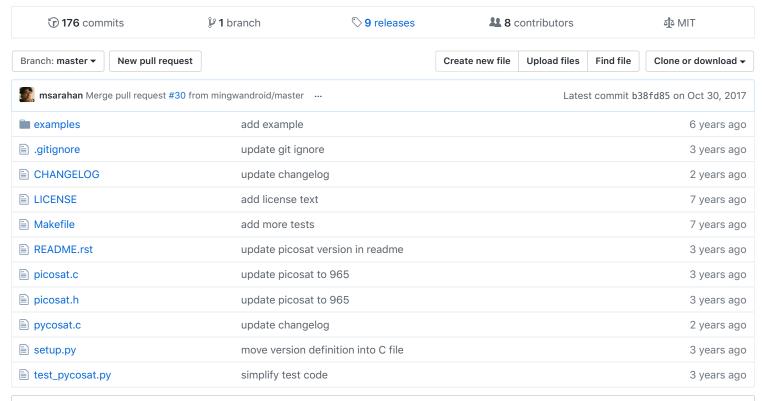
ContinuumIO / pycosat

Python bindings to picosat (a SAT solver)



README.rst

pycosat: bindings to picosat (a SAT solver)

PicoSAT is a popular SAT solver written by Armin Biere in pure C. This package provides efficient Python bindings to picosat on the C level, i.e. when importing pycosat, the picosat solver becomes part of the Python process itself. For ease of deployment, the picosat source (namely picosat.c and picosat.h) is included in this project. These files have been extracted from the picosat source (picosat-965.tar.gz).

Usage

The pycosat module has two functions solve and itersolve, both of which take an iterable of clauses as an argument. Each clause is itself represented as an iterable of (non-zero) integers.

The function solve returns one of the following:

- one solution (a list of integers)
- the string "UNSAT" (when the clauses are unsatisfiable)
- the string "UNKNOWN" (when a solution could not be determined within the propagation limit)

The function itersolve returns an iterator over solutions. When the propagation limit is specified, exhausting the iterator may not yield all possible solutions.

Both functions take the following keyword arguments:

- prop_limit : the propagation limit (integer)
- vars : number of variables (integer)
- verbose : the verbosity level (integer)

Example

Let us consider the following clauses, represented using the DIMACS cnf format:

```
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Here, we have 5 variables and 3 clauses, the first clause being $(x_1 \text{ or not } x_5 \text{ or } x_4)$. Note that the variable x_2 is not used in any of the clauses, which means that for each solution with x_2 = True, we must also have a solution with x_2 = False. In Python, each clause is most conveniently represented as a list of integers. Naturally, it makes sense to represent each solution also as a list of integers, where the sign corresponds to the Boolean value (+ for True and - for False) and the absolute value corresponds to ith variable:

```
>>> import pycosat
>>> cnf = [[1, -5, 4], [-1, 5, 3, 4], [-3, -4]]
>>> pycosat.solve(cnf)
[1, -2, -3, -4, 5]
```

This solution translates to: $x_1 = x_5 = \text{True}$, $x_2 = x_3 = x_4 = \text{False}$

To find all solutions, use itersolve:

```
>>> for sol in pycosat.itersolve(cnf):
...     print sol
...
[1, -2, -3, -4, 5]
[1, -2, -3, 4, -5]
[1, -2, -3, 4, 5]
...
>>> len(list(pycosat.itersolve(cnf)))
18
```

In this example, there are a total of 18 possible solutions, which had to be an even number because x_2 was left unspecified in the clauses.

The fact that itersolve returns an iterator, makes it very elegant and efficient for many types of operations. For example, using the itertools module from the standard library, here is how one would construct a list of (up to) 3 solutions:

```
>>> import itertools
>>> list(itertools.islice(pycosat.itersolve(cnf), 3))
[[1, -2, -3, -4, 5], [1, -2, -3, 4, -5], [1, -2, -3, 4, 5]]
```

Implementation of itersolve

How does one go from having found one solution to another solution? The answer is surprisingly simple. One adds the *inverse* of the already found solution as a new clause. This new clause ensures that another solution is searched for, as it *excludes* the already found solution. Here is basically a pure Python implementation of itersolve in terms of solve:

```
def py_itersolve(clauses): # don't use this function!
  while True:  # (it is only here to explain things)
    sol = pycosat.solve(clauses)
    if isinstance(sol, list):
        yield sol
        clauses.append([-x for x in sol])
    else: # no more solutions -- stop iteration
        return
```

This implementation has several problems. Firstly, it is quite slow as pycosat.solve has to convert the list of clauses over and over again. Secondly, after calling py_itersolve the list of clauses will be modified. In pycosat, itersolve is implemented on the C level, making use of the picosat C interface (which makes it much, much faster than the naive Python implementation above).