

111825

111779

4th April, 2017

SAT SOLVERS

SAT, also known as satisfiability. It is basically used to solve the decision problems that plague computer science.

GLUCOSE - 2012

In the application genre, this SAT solver was awarded the best single engine solver. It competed in the 2012 SAT competition using the 2.1 version. It is coded in C++ and it tries to incorporate the idea and solve the problem of multiple variables that arise from using multiple decision levels. Right now, it is being updated constantly and is at 4.1 version.

SUGAR - 2013

Sugar is a SAT based solver. It uses the encoding method to solve its problem. Their solution is similar to the Crawford and Baker method for scheduling solution. It solves a finite linear CSP and MAX-CSP. Different boolean methods were applied and an external SAT solver like miniSAT was incorporated in it as well. It won the Third International CSP Solver Competition.

RISS BLACKBOX - 2014

It is a highly configured SAT solver. It picks up a formula based on the input given. To make it work, CNF in a large amount have to be computed. From there, the configuration is set and the solver takes its course. It includes feature extraction and classification. It won the 2014 SAT Competition in the application where it was submitted as 64 bit binary to SAT.

CRYPTOMINISAT - 2015

[CryptoMiniSAT](#) was the winner of [SATRace 2015](#). It implements 3 interfaces, based in command-line, C++ and Python. It's a multi-threaded SAT solver, which among other things provides XOR manipulation, variable manipulation, data logging and display. It basically combines features of PrecoSat, MiniSat 2.0 core and Glucose SAT solvers.

RISS - 2016

RISS was the winner of [SAT race 2016 Agile Track](#). It works to solve formulae in CNF. It's mainly a research based project and provides a foundation for development of further techniques. Among other things, it provides the ability to search for solution with assumed literals, load and store clauses of a run and enumeration of solutions. It's based on CDCL procedure and works by first storing the data, handing it to the unit propagation module and then executing the desired operation.

SAT SOLVER CODE

FSM.PY

```
from __future__ import print_function

def even_ones(s):
    # Two status:

    laws = {(0, '0'): 0,
            (0, '1'): 1,
            (1, '0'): 1,
            (1, '1'): 0}

    status = 0
    for c in s:
        status = laws[status, c]
    return status == 0

# Example usage:
s = "1100110010"
print('Output {} = {}'.format(s, even_ones(s)))
```

SAT.PY

```
def solve(example, algorithm, meow=False):  
    n = len(example.variables)  
    excerpt = setup_excerpt(example)  
    if not excerpt:  
        return ()  
    ass = [None] * n  
    return algorithm.solve(example, excerpt, ass, 0, meow)  
  
def main():  
    args = parse_args()  
    example = None  
    with args.input as file:  
        example = SATInstance.from_file(file)  
  
    ass = solve(example, args.algorithm, args.meow)  
    adder = 0  
    for ass in ass:  
        if args.meow:  
            print('Found satisfying ass #{}:'.format(adder),  
                  file=stderr)  
        print(example.ass_to_string(ass,  
                                     brief=args.brief,  
                                     starting_with=args.starting_with))  
  
        adder += 1  
        if not args.all:  
            break  
  
    if args.meow and adder == 0:  
        print('No satisfying ass exists.', file=stderr)
```

Python Version
Your source code
Would you like to

SATISTANCE.PY

```
class SATInstance(object):
    def parse_and_add_clause(self, line):
        clause = []
        for literal in line.split():
            negated = 1 if literal.startswith('~') else 0
            variable = literal[negated:]
            if variable not in self.variable_table:
                self.variable_table[variable] = len(self.variables)
                self.variables.append(variable)
            encoded_literal = self.variable_table[variable] << 1 | negated
            clause.append(encoded_literal)
        self.clauses.append(tuple(set(clause)))

    def __init__(self):
        self.variables = []
        self.variable_table = dict()
        self.clauses = []

    @classmethod
    def from_file(cls, file):
        instance = cls()
        for line in file:
            line = line.strip()
            if len(line) > 0 and not line.startswith('#'):
                instance.parse_and_add_clause(line)
        return instance

    def literal_to_string(self, literal):
        s = '~' if literal & 1 else ''
        return s + self.variables[literal >> 1]

    def clause_to_string(self, clause):
```

RECURSIVE_SAT.PY


```
from __future__ import division
from __future__ import print_function

from sys import stderr

from excerpt import update_excerpt

def solve(example, excerpt, ass, d, meow):

    if d == len(example.variables):
        yield ass
        return

    for a in [0, 1]:
        if meow:
             print('Now {} = {}'.format(example.variables[d], a),
                file=stderr)
        ass[d] = a
        if update_excerpt(example,
                        excerpt,
                        (d << 1) | a,
                        ass,
                        meow):
            for a in solve(example, excerpt, ass, d + 1, meow):
                yield a


    ass[d] = None
```

ITERATIVE_SAT.PY

```
from __future__ import division
from __future__ import print_function

from sys import stderr

from excerpt import update_excerpt

def solve(example, excerpt, ass, d, meow):
    
    n = len(example.variables)
    state = [0] * n

    while True:
        if d == n:
            yield ass
            d -= 1
            continue

        tried_something = False
        for a in [0, 1]:
            if (state[d] >> a) & 1 == 0:
                if meow:
                    print('Now {} = {}'.format(example.variables[d], a),
                          file=stderr)
                tried_something = True
                # Set the bit indicating a has been tried for d
                state[d] |= 1 << a
                ass[d] = a
                if not update_excerpt(example, excerpt,
                                      d << 1 | a,
                                      ass,
                                      meow):
                    ass[d] = None
```


EXCERPT.PY

```
def dump_excerpt(example, excerpt):
    print('Current excerpt:', file=stderr)
    for l, w in enumerate(excerpt):
        literal_string = example.literal_to_string(l)
        clauses_string = ', '.join(example.clause_to_string(c) for c in w)
        print('{}: {}'.format(literal_string, clauses_string), file=stderr)
```

```
def setup_excerpt(example):
    excerpt = [deque() for __ in range(2 * len(example.variables))]
    for clause in example.clauses:
        # Make the clause watch its first literal
        excerpt[clause[0]].append(clause)
    return excerpt
```



```
def update_excerpt(example,
                    excerpt,
                    false_literal,
                    assignment,
                    meow):

    while excerpt[false_literal]:
        clause = excerpt[false_literal][0]
        found_alternative = False
        for alternative in clause:
            v = alternative >> 1
            a = alternative & 1
            if assignment[v] is None or assignment[v] == a ^ 1:
```