Sat

Release 10.4

The Sage Development Team

Jul 23, 2024
CONTENTS

1 Solvers 3
2 Converters 33
3 Highlevel Interfaces 45
4 Indices and Tables 51
Bibliography 53
Python Module Index 55
Index 57
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:

\[ x_1 \lor x_2 \lor (\neg x_3) \]

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.

\[(x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor (\neg x_3))\]

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)
```

```
>>> from sage.all import *
>>> solver = SAT()
>>> solver.add_clause( ( Integer( 1), Integer( 2), Integer( 3)) )
>>> solver.add_clause( ( Integer( 1), Integer( 2), -Integer(3)) )
>>> solver()  # random
(Non, 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)
```

(continues on next page)
Alternatively, there is `sage.sat.solvers.dimacs.DIMACS.clauses()`:

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

```python
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 pycryptosat 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:**

```python
sage: SAT(solver="LP")
# needs sage.numerical.mip
an ILP-based SAT Solver
```

```python
>>> from sage.all import *
>>> SAT(solver="LP")
# needs sage.numerical.mip
an ILP-based SAT Solver
```

```python
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
```

```>>> from sage.all import *
>>> from sage.sat.solvers.satsolver import SatSolver
>>> solver = SatSolver()
>>> solver.add_clause( (Integer(1), -Integer(2) , Integer(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
```

```>>> from sage.all import *
>>> from sage.sat.solvers.satsolver import SatSolver
>>> solver = SatSolver()
>>> 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
```

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

```python
>>> from sage.all import *
>>> from sage.sat.solvers.satsolver import SatSolver
>>> solver = SatSolver()
>>> 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:**

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

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

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

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

**nvars()**

Return the number of variables.

**EXAMPLES:**

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

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

**read** *(filename)*

Reads DIMAC files.


The differences were summarized in the discussion on the issue 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:**

```python
sage: from io import StringIO
sage: file_object = StringIO("c A sample .cnf file.
p cnf 3 2
1 -3 0
2 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:

```python
sage: from io import StringIO
sage: file_object = StringIO("c A sample .cnf file with xor clauses.
p cnf 3
1 2 0
3 0
x1 2 3 0")
```

(continues on next page)
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:

>>> from sage.all import *
>>> from io import StringIO
>>> file_object = StringIO("c A sample .cnf file with xor clauses.\n p cnf 3 3 \n 1 2 0\n 3 0\n x1 2 3 0")
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat #
˓→optional - pycryptosat
>>> solver = CryptoMiniSat() #
˓→optional - pycryptosat
>>> solver.read(file_object) #
˓→optional - pycryptosat
>>> solver.clauses() #
˓→optional - pycryptosat
[((1, 2), False, None), ((3,), False, None), ((1, 2, 3), True, True)]
>>> solver() #
˓→optional - pycryptosat
(None, True, True, True)
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 Python'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**

```python
class sage.sat.solvers.dimacs.DIMACS(command=None, filename=None, verbosity=0, **kwds):
    Bases: SatSolver
    Generic DIMACS Solver.

    __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:

```python
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)
```

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

When the problem is UNSAT:

```python
sage: solver = RSat()
sage: solver.add_clause((1, 2))
sage: solver.add_clause((-1, 2))
sage: solver.add_clause((1, -2))

sage: solver()               # optional - rsat
False
```

```python
>>> from sage.all import *

>>> from sage.sat.solvers import RSat

>>> solver = RSat()

>>> solver.add_clause((Integer(1), Integer(2)))
>>> solver.add_clause((-Integer(1), Integer(2)))
>>> solver.add_clause((Integer(1), -Integer(2)))

>>> solver()                # optional - rsat
False
```

With Glucose:

```python
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()               # optional - glucose
False
```

```python
>>> from sage.all import *

>>> from sage.sat.solvers.dimacs import Glucose

>>> solver = Glucose()

>>> solver.add_clause((Integer(1), Integer(2)))
(continues on next page)
```
.. code-block:: python

>>> solver.add_clause((-Integer(1),Integer(2)))
>>> solver.add_clause((Integer(1),-Integer(2)))
>>> solver()  # optional - glucose
(Non, True, True)
>>> solver.add_clause((-Integer(1),-Integer(2))
>>> solver()  # optional - glucose
False

With GlucoseSyrup:

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

.. code-block:: python

```python
>>> from sage.all import *
>>> from sage.sat.solvers.dimacs import GlucoseSyrup
``` 

```python
>>> solver = GlucoseSyrup()
>>> solver.add_clause((Integer(1),Integer(2)))
>>> solver.add_clause((-Integer(1),Integer(2)))
>>> solver.add_clause((Integer(1),-Integer(2)))
>>> solver()  # optional - glucose
(Non, True, True)
>>> solver.add_clause((-Integer(1),-Integer(2)))
>>> solver()  # optional - glucose
False
```

.. code-block:: python

```python
add_clause(lits)
```

Add a new clause to set of clauses.

**INPUT:**

- lits - a tuple of integers != 0

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

**EXAMPLES:**

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.var() 1
sage: solver.var(decision=True) 2
sage: solver.add_clause((1, -2, 3))
sage: solver
DIMACS Solver: ''
```
>>> from sage.all import *
>>> from sage.sat.solvers.dimacs import DIMACS
>>> solver = DIMACS()
>>> solver.var()
1
>>> solver.var(decision=True)
2
>>> solver.add_clause((Integer(1), -Integer(2), Integer(3)))

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:**

```python
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)
```

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

```python
>>> from sage.all import *
>>> from sage.sat.solvers.dimacs import DIMACS
>>> fn = tmp_filename()
>>> solver = DIMACS()
>>> solver.add_clause((Integer(1), Integer(2), Integer(3)))
>>> solver.clauses()
[((1, 2, 3), False, None)]
>>> solver.add_clause((Integer(1), Integer(2), -Integer(3)))
>>> solver.clauses(fn)
>>> print(open(fn).read())
p cnf 3 2
1 2 3 0
1 2 -3 0
```

`command = ''`

1.1. Details on Specific Solvers 13
nvars()

Return the number of variables.

EXAMPLES:

```python
define (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:

```python
define (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:
This function also accepts a “simple” format:

```python
>>> from sage.all import *
>>> solver.clauses(fn)
>>> print(open(fn).read())
p cnf 3 1
1 2 -3 0
<BLANKLINE>
```

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

```
>>> from sage.all import *
>>> DIMACS.render_dimacs([ (Integer( 1),Integer(2)), (Integer( 1),Integer(2),-Integer(3)) ], fn, Integer(3))
>>> print(open(fn).read())
p cnf 3 2
1 2 0
1 2 -3 0
<BLANKLINE>
```

```python
var(decision=None)
```

Return a new variable.

**INPUT:**

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

**EXAMPLES:**

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

>>> from sage.all import *
>>> from sage.sat.solvers.dimacs import DIMACS
>>> solver = DIMACS()
>>> solver.var()
1
```

```python
write(filename=None)
```

Write DIMACS file.

**INPUT:**

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

**EXAMPLES:**

1.1. Details on Specific Solvers
```python
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.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
```
```
>>> from sage.all import *
```
```
```
When the problem is SAT:

```python
glp = Glucose()
solver1 = Glucose()
solver1.add_clause((1, 2, 3))
solver1.add_clause((-1,))
solver1.add_clause((-2,))
solver1()  # optional - glucose
```

When the problem is UNSAT:

```python
from sage.all import *

glp = Glucose()
solver2 = Glucose()
solver2.add_clause((1, 2))
solver2.add_clause((-1, 2))
solver2.add_clause((1, -2))
solver2.add_clause((-1, -2))
solver2()  # optional - glucose
```

With one hundred variables:

```python
from sage.all import *

glp = Glucose()
solver3 = Glucose()
solver3.add_clause((1, 2, 100))
solver3.add_clause((-1,))
solver3.add_clause((-2,))
solver3()  # optional - glucose
```

(continues on next page)
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}'

    >>> from sage.all import *
    >>> from sage.sat.solvers import GlucoseSyrup
    >>> solver = GlucoseSyrup()
    >>> 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)

    >>> from sage.all import *
    >>> solver1 = GlucoseSyrup()
    >>> solver1.add_clause( (Integer(1), Integer(2), Integer(3)) )
    >>> solver1.add_clause( (-Integer(1),) )
    >>> solver1.add_clause( (-Integer(2),) )
    >>> 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

    >>> from sage.all import *
    >>> solver2 = GlucoseSyrup()
With one hundred variables:

```python
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)
```

```
>>> from sage.all import *
>>> solver3 = GlucoseSyrup()
>>> solver3.add_clause((Integer(1), Integer(2), Integer(100))
>>> solver3.add_clause((Integer(-1),))
>>> solver3.add_clause((Integer(-2),))
>>> solver3() # optional - glucose
(None, False, False, ..., True)
```

```python
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:

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

```python
>>> from sage.all import *
>>> from sage.sat.solvers import Kissat

>>> solver = Kissat()
>>> solver
DIMACS Solver: 'kissat -q {input}'
```

When the problem is SAT:

```python
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)
```

1.1. Details on Specific Solvers
>>> from sage.all import *
>>> solver1 = Kissat()
>>> solver1.add_clause( (Integer(1), Integer(2), Integer(3)) )
>>> solver1.add_clause( (-Integer(1),) )
>>> solver1.add_clause( (-Integer(2),) )
>>> solver1() # optional - kissat
(None, False, False, True, 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

>>> from sage.all import *
>>> solver2 = Kissat()
>>> solver2.add_clause((Integer(1),Integer(2)))
>>> solver2.add_clause((-Integer(1),Integer(2)))
>>> solver2.add_clause((Integer(1),-Integer(2)))
>>> solver2.add_clause((-Integer(1),-Integer(2)))

>>> solver2() # optional - kissat
False

With one hundred variables:

sage: solver3 = Kissat()
>>> solver3.add_clause( (1, 2, 100) )
>>> solver3.add_clause( (-1,) )
>>> solver3.add_clause( (-2,) )

sage: solver3() # optional - kissat
(0, False, False, ..., True)

>>> from sage.all import *
>>> solver3 = Kissat()
>>> solver3.add_clause( (Integer(1), Integer(2), Integer(100)) )
>>> solver3.add_clause( (-Integer(1),) )
>>> solver3.add_clause( (-Integer(2),) )

>>> solver3() # optional - kissat
(0, False, False, ..., True)

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'

>>> from sage.all import *
>>> from sage.sat.solvers import RSat
>>> solver = RSat()

When the problem is SAT:

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

>>> from sage.all import *
>>> from sage.sat.solvers import RSat
>>> solver = RSat()
>>> solver.add_clause( (Integer(1), Integer(2), Integer(3)) )
>>> solver.add_clause( (-Integer(1),) )
>>> solver.add_clause( (-Integer(2),) )
>>> solver() # optional - rsat
(None, False, False, True)

When the problem is UNSAT:

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

>>> from sage.all import *
>>> solver = RSat()
>>> solver.add_clause((Integer(1),Integer(2)))
>>> solver.add_clause((-Integer(1),Integer(2)))
>>> solver.add_clause((Integer(1),-Integer(2)))
>>> solver.add_clause((-Integer(1),-Integer(2)))
>>> solver() # optional - rsat
False

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

1.1. Details on Specific Solvers 21
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.

```python
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:

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

```python
>>> from sage.all import *
>>> from sage.sat.solvers.picosat import PicoSAT
>>> solver = PicoSAT() # optional - pycosat
```

```python
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:

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

```python
>>> from sage.all import *
>>> from sage.sat.solvers.picosat import PicoSAT
>>> solver = PicoSAT() # optional - pycosat
```

```python
clauses(filename=None)
Return original clauses.

INPUT:
  • filename — (optional) if given, clauses are written to filename in DIMACS format
```

```python
```
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:

```python
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:

```python
sage: # optional - pycosat
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()
sage: solver.add_clause((1, 2, 4))
sage: solver.add_clause((1, 2, -4))
sage: fn = tmp_filename()
sage: solver.clauses(fn)
```

```text
p cnf 4 2
1 2 4 0
1 2 -4 0
```

```python
>>> from sage.all import *
>>> from sage.sat.solvers.picosat import PicoSAT
>>> solver = PicoSAT()
>>> solver.add_clause((Integer(1), Integer(2), Integer(4)))
>>> solver.add_clause((Integer(1), Integer(2), -Integer(4)))
>>> fn = tmp_filename()
>>> solver.clauses(fn)
```

```text
p cnf 4 2
1 2 4 0
1 2 -4 0
<BLANKLINE>
```

`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:
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
decision=None
sage: solver.nvars()  # optional - pycosat
4
```

```
>>> from sage.all import *
>>> from sage.sat.solvers.picosat import PicoSAT
>>> solver = PicoSAT()  # optional - pycosat
>>> solver.add_clause((Integer(1),-Integer(2),Integer(4)))  # optional - pycosat
>>> 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
```

```
>>> from sage.all import *
>>> from sage.sat.solvers.picosat import PicoSAT
>>> solver = PicoSAT()  # optional - pycosat
>>> solver.var()  # optional - pycosat
1
>>> solver.add_clause((-Integer(1),Integer(2),-Integer(4)))  # optional - pycosat
>>> 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.

```python
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:**

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

>>> from sage.all import *
>>> 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 \(\text{abs}(e)\) greater than the number of variables generated so far, then new variables are created automatically.

**EXAMPLES:**

```python
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))

>>> from sage.all import *
>>> S=SAT(solver="LP"); S an ILP-based SAT Solver

>>> for u,v in graphs.CycleGraph(Integer(6)).edges(sort=False, labels=False):
...:    u,v = u+Integer(1),v+Integer(1)
...:    S.add_clause((u,v))
...:    S.add_clause((-u,-v))
```

1.1. Details on Specific Solvers 25
**nvars()**

Return the number of variables.

**EXAMPLES:**

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

```python
>>> from sage.all import *
```  
```python
>>> S=SAT(solver="LP"); S
an ILP-based SAT Solver
```  
```python
>>> S.var()
1
```  
```python
>>> S.var()
2
```  
```python
>>> S.nvars()
2
```

**var()**

Return a new variable.

**EXAMPLES:**

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

```python
>>> from sage.all import *
```  
```python
>>> S=SAT(solver="LP"); S
an ILP-based SAT Solver
```  
```python
>>> S.var()
1
```

### 1.1.5 CryptoMiniSat Solver

This solver relies on Python bindings provided by upstream cryptominisat.

**AUTHORS:**


```python
class sage.sat.solvers.cryptominisat.CryptoMiniSat (verbosity=0, confliLimit=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
```

```
>>> from sage.all import *
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat
>>> 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 \( \text{abs}(e) \) 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
```

```
>>> from sage.all import *
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat
>>> solver = CryptoMiniSat()                          # optional ~
                      → pycryptosat
>>> solver.add_clause((Integer(1), -Integer(2) , Integer(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_clause((1, -2 , 3))                    # optional ~
                      → pycryptosat
```

```
>>> from sage.all import *
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat
>>> solver = CryptoMiniSat()                          # optional ~
                      → pycryptosat
>>> solver.add_clause((Integer(1), -Integer(2) , Integer(3)))  # optional - pycryptosat
```
from sage.all import *
from sage.sat.solvers.cryptominisat import CryptoMiniSat

solver = CryptoMiniSat()
solver.add_xor_clause((Integer(1), Integer(2), Integer(3)), False)

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

\begin{Verbatim}
\begin{center}
sage: solver.add_xor_clause((1,2,3), rhs=True)  # \texttt{optional - pycryptosat}
sage: solver.clauses(fn)  # \texttt{optional - pycryptosat}
sage: print(open(fn).read())  # \texttt{optional - pycryptosat}
\end{center}
\end{Verbatim}

\begin{Verbatim}
\begin{center}
sage: solver.add_xor_clause((1,2,3), rhs=True)  # \texttt{optional - pycryptosat}
sage: solver.clauses(fn)  # \texttt{optional - pycryptosat}
sage: print(open(fn).read())  # \texttt{optional - pycryptosat}
\end{center}
\end{Verbatim}

Note that inverting an \texttt{xOR} clause is equivalent to inverting one of the variables:

\begin{Verbatim}
\begin{center}
sage: solver.add_xor_clause((1,2,5), rhs=False)  # \texttt{optional - pycryptosat}
sage: solver.clauses(fn)  # \texttt{optional - pycryptosat}
sage: print(open(fn).read())  # \texttt{optional - pycryptosat}
\end{center}
\end{Verbatim}

\begin{Verbatim}
\begin{center}
sage: solver.add_xor_clause((1,2,5), rhs=False)  # \texttt{optional - pycryptosat}
sage: solver.clauses(fn)  # \texttt{optional - pycryptosat}
sage: print(open(fn).read())  # \texttt{optional - pycryptosat}
\end{center}
\end{Verbatim}
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:

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

```
>>> from sage.all import *
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat
>>> solver = CryptoMiniSat() # optional - pycryptosat
>>> 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
```

```
>>> from sage.all import *
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat
>>> solver = CryptoMiniSat() # optional - pycryptosat
>>> solver.nvars() # optional - pycryptosat
4
```

var(decision=None)

Return a new variable.

INPUT:

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

EXAMPLES:

```python
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

```ruby
>>> from sage.all import *
>>> from sage.sat.solvers.cryptominisat import CryptoMiniSat

>>> solver = CryptoMiniSat()  # optional - pycryptosat

>>> solver.var()  # optional - pycryptosat

>>> solver.add_clause((-Integer(1),Integer(2),-Integer(4)))  # optional - pycryptosat

>>> solver.var()  # optional - pycryptosat
```
Sage supports conversion from Boolean polynomials (also known as Algebraic Normal Form) to Conjunctive Normal Form:

```python
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 Converters

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:

```python
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]
```
2.1. Details on Specific Converters

Now, we convert using the dense strategy:

```python
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.

```python
__call__(F)
```

Encode the boolean polynomials in \( F \).

INPUT:

- \( F \) – an iterable of `sage.rings.polynomial.pbori.BooleanPolynomial`

OUTPUT: An inverse map \( int \rightarrow \text{variable} \)

EXAMPLES:

```python
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
```

```
>>> from sage.all import *
>>> B = BooleanPolynomialRing(names=('a', 'b', 'c')); (a, b, c,) = B._first_→ngens(3)
>>> from sage.sat.converters.polybori import CNFEncoder
>>> from sage.sat.solvers.dimacs import DIMACS
>>> fn = tmp_filename()
>>> solver = DIMACS(filename=fn)
>>> e = CNFEncoder(solver, B, max_vars_sparse=Integer(2))
>>> e([(a*b + a + Integer(1), a*b+ a + c])
[None, a, b, c, a*b]
>>> _ = solver.write()
>>> 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
```

(continues on next page)
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]
```
>>> fn = tmp_filename()
>>> solver = DIMACS(filename=fn)
>>> e = CNFEncoder(solver, B, max_vars_sparse=Integer(2))
>>> e.clauses(a*b + a + Integer(1))
>>> _ = solver.write()
>>> print(open(fn).read())

p cnf 3 2
-2 0
1 0

>>> e.phi
[None, a, b, c]

>>> B = BooleanPolynomialRing(names=('a', 'b', 'c')); (a, b, c,) = B._first_ngens(3)

>>> from sage.sat.converters.polybori import CNFEncoder
>>> from sage.sat.solvers.dimacs import DIMACS

>>> fn = tmp_filename()
>>> solver = DIMACS(filename=fn)
>>> e = CNFEncoder(solver, B, max_vars_sparse=Integer(2))
>>> e.clauses(a*b + a + c)
>>> _ = solver.write()
>>> 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

>>> e.phi
[None, a, b, c, a*b]
4 1 0
sage: e.phi
[None, a, b, c, a*b]

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

```python
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 + Integer(1))
sage: e.phi
[None, a, b, c, a*b]
```

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

```python
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:

```python
>>> from sage.all import *
>>> e.monomial(a*b*c)
5
>>> e.phi
[None, a, b, c, a*b, a*b*c]
```
permutations = Cached version of <function CNFEncoder.permutations>

property phi

Map SAT variables to polynomial variables.

EXAMPLES:

```python
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.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:

```python
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)
sage: ce.split_xor([1,2,3,4,5,6], False)
```
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, 3, 9], True], [[9, 4, 10], True],
 [[10, 5, 11], True], [[11, 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]]

from sage.all import *
from sage.sat.converters.polybori import CNFEncoder
from sage.sat.solvers.dimacs import DIMACS

B = BooleanPolynomialRing(names=(a, b, c, d, e, f,)); (a, b, c, d, e, f,) = B._first_ngens(6)

ce = CNFEncoder(DIMACS(), B, cutting_number=Integer(3))

ce.split_xor([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
 Integer(6)], False)
[[[1, 7], False], [[7, 2, 8], True], [[8, 3, 9], True], [[9, 4, 10], True],
 [[10, 5, 11], True], [[11, 6], True]]

ce = CNFEncoder(DIMACS(), B, cutting_number=Integer(4))

ce.split_xor([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
 Integer(6)], False)
[[[1, 2, 7], False], [[7, 3, 4, 8], True], [[8, 5, 6], True]]

ce = CNFEncoder(DIMACS(), B, cutting_number=Integer(5))

ce.split_xor([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5),
 Integer(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

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

```python
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
```

```python
>>> from sage.all import *
>>> from sage.sat.converters.polybori import CNFEncoder
>>> from sage.sat.solvers.dimacs import DIMACS
>>> B = BooleanPolynomialRing(names=('a', 'b', 'c')); (a, b, c,) = B._first_ngens(3)
>>> ce = CNFEncoder(DIMACS(), B)
>>> ce.var()
4
```

**zero_blocks** *(f)*

Divide the zero set of \( f \) into blocks.

**EXAMPLES:**

```python
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]]]
```

```python
>>> from sage.all import *
>>> B = BooleanPolynomialRing(names=('a', 'b', 'c')); (a, b, c,) = B._first_ngens(3)
>>> from sage.sat.converters.polybori import CNFEncoder
>>> from sage.sat.solvers.dimacs import DIMACS
>>> e = CNFEncoder(DIMACS(), B)
>>> sorted(sorted(d.items()) for d in e.zero_blocks(a*b*c))
[[[c, 0]], [[b, 0]], [[a, 0]]]
```

---

2.1. Details on Specific Converters 43
Note: This function is randomised.
Sage provides various high-level 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:

```python
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 -- cryptosat
sage: s = solve_sat(F)  # optional -- cryptosat
sage: F.subs(s[0])  # optional -- cryptosat
Polynomial Sequence with 36 Polynomials in 0 Variables
```

```python
>>> from sage.all import *
>>> sr = mq.SR(Integer(1),Integer(1),Integer(1),Integer(4),gf2=True,polybori=True)
>>> while True:
    ... try:
    ...     F,s = sr.polynomial_system()
    ...     break
    ... except ZeroDivisionError:
    ...     pass
>>> from sage.sat.boolean_polynomials import solve as solve_sat  # optional -- cryptosat
>>> s = solve_sat(F)  # optional -- cryptosat
>>> F.subs(s[Integer(0)])  # optional -- cryptosat
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

```python
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:

```python
sage: from sage.sat.boolean_polynomials import learn as learn_sat
```

```python
>>> from sage.all import *
>>> from sage.sat.boolean_polynomials import learn as learn_sat
```

We construct a simple system and solve it:

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

\[ k033 + 1 \]
>>> from sage.all import *

>>> set_random_seed(Integer(2300))

>>> sr = mq.SR(Integer(1), Integer(2), Integer(2), Integer(4), gf2=True,
               polybori=True)

>>> F, s = sr.polynomial_system()

>>> H = learn_sat(F)

>>> H[-Integer(1)]

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 is 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 :issue:`31891`) ....:     try: ....:         F, s = sr.polynomial_system() ....:         break ....:     except ZeroDivisionError: ....:         pass
```

```null
>>> from sage.all import *

>>> sr = mq.SR(Integer(1), Integer(1), Integer(1), Integer(4), gf2=True, polybori=True)

>>> while True:  # workaround (see :issue:`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
sage: s = solve_sat(F)
sage: F.subs(s[0])
Polynomial Sequence with 36 Polynomials in 0 Variables

>>> from sage.all import *
>>> from sage.sat.boolean_polynomials import solve as solve_sat
>>> s = solve_sat(F)
>>> F.subs(s[Integer(0)])
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)
sage: F.subs(s[0])
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()
sage: f = a*b
sage: l = solve_sat([f],n=1)

>>> B = BooleanPolynomialRing(names=('a', 'b',)); (a, b,) = B._first_ngens(2)
>>> f = a*b
>>> l = solve_sat([f],n=Integer(1))
```

(continues on next page)
In the next example we see how the target_variables parameter works:

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

```
from sage.all import *
from sage.sat.boolean_polynomials import solve as solve_sat
F = [a + b, a + c + d]
```

First the normal use case:

```
sage: sorted((D[a], D[b], D[c], D[d])
....: for D in solve_sat(F, n=infinity))
[(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])
[(b: 0, a: 0), (b: 1, a: 1)]
```

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

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

s
sage.sat.boolean_polynomials, 46
sage.sat.converters.polybori, 33
sage.sat.solvers.cryptominisat, 26
sage.sat.solvers.dimacs, 10
sage.sat.solvers.picosat, 22
sage.sat.solvers.sat_lp, 25
sage.sat.solvers.satsolver, 5
Non-alphabetical
__call__() (sage.sat.converters.polybori.CNFEncoder method), 36
__call__() (sage.sat.solvers.dimacs.DIMACS method), 10
__init__() (sage.sat.converters.polybori.CNFEncoder method), 34
__init__() (sage.sat.solvers.dimacs.DIMACS method), 10
add_clause() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 27
add_clause() (sage.sat.solvers.dimacs.DIMACS method), 12
add_clause() (sage.sat.solvers.picosat.PicoSAT method), 22
add_clause() (sage.sat.solvers.sat_lp.SatLP method), 25
add_clause() (sage.sat.solvers.satsolver.SatSolver method), 5
add_xor_clause() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 27
clauses() (sage.sat.converters.polybori.CNFEncoder method), 37
clauses() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 28
clauses() (sage.sat.solvers.dimacs.DIMACS method), 13
clauses() (sage.sat.solvers.picosat.PicoSAT method), 22
clauses() (sage.sat.solvers.satsolver.SatSolver method), 6
clauses_dense() (sage.sat.converters.polybori.CNFEncoder method), 38
clauses_sparse() (sage.sat.converters.polybori.CNFEncoder method), 39
CNFEncoder (class in sage.sat.converters.polybori), 34
command (sage.sat.solvers.dimacs.DIMACS attribute), 13
command (sage.sat.solvers.dimacs.Glucose attribute), 18
command (sage.sat.solvers.dimacs.GlucoseSyrup attribute), 19
command (sage.sat.solvers.dimacs.Kissat attribute), 20
command (sage.sat.solvers.dimacs.RSat attribute), 21
conflict_clause() (sage.sat.solvers.satsolver.SatSolver method), 6
CryptoMiniSat (class in sage.sat.solvers.cryptominisat), 26
DIMACS (class in sage.sat.solvers.dimacs), 10
Glucose (class in sage.sat.solvers.dimacs), 16
GlucoseSyrup (class in sage.sat.solvers.dimacs), 18
Kissat (class in sage.sat.solvers.dimacs), 19
learn() (in module sage.sat.boolean_polynomials), 46
learnt_clauses() (sage.sat.solvers.satsolver.SatSolver method), 7
module
sage.sat.boolean_polynomials, 46
sage.sat.converters.polybori, 33
sage.sat.solvers.cryptominisat, 26
sage.sat.solvers.dimacs, 10
sage.sat.solvers.picosat, 22
sage.sat.solvers.sat_lp, 25
sage.sat.solvers.satsolver, 5
monomial() (sage.sat.converters.polybori.CNFEncoder method), 40

N
nvars() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 30
nvars() (sage.sat.solvers.dimacs.DIMACS method), 13
nvars() (sage.sat.solvers.picosat.PicoSAT method), 23

INDEX
nvars() (sage.sat.solvers.sat_lp.SatLP method), 25
nvars() (sage.sat.solvers.satsolver.SatSolver method), 7
P
permutations (sage.sat.converters.polybori.CNFEncoder attribute), 41
phi (sage.sat.converters.polybori.CNFEncoder property), 41
PicoSAT (class in sage.sat.solvers.picosat), 22
R
read() (sage.sat.solvers.satsolver.SatSolver method), 8
render_dimacs() (sage.sat.solvers.dimacs.DIMACS static method), 14
RSat (class in sage.sat.solvers.dimacs), 20
S
sage.sat.boolean_polynomials module, 46
sage.sat.converters.polybori module, 33
sage.sat.solvers.cryptominisat module, 26
sage.sat.solvers.dimacs module, 10
sage.sat.solvers.picosat module, 22
sage.sat.solvers.sat_lp module, 25
sage.sat.solvers.satsolver module, 5
SAT() (in module sage.sat.solvers.satsolver), 5
SatLP (class in sage.sat.solvers.sat_lp), 25
SatSolver (class in sage.sat.solvers.satsolver), 5
solve() (in module sage.sat.boolean_polynomials), 47
split_xor() (sage.sat.converters.polybori.CNFEncoder method), 41
T
to_polynomial() (sage.sat.converters.polybori.CNFEncoder method), 42
V
var() (sage.sat.converters.polybori.CNFEncoder method), 43
var() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 30
var() (sage.sat.solvers.dimacs.DIMACS method), 15
var() (sage.sat.solvers.picosat.PicoSAT method), 24
var() (sage.sat.solvers.sat_lp.SatLP method), 26
var() (sage.sat.solvers.satsolver.SatSolver method), 9
W
write() (sage.sat.solvers.dimacs.DIMACS method), 15