CS202A Assignment-2

Contributors

Kushagra Sharma (200539)
Mandar Wayal (200556)

Introduction

The aim was to design a **SAT Solver** in any programming language using any algorithm.
As we were a team of two, we implemented **two different approaches** in **C++**, namely, the **DPLL algorithm** and **Semantic Tableau**.

Note:
Comments have been added at appropriate places so that it is easy to follow.
Instructions to run can be found in the README.

DPLL Algorithm

The basic idea of the DPLL algorithm is that it determines a 1-assignment for a given formula in CNF in clausal notation in a backtracking fashion step by step, setting one variable after the other. Whilefinding a satisfying assignment for a propositional formula, set one of the variables to true/false and then work recursively on the two resulting simpler formulas.

Implementation:

The input DIMACS representation is stored as a 2D vector (named $encoding)$ in such a way that $encoding[i]$ represents the ith clause and $encoding[i][j]$ represents the jth literal of the ith clause.

We have defined various functions in the program for simplicity in understanding. Let our formula in DIMACS representation be called F. Note that we copy and modify this formula at many steps (i.e. it is not fixed).

function **DPLL(F)**
It is the primary function that checks if a formula F is SAT or not (and also keeps appending proper literals in our variable, *model*). The pseudo code is as shown:

```
if $F$ has no clauses, return true
if $F$ has an empty clause, return false
if $F$ contains a clause having single literal, return DPLL(simplify($F$,literal))
$v$ ← a random literal from some clause
if DPLL(simplify($F$,$v$))  is true, return true
else, return DPLL(simplidy($F$,$\neg v$))
```

Before calling the function $simplify(\ldots)$, we also store that particular literal as true in our model. Conversely, when we get a contradiction (in the form of $false$), we remove that literal from our model and try negation of that literal.
This is the essence of backtracking.

***function* simplify(** F, *literal* **)**
This function reduces the encoding in the following way:

- removes the clauses where the literal exists (that clause is satisfied)
- removes negative of the literal if it exists in some clause

The design is as shown:

```
remove clauses in $F$ where the literal is true
remove $\neg literal$ from clauses where it appears
return modified $F$
```

Rest of the auxiliary functions are quite self-explanatory in term of what they help accomplish.

<u>**Tweaking for Optimization**</u>:

While choosing a literal to split the encoding on, we took the first literal of the first clause. Later, we found this question.
A simple way to optimize DPLL is to choose the literal randomly. Hence, we have an auxiliary function to generate a randomness called *randomNumGen*. On a large sample, this tweaked program would give better results. The submitted code is the modified one.

Semantic Tableau Method

Semantic Tableau creates a tree structure of various literals. We first split the α - formulae (Conjunctions) and then the β-formulae(Disjunctions). The α formulae are just replaced with a ',' whereas splitting the β formulae creates two different branches.
Since the $DIMACS$ representation contains has the disjunction of all literals in a clause and all the clauses in conjunction, it can be observed that the $leaf$ nodes of the Semantic Tableau Tree will have a single literal from each of the original clause.

Implementation:

The clauses are stored in a 2-D vector. We keep an iterator for each of the rows of the vector, (Initially pointing to the first element of all the rows).
The current combination of elements pointed by the iterators represented one of the various $leaf$ nodes of the Semantic Tableau Tree.
We see if there is a contradiction ($p \ and \ - p$ present together).
If there is no contradiction, the current combination of literals represents a model for the system.
Else we point the iterator to the next element and check again. (If the current element is the last in the vector, we point it to the beginning of the vector and add carry to the previous row in the 2-D Vector. And the loop exits if the iterator of the first vector reaches the end).

Note:

Since Semantic Tableau is a Brute Force method, the time complexity can increase exponentially as we increase the number of clauses. So the code can take a lot of time to run when the number of clauses is quite large.