constraint managed by the propagator. Receives a single argument storing the truth assignment found by the solver.

Note: If this method returns `False`, the propagator must be ready to provide an external clause in the following callback.

Parameters **model** (*iterable(int)*) – a list of integers representing the current model

Return type bool

`decide`() → int

This method allows the propagator to influence the decision process. Namely, it is used when the solver asks the propagator for the next decision literal (if any). If the method returns `0`, the solver will make its own choice.

Return type int

`on_assignment`(*lit: int*, *fixed: bool = False*) → None

The method is called to notify the propagator about an assignment made for one of the observed variables. An assignment is set to be "fixed" if it is permanent, i.e. the propagator is not allowed to undo it.

Parameters

- **lit** (*int*) – assigned literal
- **fixed** (*bool*) – a flag to mark the assignment as "fixed"

`on_backtrack`(*to: int*) → None

The method for notifying the propagator about backtracking to a given decision level. Accepts a single argument `to` signifying the backtrack level.

Parameters **to** (*int*) – backtrack level

`on_new_level`() → None

The method called to notify the propagator about a new decision level created by the solver.

`propagate`() → List[int]

The method should invoke propagation under the current assignment. It can return either a list of literals propagated or an empty list `[]`, informing the solver that no propagation is made under the current assignment.

Return type int

`provide_reason`(*lit: int*) → List[int]

The method is called by the solver when asking the propagator for the reason / antecedent clause for a literal the propagator previously inferred. This clause will be used in the following conflict analysis.

Note: The clause must contain the propagated literal.

Parameters **lit** (*int*) – literal to provide reason for

Return type iterable(int)

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

List of classes

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

Module description

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

```
$ pip install pypblib
```

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

A *pseudo-Boolean constraint* is a constraint of the form: $(\sum_{i=1}^{n} a_i \cdot x_i) \circ k$, where $a_i \in \mathbb{N}$, $x_i \in \{y_i, \neg y_i\}$, $y_i \in \mathbb{B}$, and $\circ \in \{\leq, =, \geq\}$. Pseudo-Boolean constraints arise in a number of important practical applications. Thus, several *encodings* of pseudo-Boolean constraints into CNF formulas are known[2]. The list of pseudo-Boolean encodings supported by this module include BDD[3][4], sequential weight counters[5], sorting networks[3], adder networks[3], and binary merge[6]. Access to all cardinality encodings can be made through the main class of this module, which is *PBEnc*.

Module details

class pysat.pb.**EncType**

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

```
best       = 0
bdd        = 1
seqcounter = 2
sortnetwrk = 3
adder      = 4
binmerge   = 5
native     = 6
```

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

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

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

exception pysat.pb.**NoSuchEncodingError**

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

with_traceback()

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

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

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

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

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

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

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

class pysat.pb.**PBEnc**

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

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

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

A synonym for *PBEnc.geq()*.

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

A synonim for *PBEnc.leq()*.

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

This method can be used for creating a CNF encoding of a weighted EqualsK constraint, i.e. of $\sum_{i=1}^{n} a_i \cdot x_i = k$. The method shares the arguments and the return type with method *PBEnc.leq()*. Please, see it for details.

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

This method can be used for creating a CNF encoding of a GEQ (weighted AtLeastK) constraint, i.e. of $\sum_{i=1}^{n} a_i \cdot x_i \geq k$. The method shares the arguments and the return type with method *PBEnc.leq()*. Please, see it for details.

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

This method can be used for creating a CNF encoding of a LEQ (weighted AtMostK) constraint, i.e. of $\sum_{i=1}^{n} a_i \cdot x_i \leq k$. The resulting set of clauses is returned as an object of class *formula.CNF*.

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

Parameters

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

Return type *pysat.formula.CNFPlus*

1.1.5 Formula processing (pysat.process)

List of classes

Processor	This class provides interface to CaDiCaL's preprocessor.

Module description

This module provides access to the preprocessor functionality of CaDiCaL 1.5.3. It can be used to process[1] (also see references therein) a given CNF formula and output a another formula, which is guaranteed to be *equisatisfiable* with the original formula. The processor can be invoked for a user-provided number of rounds. Also, the following preprocessing techniques can be used when running the processor:

- blocked clause elimination
- covered clause elimination
- globally-blocked clause elimination
- equivalent literal substitution
- bounded variable elimination
- failed literal probing
- hyper binary resolution
- clause subsumption
- clause vivification

Note that the numerous parameters used in CaDiCaL for tweaking the preprocessor's behavior are currently unavailable here. (Default values are used.)

```
>>> from pysat.formula import CNF
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>>
>>> cnf = CNF(from_clauses=[[1, 2], [3, 2], [-1, 4, -2], [3, -2], [3, 4]])
>>> processor = Processor(bootstrap_with=cnf)
>>>
>>> processed = processor.process()
>>> print(processed.clauses)
[]
>>> print(processed.status)
True
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     solver.solve()
True
...     print('proc model:', solver.get_model())
proc model: []
...     print('orig model:', processor.restore(solver.get_model()))
orig model: [1, -2, 3, -4]
```

(continues on next page)

[1] Armin Biere, Matti Järvisalo, Benjamin Kiesl. *Preprocessing in SAT Solving*. In *Handbook of Satisfiability - Second Edition*. pp. 391-435

(continued from previous page)

```
>>>
>>> processor.delete()
```

Module details

class pysat.process.**Processor**(*bootstrap_with=None*)

This class provides interface to CaDiCaL's preprocessor. The only input parameter is `bootstrap_with`, which is expected to be a *CNF* formula or a list (or iterable) of clauses.

Parameters **bootstrap_with** (*CNF* or iterable(iterable(int))) – a list of clauses for processor initialization.

Once created and used, a processor 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. It is *important* to keep the processor if a user wants to restore a model of the original formula.

The main methods of this class are *process()* and *restore()*. The former calls CaDiCaL's preprocessor while the latter can be used to reconstruct a model of the original formula given a model for the processed formula as illustrated below.

Note how keeping the *Processor* object is needed for model restoration. (If it is deleted, the information needed for model reconstruction is lost.)

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>>
>>> processor = Processor(bootstrap_with=[[-1, 2], [1, -2]])
>>> processor.append_formula([[-2, 3], [1]])
>>> processor.add_clause([-3, 4])
>>>
>>> processed = processor.process()
>>> print(processed.clauses)
[]
>>> print(processed.status)
True
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     solver.solve()
True
...     print('proc model:', solver.get_model())
proc model: []
...     print('orig model:', processor.restore(solver.get_model()))
orig model: [1, 2, 3, 4]
>>>
>>> processor.delete()
```

add_clause(*clause*)

Add a single clause to the processor.

Parameters **clause** (*list(int) or any iterable(int)*) – a clause to add

```
>>> processor = Processor()
>>> processor.add_clause([-1, 2, 3])
```

append_formula(*formula*)

Add a given list of clauses into the solver.

Parameters **formula** (iterable(iterable(int)), or *CNF*) – a list of clauses.

```
>>> cnf = CNF()
... # assume the formula contains clauses
>>> processor = Processor()
>>> processor.append_formula(cnf)
```

delete()

Actual destructor.

get_status()

Preprocessor's status as the result of the previous call to *process()*. A `False` status indicates that the formula is found to be unsatisfiable by the preprocessor. Otherwise, the status equals `True`.

Return type bool

process(*rounds=1*, *block=False*, *cover=False*, *condition=False*, *decompose=True*, *elim=True*, *probe=True*, *probehbr=True*, *subsume=True*, *vivify=True*)

Runs CaDiCaL's preprocessor for the internal formula for a given number of rounds and using the techniques specified in the arguments. Note that the default values of all the arguments used are set as in the default configuration of CaDiCaL 1.5.3.

As the result, the method returns a *CNF* object containing the processed formula. Additionally to the clauses, the formula contains a `status` variable, which is set to `False` if the preprocessor found the original formula to be unsatisfiable (and `True` otherwise). The same status value is set to the `status` variable of the processor itself.

It is important to note that activation of some of the preprocessing techniques conditionally depends on the activation of other preprocessing techniques. For instance, subsumed, blocked and covered clause elimination is invoked only if bounded variable elimination is active. Subsumption elimination in turn may trigger vivification and transitive reduction if the corresponding flags are set.

Parameters

- **rounds** (*int*) – number of preprocessing rounds
- **block** (*bool*) – apply blocked clause elimination
- **cover** (*bool*) – apply covered clause elimination
- **condition** (*bool*) – detect conditional autarkies and apply globally-blocked clause elimination
- **decompose** (*bool*) – detect strongly connected components (SCCs) in the binary implication graph (BIG) and apply equivalent literal substitution (ELS)
- **elim** (*bool*) – apply bounded variable elimination
- **probe** (*bool*) – apply failed literal probing
- **probehbr** (*bool*) – learn hyper binary resolvents while probing
- **subsume** (*bool*) – apply global forward clause subsumption
- **vivify** (*bool*) – apply clause vivification

Returns processed formula

Return type *CNF*

```
>>> from pysat.process import Processor
>>>
>>> processor = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3]])
>>> processor.add_clause([1])
>>>
>>> processed = processor.process()
>>> print(processed.clauses)
[[]]
>>> print(processed.status)
False  # this means the processor decided the formula to be unsatisfiable
>>>
>>> with Solver(bootstrap_with=processed) as solver:
...     solver.solve()
False
>>> processor.delete()
```

restore(*model*)

Reconstruct a model for the original formula given a model for the processed formula. Done by using CaDiCaL's extend() and reconstruction stack functionality.

Parameters **model** (*iterable(int)*) – a model for the preprocessed formula

Returns extended model satisfying the original formula

Return type list(int)

```
>>> from pysat.process import Processor
>>>
>>> with Processor(bootstrap_with=[[-1, 2], [-2, 3]]) as proc:
...     proc.add_clause([1])
...     processed = proc.process()
...     with Solver(bootstrap_with=processed) as solver:
...         solver.solve()
...         print('model:', proc.restore(solver.get_model()))
...
model: [1, 2, 3]
```

1.1.6 SAT solvers' API (pysat.solvers)

List of classes

SolverNames	This class serves to determine the solver requested by a user given a string name.
Solver	Main class for creating and manipulating a SAT solver.
Cadical103	CaDiCaL 1.0.3 SAT solver.
Cadical153	CaDiCaL 1.5.3 SAT solver.
Cadical195	CaDiCaL 1.9.5 SAT solver.
CryptoMinisat	CryptoMinisat solver accessed through `pycryptosat` package.
Gluecard3	Gluecard 3 SAT solver.
Gluecard4	Gluecard 4 SAT solver.
Glucose3	Glucose 3 SAT solver.
Glucose4	Glucose 4.1 SAT solver.
Glucose42	Glucose 4.2.1 SAT solver.
Lingeling	Lingeling SAT solver.
MapleChrono	MapleLCMDistChronoBT SAT solver.
MapleCM	MapleCM SAT solver.
Maplesat	MapleCOMSPS_LRB SAT solver.
Mergesat3	MergeSat 3 SAT solver.
Minicard	Minicard SAT solver.
Minisat22	MiniSat 2.2 SAT solver.
MinisatGH	MiniSat SAT solver (version from github).

Module description

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

- CaDiCaL (rel-1.0.3)
- Glucose (3.0)
- Glucose (4.1)
- Glucose (4.2.1)
- Lingeling (bbc-9230380-160707)
- MapleLCMDistChronoBT (SAT competition 2018 version)
- MapleCM (SAT competition 2018 version)
- Maplesat (MapleCOMSPS_LRB)
- Mergesat (3.0)
- Minicard (1.2)
- Minisat (2.2 release)
- Minisat (GitHub version)

Additionally, PySAT includes the versions of `Glucose3` and `Glucose4` that support native cardinality constraints, ported from `Minicard`:

- Gluecard3
- Gluecard4

Finally, PySAT offers rudimentary support of CryptoMiniSat5[3] through the interface provided by the `pycryptosat` package. The functionality is exposed via the class `CryptoMinisat`. Note that the solver currently implements only the basic functionality, i.e. adding clauses and XOR-clauses as well as making (incremental) SAT calls.

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

The module provides direct access to all supported solvers using the corresponding classes `Cadical103`, `Cadical153`, `Cadical195`, `CryptoMinisat`, `Gluecard3`, `Gluecard4`, `Glucose3`, `Glucose4`, `Lingeling`, `MapleChrono`, `MapleCM`, `Maplesat`, `Mergesat3`, `Minicard`, `Minisat22`, and `MinisatGH`. However, the solvers can also be accessed through the common base class *`Solver`* using the solver `name` argument. For example, both of the following pieces of code create a copy of the `Glucose3` solver:

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

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

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

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

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

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

```
>>> from pysat.solvers import Solver
>>> s = Solver()
>>>
```

(continues on next page)

[3] Mate Soos, Karsten Nohl, Claude Castelluccia. *Extending SAT Solvers to Cryptographic Problems*. SAT 2009. pp. 244-257
[1] Niklas Eén, Niklas Sörensson. *An Extensible SAT-solver*. SAT 2003. pp. 502-518
[2] Niklas Eén, Niklas Sörensson. *Temporal induction by incremental SAT solving*. Electr. Notes Theor. Comput. Sci. 89(4). 2003. pp. 543-560

(continued from previous page)

```
... # 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 `Cadical103`, `Cadical153`, `Cadical195`, `CryptoMinisat`, `Gluecard3`, `Gluecard4`, `Glucose3`, `Glucose4`, `Glucose42`, `Lingeling`, `MapleChrono`, `MapleCM`, `Maplesat`, `Mergesat3`, `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 *'cadical103'*, *'cadical153'*, *'cadical195'*, *'cryptosat'*, *'gluecard3'*, *'gluecard4'*, *'glucose3'*, *'glucose4'*, *glucose42*, *'lingeling'*, *'maplechrono'*, *'maplecm'*, *'maplesat'*, *'mergesat3'*, *'minicard'*, *'minisat22'*, and *'minisatgh'*.

with_traceback()

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

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

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

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

Parameters

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

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

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

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

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

```
>>> from pysat.solvers import Solver, Minisat22
>>>
>>> s = Solver(name='g4')
>>> s.add_clause([-1, 2])
```

(continues on next page)

(continued from previous page)

```
>>> 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 `Cadical103`, `Cadical153`, `Cadical195`, `Lingeling`, `Gluecard3`, `Gluecard4`, `Glucose3`, `Glucose4`, `Glucose42`, `MapleChrono`, `MapleCM`, and `Maplesat` can have additional default arguments. One such argument supported by is DRUP proof logging. This can be enabled by setting the `with_proof` argument to `True` (`False` by default):

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

Additionally, Glucose-based solvers, namely `Glucose3`, `Glucose4`, `Glucose42`, `Gluecard3`, and `Gluecard4` have one more default argument `incr` (`False` by default), which enables incrementality features introduced in Glucose3[4]. To summarize, the additional arguments of Glucose are:

Parameters

- **incr** (*bool*) – enable the incrementality features of Glucose3[4].
- **with_proof** (*bool*) – enable proof logging in the DRUP format.

Finally, most MiniSat-based solvers can be exploited in the "warm-start" mode in the case of *satisfiable* formulas. This may come in handy in various model enumeration settings. Note that warm-start mode is disabled in the case of limited solving with *"unknown"* outcomes. Warm-start mode can be set with the use of the *warm_start* parameter:

Parameters **warm_start** (*bool*) – use the solver in the "warm-start" mode

accum_stats()

Get accumulated low-level stats from the solver. Currently, the statistics includes the number of restarts, conflicts, decisions, and propagations.

Return type dict.

Example:

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

```
>>> from pysat.examples.genhard import PHP
>>> cnf = PHP(5)
>>> from pysat.solvers import Solver
>>> with Solver(bootstrap_with=cnf) as s:
...     print(s.solve())
...     print(s.accum_stats())
False
{'restarts': 2, 'conflicts': 201, 'decisions': 254, 'propagations': 2321}
```

activate_atmost()

Activate native linear (cardinality or pseudo-Boolean) constraint reasoning. This is supported only by `Cadical195` by means of its external propagators functionality and the use of `BooleanEngine`.

Note: IPASIR-UP related. `Cadical195` only.

---

add_atmost(*lits*, *k*, *weights=[]*, *no_return=True*)

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

Note that most of the solvers do not support 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.

Also, besides *unweighted* AtMostK constraints, some solvers (see `Cadical195`) support their weighted counterparts, i.e. pseudo-Boolean constraints of the form $\sum_{i=1}^{n} w_i \cdot x_i \leq k$. The weights of the literals can be specified using the argument `weights`.

Parameters

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

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

A usage example is the following:

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

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

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

Parameters

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

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

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

A usage example is the following:

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

add_xor_clause(*lits*, *value=True*)

Add a new XOR clause to solver's internal formula. The input parameters include the list of literals and the right-hand side (the value of the sum).

Note that XOR clauses are currently supported only by CryptoMinisat.

Parameters

- **lits** (*iterable(int)*) – list of literals in the clause (left-hand side)
- **value** (*bool*) – value of the sum (right-hand-side)

A usage example is the following:

```
>>> from pysat.solvers import Solver
>>> with Solver(name='cms', bootstrap_with=[[1, 3]]) as solver:
...     solver.add_xor_clause(lits=[1, 2], value=True)
...     for model in solver.enum_models():
...         print(model)
...
[-1, 2, 3]
[1, -2, 3]
[1, -2, -3]
```

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

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

Parameters

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

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

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

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

clear_interrupt()

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

conf_budget(*budget=- 1*)

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

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

Example:

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

configure(*parameters*)

Configure `Cadical153` or `Cadical195` by setting some of the predefined parameters to selected values. Note that this method can be invoked only for `Cadical153` and *Cadical195* – no other solvers support this for now.

Also note that this call must follow the creation of the new solver object; otherwise, an exception may be thrown.

The list of available options of `Cadical153` and `Cadical195` and the corresponding values they can be assigned to is provided here.

Parameters **parameters** (*dict*) – parameter names mapped to integer values

connect_propagator(*propagator*)

Attach an external propagator through the IPASIR-UP interface. The only expected argument is `propagator`, which must be an object of a class inheriting from the abstract class *Propagator*.

Note: IPASIR-UP related. `Cadical195` only.

---

dec_budget(*budget*)

Set limit (i.e. the upper bound) on the number of decisions 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 decisions is disabled.

Note that this functionality is supported by `Cadical103`, `Cadical153`, and `Cadical195` only!

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

Example:

```
>>> from pysat.solvers import Cadical153
>>> from pysat.examples.genhard import Parity
>>>
>>> cnf = Parity(size=10)  # too hard for a SAT solver
>>> c = Cadical153(bootstrap_with=cnf.clauses)
>>>
>>> c.dec_budget(500)  # doing at most 500 decisions
>>> print(c.solve_limited())  # making a limited oracle call
None
>>> c.delete()
```

delete()

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

disable_propagator()

Ask the solver to disable the propagator on the fly. This will it in passive mode, i.e. it will be invoked only to check assignments found by the solver.

Note: IPASIR-UP related. `Cadical195` only.

---

disconnect_propagator()

Disconnect the previously attached propagator. This will also reset all the variables marked in the solver as observed by the propagator.

Note: IPASIR-UP related. `Cadical195` only.

---

enable_propagator()

Ask the solver to enable the propagator on the fly. This will put it in active mode.

Note: IPASIR-UP related. `Cadical195` only.

---

enum_models(*assumptions=[]*)

This method can be used to enumerate models of a CNF formula and it performs as a standard Python iterator. The method can be called without arguments but also with an argument `assumptions`, which represents a list of literals to "assume".

Note that the method expects the list of assumption literals (if any) to contain **no duplicate literals**. Otherwise, it is not guaranteed to run correctly. As such, a user is recommended to explicitly filter out duplicate literals from the assumptions list before calling *solve()*, *solve_limited()*, *propagate()*, or *enum_models()*.

Warning: Once finished, model enumeration results in the target formula being *unsatisfiable*. This is because the enumeration process *blocks* each previously computed model by adding a new clause until no more models of the formula exist.

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

Return type list(int).

Example:

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

(continues on next page)

(continued from previous page)

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

get_core()

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

Return type list(int) or `None`.

Usage example:

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

get_model()

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

Return type list(int) or `None`.

Example:

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

get_proof()

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

Return type list(str) or `None`.

Example:

```
>>> from pysat.solvers import Solver
>>> from pysat.examples.genhard import PHP
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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.

ignore(*var*)

Inform the solver that a given variable is ignored by the propagator attached to it.

Note: IPASIR-UP related. Cadical195 only.

---

interrupt()

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

Note that this method can be called if limited SAT calls are made with the option expect_interrupt set to True.

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

Example:

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

is_decision(*lit*)

Check whether a given literal that is currently observed by the attached propagator is a assigned a value by branching or by propagation, i.e. whether it is decision or not. In the former case, the method returns True; otherwise, it returns False.

Note: IPASIR-UP related. `Cadical195` only.

new(*name='m22'*, *bootstrap_with=None*, *use_timer=False*, ***kwargs*)

The actual solver constructor invoked from `__init__()`. Chooses the solver to run, based on its name. See *Solver* for the parameters description.

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

nof_clauses()

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

Return type int.

Example:

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

nof_vars()

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

Return type int.

Example:

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

observe(*var*)

Inform the solver that a given variable is observed by the propagator attached to it.

Note: IPASIR-UP related. `Cadical195` only.

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 propagations is disabled.

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

Example:

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

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

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

Note that the method expects the list of assumption literals (if any) to contain **no duplicate literals**. Otherwise, it is not guaranteed to run correctly. As such, a user is recommended to explicitly filter out duplicate literals from the assumptions list before calling *solve()*, *solve_limited()*, *propagate()*, or *enum_models()*.

Note that only MiniSat-like solvers support this functionality (e.g. `Cadical103`, class:*Cadical153*, `Cadical195`, and `Lingeling` do not support it).

Parameters

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

Return type tuple(bool, list(int)).

Usage example:

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

propagator_active()

Check if the propagator is currently active or passive. In the former case, the method will return `True`; otherwise, it will return `False`.

Note: IPASIR-UP related. `Cadical195` only.

---

reset_observed()

Ask the solver to reset all observed variables.

Note: IPASIR-UP related. `Cadical195` only.

---

set_phases(*literals=[]*)

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

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

Also **note** that `Cadical103`, `Cadical153`, and `Cadical195` do not support this functionality.

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

Usage example:

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

solve(*assumptions=[]*)

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

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

Note that the method expects the list of assumption literals (if any) to contain **no duplicate literals**. Otherwise, it is not guaranteed to run correctly. As such, a user is recommended to explicitly filter out duplicate literals from the assumptions list before calling *solve()*, *solve_limited()*, *propagate()*, or *enum_models()*.

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=[]*, *expect_interrupt=False*)

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. `Cadical103`, `Cadical153`, `Cadical195`, and `Lingeling` do not support it).

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

Note that the method expects the list of assumption literals (if any) to contain **no duplicate literals**. Otherwise, it is not guaranteed to run correctly. As such, a user is recommended to explicitly filter out duplicate literals from the assumptions list before calling *solve()*, *solve_limited()*, *propagate()*, or *enum_models()*.

Note that since SIGINT handling and *interrupt()* are not configured to work *together* at this point, additional input parameter `expect_interrupt` is assumed to be given, indicating what kind of interruption may happen during the execution of *solve_limited()*: whether a SIGINT signal or internal *interrupt()*. By default, a SIGINT signal handling is assumed. If `expect_interrupt` is set to `True` and eventually a SIGINT is received, the behavior is **undefined**.

Parameters

- **assumptions** (*iterable(int)*) – a list of assumption literals.
- **expect_interrupt** (*bool*) – whether *interrupt()* will be called

Return type Boolean or `None`.

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

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

Usage example:

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

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

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

start_mode(*warm=False*)

Set start mode: either warm or standard. Warm start mode can be beneficial if one is interested in efficient model enumeration.

Note that warm start mode is disabled in the case of limited solving with *"unknown"* outcomes. Moreover, warm start mode may lead to unexpected results in case of assumption-based solving with a *varying* list of assumption literals.

Example:

```
>>> def model_count(solver, formula, vlimit=None, warm_start=False):
...     with Solver(name=solver, bootstrap_with=formula, use_timer=True, warm_
↪start=warm_start) as oracle:
...         count = 0
...         while oracle.solve() == True:
...             model = oracle.get_model()
...             if vlimit:
...                 model = model[:vlimit]
...             oracle.add_clause([-l for l in model])
...             count += 1
...         print('{0} models in {1:.4f}s'.format(count, oracle.time_accum()))
>>>
>>> model_count('mpl', cnf, vlimit=16, warm_start=False)
58651 models in 7.9903s
>>> model_count('mpl', cnf, vlimit=16, warm_start=True)
58651 models in 0.3951s
```

supports_atmost()

This method can be called to determine whether the solver supports native AtMostK (see *pysat.card*) constraints.

Return type bool

A usage example is the following:

```
>>> s = Solver(name='mc')
>>> s.supports_atmost()
True
>>> # there is support for AtMostK constraints in this solver
```

time()

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

Return type float.

Example usage:

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

time_accum()

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

Return type float.

Example usage:

```
>>> from pysat.solvers import Solver
>>> from pysat.examples.genhard import PHP
>>>
>>> cnf = PHP(nof_holes=10)
>>> with Solver(bootstrap_with=cnf.clauses, use_timer=True) as s:
...     print(s.solve(assumptions=[1]))
False
...     print('{0:.2f}s'.format(s.time()))
1.76s
...     print(s.solve(assumptions=[-1]))
False
...     print('{0:.2f}s'.format(s.time()))
113.58s
...     print('{0:.2f}s'.format(s.time_accum()))
115.34s
```

class pysat.solvers.**SolverNames**

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

```
cadical103  = ('cd', 'cd103', 'cdl', 'cdl103', 'cadical103')
cadical153  = ('cd15', 'cd153', 'cdl15', 'cdl153', 'cadical153')
cadical195  = ('cd19', 'cd195', 'cdl19', 'cdl195', 'cadical195')
cryptosat   = ('cms', 'cms5', 'crypto', 'crypto5', 'cryptominisat', 'cryptominisat5
↪')
gluecard3   = ('gc3', 'gc30', 'gluecard3', 'gluecard30')
gluecard41  = ('gc4', 'gc41', 'gluecard4', 'gluecard41')
glucose3    = ('g3', 'g30', 'glucose3', 'glucose30')
glucose4    = ('g4', 'g41', 'glucose4', 'glucose41')
glucose42   = ('g42', 'g421', 'glucose42', 'glucose421')
lingeling   = ('lgl', 'lingeling')
maplechrono = ('mcb', 'chrono', 'maplechrono')
maplecm     = ('mcm', 'maplecm')
maplesat    = ('mpl', 'maple', 'maplesat')
mergesat3   = ('mg3', 'mgs3', 'mergesat3', 'mergesat30')
minicard    = ('mc', 'mcard', 'minicard')
minisat22   = ('m22', 'msat22', 'minisat22')
minisatgh   = ('mgh', 'msat-gh', 'minisat-gh')
```

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

1.2 Supplementary `examples` package

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

List of classes

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

Module description

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

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

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

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

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

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

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

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

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

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

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

Module details

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

A non-incremental implementation of the FM (Fu&Malik, or WMSU1) algorithm. The algorithm (see details in[Page 72, 5]) is *core-guided*, i.e. it solves maximum satisfiability with a series of unsatisfiability oracle calls, each producing an unsatisfiable core. The clauses involved in an unsatisfiable core are *relaxed* and a new AtMost1 constraint on the corresponding *relaxation variables* is added to the formula. The process gets a bit more sophisticated in the case of weighted formulas because of the *clause weight splitting* technique.

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

Parameters

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

_compute()

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

compute()

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

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

delete()

Explicit destructor of the internal SAT oracle.

init(*with_soft=True*)

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

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

oracle_time()

Method for calculating and reporting the total SAT solving time.

reinit()

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

relax_core()

Relax and bound the core.

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

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

remove_unit_core()

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

split_core(*minw*)

Split clauses in the core whenever necessary.

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

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

treat_core()

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

After the core is extracted, its *minimum weight* `minw` is computed, i.e. it is the minimum weight among the weights of all soft clauses involved in the core (see[Page 72, 5]). Note that the cost of the MaxSAT solution is incremented by `minw`.

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

1.2.2 Hard formula generator (pysat.examples.genhard)

List of classes

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

Module description

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

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

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

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

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

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

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

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

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

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

Module details

class examples.genhard.**CB**(**args*, ***kwargs*)

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

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

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

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

Parameters

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

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

class examples.genhard.**GT**(**args*, ***kwargs*)

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

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

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

Parameters

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

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

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

class examples.genhard.**PAR**(**args*, ***kwargs*)

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

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

The parity principle is known to be hard for resolution[Page 75, 4].

Parameters

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

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

class examples.genhard.**PHP**(**args*, ***kwargs*)

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

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

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

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

Parameters

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

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

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

List of classes

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

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

Module description

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

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

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

Note that this implementation additionally supports *pure* SAT-based minimal hitting set enumeration with the use of preferred variable polarity setting following the approach of[7].

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

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

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

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

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

(continues on next page)

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

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

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

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

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

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

[7] Enrico Giunchiglia, Marco Maratea. *Solving Optimization Problems with DLL*. ECAI 2006. pp. 377-381

(continued from previous page)

```
...     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[8] algorithm for MaxSAT:

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

Module details

class examples.hitman.**Atom**(*obj*, *sign=True*)

Atoms are elementary (signed) objects necessary when dealing with hitting sets subject to hard constraints.

class examples.hitman.**Hitman**(*bootstrap_with=[]*, *weights=None*, *subject_to=[]*, *solver='g3'*, *htype='sorted'*, *mxs_adapt=False*, *mxs_exhaust=False*, *mxs_minz=False*, *mxs_trim=0*, *mcs_usecld=False*)

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

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

enumerated in an arbitrary order, i.e. *unsorted*. Additionally, Hitman supports pure SAT-based minimal hitting set enumeration with the use of polarity preferences.

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. Finally, value `'sat'` can be given, in which case minimal hitting set enumeration will performed by means of a SAT solver (can be either MiniSat-GH, or Lingeling, or CaDiCaL 153) with polarity setting.

In either case, unless pure SAT-based hitting set enumeration is selected, 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). For SAT-based enumeration, MinisatGH is used as an underlying SAT solver.

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()*.

Another optional parameter `subject_to` can be used to specify arbitrary hard constraints that must be respected when computing hitting sets of the given sets. Note that `subject_to` should be an iterable containing pure clauses and/or native AtMostK constraints. Note that native cardinality constraints supported only by MiniCard-like solvers. Finally, note that these hard constraints must be defined over the set of signed atomic objects, i.e. instances of class *Atom*.

A few other optional parameters include the possible options for RC2 as well as for LBX- and MCSls-like MCS enumerators that control the behaviour of the underlying solvers.

Parameters

- **bootstrap_with** (*iterable(iterable(obj))*) – input set of sets to hit
- **weights** (*dict(obj)*) – a mapping from objects to their weights (if weighted)
- **subject_to** (*iterable(iterable(Atom))*) – hard constraints (either clauses or native AtMostK constraints)
- **solver** (*str*) – name of SAT solver
- **htype** (*str*) – enumerator type
- **mxs_adapt** (*bool*) – detect and process AtMost1 constraints in RC2
- **mxs_exhaust** (*bool*) – apply unsatisfiable core exhaustion in RC2
- **mxs_minz** (*bool*) – apply heuristic core minimization in RC2
- **mxs_trim** (*int*) – trim unsatisfiable cores at most this number of times
- **mcs_usecld** (*bool*) – use clause-D heuristic in the MCS enumerator

add_hard(*clause*, *weights=None*)

Add a hard constraint, which can be either a pure clause or an AtMostK constraint.

Note that an optional parameter that can be passed to this method is `weights`, which contains a mapping the objects under question into weights. Also note that the weight of an object must not change from one call of *hit()* to another.

Parameters

- **clause** (*iterable(obj)*) – hard constraint (either a clause or a native AtMostK constraint)
- **weights** (*dict(obj)*) – a mapping from objects to weights

block(*to_block*, *weights=None*)

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.

Note that an optional parameter that can be passed to this method is `weights`, which contains a mapping the objects under question into weights. Also note that the weight of an object must not change from one call of *hit()* to another.

Parameters

- **to_block** (*iterable(obj)*) – a set to block
- **weights** (*dict(obj)*) – a mapping from objects to weights

delete()

Explicit destructor of the internal hitting set oracle.

enumerate()

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

Return type list(obj)

get()

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

Return type list(obj)

hit(*to_hit*, *weights=None*)

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.

Note that an optional parameter that can be passed to this method is `weights`, which contains a mapping the objects under question into weights. Also note that the weight of an object must not change from one call of *hit()* to another.

Parameters

- **to_hit** (*iterable(obj)*) – a new set to hit
- **weights** (*dict(obj)*) – a mapping from objects to weights

init(*bootstrap_with*, *weights=None*, *subject_to=[]*)

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.

An additional optional parameter is `weights`, which can be used to specify non-unit weights for the target objects in the sets to hit. This only works if `'sorted'` enumeration of hitting sets is applied.

Another optional parameter is available, namely, `subject_to`. It can be used to specify arbitrary hard constraints that must be respected when computing hitting sets of the given sets. Note that `subject_to` should be an iterable containing pure clauses and/or native AtMostK constraints. Finally, note that these hard constraints must be defined over the set of signed atomic objects, i.e. instances of class *Atom*.

Parameters

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

- **weights** (*dict(obj)*) – weights of the objects in case the problem is weighted
- **subject_to** (*iterable(iterable(Atom))*) – hard constraints (either clauses or native AtMostK constraints)

oracle_time()

Report the total SAT solving time.

switch_phase()

If a pure SAT-based hitting set enumeration is used, it is possible to instruct it to switch from enumerating target sets to enumerating dual sets, by polarity switching. This is what this method enables a user to do.

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

List of classes

LBX	LBX-like algorithm for computing MCSes.

Module description

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

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

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

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

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

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

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

Module details

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

LBX-like algorithm for computing MCSes. Given an unsatisfiable partial CNF formula, i.e. formula in the *WCNF* format, this class can be used to compute a given number of MCSes of the formula. The implementation follows the LBX algorithm description in[Page 82, 1]. It can use any SAT solver available in PySAT. Additionally, the "clause D" heuristic can be used when enumerating MCSes.

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

Parameters

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

_compute()

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

The method is essentially a simple loop going over all literals unsatisfied by the previous model, i.e. the literals of `self.setd` and checking which literals can be satisfied. This process can be seen a refinement of the over-approximation of the MCS. The algorithm follows the pseudo-code of the LBX algorithm presented in[Page 82, 1].

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

_filter_satisfied(*update_setd=False*)

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

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

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

_map_extlit(*l*)

Map an external variable to an internal one if necessary.

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

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

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

Return type int

_satisfied(*cl*, *model*)

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

Parameters

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

Return type bool

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

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

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

Also note that besides pure clauses, the method can also expect native cardinality constraints represented as a pair `(lits, bound)`. Only hard cardinality constraints can be added.

Parameters

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

block(*mcs*)

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

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

compute(*enable=[]*)

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

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

An optional input parameter is `enable`, which represents a sequence (normally a list) of soft clause indices that a user would prefer to enable/satisfy. Note that this may result in an unsatisfiable oracle call, in which case `None` will be reported as solution. Also, the smallest clause index is assumed to be 1.

Parameters **enable** (*iterable(int)*) – a sequence of clause ids to enable

Return type list(int)

delete()

Explicit destructor of the internal SAT oracle.

do_cld_check(*cld*)

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

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

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

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

enumerate()

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

oracle_time()

Report the total SAT solving time.

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

List of classes

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

Module description

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

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

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

(continues on next page)

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

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

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

(continued from previous page)

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

$ lsu.py -s glucose3 -m -v formula.wcnf.xz
c formula: 3 vars, 3 hard, 3 soft
o 2
s OPTIMUM FOUND
v -1 -2 3 0
c oracle time: 0.0000
```

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

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

Module details

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

Linear SAT-UNSAT algorithm for MaxSAT[?]. The algorithm can be seen as a series of satisfiability oracle calls refining an upper bound on the MaxSAT cost, followed by one unsatisfiability call, which stops the algorithm. The implementation encodes the sum of all selector literals using the *iterative totalizer encoding*[Page 85, 2]. At every iteration, the upper bound on the cost is reduced and enforced by adding the corresponding unit size clause to the working formula. No clauses are removed during the execution of the algorithm. As a result, the SAT oracle is used incrementally.

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

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

Parameters

- **formula** (*WCNF*) – input MaxSAT formula
- **solver** (*str*) – name of SAT solver
- **incr** (*bool*) – enable incremental mode of Glucose
- **expect_interrupt** (*bool*) – whether or not an *interrupt()* call is expected

- **verbose** (*int*) – verbosity level

_assert_lt(*cost*)

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

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

_get_model_cost(*formula*, *model*)

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

Parameters

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

Return type int

_init(*formula*)

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

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

clear_interrupt()

Clears an interruption.

delete()

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

found_optimum()

Checks if the optimum solution was found in a prior call to *solve()*.

Return type bool

get_model()

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

Return type list(int)

interrupt()

Interrupt the current execution of LSU's *solve()* method. Can be used to enforce time limits using timer objects. The interrupt must be cleared before running the LSU algorithm again (see *clear_interrupt()*).

oracle_time()

Method for calculating and reporting the total SAT solving time.

solve()

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

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

Return type bool

class examples.lsu.**LSUPlus**(*formula*, *solver='g4'*, *incr=False*, *expect_interrupt=False*, *verbose=0*)

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

Parameters

- **formula** (*WCNFPlus*) – input MaxSAT formula in WCNF+ format
- **expect_interrupt** (*bool*) – whether or not an interrupt() call is expected
- **verbose** (*int*) – verbosity level

_assert_lt(*cost*)

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

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

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

List of classes

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

Module description

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

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

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

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

(continues on next page)

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

(continued from previous page)

```
c cost: 2
c oracle time: 0.0002
```

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

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

Module details

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

Algorithm BLS for computing MCSes, augmented with "clause D" calls. Given an unsatisfiable partial CNF formula, i.e. formula in the *WCNF* format, this class can be used to compute a given number of MCSes of the formula. The implementation follows the description of the basic linear search (BLS) algorithm description in[Page 88, 1]. It can use any SAT solver available in PySAT. Additionally, the "clause D" heuristic can be used when enumerating MCSes.

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

Parameters

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

_compute()

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

The method is essentially a simple loop going over all literals unsatisfied by the previous model, i.e. the literals of `self.setd` and checking which literals can be satisfied. This process can be seen a refinement of the over-approximation of the MCS. The algorithm follows the pseudo-code of the BLS algorithm presented in[Page 88, 1].

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

_map_extlit(*l*)

Map an external variable to an internal one if necessary.

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

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

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

Return type int

_overapprox()

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

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

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

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

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

Also note that besides pure clauses, the method can also expect native cardinality constraints represented as a pair `(lits, bound)`. Only hard cardinality constraints can be added.

Parameters

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

block(*mcs*)

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

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

compute(*enable=[]*)

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

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

An optional input parameter is `enable`, which represents a sequence (normally a list) of soft clause indices that a user would prefer to enable/satisfy. Note that this may result in an unsatisfiable oracle call, in which case `None` will be reported as solution. Also, the smallest clause index is assumed to be 1.

Parameters **enable** (*iterable(int)*) – a sequence of clause ids to enable

Return type list(int)

delete()

Explicit destructor of the internal SAT oracle.

do_cld_check(*cld*)

Do the "clause D" check. This method receives a list of literals, which serves a "clause D"[Page 88, 1], and checks whether the formula conjoined with D is satisfiable.

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

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

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

enumerate()

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

oracle_time()

Report the total SAT solving time.

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

List of classes

enumerate_models	Enumeration procedure.

Module description

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

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

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

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

Module details

examples.models.**enumerate_models**(*formula*, *to_enum*, *solver*, *warm=False*)

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

Parameters

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

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

List of classes

MUSX	MUS eXtractor using the deletion-based algorithm.

Module description

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

The following extraction procedure is implemented:

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

i = 0

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

return assump
```

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

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

(continues on next page)

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

(continued from previous page)

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

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

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

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

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

Module details

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

MUS eXtractor using the deletion-based algorithm. The algorithm is described in[Page 92, 1] (also see the module description above). Essentially, the algorithm can be seen as an iterative process, which tries to remove one soft clause at a time and check whether the remaining set of soft clauses is still unsatisfiable together with the hard clauses.

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

Parameters

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

_compute(*approx*)

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

Soft clauses are (de)activated using the standard MiniSat-like assumptions interface[2]. Each soft clause c is augmented with a selector literal s, e.g. $(c) \leftarrow (c \vee \neg s)$. As a result, clause c can be activated by assuming

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

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.

1.2.9 OptUx optimal MUS enumerator (pysat.examples.optux)

List of classes

OptUx	A simple Python version of the implicit hitting set based optimal MUS extractor and enumerator.

Module description

An implementation of an extractor of a smallest size minimal unsatisfiable subset (smallest MUS, or SMUS)[1][2][3][4] and enumerator of SMUSes based on *implicit hitting set enumeration*[?]. This implementation tries to replicate the well-known SMUS extractor Forqes[?]. In contrast to Forqes, this implementation supports not only plain DIMACS *CNF* formulas but also weighted *WCNF* formulas. As a result, the tool is able to compute and enumerate *optimal* MUSes in case of weighted formulas. On the other hand, this prototype lacks a number of command-line options used in Forqes and so it may be less efficient compared to Forqes but the performance difference should not be significant.

The file provides a class *OptUx*, which is the basic implementation of the algorithm. It can be applied to any formula in the *CNF* or *WCNF* format.

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

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

(continues on next page)

[1] Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, Joao Marques-Silva. *Smallest MUS Extraction with Minimal Hitting Set Dualization*. CP 2015. pp. 173-182

[2] Mark H. Liffiton, Maher N. Mneimneh, Ines Lynce, Zaher S. Andraus, Joao Marques-Silva, Karem A. Sakallah. *A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas*. Constraints An Int. J. 14(4). 2009. pp. 415-442

[3] Alexey Ignatiev, Mikolas Janota, Joao Marques-Silva. *Quantified Maximum Satisfiability: A Core-Guided Approach*. SAT 2013. pp. 250-266

[4] Alexey Ignatiev, Mikolas Janota, Joao Marques-Silva. *Quantified maximum satisfiability*. Constraints An Int. J. 21(2). 2016. pp. 277-302

(continued from previous page)

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

$ optux.py -vvv formula.wcnf.xz
c mcs: 1 2 0
c mcses: 0 unit, 1 disj
c mus: 1 2 0
c cost: 2
c oracle time: 0.0001
```

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

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

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

Module details

class examples.optux.**OptUx**(*formula*, *solver='g3'*, *adapt=False*, *cover=None*, *dcalls=False*, *exhaust=False*, *minz=False*, *puresat=False*, *unsorted=False*, *trim=False*, *verbose=0*)

A simple Python version of the implicit hitting set based optimal MUS extractor and enumerator. Given a (weighted) (partial) CNF formula, i.e. formula in the *WCNF* format, this class can be used to compute a given number of optimal MUS (starting from the *best* one) of the input formula. *OptUx* roughly follows the implementation of Forqes[?] but lacks a few additional heuristics, which however aren't applied in Forqes by default.

As a result, OptUx applies exhaustive *disjoint* minimal correction subset (MCS) enumeration[?,?,?,Page 94, 4] with the incremental use of RC2[5] as an underlying MaxSAT solver. Once disjoint MCSes are enumerated, they are used to bootstrap a hitting set solver. This implementation uses *Hitman* as a hitting set solver, which is again based on RC2.

Note that in the main implicit hitting enumeration loop of the algorithm, OptUx follows Forqes in that it does not reduce correction subsets detected to minimal correction subsets. As a result, correction subsets computed in the main loop are added to `Hitman` *unreduced*.

OptUx can use any SAT solver available in PySAT. The default SAT solver to use is `g3`, which stands for Glucose 3[6] (see *SolverNames*). Boolean parameters `adapt`, `exhaust`, and `minz` control whether or not the underlying *RC2* oracles should apply detection and adaptation of intrinsic AtMost1 constraints, core exhaustion, and core reduction. Also, unsatisfiable cores can be trimmed if the `trim` parameter is set to a non-zero integer. Finally, verbosity level can be set using the `verbose` parameter.

[5] Alexey Ignatiev, Antonio Morgado, Joao Marques-Silva. *RC2: an Efficient MaxSAT Solver*. J. Satisf. Boolean Model. Comput. 11(1). 2019. pp. 53-64

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

Two additional optional parameters `unsorted` and `dcalls` can be used to instruct the tool to enumerate MUSes in the unsorted fashion, i.e. optimal MUSes are not guaranteed to go first. For this, *OptUx* applies LBX-like MCS enumeration (it uses *LBX* directly). Parameter `dcalls` can be applied to instruct the underlying MCS enumerator to apply clause D oracle calls.

Another optional paramater `puresat` can be used to instruct OptUx to run a purely SAT-based minimal hitting set enumerator, following the ideas of[7]. The value of `puresat` can be either `False`, meaning that no pure SAT enumeration is to be done or be equal to `'mgh'`, `'cd15'`, or `'lgl'` - these are the solvers that support *hard* phase setting, i.e. user preferences will not be overwritten by the *phase saving* heuristic[8].

Finally, one more optional input parameter `cover` is to be used when exhaustive enumeration of MUSes is not necessary and the tool can stop as soon as a given formula is covered by the set of currently computed MUSes. This can be made to work if the soft clauses of `formula` are of size 1.

Parameters

- **formula** (*WCNFPlus*) – (weighted) (partial) CNFPlus formula
- **solver** (*str*) – SAT oracle name
- **adapt** (*bool*) – detect and adapt intrinsic AtMost1 constraints
- **cover** (*CNFPlus*) – CNFPlus formula to cover when doing MUS enumeration
- **dcalls** (*bool*) – apply clause D oracle calls (for unsorted enumeration only)
- **exhaust** (*bool*) – do core exhaustion
- **minz** (*bool*) – do heuristic core reduction
- **puresat** (*str*) – use pure SAT-based hitting set enumeration
- **unsorted** (*bool*) – apply unsorted MUS enumeration
- **trim** (*int*) – do core trimming at most this number of times
- **verbose** (*int*) – verbosity level

_disjoint(*formula*, *solver*, *adapt*, *exhaust*, *minz*, *trim*)

This method constitutes the preliminary step of the implicit hitting set paradigm of Forqes. Namely, it enumerates all the disjoint *minimal correction subsets* (MCSes) of the formula, which will be later used to bootstrap the hitting set solver.

Note that the MaxSAT solver in use is *RC2*. As a result, all the input parameters of the method, namely, `formula`, `solver`, `adapt`, *exhaust* `, `minz`, and `trim` - represent the input and the options for the RC2 solver.

Parameters

- **formula** (*WCNF*) – input formula
- **solver** (*str*) – SAT solver name
- **adapt** (*bool*) – detect and adapt AtMost1 constraints
- **exhaust** (*bool*) – exhaust unsatisfiable cores
- **minz** (*bool*) – apply heuristic core minimization
- **trim** (*int*) – trim unsatisfiable cores at most this number of times

[7] Enrico Giunchiglia, Marco Maratea. *Solving Optimization Problems with DLL*. ECAI 2006. pp. 377-381
[8] Knot Pipatsrisawat, Adnan Darwiche. *A Lightweight Component Caching Scheme for Satisfiability Solvers*. SAT 2007. pp. 294-299

_process_soft(*formula*)

The method is for processing the soft clauses of the input formula. Concretely, it checks which soft clauses must be relaxed by a unique selector literal and applies the relaxation.

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

compute()

This method implements the main look of the implicit hitting set paradigm of Forqes to compute a best-cost MUS. The result MUS is returned as a list of integers, each representing a soft clause index.

Return type list(int)

delete()

Explicit destructor of the internal hitting set and SAT oracles.

enumerate()

This is generator method iterating through MUSes and enumerating them until the formula has no more MUSes, or a user decides to stop the process.

Return type list(int)

oracle_time()

This method computes and returns the total SAT solving time involved.

Return type float

1.2.10 RC2 MaxSAT solver (pysat.examples.rc2)

List of classes

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

Module description

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

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

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

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

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

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

[4] Alexey Ignatiev, António Morgado, Joao Marques-Silva. *RC2: An Efficient MaxSAT Solver*. MaxSAT Evaluation 2018. JSAT 11. 2019. pp. 53-64

[5] Joao Marques-Silva, Josep Argelich, Ana Graça, Inês Lynce. *Boolean lexicographic optimization: algorithms & applications*. Ann. Math. Artif. Intell. 62(3-4). 2011. pp. 317-343

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

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

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

$ rc2.py -vv formula.wcnf.xz
c formula: 3 vars, 3 hard, 3 soft
c cost: 1; core sz: 2; soft sz: 2
c cost: 2; core sz: 2; soft sz: 1
s OPTIMUM FOUND
o 2
v -1 -2 3
c oracle time: 0.0001
```

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

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

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

Module details

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

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

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

RC2 can use any SAT solver available in PySAT. The default SAT solver to use is `g3` (see *SolverNames*). Additionally, if Glucose is chosen, the `incr` parameter controls whether to use the incremental mode of Glucose[7] (turned off by default). Boolean parameters `adapt`, `exhaust`, and `minz` control whether or to apply detection and adaptation of intrinsic AtMost1 constraints, core exhaustion, and core reduction. Unsatisfiable cores can be trimmed if the `trim` parameter is set to a non-zero integer. Finally, verbosity level can be set using the `verbose` parameter.

Parameters

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

_map_extlit(*l*)

Map an external variable to an internal one if necessary.

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

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

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

Return type int

adapt_am1()

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

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

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

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

The method for adding a new hard of soft clause to the problem formula. Although the input formula is to be specified as an argument of the constructor of *RC2*, adding clauses may be helpful when *enumerating* MaxSAT solutions of the formula. This way, the clauses are added incrementally, i.e. *on the fly*.

The clause to add can be any iterable over integer literals. The additional integer parameter `weight` can be set to meaning the the clause being added is soft having the corresponding weight (note that parameter `weight` is set to `None` by default meaning that the clause is hard).

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

Also note that besides pure clauses, the method can also expect native cardinality constraints represented as a pair `(lits, bound)`. Only hard cardinality constraints can be added.

Parameters

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

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

compute()

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

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

Returns a MaxSAT model

Return type list(int)

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

(continues on next page)

(continued from previous page)

```
>>> model = rc2.compute()
>>> print(model)
[-1, -2, 3]
>>> print(rc2.cost)
2
>>> rc2.delete()
```

compute_()

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

Return type bool

create_sum(*bound=1*)

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

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

Return type *ITotalizer*

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

delete()

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

enumerate(*block=0*)

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

An optional parameter can be used to enforce computation of MaxSAT models corresponding to different maximal satisfiable subsets (MSSes) or minimal correction subsets (MCSes). To block MSSes, one should set the `block` parameter to `1`. To block MCSes, set it to `-1`. By the default (for blocking MaxSAT models), `block` is set to `0`.

Parameters **block** (*int*) – preferred way to block solutions when enumerating

Returns a MaxSAT model

Return type list(int)

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

(continues on next page)

(continued from previous page)

```
>>> for model in rc2.enumerate():
...     print(model, rc2.cost)
[-1, -2, 3] 2
[1, -2, -3] 2
[-1, 2, -3] 2
[-1, -2, -3] 3
>>> rc2.delete()
```

exhaust_core(*tobj*)

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

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

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

filter_assumps()

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

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

get_core()

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

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

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

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

Optional input parameter `incr` (`False` by default) regulates whether or not Glucose's incremental mode[Page 99, 7] is turned on.

Parameters

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

minimize_core()

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

The idea is to try to deactivate soft clauses of the unsatisfiable core one by one while checking if the remaining soft clauses together with the hard part of the formula are unsatisfiable. Clauses that are necessary for

preserving unsatisfiability comprise an MUS of the input formula (it is contained in the given unsatisfiable core) and are reported as a result of the procedure.

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

oracle_time()

Report the total SAT solving time.

process_am1(*am1*)

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

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

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

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

process_core()

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

process_sels()

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

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

process_sums()

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

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

set_bound(*tobj*, *rhs*, *weight=None*)

Given a totalizer sum, its right-hand side to be enforced, and a weight, the method creates a new sum assumption literal, which will be used in the following SAT oracle calls. If `weight` is left unspecified, the current core's weight, i.e. `self.minw`, is used.

Parameters

- **tobj** (*ITotalizer*) – totalizer sum
- **rhs** (*int*) – right-hand side
- **weight** (*int*) – numeric weight of the assumption

trim_core()

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

update_sum(*assump*)

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

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

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

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

Return type *ITotalizer*, int

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

class examples.rc2.**RC2Stratified**(*formula*, *solver='g3'*, *adapt=False*, *blo='div'*, *exhaust=False*, *incr=False*, *minz=False*, *nohard=False*, *trim=0*, *verbose=0*)

RC2 augmented with BLO and stratification techniques. Although class *RC2* can deal with weighted formulas, there are situations when it is necessary to apply additional heuristics to improve the performance of the solver on weighted MaxSAT formulas. This class extends capabilities of *RC2* with two heuristics, namely

1. Boolean lexicographic optimization (BLO)[?]
2. diversity-based stratification[?]
3. cluster-based stratification

To specify which heuristics to apply, a user can assign the `blo` parameter to one of the values (by default it is set to `'div'`):

- `'basic'` ('BLO' only)
- `div` ('BLO' + diversity-based stratification)
- `cluster` ('BLO' + cluster-based stratification)
- `full` ('BLO' + diversity- + cluster-based stratification)

Except for the aforementioned additional techniques, every other component of the solver remains as in the base class *RC2*. Therefore, a user is referred to the documentation of *RC2* for details.

activate_clauses(*beg*)

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

compute()

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

finish_level()

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

init_wstr()

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

next_level()

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

- partial BLO condition
- diversity-based stratification condition
- cluster-based stratification condition

If any of these holds, the loop stops.

process_am1(*am1*)

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

process_sels()

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

process_sums()

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

1.3 Supplementary `allies` package

This module provides interface to a list of external tools useful in practical SAT-based problem solving. Although only ApproxMCv4 is currently present here, the list of tools will grow.

1.3.1 ApproxMC model counter (`pysat.allies.approxmc`)

List of classes

Counter	A wrapper for ApproxMC, a state-of-the-art *approximate* model counter.

Module description

This module provides interface to ApproxMCv4, a state-of-the-art *approximate* model counter utilising an improved version of CryptoMiniSat to give approximate model counts to problems of size and complexity that are out of reach for earlier approximate model counters. The original work on ApproxMCv4 has been published in[1] and[2].

Note that to be functional, the module requires package `pyapproxmc` to be installed:

```
$ pip install pyapproxmc
```

[1] Mate Soos, Kuldeep S. Meel. *BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting*. AAAI 2019. pp. 1592-1599

[2] Mate Soos, Stephan Gocht, Kuldeep S. Meel. *Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling*. CAV 2020. pp. 463-484

The interface gives access to *Counter*, which expects a formula in *CNF* as input. Given a few additional (optional) arguments, including a random seed, *tolerance factor* ε, and *confidence* δ, the class can be used to get an approximate number of models of the formula, subject to the given tolerance factor and confidence parameter.

Namely, given a CNF formula $\mathcal{F}$ with $\#\mathcal{F}$ as the exact number of models, and parameters $\varepsilon \in (0, 1]$ and $\delta \in [0, 1)$, the counter computes and reports a value C, which is an approximate number of models of $\mathcal{F}$, such that $\Pr\left[\frac{1}{1+\varepsilon}\#\mathcal{F} \leq C \leq (1+\varepsilon)\#\mathcal{F}\right] \geq 1 - \delta$.

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

```
$ xzcat formula.cnf.xz
p cnf 20 2
1 2 3 0
3 20 0

$ approxmc.py -p 1,2,3-9 formula.cnf.xz
s mc 448
```

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

```
>>> from pysat.allies.approxmc import Counter
>>> from pysat.formula import CNF
>>>
>>> cnf = CNF(from_file='formula.cnf.xz')
>>>
>>> with Counter(cnf) as counter:
...     print(counter.counter(projection=range(1, 10))
448
```

As can be seen in the above example, besides model counting across all the variables in a given input formula, the counter supports *projected* model counting, i.e. when one needs to approximate the number of models with respect to a given list of variables rather than with respect to all variables appearing in the formula. This feature comes in handy when the formula is obtained, for example, through Tseitin transformation[3] with a number of auxiliary variables introduced.

Module details

class allies.approxmc.**Counter**(*formula=None*, *seed=1*, *epsilon=0.8*, *delta=0.2*, *verbose=0*)

A wrapper for ApproxMC, a state-of-the-art *approximate* model counter. Given a formula in *CNF*, this class can be used to get an approximate number of models of the formula, subject to *tolerance factor* `epsilon` and *confidence parameter* `delta`.

Namely, given a CNF formula $\mathcal{F}$ and parameters $\varepsilon \in (0, 1]$ and $\delta \in [0, 1)$, the counter computes and reports a value C such that $\Pr\left[\frac{1}{1+\varepsilon}\#\mathcal{F} \leq C \leq (1+\varepsilon)\#\mathcal{F}\right] \geq 1 - \delta$. Here, $\#\mathcal{F}$ denotes the exact model count for formula $\mathcal{F}$.

The `formula` argument can be left unspecified at this stage. In this case, a user is expected to add all the relevant clauses using *add_clause()*.

An additional parameter a user may want to specify is integer `seed` used by ApproxMC. The value of `seed` is set to 1 by default.

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

Parameters

- **formula** (*CNF*) – CNF formula
- **seed** (*int*) – integer seed value
- **epsilon** (*float*) – tolerance factor
- **delta** (*float*) – confidence parameter
- **verbose** (*int*) – verbosity level

```
>>> from pysat.allies.approxmc import Counter
>>> from pysat.formula import CNF
>>>
>>> cnf = CNF(from_file='some-formula.cnf')
>>> with Counter(formula=cnf, epsilon=0.1, delta=0.9) as counter:
...     num = counter.count() # an approximate number of models
```

add_clause(*clause*)

The method for adding a clause to the problem formula. Although the input formula can be specified as an argument of the constructor of *Counter*, adding clauses may also be helpful afterwards, *on the fly*.

The clause to add can be any iterable over integer literals.

Parameters **clause** (*iterable(int)*) – a clause to add

```
>>> from pysat.allies.approxmc import Counter
>>>
>>> with Counter() as counter:
...     counter.add_clause(range(1, 4))
...     counter.add_clause([3, 20])
...
...     print(counter.count())
720896
```

count(*projection=None*)

Given the formula provided by the user either in the constructor of *Counter* or through a series of calls to *add_clause()*, this method runs the ApproxMC counter with the specified values of tolerance ε and confidence δ parameters, as well as the random `seed` value, and returns the number of models estimated.

A user may specify an argument `projection`, which is a list of integers specifying the variables with respect to which projected model counting should be performed. If `projection` is left as `None`, approximate model counting is performed wrt. all the variables of the input formula.

Parameters **projection** (*list(int)*) – variables to project on

```
>>> from pysat.allies.approxmc import Counter
>>> from pysat.card import CardEnc, EncType
>>>
>>> # cardinality constraint with auxiliary variables
>>> # there are exactly 70 models for the constraint
>>> # over the 8 original variables
>>> cnf = CardEnc.equals(lits=range(1, 9), bound=4, encoding=EncType.cardnetwrk)
>>>
>>> with Counter(formula=cnf, epsilon=0.05, delta=0.95) as counter:
...     print(counter.count())
123840
```

(continues on next page)

(continued from previous page)

```
>>>
>>> with Counter(formula=cnf, epsilon=0.05, delta=0.95) as counter:
...     print(counter.count(projection=range(1, 8)))
70
```

delete()

Explicit destructor of the internal Counter oracle. Delete the actual counter object and sets it to `None`.

1.3.2 UniGen almost-uniform sampler (`pysat.allies.unigen`)

List of classes

Sampler	A wrapper for UniGen3, a state-of-the-art almost-uniform sampler.

Module description

This module provides interface to UniGen3, a state-of-the-art almost-uniform sampler utilising an improved version of CryptoMiniSat to handle problems of size and complexity that were not possible before. . The original work on UniGen3 has been published in[1],[2], and[3].

Note that to be functional, the module requires package `pyunigen` to be installed:

```
$ pip install pyunigen
```

The interface gives access to *Sampler*, which expects a formula in *CNF* as input. Given a few additional (optional) arguments, including a random seed, *tolerance factor* ε, *confidence* δ (to be used by ApproxMC), and *uniformity parameter* κ, the class can be used to get apply almost-uniform sampling and to obtain a requested number of samples as a result, subject to the given tolerance factor and confidence parameter.

Namely, given a CNF formula $\mathcal{F}$ with the set of satisfying assignments (or models) denoted by $sol(\mathcal{F})$ and parameter $\varepsilon \in (0, 1]$, a uniform sampler outputs a model $y \in sol(\mathcal{F})$ such that $\Pr[y \text{ is output}] = \frac{1}{|sol(\mathcal{F})|}$. Almost-uniform sampling relaxes the uniformity guarantee and ensures that $\frac{1}{(1+\varepsilon)|sol(\mathcal{F})|} \leq \Pr[y \text{ is output}] \leq \frac{1+\varepsilon}{|sol(\mathcal{F})|}$.

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

```
$ xzcat formula.cnf.xz
p cnf 6 2
1 5 0
1 6 0

$ unigen.py -n 4 formula.cnf.xz
v +1 -2 +3 -4 -5 -6 0
v +1 +2 +3 -4 +5 +6 0
```

(continues on next page)

[1] Supratik Chakraborty, Kuldeep S. Meel, Moshe Y. Vardi. *Balancing Scalability and Uniformity in SAT Witness Generator*. DAC 2014. pp. 60:1-60:6

[2] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, Moshe Y. Vardi. *On Parallel Scalable Uniform SAT Witness Generation*. TACAS 2015. pp. 304-319

[3] Mate Soos, Stephan Gocht, Kuldeep S. Meel. *Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling*. CAV 2020. pp. 463-484

(continued from previous page)

```
v +1 -2 -3 -4 +5 -6 0
v -1 -2 -3 -4 +5 +6 0
```

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

```
>>> from pysat.allies.unigen import Sampler
>>> from pysat.formula import CNF
>>>
>>> cnf = CNF(from_file='formula.cnf.xz')
>>>
>>> with Sampler(cnf) as sampler:
...     print(sampler.sample(nof_samples=4, sample_over=[1, 2, 3])
[[1, 2, 3, 4, 5], [1, -2, -3, -4, -5], [1, -2, -3, -4, 5], [1, 2, -3, 4, 5]]
```

As can be seen in the above example, sampling can be done over a user-defined set of variables (rather than the complete set of variables).

Module details

class allies.unigen.**Sampler**(*formula=None*, *seed=1*, *epsilon=0.8*, *delta=0.2*, *kappa=0.638*, *verbose=0*)

A wrapper for UniGen3, a state-of-the-art almost-uniform sampler. Given a formula in *CNF*, this class can be used to apply almost-uniform sampling of the formula's models, subject to a few input parameters.

The class initialiser receives a number of input arguments. The `formula` argument can be left unspecified at this stage. In this case, a user is expected to add all the relevant clauses using *add_clause()*.

Additional parameters a user may want to specify include integer `seed` (used by ApproxMC), tolerance factor `epsilon` (used by the probabilistic guarantees of almost-uniformity), confidence parameter `delta` (used by ApproxMC), and uniformity parameter `kappa` (see[Page 108, 2]).

Parameters

- **formula** (*CNF*) – CNF formula
- **seed** (*int*) – seed value
- **epsilon** (*float*) – tolerance factor
- **delta** (*float*) – confidence parameter (used by ApproxMC)
- **kappa** (*float*) – uniformity parameter
- **verbose** (*int*) – verbosity level

```
>>> from pysat.allies.unigen import Sampler
>>> from pysat.formula import CNF
>>>
>>> cnf = CNF(from_file='some-formula.cnf')
>>> with Sampler(formula=cnf, epsilon=0.1, delta=0.9) as sampler:
...     for model in sampler.sample(nof_samples=100):
...         print(model)  # printing 100 result samples
```

add_clause(*clause*)

The method for adding a clause to the problem formula. Although the input formula can be specified as an argument of the constructor of *Sampler*, adding clauses may also be helpful afterwards, *on the fly*.

The clause to add can be any iterable over integer literals.

Parameters **clause** (*iterable(int)*) – a clause to add

```
>>> from pysat.allies.unigen import Sampler
>>>
>>> with Sampler() as sampler:
...     sampler.add_clause(range(1, 4))
...     sampler.add_clause([3, 4])
...
...     print(sampler.sample(nof_samples=4))
[[1, 2, -3, 4], [-1, 2, -3, 4], [1, 2, 3, -4], [-1, 2, 3, 4]]
```

delete()

Explicit destructor of the internal Sampler oracle. Delete the actual sampler object and sets it to None.

sample(*nof_samples*, *sample_over=None*, *counts=None*)

Given the formula provided by the user either in the constructor of *Sampler* or through a series of calls to *add_clause()*, this method runs the UniGen3 sampler with the specified values of tolerance ε, confidence δ parameters, and uniformity parameter *kappa* as well as the random seed value, and outputs a requested number of samples.

A user may specify an argument sample_over, which is a list of integers specifying the variables with respect to which sampling should be performed. If sample_over is left as None, almost-uniform sampling is done wrt. all the variables of the input formula.

Finally, argument counts can be specified as a pair of integer values: *cell count* and *hash count* (in this order) used during sampling. If left undefined (None), the values are determined by ApproxMC.

Parameters

- **nof_samples** (*int*) – number of samples to output
- **sample_over** (*list(int)*) – variables to sample over
- **counts** (*[int, int]*) – cell count and hash count values

Returns a list of samples

```
>>> from pysat.allies.unigen import Sampler
>>> from pysat.card import CardEnc, EncType
>>>
>>> # cardinality constraint with auxiliary variables
>>> # there are exactly 6 models for the constraint
>>> # over the 6 original variables
>>> cnf = CardEnc.equals(lits=range(1, 5), bound=2, encoding=EncType.totalizer)
>>>
>>> with Sampler(formula=cnf, epsilon=0.05, delta=0.95) as sampler:
...     for model in sampler.sample(nof_samples=3):
...         print(model)
[1, -2, 3, -4, 5, 6, -7, -8, 9, -10, 11, -12, 13, 14, -15, 16, 17, -18, 19, -20]
[1, -2, -3, 4, 5, 6, -7, -8, 9, -10, 11, -12, 13, 14, -15, 16, 17, -18, 19, -20]
[1, 2, -3, -4, 5, 6, -7, 8, -9, -10, 11, 12, 13, 14, -15, 16, 17, 18, -19, -20]
>>>
>>> # now, sampling over the original variables
>>> with Sampler(formula=cnf, epsilon=0.05, delta=0.95) as sampler:
...     for model in sampler.sample(nof_samples=3, sample_over=range(1, 5)):
...         print(model)
[1, 2, -3, -4]
```

(continues on next page)

(continued from previous page)

```
[1, -2, 3, -4]
[-1, 2, 3, -4]
```

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X