50.001

Intro to Information Systems & Programming

2D Challenge Report

Group: *4*

Group members:
Cao Bing Quan (1003881)
Tjoa Jing Sen (1003456)
Kaye Tan (1003362)
Tan MeiZi Sherene (1003813)
Tiong Shan Kai (1003469)

Introduction

We implemented SATSolver.solve() by following the DDPL algorithm provided in the handout. The input for the function would be a CNF and the function will generate a text file called "Bool_Assignment.txt" of all the literal boolean assignments if the CNF is satisfiable. The text file shows the assignments of all literals as TRUE or FALSE. If the CNF is unsatisfiable, no text file will be generated and the system will print "Results: Unsatisfiable".

To run the script, first change directory to the src folder in the command prompt. Compile the script by using "javac sat/SATSolverTest.java" and run by passing the full filename as the main argument using "java sat/SATSolverTest test_2020.cnf".

Results

Time: 950.5377ms

Results: Unsatisfiable

Specification: Windows Desktop, AMD Ryzen 5 3600 6-Core Processor, 3593 Mhz, 6 Core(s), 12 Logical Processor(s)

Approach

"solve" function

Once the CNF file has been parsed, we first check if the clause list is empty. If it is empty we will return an empty environment. This means that formula is satisfiable. Next, we loop through the clause list to check for empty clauses and to find the smallest clause. Using a for loop, we replace the *smallestClause* variable with the next iteration if it has less literals. This will allow us to find the clause with the smallest size stored in the *smallestClause* variable. If the clause list contains an empty clause, we can immediately return that the CNF is unsatisfiable. We instantiate the first literal of *smallestClause* in the variable *literal*. If the *smallestClause* variable contains only one literal, we bind its variable in the environment and call the *substitute* function for this literal. We then recursively call the *solve* function. If the clause contains more than one literal, we will pick the first literal that was instantiated previously as an arbitrary literal. We call the substitute method for this literal, and solve it recursively using *solve* function. If the output is null, we then repeat the previous step with the literal set to FALSE.

Our interpretation of this algorithm is like switching n number of literals on or off. As such, the worst-case run time algorithm is $O(2^n)$. We ensured that no unnecessary nested for or while loops have been implemented and kept instantiating new variables to the bare minimum to minimize the run time of our code.

"substitute" function

The *substitute* method produces a new clause list resulting from setting a target literal to TRUE. When the *substitute* method is called, a new clause list is instantiated. The function iterates through all the clauses in the clause list and checks if the negation of the target literal exists in the current clause. If the negation exists, this means that the literal has not been negated yet so we set the target literal to be TRUE by calling the reduce method to obtain a new clause. The new clause is added to the new clause list. If the clause does not contain the target literal, just add it to the new clause list. We do not add already negated literals to save on computation time for future *substitute* calls. Finally, the updated clause list is outputted.