Exercises on SAT Solving
Problem set 2

**Exercise 3:** Let \( C \) be a clause, and \( a \) a literal of \( C \). Then we say \( C \) is **blocked with respect to** \( a \) for \( F \), if: for every clause \( D \) of \( F \) which contains \( \overline{a} \), there is a literal \( b \) of \( D \), such that \( \overline{b} \) is in \( C \).

Prove that addition and removal of blocked clauses preserves satisfiability. How can this help speeding up a DLL algorithm?

**Exercise 4:** Write pseudo-code for a DLL-style algorithm which solves 2-SAT. Compare your algorithm to the graph-based algorithm from the slides!

**Practical Exercise 3:** Implement a simple DLL solver.

In the lecture, you learned the core DLL algorithm. Write a simple SAT solver which reads a 3-SAT DIMACS formula from standard input (or a file, if you prefer), and solves it using backtracking.

Use the first literal of the shortest clause to decide which variable to try next.

Note: you do not need to come up with fancy data structures, using arrays or lists is perfectly fine.

**Practical Exercise 4:** Benchmark your DLL solver.

We install the tool CNFgen to generate random CNF formulas:

```bash
% pip install -t cnfgen cnfgen
% PYTHONPATH=cnfgen python -m cnfformula.cnfgen -q randkcnf 3 4 5
```

Measure average runtime for your DLL solver on random instances of 3-SAT for \( v \) variables and \( 4 \cdot v \) clauses on a suitable range. Plot the results.

Hand in on UniWorX until Tuesday, November 10, 2015, 4pm.