PA 5 - Logic

Julien Blanchet | CS 276 F20 (Artificial Intelligence) | Prof. Li.

October 25, 2020

Description

This assignment called for us to implement a solver for propositional logic satisfiability problems. My approach uses object-orientation to provide an API for solving the problems.

`SAT.py` contains the code that implements the GSAT and WalkSAT algorithms. The algorithms are acessible via the `sat.walksat()` and `sat.gsat()` methods.

Since these algorithms are quite similar, they share much of the same code - `sat._sat_common()` contains the core iterative logic and takes as input a function that decides which variable to flip in the next iteration. The difference between *GSAT* and *WalkSAT* lies in how they chose which variable to flip: *GSAT* looks at all the variables and chooses the one that will satisfy the most clauses, whereas *WalkSAT* picks an unsatisfied clause at random and chooses the variable within the clause that would satisfy the greatest number of clauses if flipped.

In addition to the core algorithms, my implementation contains a few additional features to help with debugging:

- A listing of current assignments of variables can be printed along with the original variable names.
- The algorithm can be run on a `.cnf` file or a passed string.
- The program keeps track of statistics like run time and flip count.

Evaluation

Both the *GSAT* AND *WalkSAT* implementations are functional, and as expected, the *WalkSAT* algorithm is significantly faster than the *GSAT* algorithm. I crafted some basic unit tests (executable with `pytest`) in `test_sat.py`, and manually ran more complex tests using the `*.cnf` files provided and the `solve_sodoku.py`script. The following is a table of the runtimes of all manual test cases I ran:

Algorithm	Puzzle	Solve Time	Flips
GSAT	one_cell	~0	3
WalkSAT	one_cell	~0	3
GSAT	all_cells	346s	531
WalkSAT	all_cells	0.79s	297
GSAT	rows	aborted after 1041s	

Algorithm	Puzzle	Solve Time	Flips
WalkSAT	rows	4.14s	823
WalkSAT	rows_and_cols	14.28s	2659
WalkSAT	rules	44.08s	7149
WalkSAT	puzzle1	503.3s	99725
WalkSAT	puzzle2	502.9s	99999

A few points of analysis: * $WalkSAT$ is much faster $GSAT$. This is expected - $GSAT$ looks through a larger number of options (all variables) when considering which one to flip, whereas $WalkSAT$ only looks at a few (however many are in the one clause it chose). * The time spent seems to increase exponentially with the difficulty of the puzzle. * Both algorithms are able to output valid solutions for the puzzles.

To investigate areas for potential future improvement, I ran used python's builtin `cProfile` tool. Here is an exerpt of the output:

```
9867519 function calls (9867468 primitive calls) in 5.416 seconds
  ncalls  tottime  percall cumtime  percall filename:lineno(function)
  ---
     823    0.003    0.000    3.934    0.005 SAT.py:101(pick_flip_var)
       1    0.018    0.018    5.376    5.376 SAT.py:109(_sat_common)
       1    0.000    0.000    0.000    0.000 SAT.py:112(<listcomp>)
     729    0.000    0.000    0.000    0.000 SAT.py:12(randbool)
   95147    0.608    0.000    1.423    0.000 SAT.py:141  (get_unsatisfied_clauses)
    2326    1.688    0.001    3.922    0.002 SAT.py:146  (count_unsatisfied_clauses)
       1    0.000    0.000    0.000    0.000 SAT.py:15(SAT)
 9695700    3.049    0.000    3.049    0.000 SAT.py:153(is_clause_satisfied)
     730    0.000    0.000    0.000    0.000 SAT.py:166  (generate_solution_lines)
       1    0.000    0.000    0.001    0.001 SAT.py:172(write_solution)
       1    0.000    0.000    0.000    0.000 SAT.py:181(stats_str)
    7290    0.004    0.000    0.006    0.000 SAT.py:184(_parse_token)
       1    0.000    0.000    0.004    0.004 SAT.py:2(<module>)
       1    0.006    0.006    0.023    0.023 SAT.py:23(__init__)
    3078    0.002    0.000    0.002    0.000 SAT.py:55(_add_missing_variables)
    3078    0.004    0.000    0.007    0.000 SAT.py:62(_add_cnf_clause)
   10368    0.003    0.000    0.003    0.000 SAT.py:64(generate_clause_dict)
     823    0.005    0.000    3.928    0.005 SAT.py:71(select_best_variable)
       1    0.000    0.000    5.376    5.376 SAT.py:99(walksat)
```

As you can see, the majority of the computation is spent determining and counting which clauses are satisfied vs unsatisfied. Therefore, I believe that optimizing the clause-testing code would yield significant performance benefits.