SAT solver

A C++ SAT solver implementation, based on the DPLL algorithm.

This solver is an assignment for the Logic in Information Technology course and is meant to provide us with the basics of SAT solving techniques.

Table of Contents

Please see the *Results* and *Source and scripts* sections to have an overview of the structure of the project.

Implemented features

Based on the code that was already provided, the following enhancements have been implemented in order to lower the execution time of the solver:

Occur lists

The BCP (Boolean Constraint Propagation) procedure (called `propagateGivesConflict()` in the code) does not traverse the whole set of clauses anymore. Instead, a couple of additional data structures are used in order to minimize the amount of clauses being visited each time a literal is propagated.

These data structures are called *occurrence lists* and are a couple of lists, indexed by variable. Each item in these lists is, indeed, another list, containing the clauses in which the variable appears as a positive or negative literal. Thus, the lists are named `positiveClauses` and `negativeClauses`, respectively.

The occurrence lists are built during initialization time, as clauses are fed into the solver. In order to reduce the number of memory read operations, instead of just storing the index of the clause, a pointer to the clause is kept.

Activity-based decision heuristic

The heuristic used to choose the next literal to be *decided* by the DPLL algorithm has also been updated with respect of what was implemented in the example code that was provided.

The decision heuristic works as follows:

- An *activity* counter is kept for each literal, be it positive or negative, indicating the number of

conflicts in which the literal was involved.
- Each time a conflict is found, the activity of *all* the literals in the clause that causes the conflict is incremented by a constant factor (1, in our case).
- Because recent conflicts should be given more importance than older ones (in order to rapidly force a backtrack operation), the activity of all the literals is diminished from time to time. More concretely, it is divided by 2 every 1000 conflicts.
- When a *new decision* has to be made, the literal with the highest activity of those variables still undefined in the model is chosen and returned to the DPLL procedure.

Note: the amount of conflicts between each time the activity is decreased was obtained by testing different values on medium-sized problems. Values in the [10, 100000] range were tested and, finally, 1000 was chosen as the most suitable one.

Benchmark framework

The environment in which the solver was benchmarked consists of the following hardware and software components:

Hardware

- **CPU:** Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz (2 cores, 4 threads)
- **CPU cache:** 128 KB L1, 512 KB L2, 3072 KB L3
- **Memory:** dual channel 8GB @ 1333 MHz

Software

- **OS:** Ubuntu 12.04.5 LTS; 3.2.0-77-generic Linux kernel
- **Compiler:** g++ (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
- **Picosat:** Picosat v936

Results

The `results` directory contains the following files:

- `results.txt`: a quick reference with the expected results for each of the problem instances found at the `sample_problems` directory.
- `times-to-beat.txt`: a file indicating which are the maximum execution times that our SAT solver implementation should perform.
- `sat-results.ods`: a spreadsheet containing **the requested results** for the assignment: execution times, propagations and decisions per second of both our SAT solver implementation and the Picosat package for each of the sample problems provided.
- `sat-results.xlsx`: the same spreadsheet, in a different format.
- `sat-results.pdf`: a printed version of the contents of the spreadsheet, in order to facilitate reading it.

Concerning the `output` and `output_pico` directories, they just contain the raw output of the solvers

for each problem instance.

Source and scripts

The source of the implemented solver is located under the `src` directory, in the `sat.cpp` file.

Some scripts are also provided in the project, at the top-most level of the directory hierarchy:

- `Makefile`: a `make` script, to compile the solver.
- `sat.sh`: a wrapper over the main executable file, which is located under the `bin` directory, after being compiled. The script allows to either compile or run the solver with a given input problem file.
- `runner.sh`: a simple script that executes the solver with each problem file under the `sample_problems` directory.
- `picorunner.sh`: a simple script that executes the `picosat` solver with each problem file under the `sample_problems` directory.

Compiling and executing the solver

To compile the solver, execute:

```
./sat.sh compile
```

To run the solver with an example input problem:

```
./sat.sh run sample_problems/vars-100-1.cnf
```