ISTD 2D Design Challenge
50.001
SAT Solver for Combinational Equivalence Checking

To get started, pull out the starting code from 50.001 course webpage.

SAT Solver, CNF, DPLL

A *propositional formula* is a logical formula formed from boolean variables and the boolean operators and, or and not. The satisfiability problem is to find an assignment of truth values to the variables that makes the formula true.

A *SAT solver* is a program that solves the satisfiability problem: given a formula, it either returns an assignment that makes it true, or says that no such assignment exists. SAT solvers typically use a restricted form of propositional formula called CNF.

A formula in *conjunctive normal form (CNF)* consists of a set of clauses, each of which is a set of literals. A literal is a variable or its negation. Each clause is interpreted as the disjunction of its literals, and the formula as a whole is interpreted as the conjunction of the clauses. So an empty clause represents false, and a problem containing an empty clause is unsatisfiable. But an empty problem (containing no clauses) represents true, and is trivially satisfiable.

Davis-Putnam-Logemann-Loveland (DPLL) is a simple and effective algorithm for a SAT solver. The basic idea is just *backtracking search*: pick a variable, try setting it to true, obtaining a new problem, and recursively try to solve that problem; if you fail, try setting the variable to false and recursively solving from there. DPLL adds a powerful but simple optimization called *unit propagation*: if a clause contains just one literal, then you can set the literal's variable to the value that will make that literal true. (There's actually another optimization included in the original algorithm for 'pure literals', but it's not necessary and doesn't seem to improve performance in practice.)

Note: You are **NOT** allowed to use an existing open source SAT solver as a part of your implementation in this project.

Task: SAT Solver

You need to implement a SAT solver in this project. A SAT solver takes a CNF propositional formula and finds an assignment to its variables that makes the formula true. In this problem, you will implement a SAT solver. You should develop and test your SAT solver independently of the hardware verification problem; i.e., don't feed it formulas produced from hardware verification, but choose formula test cases appropriately.

[70 points] Implement SATSolver.solve(). Here is the pseudocode.

- If there are no clauses, the formula is trivially satisfiable.
- If there is an empty clause, the clause list is unsatisfiable -- fail and backtrack. (use empty clause to denote a clause evaluated to FALSE based on the variable binding in the environment)
- Otherwise, find the smallest clause (by number of literals).
    - If the clause has only one literal, bind its variable in the environment so that the clause is satisfied, substitute for the variable in all the other clauses (using the suggested substitute() method), and recursively call solve().
    - Otherwise, pick an arbitrary literal from this small clause:
        - First try setting the literal to TRUE, substitute for it in all the clauses, then solve() recursively.
        - If that fails, then try setting the literal to FALSE, substitute, and solve() recursively.

Putting it together: CNF File Parser, Call SAT Solver

[30 points] Implement `public static void main(String [])` in `SATSolverTest`. In main(), read in and parse a .cnf file (see cnf file handout for file format), and construct the corresponding Formula instance as input to your SAT solver. Print out "satisfiable" or "not satisfiable" depending on the result. If the Formula is satisfiable, output your variable assignments to a result file "BoolAssignment.txt" with format of one variable per line: <variable>:<assignment>, e.g.,

1:TRUE

2:FALSE

Report the execution time of your SAT solver by including the following code surrounding your SATSolver.solve():

```
Formula f2 = ...

System.out.println("SAT solver starts!!!");

long started = System.nanoTime();

Environment e = SATSolver.solve(f2);

long time = System.nanoTime();

long timeTaken= time - started;

 System.out.println("Time:" + timeTaken/1000000.0 + "ms");
```

We will test your program SATSolverTest by providing you a .cnf file with about 6000 clauses and variables. You may need to increase the amount of memory that the Java interpreter has allocated for heap space / stack space. Every time you run your project (as a Java program, with JUnit, etc.), Android Studio / Eclipse creates a *run configuration* that specifies what to run and how to run it. To increase the maximum heap / stack space for a run configuration in Android Studio, open the Run dialog (**Run → Edit Configurations...**), select the relevant configuration, and enter under VM options `-Xmx2048m -Xss128m` (for example, which sets the maximum heap size to 2048MB, maximum stack size to 128MB).

Your SAT solver needs to solve our 6000 clauses .cnf file in less than 15 seconds on a workstation machine that we provide.

Before You're Done...

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods, and none of the methods we gave you to write should produce any side effects.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Hints

You need to understand how to use the ImList<T> data type (immutable list), which is the underlying data type for Clause and Formula. Instantiation of ImList<T> can be done by:

ImList<T> l = new EmptyImList<T>();

Other list operations are similar to Java API lists, except ImList<T> is immutable, and as a result, a new reference is returned for any modification of the data structure, e.g.,

l = l.add(r);

You also need to familiar with the use of Clause, Formula, Environment, Literal.

This part applies to 2D Design Competition only

We will map your execution time into a score between 0 and 100 using the following formula and this would be your score for the SAT solver part:

$$f(x) = \max\left(0, 100 \times \left(1 - \frac{x - \min(X)}{a - \min(X)}\right)\right)$$

where $a = \min(15, \max(X))$, and X is the set of the all the execution time. For example, if the execution time for 3 groups is X = {14, 7, 26}, their scores are 12.5, 100 and 0, resp.