CSC 320 REPORT

Ushanth Loganathan

Kushal Patel

This report is about the Sudoku SAT solver that solves a 9X9 grid Sudoku puzzle. We have used pycosat to solve the 9X9 puzzle. It is considered an efficient encoder and our program solves the Sudoku with it. The developers have PicoSAT included in a package called the Anaconda which is available online. (link to download is given in the README file). Although PicoSAT is written in C, the Anaconda package binds it to Python, which is one key reason for its efficient performance.

PicoSAT (pycosat)

PicoSAT is a popular SAT solver written by Armin Biere in pure C. The Picosat is a solver that is available in the Python language package. It is implemented with c and the files namely picosat.c and picosat.h are included in the project.

Usage

The picosat has the two functions which are known as solve and itersolve. Both take list or iterable of clauses as an argument. Each clause is itself represented as a list 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

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 (x₁ or not x₅ or x₄). Note that the variable x₂ is not used in any of the clauses, which means that for each solution with x₂ = True, we must also have a solution with x₂ = 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: x₁ = x₅ = True, x₂ = x₃ = x₄ = 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 x₂ was left unspecified in the clauses.

The itersolve returns an iterator and makes it efficient for many types of operations.the following example shows how the itertools module from library construct a list with 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

After we find one solution in order to find a new solution we add the inverse of the found solution to the new clause.

This new clause ensures that another solution is searched for, as it *excludes* the already found solution.

Implementations 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
```

There are some problems related to this implementation:

1. The picosat.solve is slow and needs to transform the list of clauses many times over and over again.
2. After calling py_itersolve the list of clauses will be modified

However, the picosat itersolve is implemented in C and this is way more faster than python implementation.

More N X N Sudoku Puzzles

The size of the input puzzle will impact the size of the encoded Boolean equation. It is expected to increase exponentially with respect to the number of clauses in the result encoding. The following table shows the different number of clauses required for different sized puzzles.

Size of input puzzle	Number of clauses with pycosat
9 x 9	11745
16 x 16	123129
25 x 25	750628
36 x 36	3267226
49 x 49	11296703
64 x 64	33034237
81 x 81	85037095

The table clearly shows that with the input size growth, the number of clauses also increase vastly. This means ‘the larger the puzzle, the longer it will take the solver to solve it.’ This does not vary with the encoding type (minimal or efficient) as clauses increase no matter the encoding type.