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 `satsolver3.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.

1.1 Compiling

Following is the exact command we have been using to compile it on Windows:

```
gcc -O3 -Wall .\satsolver3.c -o s3.exe
```

The Ubuntu version is similar, but without the `.exe` file extension. You may use any other output file-name. The suffix 3 indicates that this is our third major release.

1.2 Running

Following is an example command we have written to our command line on Ubuntu to use our program:

```
./s3 SATproblem/problem11.cnf SATproblem/problem11.sol
```

`SATproblem` is the directory where we were keeping the problems sent to us. First of the arguments given to the program is the input file in DIMACS CNF format.

Second argument is optional, and when given, will be the output file of assignments. Each row in the output will start with a variable index and will have either a 0 for False assignment to that variable, or 1 for True, after a space character. When omitted, the output is printed to the standard output stream.

The output file is only used for variable assignments, and not for any other output. There will be no variable assignments printed anywhere, if the CNF formula is decided to be unsatisfiable. Satisfiability of the problem will always be printed in the standard output stream. Finally, the runtime of the program in terms of wall-clock (not the CPU clock) will be printed to the standard output stream.

For smaller problems, our program spends most of its time on printing the result onto screen, whenever the second argument is not given. Evidently, file buffer flushed onto file takes much less time than 50-200 lines flushed onto screen line-buffered.

While we could change our code to make standard output also full-buffered programmatically, we considered this issue to be rather minor, and instead used a dummy file to dump the results onto, for the times we were just measuring the time per problem. Following is the command we have used for all of the timing results that you will see in the rest of the report:

```
./s3.exe SATproblem/problem9.cnf dump.sol
```

2 Satisfying solutions to the first 5 problems

Our latest implementation successfully provides satisfying assignments to each one of the problems ranging from 1 to 5 in 0.030 seconds (30 milliseconds) on average, and at most in 0.078 seconds (78 milliseconds, for question #4) on a 9-years-old desktop computer with Intel® i7-960 CPU[1] in it.

The generated solution files using the second argument passed to our program are bundled with this report. The problem with the number `n` will have its solution in file `problem<n>.sol`, without the angle-brackets.

3 Determining satisfiability of the last 7 problems

Determined by our program, 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

[1] `https://ark.intel.com/products/37151/Intel-Core-i7-960-Processor-8M-Cache-3-20-GHz-4-80-GT-s-Intel-QPI-`

4 Discussion about problem size and time it takes

As per request, here is the table of times it took for our final SAT solver implementation to solve the last 7 out of the given 12 problems. We mention the problem characteristics in the table for further discussion.

Problem#	Sat?	#Vars	#Clauses	#C/#V	Time (seconds)
6	Yes	50	100	2	0.001
7	No	50	100	2	0.022
8	Yes	50	300	6	0.001
9	Yes	100	200	2	0.003
10	No	100	200	2	12.90
11	Yes	100	600	6	0.005
12	Yes	200	1200	6	0.090

All the computations were carried out on the same machine as described in the previous sections. The timing is done using the `clock_gettime` function provided by the standard library, and we have tested whether it is actually regarding the wall-clock time or not, by using a real-life chronometer on problem 10 at the same time. Given that both the chronometer reading and the program were telling very similar times, we believe the measurements done by the library function represents the wall-clock time passed accurately enough.

As an additional note, the timer in the code encloses everything, including the I/O on reading the file and outputing the assignments, which are time-consuming operations in terms of the wall-clock time.

4.1 Analysis

We first have to note here that our implementation incorporates a breadth-first manner of searching for a satisfying assignment to the formula. This makes our program robust against the positive/negative branch taken with the variable choice after each turn of reductions, as both of the branches are taken at essentially the same time.

As we can see from the timings of problems 6 and 8, and 9 and 11, execution time for a problem is practically only dependent on the number of variables, and is not changing as the number of clauses increases/decreases, with our implementation. The very slight increase can partly be attributed to the longer reading time of the file, and increased amount of memory allocations for the data structures.

For satisfiable problems, execution time is around 0.002, 0.012 and 0.207 seconds for variable counts 50, 100 and 200, respectively. We cannot tell much with just this data, but we can say that the runtimes are super-linear in terms of number of variables in the formula.

The execution times with the unsatisfiable problems are noticably higher than the execution times of the satisfiable problems with similar variable counts, and even more than that. Problem 7, for example, requires much more time than problem 6 and 8, but also 9 and 11. This is due to the fact that our algorithm has to exhaust the whole search tree to tell that the formula is not satisfiable.

Results are as expected, with the most interesting outcome being the execution time being almost entirely indifferent about the number of clauses in the problem set provided to us.

5 Bonus: Comparison with Minisat

Compared to the Minisat, our final implementation is not performing that bad, except for perhaps the problem #10. We have tried very hard to make our program as memory efficient as possible. It now uses

around 256 KB memory with problems other than #10, and about 32 MB memory to decide that #10 is unsatisfiable.

Clause-learning, we believe, would be of great help to our program in its journey towards exhausting the search tree faster to conclude that a problem is unsatisfiable. Despite the fact that problem #10 takes a considerable amount of time, we are very much content with our results.

We can only wish that the students taking this course in the following years are not made to bother much about low-level optimization opportunities and technicalities of SAT solvers, and are allowed to hand in their solvers in higher level languages. We believe that students will be able to learn more and implement more of the available techniques if they were to stay away from the cumbersome nature of C and Java.

5.1 Discussion about problem 4

I had been discussing this homework with a friend of mine studying SAT problems abroad in CMU, and at some point he got curious about this specific problem 4 that my older implementation was struggling to solve. 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.

My latest SAT solver does not really suffer against problem 4 anymore, but I believe this to be an important thing to note regardless.