Automated Reasoning Project Report
Daniel Bodin and Matthew Dutton
5/1/2024

Project 1: CDCL implementation with VSIDS

1. Overview

We implemented a CDCL solver with two watched literals using Python. We adapted this Python solver https://kienyew.github.io/CDCL-SAT-Solver-from-Scratch/ by adding a VSIDS heuristic. The original solver chose an unassigned variable at random during the during the decision step. The VSIDS heuristic makes this choice algorithmically by assigning a priority to each variable.

Our project is organized so that most of our CDCL solving logic is done in cdcl_solver.py. This file contains the functions needed for CDCL including the logic for the VSIDS implementation. The original solver we based our implementation on can be found at cdcl_solver_original.py. To see the affect VSIDS has on this solver, run the tests on these two files and compare their speeds. This is done below.

Additional files include common_classes.py. This file is shared between multiple other files in our project. It stores the definitions for a handful of classes fundamental to SAT solving, such as formulas, literals, etc. For testing, we implemented testall.py to run all the test cases on either cdcl_solver_original or cdcl_solver (our implementation). This automatically runs all the tests, so they do not have to be manually entered. Lastly, our project includes main.py. This file is used to run individual tests on either cdcl_solver_original or cdcl_solver. It uses system arguments to determine which solver to use.

2. VSIDS

The VSIDS heuristic prioritizes the more relevant variables, and it weighs newer clauses more heavily. Because of this our group figured implementing this heuristic would be advantageous. A random search of unassigned variables would be inefficient and wouldn't utilize any of the information about the variables. Therefore, the VSIDS heuristic allows the receiver to make a more informed decision.

The VSIDS heuristic is implemented in the VSIDS class in cdcl_solver. This class contains a priority queue of variable/score pairs. The score is the variables weight, since it is the number of occurrences of that variable. These scores increase when additional clauses are added. The scores are stored in a priority queue using the Python heapq module (https://docs.python.org/3/library/heapq.html). This module allows for efficient and easy to use heap operations.

An additional step of the VSIDS heuristic is the decaying of priority. This involves decreasing all the scores in the priority queue by a fixed percentage, after a fixed amount of time. The amount the scores decrease and the amount of time between decreases are greatly important to the effectiveness of the VSIDS heuristic. The amount of time between decays in our implementation depends on the number of conflicts encountered. When enough conflicts are encountered, all scores are halved.

3. Test Script

In order to repeatably run our solver we implemented a to do that for us. To run the 'testall.py' script you must pass in two arguments. The first is the solver you want to use. The original, or our implementation of VSIDS heuristic on top of the original. This is achieved by passing in 'original' or 'vsids' as an argument. The second is the amount of time in seconds you want the solver to try and find the satisfiability of one of the problems. This number is how long the test script will spend trying to solve the problem and it will timeout if it takes longer. Because some of the problems take a long time to solve, we did not want to wait for all of them to complete. So, by implementing this we can run the solvers on all the simpler problems relatively quickly. Refer to the readme for more specific commands of how to run all of the tests. The test script times how long it takes for each problem to be solved and saves it to a table. After running all the problems it prints the tables so the user can view how long it took to solve each test.

4. Testing Data

The following testing was done on a base 2020 MacBook Pro.

Processor: 1.4GHz Quad-Core Intel Core i5

Memory: 8 GB 2133 MHz LPDDR3

Original Python solver on revised tests with 180 second timeout:

SAT Files	Result	Execution Time
sqrt1042441.cnf	sat	0.060552
prime121.cnf	sat	0.833240
elimredundant.cnf	sat	0.002067
prime169.cnf	sat	1.808916
prime961.cnf	sat	4.492535
sqrt10201.cnf	sat	0.134692
sat10.cnf	sat	0.002448
sat12.cnf	sat	0.001077
uf20-0100.cnf	sat	0.004059
uf20-0101.cnf	sat	0.005934
uf20-01000.cnf	sat	0.004503
uf20-0103.cnf	sat	0.004973
prime1849.cnf	sat	8.217051
uf20-0102.cnf	sat	0.011163
sqrt10609.cnf	sat	0.052871
uf20-0106.cnf	sat	0.007841
prime1681.cnf	sat	82.390177
uf20-0105.cnf	sat	0.013614
uf20-0104.cnf	sat	0.004148
sqrt11449.cnf	sat	0.198841
prime841.cnf	sat	5.857044
prime1369.cnf	sat	18.229610
block0.cnf	sat	0.003271
cnfgen-php-10-10.cnf	Timeout	>180sec

UNSAT Files	Result	Execution Time
ph6.cnf	unsat	19.201724
unit7.cnf	unsat	0.001753
uuf100-0182.cnf	unsat	129.408305
uuf100-0151.cnf	unsat	47.372786
false.cnf	unsat	0.001669
uuf100-0147.cnf	unsat	69.796790
uuf100-010.cnf	Timeout	>180sec
uuf100-0120.cnf	unsat	94.298576
full5.cnf	unsat	0.009696
full7.cnf	unsat	0.096316
cnfgen-parity-9.cnf	unsat	3.281838
cnfgen-tseitin-10-4.cnf	unsat	7.329235
elimclash.cnf	unsat	0.001796
uuf100-012.cnf	unsat	71.686958
full3.cnf	unsat	0.003378
cnfgen-peb-pyramid-20.cnf	unsat	0.007735
full1.cnf	unsat	0.000470
uuf100-0130.cnf	unsat	97.958187
add4.cnf	unsat	0.031905
add8.cnf	unsat	4.707284
uuf100-0117.cnf	unsat	178.203761
uuf100-0161.cnf	unsat	141.935478
uuf100-0175.cnf	Timeout	>180sec
cnfgen-ram-4-3-10.cnf	Timeout	>180sec
cnfgen-php-5-4.cnf	unsat	0.020655

Our revised Python solver with the VSIDS implementation. Ran on revised tests with 30 second timeout: (rerun with 3 min timeout later)

SAT Files	Result	Execution Time
sqrt1042441.cnf	sat	0.044484
prime121.cnf	Timeout	>180sec
elimredundant.cnf	sat	0.001773
prime169.cnf	sat	0.191511
prime961.cnf	Timeout	>180sec
sqrt10201.cnf	Timeout	>180sec
sat10.cnf	sat	0.002106
sat12.cnf	sat	0.000897
uf20-0100.cnf	sat	0.020758
uf20-0101.cnf	Timeout	>180sec
uf20-01000.cnf	sat	0.003596
uf20-0103.cnf	sat	0.016876
prime1849.cnf	Timeout	>180sec
uf20-0102.cnf	sat	0.018809
sqrt10609.cnf	sat	0.026454
uf20-0106.cnf	sat	0.002916
prime1681.cnf	Timeout	>180sec
uf20-0105.cnf	Timeout	>180sec
uf20-0104.cnf	sat	0.008199
sqrt11449.cnf	sat	0.024122
prime841.cnf	Timeout	>180sec
prime1369.cnf	Timeout	>180sec
block0.cnf	sat	0.002555
cnfgen-php-10-10.cnf	Timeout	>180sec

UNSAT Files	Result	Execution Time
ph6.cnf	Timeout	>180sec
unit7.cnf	unsat	0.002103
uuf100-0182.cnf	Timeout	>180sec
uuf100-0151.cnf	Timeout	>180sec
false.cnf	unsat	0.002043
uuf100-0147.cnf	Timeout	>180sec
uuf100-010.cnf	Timeout	>180sec
uuf100-0120.cnf	Timeout	>180sec
full5.cnf	unsat	0.007616
full7.cnf	unsat	0.108104
cnfgen-parity-9.cnf	Timeout	>180sec
cnfgen-tseitin-10-4.cnf	Timeout	>180sec
elimclash.cnf	unsat	0.002386
uuf100-012.cnf	Timeout	>180sec
full3.cnf	unsat	0.001559
cnfgen-peb-pyramid-20.cnf	unsat	0.005528
full1.cnf	unsat	0.000475
uuf100-0130.cnf	Timeout	>180sec
add4.cnf	Timeout	>180sec
add8.cnf	Timeout	>180sec
uuf100-0117.cnf	Timeout	>180sec
uuf100-0161.cnf	Timeout	>180sec
uuf100-0175.cnf	Timeout	>180sec
cnfgen-ram-4-3-10.cnf	Timeout	>180sec
cnfgen-php-5-4.cnf	Timeout	>180sec

5. Testing Evaluation & Conclusion

This data shows an increase in timeouts after implementing the VSIDS heuristic. Our implementation maintains the soundness of the original implementation, but it does have noticeably more timeouts. This could be due to increased overhead from increased heap operations. In the original, all the runtime at the decision step was a random step. Now the VSIDS heuristic, there's more overhead due to various heapify operations when clauses are added and when an old heap is copied to a new heap.

Although the VSIDS heuristic provides a more sophisticated method of variable selection, it introduces more overhead which increases delay. It does preserve the correctness and soundness of the implementation we added to, but it increases the number of tests that result in a timeout.