ALGORITHM ANALYSIS AND DESIGN

Module 2 Part 2

CST306

GRAPH TRAVERSAL ALGORITHMS:

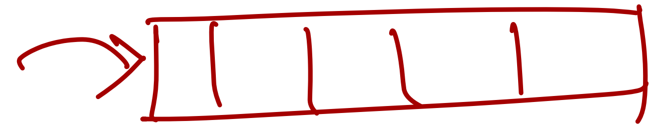
- Graph traversal algorithms visit the vertices of a graph, according to some strategy.

  Different graph traversal algorithms are:

  - Breadth First Search(BFS)
  - Depth First Search(DFS)

BREADTH FIRST SEARCH(BFS)

Q (Queue)



Algorithm BFS(G,u)

1. Set all nodes are unvisited
2. Mark the starting vertex u as visited and put it into an empty Queue Q
3. While Q is not empty

   3.1 Dequeue v from Q

   3.2 While v has an unvisited neighbor w

3.2.1 Mark w as visited

3.2.2 Enqueue w into Q

4. If there is any unvisited node x

   4.1 Visit x and Insert it into Q. Goto step 3

COMPLEXITY

O(V) Visit E O(E)

Visit

- If the graph is represented as an adjacency list
  - Each vertex is enqueued & dequeued atmost once. Each queue operation take O(1) time. So the time devoted to the queue operation is $\boxed{O(V)}$.
  - The adjacency list of each vertex is scanned only when the vertex is dequeued. Each adjacency list is scanned atmost once. Sum of the lengths of all adjacency list is |E|. Total time spend in scanning adjacency list is O(E).
  - Time complexity of BFS = O(V) + O(E) = $\boxed{O(V+ E)}$.

$O(V^2)$

- In a dense graph: $O(V) + O(E)$.

  $E = O(V^2)$

  Time complexity $= O(V) + O(V^2) = O(V^2)$

- If the graph is represented as an adjacency matrix
  - There are V^2 entries in the adjacency matrix. Each entry is checked once.
  - Time complexity of BFS $= O(V^2)$

$\boxed{T(BFS) = O(V^2)}$

APPLICATIONS OF BFS

- Finding shortest path between 2 nodes u and v, with path length measured by number of edges
- Testing graph for bipartiteness
- Minimum spanning tree for unweighted graph
- Finding nodes in any connected component of a graph
- Serialization/deserialization of a binary tree
- Finding nodes in any connected component of a graph

DEPTH FIRST SEARCH(DFS)

Algorithm DFS(G, u)

1. Mark vertex u as visited
2. For each adjacent vertex v of u
   - 2.1 if v is not visited
     - 2.1.1 DFS(G,v)

Algorithm main(G,u)

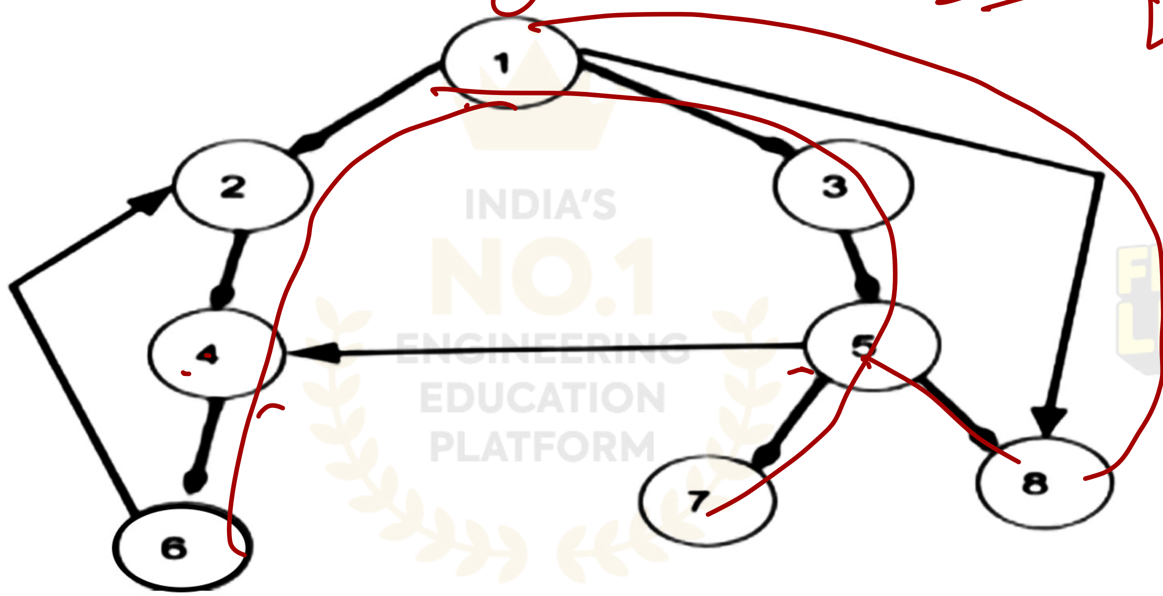
1. Set all nodes are unvisited.
2. DFS(G, u) visit
3. For any node x which is not yet visited
   3.1 DFS(G, x)

COMPLEXITY

- If the graph is represented as an adjacency list
  - Each vertex is visited atmost once. So the time devoted is O(V)
  - Each adjacency list is scanned atmost once. So the time devoted is O(E)
  - Time complexity of DFS = O(V + E).
- If the graph is represented as an adjacency matrix
  - There are V^2 entries in the adjacency matrix. Each entry is checked once.
  - Time complexity of DFS = $O(V^2)$

$\boxed{T(DFS) = O(V^2)}$

CLASSIFICATION OF EDGES



The DFS traversal of the above graph is 1 2 4 6 3 5 7 8

- Tree Edge: It is a edge in tree obtained after applying DFS on the graph

  Eg:-(1,2) ,(2,4),(4,6),(1,3),(3,5),(5,7) and (5,8)

- Forward Edge:

  It is an edge (u, v) such that v is descendant but not part of the DFS tree eg:-(~~6,2~~) (1, 8) .

- Backward Edge

  It is an edge (u, v) such that v is ancestor of edge u but not part of DFS tree.

  Eg: (6,2)

  2 was visited before 6.

  ∴ 2 is an ancestor of 6.

- Cross Edge

It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them.

Eg: (5,4)

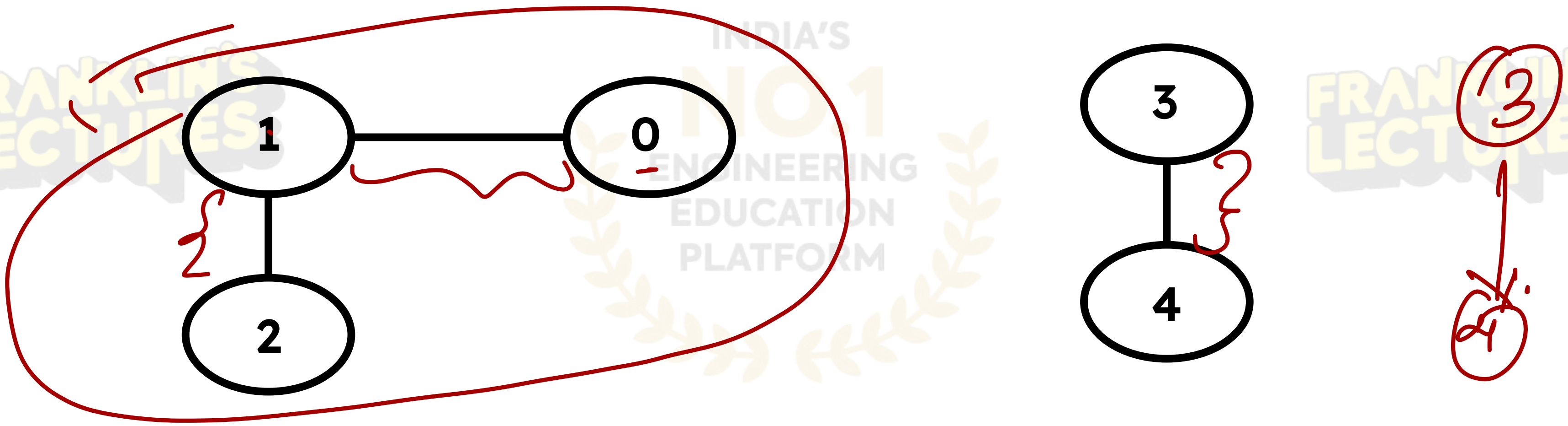
APPLICATIONS OF DFS

- Finding connected components in a graph
- Topological sorting in a DAG
- Scheduling problems
- Cycle detection in graphs
- Finding 2-(edge or vertex)-connected components
- Finding 3-(edge or vertex)-connected components
- Finding the bridges of a graph
- Finding strongly connected components

- Solving puzzles with only one solution, such as mazes
- Finding biconnectivity in graphs

Connected Components:

- Connected component of a graph G is a connected subgraph of G of maximum size
- A graph may have more than one connected components.



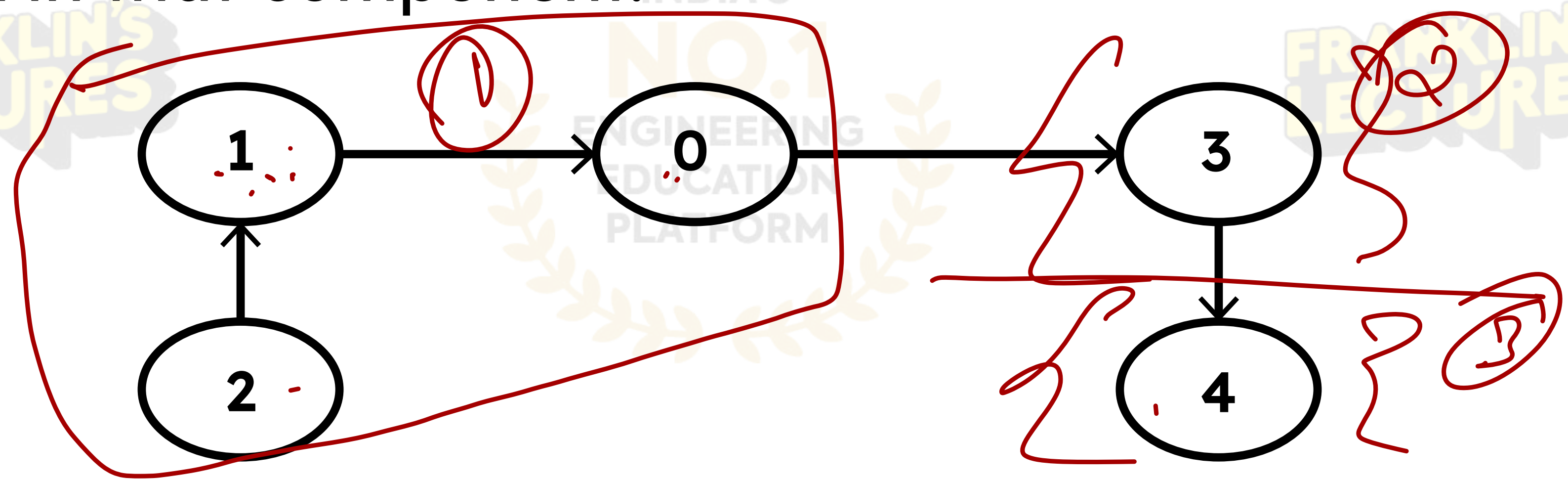
There are two connected components in above undirected graph

0 1 2

3 4

Strongly Connected Components(SCC)

- Strong Connectivity applies only to directed graphs.
- A strongly connected component of a directed graph is a compliment such that all the vertices in that component is reachable from every other vertex in that component.



Here we have 3 SCCs: {0,1,2}, {3}, {4}

THANK YOU