50.004 Introduction to Algorithms | **2D Project**

Liew Cher Don | **1002662**

**| Problem Analysis and Proposal**

**Problem Definition**

A 2-Satisfiability (2-SAT) problem is a computational problem that finds the assignment for a set of Boolean variables, such that they satisfy a system of clauses, that have a maximum limit of two literals for each clause [1].

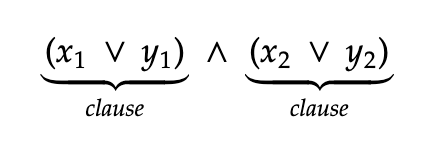


Figure 1.1: Example of a system of clauses in 2-SAT problem

Figure 1.1 above shows an example of a system of clauses, where “” represents OR and “” represents AND.

**Problem Specifications**

* Input of the problem must be in Conjunctive Normal Form (CNF), which would state the number of variables as well as the system of clauses
* Output must be the satisfiability analysis, either as “FORMULA SATISFIABLE” or “FORMULA UNSATISFIABLE”
* Output must also include the Boolean value (1 or 0) of each variable if the formula is satisfiable.
  + Example output for a 4-variable problem: 0 1 1 0

CNF Format

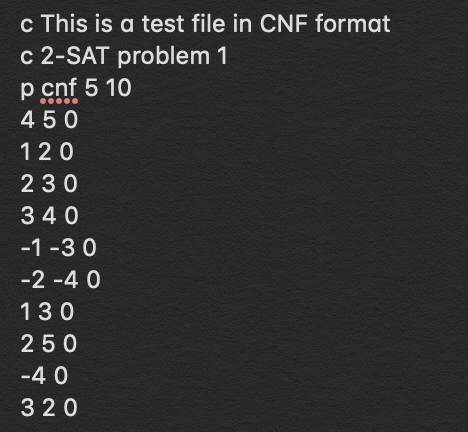


Figure 1.2: Example of 2-SAT problem in CNF format

Figure 1.2 shows an example of a problem representation in CNF format, where each line starting with:

* “c” represents a comment
* “p” represents the problem parameters, which would be followed in the sequence
  + p <FORMAT> <NUMBER OF VARIABLES> <NUMBER OF CLAUSES>

Subsequent lines are the clauses, where each number represents the variable number, and “0” representing next clause.

For example, lines 7 and 8 in Figure 1.2 represents:

Which is

**Proposed Algorithm**

The proposed solution is to use Tarjan’s Algorithm, which is a type of Topolocial Sorting method based on depth-first search (DFS) to find Strongly Connected Components (SCC).

To do that, we have to begin with creating an implication graph that represents variables and clauses as nodes and edges respectively, before performing Tarjan’s Algorithm.

All code can be found at <https://github.com/cherdon/algo2sat>

**| Solution**

**Implication Graph**

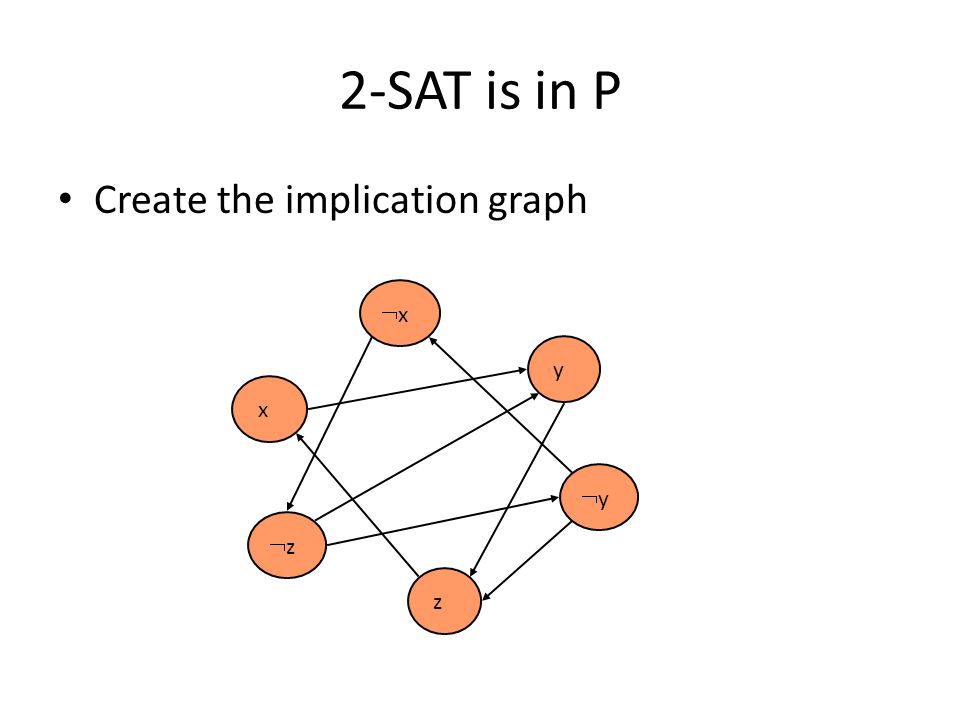
Vertices

Given all the variables of the problem, two vertices are created per variable (e.g. for the variable the vertices will be and ).

Edges

Given a clause , if one variable were to be false, the other has to be true for the clause to be valid. Hence, the clause can be interpreted and transformed into the edges and .

For unit clauses such as , the edge can be drawn as .

Figure 2.1: Example Implication Graph[2]

For example, Figure 2.1 shows the following system of clauses:

**Tarjan’s Algorithm**

Tarjan’s Algorithm is a Topological Sorting method, based on DFS (for graph traversal) and using a stack (for dynamic assignment of SCCs). The end result of Tarjan’s Algorithm is to produce SCC loops, via assignment of an index and low-link value to each node.

The index is the order in which the nodes are visited, while a low-link value is the smallest index of all its reachable nodes.

A random node is selected and assigned the smallest index. DFS will then be executed, where each node will be assigned an index based on the order it was visited and added to the stack

The nodes will remain on the stack until an earlier node on the stack is visited. This node will then be the core of the SCC loop, where all nodes after and including itself will be removed from the stack and have the same low-link value as the index of the SCC core node.

This recursive function will repeat itself until all nodes are visited and assigned a low-link value.

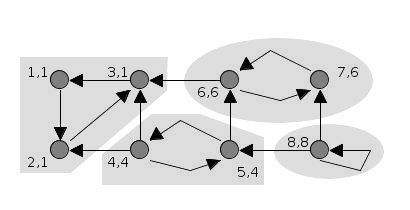


Figure 2.2: Tarjan’s Algorithm with low-link values and index[3]

Figure 2.2 shows an example of how the SCC loops (highlighted areas) are formed, where each node has an index (first integer) and low-link value (second integer).

Pseudocode

index = 0

while not all nodes visited:

visit(node)

function visit(node):

node.index = index

node.status = “visiting”

if it has child:

if child.status = “visited”:

pass

else if child.status = “visiting” and SCC = None:

SCC = list of nodes in stack from child to node

else:

**visit(children) // This is a recursive function**

if SCC exists and node in SCC:

node.lowlink = SCC first index

node.status = “visited”

SCC.pop(node)

if SCC is empty:

SCC = None

else:

node.status = “visited”

node.lowlink = minimum(node.index, first index in stack)

**Satisfiability**

Checking for satisfiability conditions is checking the SCC for any negations of variables. For example, would not be satisfiable since and contradict each other.

Otherwise, the formula is said to be satisfiable. The solution can be simply obtained by assigning the Boolean value of each SCC to be True, starting by the SCC with the largest low-link value. Subsequent SCC that have negation variables of already assigned SCC would be assigned a False Boolean value.

**| Output Analysis**

**Performance and Optimization**

Tarjan’s Algorithm is an effective and efficient method to obtain SCCs in linear/polynomial time. Randomly choosing a starting node to perform DFS traversal might result in getting incorrect low-link values, resulting in inaccurate SCCs. Hence, Tarjan’s Algorithm prevents this by removing the nodes from the stack once an SCC system has been found, preventing the revisiting of a node [4].

**Time Complexity**

Since Tarjan’s Algorithm is built upon DFS, every node is visited only once, and each edge is checked once. Hence, the asymptotic complexity is [4]. Since the time complexity is polynomial (for complex implication graphs) and linear (for simple implication graphs), this shows that the 2-SAT problem can be solved in polynomial time with essentially DFS.

**Limitations**

This algorithm would not work for 3-Satisfiability (3-SAT) problems, due to the fact that the edges in the implication graph would increase exponentially with the number of clauses [5].

Negations will also be complicated in 3-SAT problems. As such, implication graphs would not be the best solution for 3-SAT problems. Instead, other more suitable algorithms such as Binary Decision Diagrams (BDD) could be used [6].

**Other Algorithms**

Other algorithms that use implication graphs to find out SCC are suitable as well. Such algorithms are, for example, path-based strong component algorithm as well as Kosaraju’s algorithm.

**| References**

1. Wikipedia (2019, 23rd September), 2-satisfiability. Retrieved from <https://en.wikipedia.org/wiki/2-satisfiability>
2. Tel Aviv University, Blavatnik School of Computer Science (2014, November), 2-SAT Lecture Handouts. Retrieved from https:// [www.cs.tau.ac.il/~safra/Complexity/2SAT\_Handouts.pdf](http://www.cs.tau.ac.il/~safra/Complexity/2SAT_Handouts.pdf)
3. Wikipedia (2019, 5th September), Tarjan’s strongly connected components algorithm. Retrieved from <https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm>
4. William Fiset (2018, 21st March), Tarjan’s Strongly Connected Components algorithm | Graph theory. Retrieved from <https://www.youtube.com/watch?v=TyWtx7q2D7Y>
5. Stack Exchange (2018, 21st August), Why doesn’t implication graph work for 3SAT as it does for 2SAT? Retrieved from <https://math.stackexchange.com/questions/2684404/why-doesnt-implication-graph-work-for-3sat-as-it-does-for-2sat>
6. Peter Maandag, Radbound University (2012, 2nd July), Solving 3-SAT. Retrieved from <https://www.cs.ru.nl/bachelors-theses/2012/Peter_Maandag___3047121___Solving_3-Sat.pdf>