Institut für Informatik
der Universität München
Dr. J. Johannsen
Florian Leimgruber

WS 2020/21

10. December 2020

Exercises
SAT Solving

Problem set 1

Project 1: Your own DPLL-Solver

Your task is to implement a SAT solver based on the DPLL algorithm.

Similar to the first project, the programming languages C/C++, Java, Python, Rust and Haskell can be used. Other languages might be permitted on a case by case basis; simply ask us. To attain competitive performance, we strongly recommend using a lower-level language like C++.

Solutions may be handed in groups of two to four students. Everyone in the group is expected to be able to explain all the code.

This project consists of four parts. The first part is necessary to pass, while the other parts will get you a better grade. Keep in mind that quality is more important than quantity, though.

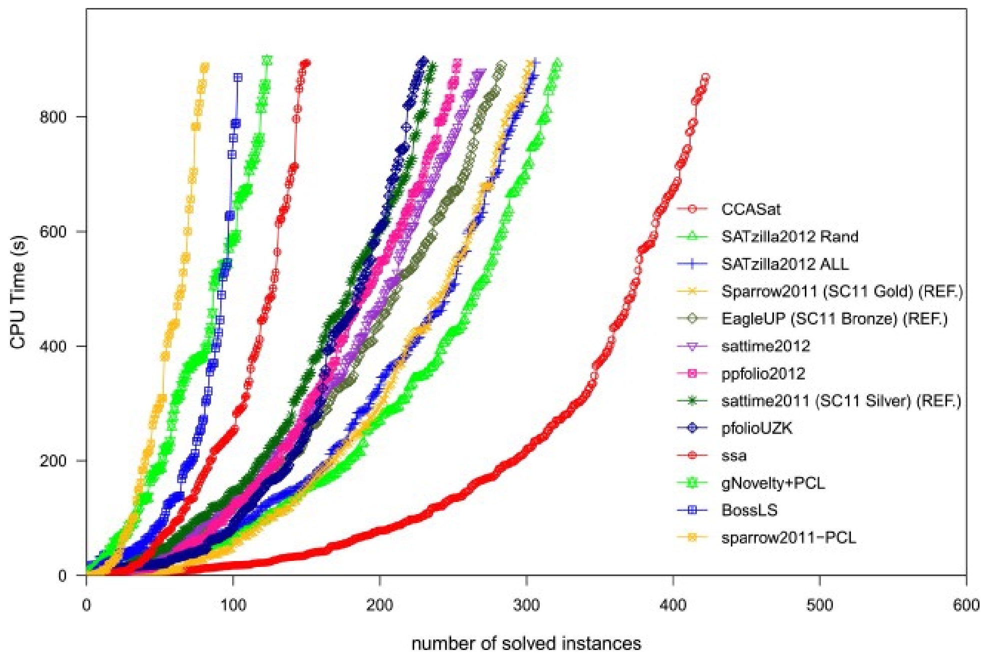
1. Compulsory part: Implement a program that can read a SAT instance as DIMACS CNF format, solve it using the DPLL algorithm (with Unit Propagation), and output the solution in DIMACS format (generate s- and v-lines).

   We include a set of basic tests (in test/) that this solver needs to pass correctly.

   Benchmark your solver using the included SATlib benchmarks. Given one minute of CPU time for each input, how many problems is your solver able to solve? There are some hard problems included, you are not supposed to solve all within that time limit.

2. Bonus part: Implement Pure Literal Elimination.

3. Bonus part: Generate a cactus plot of your solver for the given SATlib benchmarks. First, measure the run time of the solver for each problem. Sort these numbers. Then plot the number of solved instances on the x-axis, and the running total of the CPU time on the y-axis. Example from the SAT 2012 competition:



4. Bonus part: Implement various selection heuristics, such as DLIS, DLCS, MOM, Jeroslow-Wang, and compare them, e.g. by overlaying cactus plots. Feel free to invent your own selection heuristics and experiment.

Solutions can be submitted via uni2work until Thursday, 21. Januar 2021, 14:00. Please add a short readme file, explaining how to build/run your program.

Exercise 1: CNFgen is a useful tool to generate arbitrary sized DIMAC CNF formulas of various types. These can be helpful when testing your DPLL solver:

1. Install CNFgen using `pip3 install cnfgen`.
2. Use `cnfgen randkcnf k n m` to generate a random k-SAT instance over n variables and m clauses.

Aside from random formulas, CNFgen supports many different formula types, mostly from research in Proof Complexity. Examples include the Pigenhole principle (`cnfgen php n`) and the Ordering Principle (`cnfgen op n`), as documented on the website.

Exercise 2: Kullmann's method relies on a distance function d, assigning a weight $d_i = d(v, w_i)$ to each edge of the recursion tree. In the lecture, $d_i > 0$ was stated as a requirement.

Why is this condition necessary and when can it be relaxed?

Exercise 3: Monien-Speckenmeyer's algorithm is based on autark assignments, which are a generalisation of pure literals. By using pure literals instead, one would expect to get an algorithm with a runtime between simple–MS and MS. Consider the following algorithm:

```
Fl(F, $\alpha$)
  if F = 0 then return UNSAT
  if F = 1 then return $\alpha$

  if F contains pure literal p
    return Fl($F[p \leftarrow 1], \alpha \cup [p \leftarrow 1]$)

  a := arbitrary literal of a shortest clause
  $\beta$ := Fl($F[a \leftarrow 1], \alpha \cup [a \leftarrow 1]$)
  if $\beta \neq$ UNSAT then return $\beta$
  return Fl($F[a \leftarrow 0], \alpha \cup [a \leftarrow 0]$)
```

Analyse its runtime on 3-SAT using Kullmann's method.

Hint: Define different distances depending on the size of the clause a is from.