Assignment 4: Satisfiability
CS 730/830, Fall 2015

Electronic submission due at **11:30pm on Wed, Oct 7**
Hardcopy submission due at **start of class on Thu, Oct 8**

Overview

You will write a program to solve the most fundamental NP-Complete problem of all: determining the satisfiability of a propositional (or 'Boolean') formula presented in conjunctive normal form (CNF). You should use the backtracking-based Davis-Logemann-Loveland algorithm. You should implement unit propagation but you can skip checking for pure literals. Graduate students will also implement an incomplete model finder — this should use the WalkSAT/SKC stochastic local search algorithm. For WalkSAT's *maxFlips* parameter, just hardcode a value that causes about 10 seconds between tries for a largish problem, and for the noise probability p, just use 0.5.

Input

Your program will read a formula from standard input—no command-line arguments. Graduate students should write two separate programs.

We will use a subset of the standard DIMACS CNF format:

```
c This is a tiny example.
p cnf 4 3
1 -2 3 0
2 3 -4 0 1 4 0
```

Our format is:

- Comments lines start with `c`.
- Before any clauses comes a line starting with `p cnf`. The rest of the line will be two numbers: the number of variables and the number of clauses.
- Clauses are a list of variables, terminated by `0`. Negative literals have the variable number negated. Note that newlines have no meaning — the `0` is the clause terminator.

Output

We will use the standard DIMACS output format:

```
s cnf 1 4 3
v 1
v -2
v 3
v -4
```

The format is:

s After `s cnf` comes the solution: `1` for satisfiable, `0` for unsatisfiable, and `-1` if the solver couldn't tell (this would be useful if you implemented a timeout for WalkSAT, for example). After this comes the number of variable and number of clauses (just as in the input `p` line).

v introduces a variable assignment, where negative means the variable should be set to false.

If the formula is not satisfiable, the `v` lines are omitted.

For DLL, your output should end with the number of branching nodes explored during the search:

```
345 branching nodes explored.
```

For WalkSAT, you should list the number of tries and the total number of flips:

```
14 tries.
2345 total flips.
```

Execution

Please use `make.sh` and `run.sh` scripts as usual.
We supply:

`*.cnf` a few example benchmark problems.

`make-3cnf` takes two command line arguments: the number of variables and the number of clauses. Outputs a random 3-CNF formula. The critical constrainedness should be around 4.3 clauses per variable.

`dll-reference` a sample DLL solution. Also available as a jar file.

`walksat-reference` a sample WalkSAT solution. Also available as a jar file.

`sat-validator` runs your program and validates its output. For example:

```
sat-validator ./run.sh < tiny-1.cnf
```

The validator expects a formula on standard input and will pass it to your program. A one minute timeout is hardcoded into the validator.

Submission

Electronically submit your source code using the instructions on the course web page.
Submit a brief write-up in class answering the following questions:

1. What is the size of the state space for this problem?
2. Describe any implementation choices you made that you felt were important. Mention anything else that we should know when evaluating your program.
3. When there are around 3.3 clauses per variable, how does the largest formula your DLL can solve compare to the largest for your WalkSAT? (Undergrads can run the reference WalkSAT.)
4. Repeat the previous question at 4.3 clauses per variable, and explain the results you obtain.
5. What suggestions do you have for improving this assignment in the future?

Evaluation

Grading is done with an automated script, so be sure you have one algorithm working before starting the next, so that we can give you some credit even if you don't implement both.
The written questions are one point. Grad students will have 5 points for DLL, 4 points for WalkSAT.

Design Suggestions

It's better to be slow and correct than fast and buggy.
Just as in a CSP, if you want to be fast, only some of the clauses need to be checked after a variable is assigned or flipped.