Sat

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

Sage supports solving clauses in Conjunctive Normal Form (see Wikipedia article Conjunctive_normal_form), i.e., SAT solving, via an interface inspired by the usual DIMACS format used in SAT solving [SG09]. For example, to express that:

```
x1 OR x2 OR (NOT x3)
```

should be true, we write:

```
(1, 2, -3)
```

Warning: Variable indices **must** start at one.

CHAPTER

ONE

SOLVERS

By default, Sage solves SAT instances as an Integer Linear Program (see sage.numerical.mip), but any SAT solver supporting the DIMACS input format is easily interfaced using the *sage.sat.solvers.dimacs.DIMACS* blueprint. Sage ships with pre-written interfaces for *RSat* [RS] and *Glucose* [GL]. Furthermore, Sage provides an interface to the *CryptoMiniSat* [CMS] SAT solver which can be used interchangeably with DIMACS-based solvers. For this last solver, the optional CryptoMiniSat package must be installed, this can be accomplished by typing the following in the shell:

```
sage -i cryptominisat sagelib
```

We now show how to solve a simple SAT problem.

```
(x1 OR x2 OR x3) AND (x1 OR x2 OR (NOT x3))
```

In Sage's notation:

```
sage: solver = SAT()
sage: solver.add_clause( ( 1,  2,  3) )
sage: solver.add_clause( ( 1,  2, -3) )
sage: solver()       # random
(None, True, True, False)
```

Note: *add_clause()* creates new variables when necessary. When using CryptoMiniSat, it creates *all* variables up to the given index. Hence, adding a literal involving the variable 1000 creates up to 1000 internal variables.

DIMACS-base solvers can also be used to write DIMACS files:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: solver.add_clause( ( 1,  2,  3) )
sage: solver.add_clause( ( 1,  2, -3) )
sage: _ = solver.write()
sage: for line in open(fn).readlines():
....:     print(line)
p cnf 3 2
1 2 3 0
1 2 -3 0
```

Alternatively, there is *sage.sat.solvers.dimacs.DIMACS.clauses()*:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( ( 1,  2,  3) )
sage: solver.add_clause( ( 1,  2, -3) )
sage: solver.clauses(fn)
sage: for line in open(fn).readlines():
....:    print(line)
p cnf 3 2
1 2 3 0
1 2 -3 0
```

These files can then be passed external SAT solvers.

1.1 Details on Specific Solvers

1.1.1 Abstract SAT Solver

All SAT solvers must inherit from this class.

Note: Our SAT solver interfaces are 1-based, i.e., literals start at 1. This is consistent with the popular DIMACS format for SAT solving but not with Python's 0-based convention. However, this also allows to construct clauses using simple integers.

AUTHORS:

- Martin Albrecht (2012): first version

`sage.sat.solvers.satsolver.`**`SAT`**(*solver=None*, **args*, ***kwds*)

Return a *`SatSolver`* instance.

Through this class, one can define and solve SAT problems.

INPUT:

- `solver` (string) – select a solver. Admissible values are:
  - `"cryptominisat"` – note that the cryptominisat package must be installed.
  - `"picosat"` – note that the pycosat package must be installed.
  - `"glucose"` – note that the glucose package must be installed.
  - `"glucose-syrup"` – note that the glucose package must be installed.
  - `"LP"` – use *`SatLP`* to solve the SAT instance.
  - `None` (default) – use CryptoMiniSat if available, else PicoSAT if available, and a LP solver otherwise.

EXAMPLES:

```
sage: SAT(solver="LP")
an ILP-based SAT Solver
```

class `sage.sat.solvers.satsolver.`**`SatSolver`**

Bases: `object`

add_clause(*lits*)

Add a new clause to set of clauses.

INPUT:

- `lits` - a tuple of integers != 0

Note: If any element `e` in `lits` has `abs(e)` greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.add_clause( (1, -2 , 3) )
Traceback (most recent call last):
...
NotImplementedError
```

clauses(*filename=None*)

Return original clauses.

INPUT:

- `filename'' - if not ``None` clauses are written to `filename` in DIMACS format (default: `None`)

OUTPUT:

If `filename` is `None` then a list of `lits, is_xor, rhs` tuples is returned, where `lits` is a tuple of literals, `is_xor` is always `False` and `rhs` is always `None`.

If `filename` points to a writable file, then the list of original clauses is written to that file in DIMACS format.

EXAMPLES:

```
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.clauses()
Traceback (most recent call last):
...
NotImplementedError
```

conflict_clause()

Return conflict clause if this instance is UNSAT and the last call used assumptions.

EXAMPLES:

```
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.conflict_clause()
Traceback (most recent call last):
...
NotImplementedError
```

learnt_clauses(*unitary_only=False*)

Return learnt clauses.

INPUT:

- unitary_only - return only unitary learnt clauses (default: False)

EXAMPLES:

```
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.learnt_clauses()
Traceback (most recent call last):
...
NotImplementedError

sage: solver.learnt_clauses(unitary_only=True)
Traceback (most recent call last):
...
NotImplementedError
```

nvars()

Return the number of variables.

EXAMPLES:

```
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.nvars()
Traceback (most recent call last):
...
NotImplementedError
```

read(*filename*)

Reads DIMAC files.

Reads in DIMAC formatted lines (lazily) from a file or file object and adds the corresponding clauses into this solver instance. Note that the DIMACS format is not well specified, see http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html, http://www.satcompetition.org/2009/format-benchmarks2009.html, and http://elis.dvo.ru/~lab_11/glpk-doc/cnfsat.pdf.

The differences were summarized in the discussion on the issue github issue #16924. This method assumes the following DIMACS format:

- Any line starting with “c” is a comment
- Any line starting with “p” is a header
- Any variable 1-n can be used
- Every line containing a clause must end with a “0”

The format is extended to allow lines starting with “x” defining xor clauses, with the notation introduced in cryptominisat, see https://www.msoos.org/xor-clauses/

INPUT:

- filename - The name of a file as a string or a file object

EXAMPLES:

```
sage: from io import StringIO
sage: file_object = StringIO("c A sample .cnf file.\np cnf 3 2\n1 -3 0\n2 3 -1␣
↪0 ")
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.read(file_object)
sage: solver.clauses()
[((1, -3), False, None), ((2, 3, -1), False, None)]
```

With xor clauses:

```
sage: from io import StringIO
sage: file_object = StringIO("c A sample .cnf file with xor clauses.\np cnf 3 3\
↪n1 2 0\n3 0\nx1 2 3 0")
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat          #␣
↪optional - pycryptosat
sage: solver = CryptoMiniSat()                                          #␣
↪optional - pycryptosat
sage: solver.read(file_object)                                          #␣
↪optional - pycryptosat
sage: solver.clauses()                                                  #␣
↪optional - pycryptosat
[((1, 2), False, None), ((3,), False, None), ((1, 2, 3), True, True)]
sage: solver()                                                          #␣
↪optional - pycryptosat
(None, True, True, True)
```

var(*decision=None*)

Return a *new* variable.

INPUT:

- `decision` - is this variable a decision variable?

EXAMPLES:

```
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.var()
Traceback (most recent call last):
...
NotImplementedError
```

1.1.2 SAT-Solvers via DIMACS Files

Sage supports calling SAT solvers using the popular DIMACS format. This module implements infrastructure to make it easy to add new such interfaces and some example interfaces.

Currently, interfaces to **RSat** and **Glucose** are included by default.

Note: Our SAT solver interfaces are 1-based, i.e., literals start at 1. This is consistent with the popular DIMACS format for SAT solving but not with Pythion's 0-based convention. However, this also allows to construct clauses using simple integers.

AUTHORS:

- Martin Albrecht (2012): first version
- Sébastien Labbé (2018): adding Glucose SAT solver
- Sébastien Labbé (2023): adding Kissat SAT solver

Classes and Methods

class sage.sat.solvers.dimacs.**DIMACS**(*command=None*, *filename=None*, *verbosity=0*, ***kwds*)

Bases: SatSolver

Generic DIMACS Solver.

Note: Usually, users won't have to use this class directly but some class which inherits from this class.

__init__(*command=None*, *filename=None*, *verbosity=0*, ***kwds*)

Construct a new generic DIMACS solver.

INPUT:

- `command` - a named format string with the command to run. The string must contain {input} and may contain {output} if the solvers writes the solution to an output file. For example "sat-solver {input}" is a valid command. If `None` then the class variable `command` is used. (default: `None`)
- `filename` - a filename to write clauses to in DIMACS format, must be writable. If `None` a temporary filename is chosen automatically. (default: `None`)
- `verbosity` - a verbosity level, where zero means silent and anything else means verbose output. (default: `0`)
- `**kwds` - accepted for compatibility with other solves, ignored.

__call__(*assumptions=None*)

Solve this instance and return the parsed output.

INPUT:

- `assumptions` - ignored, accepted for compatibility with other solvers (default: `None`)

OUTPUT:

- If this instance is SAT: A tuple of length `nvars()+1` where the `i`-th entry holds an assignment for the `i`-th variables (the `0`-th entry is always `None`).
- If this instance is UNSAT: `False`

EXAMPLES:

When the problem is SAT:

```
sage: from sage.sat.solvers import RSat
sage: solver = RSat()
sage: solver.add_clause( (1, 2, 3) )
sage: solver.add_clause( (-1,) )
sage: solver.add_clause( (-2,) )
sage: solver()                              # optional - rsat
(None, False, False, True)
```

When the problem is UNSAT:

```
sage: solver = RSat()
sage: solver.add_clause((1,2))
sage: solver.add_clause((-1,2))
sage: solver.add_clause((1,-2))
sage: solver.add_clause((-1,-2))
sage: solver()                               # optional - rsat
False
```

With Glucose:

```
sage: from sage.sat.solvers.dimacs import Glucose
sage: solver = Glucose()
sage: solver.add_clause((1,2))
sage: solver.add_clause((-1,2))
sage: solver.add_clause((1,-2))
sage: solver()                              # optional - glucose
(None, True, True)
sage: solver.add_clause((-1,-2))
sage: solver()                              # optional - glucose
False
```

With GlucoseSyrup:

```
sage: from sage.sat.solvers.dimacs import GlucoseSyrup
sage: solver = GlucoseSyrup()
sage: solver.add_clause((1,2))
sage: solver.add_clause((-1,2))
sage: solver.add_clause((1,-2))
sage: solver()                             # optional - glucose
(None, True, True)
sage: solver.add_clause((-1,-2))
sage: solver()                             # optional - glucose
False
```

add_clause(*lits*)

Add a new clause to set of clauses.

INPUT:

- `lits` - a tuple of integers != 0

Note: If any element `e` in `lits` has `abs(e)` greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.var()
1
sage: solver.var(decision=True)
2
```

(continues on next page)

(continued from previous page)

```
sage: solver.add_clause( (1, -2 , 3) )
sage: solver
DIMACS Solver: ''
```

clauses(*filename=None*)

Return original clauses.

INPUT:

- `filename` - if not `None` clauses are written to `filename` in DIMACS format (default: `None`)

OUTPUT:

If `filename` is `None` then a list of `lits, is_xor, rhs` tuples is returned, where `lits` is a tuple of literals, `is_xor` is always `False` and `rhs` is always `None`.

If `filename` points to a writable file, then the list of original clauses is written to that file in DIMACS format.

EXAMPLES:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( (1, 2, 3) )
sage: solver.clauses()
[((1, 2, 3), False, None)]

sage: solver.add_clause( (1, 2, -3) )
sage: solver.clauses(fn)
sage: print(open(fn).read())
p cnf 3 2
1 2 3 0
1 2 -3 0
```

command = ''

nvars()

Return the number of variables.

EXAMPLES:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.var()
1
sage: solver.var(decision=True)
2
sage: solver.nvars()
2
```

static render_dimacs(*clauses, filename, nlits*)

Produce DIMACS file `filename` from `clauses`.

INPUT:

- `clauses` - a list of clauses, either in simple format as a list of literals or in extended format for CryptoMiniSat: a tuple of literals, `is_xor` and `rhs`.
- `filename` - the file to write to
- `nlits -- the number of literals appearing in ``clauses`

EXAMPLES:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( (1, 2, -3) )
sage: DIMACS.render_dimacs(solver.clauses(), fn, solver.nvars())
sage: print(open(fn).read())
p cnf 3 1
1 2 -3 0
```

This is equivalent to:

```
sage: solver.clauses(fn)
sage: print(open(fn).read())
p cnf 3 1
1 2 -3 0
```

This function also accepts a "simple" format:

```
sage: DIMACS.render_dimacs([ (1,2), (1,2,-3) ], fn, 3)
sage: print(open(fn).read())
p cnf 3 2
1 2 0
1 2 -3 0
```

var(*decision=None*)

Return a *new* variable.

INPUT:

- `decision` - accepted for compatibility with other solvers, ignored.

EXAMPLES:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.var()
1
```

write(*filename=None*)

Write DIMACS file.

INPUT:

- `filename` - if `None` default filename specified at initialization is used for writing to (default: `None`)

EXAMPLES:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
```

(continues on next page)

(continued from previous page)

```
sage: solver = DIMACS(filename=fn)
sage: solver.add_clause( (1, -2 , 3) )
sage: _ = solver.write()
sage: for line in open(fn).readlines():
....:     print(line)
p cnf 3 1
1 -2 3 0

sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( (1, -2 , 3) )
sage: _ = solver.write(fn)
sage: for line in open(fn).readlines():
....:      print(line)
p cnf 3 1
1 -2 3 0
```

class `sage.sat.solvers.dimacs.`**`Glucose`**(*command=None*, *filename=None*, *verbosity=0*, ***kwds*)

Bases: *DIMACS*

An instance of the Glucose solver.

For information on Glucose see: http://www.labri.fr/perso/lsimon/glucose/

EXAMPLES:

```
sage: from sage.sat.solvers import Glucose
sage: solver = Glucose()
sage: solver
DIMACS Solver: 'glucose -verb=0 -model {input}'
```

When the problem is SAT:

```
sage: from sage.sat.solvers import Glucose
sage: solver1 = Glucose()
sage: solver1.add_clause( (1, 2, 3) )
sage: solver1.add_clause( (-1,) )
sage: solver1.add_clause( (-2,) )
sage: solver1()                          # optional - glucose
(None, False, False, True)
```

When the problem is UNSAT:

```
sage: solver2 = Glucose()
sage: solver2.add_clause((1,2))
sage: solver2.add_clause((-1,2))
sage: solver2.add_clause((1,-2))
sage: solver2.add_clause((-1,-2))
sage: solver2()                          # optional - glucose
False
```

With one hundred variables:

```
sage: solver3 = Glucose()
sage: solver3.add_clause( (1, 2, 100) )
sage: solver3.add_clause( (-1,) )
sage: solver3.add_clause( (-2,) )
sage: solver3()                             # optional - glucose
(None, False, False, ..., True)
```

command = 'glucose -verb=0 -model {input}'

class sage.sat.solvers.dimacs.**GlucoseSyrup**(*command=None*, *filename=None*, *verbosity=0*, ***kwds*)

Bases: *DIMACS*

An instance of the Glucose-syrup parallel solver.

For information on Glucose see: http://www.labri.fr/perso/lsimon/glucose/

EXAMPLES:

```
sage: from sage.sat.solvers import GlucoseSyrup
sage: solver = GlucoseSyrup()
sage: solver
DIMACS Solver: 'glucose-syrup -model -verb=0 {input}'
```

When the problem is SAT:

```
sage: solver1 = GlucoseSyrup()
sage: solver1.add_clause( (1, 2, 3) )
sage: solver1.add_clause( (-1,) )
sage: solver1.add_clause( (-2,) )
sage: solver1()                             # optional - glucose
(None, False, False, True)
```

When the problem is UNSAT:

```
sage: solver2 = GlucoseSyrup()
sage: solver2.add_clause((1,2))
sage: solver2.add_clause((-1,2))
sage: solver2.add_clause((1,-2))
sage: solver2.add_clause((-1,-2))
sage: solver2()                             # optional - glucose
False
```

With one hundred variables:

```
sage: solver3 = GlucoseSyrup()
sage: solver3.add_clause( (1, 2, 100) )
sage: solver3.add_clause( (-1,) )
sage: solver3.add_clause( (-2,) )
sage: solver3()                             # optional - glucose
(None, False, False, ..., True)
```

command = 'glucose-syrup -model -verb=0 {input}'

class sage.sat.solvers.dimacs.**Kissat**(*command=None*, *filename=None*, *verbosity=0*, ***kwds*)

Bases: *DIMACS*

An instance of the Kissat SAT solver

For information on Kissat see: http://fmv.jku.at/kissat/

EXAMPLES:

```
sage: from sage.sat.solvers import Kissat
sage: solver = Kissat()
sage: solver
DIMACS Solver: 'kissat -q {input}'
```

When the problem is SAT:

```
sage: solver1 = Kissat()
sage: solver1.add_clause( (1, 2, 3) )
sage: solver1.add_clause( (-1,) )
sage: solver1.add_clause( (-2,) )
sage: solver1()                          # optional - kissat
(None, False, False, True)
```

When the problem is UNSAT:

```
sage: solver2 = Kissat()
sage: solver2.add_clause((1,2))
sage: solver2.add_clause((-1,2))
sage: solver2.add_clause((1,-2))
sage: solver2.add_clause((-1,-2))
sage: solver2()                          # optional - kissat
False
```

With one hundred variables:

```
sage: solver3 = Kissat()
sage: solver3.add_clause( (1, 2, 100) )
sage: solver3.add_clause( (-1,) )
sage: solver3.add_clause( (-2,) )
sage: solver3()                          # optional - kissat
(None, False, False, ..., True)
```

command = 'kissat -q {input}'

class sage.sat.solvers.dimacs.**RSat**(*command=None*, *filename=None*, *verbosity=0*, ***kwds*)

Bases: *DIMACS*

An instance of the RSat solver.

For information on RSat see: http://reasoning.cs.ucla.edu/rsat/

EXAMPLES:

```
sage: from sage.sat.solvers import RSat
sage: solver = RSat()
sage: solver
DIMACS Solver: 'rsat {input} -v -s'
```

When the problem is SAT:

```
sage: from sage.sat.solvers import RSat
sage: solver = RSat()
sage: solver.add_clause( (1, 2, 3) )
sage: solver.add_clause( (-1,) )
sage: solver.add_clause( (-2,) )
sage: solver()                          # optional - rsat
(None, False, False, True)
```

When the problem is UNSAT:

```
sage: solver = RSat()
sage: solver.add_clause((1,2))
sage: solver.add_clause((-1,2))
sage: solver.add_clause((1,-2))
sage: solver.add_clause((-1,-2))
sage: solver()                          # optional - rsat
False
```

command = 'rsat {input} -v -s'

1.1.3 PicoSAT Solver

This solver relies on the `pycosat` Python bindings to `PicoSAT`.

The `pycosat` package should be installed on your Sage installation.

AUTHORS:

- Thierry Monteil (2018): initial version.

class `sage.sat.solvers.picosat.`**`PicoSAT`**(*verbosity=0*, *prop_limit=0*)

Bases: *SatSolver*

PicoSAT Solver.

INPUT:

- `verbosity` – an integer between 0 and 2 (default: 0); verbosity
- `prop_limit` – an integer (default: 0); the propagation limit

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()                          # optional - pycosat
```

`add_clause`(*lits*)

Add a new clause to set of clauses.

INPUT:

- `lits` – a tuple of nonzero integers

Note: If any element `e` in `lits` has `abs(e)` greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()                              # optional - pycosat
sage: solver.add_clause((1, -2 , 3))                  # optional - pycosat
```

clauses(*filename=None*)

Return original clauses.

INPUT:

- `filename` – (optional) if given, clauses are written to `filename` in DIMACS format

OUTPUT:

If `filename` is `None` then a list of `lits` is returned, where `lits` is a list of literals.

If `filename` points to a writable file, then the list of original clauses is written to that file in DIMACS format.

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()                              # optional - pycosat
sage: solver.add_clause((1,2,3,4,5,6,7,8,-9))         # optional - pycosat
sage: solver.clauses()                                # optional - pycosat
[[1, 2, 3, 4, 5, 6, 7, 8, -9]]
```

DIMACS format output:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()                              # optional - pycosat
sage: solver.add_clause((1, 2, 4))                    # optional - pycosat
sage: solver.add_clause((1, 2, -4))                   # optional - pycosat
sage: fn = tmp_filename()                             # optional - pycosat
sage: solver.clauses(fn)                              # optional - pycosat
sage: print(open(fn).read())                          # optional - pycosat
p cnf 4 2
1 2 4 0
1 2 -4 0
```

nvars()

Return the number of variables.

Note that for compatibility with DIMACS convention, the number of variables corresponds to the maximal index of the variables used.

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()                              # optional - pycosat
sage: solver.nvars()                                  # optional - pycosat
0
```

If a variable with intermediate index is not used, it is still considered as a variable:

```
sage: solver.add_clause((1,-2,4))                     # optional - pycosat
sage: solver.nvars()                                  # optional - pycosat
4
```

var(*decision=None*)

Return a *new* variable.

INPUT:

- `decision` – ignored; accepted for compatibility with other solvers

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()                          # optional - pycosat
sage: solver.var()                                # optional - pycosat
1

sage: solver.add_clause((-1,2,-4))                # optional - pycosat
sage: solver.var()                                # optional - pycosat
5
```

1.1.4 Solve SAT problems Integer Linear Programming

The class defined here is a *SatSolver* that solves its instance using `MixedIntegerLinearProgram`. Its performance can be expected to be slower than when using `CryptoMiniSat`.

class sage.sat.solvers.sat_lp.**SatLP**(*solver=None*, *verbose=0*, *, *integrality_tolerance=0.001*)

Bases: *SatSolver*

Initializes the instance

INPUT:

- `solver` – (default: `None`) Specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`). Sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

EXAMPLES:

```
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
```

add_clause(*lits*)

Add a new clause to set of clauses.

INPUT:

- `lits` - a tuple of integers != 0

Note: If any element `e` in `lits` has `abs(e)` greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
sage: for u,v in graphs.CycleGraph(6).edges(sort=False, labels=False):
....:     u,v = u+1,v+1
....:     S.add_clause((u,v))
....:     S.add_clause((-u,-v))
```

nvars()

Return the number of variables.

EXAMPLES:

```
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
sage: S.var()
1
sage: S.var()
2
sage: S.nvars()
2
```

var()

Return a *new* variable.

EXAMPLES:

```
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
sage: S.var()
1
```

1.1.5 CryptoMiniSat Solver

This solver relies on Python bindings provided by upstream cryptominisat.

The `cryptominisat` package should be installed on your Sage installation.

AUTHORS:

- Thierry Monteil (2017): complete rewrite, using upstream Python bindings, works with cryptominisat 5.
- Martin Albrecht (2012): first version, as a cython interface, works with cryptominisat 2.

class sage.sat.solvers.cryptominisat.**CryptoMiniSat**(*verbosity=0*, *confl_limit=None*, *threads=None*)

Bases: `SatSolver`

CryptoMiniSat Solver.

INPUT:

- `verbosity` – an integer between 0 and 15 (default: 0). Verbosity.
- `confl_limit` – an integer (default: `None`). Abort after this many conflicts. If set to `None`, never aborts.
- `threads` – an integer (default: None). The number of thread to use. If set to `None`, the number of threads used corresponds to the number of cpus.

EXAMPLES:

```
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()                                  # optional -␣
↪pycryptosat
```

add_clause(*lits*)

Add a new clause to set of clauses.

INPUT:

- `lits` – a tuple of nonzero integers.

Note: If any element `e` in `lits` has `abs(e)` greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()                                  # optional -␣
↪pycryptosat
sage: solver.add_clause((1, -2 , 3))                            # optional -␣
↪pycryptosat
```

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

Add a new XOR clause to set of clauses.

INPUT:

- `lits` – a tuple of positive integers.
- `rhs` – boolean (default: `True`). Whether this XOR clause should be evaluated to `True` or `False`.

EXAMPLES:

```
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()                                  # optional -␣
↪pycryptosat
sage: solver.add_xor_clause((1, 2 , 3), False)                  # optional -␣
↪pycryptosat
```

clauses(*filename=None*)

Return original clauses.

INPUT:

- `filename` – if not `None` clauses are written to `filename` in DIMACS format (default: `None`)

OUTPUT:

If `filename` is `None` then a list of `lits, is_xor, rhs` tuples is returned, where `lits` is a tuple of literals, `is_xor` is always `False` and `rhs` is always `None`.

If `filename` points to a writable file, then the list of original clauses is written to that file in DIMACS format.

EXAMPLES:

```
sage: from sage.sat.solvers import CryptoMiniSat
sage: solver = CryptoMiniSat()                              # optional -␣
↪pycryptosat
sage: solver.add_clause((1,2,3,4,5,6,7,8,-9))               # optional -␣
↪pycryptosat
sage: solver.add_xor_clause((1,2,3,4,5,6,7,8,9), rhs=True)  # optional -␣
↪pycryptosat
sage: solver.clauses()                                      # optional -␣
↪pycryptosat
[((1, 2, 3, 4, 5, 6, 7, 8, -9), False, None),
((1, 2, 3, 4, 5, 6, 7, 8, 9), True, True)]
```

DIMACS format output:

```
sage: from sage.sat.solvers import CryptoMiniSat
sage: solver = CryptoMiniSat()                      # optional - pycryptosat
sage: solver.add_clause((1, 2, 4))                  # optional - pycryptosat
sage: solver.add_clause((1, 2, -4))                 # optional - pycryptosat
sage: fn = tmp_filename()                           # optional - pycryptosat
sage: solver.clauses(fn)                            # optional - pycryptosat
sage: print(open(fn).read())                        # optional - pycryptosat
p cnf 4 2
1 2 4 0
1 2 -4 0
```

Note that in cryptominisat, the DIMACS standard format is augmented with the following extension: having an `x` in front of a line makes that line an XOR clause:

```
sage: solver.add_xor_clause((1,2,3), rhs=True)      # optional - pycryptosat
sage: solver.clauses(fn)                            # optional - pycryptosat
sage: print(open(fn).read())                        # optional - pycryptosat
p cnf 4 3
1 2 4 0
1 2 -4 0
x1 2 3 0
```

Note that inverting an xor-clause is equivalent to inverting one of the variables:

```
sage: solver.add_xor_clause((1,2,5),rhs=False)      # optional - pycryptosat
sage: solver.clauses(fn)                            # optional - pycryptosat
sage: print(open(fn).read())                        # optional - pycryptosat
p cnf 5 4
1 2 4 0
1 2 -4 0
x1 2 3 0
x1 2 -5 0
```

nvars()

Return the number of variables.

Note that for compatibility with DIMACS convention, the number of variables corresponds to the maximal index of the variables used.

EXAMPLES:

```
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()                                # optional -␣
↪pycryptosat
sage: solver.nvars()                                          # optional -␣
↪pycryptosat
0
```

If a variable with intermediate index is not used, it is still considered as a variable:

```
sage: solver.add_clause((1,-2,4))                             # optional -␣
↪pycryptosat
sage: solver.nvars()                                          # optional -␣
↪pycryptosat
4
```

var(*decision=None*)

Return a *new* variable.

INPUT:

- `decision` – accepted for compatibility with other solvers, ignored.

EXAMPLES:

```
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()                                # optional -␣
↪pycryptosat
sage: solver.var()                                            # optional -␣
↪pycryptosat
1

sage: solver.add_clause((-1,2,-4))                            # optional -␣
↪pycryptosat
sage: solver.var()                                            # optional -␣
↪pycryptosat
5
```

CHAPTER

TWO

CONVERTERS

Sage supports conversion from Boolean polynomials (also known as Algebraic Normal Form) to Conjunctive Normal Form:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_sparse(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 3 2
-2 0
1 0
```

2.1 Details on Specific Converterts

2.1.1 An ANF to CNF Converter using a Dense/Sparse Strategy

This converter is based on two converters. The first one, by Martin Albrecht, was based on [CB2007], this is the basis of the "dense" part of the converter. It was later improved by Mate Soos. The second one, by Michael Brickenstein, uses a reduced truth table based approach and forms the "sparse" part of the converter.

AUTHORS:

- Martin Albrecht - (2008-09) initial version of 'anf2cnf.py'
- Michael Brickenstein - (2009) 'cnf.py' for PolyBoRi
- Mate Soos - (2010) improved version of 'anf2cnf.py'
- Martin Albrecht - (2012) unified and added to Sage

Classes and Methods

class sage.sat.converters.polybori.**CNFEncoder**(*solver*, *ring*, *max_vars_sparse=6*, *use_xor_clauses=None*, *cutting_number=6*, *random_seed=16*)

Bases: `ANF2CNFConverter`

ANF to CNF Converter using a Dense/Sparse Strategy. This converter distinguishes two classes of polynomials.

1. Sparse polynomials are those with at most `max_vars_sparse` variables. Those are converted using reduced truth-tables based on PolyBoRi's internal representation.

2. Polynomials with more variables are converted by introducing new variables for monomials and by converting these linearised polynomials.

Linearised polynomials are converted either by splitting XOR chains – into chunks of length `cutting_number` – or by constructing XOR clauses if the underlying solver supports it. This behaviour is disabled by passing `use_xor_clauses=False`.

__init__(*solver*, *ring*, *max_vars_sparse=6*, *use_xor_clauses=None*, *cutting_number=6*, *random_seed=16*)

Construct ANF to CNF converter over `ring` passing clauses to `solver`.

INPUT:

- `solver` - a SAT-solver instance
- `ring` - a `sage.rings.polynomial.pbori.BooleanPolynomialRing`
- `max_vars_sparse` - maximum number of variables for direct conversion
- `use_xor_clauses` - use XOR clauses; if `None` use if `solver` supports it. (default: `None`)
- `cutting_number` - maximum length of XOR chains after splitting if XOR clauses are not supported (default: 6)
- `random_seed` - the direct conversion method uses randomness, this sets the seed (default: 16)

EXAMPLES:

We compare the sparse and the dense strategies, sparse first:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_sparse(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 3 2
-2 0
1 0
sage: e.phi
[None, a, b, c]
```

Now, we convert using the dense strategy:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_dense(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 4 5
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 0
4 1 0
sage: e.phi
[None, a, b, c, a*b]
```

Note: This constructor generates SAT variables for each Boolean polynomial variable.

`__call__`(*F*)

Encode the boolean polynomials in `F` .

INPUT:

- `F` - an iterable of `sage.rings.polynomial.pbori.BooleanPolynomial`

OUTPUT: An inverse map int -> variable

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: e([a*b + a + 1, a*b+ a + c])
[None, a, b, c, a*b]
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 4 9
-2 0
1 0
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 -3 0
4 1 -3 0
4 -1 3 0
-4 1 3 0

sage: e.phi
```

(continues on next page)

(continued from previous page)

```
[None, a, b, c, a*b]
```

clauses(*f*)

Convert f using the sparse strategy if `f.nvariables()` is at most `max_vars_sparse` and the dense strategy otherwise.

INPUT:

- `f` - a `sage.rings.polynomial.pbori.BooleanPolynomial`

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: e.clauses(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 3 2
-2 0
1 0
sage: e.phi
[None, a, b, c]

sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: e.clauses(a*b + a + c)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 4 7
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 -3 0
4 1 -3 0
4 -1 3 0
-4 1 3 0

sage: e.phi
[None, a, b, c, a*b]
```

clauses_dense(*f*)

Convert f using the dense strategy.

INPUT:

- `f` - a `sage.rings.polynomial.pbori.BooleanPolynomial`

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_dense(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 4 5
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 0
4 1 0
sage: e.phi
[None, a, b, c, a*b]
```

clauses_sparse(*f*)

Convert `f` using the sparse strategy.

INPUT:

- `f` - a `sage.rings.polynomial.pbori.BooleanPolynomial`

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_sparse(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 3 2
-2 0
1 0
sage: e.phi
[None, a, b, c]
```

monomial(*m*)

Return SAT variable for `m`

INPUT:

- `m` - a monomial.

OUTPUT: An index for a SAT variable corresponding to `m`.

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
```

(continues on next page)

(continued from previous page)

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_dense(a*b + a + 1)
sage: e.phi
[None, a, b, c, a*b]
```

If monomial is called on a new monomial, a new variable is created:

```
sage: e.monomial(a*b*c)
5
sage: e.phi
[None, a, b, c, a*b, a*b*c]
```

If monomial is called on a monomial that was queried before, the index of the old variable is returned and no new variable is created:

```
    sage: e.monomial(a*b)
    4
    sage: e.phi
    [None, a, b, c, a*b, a*b*c]

.. note::

    For correctness, this function is cached.
```

permutations = Cached version of <function CNFEncoder.permutations>

property phi

Map SAT variables to polynomial variables.

EXAMPLES:

```
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: ce = CNFEncoder(DIMACS(), B)
sage: ce.var()
4
sage: ce.phi
[None, a, b, c, None]
```

split_xor(*monomial_list*, *equal_zero*)

Split XOR chains into subchains.

INPUT:

- `monomial_list` - a list of monomials
- `equal_zero` - is the constant coefficient zero?

EXAMPLES:

```
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: ce = CNFEncoder(DIMACS(), B, cutting_number=3)
sage: ce.split_xor([1,2,3,4,5,6], False)
[[[1, 7], False], [[7, 2, 8], True], [[8, 3, 9], True], [[9, 4, 10], True],␣
↪[[10, 5, 11], True], [[11, 6], True]]

sage: ce = CNFEncoder(DIMACS(), B, cutting_number=4)
sage: ce.split_xor([1,2,3,4,5,6], False)
[[[1, 2, 7], False], [[7, 3, 4, 8], True], [[8, 5, 6], True]]

sage: ce = CNFEncoder(DIMACS(), B, cutting_number=5)
sage: ce.split_xor([1,2,3,4,5,6], False)
[[[1, 2, 3, 7], False], [[7, 4, 5, 6], True]]
```

to_polynomial(*c*)

Convert clause to `sage.rings.polynomial.pbori.BooleanPolynomial`

INPUT:

- `c` - a clause

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: _ = e([a*b + a + 1, a*b+ a + c])
sage: e.to_polynomial( (1,-2,3) )
a*b*c + a*b + b*c + b
```

var(*m=None*, *decision=None*)

Return a *new* variable.

This is a thin wrapper around the SAT-solvers function where we keep track of which SAT variable corresponds to which monomial.

INPUT:

- `m` - something the new variables maps to, usually a monomial
- `decision` - is this variable a decision variable?

EXAMPLES:

```
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: ce = CNFEncoder(DIMACS(), B)
sage: ce.var()
4
```

zero_blocks(*f*)

Divide the zero set of f into blocks.

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: e = CNFEncoder(DIMACS(), B)
sage: sorted(sorted(d.items()) for d in e.zero_blocks(a*b*c))
[[(c, 0)], [(b, 0)], [(a, 0)]]
```

Note: This function is randomised.

CHAPTER

THREE

HIGHLEVEL INTERFACES

Sage provides various highlevel functions which make working with Boolean polynomials easier. We construct a very small-scale AES system of equations and pass it to a SAT solver:

```
sage: sr = mq.SR(1,1,1,4,gf2=True,polybori=True)
sage: while True:
....:     try:
....:         F,s = sr.polynomial_system()
....:         break
....:     except ZeroDivisionError:
....:         pass
sage: from sage.sat.boolean_polynomials import solve as solve_sat # optional -␣
↪pycryptosat
sage: s = solve_sat(F)                                             # optional -␣
↪pycryptosat
sage: F.subs(s[0])                                                 # optional -␣
↪pycryptosat
Polynomial Sequence with 36 Polynomials in 0 Variables
```

3.1 Details on Specific Highlevel Interfaces

3.1.1 SAT Functions for Boolean Polynomials

These highlevel functions support solving and learning from Boolean polynomial systems. In this context, "learning" means the construction of new polynomials in the ideal spanned by the original polynomials.

AUTHOR:

- Martin Albrecht (2012): initial version

Functions

`sage.sat.boolean_polynomials.`**`learn`**(*F*, *converter=None*, *solver=None*, *max_learnt_length=3*, *interreduction=False*, ***kwds*)

Learn new polynomials by running SAT-solver `solver` on SAT-instance produced by `converter` from `F`.

INPUT:

- `F` - a sequence of Boolean polynomials

- converter - an ANF to CNF converter class or object. If converter is None then *sage.sat.converters.polybori.CNFEncoder* is used to construct a new converter. (default: None)
- solver - a SAT-solver class or object. If solver is None then *sage.sat.solvers.cryptominisat.CryptoMiniSat* is used to construct a new converter. (default: None)
- max_learnt_length - only clauses of length <= max_length_learnt are considered and converted to polynomials. (default: 3)
- interreduction - inter-reduce the resulting polynomials (default: False)

Note: More parameters can be passed to the converter and the solver by prefixing them with c_ and s_ respectively. For example, to increase CryptoMiniSat's verbosity level, pass s_verbosity=1.

OUTPUT:

A sequence of Boolean polynomials.

EXAMPLES:

```
sage: from sage.sat.boolean_polynomials import learn as learn_sat # optional -␣
↪pycryptosat
```

We construct a simple system and solve it:

```
sage: set_random_seed(2300)                      # optional - pycryptosat
sage: sr = mq.SR(1,2,2,4,gf2=True,polybori=True) # optional - pycryptosat
sage: F,s = sr.polynomial_system()               # optional - pycryptosat
sage: H = learn_sat(F)                           # optional - pycryptosat
sage: H[-1]                                      # optional - pycryptosat
k033 + 1
```

sage.sat.boolean_polynomials.**solve**(*F*, *converter=None*, *solver=None*, *n=1*, *target_variables=None*, ***kwds*)

Solve system of Boolean polynomials F by solving the SAT-problem – produced by converter – using solver.

INPUT:

- F - a sequence of Boolean polynomials
- n - number of solutions to return. If n is +infinity then all solutions are returned. If n <infinity then n solutions are returned if F has at least n solutions. Otherwise, all solutions of F are returned. (default: 1)
- converter - an ANF to CNF converter class or object. If converter is None then *sage.sat.converters.polybori.CNFEncoder* is used to construct a new converter. (default: None)
- solver - a SAT-solver class or object. If solver is None then *sage.sat.solvers.cryptominisat.CryptoMiniSat* is used to construct a new converter. (default: None)
- target_variables - a list of variables. The elements of the list are used to exclude a particular combination of variable assignments of a solution from any further solution. Furthermore target_variables denotes which variable-value pairs appear in the solutions. If target_variables is None all variables appearing in the polynomials of F are used to construct exclusion clauses. (default: None)
- ****kwds - parameters can be passed to the converter and the**
  solver by prefixing them with c_ and s_ respectively. For example, to increase CryptoMiniSat's verbosity level, pass s_verbosity=1.

OUTPUT:

A list of dictionaries, each of which contains a variable assignment solving `F`.

EXAMPLES:

We construct a very small-scale AES system of equations:

```
sage: sr = mq.SR(1,1,1,4,gf2=True,polybori=True)
sage: while True:  # workaround (see :trac:`31891`)
....:     try:
....:         F, s = sr.polynomial_system()
....:         break
....:     except ZeroDivisionError:
....:         pass
```

and pass it to a SAT solver:

```
sage: from sage.sat.boolean_polynomials import solve as solve_sat # optional -␣
↪pycryptosat
sage: s = solve_sat(F)                                             # optional -␣
↪pycryptosat
sage: F.subs(s[0])                                                 # optional -␣
↪pycryptosat
Polynomial Sequence with 36 Polynomials in 0 Variables
```

This time we pass a few options through to the converter and the solver:

```
sage: s = solve_sat(F, c_max_vars_sparse=4, c_cutting_number=8) # optional -␣
↪pycryptosat
sage: F.subs(s[0])                                              # optional -␣
↪pycryptosat
Polynomial Sequence with 36 Polynomials in 0 Variables
```

We construct a very simple system with three solutions and ask for a specific number of solutions:

```
sage: B.<a,b> = BooleanPolynomialRing() # optional - pycryptosat
sage: f = a*b                           # optional - pycryptosat
sage: l = solve_sat([f],n=1)            # optional - pycryptosat
sage: len(l) == 1, f.subs(l[0])         # optional - pycryptosat
(True, 0)

sage: l = solve_sat([a*b],n=2)        # optional - pycryptosat
sage: len(l) == 2, f.subs(l[0]), f.subs(l[1]) # optional - pycryptosat
(True, 0, 0)

sage: sorted((d[a], d[b]) for d in solve_sat([a*b],n=3))  # optional - pycryptosat
[(0, 0), (0, 1), (1, 0)]
sage: sorted((d[a], d[b]) for d in solve_sat([a*b],n=4))   # optional - pycryptosat
[(0, 0), (0, 1), (1, 0)]
sage: sorted((d[a], d[b]) for d in solve_sat([a*b],n=infinity))  # optional -␣
↪pycryptosat
[(0, 0), (0, 1), (1, 0)]
```

In the next example we see how the `target_variables` parameter works:

```
sage: from sage.sat.boolean_polynomials import solve as solve_sat # optional -␣
↪pycryptosat
sage: R.<a,b,c,d> = BooleanPolynomialRing()                    # optional -␣
↪pycryptosat
sage: F = [a+b,a+c+d]                                          # optional -␣
↪pycryptosat
```

First the normal use case:

```
sage: sorted((D[a], D[b], D[c], D[d]) for D in solve_sat(F,n=infinity))      #␣
↪optional - pycryptosat
[(0, 0, 0, 0), (0, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0)]
```

Now we are only interested in the solutions of the variables a and b:

```
sage: solve_sat(F,n=infinity,target_variables=[a,b])           # optional -␣
↪pycryptosat
[{b: 0, a: 0}, {b: 1, a: 1}]
```

Here, we generate and solve the cubic equations of the AES SBox (see github issue #26676):

```
sage: from sage.rings.polynomial.multi_polynomial_sequence import␣
↪PolynomialSequence     # optional - pycryptosat, long time
sage: from sage.sat.boolean_polynomials import solve as solve_sat                  ␣
↪     # optional - pycryptosat, long time
sage: sr = sage.crypto.mq.SR(1, 4, 4, 8, allow_zero_inversions = True)             ␣
↪     # optional - pycryptosat, long time
sage: sb = sr.sbox()                                                               ␣
↪     # optional - pycryptosat, long time
sage: eqs = sb.polynomials(degree = 3)                                             ␣
↪     # optional - pycryptosat, long time
sage: eqs = PolynomialSequence(eqs)                                                ␣
↪     # optional - pycryptosat, long time
sage: variables = map(str, eqs.variables())                                        ␣
↪     # optional - pycryptosat, long time
sage: variables = ",".join(variables)                                              ␣
↪     # optional - pycryptosat, long time
sage: R = BooleanPolynomialRing(16, variables)                                     ␣
↪     # optional - pycryptosat, long time
sage: eqs = [R(eq) for eq in eqs]                                                  ␣
↪               # optional - pycryptosat, long time
sage: sls_aes = solve_sat(eqs, n = infinity)                                       ␣
↪     # optional - pycryptosat, long time
sage: len(sls_aes)                                                                 ␣
↪     # optional - pycryptosat, long time
256
```

Note: Although supported, passing converter and solver objects instead of classes is discouraged because these objects are stateful.

REFERENCES:

CHAPTER

FOUR

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[RS] http://reasoning.cs.ucla.edu/rsat/

[GL] http://www.lri.fr/~simon/?page=glucose

[CMS] http://www.msoos.org

[SG09] http://www.satcompetition.org/2009/format-benchmarks2009.html

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

D

G

K

L

M

N