Mini-project: DPLL satisfiability solver

Flavio Pfaffhauser
#0837079

1. Implementation

I carried out the implementation of the DPLL satisfiability solver in Java using Eclipse as the integrated development environment and composed the system into three classes:

Class name	Description
`Solver.java`	Contains the main solving method to solve a formula in conjunctive normal form. A formula is a `LinkedList<Clause>`.
`Clause.java`	A clause represents disjunctions of literals. A clause is represented as a `Hashmap<String,Literal>` which allows fast access to a literal with a given name.
`Literal.java`	A literal represents an atom that is either negated or not. An atom is basically a string of length one. If the atom is negated a prefix "not" is added to the string.

The implementation is based on the algorithm discussed in class "05-logic2.ppt" on slide 45: DPLL (for real!). It considers unit literals and pure literals as a speedup. For all the methods in the classes there exist header comments and all the execution steps are documented really well, that's why I won't write more about the execution of the algorithm here.

Another point to mention is the facility to create random formulas. Since I wanted to test my satisfiability solver with large inputs I created a random formula generator that creates formuolas given two parameters. One of the parameters is the total number of literals that should be in the formula and the other is the maximum size of a clause. The algorithm `generateRandomFormula(n,m)` will then create a random formula with n literals which are grouped into clauses with the maximum size of m (randomly).

2. Instructions

To run the DPLL algorithm import the project into Eclipse and run Solver.java which contains the main() method. You can adjust several parameters in the main() procedure:

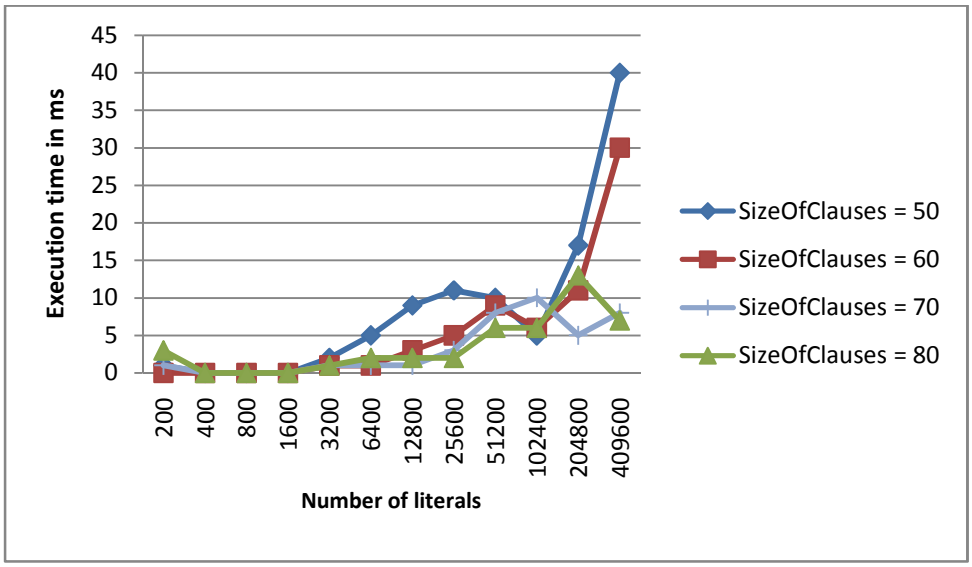
Parameter name	Description
`numberOfExecutions`	Number of times the solver is invoked
`numberOfLiterals`	Total number of literals that will be in the randomly created formula
`maxSizeOfClauses`	The maximum size of a clause. The size of clauses is randomized but must be at least one.

Those are all the inputs needed to run the algorithm. The output will be the execution time and the answer whether the formula is satisfiable or not.

3. Evaluation

Experiment 1: Vary the number of literals in the formula

In this experiment I varied the number of literals in the formula to solve for satisfiability. On the x axis you find the number of variables and on the y axis the execution time in milliseconds. One can observe a complexity peak similar to the one presented in the lecture (*#clauses/#variables* = 4.3). Once the complexity peak is reached the execution time decreases again for a short time but then increases significantly since the algorithm has exponential runtime in general.



Experiment 2: 3-SAT problem

In this experiment I solved the 3-SAT problem with a varying number of literals. If I would try to solve 3-SAT formulas with more than ~170'000 clauses the program would crash with an out of memory exception, but otherwise the algorithm scales pretty linearly. On the x-axis I plotted the number of literals and on the y-axis the execution time.