Sage Reference Manual: Sat

Release 6.3

The Sage Development Team

August 11, 2014

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

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 a C++ interface to the *CryptoMiniSat* [CMS] SAT solver which can be used interchangably with DIMACS-based solvers, but also provides advanced features. For this, the optional CryptoMiniSat package must be installed, this can be accomplished by typing:

```
sage: install_package('cryptominisat') # not tested
```

and by running `sage -b` from the shell afterwards to build Sage's CryptoMiniSat extension module.

Since by default Sage does not include any SAT solver, we demonstrate key features by instantiating a fake DIMACS-based solver. We start with a trivial example:

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

In Sage's notation:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS(command="sat-solver")
sage: solver.add_clause( ( 1,  2,  3) )
sage: solver.add_clause( ( 1,  2, -3) )
```

Note: sage.sat.solvers.dimacs.DIMACS.add_clause() creates new variables when necessary. In particular, 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.

We demonstrate solving using CryptoMiniSat:

```
sage: from sage.sat.solvers import CryptoMiniSat # optional - cryptominisat
sage: cms = CryptoMiniSat()                      # optional - cryptominisat
sage: cms.add_clause((1,2,-3))                   # optional - cryptominisat
sage: cms()                                      # optional - cryptominisat
(None, True, True, False)
```

1.1 Details on Specific Solvers

1.1.1 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** [RS] and **Glucose** [GL] 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

Classes and Methods

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

Bases: `sage.sat.solvers.satsolver.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.

TESTS:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: DIMACS()
DIMACS Solver: ''
```

`__call__`(*assumptions=None*)

Run 'command' and collect output.

INPUT:

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

TESTS:

This class is not meant to be called directly:

```
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()
Traceback (most recent call last):
...
ValueError: No SAT solver command selected.
```

`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.

EXAMPLE:

```
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: ''
```

`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.

EXAMPLE:

```
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
```

nvars()

Return the number of variables.

EXAMPLE:

```
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`

EXAMPLE:

```
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.

EXAMPLE:

```
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`)

EXAMPLE:

```
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
```

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

Bases: `sage.sat.solvers.dimacs.DIMACS`

An instance of the Glucose solver.

For information on Glucose see: http://www.lri.fr/~simon/?page=glucose

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

Bases: `sage.sat.solvers.dimacs.DIMACS`

An instance of the RSat solver.

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

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
1 0
-2 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 [CB07], 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

REFERENCES:

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: `sage.sat.converters.anf2cnf.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.rins.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)

EXAMPLE:

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
1 0
-2 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 constructer 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

EXAMPLE:

```
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
1 0
-2 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
[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`

EXAMPLE:

```
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
1 0
-2 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

EXAMPLE:

```
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`

EXAMPLE:

```
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
1 0
-2 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`.

EXAMPLE:

```
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: 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 (*length*, *equal_zero*)

Return permutations of length `length` which are equal to zero if `equal_zero` and equal to one otherwise.

A variable is false if the integer in its position is smaller than zero and true otherwise.

INPUT:

- `length` - the number of variables
- `equal_zero` - should the sum be equal to zero?

EXAMPLE:

```
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.permutations(3, True)
[[-1, -1, -1], [1, 1, -1], [1, -1, 1], [-1, 1, 1]]

sage: ce.permutations(3, False)
[[1, -1, -1], [-1, 1, -1], [-1, -1, 1], [1, 1, 1]]
```

phi

Map SAT variables to polynomial variables.

EXAMPLE:

```
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?

EXAMPLE:

```
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], Tr

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

EXAMPLE:

```
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 deicison variable?

EXAMPLE:

```
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*)

Divides the zero set of `f` into blocks.

EXAMPLE:

```
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(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: F,s = sr.polynomial_system()
sage: from sage.sat.boolean_polynomials import solve as solve_sat # optional - cryptominisat
sage: s = solve_sat(F)                                            # optional - cryptominisat
sage: F.subs(s[0])                                                # optional - cryptominisat
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.

EXAMPLE:

```
sage: from sage.sat.boolean_polynomials import learn as learn_sat # optional - cryptominisat
```

We construct a simple system and solve it:

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

We construct a slightly larger equation system and recover some equations after 20 restarts:

```
sage: set_random_seed(2300)                            # optional - cryptominisat
sage: sr = mq.SR(1,4,4,4,gf2=True,polybori=True)   # optional - cryptominisat
sage: F,s = sr.polynomial_system()                     # optional - cryptominisat
sage: from sage.sat.boolean_polynomials import learn as learn_sat # optional - cryptominisat
sage: H = learn_sat(F, s_maxrestarts=20, interreduction=True)     # optional - cryptominisat
sage: H[-1]                                            # optional - cryptominisat, output random
k001200*s031*x011201 + k001200*x011201
```

Note: This function is meant to be called with some parameter such that the SAT-solver is interrupted. For CryptoMiniSat this is max_restarts, so pass 'c_max_restarts' to limit the number of restarts CryptoMiniSat will attempt. If no such parameter is passed, then this function behaves essentially like `solve()` except that this function does not support `n>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.

EXAMPLE:

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

```
sage: sr = mq.SR(1,1,1,4,gf2=True,polybori=True)
sage: F,s = sr.polynomial_system()
```

and pass it to a SAT solver:

```
sage: from sage.sat.boolean_polynomials import solve as solve_sat # optional - cryptominisat
sage: s = solve_sat(F)                                            # optional - cryptominisat
sage: F.subs(s[0])                                                # optional - cryptominisat
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, s_verbosity=1, c_max_vars_sparse=4, c_cutting_number=8) # optional - cryp
sage: F.subs(s[0])                                                             # optional - cryp
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 - cryptominisat
sage: f = a*b                           # optional - cryptominisat
sage: l = solve_sat([f],n=1)            # optional - cryptominisat
sage: len(l) == 1, f.subs(l[0])         # optional - cryptominisat
(True, 0)

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

sage: sorted(solve_sat([a*b],n=3))          # optional - cryptominisat
[{b: 0, a: 0}, {b: 0, a: 1}, {b: 1, a: 0}]
sage: sorted(solve_sat([a*b],n=4))          # optional - cryptominisat
[{b: 0, a: 0}, {b: 0, a: 1}, {b: 1, a: 0}]
sage: sorted(solve_sat([a*b],n=infinity))   # optional - cryptominisat
[{b: 0, a: 0}, {b: 0, a: 1}, {b: 1, a: 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 - cryptominisat
sage: R.<a,b,c,d> = BooleanPolynomialRing()                       # optional - cryptominisat
sage: F = [a+b,a+c+d]                                             # optional - cryptominisat
```

First the normal use case:

```
sage: solve_sat(F,n=infinity)                                     # optional - cryptominisat
[{d: 0, b: 0, c: 0, a: 0}, {d: 1, b: 1, c: 0, a: 1},
```

```
 {d: 1, b: 0, c: 1, a: 0}, {d: 0, b: 1, c: 1, a: 1}]
```

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 - cryptominisat
[{b: 0, a: 0}, {b: 1, a: 1}]
```

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

[CB07] Nicolas Courtois, Gregory V. Bard: Algebraic Cryptanalysis of the Data Encryption Standard, In 11-th IMA Conference, Cirencester, UK, 18-20 December 2007, Springer LNCS 4887. See also http://eprint.iacr.org/2006/402/.

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

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

[CMS] http://www.msoos.org/cryptominisat2/

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

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

D

G

L

M

N

P

R

S

T

V

W

Z