Department of Computer Science & Engineering

ASSINGMENT-01

Topic

GRAPH QUEST:

Mapping, Representation, Traversal, Connectivity.

Course Title: **Algorithm**
Course Code: **CSE246**
Secction: **02**

Submitted By

Abrar Khatib Lajim
2022-3-60-043

Submitted To

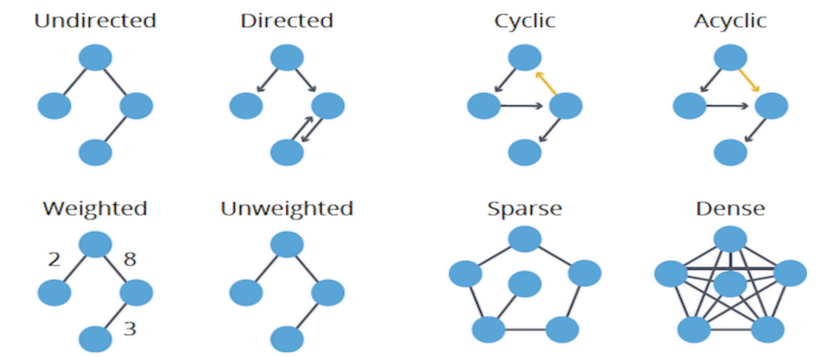
Dr. Tania Sultana
Assistant Professor
Department of CSE

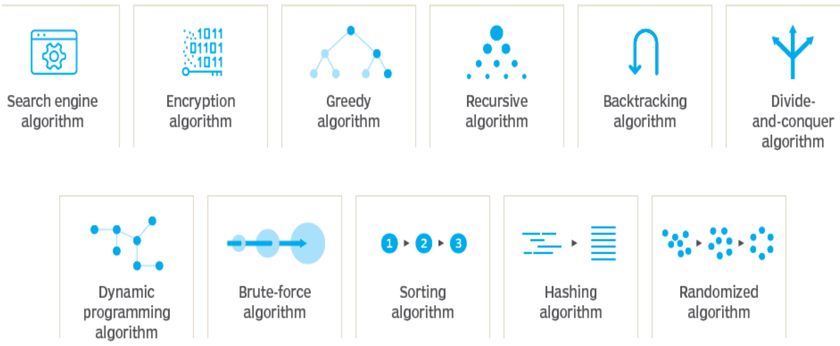
Date of Submission:
18/11/24

Introduction

This document explains the implementation and theory behind fundamental graph algorithms, focusing on concepts like graph representation, traversal methods, and advanced techniques such as topological sorting, articulation points, and strongly connected components (SCC).

Table of Contents

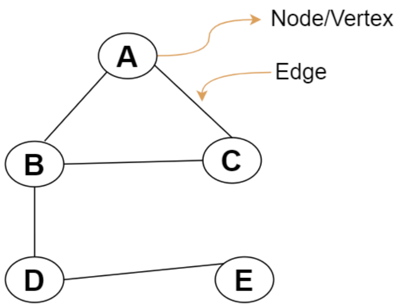




Objective:
To understand and implement foundational graph theory concepts, including graph representations, traversal algorithms (BFS and DFS), topological sorting, and methods to identify critical structures like articulation points and strongly connected components (SCCs).

---

1. Graph Representation

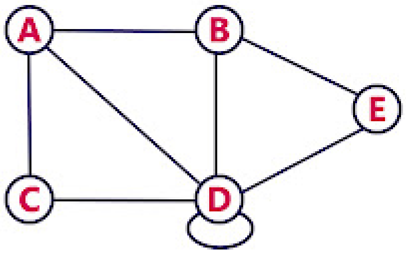


Graphs are fundamental data structures consisting of nodes (vertices) and edges.

They can be represented in multiple ways:

1. Sequential representation (or, **Adjacency matrix** representation)
2. Linked list representation (or, **Adjacency list** representation)

Graph: An undirected graph is represented as,



Nodes: **A, B, C, D, E**

Edges: The graph is undirected, meaning connections go both ways.

A is connected to **B, C, D**

B is connected to **A, D, E**

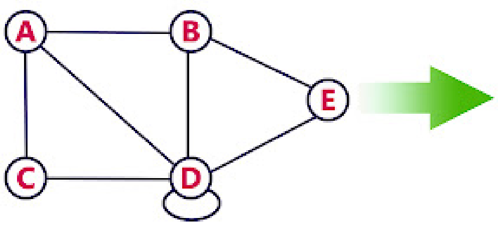
C is connected to **A, D**

D is connected to **A, B, C, E**, and has a **self-loop** (connected to itself)

E is connected to **B, D**

Adjacency Matrix

- The matrix is **symmetric** for an **undirected graph**, meaning**: Matrix[u][v]=Matrix[v][u]**



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

*The matrix shows **1** for connected nodes and **0** for no connection.* Diagonal entries are **0**, except for **D**, which has a self-loop.

- ➔ An adjacency matrix is a 2D array (matrix) used to represent a graph.
- ➔ The rows and columns of the matrix correspond to the nodes (vertices) of the graph.
- ➔ Each cell in the matrix indicates whether there is an edge between two nodes.

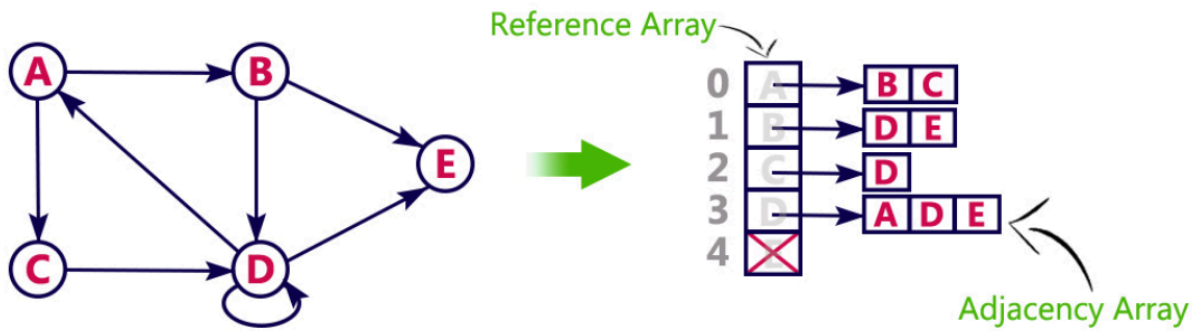
Matrix[A][B] = 1 → There is an edge between **A** and **B**.

Matrix[E][C] = 0 → No edge between **E** and **C**.

Matrix[D][D] = 1 → Self-loop on **D**.

Adjacency List

- A directed graph is represented as an array of lists.



- Each list contains all nodes connected to a vertex.
- Nodes (A, B, C, D, E) are connected by directed edges (arrows).
- Each edge points from one node to another.

Graph (on the left):

- Nodes: **A, B, C, D, E**
- Edges:
  - (Node A points to B and C) **A → C, A → B -**
  - (Node B points to D & E) **B → D, B → E**
  - (Node C points to D) **C → D**
  - (Node D points to A and E) **D → A, D → E, D → D** (self-loop)
  - (**E** has no outgoing edges) **E → ' X '** (indicated by an "X").

Comparison with Adjacency Matrix:

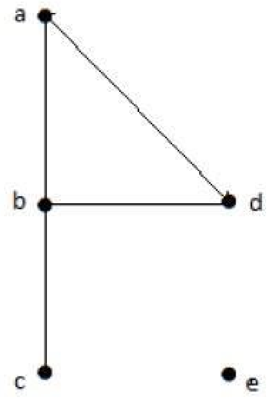
Feature	Adjacency List	Adjacency Matrix
Space Complexity	$O(V+E)$ (efficient for sparse)	$O(V^2)$ (efficient for dense)
Edge Lookup	$O(d)$ (degree of vertex)	$O(1)$
Neighbor Traversal	$O(d)$	$O(V)$
Dynamic Graphs	Easy to modify	Difficult to resize
Best For	Sparse graphs	Dense graphs

Degree

- **In-degree**: Number of edges pointing to a vertex.
  Notation – $\deg^{-}(V)$.

- **Out-degree**: Number of edges going out from a vertex.
  Notation – $\deg^{+}(V)$.

In the Right side Undirected Graph,



deg(a) = 2, as there are 2 edges meeting at vertex 'a'.

deg(b) = 3, as there are 3 edges meeting at vertex 'b'.

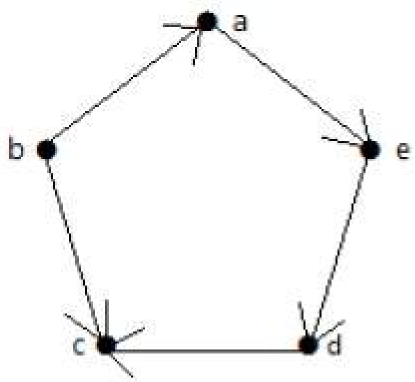
deg(c) = 1, as there is 1 edge formed at vertex 'c'

So 'c' is a **pendant vertex/**degree of 1.

deg(d) = 2, as there are 2 edges meeting at vertex 'd'.

deg(e) = 0, as there are 0 edges formed at vertex 'e'.

So 'e' is an **isolated vertex**.



Here is a directed graph. Vertex 'a' has an edge 'ae' going outwards from vertex 'a'. Hence its outdegree is 1. Similarly, the graph has an edge 'ba' coming towards vertex 'a'. Hence the indegree of 'a' is 1.

The indegree and outdegree of other vertices are shown in the following table –

Vertex	Indegree	Outdegree
a	1	1
b	0	2
c	2	0
d	1	1
e	1	1

Pseudocodes

Degree of a Node

Undirected Graph

```
For each node in the graph:

    Degree[node] = Number of neighbors in adjacency list
```

Directed Graph

```
For each node in the graph:

    In-Degree[node] = Count of edges pointing to the node

    Out-Degree[node] = Count of edges going out from the node
```

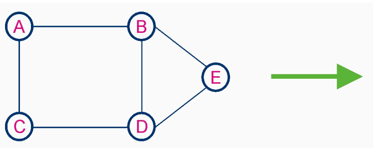
Adjacency Matrix

```
Input: Number of nodes (n), List of edges (edges)

Output: Adjacency matrix (matrix)

1. Create an n x n matrix filled with 0

2. For each edge (u, v) in edges:

      a. Set matrix[u][v] = 1

      b. If the graph is undirected, also set matrix[v][u] = 1

3. Return the matrix
```

Adjacency NodeList

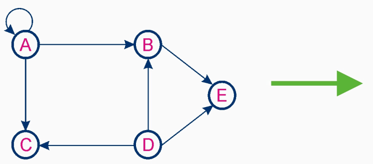
```
function CreateAdjacencyList(n, edges):

    // Initialize an empty adjacency list with n nodes

    adjacencyList ← empty list of lists of size n

    for each edge (u, v) in edges:

        append v to adjacencyList[u]

    return adjacencyList
```

More Examples:



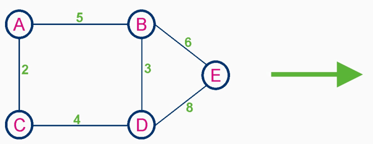
Undirected Graph

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	1
C	1	0	0	1	0
D	0	1	1	0	1
E	0	1	0	1	0



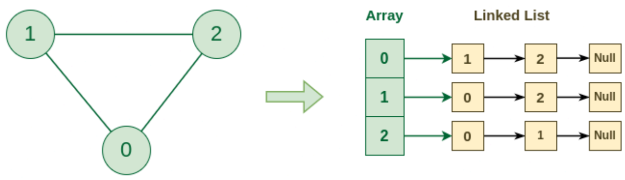
Directed Graph

	A	B	C	D	E
A	1	1	1	0	0
B	0	0	0	0	1
C	0	0	0	0	0
D	0	1	1	0	1
E	0	0	0	0	0



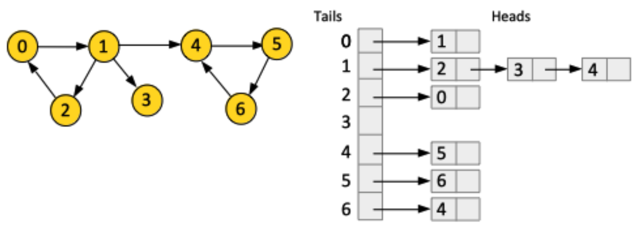
Weighted Graph

	A	B	C	D	E
A	0	5	2	0	0
B	5	0	0	3	6
C	2	0	0	4	0
D	0	3	4	0	8
E	0	6	0	8	0



Undirected Graph

Adjacency List



Reference:

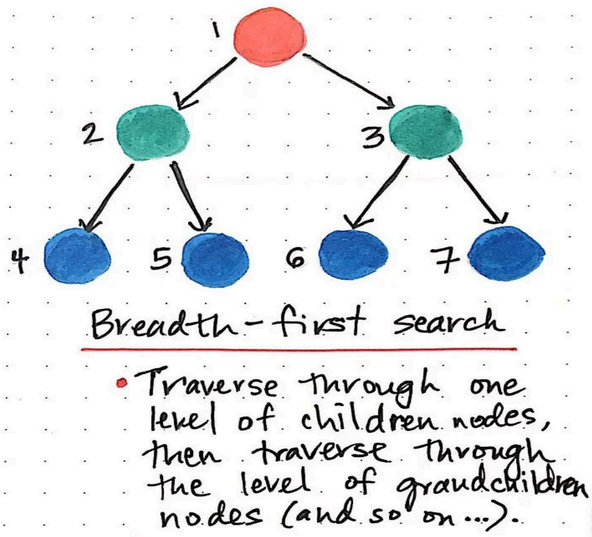
https://algodaily.com/lessons/implementing-graphs-edge-list-adjacency-list-adjacency-matrix

https://www.tutorialspoint.com/degree-of-vertex-of-a-graph

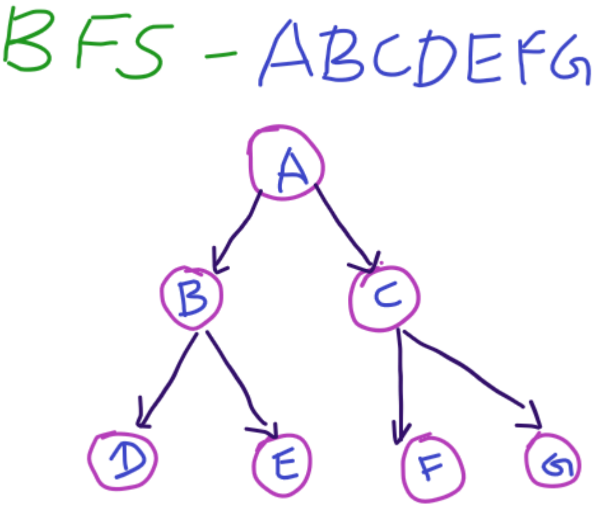
http://www.btechsmartclass.com/data_structures/graph-representations.html

---

2. Breadth-First Search (BFS)



Description:
BFS is a traversal technique used to explore nodes layer by layer. It's especially useful for finding the shortest path in unweighted graphs.



The root node is A. Its left and right children are B and C. Further its children are D,E,F and G. So the BFS of the tree is `ABCDEFG`

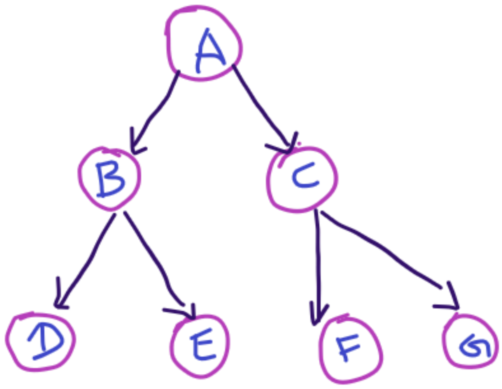
For better visualization :
https://www.cs.usfca.edu/~galles/visualization/BFS.html

https://visualgo.net/en

Steps of BFS in Code:

1. **Initialize**:
   - Create a **queue** and add the starting node.
   - Maintain a **visited list** to keep track of nodes that have been visited.
2. **Visit Nodes**:
   - Dequeue a node, mark it as visited, and process it.
   - Add all its unvisited neighbors to the queue.
3. **Repeat**:
   - Repeat until the queue is empty.

In this tree,



1. We first put the root node A in the queue. A's children are B and C. Add them to the queue and remove A.
2. Further pull B, add its children D and E to queue and remove B.
3. Pull C, add its children F and G to queue and remove C.
4. D,E,F and G have no children, so pop them from the queue.

Code Sample:

```
public void BFS(TreeNode root)
    {
        Queue<Integer> queue = new LinkedList<>();
         queue.add(root);
        while (!queue.isEmpty())
        {
            int size = queue.size();
             for (int i=0;i<size;i++)
                  {
                    TreeNode currentNode = queue.poll();
                if(currentNode.left!=null)   queue.add(currentNode.left);
                    if(currentNode.right!=null)
queue.add(currentNode.right);
                    System.out.print(currentNode.val);
     } } }
```

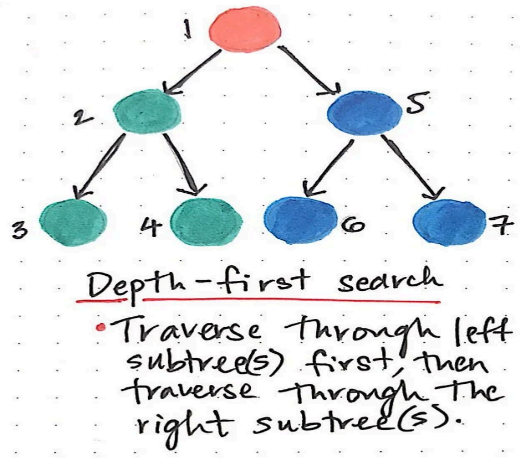
Reference:

https://medium.com/basecs/breaking-down-breadth-first-search-cebe696709d9

https://leetcode.com/discuss/study-guide/1072548/A-Beginners-guid-to-BFS-and-DFS

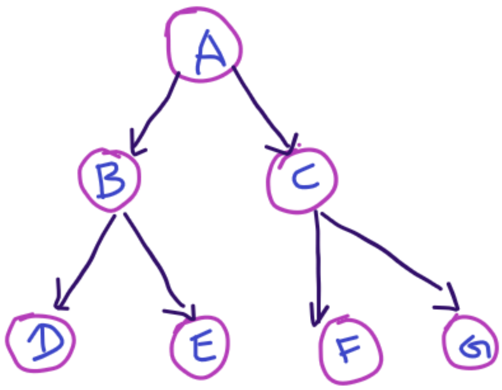
https://github.com/AbrarBb/DSA

3: Depth-First Search (DFS)

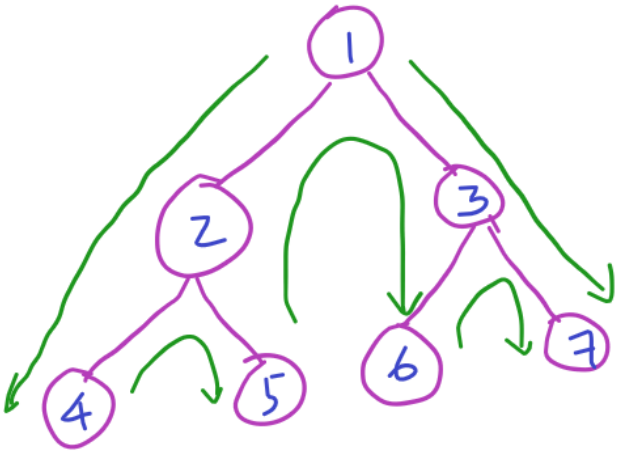


Description:
DFS (Depth-First Search) is a graph traversal algorithm that explores as far as possible along one branch before backtracking. It's widely used in scenarios where exploring all possible paths or structures is essential, such as solving mazes, detecting cycles, or finding connected components.

As BFS traverses wide, DFS goes deep. It starts from the first node, and goes deep in a path till it traverses the leaf node/the last node before starting traversing its next path.



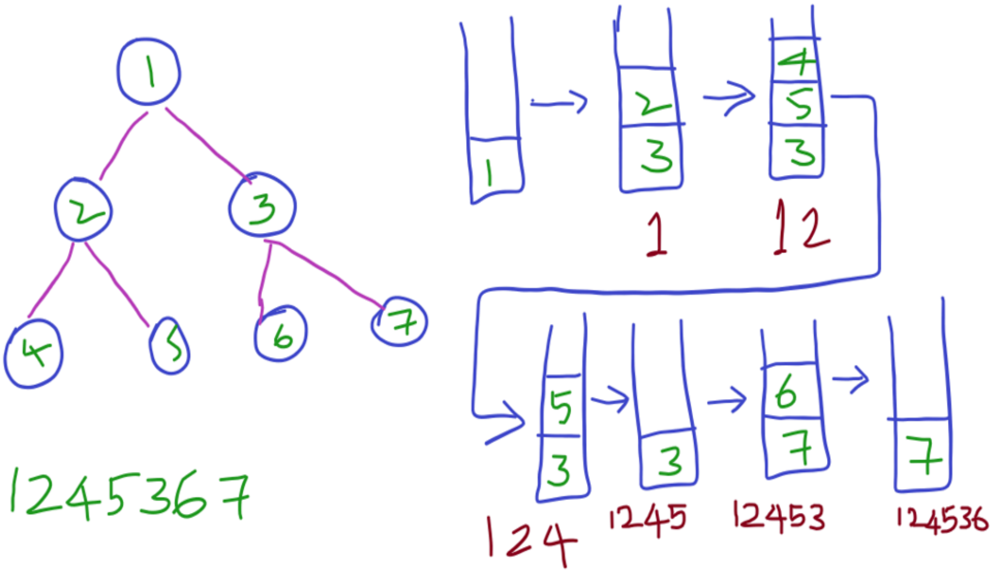
DFS – ABDECFG



In left tree, it first starts from root node 1, does deep to 2 and more deeper to 4. Node 4 has no children, so it is done with that path, visits 5 and traverse back to 1 and starts visiting deep 3,6 and 7. **DFS- 1245367**

In Right side example,

1. First we are visiting Node 1, add them to stack, explore its children and add its right child 3 first and then left child 2.
2. Now the stack has nodes 2 and 3. Explore 2's children and add its right child 5 and then its left child 4 to stack. And pop 2.
3. Further pop nodes from stack and add its children until stack becomes empty.



For Better Visualization:

https://www.cs.usfca.edu/~galles/visualization/DFS.html

https://visualgo.net/en/dfsbfs

Pseudo Code Sample:

```
DFS(node):
    Mark node as visited
    Process the node (if needed)
    For each neighbor of node:
        If the neighbor is not visited:
            Call DFS(neighbor)
```

Steps:

1. Initialize a stack and push the starting node.
2. While the stack is not empty:
   - Pop the top node.
   - If the node hasn't been visited:
     - Mark it as visited and process it.
     - Push all its unvisited neighbors onto the stack.

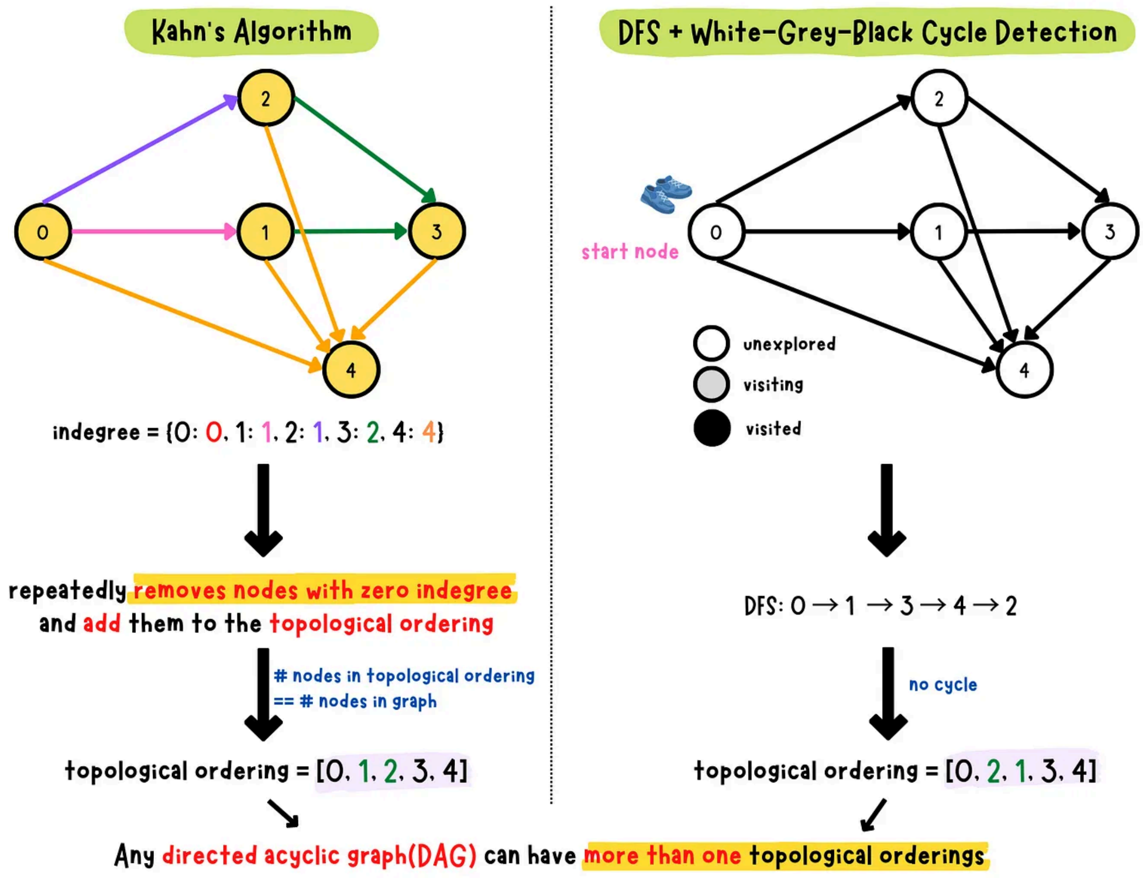
```
public void DFSInOrder(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    TreeNode current = root;
    stack.push(root);
    while(!stack.isEmpty()) {
        while(current.left != null) {
            current = current.left;
            stack.push(current);   }
        current = stack.pop();
        System.out.print(current.value);
        if(current.right != null) {
            current = current.right;
            stack.push(current);
        }   } }
```

Feature	BFS (Breadth-First Search)	DFS (Depth-First Search)
Data Structure	Queue (FIFO)	Stack (LIFO) or Recursion
Traversal Pattern	Explores all neighbors before going deeper	Explores as deep as possible, then backtracks
Use Case	Shortest path in unweighted graphs	Pathfinding, cycle detection, and maze solving
Memory Usage	High (stores all neighbors in a queue)	Low (only stores current path)
Pathfinding	Finds shortest path in unweighted graphs	Does not guarantee shortest path
Search Completeness	Guaranteed if the graph is finite	May not terminate on infinite graphs
Speed	Slower for large, dense graphs	Faster for paths in large, dense graphs
Cycle Detection	Not ideal for detecting cycles	Effective for detecting cycles
Works Well For	Finding shortest paths, level-order traversal	Topological sorting, solving puzzles
Graph Type	Best for unweighted graphs	Works for all graph types

---

4: Topological Sort

Description:
Topological sorting arranges nodes in a Directed Acyclic Graph (DAG) such that for every directed edge u → v, node u comes before v. It's critical in scheduling and dependency management.



Algorithms:

➔ **DFS-Based Topological Sort**:
- Perform a **DFS** on the graph.
- Push each vertex onto a stack **after visiting all its neighbors** (post-order traversal).
- Pop all nodes from the stack for the sorted order.

➔ **Kahn's Algorithm (Indegree-Based)**:
- Count the **indegree** of each vertex.
- Start with nodes having indegree = 0 (nodes with no dependencies).
- Remove these nodes and update the indegrees of their neighbors.
- Continue until all nodes are processed.

DFS-Based Algorithm Pseudocode:

```
TopologicalSortDFS(graph):
  Create an empty stack
  Create a visited set
  For each vertex in graph:
    If vertex is not visited:
      DFS(vertex, stack, visited)
  Return stack in reverse order

DFS(vertex, stack, visited):
  Mark vertex as visited
  For each neighbor in graph[vertex]:
    If neighbor is not visited:
      DFS(neighbor, stack, visited)
  Push vertex onto the stack
```

Kahn's Algorithm Pseudocode:

```
TopologicalSortKahn(graph):
  Create a list to store sorted order
  Create a queue for nodes with indegree = 0
  Compute indegree of each vertex
  Add all vertices with indegree = 0 to the queue

  While the queue is not empty:
    Remove a vertex from the queue
    Add it to the sorted order
    For each neighbor of the vertex:
      Decrease indegree of the neighbor
      If indegree becomes 0, add the neighbor to the queue

  If sorted order contains all vertices:
    Return sorted order
  Else:
   Return "Graph has a cycle"   //If the graph has a cycle, topological sorting is not possible.
```

Part 5: Articulation Points

Description:
Articulation points (or cut vertices) in a graph are vertices that, if removed, cause the graph to become disconnected. These points are critical for ensuring the resilience of networks, such as telecommunications or transportation systems. Identifying these points can reveal vulnerabilities in a network.

In a graph, a vertex is called an articulation point if removing it and all the edges associated with it results in the increase of the number of connected components in the graph. For example, consider the graph given in the following figure.

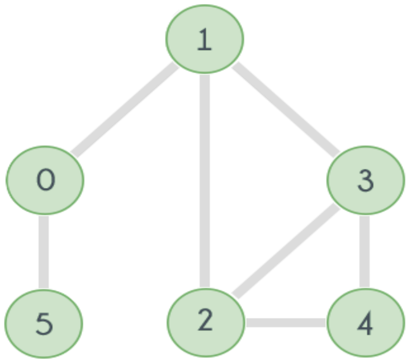


Fig. 1

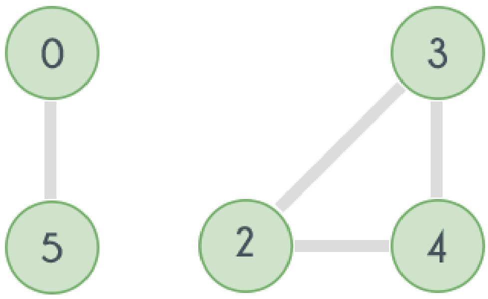


Fig. 2

If in the above graph, vertex 1 and all the edges associated with it, i.e. the edges 1-0, 1-2 and 1-3 are removed, there will be no path to reach any of the vertices 2, 3 or 4 from the vertices 0 and 5, that means the graph will split into two separate components. One consisting of the vertices 0 and 5 and another one consisting of the vertices 2, 3 and 4 as shown in the following figure.

removing the vertex 0 will disconnect the vertex 5 from all other vertices. Hence the given graph has two articulation points: 0 and 1.Articulation Points represent vulnerabilities in a network. In order to find all the articulation points in a given graph, the brute force approach is to check for every vertex if it is an articulation point or not, by removing it and then counting the number of connected components in the graph. If the number of components increases then the vertex under consideration is an articulation point otherwise not.

Algorithm to Find Articulation Points Using DFS

Articulation points can be found using **DFS** and by tracking:

1. **Discovery Time (disc)**: The time when a node is first visited.

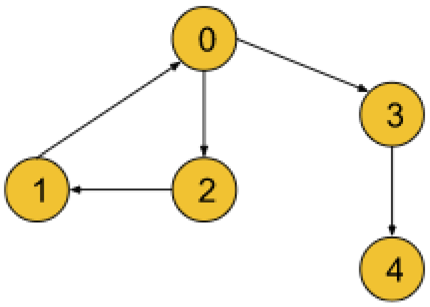
2. **Lowest Reachable Vertex (low)**: The lowest discovery time reachable from a node through its subtree.
3. **Parent**: The node's parent in the DFS tree.

```
FindArticulationPoints(Graph):
  Initialize:
    visited[] = False for all vertices
    disc[] = -1 for all vertices (discovery time)
    low[] = -1 for all vertices (lowest reachable vertex)
    parent[] = None for all vertices
    ap[] = False for all vertices (to mark articulation points)
  time = 0

  For each vertex u in Graph:
    If u is not visited:
      DFS(u)
  Return all vertices where ap[u] is True

DFS(u):
  Mark u as visited
  Set disc[u] = low[u] = time (increment time)
  children = 0
  For each neighbor v of u:
    If v is not visited:
      children += 1
      Set parent[v] = u
      DFS(v)
      Update low[u] = min(low[u], low[v])
      If parent[u] is None and children > 1:
        Mark u as an articulation point
      If parent[u] is not None and low[v] >= disc[u]:
        Mark u as an articulation point
    Else if v is not the parent of u:
      Update low[u] = min(low[u], disc[v])
```

Part 6: Strongly Connected Components (SCC)

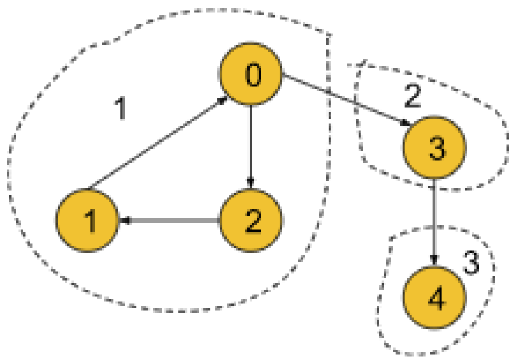
Description:
In directed graphs, SCCs are subgraphs where every vertex can reach every other vertex within the component. Detecting SCCs is essential for understanding cycles, dependencies, and structural properties.

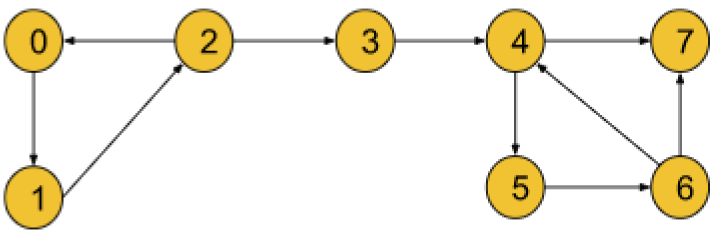
A component is called a Strongly Connected Component(SCC) only if for every possible pair of vertices (u, v) inside that component, u is reachable from v and v is reachable from u.



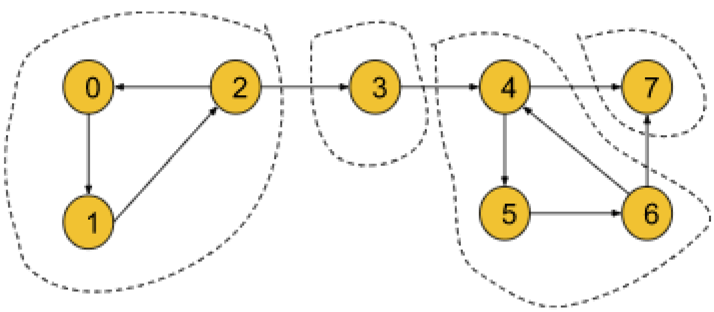
In the following directed graph, the SCCs have been marked:

Three strongly connected components are marked :





If we take 1st SCC in the above graph, we can observe that each node is reachable from any of the other nodes. For example, if we take the pair (0, 1) from the 1st SCC, we can see that 0 is reachable from 1 and 1 is also reachable from 0. Similarly, this is true for all other pairs of nodes in the SCC like (0,2), and (1,2). But if we take node 3 with the component, we can notice that for pair (2,3) 3 is reachable from 3 but 2 is not reachable from 3. So, the first SCC only includes vertices 0, 1, and 2.



By definition, a component containing a single vertex is always a strongly connected component. For that vertex 3 in the above graph is itself a strongly connected component.

Algorithm: Kosaraju's Algorithm

Kosaraju's algorithm is a simple and efficient way to find SCCs:

1. **Step 1**: Perform a DFS on the original graph and store vertices in a stack based on their **finish time**.
2. **Step 2**: Transpose (reverse) the graph.
3. **Step 3**: Perform DFS on the transposed graph in the order of the stack, grouping nodes into SCCs.

Code Sample:

```
FindSCCs(Graph):

  stack = Empty

  visited[] = False for all vertices

  # Step 1: Perform DFS and store nodes by finish time

  For each vertex u in Graph:

    If u is not visited:

      FillStack(u)

  # Step 2: Transpose the graph

  TransposedGraph = ReverseEdges(Graph)

  # Step 3: Process nodes in stack order on TransposedGraph

  visited[] = False for all vertices

  SCCs = Empty


  While stack is not empty:

    u = stack.pop()

    If u is not visited:

      component = Empty

      DFS_Transpose(u, component)
```

```
            SCCs.append(component)

        Return SCCs

    FillStack(u):

        Mark u as visited

        For each neighbor v of u:

            If v is not visited:

                FillStack(v)

        Add u to stack

    ReverseEdges(Graph):

        Create TransposedGraph

        For each vertex u in Graph:

            For each neighbor v of u:

                Add edge v → u to TransposedGraph

        Return TransposedGraph

    DFS_Transpose(u, component):

        Mark u as visited

        Add u to component

        For each neighbor v of u in TransposedGraph:

            If v is not visited:

                DFS_Transpose(v, component)
```

Applications

1. **Web Crawling**:
    - Identifies cycles in hyperlink structures to avoid endless loops.
2. **Dependency Resolution**:
    - Clusters strongly related modules or components in software systems.
3. **Network Analysis**:
    - Detects tightly connected subnetworks, useful in social or communication networks.

Final Summary

This assignment covered essential graph theory concepts, from basic graph representation to advanced algorithms like articulation points and strongly connected components (SCC). Key highlights include:

1. **Graph Representation**: Adjacency lists/matrices for modeling graphs.
2. **Traversal Algorithms**: BFS and DFS for pathfinding and connected components.
3. **Articulation Points**: Identified critical nodes for network resilience.
4. **SCC Detection**: Applied Kosaraju's algorithm to analyze dependencies and cycles.

These concepts demonstrated practical applications in **network resilience**, **dependency analysis**, and **pathfinding**, enhancing my understanding of their importance in real-world scenarios.

github.com/AbrarBb/DSA

cs246 Ass01.doc