CS227: Assignment 1

The first assignment involves implementing and evaluating **propositional satisfiability algorithms**, and writing up a report on your experiments.

Algorithms: Compare the relative performance of GSAT, HSAT, GSAT with random walk, and WalkSat. In addition, compare HSAT with and without score caching. You are free to augment these algorithms with other algorithms (e.g., DPLL or DLM-based algorithms) or variants of your own (though this is not required by the assignment).

Problems for evaluation: Programs must be evaluated on the SAT problems available for download from the Coursework site. All these problems are known to be satisfiable. We have included random 3-SAT problems with 50, 100, 150, 200, 250, 300, 400, 500, 600, 700 and 800 variables. There are 10 instances generated with a physics-inspired spin-glass model that are hard for many modern SAT solvers. In addition, there are 40 SAT problems generated by converting other problems into SAT (e.g., graph coloring, logistics, blocks world planning, latin square). Give your program a maximum of 10 minutes to solve each problem instance. The file syntax is the DIMACS CNF format included below.

Make sure you meter all relevant parameters (e.g., run time, number of flips). Results should include at least the number of problems solved, median time, and flips for each variable setting.

Report: Your report should contain descriptions of the algorithms you are evaluating, including discussions of any optimizations you may have used to make the algorithms run fast. In particular:

- Describe the details of all your caching algorithms (for scores, unsatisfied clauses, occurrence in unsatisfied clauses, etc.).

The report should contain the results of running the experiments and a discussion of your conclusions.

Submission: The report can be submitted electronically, in class, or directly to the TA. Submit source code electronically as a single .tgz or .zip file that unpacks into its own directory. Please include a small README file describing how to build and run your code. Clearly identify all members of the group both on the report, and in the electronic submission. Send electronic submissions to cs227-submit@lists.stanford.edu.

Assignments will be graded on the description of the algorithms, the descriptions of the optimizations used, the raw results, and your analysis of the results.

Assignments may be done in groups of 2-3 students. You may choose any programming language for implementation purposes, though we recommend either C or C++ for maximum efficiency. Assignments are due by noon on **23 April**.

DIMACS CNF format

A satisfiability problem in conjunctive normal form consists of a conjunction of a number of clauses, where is clause is a disjunction of a number of variables or their negations. If we let x_i represent variables that can assume only the values *true* or *false*, then a sample formula in conjunctive normal form would be

(x_1 or x_3 or (not x_4)) and (x_4) and (x_2 or (not x_3))

Given a set of clauses $C_1, C_2,\ldots, C_m$ on the variables $x_1, x_2,\ldots, x_n$, the satisfiability problem is to determine if the formula

C_1 and C_2 and … and C_m

is satisfiable. That is, is there an assignment of values to the variables so that the above formula evaluates to *true*. Clearly, this requires that each C_j evaluate to *true*.

To represent an instance of such problems, we will create an input file that contains all of the information needed to define a satisfiability problem. This file will be an ASCII file consisting of a two major sections: the preamble and the clauses.

Preamble: The preamble contains information about the instance. This information is contained in lines. Each line begins with a single character (followed by a space) that determines the type of line. These types are as follows:

Comments. Comment lines give human-readable information about the file and are ignored by programs. Comment lines appear at the beginning of the preamble. Each comment line begins with a lower-case character `c`.

```
c This is an example of a comment line.
```

Problem line. There is one problem line per input file. The problem line must appear before any node or arc descriptor lines. For cnf instances, the problem line has the following format.

`p` *FORMAT VARIABLES CLAUSES*

The lower-case character `p` signifies that this is the problem line. The *FORMAT* field allows programs to determine the format that will be expected, and should contain the word ``cnf''. The *VARIABLES* field contains an integer value specifying *n*, the number of variables in the instance. The *CLAUSES* field contains an integer value specifying *m*, the number of clauses in the instance. This line must occur as the last line of the preamble.

Clauses: The clauses appear immediately after the problem line. The variables are assumed to be numbered from 1 up to *n*. It is not necessary that every variable appear in an instance. Each clause will be represented by a sequence of numbers, each separated by a space, a tab, or a newline character. The non-negated version of a variable *i* is represented by *i*; the negated version is represented by – *i*.

Each clause is terminated by the value 0. Unlike many formats that represent the end of a clause by a new-line character, this format allows clauses to be on multiple lines.

Example: Using the example

(x_1 or x_3 or (not x_4)) and (x_4) and (x_2 or (not x_3))

a possible input file would be

```
c Example CNF format file
c
p cnf 4 3
1 3 -4 0
4 0 2
-3
```