

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.004 2D 2SAT ALGORITHM ANALYSIS

PREPARED BY: TAN LI YANG 1002001

SOH CHI TONG KENNY 1002000

LEE GUAN WEI VICTOR 1002237

BRIAN CHEW EE DA 1001578

CHERYL GOH QIAN PING 1002421

ONG JING XUAN 1002065



**Table of Contents**

[Document Purpose and Scope 1](#_Toc497424767)

[Implementation 1](#_Toc497424768)

[File Processing 1](#_Toc497424769)

[Plot Implication Graph 1](#_Toc497424770)

[Determine Strongly Connected Components(SCC) 2](#_Toc497424771)

[Check for Contradiction 3](#_Toc497424772)

[Output 4](#_Toc497424773)

[Performance Analysis 4](#_Toc497424774)

[Conclusion 4](#_Toc497424775)

**Table of Figures**

[Figure 1 - Implication Graph 2](#_Toc497424537)

[Figure 2 - DFS Traversal 3](#_Toc497424538)

[Figure 3 - DFS Traversal on Reverse Graph 3](#_Toc497424539)

[Figure 4 - Directed Acyclic Graph 4](#_Toc497424540)

# **Document Purpose and Scope**

This document has three sections, Implementation, Performance Analysis and finally Conclusion. Implementation section will elaborate in detail how the 2-SAT Solver is implemented after which the performance of the proposed algorithm will be analysed.

# **Implementation**

The files that are needed to run the 2-SAT Solver are as follows:

1. FileExtract
2. Literal
3. Clause
4. Graph
5. SCC
6. TwoSATSolver

## File Processing

FileExtract.Java is used to extract the variables, with the CNF File as input. If the input is not a CNF File, an error will be shown. These variables will be converted to literals using Literal.Java. Clause.Java is used to create clauses by grouping the variables together. The clauses will be used to add edges to the graph to ensure that the nodes are connected correctly.

## Plot Implication Graph

To create the implication graph, first a Directed Graph is implemented using Adjacency List. Graph.Java will create the Directed Graph, allowing using the add Vertices and Edges. Vertices are created by adding the literals obtained from Literal.Java. Edges are created by connecting two Vertices together, with the output from Clause.Java as inputs which places two Literals together. This would ensure that the connection made between two Vertices is correct, as dictated by the Clause. These would be placed in a Hashmap, which sets the Vertices as keys and a Set of Outgoing Edges as Values. This keeps track the connection made for each individual Vertices to the other Vertices.

Using the example given, with the formula: (X1 or X2) and (X2 or X3) and (X3 or X4) and (~X1 or ~X3) and (~X2 or X4), the Clauses have a set of implications. The expected graph output will be as shown in Figure 1.

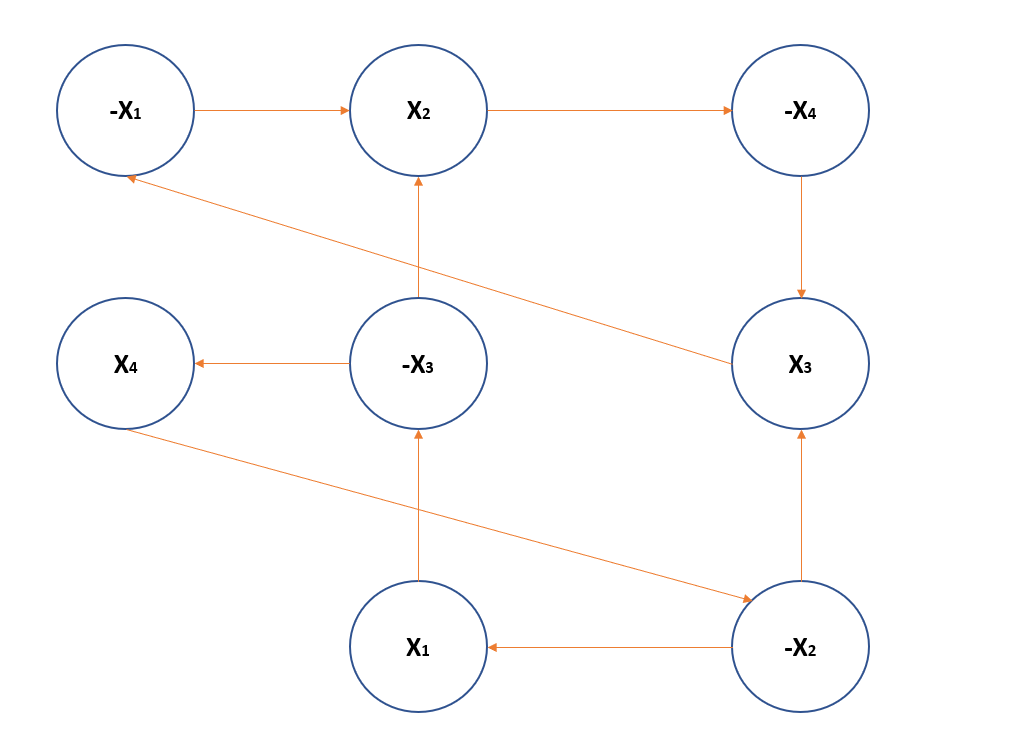


Figure - Implication Graph

## Determine Strongly Connected Components(SCC)

With the Implication Graph, SCC.Java is used to the create Strongly Connected Components (SCC) and group Vertices that are along the same path. This can be achieved using Kosaraju’s Algorithm. To start, a Stack is created and Depth – First – Search (DFS) is implemented to traverse through the graph. Adjacent Vertices are pushed to the Stack. A reverse Implication Graph in which the direction of all the edges would reversed. After which, DFS is implemented again and using the Stack, while it is not empty, pop the Vertices from the Stack. An example is shown in Figure 3.

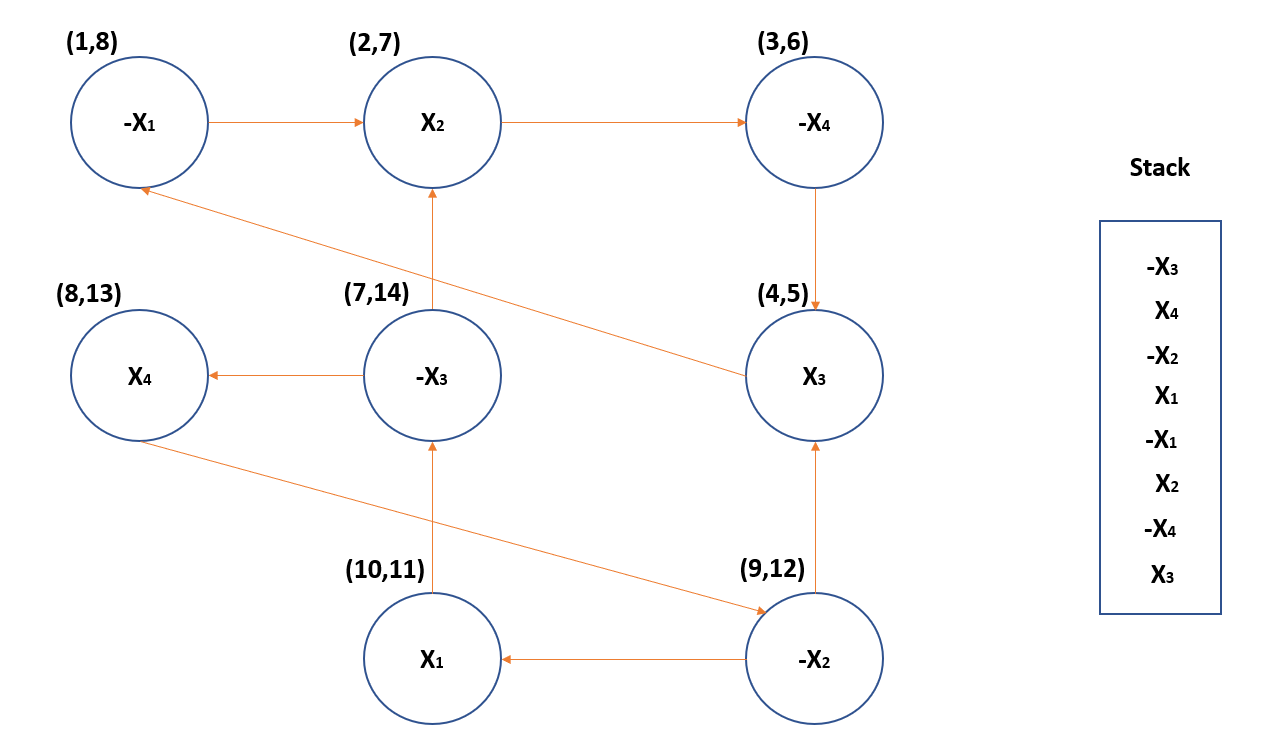


Figure - DFS Traversal

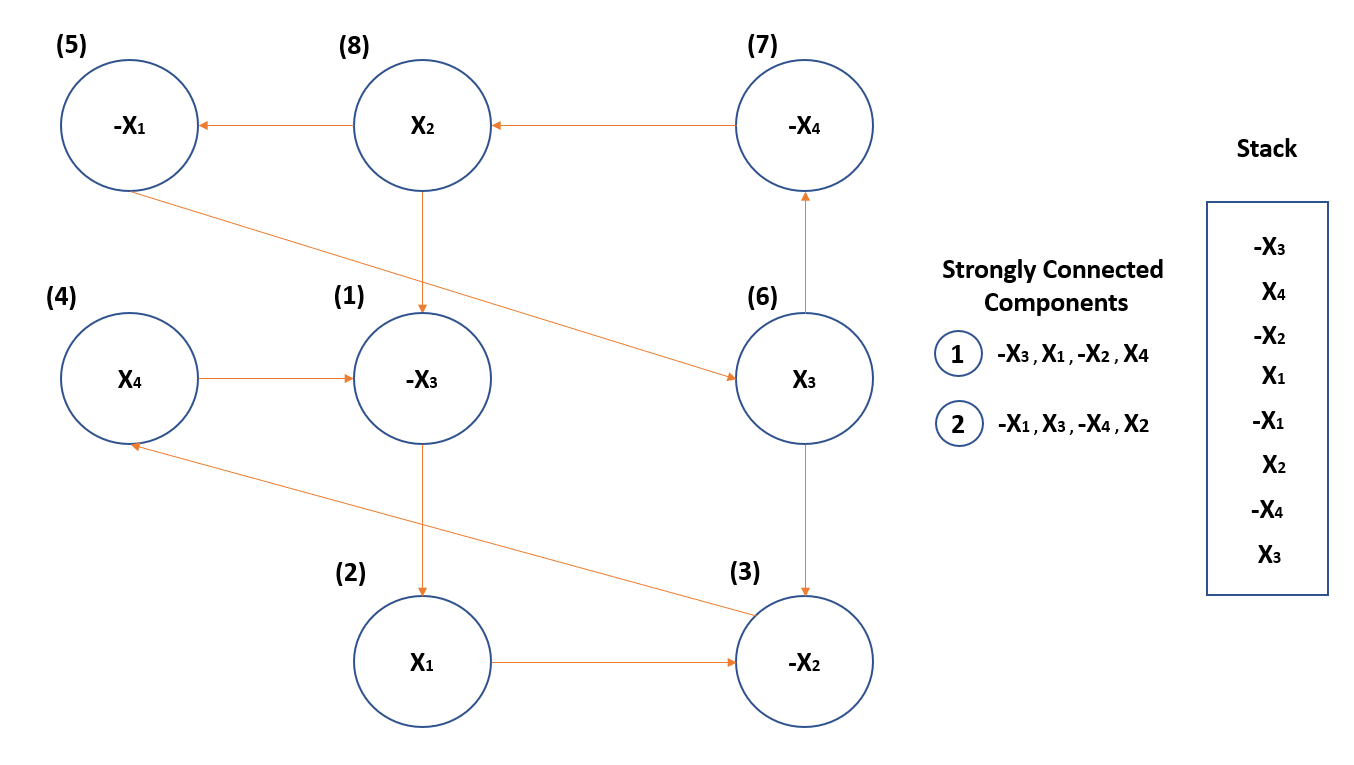


Figure - DFS Traversal on Reverse Graph

## Check for Contradiction

By implementing the SCC, we can check for contradiction to determine whether the formula is satisfiable or not. If a variable and its negation are in the SCC, the formula is deemed not satisfiable, as it is impossible to assign two Literals to the same variable. The formula is satisfiable if the variable and its negation do not exist within the same SCC.

## Output

By completing the check for contradiction, an output is shown to declare if the formula is satisfiable or not, mainly “FORMULA SATISFIABLE” and “FORMULA UNSATISFIABLE”. If the output is “FORMULA SATISFIABLE”, the solution for the formula will be shown. Do to this, a Directed Acyclic Graph is used to group the SCCs together connect them together. After which, going in topological order, 0 is assign to first Vertex and 1 to the next Vertex. This is shown in Figure 4.

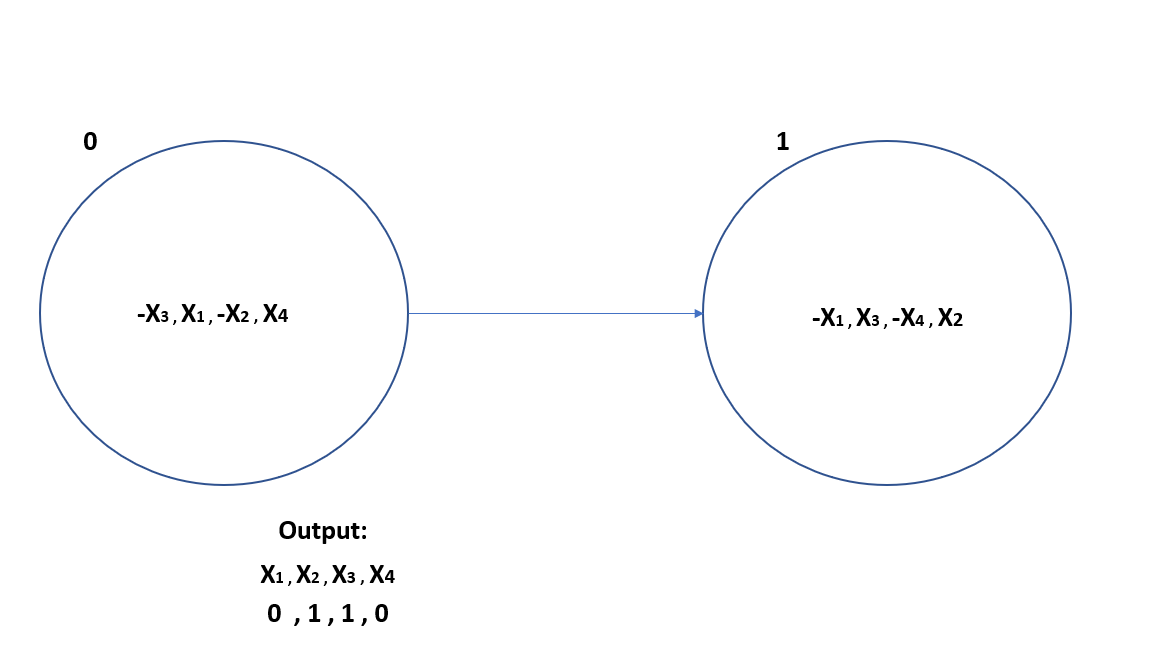


Figure - Directed Acyclic Graph

# **Performance Analysis**

With the proposed algorithm, the time complexity for the algorithm are as follows:

1. To create the Implication Graph, setting up the Vertices and Edges: O(V+E)
2. Implementing Depth – First – Search (DFS) to traverse through the Vertices in the Implication graph: O(V+E)
3. To create the Reverse Implication graph, setting up Nodes and Edges: O(V+E)
4. Implementing DFS again in the reverse Implication Graph: O(V+E)

Total Time Complexity: O(V+E)

Therefore, the proposed algorithm is running in linear-time. By running the 2-SAT Solver, the time taken is as follows:

# **Conclusion**

In conclusion,