PySAT: A Python Toolkit for Prototyping with SAT Oracles

A. Ignatiev, A. Morgado and **J. Marques-Silva**

University of Lisbon

Workshop on Theory and Practice of Satisfiability Solving

CMO, Oaxaca, México

August 2018

How to solve problems with SAT?

- Use SAT solvers as oracles
  - ASP, CP, SMT, ... often not an alternative
- Should be quick to prototype
- Should be reasonably efficient
- Should enable fiddling with the algorithms
- Avoid steep learning curves
- ...

This talk

- Problem solving with SAT oracles
- PySAT
  - Open-source Python API prototyping with SAT oracles
- Example(s)
- Some results

Outline

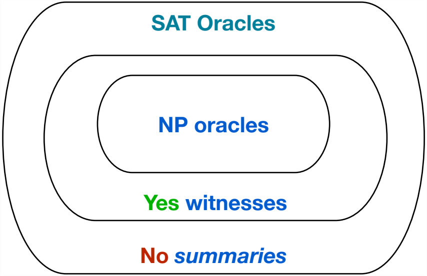
SAT Oracles

Introducing PySAT

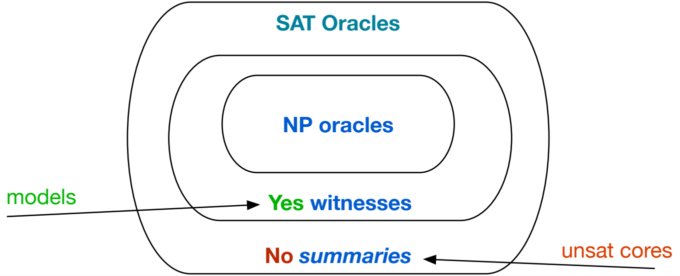
Using PySAT

Experimental Evidence

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

And also many tools ...

msuncore, mscg, mcsls, lbx,
muser, emus, bbones, forqes, bica,
minuc, hgmus, beacon, minds,
primer, primels, hyper, packup, ...

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
  - ...

Outline

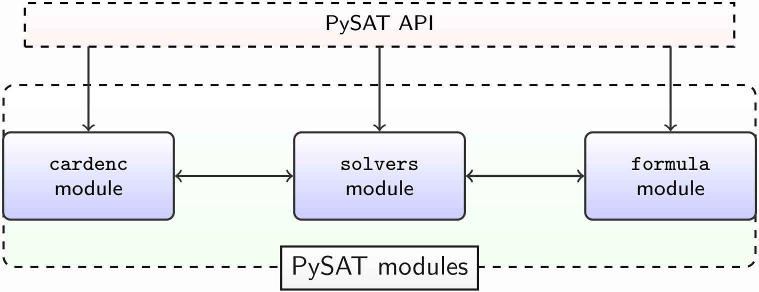
SAT Oracles

Introducing PySAT

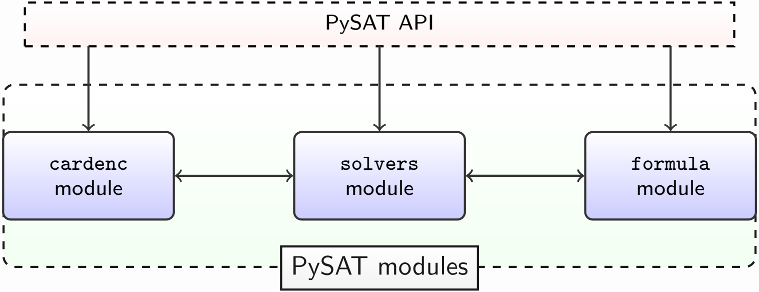
Using PySAT

Experimental Evidence

Overview of PySAT

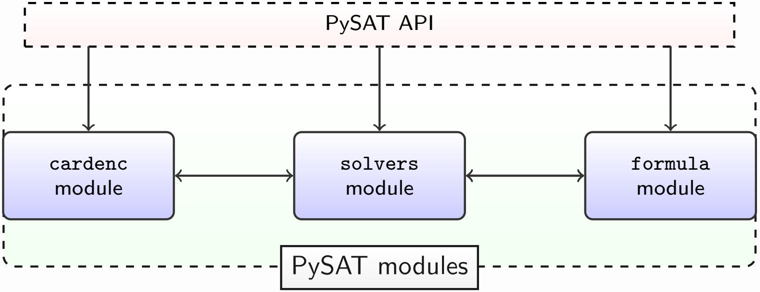


Overview of PySAT



- 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
- Several use cases

Available solvers

Solver	Version
Glucose	3.0
Glucose	4.1
Lingeling	bbc-9230380-160707
Minicard	1.2
Minisat	2.2 release
Minisat	GitHub version

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

Available cardinality encodings

Name	Type
pairwise	AtMost1
bitwise	AtMost1
ladder	AtMost1
sequential counter	AtMost$k$
sorting network	AtMost$k$
cardinality network	AtMost$k$
totalizer	AtMost$k$
mtotalizer	AtMost$k$
kmtotalizer	AtMost$k$

- Also AtLeast$K$ and Equals$K$ constraints
- **URL**: `https://pysathq.github.io/docs/html/api/card.html`

Formula manipulation

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
ID manager

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

Installation & info

- Installation:
  `$ [sudo] pip2|pip3 install python-sat`

- Website: https://pysathq.github.io/

Outline

Basic interface – Python3

```
>>> from pysat.card import *
>>> am1 = CardEnc.atmost(lits=[1, -2, 3], encoding=EncType.pairwise)
>>> print(am1.clauses)
[[-1, 2], [-1, -3], [2, -3]]
>>>
>>> from pysat.solvers import Solver
>>> with Solver(name='m22', bootstrap_with=am1.clauses) as s:
...     if s.solve(assumptions=[1, 2, 3]) == False:
...         print(s.get_core())
[3, 1]
```

Example: encoding PHP

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

class PHP(CNF, object):
    def __init__(self, nof_holes, topv=0):
        # initializing CNF's internal parameters
        super(PHP, self).__init__()
        holes = range(1, nof_holes + 1)  # [1, ..., nof_holes]
        pigeons = range(1, nof_holes + 2)  # [1, ..., nof_holes + 1]
        vpool = IDPool(start_from=topv + 1)    # Pool of var IDs
        var = lambda i, j: vpool.id('v_{0}_{1}'.format(i, j))
        for i in pigeons:    # Place every pigeon in hole
            self.append([var(i, j) for j in holes])
        for j in holes:    # No more than 1 pigeon in each hole
            for comb in itertools.combinations(pigeons, 2):
                self.append([-var(i, j) for i in comb])

def main():
    cnf = PHP(int(sys.argv[1]))
    cnf.to_file("php-gen.cnf")

if __name__== "__main__":
  main()
```

Example: naive (deletion) MUS extraction

Input : Set $\mathcal{F}$
Output: Minimal subset $\mathcal{M}$
begin
  $\mathcal{M} \leftarrow \mathcal{F}$
  foreach $c \in \mathcal{M}$ **do**
    if $\neg\text{SAT}(\mathcal{M} \setminus \{c\})$ **then**
      $\mathcal{M} \leftarrow \mathcal{M} \setminus \{c\}$ // If $\neg\text{SAT}(\mathcal{M} \setminus \{c\})$, then $c \notin$ MUS
  return $\mathcal{M}$ // Final $\mathcal{M}$ is MUS
end

- Number of predicate tests: $\mathcal{O}(m)$ [CD91,BDTW93]

Naive MUS extraction I

```python
def main():
    cnf = CNF(from_file=argv[1])   # create a CNF object from file
    (rnv, assumps) = add_assumps(cnf)

    oracle = Solver(name='g3', bootstrap_with=cnf.clauses)

    mus = find_mus(assumps, oracle)
    mus = [ref - rnv for ref in mus]
    print("MUS: ", mus)

if __name__ == "__main__":
  main()
```

Naive MUS extraction II

```python
def add_assumps(cnf):
    rnv = topv = cnf.nv
    assumps = []                     # list of assumptions to use
    for i in range(len(cnf.clauses)):
        topv += 1
        assumps.append(topv)         # register literal
        cnf.clauses[i].append(-topv) # extend clause with literal
    cnf.nv = cnf.nv + len(assumps)   # update # of vars
    return rnv, assumps


def main():
    cnf = CNF(from_file=argv[1])   # create a CNF object from file
    (rnv, assumps) = add_assumps(cnf)

    oracle = Solver(name='g3', bootstrap_with=cnf.clauses)

    mus = find_mus(assumps, oracle)
    mus = [ref - rnv for ref in mus]
    print("MUS: ", mus)

if __name__=="__main__":
  main()
```

Naive MUS extraction III

```
from sys import argv

from pysat.formula import CNF
from pysat.solvers import Solver


def find_mus(assmp, oracle):
    i = 0
    while i < len(assmp):
        ts = assmp[:i] + assmp[(i+1):]
        if not oracle.solve(assumptions=ts):
            assmp = ts
        else:
            i += 1
    return assmp
```

Naive MUS extraction III

```
from sys import argv

from pysat.formula import CNF
from pysat.solvers import Solver


def find_mus(assmp, oracle):
    i = 0
    while i < len(assmp):
        ts = assmp[:i] + assmp[(i+1):]
        if not oracle.solve(assumptions=ts):
            assmp = ts
        else:
            i += 1
    return assmp
```

Demo

A less naive MUS extractor

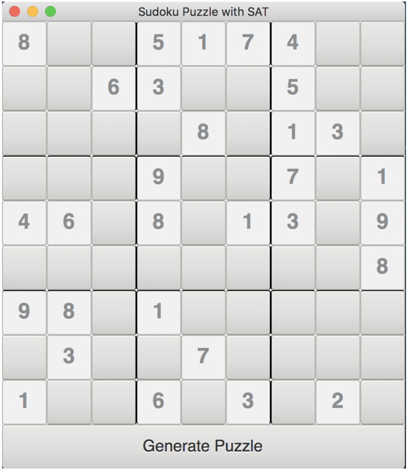
```
def clset_refine(assmp, oracle):
    assmp = sorted(assmp)
    while True:
        oracle.solve(assumptions=assmp)
        ts = sorted(oracle.get_core())
        if ts == assmp:
            break
        assmp = ts
    return assmp
# ...
def main():
    cnf = CNF(from_file=argv[1])    # create a CNF object from file
    (rnv, assumps) = add_assumps(cnf)

    oracle = Solver(name='g3', bootstrap_with=cnf.clauses)

    assumps = clset_refine(assumps, oracle)
    mus = find_mus(assumps, oracle)
    mus = [ref - rnv for ref in mus]
    print("MUS: ", mus)

if __name__== "__main__":
  main()
```

Encoding sudoku

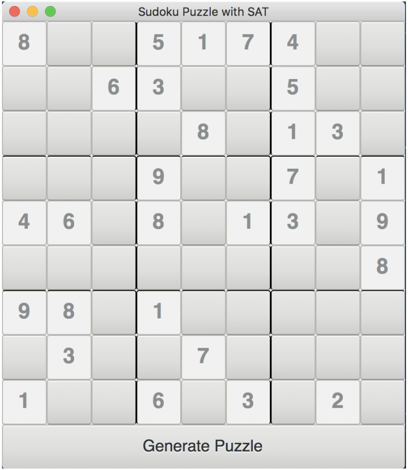
```
class SudokuEncoding(CNF, object):
  def __init__(self):
      # initializing CNF's internal parameters
      super(SudokuEncoding, self).__init__()
      self.vpool = IDPool()
      # at least one value in each cell
      for i, j in itertools.product(range(9), range(9)):
          self.append([self.var(i, 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):
                  self.append([-self.var(i, j1, val), -self.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):
                  self.append([-self.var(i1, j, val), -self.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):
                      self.append([-self.var(c[0][0],c[0][1],val),
                                   -self.var(c[1][0],c[1][1],val)])

  def var(self, i, j, v):
      return self.vpool.id(tuple([i + 1, j + 1, v + 1]))

  def cell(self, var):
      return self.vpool.obj(var)
```

A prototype sudoku game

A prototype sudoku game



A prototype sudoku game



Demo

Outline

Implementing a MaxSAT solver – FM06

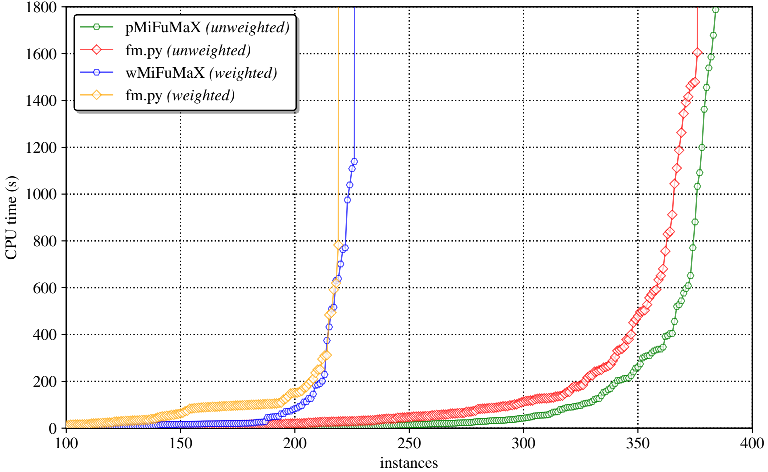
- First core-guided MaxSAT algorithm: Fu&Malik 2006
  - Soft clauses can be relaxed multiple times, and remain soft
  - Multiple AtMost1 constraints used
  - Lower bound refined until formula becomes satisfied
- Prototype with PySAT
- Compare with state of the art implementation – MiFuMaX
  - MiFuMaX won unweighted category in 2013 MaxSAT evaluation

Implementing a MaxSAT solver – FM06

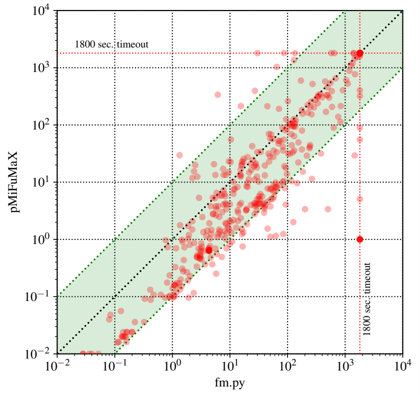
- First core-guided MaxSAT algorithm: Fu&Malik 2006
  - Soft clauses can be relaxed multiple times, and remain soft
  - Multiple AtMost1 constraints used
  - Lower bound refined until formula becomes satisfied
- Prototype with PySAT
- Compare with state of the art implementation – MiFuMaX
  - MiFuMaX won unweighted category in 2013 MaxSAT evaluation

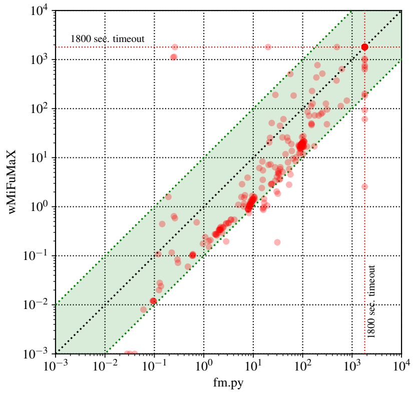
Code

Cactus plot


pMiFuMaX (unweighted)
fm.py (unweighted)
wMiFuMaX (weighted)
fm.py (weighted)
CPU time (s)
0
200
400
600
800
1000
1200
1400
1600
1800
100
150
200
250
300
350
400
instances

Scatter plots – unweighted + weighted





Conclusions & future plans

- The paper outlines the development of PySAT
  - Open source, publicly available through github: `https://pysathq.github.io/`
- PySAT has been used in recent projects:
  1. satclq (MaxClique solver) - IJCAI17
  2. minds (decision set learner) - IJCAR18
  3. mindt (decision tree learner) - IJCAI18
  4. rc2 (MaxSAT solver) - MaxSAT Evaluation 2018

Conclusions & future plans

- The paper outlines the development of PySAT
  - Open source, publicly available through github: `https://pysathq.github.io/`
- PySAT has been used in recent projects:
  1. satclq (MaxClique solver) - IJCAI17
  2. minds (decision set learner) - IJCAR18
  3. mindt (decision tree learner) - IJCAI18
  4. rc2 (MaxSAT solver) - MaxSAT Evaluation 2018
- Near term plans:
  - Integrate more SAT solvers & try to keep up with progress
  - Implement PB constraints
  - Reference implementations, e.g. MUS, MCS, MaxSAT, ...?
  - ...

Questions?