CMPE 58S: Sp. Tp. Computer Aided Verification
Homework III: DPLL

Mehmet Utkan Gezer
2018700060

2018 October 22

This homework is about making a SAT solver of our own. We are expected to implement a DPLL SAT solver that is written in C/C++ or Java. To be fast, it should incorporate heuristics such as backtracking, unit propagation and pure literal elimination. We are, however, forbidden to implement other heuristics, such as CDCL or conflict-directed backjumping.

1 Implementing the DPLL

With the speed stressed in the description, I have decided to implement my sat solver in C. even when I disregard the speed requirement, I would prefer C over C++ or Java, although I would much rather use a higher level language if it was allowed. For an assignment to teach the basics, I believe we should be able to deal with this task on a superficial level with a higher level language like Python or Julia, and concern ourselves less with optimizing the structures we use for the heuristics we'll be employing.

My implementation of the SAT solver is, again, in C, and is bundled with this document under the name `satsolver.c`. We have compiled it both on Windows (MinGW/gcc) and Ubuntu (gcc), with the `-Wall` flag enabled, and it should compile without warnings.

We have tested our implementation against memory leaks using `valgrind`, and it should have no memory leaks.

2 Satisfying solutions to 5 problems

Our implementation fails to provide a satisfying solution to the 4th problem under a reasonable amount of time, so we omit the output for it. While it was, at some point, able to give a satisfying assignment to the 5th problem, it cannot do so now, unfortunately. It is possible that the desktop computer we were trying with before was more capable, or some other reason during the course of development to improve the algorithm in speed and space efficiency.

For problems 1, 2 and 3, the solutions can be found in files `problem1.sol`, `problem2.sol` and `problem3.sol`, respectively.

3 Determining satisfiability of 7 problems

Following are the satisfiability status of the corresponding problems, determined by our implementation.

`problem6.cnf`	Satisfiable
`problem7.cnf`	Not Satisfiable
`problem8.cnf`	Satisfiable
`problem9.cnf`	Satisfiable
`problem10.cnf`	Not Satisfiable
`problem11.cnf`	Satisfiable
`problem12.cnf`	Satisfiable

4 Discussion about problem size and time it takes

As per request, here is the table of times it took for our SAT solver implementation to solve the previous 7 problems. We mention the problem characteristics in the table for further discussion. Problem #10 was solved on a desktop computer with our algorithm before, but we could not get it solved in under 15 minutes on a 3-years-old tablet-like laptop, with which we are timing now.

Problem#	Sat?	#Vars	#Clauses	#C/#V	Time (seconds)
6	Yes	50	100	2	immediate
7	No	50	100	2	immediate
8	Yes	50	300	6	immediate
9	Yes	100	200	2	about 55
10	No	100	200	2	over 15 minutes
11	Yes	100	600	6	immediate
12	Yes	200	1200	6	about 25

It is hard to find a correlation in this data, as there are too many data points with the immediate output.

It looks like the problem is solved rather quickly for;

- Problems with a fewer variables.
- Problems with a higher clause-to-variable ratio.

It makes sense that fewer variables makes the problem easier, as the search space is exponential with the power of number of variables. It also makes sense that increase in clause-to-variable ratio affects runtimes positively, since more clauses increase the chance for us to encounter more unit-clauses and empty-clauses which are the boosting heuristics of our implementation of DPLL.

Problems with many variables and too few clauses seem to take the longest. Specifically when the problem is unsatisfiable, the attempting to exhaust the search space becomes much harder when there aren't enough clauses to introduce unit-clauses and empty-clauses.

5 Bonus: Comparison with Minisat

Compared to the Minisat, our implementation suffers from heavy memory usage and exessive computation time, specifically with the 4th problem.

I had been discussing this homework with a friend of mine studying SAT problems abroad in CMU, and just today he got curious about this specific problem that my implementation is struggling with. He said it could be a "Random SAT", and then tested this using his WalkSAT algorithm written in Python.

Since his Python implementation of WalkSAT was able to solve the question in under 4 seconds, he was certain that it is a Random SAT problem, and then informed me that it is very much expected that a DPLL implementation will suffer against it.

While I am sure I could have improved my implementation, specifically at its memory consumption, this explanation of his did bring me some relief.