Software Analysis & Verification **Due: Mar 17, 2020**

Programming Assignment 1

Instructor: Fei He YOUR NAME (YOUR TSINGHUA ID)
TA: Jianhui Chen, Fengmin Zhu

Read the instructions below carefully before you start working on the assignment:

- This programming assignment asks you to implement the Davis-Putnam-Logemann-Loveland (DPLL) algorithm in C++.
- A grading program will automatically test your executable against a test suite (some test cases are not public), so please do not change any interfaces that will be invoked by the grading program.
- You MUST zip all your code and report (in PDF) in a `zip` (other formats are NOT allowed, see Section 3 for the required folder structure) and finally hand it to Tsinghua Web Learning *before the due date*.
- To prevent others from copying your code, please do NOT publish it at any place that is accessible to others. Instead, you are allowed to host it on a private repository.
- As a kind remind, a programming assignment worths more than an ordinary homework. You should get your hands dirty as early as possible, as you may spend an unpredictable amount of time on debugging.

1 DPLL Review

A DPLL algorithm takes a propositional formula as input, checks its satisfiability and produces a satisfying model (or "interpretation" mentioned in the slide) if the formula is satisfiable. There may be millions of ways to represent a propositional formula, but DIMACS is a widely-used format that plenty of SAT solvers accept.

DIMACS Format DIMACS is a simple ASCII representation for propositional CNF formulas that is both easy-to-read and easy-to-parse. A valid DIMACS file has the following structure:

- It may begin with several comment lines, which are prefixed by a '`c`'.
- After the comment lines, there is a header line of the format `p cnf nvars nclauses`, where `nvars` is the number of variables in the formula, and `nclauses` is the number of clauses.
- After the header line, there will be one line for each clause. A clause is a sequence of distinct non-zero integers, ended by `0`. A positive integer corresponds to a variable literal (from `1` to `nvars`, both inclusive), and a negative integer corresponds to a negated literal.

For example, the following DIMACS file

```
c A simple example
p cnf 3 3
-1 2 0
-2 3 0
-1 -3 0
```

represents the formula $(\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_3) \wedge (\neg P_1 \vee \neg P_3)$.

The homework artifact already comes with a DIMACS parser (`dpll/DimacsParser.h`) so you don't need to build your own. At the meantime, we provide a few test cases in `dpll/tests/` so you should, at least, pass all of them. However, we have more test cases for grading.

To obtain the solutions of these test cases, you may invoke `MiniSat` (see Appendix B for installation).

Interface The DPLL algorithm has a quite simple interface: it takes a formula as input, and returns either "satisfiable" or "unsatisfiable". If the formula is satisfiable, the algorithm also produces a satisfying model. In terms of C++ (`dpll/DPLL.h`):

```
class DPLL {
public:
  DPLL(const formula &phi);
  bool check_sat();
  model get_model();
}
```

The constructor simply accepts a formula. In `dpll/common.h`, you will see that the type `formula` is a structure that contains (1) the total number of variables, and (2) a vector of `clauses`. A `clause` is simply a vector of `literals`. A `literal` is simply an integer – negative means negation, like in DIMACS.

```
typedef int literal;
typedef std::vector<literal> clause;

struct formula {
  int num_variable;
  std::vector<clause> clauses;
};
```

Function `check_sat` should return the satisfiability as a boolean value – true for "satisfiable", and false for "unsatisfiable".

Function `get_model` should return an arbitrary (since there could be many) satisfying model. In `dpll/common.h`, you will see that a `model` is actually a map from integers to booleans, say $2 \mapsto$ true stands for: variable 2 is assigned to value true.

```
typedef std::unordered_map<int, bool> model;
```

You model must be *complete*, that is, it assigns a truth value for *every* variable, rather than a subset of them! For instance, the formula parsed from the following DIMACS

```
c A simple example
p cnf 4 3
-1 2 0
-2 3 0
```

is satisfiable, and a complete model could be $\{1 \mapsto \text{false}, 2 \mapsto \text{true}, 3 \mapsto \text{true}, 4 \mapsto \text{true}\}$ (note: variable 4 is not presented in any clause, but you still need to assign a truth value for it). This function will be invoked if and only if `check_sat` returns true.

The above three functions will be *directly* invoked by our grading program, so *NEVER change* them!

Algorithm In general, the DPLL algorithm really has nothing advanced – its underlying strategy is just a depth-first search with backtracking, which is taught in every undergraduate algorithm course. What really makes it intelligent is the clever usage of unit propagation and backjump.

Search order (or decision order) is actually a very important topic in designing a "good" search algorithm. A "bad" order sometimes leads to appearance of worst case scenarios and thus deteriorates efficiency. Recall that the *decide* operation introduced in the lecture actually does a random choice – it does not have any preference on the variables. So we see *decide* is not a good enough operation and should be triggered as late as possible. Is there any way to optimize this? Yes, when the current partial interpretation entails that some (unassigned) variable must be true (or false), it gives us a chance to simply *decide* that variable, as its truth value is fixed in the current situation. And that's what unit propagation exactly does.

Another important topic in search algorithms is pruning the search space. In DPLL, when we reach a point where the current partial interpretation is unsatisfiable, it is possible for us to find out the reason and avoid it in future. Conflict clauses tell us exactly how the contradicted situation happens. Using them, we could figure out a formula that must be true (or else it produces conflicts), which reduces the search space where the formula does not hold. And that's why backjump can accelerate the search.

2 Grading

Your main task is to implement a DPLL algorithm, with the help of the template code located at `dpll/`. The project is built by CMake (see Appendix A for a quick tutorial). By default we have enabled `-std=c++17`. Hope that works for you. To make your code compile on the grader's machine, please avoid using any system-dependant or compiler-dependant features.

The score is divided into two parts.

Part 1 (80%) In this part, you do NOT need to implement backjump. In other words, you only need to implement decide, backtrack and unit propagation. If your implementation is correct, it should *at least* pass all the test cases given to you. Remember the correctness means two things: (1) the satisfiable/unsatisfiable judgement must be correct, and (2) the satisfying model must be correct (though you are allowed to return any of them, if the formula is satisfiable). To receive full credits, your program must also pass a couple of hidden test cases and each within a reasonable time – the time limit for each test case is 5 seconds. Only the execution time of DPLL will be measured (parsing is NOT included). Also, we will compile your code with `-O2` optimization (release mode) when grading.

Part 2 (20%) To receive full credit of this part, you must add backjump and conflict clause generation. We will not use any test case to grade. Instead, please write a report to briefly introduce how you implemented, and also provide at least 3 test cases to demonstrate that backjump really accelerates the execution. In detail, you should compare the execution time of your algorithm with and without backjump on these test cases. Please include the results (time) in your report, and the test cases in `dpll/tests`. Moreover, you are encouraged to use your algorithm for solving the N-queues problem (try $N = 2, 3, 4, \ldots$) and present the results. No matter you write the report by LaTeX, or MS Word, or Markdown, or whatever, please export it as a PDF!

3 Submission

Upload a `.zip` file (do NOT use other formats such as `.rar`, `.tar`) of all your code and report, following the folder structure as below (we should see this after unzip):

```
pa1/
    dpll/             # your code
        CMakeLists.txt
        DPLL.cpp
        DPLL.h
        DimacsParser.h
        common.h
        main.cpp
        tests/  # your tests
            ...
    report.pdf  # your report
```

A CMake from CLI

To make our life easier, we pick CMake, which is cross-platform, as the official build tool of this assignment. This tool generates `Makefile` that is suitable on your platform. Many IDEs such as CLion has good support for CMake projects. However, if you prefer to build from command line: in folder `dpll/`, type

```
mkdir Debug/
cd Debug/
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

To build a release (`-O2` enabled), type

```
mkdir Release/
cd Release/
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

B MiniSat Installation

Online version: `https://www.msoos.org/2013/09/minisat-in-your-browser/`.

Linux Many package managers have prebuilt. For example on Ubuntu:

```
sudo apt-get install minisat
```

To build from source, clone and follow "Quick Install" of this repository.

Mac OS Using Homebrew is the easiest way:

```
brew install minisat
```

Windows Follow the long installation guide on this page. You will need to install `cygwin` first.

Alternatively It seems `cryptominisat` is easier to be installed for non-Linux users. On Windows, try the exe download link on this page. On Mac OS,

```
brew install cryptominisat
```

also works.

Usage Simply

```
minisat DIMACS_FILE
```

or

```
cryptominisat DIMACS_FILE
```