Report

Implementation

We have implemented an SAT Solver in C++ using the Davis-Putnam-Logemann-Loveland (DPLL) algorithm.

The DPLL algorithm is essentially a backtracking algorithm, and we have implemented a recursive function for the same. The algorithm is based on partial assignments and progressing depending on whether the partial assignments prove successful or not. For deciding which variable to assign first, we used two concepts – **pure literals** and **unit literals.**

Unit Literals – Consider the following formula:

$$(\sim X_1 \vee X_2) \wedge (X_3) \wedge (X_4 \vee \sim X_5) \wedge (X_3 \vee X_1)$$

In the formula shown above, the second clause contains only the literal X_3. As the formula is in CNF form, every clause must be true for the formula to be SAT. So, for the 2^{nd} clause to be true, X_3 must be set true. This reduces one clause which we no longer need to check and if we have X_3 in some other clause, then we can set it to true there as well.

Pure Literals – Again, considering an example:

$$(X_1 \vee X_2) \wedge (\sim X_3) \wedge (\sim X_4 \vee X_5) \wedge (X_3 \vee X_1)$$

The aim of the SAT solver is to find only one satisfying assignment, even though it is possible that multiple of them exist. Thus, if a certain variable exists only in its positive or negated form, we can assign it in the initial steps of the algorithm. Note that since it only exists in form, this won't affect the outcome of the solution. In the example above, the variable X_1 exists only in its positive form, so assigning it true reduces branching and helps in reducing the number of clauses required.

We have further used 3 concepts for reducing the number of literals and clauses to be considered in the next step of the DPLL algorithm.

1. We can delete clauses with a true literal

If L_1 is true, then $L_1 \vee L_2 \vee L_3 \ldots$ is true. Therefore, we can delete a clause from the formula list if we have one clause to be true.

2. We can shorten clauses with a false literal

If L_1 is false, then the result of $L_1 \vee L_2 \vee L_3 \ldots$ is same as the result of $L_2 \vee L_3 \ldots$

3. An empty clause means the formula is false

From the 2nd point, if we keep on getting false literals and removing them from the clause, there may come a point where we're left with only one literal in the clause, and if that is false too, we remove it to end up with an empty clause. Thus, the clause is false and so is the conjunction in which it is involved i.e., the formula.

The input in CNF form is then used to get the final answer using DPLL. The variables are stored in a list and the formula itself is stored in a vector of vectors. We also use the frequency of each variable so that assigning variables which occur most in the early stages would increase the chances of getting a model solution.

We first look for unit literal and pure literal and then go for most assigning the most frequently occurring variable. The code recursively calls DPLL until it has found a solution or has concluded that no solution exists. The output shows either UNSAT or if SAT, displays a model with variables in positive or negated form.

Assumptions

The implementation assumes that the input is given in the CNF form in a file named "input.cnf" and this file is kept in the same directory as the source code.

Limitations

The implementation is on C++ with DPLL but faces a time limit issue in cases with large number of clauses. With increasing variables and clauses, the time grows exponentially and the solver takes a lot of time to show the result for large cases.