2D Challenge 2020

2-SAT algorithm implementation

Azeez Raasheeda Fathima (1004493)

Leong Yun Qin Melody (1004489)

Priscilia Pearly Tan Pei En (1004668)

Leon Tjandra (1004353)

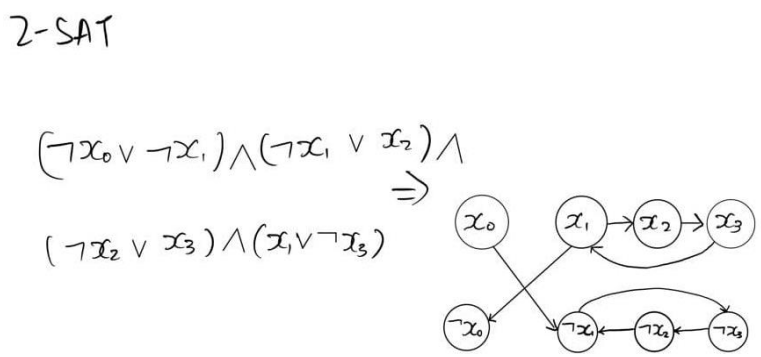
Mathias Koh Chin An (1004285)

Introduction

Our 2-SAT algorithm implementation uses the notion of strongly connected components from graph theory and Kosaraju's Algorithm. The 2-SAT problem involves making the CNF true. For example, AvB. The 2-SAT is unsatisfiable if there exists a variable x such that:

1. There is a path from x to ¬ x in the graph
2. There is a path from ¬ x to x in the graph.

Implication Graph



As shown in the figure, in order to satisfy all the clauses, when a literal is TRUE, all the output edges from its corresponding vertex carryout the adjacent literal to be TRUE. Moreover, there must not be any other variables where there is a path from it to its inversion.

In the later parts of the report, we will bring you through how in order for this 2-SAT problem to have a valid solution, it is required that for any variable xx, the vertices xx and ¬x¬x are found to be in different strongly connected components of the implication graph. For these criteria to be verified, the time complexity to find all the strongly connected components is to be O(n+m) where n is the number of vertices while m is the number of edges.

Strongly Connected Components

Kosaraju's algorithm is programmed to find strongly connected components of the graph. In brief, the algorithm runs Depth First Search (DFS) of the graph in two cycles where the first DFS of the graph identifies the stack order of each node, and the second DFS of the graph is done by utilising this stack order that we have found in the first cycle. At the end of the algorithm, each node, also known as the child node, will be assigned to a parent node, and nodes with the same leader nodes is grouped as one SCC. The algorithm complexity is O (m + n) where m is the number of edges and n is the number of vertices.

Next, we proceed to the implication graph into strongly connected components using the Kosaraju's Algorithm. A depth first search is performed on the whole graph starting at the first vertex, then visiting all of its child vertices, marking each done once it is visited. If a vertex leads to a visited vertex, then it pushes the visited vertex into the stack. Traversing through the implication allows us to check for direct neighbours next to the vertex if such a path exists. Hence, we apply the depth first search algorithm.

For an example, referring to the graph below,

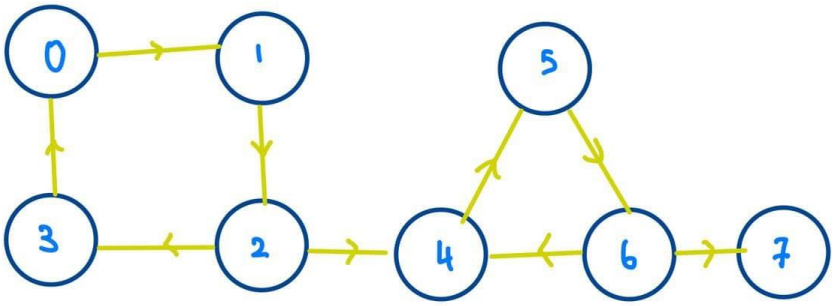


Diagram 1: How the implication graph is formed.

As we start from vertex-0, it visits its child vertex which is vertex-1, marking vertex-0.

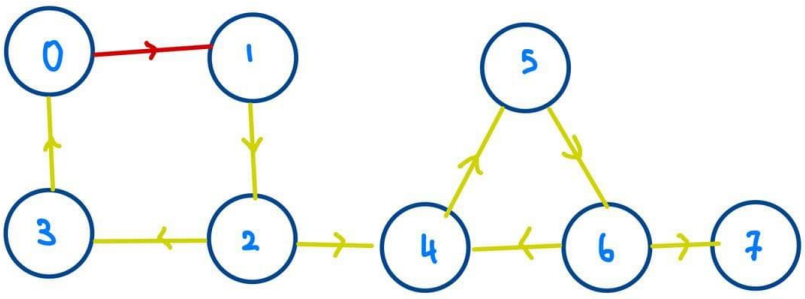


Diagram 2: How the graph travels from the source node to the next node highlighted in red

Vertex-1 then visits it child vertex-2 which then leads to vertex-3. As vertex-3 proceeds to visit its child vertex-0, since vertex-0 is already visited, it pushes vertex-3 into the stack.

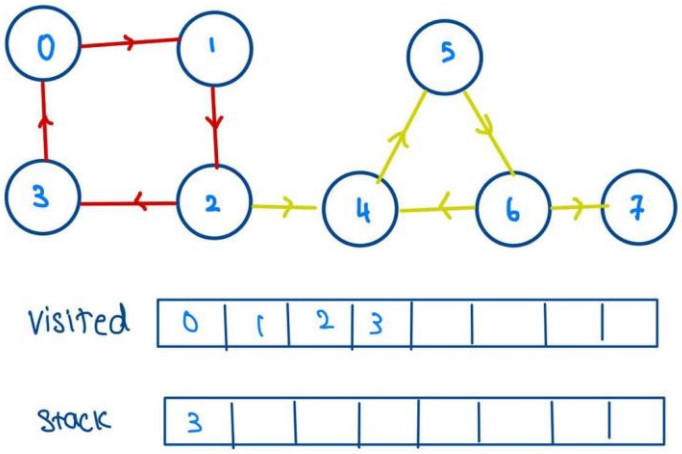


Diagram 3: The link between what vertex and its adjacent vertex is visited and then pushed to stack.

After the visited vertex is placed in the stack, the DFS proceeds to the previous vertex and visits its other child vertices in sequence. If the vertex has nowhere to go, it gets pushed into the stack after which it again proceeds to the previous vertex and visits its child vertices again. If all its child vertices are visited, the vertex will be pushed into the stack. In this case, this then results in returning to the previous vertex (vertex-2) and proceeds to the other non-visited vertices (4,5,6 and 7 in a sequence) as shown below.

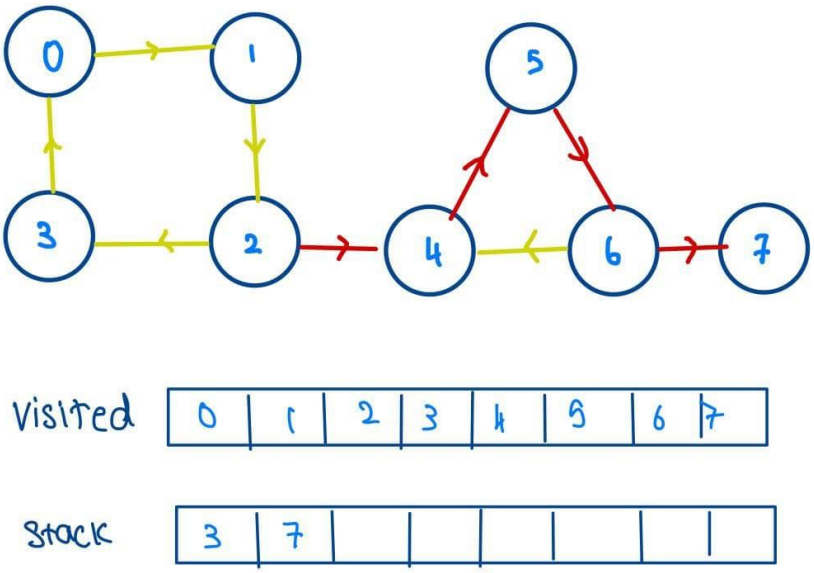


Diagram 4: How the travelling works to push to stack.

This repeats till all the vertices are pushed to the stack in the program manner of "*marked visited--> push to stack --> go to previous vertex --> check if all child vertices are marked, if yes --> push the current vertex to stack".*

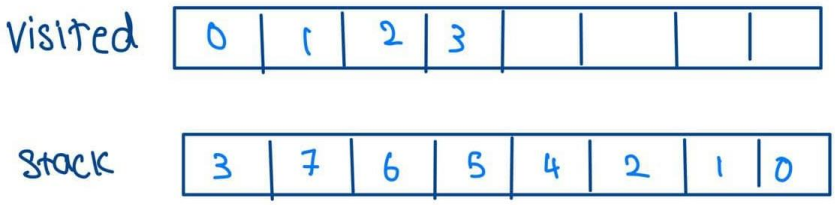


Diagram 5: The final stack before popping

Then, after all the vertices are placed in the stack, the directions of the edges on the original graph is then reversed (shown below) on which the depth-first search is performed again.

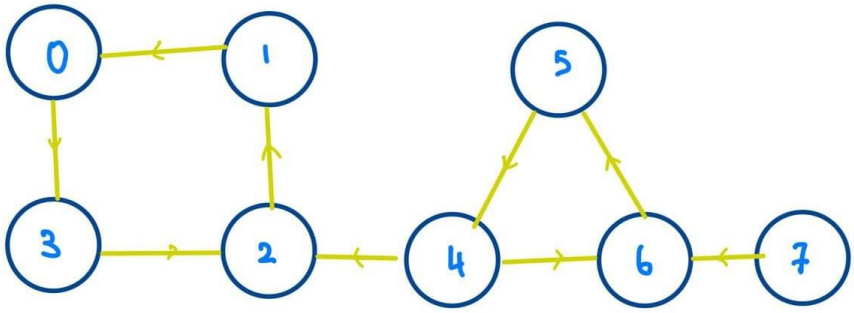


Diagram 6: Transposed graph

The DFS starts from the top vertex of the stack and traverses through all of the child vertices. When the child vertex is marked visited again, it is formed as a **strongly connected component (SCC).**

In this example, the DFS starts from vertex-0 and transverses through its child vertices in the sequence vertex-0 --> vertex-1 --> vertex 2 --> vertex-3 (stops at vertex-3 as the child of vertex-3 is already visited i.e., child vertex of vertex-3 is vertex-0 which is already marked) This therefore makes the vertices 0 to 3 as one strongly connected component is formed (as shown below)

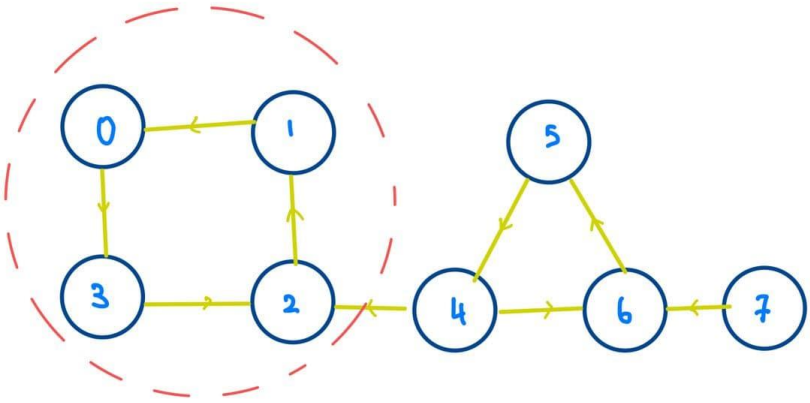


Diagram 7: Going to the stack and pop each vertex from the stack to trace the path from vertex A to B as well as B to A to find strongly connected components using DFS.

The same cycle repeats to find its SCC, where the SCC for this example is as follows:

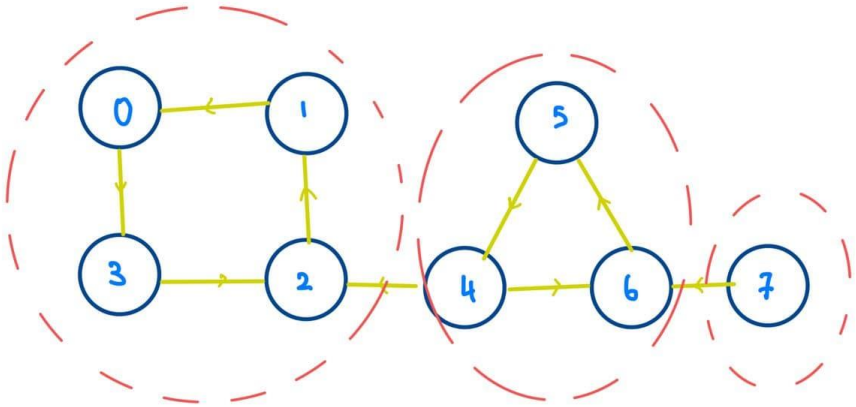


Diagram 8: Placing all the strongly connected components in groups to solve the 2-SAT problem.

In other words, this can be simply put in a recursive format where there are 2 subroutines that occur to find the SCC in the Kosaraju's Algorithm.

<u>Checking Satisfiability in the 2-SAT</u>

After finding the strongly connected components, to determine satisfiability, we will need to check whether the variable and its negation exists within the same strongly connected component.

If all the edges are satisfied by an assignment and there exists a path from A to B, then the truth value of A!=B. This means that all literals in the same SCC must have the same truth value. A formula is found to be unsatisfiable when both the variable and its negation exists

in a SCC. This is because it is impossible for a variable to be both TRUE and FALSE at the same time.

Let's consider the following 3 cases to explain why:

1. If edge (X → X') exists, that means X implies X'. To use Boolean for the above, if X== True, X' == true. Contradiction. If X==False, no implication constraints.
2. If edge (X' → X) exists, that means X' implies X. X' true means X is true. Same thing, contradiction. If X' is false, no implication constraints.
3. If edge (X' → X) and edge (X → X') exists. No solution.

Hence, for the 3rd case, the CNF is unsatisfiable. Otherwise, there is a possible assignment, and the CNF is satisfiable.

Solving details

The code involves two traversals of the implication graph. Kosaraju's algorithm does backtracking on the implication graph using depth first search. This ensures that each vertex of the graph is visited in order to solve the 2-SAT problem.

The algorithm solves the 2-SAT problem in the linear time complexity of O(n+m) where n is the number of vertices and m is the number of edges in the directed graph as explained briefly in the report in the earlier section.

A strong property of the Kosaraju algorithm uses the fact that the transpose graph has the same strongly connected components as the original graph.

Here are the respective purposes for the respective depth first searches:

- The first Depth first search aims to arrange the nodes in the stack based on which vertex comes first.
- Before proceeding to the second depth first search, a transpose occurs to identify paths between the nodes where the arrow directions are reversed to check for strongly connected components that exist on both path(a→b) and path(b→a).
- The second depth first search is then performed on the reversed graph and aims to find the strongly connected components between the adjacent vertices.

The following describes the steps for the algorithm and how the code is crafted. As for the variables stated below, L is the order of the list of graph vertices, that grows to contain each visited vertex once. *u* is the current vertex and *v* is the neighbouring vertex of *u*

1. Let *L* be empty. As for every vertex of the graph, it will be marked as unvisited.
2. When each vertex *u* of the graph is visited, Visit(*u*) occurs, where Visit(*u*) is the recursive subroutine:
    a. for *u* is unvisited:
        i. *u* will be visited

ii. *u* is marked as visited

iii. For neighbour *v* of *u*, do Visit(*v*).

b. Prepend u to L

Else, exit the routine.

3. For every vertex *u* of *L* which is placed in order in accordance with the first subroutine, do Assign (*u*, *u*) where Assign (*u*, *root*) is the recursive subroutine:

However, If *u* has not been assigned to a strongly connected component then:

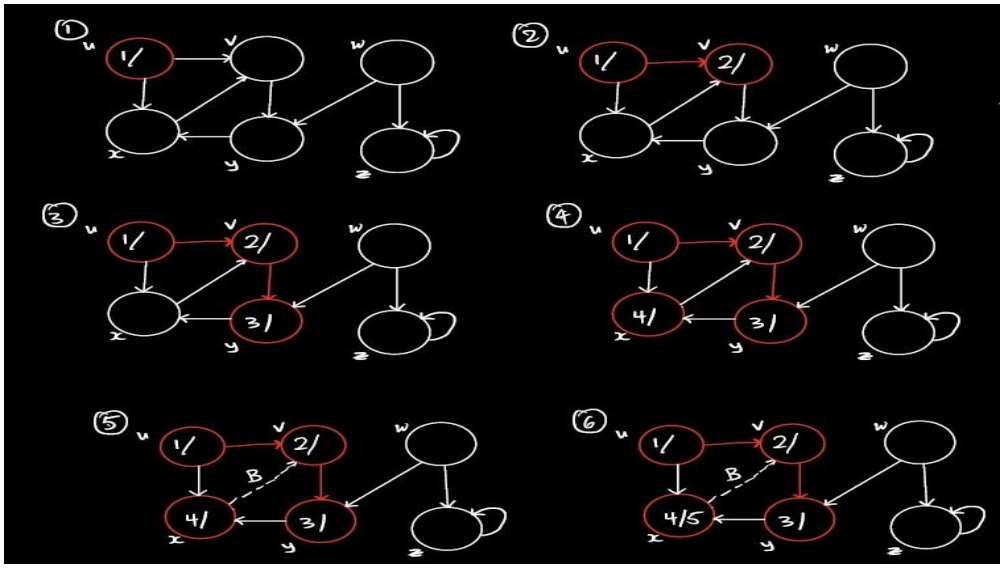
a. Assign *u* to the component whose root is *root*.
b. Continue by proceeding to neighbouring vertex *v* of *u and* assign(*v*, *root*).

Else, exit the subroutine.

<u>Kosaraju to depth first search (In more detail)</u>

The following properties of the Kosaraju algorithm and the depth first searches show a close relation with each other.

Depth first search has the following diagram



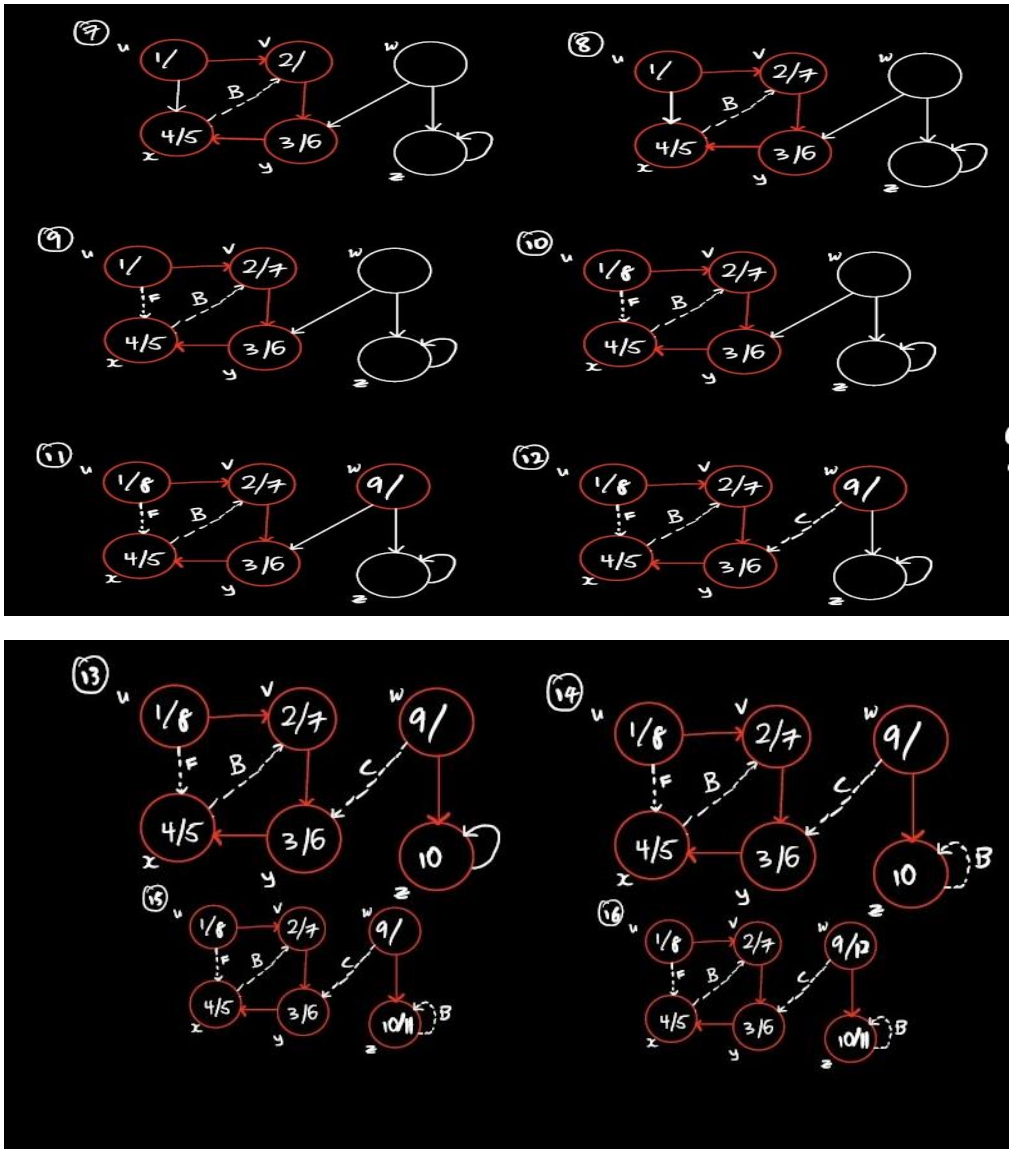


Diagram 9: Depth First Search. The pseudocode below explains the above diagrams.

From the diagrams above and as explained under SCC, each edge is explored where it is highlighted in red if the edges are free, else the edges will be dashed. Within each node, there is a time stamp:

Input: G

DFS(G):

For each vertex $u \in G.V$

u.colour = WHITE

$u.\ \pi$ = NIL

Time = 0

For each vertex $u \in G.V$

```
        If u.colour == WHITE
            DFS-VISIT (G, v)
DFS-VISIT (G, u)
    Time = time + 1    // white vertex u has just been discovered
    u.d = time
    u.colour = highlighted_red
    For each v ∈ G.Adj[u] //explore edge (u, v)
        If  v.colour == WHITE
            v. π = u
            DFS-VISIT (G, v)
    u.colour = RED    // colour u red; it is finished
    Time = time + 1
    u.f = time
```

The above pseudocode of the DFS to explains the sequence of diagrams in the traversing of the nodes above.

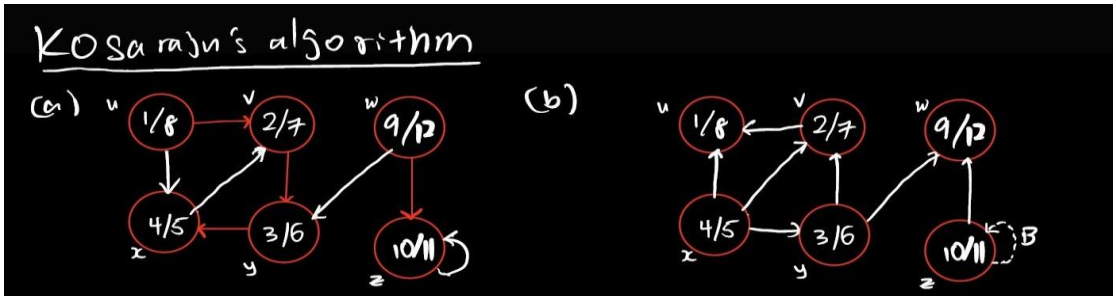


Diagram 10: Kosaraju's algorithm in two steps summarise.

For the Kosaraju's algorithm, part (a) is the result from the DFS, and part (b) is the transpose i.e., shift in the direction of the arrow. After which it identifies the SCC after performing the 2nd round of the DFS. This is done through recursion.

Note: Although the stack above is not drawn, the nodes are slowly added to the stack when explored and removed from the stack when visited to check for strongly connected components.

Analysis and explanation of the code:

1. Adding the edge (u, v) to the directed graph. Finding the dictionary entry for node u (O (1) time complexity) and appending node v gives a complexity of O (1). This is explained by the cost model of the python code (Week 1 L01.01) (Refer to code where class Graph)
2. Performance of the depth first search in the directed graph gives time complexity of O (|V| + |E|)
3. Performing another round of Depth first search gives the time complexity of O (|V| +|E|) on a directed graph.
4. Time to create the transpose graph is O(V+E) (Refer to code where transpose function is located) because you need to look at each vertex and note its neighbours.
5. Calling recursively for adjacent vertices, add the vertex to the stack. O(E)

Based on the analysis above, we can therefore say that the Kosaraju algorithm has the overall time complexity of O(V+E).

Why the Kosaraju cannot solve the 3-SAT

As compared to 2-SAT problems, in 3-SAT problems, each clause has 3 literals. Therefore, when we convert a literal (A, B or C) into an implication, either one of the implications can be not(A or B) implies C. This results in the drawn implication graph to be not(A or B) leading to C, where the not(A or B) further expands into 2 other implications as not(*not A implies B)* and not(*not B implies A)*. For which A and B might also be respectively connected to other variables through other respective clauses. Such multiple clauses therefore lead us to have multiple implication graphs that are interconnected yet not be merged. In addition, every literal would be linked to an exponential number of various other literals as all the graphs are transverse.

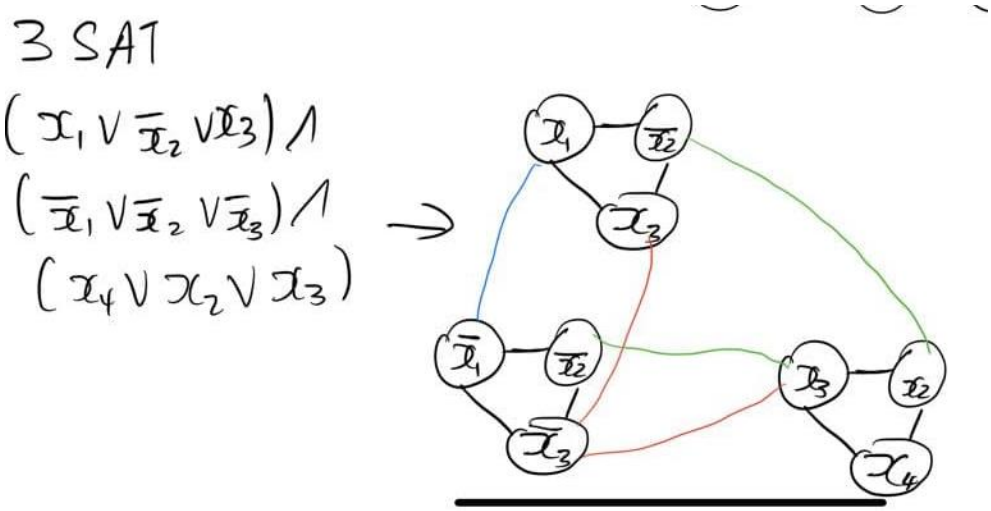


Diagram 11: Explanation of the 3-SAT using implication graph

By proceeding with the linear depth first searches which we had used in the 2-SAT problems will therefore be impossible to be used for 3-SAT problems as all the vertices are not linearly connected.

Part 2

Analysis

The time complexity for graph traversal of the whole implication graph is O(n^2) which is in polynomial time, the reason is because:

Referring to our java random algorithm code, the nested for loops allows for checking through n^2 clauses. Considering that the worst-case scenario n^2 clauses (n literals, max total number of clauses, due to unique combinations of the 2 literals for 2SAT).

T(n) = n^2 + n^2 + n^2 + n^2 + ... checking through the whole of n^2 clauses each time. As n tends to a large value/number, changing the truth value of each of the n literals will cause time complexity to increase at most as fast as n to the power 2. When we traverse our graph, each step is executed in complexity O(n^2) time, given n^2 clauses and n number of variables.

Is the Randomised algorithm more practical than a deterministic one?

The randomizing algorithms are non-deterministic because the order of execution or the result of the algorithm may vary for each run of the algorithm on the same input.

Depending on the cnf input, for some problems, the best known randomized algorithms are faster than the best known deterministic algorithms. This is achieved by requiring that the correct answer be found only with high probability or that the algorithm should run in expected polynomial time. This means that the randomized algorithm may not find a correct answer in some cases or may take very long time. The deterministic algorithm might be slightly more practical and efficient than the random one, especially for larger input sizes of n. This is because they have faster run time compared to the random counterparts. In addition, deterministic algorithms are generally also comparatively conceptually simpler and easy to implement which has its own practical uses.

However in reality, applied randomizing algorithms might not actually be entirely deterministic or non-deterministic, since they are deterministic for some input cases and non-deterministic for other cases, but overall, still considered to be non-deterministic. For example, non-deterministic algorithms for inputs of arbitrary length may work in a deterministic manner given an empty input string.

Considering a modern SAT (satisfiability) problem, decisions are seemingly non-deterministic and they are backtracked whenever needed. In order to make the algorithm efficient, the principles of clause learning and unit propagation are applied, which reduce the number of decisions that the algorithm needs to make. The concept of unit propagation itself offers simpler input instances where no non-deterministic decision needs to be carried out. The SAT

problem is an example of a non-deterministic searching algorithm with unit propagation executed on a deterministic machine by performing decision making and back-tracking.

General Idea of Random algorithm (concept/ strategy)

Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied, or you get tired of flipping coins.

Further research into Random Algorithms as a form of probabilistic Turing Machine with a time complexity class of BPP (bounded-error probabilistic polynomial time).

https://docs.google.com/document/d/1eaGv0gLEvy2iqT5baBYoXBvZPhIF6Cjq27AF3TJKWr0/edit?usp=sharing

References:

1. Chapter 22 CLRS Section 22.5 (Introduction to Algorithms, 3rd Edition pg.616)

Rando Algo

http://people.seas.harvard.edu/~cs125/fall14/section-notes/sec12.pdf

http://people.seas.harvard.edu/~cs125/fall16/lec20.pdf

http://people.seas.harvard.edu/~cs125/fall14/lec19.pdf

http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf

https://www.cs.yale.edu/homes/aspnes/pinewiki/RandomizedAlgorithms.html

Deterministic

https://en.wikipedia.org/wiki/Deterministic_algorithm

https://cs.stackexchange.com/questions/30160/algorithms-which-are-both-deterministic-and-non-deterministic

http://www.mi.fu-berlin.de/wiki/pub/ABI/DiscretMathWS10/runtime.pdf