Singapore University of Technology and Design

50.004: Introduction to Algorithms

2D Group 13

Zhang Peiyuan 1004539

Qiao Yingjie 1004514

Ryan Seh Hsien En 1004669

Chua Qi Bao 1004494

Lim Hng Yi 1004289

Peh Jing Xiang 1004276

Date of submission: November 6, 2020

1. Introduction

The task is to build a 2-Satisfiability Problem Solver executed in polynomial time, where the input expression is to be read from a .cnf file and the algorithm to be implemented in Java.

In our deterministic approach, we construct a directed graph with the given proposition formula and apply Tarjan's Algorithm[1], which is essentially Depth-First-Search, to produce a sequence of Strongly Connected Components (SSC)[2] of the implications graph, based on which we can reason the satisfiability of the given expression. The working code is available at our GitHub repository[3], which is also submitted as a .zip file including this report on edimension, and will also be presented along with detailed reasonings in the following sections. Notably, our algorithm is able to solve this question in linear time, where the derivation can be found in Section 7.

It should be particularly noted that this solution only works for 2-SAT propositional expressions, since each clause in a 2-SAT expression can only have 1 or 2 literals, indicating that each clause will be true as long as at least 1 literal in the clause is true. More detailed elaborations will be presented in Section 8.

2. Problem Analysis

The 2-SAT propositional expression can be illustrated as:

F = (x1 OR x2) AND (x2 OR X3) AND (NOT x1 OR x3) AND … (xn-1 OR xn)

For the expression to be satisfiable, each clause should be true. Since each clause in a 2-SAT expression consists of either 1 literal or 2 literals:

Case 1: Suppose the clause consists of a single literal *A*. Obviously, the literal *A* should be true if the clause is true and vice versa.

Case 2: Suppose the clause consists of 2 literals, represented as (*A* OR *B*). If the clause is true, then:

- If A == false, B must be true. If A == true, B can be either true or false. Thus, $NOT\ A\ \Rightarrow\ B$.
- If B == false, A must be true. If B == true, A can be either true or false. Thus, $NOT\ B\ \Rightarrow\ A$.

Therefore, in an implication graph of 2-SAT expression, the 2 clause expression (*A* OR *B*) is equivalent to *edge(NOT A → B)* and *edge(NOT B → A)*.

With this discovery, considering that:

Case A: if edge(X → NOT X) exists in the implication graph, which can be interpreted to be $X\ \Rightarrow\ NOT\ X$.

Based on this expression:

- If X == true, NOT X == true, which is apparently a contradiction. Abort.
- If X == false, the implicated NOT X can be either false or true arbitrarily.

Therefore, X == false in Case A.

Case B: if edge(NOT X → X) exists in the implication graph, which can be interpreted to be $NOT\ X\ \Rightarrow X$.

Based on this expression:

- If NOT X == true, X == true, which is apparently a contradiction. Abort.
- If NOT X == false, the implicated X can be either false or true arbitrarily.

Therefore, X == true in Case B.

Case C: if both edge(X → NOT X) and edge(NOT X → X) exist in the implication graph, which can be interpreted to be $X\ \Rightarrow\ NOT\ X$ and $NOT\ X\ \Rightarrow X$.

According to the analysis on Case A and Case B above, it can be inferred from the first edge that X == false while it can also be inferred from the second edge that X == true, which is a contradiction.

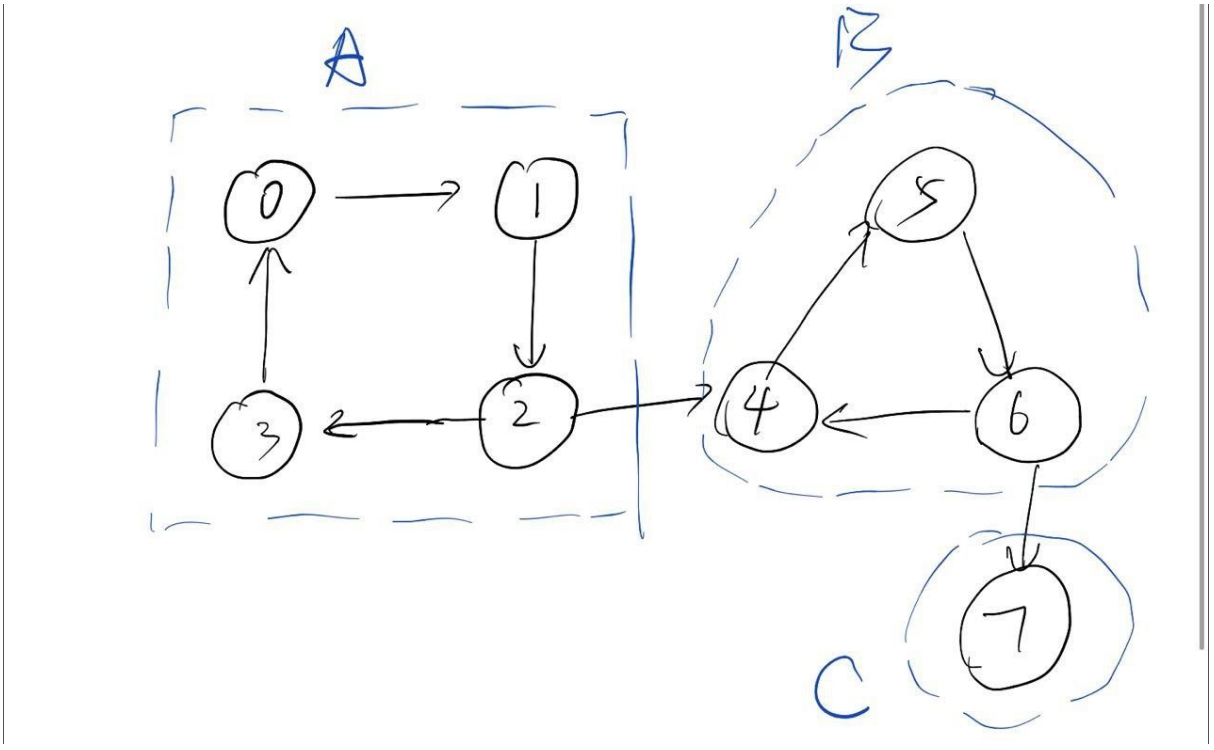
Therefore, X does not have a valid solution in this case and the clause is thus false.

As such, if 2 variables *X* and *NOT X* present as a "cycle" in the implication graph, such as X → A, A → NOT X, NOT X → B, B → X. Using the property that if X → A, A → NOT X, then X → NOT X, the cycle indicates that at least 1 clause is false and the propositional expression is thus unsatisfiable. Otherwise, there would be a possible solution that the propositional expression is satisfiable.

If X and NOT X are present in a cycle, X and NOT X lie in the same Strongly Connected Subgraph and vice versa. Thus finding whether the 2-SAT problem is satisfiable is equivalent to finding whether both X and NOT X lie in the same SCC.

An SCC of a directed graph is a maximal strongly connected subgraph where every vertex is reachable from every other vertex, which can be found by Tarjan's Algorithm.

For instance, the following figure serves as an example to illustrate SCCs in a directed graph. There are 3 SCCs in the given graph denoted by A, B and C, where every vertex is reachable from every other vertex in each SCC. It should be noted that a single vertex could also be regarded as a standalone SCC based on the definition, which is the case of SCC "C" in this graph.



The Tarjan's algorithm will be discussed in more detail in Section 4 as the flow of reasoning proceeds.

In conclusion, an implication graph should be constructed, and then Tanjan's Algorithm can be applied to find the SCCs. Finally, the satisfiability can be determined by checking whether X and NOT X are present in the same SCC.

3. Input Loading and Implication Graph Construction

As explained in the previous section, the "inference" relationship is similar to a directional relationship in a graph, for which the graph can be constructed as a singly-linked graph.

The constructor of the implication graph can be found in Graph.java:

```
public class Graph {
  HashMap<Node, HashSet<Node>> adjacencyLists;
  HashMap<Integer, Node> nodes;
  public int index;  // index used in Tarjan's algorithm
  public Stack<Node> stack = new Stack<>();
  public List<List<Node>> SCCs = new ArrayList<>();

  Graph(List<Integer> vertices, List<List<Integer>> edges) {
    adjacencyLists = new HashMap<>();
    nodes = new HashMap<>();
    for (Integer id: vertices) {
      Node vertex = new Node();
      vertex.id = id;
      adjacencyLists.put(vertex, new HashSet<>());
      nodes.put(id, vertex);
    }
    // add all edges
```

```
        for(List<Integer> edge: edges) {
            Node from = nodes.get(edge.get(0));
            Node to = nodes.get(edge.get(1));
            Set<Node> set = adjacencyLists.get(from);
            set.add(to);
        }
    }
    @Override
    public String toString() {
        return adjacencyLists.toString();
    }
}
```

The vertices input of the graph constructor represents all the individual literals presented in the propositional expression and the edges input represents each clause in the expression, i.e. every pair of literals.

To aid the application of Tarjan's algorithm, a Node.java class is created to define each vertice as a Node object, which comes with the attributes id, index, lowlink, instack:

```
public class Node {
    public int id;
    public int index;
    public int lowlink;
    public boolean inStack;

    public String toString() {
        return Integer.valueOf(this.id).toString();
    }
```

}

Time Complexities:

In the following analysis, *V* stands for the total number of vertices, *E* stands for the total number of edges, and *C* represents a positive constant which stands for the number of comment lines in the input .cnf file.

1. Loading the .cnf file and extracting the variables in it essentially iterates through the entire propositional expression, which thus takes *O(E + C)* time.
2. Iterating through the list of vertices and assigning each vertice to a Node takes *O(V)* time.
3. Iterating the list of edges and adding each vertice to the corresponding adjacencyList takes *O(E)* time.

Therefore, it takes *O(E + V + C)* time to construct an implication graph of the input expression.

4. Finding Strongly Connected Components (SCCs)

Tarjan's algorithm can be implemented in such a way:

1. An integer variable index is created to mark visited nodes.
2. A stack is created, which will be used to store visited nodes in Tarjan's algorithm later.
3. An arraylist SCCs is created to store the Strongly Connected Components to be discovered.
4. Maintain an integer attribute lowlink for every node, which indicates the smallest/lowest node index reachable from that node (including itself) when iterating through the graph, which is essentially DFS.
5. Iterate through every node, which is essentially DFS:
    a. Assign the value of index to node.index to mark this node as visited, and also assign index to node.lowlink for SCC later.
    b. Push the visited node into stack and increase the value of index by 1.
    c. Iterate through the nodes that are connected with the current node where there are 3 possible cases:
        i. If the node is not visited, repeat the Step a - c with that unvisited node, **which is essentially a <u>recursive</u> call of _tarjan in deterministic.java.** Update node.lowlink after visiting.
        ii. If the node is visited, and the node is in the stack, update node.lowlink.
        iii. Pass and do nothing if both cases above are satisfied.
    d. If node.lowlink equals to node.index, then an SCC is found. Pop out all the nodes currently in the stack including the current node and then create an arraylist, SCC, consisting of these nodes to represent the

Strongly Connected Component found. Add the arraylist SCC to the arraylist SCCs.

6. Return SCCs.

Nodes in the stack with the same lowlink values implies that these nodes are in the same SCC because lowlink represents the smallest/lowest node index reachable from that node including itself as stated earlier. In the edge case where the entire graph is a cycle itself, then the entire graph is one big SCC. Otherwise, in more general cases where the graph is acyclic, it also covers the case in which there is only 1 node in the stack. The single node thus forms a standalone SCC with only 1 node, where an example is illustrated in the figure in Section 2 near the end (SCC "C").

The advantage of Tarjan's algorithm is that the Depth First Search only needs to be performed once, which is Step 5.

The Tarjan's algorithm implementation can be found in deterministic.java:

```
public static void tarjan(Graph g, Set<Node> nodes) {
  for(Node node: nodes) {
     if(node.index != 0) {
        return;
     } else {
        _tarjan(g, node);
     }
  }
}
```

```
private static void _tarjan(Graph g, Node node) {
    node.index = g.index;    //dfs start time of current node
    node.lowlink = g.index; //the minimum dfs start time among all the nodes it can visit
    g.index += 1;
    g.stack.push(node);
    node.inStack = true;

    for(Node next: g.adjacencyLists.get(node)) {
        if(next.index == 0) {
            _tarjan(g, next);
            node.lowlink = Math.min(node.lowlink, next.lowlink);
        } else if(node.inStack) {
            node.lowlink = Math.min(node.lowlink, next.index);
        }
    }

    if(node.lowlink == node.index) {
        List<Node> SCC = new ArrayList<Node>();
        Node poppedNode = null;
        while(node != poppedNode) {
            poppedNode = g.stack.pop();
            poppedNode.inStack = false;
            SCC.add(poppedNode);
        }
        g.SCCs.add(SCC);
    }
}
```

Time Complexities:

Due to the fact that visited nodes are marked and not visited repeatedly, **each node in the implication graph is visited exactly once by _tarjan(Graph g, Node node).** Due to the very same reason, each node can only be pushed into the stack and popped out once.

For each node, we need to examine all of its outgoing edges. We only visit each node once, so we only examine those edges once.

Therefore, the time taken to visit all the nodes in the graph is

O(total number of edges + total number of nodes)

With *V* standing for the total number of vertices, *E* stands for the total number of edges, the time complexity is

$$O(V + E)$$

A similar algorithm, Kosaraju's Algorithm, is also discussed in Section 22.5 of the textbook, Introduction to Algorithms. Even though Tarjan's Algorithm and Kosaraju's Algorithm's time complexities are both linear. The advantage of Tarjan's algorithm is that it only needs to perform DFS once, while Kosaraju's algorithm needs to do 2 DFS.

5. Determine Satisfiability

As analyzed in Section 2, a graph is unsatisfiable if there is at least 1 variable *X* such that both *X* and *NOT X* exist in an SCC; otherwise it is satisfiable. With Tarjan's algorithm explained in the previous section, the SCCs of the implication graph can be found by applying Tarjan's algorithm and the satisfiability can thus be determined by checking whether there is a variable and its inversion exist in any of the SCCs at the same time.

The algorithm to determine satisfiability can be found in deterministic.java:

```
public static boolean satisfiable(Graph g) {
  for(List<Node> SCC: g.SCCs) {
    HashMap<Integer, Boolean> variables = new HashMap<>();
    for(Node n: SCC) {
      int varId = Math.abs(n.id);
      if(variables.containsKey(varId)) {
        return false;
      } else {
        variables.put(varId, true);
      }
    }
  }
  return true;
}
```

Time Complexities:

A hashmap is created for each SCC which enables *O(1)* amortized time for membership checking. As the for loop effectively iterates through every vertice in the graph, the overall time complexity is *O(V)*, where *V* represents the total vertices.

6. Finding a Solution

If a propositional expression is determined to be unsatisfiable, then there is no viable solution to be found.

If a propositional expression is determined to be satisfiable, the solution can be found by iterating each node in reverse topological order. For each node we visit, if this node's corresponding variable's boolean value has not been assigned, assign its value so that this node is set to true. Because our list of SCCs and nodes in each SCC are already in reverse topological order, we just need to iterate them and perform this algorithm.

```
public static Map<Integer, Boolean> solve(Graph g) {
  HashMap<Integer, Boolean> solution = new HashMap<>();

  for(List<Node> SCC: g.SCCs) {
    for(Node n: SCC) {
      if (solution.containsKey(Math.abs(n.id))) continue;
      int id = n.id;
      if(id > 0) {
        solution.put(id, true);
      } else {
        solution.put(-id, false);
      }
    }
  }
  return solution;
}
```

Proof of correctness:

Let's assume in our algorithm, we visit node A first and then node B. Because B is before A in terms of topological order, there will only possibly be an edge from node B pointing to node A. A very important property can be deduced: If there is an edge from B to A ($B \Rightarrow A$) and B is true, A must be true.

Let's prove this property by contradiction. Firstly, we assume A is false, B is true and edge B → A. According to the way we construct our implication graph, there must also be an edge NOT A → NOT B. According to our algorithm, the only possible reason why node A is set to false is because we set NOT A to true beforehand. From the proposition NOT A → NOT B, we can deduce that NOT B has a bigger topological order than NOT A. In other words, we must have visited NOT B before NOT A in our algorithm. Again, NOT B is set to false and it can only be caused by the fact that we set B to true beforehand. This contradicts our assumption that B is visited after A, So the property is proved.

Using this property, we can say that for any edge B → A. Our algorithm will never assign B as true and A as false at the same time. So the proposition is always true. Then the whole implication graph is true and satisfied.

Time Complexities:

Since it is iterating through every vertice in the graph, the time complexity of finding a solution is *O(V)* where *V* represents the total vertices.

7. Overall Time Complexity

With *V* standing for the total number of vertices, *E* standing for the total number of edges, and *C* representing a positive constant which stands for the number of comment lines in the input .cnf file, the overall time complexity is:

$$O(V + E + C) + O(E + V) + O(V) + O(V) = O(V + E + C).$$

Since *C* is a positive constant which can be neglected in asymptotics notations, the overall time complexity can be simplified to:

$$O(V + E),$$

which is essentially **linear time** as the algorithm iterates through every vertice and every node in the implication graph, and therefore satisfies the polynomial time requirement.

8. Feasibility on 3-SAT Problems

This core idea of this algorithm for the 2-SAT problem lies in the derivation of the implication graph presented in Section 2. Since each clause consists of at most 2 literals, as explained in Section 2, each clause (A OR B) can be interpreted as either $X \Rightarrow NOT\ X$ or $NOT\ X \Rightarrow X$, for which a directed implication graph can be constructed and then the Tarjan's algorithm can be applied to generate SCCs to determine satisfiability.

However this is not the case for 3-SAT problems or, more generally, N-SAT problem where N > 2. Referring to the Case 1 and Case 2 discussed in Section 2, given a 3-literal clause in a 3-SAT problem (A OR B OR C), if this clause is true, let's just consider 1 case first:

Case 1: if (A OR B) is false, C must be true. Therefore, $NOT\ A\ OR\ NOT\ B \Rightarrow C$.

It can be observed that it is no longer a 1-node to 1-node relationship in a 3-SAT problem, but a 2-node to 1-node relationship in this simple case. For the $NOT\ A\ OR\ NOT\ B$ to be false, both A and B need to be true and the resulted 4 implications are: $A \Rightarrow B$, $B \Rightarrow A$, $NOT\ A \Rightarrow NOT\ B$, and $NOT\ B \Rightarrow NOT\ A$.

Unlike a 1-node to 1-node relationship where only 1 implication (either $B \Rightarrow A$ or $A \Rightarrow B$) is derived from each clause, there are multiple possible implications for each clause in 3-SAT, and thus a directed implication graph is no longer feasible.

In conclusion, the implication graph approach is not feasible on 3-SAT problem or N-SAT problem where N > 2 in general.

9. Randomized Algorithm

A randomized approach to the 2-SAT problem is also developed:

1. Start by an arbitrary truth assignment. In our algorithm, we always start by setting all literals to be false.
2. Set an upper bound of times of trying. In our algorithm, this upper bound is set to be 100 multiplies the square of the total number of literals.
3. Check the satisfiability after the random assignment:
    a. If the formula is satisfiable, then the random assignment is indeed a solution.
    b. If the formula is unsatisfiable, then
        i. find the clause that makes the formula unsatisfiable and change its value to be the opposite
        ii. Go back to Step 3 and check the satisfiability again. The iteration of trying to find a satisfiable solution is to be terminated until the upper bound set defined in Step 2 is reached.
4. If an assignment that makes the formula satisfiable is found in Step 3, then this formula is indeed satisfiable as we already found a solution by randomly trying. If an assignment that makes the formula satisfiable is not found in Step 3 until we exhausted the upper bound of trying defined in Step 2, then a conclusion can be drawn that the formula is unsatisfiable.

Time Complexities

Referring to the randomized algorithm implemented in randomized.java,

Step 1 and 2 take constant time $O(1)$.

Step 3, which is the nested for loop in solve(), takes $O(2000 * N^2) = O(N^2)$ time where N represents the total number of literals in the propositional expression.

Therefore, the time complexity of the randomized algorithm is $O(N^2)$.

Comparing to the Deterministic Approach

1. **Performance**

Since our deterministic algorithm implemented in Tarjan's algorithm has a linear time complexity $O(V + E)$ where V stands for the total number of vertices and E stands for the total number of edges in the implication graph, while the randomized algorithm has a polynomial time $O(N^2)$, the deterministic algorithm is a more optimized and faster approach to solve the 2-SAT problem with much better performance.

2. **Rigor of the Correctness**

Our deterministic algorithm is guaranteed to give a correct conclusion on the satisfiability of a propositional expression, with the rigorous reasoning provided in earlier sections. While the randomized algorithm cannot guarantee to draw an absolutely correct conclusion since it is not trying to flip the coin on every literal to make sure the clause is true. Our randomized algorithm is only trying until $2000 * N^2$ literals causing the expression to be false are fixed. However, it may very well be the last few literals near the end of the expression that are beyond the pre-defined upper

bound causing the expression to be unsatisfiable, which, if fixed, could make the expression to be satisfiable! But since the algorithm would stop at the upper bound and draw the conclusion of unsatisfiable. The conclusion given here would be incorrect.

Moreover, if we are to fix this issue of the randomized algorithm, the upper bound has to be set to be 2^n where n represents the total number of literals. However, the time complexity would also increase to $O(2^n)$ as a result, which is essentially a brute force search and thus defeats the purpose of developing a more optimized algorithm with better performance than $O(2^n)$.

Therefore, in conclusion, **the randomized algorithm is not a practical substitute of the deterministic algorithm.**

References:

1. Tarjan, R. E. (1972), Depth-first search and linear graph algorithms, SIAM Journal on Computing, 1 (2): 146–160, doi:10.1137/0201010
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. MIT Press and McGraw-Hill. Section 22.5, pp. 615–616.
3. Our SAT Solver Code Repository: https://github.com/YingjieQiao/SATSolver