Clauses and Beyond: On Fast Prototyping with SAT Oracles

Alexey Ignatiev

MONASH University

September 8, 2021

Motivation


Logic Programming
Automated Theorem Proving
Answer Set Programming
CDCL-inspired engine
Reasoning
Interpolation
Instantiation
stable models
Mixed Integer Programming
Branch & Bound
Superposition
Resolution
CDCL for PBO
Pseudo-Boolean Optimization
CDCL
SAT
MaxSAT
Arrays
Optimization modulo theories
Bitvectors
CDCL(T)
cutting planes
LCG
global constraints
custom search
propagators
Satisfiability Modulo Theories
Constraint Satisfaction and Optimization
Integers
EUF
Reals
MiniZinc
Local search

The SAT disrupion

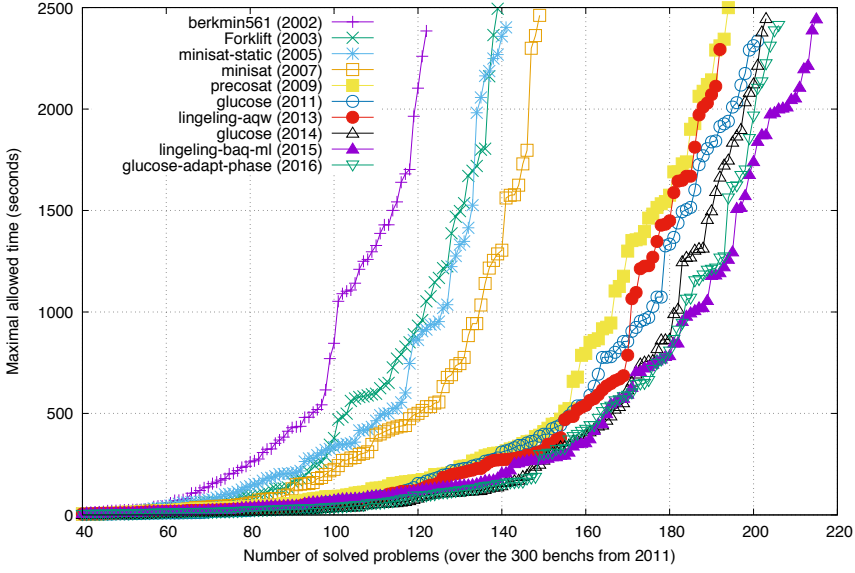
- **SAT is NP-complete**

The SAT disrupion

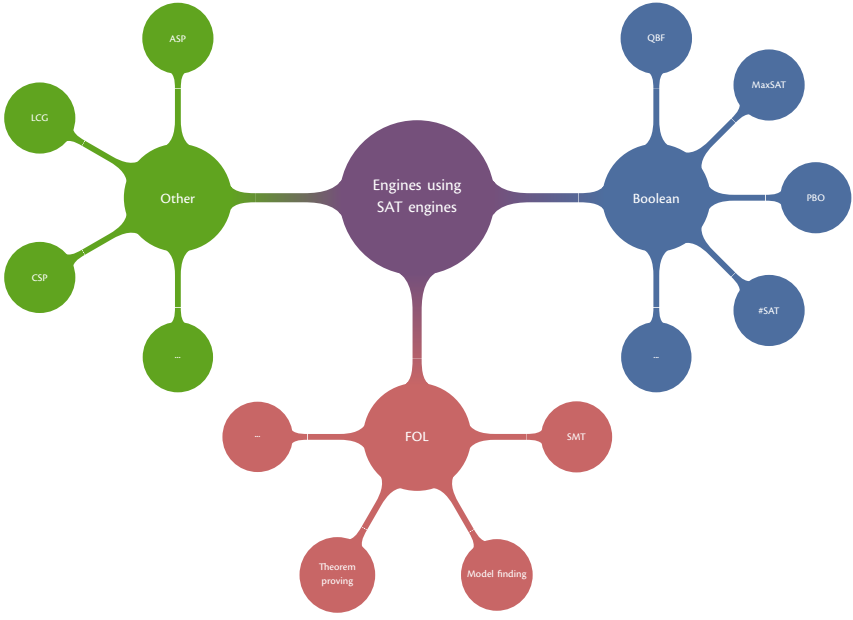
- SAT is NP-complete
  - But CDCL SAT solving is a success story of Computer Science
  - Hundreds (thousands?) of practical applications

SAT solver improvement

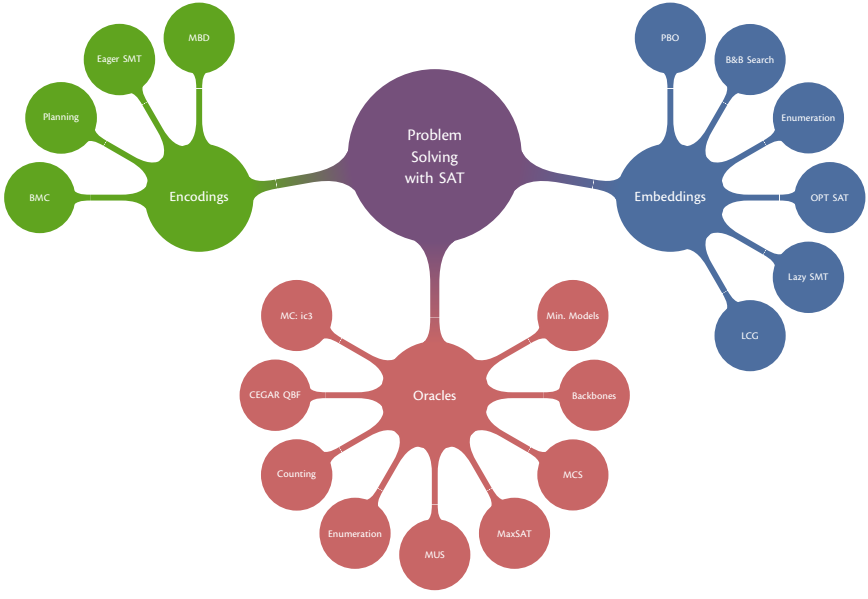
[Source: Simon 2015]



SAT is the engines' engine



SAT is ubiquitous in problem solving

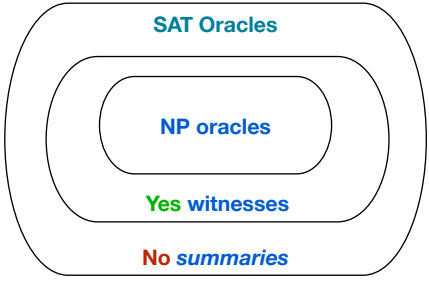


How to solve problems with SAT?

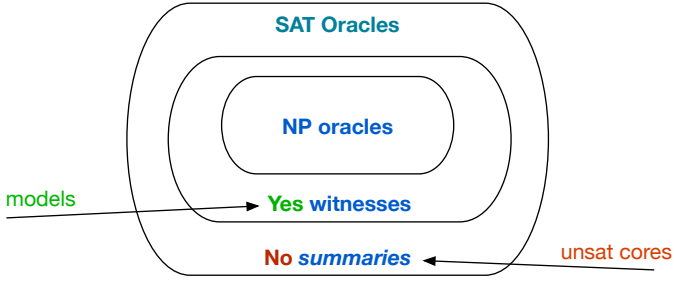
- Use SAT solvers as oracles
- Should be quick to prototype
- Should be reasonably efficient
- Should enable fiddling with the algorithms
- Avoid steep learning curves
- Combine with other technologies
- ...

SAT Oracles

What are SAT oracles?



What are SAT oracles?



Where are we using SAT oracles?

MaxSAT

Where are we using SAT oracles?

MaxSAT

MinSAT; Maximal Falsifiability

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

MSMP

Where are we using SAT oracles?

MaxSAT	MCS Extraction & Enumeration	MUS Extraction & Enumeration
MinSAT; Maximal Falsifiability	Lean Kernels; Backbones	MSMP

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

MES Extraction

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

MES Extraction

Maximum Cliques

Where are we using SAT oracles?

MaxSAT

MCS Extraction & Enumeration

MUS Extraction & Enumeration

MinSAT; Maximal Falsifiability

Lean Kernels; Backbones

MSMP

Function Minimization; Prime Enumeration; Propositional Abduction;

Model-based Diagnosis; Axiom Pinpointing; Package Management;

QBF & QMaxSAT

MES Extraction

Maximum Cliques

Explainable AI

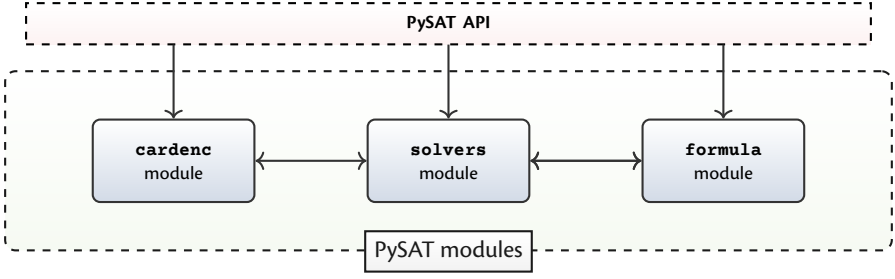
Some challenges

- Low-level (C/C++, even Java) implementations are important:
  - **Iterative** SAT solving
  - Often using **incremental** SAT
  - Need to analyze **models**
  - Need to extract **unsatisfiable cores**
  - **Many practical successes**

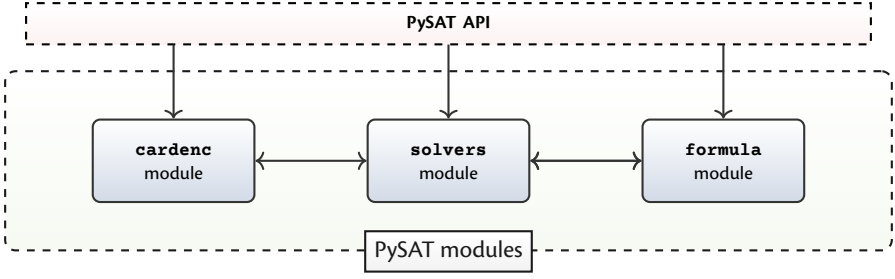
- But low-level implementations can be **problematic**:
  - Development time
  - Error prone
  - Difficult to maintain & change
  - ...

PySAT

Overview of PySAT

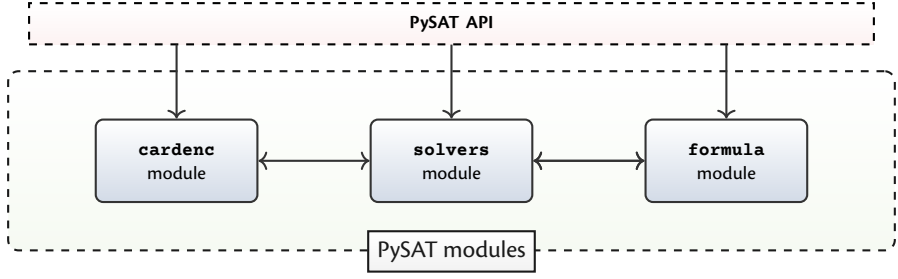


Overview of PySAT



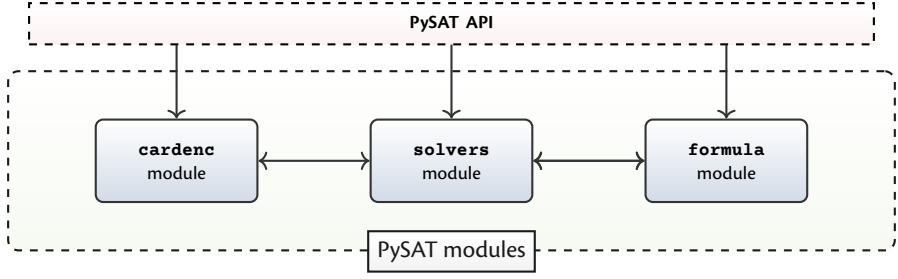
- **Open source, available on github**

Overview of PySAT



- **Open source**, **available on github**
- Comprehensive list of **SAT solvers**
- Comprehensive list of **cardinality encodings**
- Fairly **comprehensive documentation**
- A number of **examples**

Overview of PySAT



- **Open source**, **available on github**
- Comprehensive list of **SAT solvers**
- Comprehensive list of **cardinality encodings**
- Fairly **comprehensive documentation**
- A number of **examples**
- Optional (external) **pb** module

Formula manipulation (`pysat.formula` module)

Features
CNF & Weighted CNF (WCNF)
Read formulas from file/string
Write formulas to file
Append clauses to formula
Negate CNF formulas
Translate between CNF and WCNF
Non-clausal formula support
ID manager

- **URL**: `https://pysathq.github.io/docs/html/api/formula.html`

`pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq 1$ while $x_5 \triangleq 5$

`pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq 1$ while $x_5 \triangleq 5$
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq -1$ while $x_5 \triangleq 5$

`pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq$ `1` while $x_5 \triangleq$ `5`
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq$ `-1` while $x_5 \triangleq$ `5`
- **a clause is a *list* of literals**
  - clause $(\neg x_3 \vee x_2) \triangleq$ `[-3, 2]`

`pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq$ `1` while $x_5 \triangleq$ `5`
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq$ `-1` while $x_5 \triangleq$ `5`
- **a clause is a *list* of literals**
  - clause $(\neg x_3 \vee x_2) \triangleq$ `[-3, 2]`
- **a CNF formula is an object of class `CNF`**
  - i.e. basically it is a *list of clauses*

`pysat.formula` module

- **a variable in PySAT is a number from $\mathbb{N}_{>0}$**
  - e.g. $x_1 \triangleq 1$ while $x_5 \triangleq 5$
- **a literal is an integer**
  - e.g. $\neg x_1 \triangleq -1$ while $x_5 \triangleq 5$
- **a clause is a *list* of literals**
  - clause $(\neg x_3 \vee x_2) \triangleq$ `[-3, 2]`
- **a CNF formula is an object of class `CNF`**
  - i.e. basically it is a *list of clauses*
- `IDPool` is a variable ID manager

```
>>> from pysat.formula import IDPool
>>> vpool = IDPool(occupied=[[12, 18], [3, 10]])
>>>
>>> # creating 5 unique variables for the following strings ('v1', 'v2', ..., 'v5')
>>> for i in range(5):
...     print(vpool.id('v{0}'.format(i + 1)))
1
2
11
19
20
```

`pysat.formula.CNF` basic example

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

`pysat.formula.CNF` basic example

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

- + **`WCNF`** and **`WCNFPlus`** formulas!

`pysat.formula.CNF` basic example

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

- + **`WCNF`** and **`WCNFPlus`** formulas!
- much more functionality!
  (`https://pysathq.github.io/docs/html/api/formula.html`)

`pysat.formula.CNF` example (encoding PHP)

$$x_{ij} = \texttt{true} \Leftrightarrow \text{pigeon } i \text{ is at hole } j$$

$$\mathsf{PHP}_m \triangleq \bigwedge_{i=1}^{m+1} \mathsf{AtLeast1}(x_{i1}, \ldots, x_{im}) \land \bigwedge_{j=1}^{m} \mathsf{AtMost1}(x_{1j}, \ldots, x_{m+1,j})$$

pysat.formula.CNF example (encoding PHP)

$$x_{ij} = \mathtt{true} \Leftrightarrow \text{pigeon } i \text{ is at hole } j$$

$$\mathsf{PHP}_m \triangleq \bigwedge_{i=1}^{m+1} \mathsf{AtLeast1}(x_{i1}, \ldots, x_{im}) \wedge \bigwedge_{j=1}^{m} \mathsf{AtMost1}(x_{1j}, \ldots, x_{m+1,j})$$

```python
import itertools
from pysat.formula import CNF, IDPool

def PHP(nof_holes):
    cnf = CNF()

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

    holes   = list(range(1, nof_holes + 1))  # 1 .. m
    pigeons = list(range(1, nof_holes + 2))  # 1 .. m + 1

    for i in pigeons:  # at least one hole for each pigeon
        cnf.append([x(i, j) for j in range(1, nof_holes + 1)])

    for j in holes:  # at most one pigeon in a hole
        for comb in itertools.combinations(pigeons, 2):
            cnf.append([-x(i, j) for i in comb])

    return cnf
```

Non-clausal formulas

- **requires `py-aiger` package**

Non-clausal formulas

- **requires `py-aiger` package**

```
>>> import aiger
>>> x, y, z = aiger.atom('x'), aiger.atom('y'), aiger.atom('z')
>>> expr = ~(x | y) & z
>>> print(expr.aig)
aag 5 3 0 1 2
2
4
8
10
6 3 5
10 6 8
i0 y
i1 x
i2 z
o0 6c454aea-c9e1-11e9-bbe3-3af9d34370a9
>>>
>>> from pysat.formula import CNF
>>> cnf = CNF(from_aiger=expr.aig)
>>> print(cnf.clauses)
[[3, 2, 4], [-3, -4], [-2, -4], [-4, -1, 5], [4, -5], [1, -5]]
>>> print(['{0} <-> {1}'.format(v, cnf.vpool.obj(v)) for v in cnf.inps])
['3 <-> y', '2 <-> x', '1 <-> z']
>>> print(['{0} <-> {1}'.format(v, cnf.vpool.obj(v)) for v in cnf.outs])
['5 <-> 6c454aea-c9e1-11e9-bbe3-3af9d34370a9']
```

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally
- Tools can use multiple solvers, e.g. for hitting set dualization or CEGAR-based QBF solving

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally
- Tools can use multiple solvers, e.g. for hitting set dualization or CEGAR-based QBF solving
- Portfolio-like use of SAT oracles is possible

Available solvers (`pysat.solvers` module)

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version
MapleCM	SAT competition 2018
Maplesat	MapleCOMSPS_LRB
...	...

- Solvers can either be used incrementally or non-incrementally
- Tools can use multiple solvers, e.g. for hitting set dualization or CEGAR-based QBF solving
- Portfolio-like use of SAT oracles is possible
- **URL**: `https://pysathq.github.io/docs/html/api/solvers.html`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`
- `s.add_clause()` & `s.append_formula()`

`pysat.solvers` module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`
- `s.add_clause()` & `s.append_formula()`
- `s.get_model()` & `s.get_core()` & `get_proof()`

pysat.solvers module (basic interface)

- `s = Solver()` & `s.delete()`
- `with Solver() as s:` environment
- `s.solve()` & `s.solve_limited()`
- `s.add_clause()` & `s.append_formula()`
- `s.get_model()` & `s.get_core()` & `get_proof()`

```
>>> from pysat.solvers import Solver, Minisat22
>>>
>>> s = Solver(name='g4')  # accessing Glucose4 by name
>>> s.add_clause([-1, 2])
>>> s.add_clause([-1, -2])
>>> s.solve()
True
>>> print(s.get_model())
[-1, -2]
>>> s.delete()
>>>
>>> with Minisat22(bootstrap_with=[[-1, 2], [-1, -2]]) as m:
...     print(m.solve())
...     print(m.get_model())
True
[-1, -2]
```

`pysat.solvers` — incremental SAT

- SAT solver is used as a witness producing oracle

`pysat.solvers` — incremental SAT

- SAT solver is used as a witness producing oracle
  - satisfiable formula ➡ return a model

`pysat.solvers` — incremental SAT

- SAT solver is used as a witness producing oracle
  - satisfiable formula ⇒ return a model
  - unsatisfiable formula ⇨ return an unsatisfiable core

pysat.solvers — incremental SAT

- SAT solver is used as a witness producing oracle
  - satisfiable formula ⇒ return a model
  - unsatisfiable formula ⇨ return an unsatisfiable core

```
>>> from pysat.formula import CNF, IDPool
>>> from pysat.solvers import Solver

>>> vpool = IDPool()
>>> v = lambda obj: vpool.id(obj)

>>> cnf = CNF()
>>> cnf.extend([[-v('a'), v('b')], [-v('a'), v('c')]])
>>> cnf.extend([[v('a'), -v('s1')], [-v('b'), -v('c'), -v('s2')], [v('a'), -v('c'), -v('s3')], [v('a'), -v('b'), -v('s4')]])

>>> with Solver(bootstrap_with=cnf) as s:
...     if s.solve(assumptions=[v('s1'), -v('s2'), -v('s3'), v('s4')]) == True:
...         print('model:', ['{0}{1}'.format('' if v > 0 else '-', vpool.obj(abs(v))) for v in s.get_model()])
...
...     if s.solve(assumptions=[v('s1'), v('s2'), v('s3'), v('s4')]) == False:
...         print('ucore:', ['{0}{1}'.format('' if v > 0 else '-', vpool.obj(abs(v))) for v in s.get_core()])
...
model: ['a', 'b', 'c', 's1', '-s2', '-s3', 's4']
ucore: ['s2', 's1']
```

pysat.solvers — solve_limited()

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

pysat.solvers — solve_limited() and interrupt()

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

pysat.solvers — propagate()

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

- **+ lots of other cool stuff!**

  https://pysathq.github.io/docs/html/api/solvers.html

Available cardinality encodings (`pysat.card` module)

Name	Type
pairwise	AtMost1
bitwise	AtMost1
ladder	AtMost1
sequential counter	AtMostk
sorting network	AtMostk
cardinality network	AtMostk
totalizer	AtMostk
mtotalizer	AtMostk
kmtotalizer	AtMostk

- Also AtLeastK and EqualsK constraints
- **URL**: `https://pysathq.github.io/docs/html/api/card.html`

`pysat.card` module

- `atleast()`, `atmost()`, and `equals()`

`pysat.card` module

- `atleast()`, `atmost()`, and `equals()`
- `ITotalizer` — *iterative totalizer*

pysat.card module

- `atleast()`, `atmost()`, and `equals()`
- `ITotalizer` — *iterative totalizer*

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

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`
- *merge* with another totalizer, with `merge_with()`

`pysat.card` — `ITotalizer`

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`
- *merge* with another totalizer, with `merge_with()`
- set bound incrementally

pysat.card — ITotalizer

- encode $\sum x_i \leqslant K$ constraints
- extend left-hand side with `extend()`
- increase right-hand side with `increase()`
- *merge* with another totalizer, with `merge_with()`
- set bound incrementally

```
>>> from pysat.card import ITotalizer
>>>
>>> t = ITotalizer(lits=[1, 2], ubound=1)
>>> print(t.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4]]
>>> print(t.rhs)
[3, 4]
>>>
>>> t.extend(lits=[5], ubound=2)  # adding one more literal and increasing the bound
>>> print(t.cnf.clauses)
[[-2, 3], [-1, 3], [-1, -2, 4], [-5, 6], [-3, 6], [-4, 7], [-3, -5, 7], [-4, -5, 8]]
>>>
>>> print(t.cnf.clauses[-t.nof_new:])  # these are newly added clauses
[[-5, 6], [-3, 6], [-4, 7], [-3, -5, 7], [-4, -5, 8]]
>>>
>>> print(t.rhs)
[6, 7, 8]  # these literals represent bounds 0, 1, and 2
>>> t.delete()
```

More examples — `pysat.examples` module

- `models.py` — simple model enumerator

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor
- `optux.py` — smallest MUS extractor

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor
- `optux.py` — smallest MUS extractor
- `lbx.py` — LBX-like MCS enumeration

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor
- `optux.py` — smallest MUS extractor
- `lbx.py` — LBX-like MCS enumeration
- `mcsls.py` — CLD-like MCS enumeration

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor
- `optux.py` — smallest MUS extractor
- `lbx.py` — LBX-like MCS enumeration
- `mcsls.py` — CLD-like MCS enumeration
- `fm.py` — Fu&Malik MaxSAT algorithm

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor
- `optux.py` — smallest MUS extractor
- `lbx.py` — LBX-like MCS enumeration
- `mcsls.py` — CLD-like MCS enumeration
- `fm.py` — Fu&Malik MaxSAT algorithm
- `lsu.py` – LSU MaxSAT algorithm

More examples — `pysat.examples` module

- `models.py` — simple model enumerator
- `genhard.py` — a few hard combinatorial principles
- `hitman.py` — minimal hitting set enumerator
- `musx.py` — deletion-base MUS extractor
- `optux.py` — smallest MUS extractor
- `lbx.py` — LBX-like MCS enumeration
- `mcsls.py` — CLD-like MCS enumeration
- `fm.py` — Fu&Malik MaxSAT algorithm
- `lsu.py` – LSU MaxSAT algorithm
- **`rc2.py` — award-winning MaxSAT solver**

Encoding Sudoku

```python
def encode_sudoku():
    cnf = CNF()  # this is where we store the result formula

    pool = IDPool()  # pool of variable ids
    var = lambda i, j, v: pool.id(tuple([i + 1, j + 1, v + 1]))

    # at least one value in each cell
    for i, j in itertools.product(range(9), range(9)):
        cnf.append([var(i, j, val) for val in range(9)])

    # at most one value in each row
    for i in range(9):
        for val in range(9):
            for j1, j2 in itertools.combinations(range(9), 2):
                cnf.append([-var(i, j1, val), -var(i, j2, val)])

    # at most one value in each column
    for j in range(9):
        for val in range(9):
            for i1, i2 in itertools.combinations(range(9), 2):
                cnf.append([-var(i1, j, val), -var(i2, j, val)])

    # at most one value in each square
    for val in range(9):
        for i in range(3):
            for j in range(3):
                subgrid = itertools.product(range(3 * i, 3 * i + 3), range(3 * j, 3 * j + 3))
                for c in itertools.combinations(subgrid, 2):
                    cnf.append([-var(c[0][0], c[0][1], val), -var(c[1][0], c[1][1], val)])
```

Interaction with **plenty** of other libraries in Python!

- **Sudoku demo** — `pygtk`
- **MAPF demo** — `mcpi.minecraft`

Interaction with **plenty** of other libraries in Python!

- **Sudoku demo** — `pygtk`
- **MAPF demo** — `mcpi.minecraft`
- **math and data** — `numpy`, `scipy`, `pandas`, `gurobi`, ...

Interaction with **plenty** of other libraries in Python!

- **Sudoku demo** — `pygtk`
- **MAPF demo** — `mcpi.minecraft`
- **math and data** — `numpy`, `scipy`, `pandas`, `gurobi`, ...
- **XAI** — `scikit`, `pytorch`, `tensorflow`, ...
- ...

Interaction with **plenty** of other libraries in Python!

- **Sudoku demo** — `pygtk`
- **MAPF demo** — `mcpi.minecraft`
- **math and data** — `numpy`, `scipy`, `pandas`, `gurobi`, ...
- **XAI** — `scikit`, `pytorch`, `tensorflow`, ...
- ...
- **DL — ?**

Installation & platforms

- `$ pip install python-sat[pblib,aiger]`

Installation & platforms

- `$ pip install python-sat[pblib,aiger]`
- **Linux, MacOS, Windows**

Installation & platforms

- `$ pip install python-sat[pblib,aiger]`
- **Linux, MacOS, Windows**
- **no compilation needed**

Installation & platforms

- `$ pip install python-sat[pblib,aiger]`
- **Linux, MacOS, Windows**
- **no compilation needed**
- **+ online REPL platforms, e.g. repl.it**

Installation & platforms

- `$ pip install python-sat[pblib,aiger]`
- **Linux, MacOS, Windows**
- **no compilation needed**
- **+ online REPL platforms, e.g. repl.it**
- **+ PySAT is a part of Pyodide**
  (Python scientific stack compiled to WebAssembly)

https://pysathq.github.io/