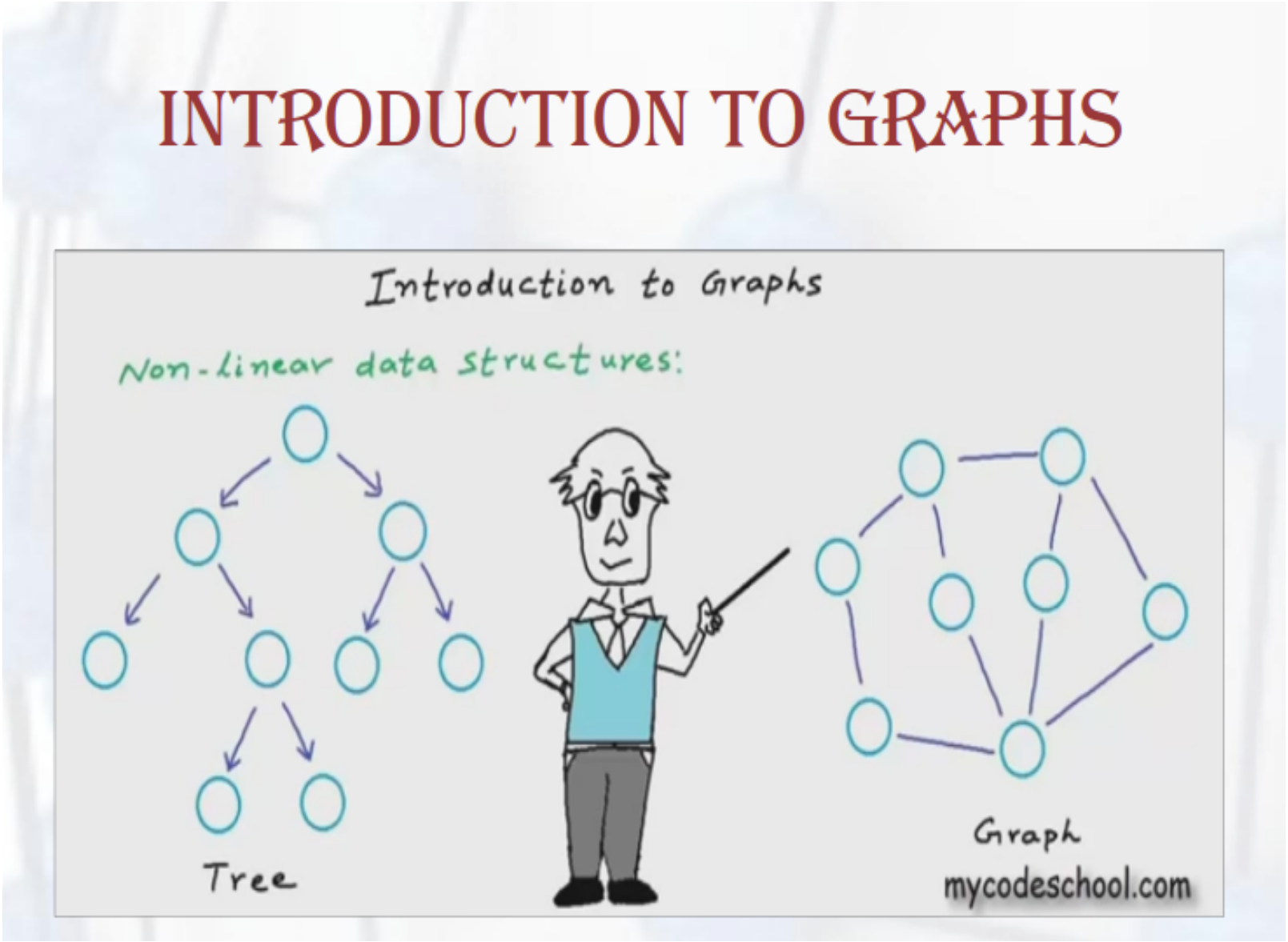
Graph

Manoj Wairiya

Department of Computer Science
MNNIT ALLAHABAD

February 26, 2018

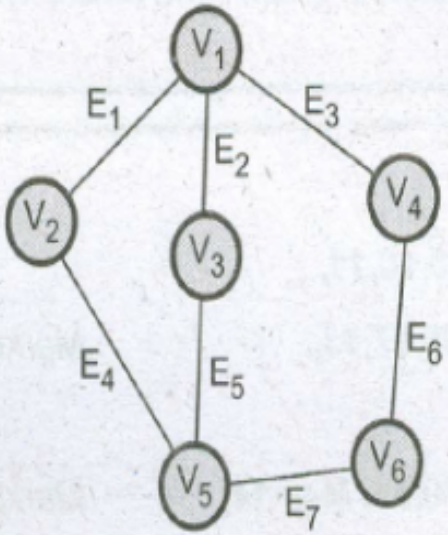
Introduction



Introduction

A graph is a collection of two sets V and E where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

- **Vertices** are nothing but the nodes in the graph.
- Two adjacent vertices are joined by **edges**.
- Any graph is denoted as G = {V, E}.
- **For example :**



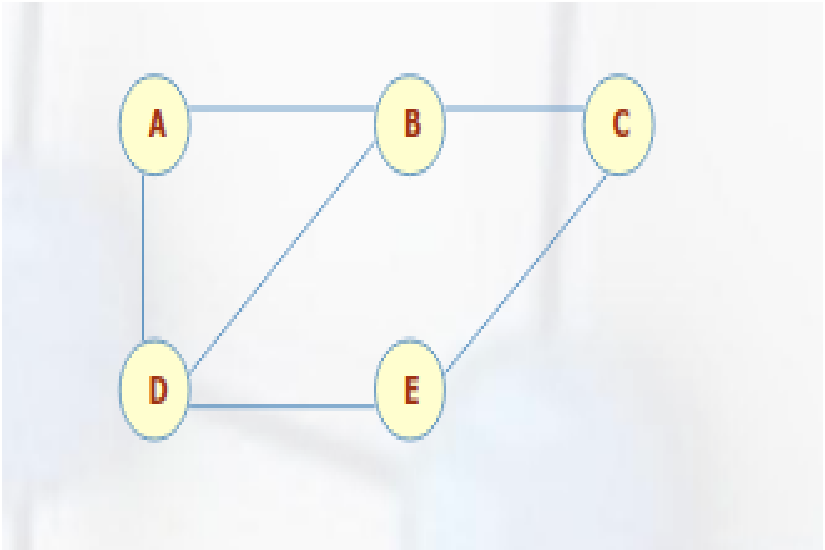
$$G = \{\{V_1, V_2, V_3, V_4, V_5, V_6\}, \{E_1, E_2, E_3, E_4, E_5, E_6, E_7\}\}$$

Introduction

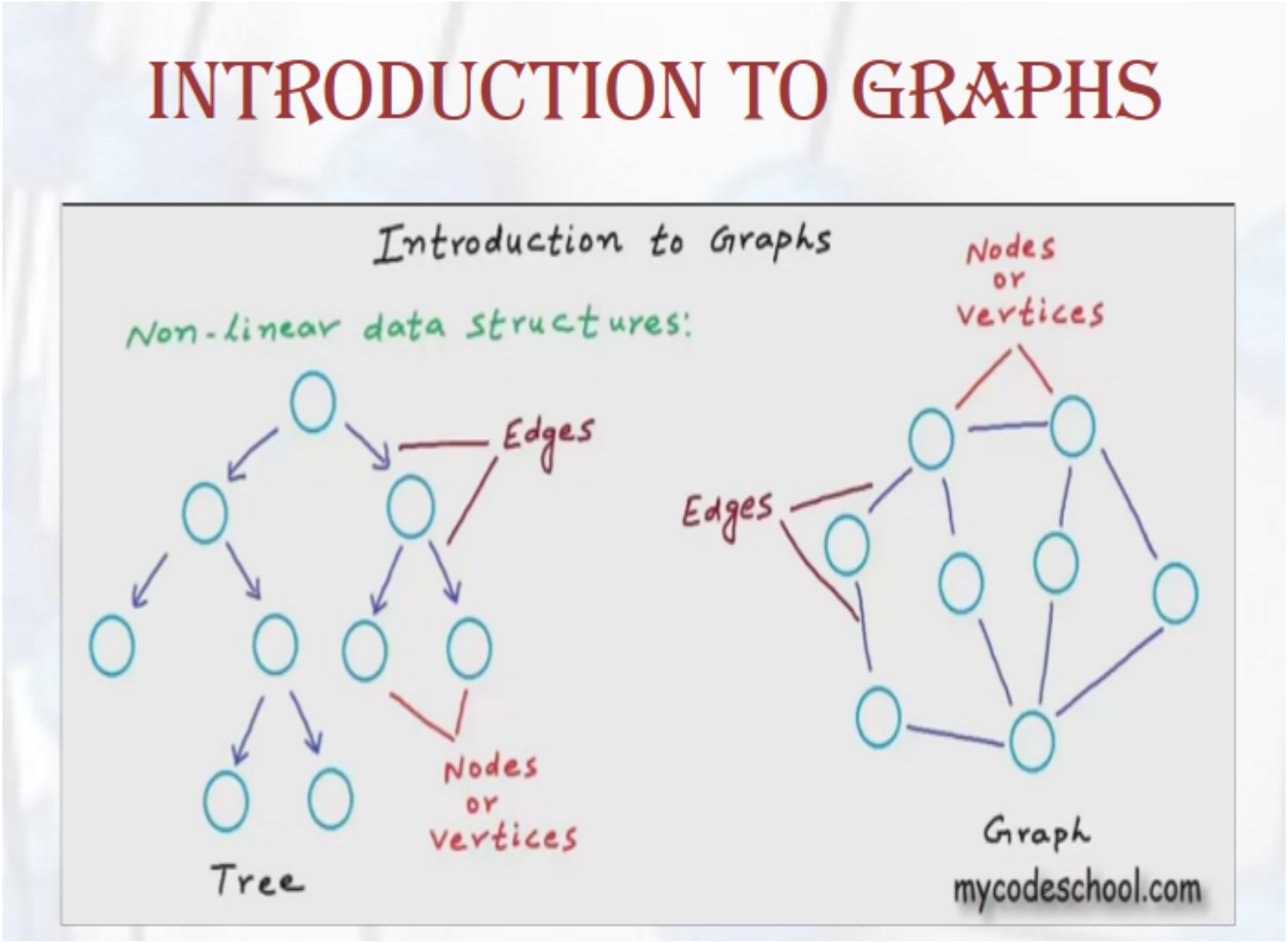
- A graph is an **abstract data structure** that is used to implement the graph concept from mathematics. A graph is basically, a collection of vertices (also called nodes) and edges that connect these vertices. **A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can be represented.**
  WHY GRAPHS ARE USEFUL?
- Graphs are widely used to model any situation where entities or things are related to each other in pairs; for example, the following information can be represented by graphs:
- *Family trees* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Introduction

- A graph G is defined as an ordered set (V, E), where V(G) represent the set of vertices and E(G) represents the edges that connect the vertices.
- The figure given shows a graph with V(G) = A, B, C, D and E and E(G) = (A, B), (B, C), (A, D), (B, D), (D, E), (C, E) . Note that there are 5 vertices or nodes and 6 edges in the graph.

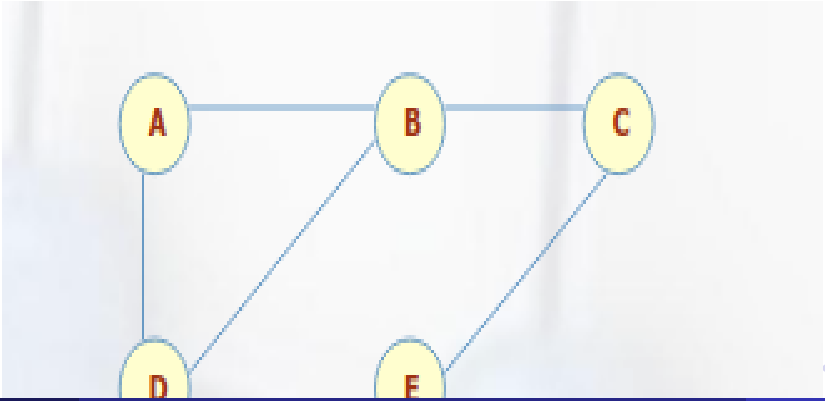


Introduction

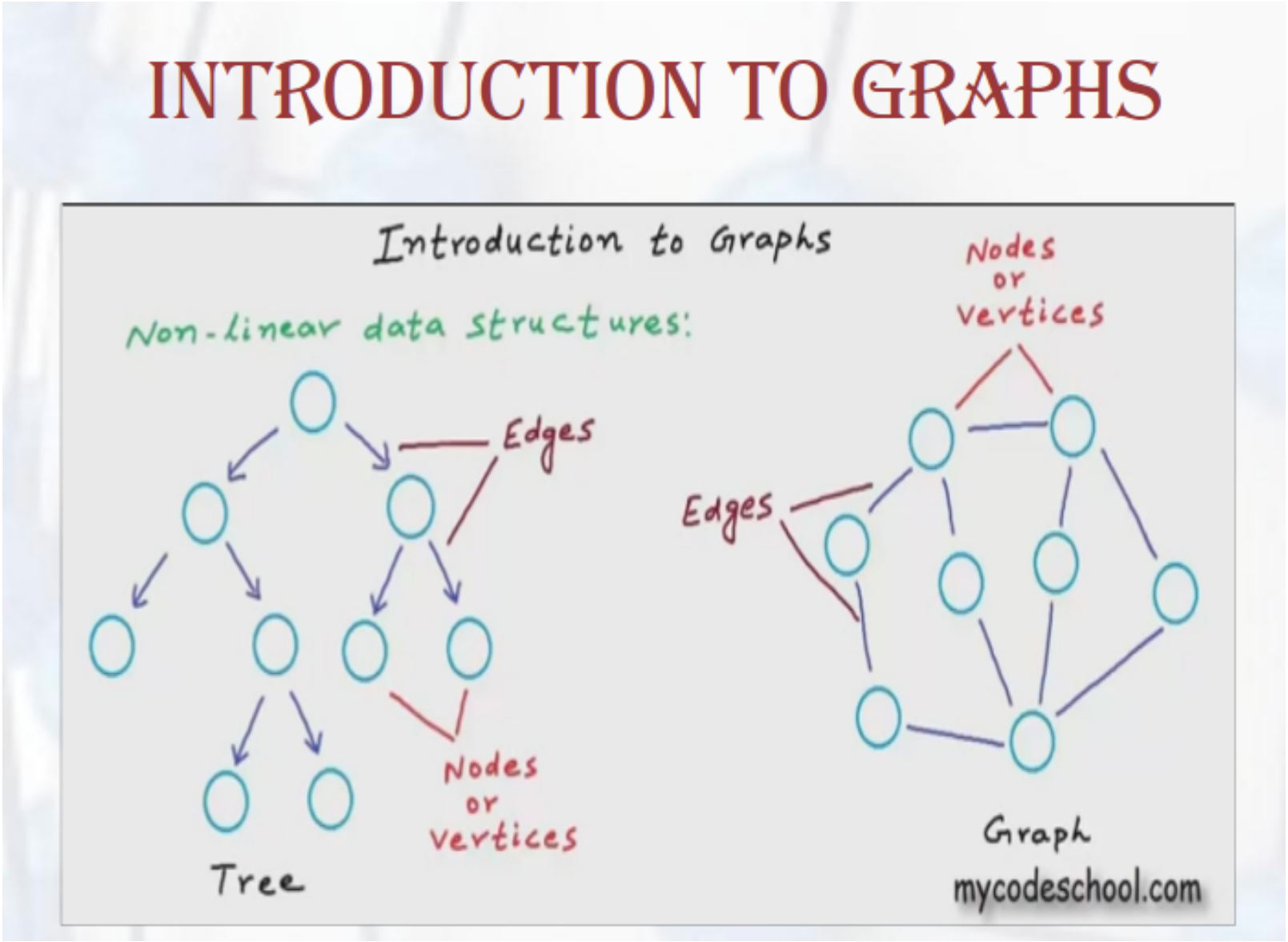


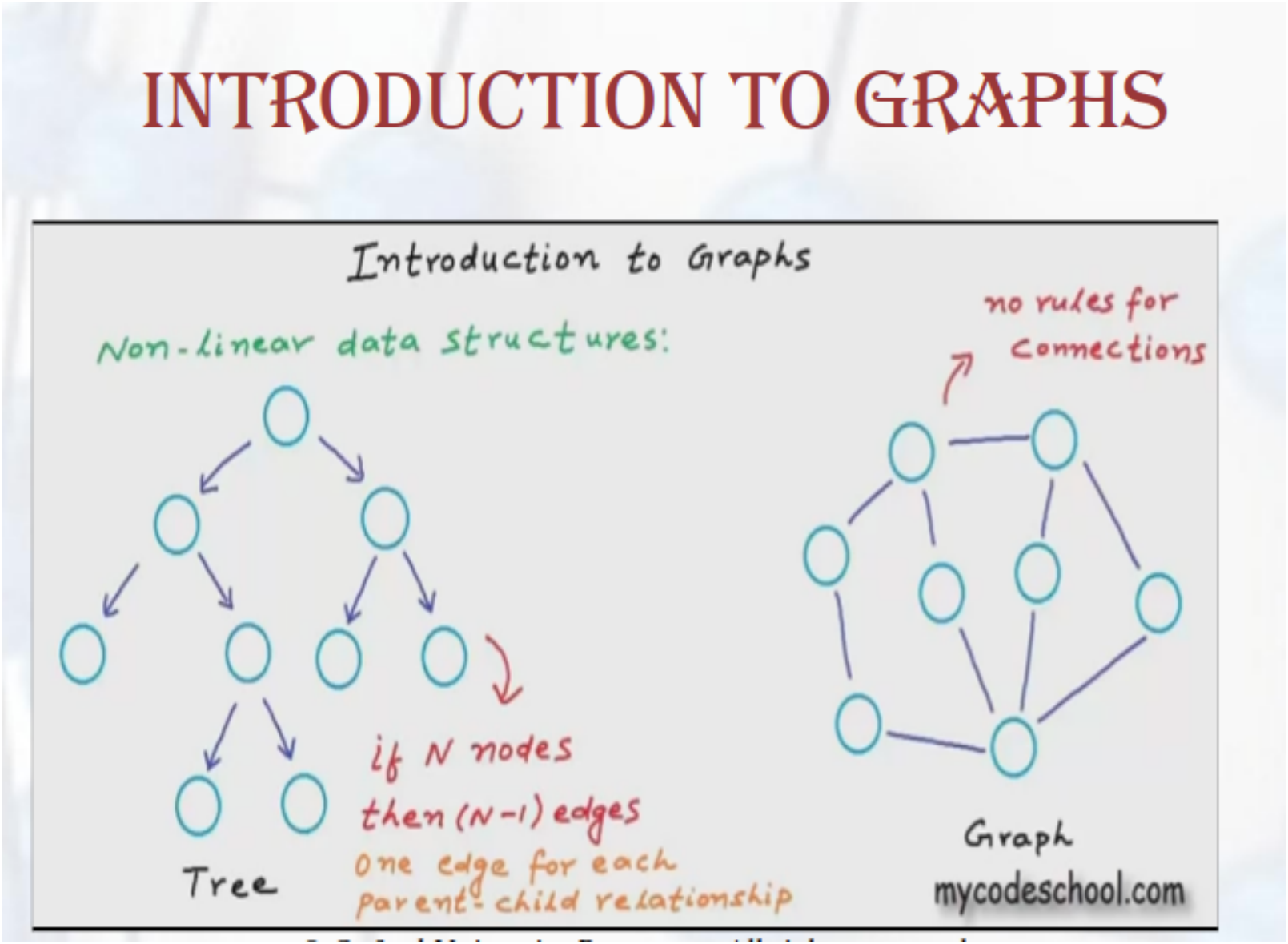
Definition

- A graph can be directed or undirected. In an undirected graph, the edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. The above figure shows an undirected graph because it does not gives any information about the direction of the edges.
- The given figure shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).



Introduction



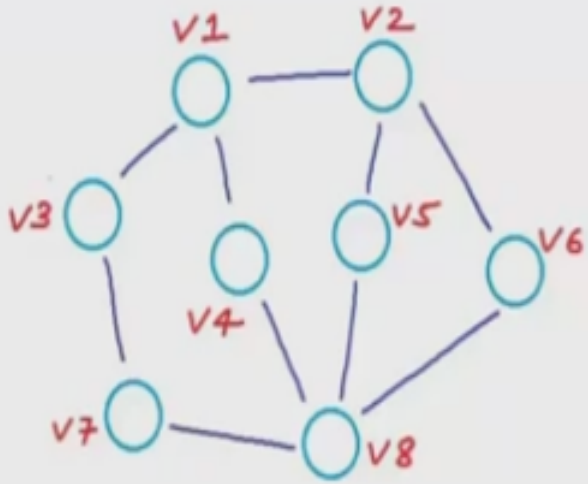

INTRODUCTION TO GRAPHS
Introduction to Graphs
Non-linear data structures:
no rules for connections
if N nodes
then (N-1) edges
One edge for each parent-child relationship
Tree
Graph
mycodeschool.com

INTRODUCTION TO GRAPHS

Graph:

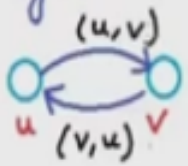
A graph G is an ordered pair of a set V of vertices and a set E of edges.

$$G = (V, E)$$



$$V = \{v1, v2, v3, v4, v5, v6, v7, v8\}$$

Edges:



directed

(u, v)

undirected

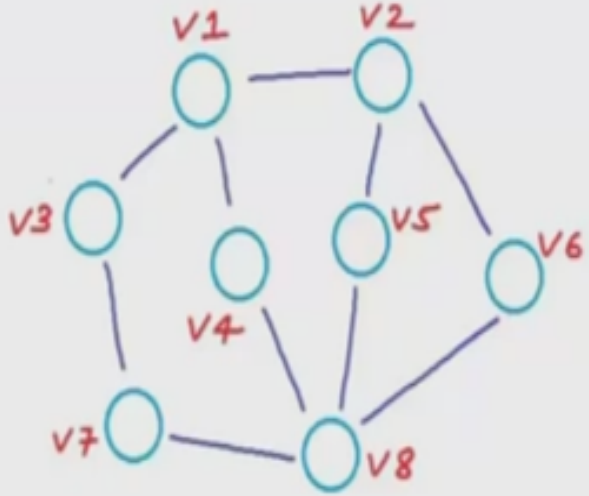
mycodeschool.com

INTRODUCTION TO GRAPHS

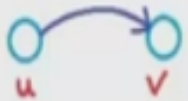
Graph:

A graph G is an ordered pair of a set V of vertices and a set E of edges.

$$G = (V, E)$$



Edges:



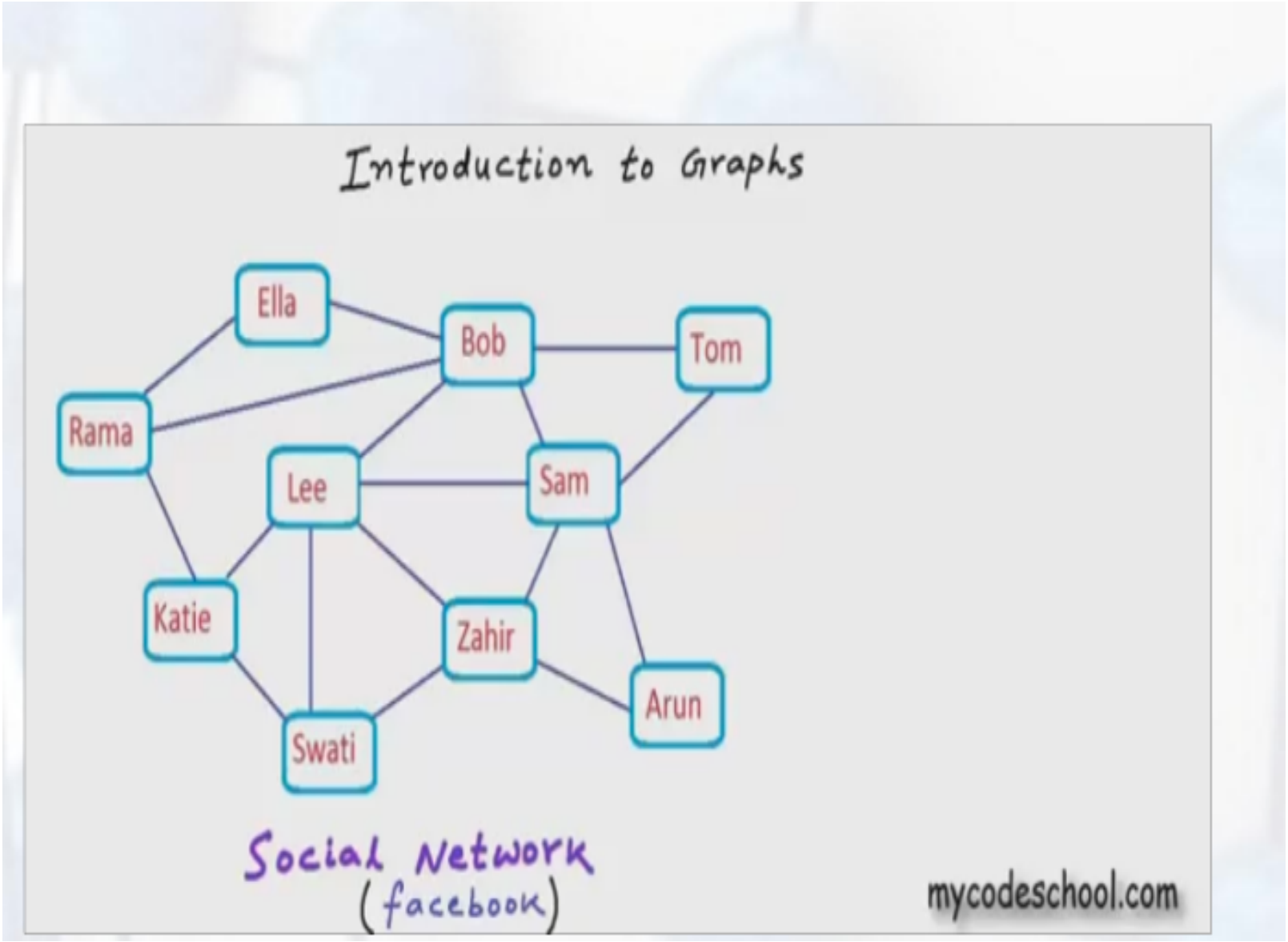
directed
(u, v)

undirected
$\{u, v\}$

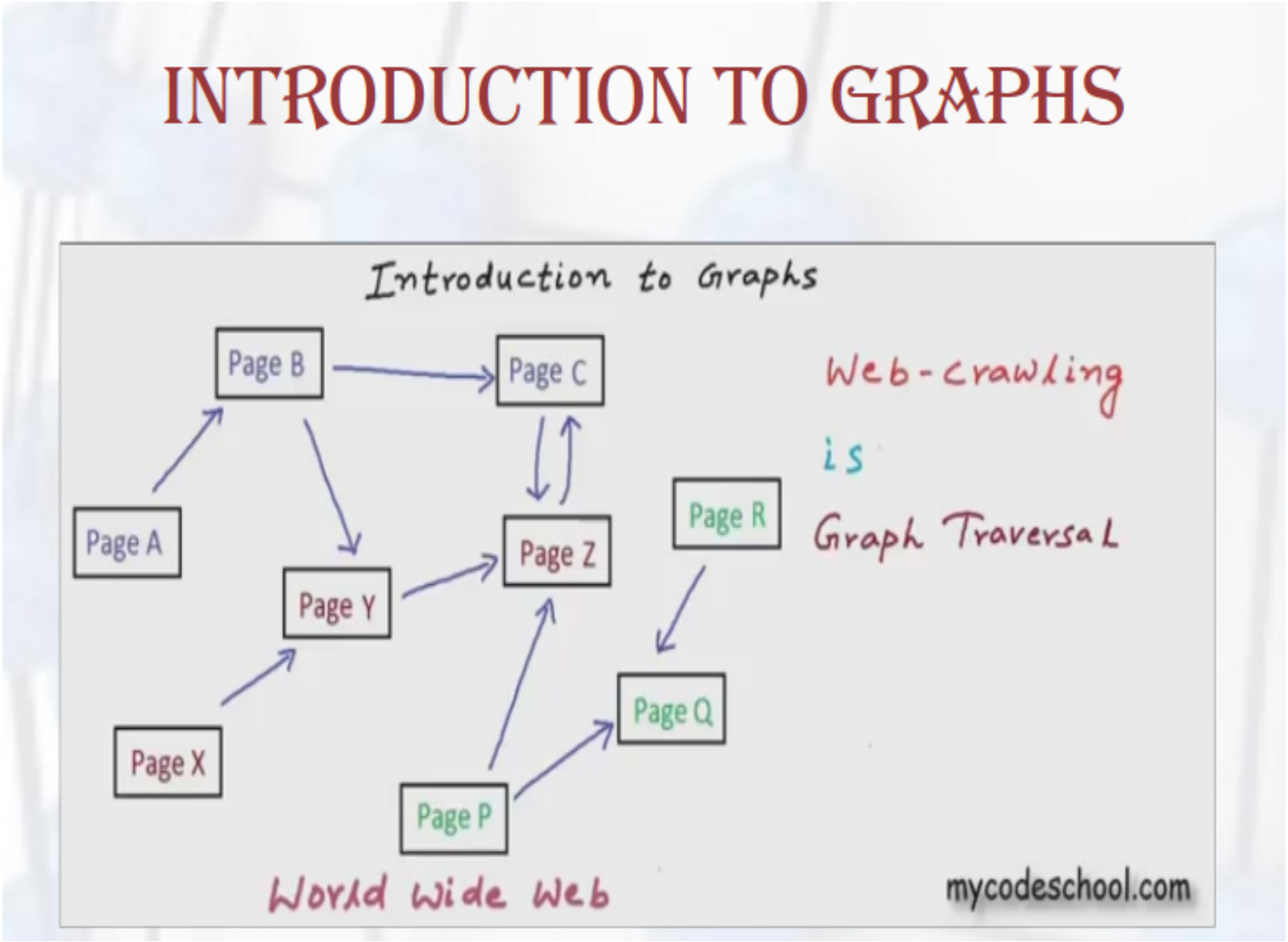
$$V = \{v1, v2, v3, v4, v5, v6, v7, v8\}$$

$$E = \{\{v1, v2\}, \{v1, v3\}, \{v1, v4\}, \{v2, v5\}, \{v2, v6\}, \{v3, v7\}, \{v4, v8\}, \{v7, v8\}, \{v5, v8\}, \{v6, v8\}\}$$

APPLICATIONS OF GRAPHS



APPLICATIONS OF GRAPHS

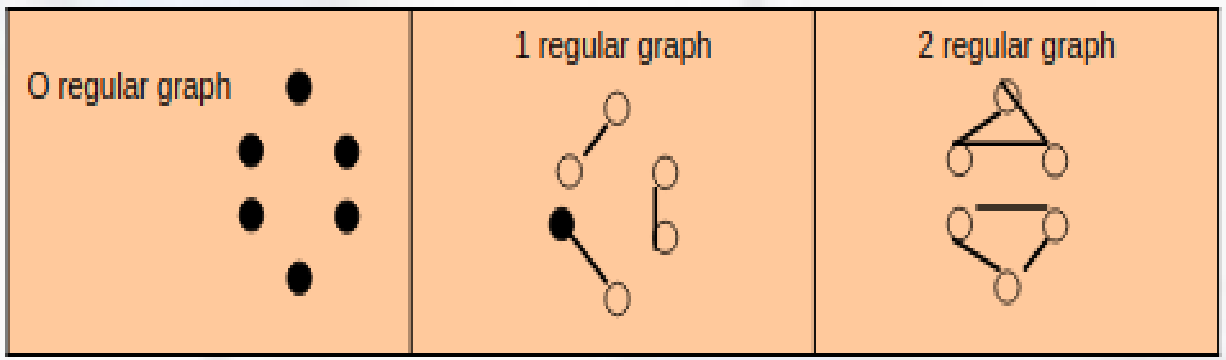


APPLICATIONS OF GRAPHS

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges

Graph Terminology

- *Adjacent Nodes or Neighbors:* For every edge, e = (u, v) that connects nodes u and v; the nodes u and v are the end-points and are said to be the adjacent nodes or neighbors.
- *Degree of a node:* Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.
- *Regular graph:* Regular graph is a graph where each vertex has the same number of neighbors. That is every node has the same degree. A regular graph with vertices of degree k is called a k-regular graph or regular graph of degree k.



Graph Terminology

- *Path:* A path P, written as P = (v0, v1, v2,. . . .., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi-1 is adjacent to vi for i = 1, 2, 3, . . . , n.
- *Closed path*: A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn.
- *Simple path*: A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.
- *Cycle:* A closed simple path with length 3 or more is known as a cycle. A cycle of length k is called a k – cycle.

Graph Terminology

- *Connected graph:* A graph in which there exists a path between any two of its nodes is called a connected graph. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.
- *Complete graph:* A graph G is said to be a complete, if all its nodes are fully connected, that is, there is a path from one node to every other node in the graph. A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.
- *Labeled graph or weighted graph:* A graph is said to be labeled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. Weight of the edge, denoted by w(e) is a positive value which indicates the cost of traversing the edge.

Graph Terminology

- *Multiple edges:* Distinct edges which connect the same end points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.
- *Loop:* An edge that has identical end-points is called a loop. That is, e = (u, u).
- *Multi- graph:* A graph with multiple edges and/or a loop is called a multi-graph.
- *Size of the graph:* The size of a graph is the total number of edges in it.

Graph Terminology

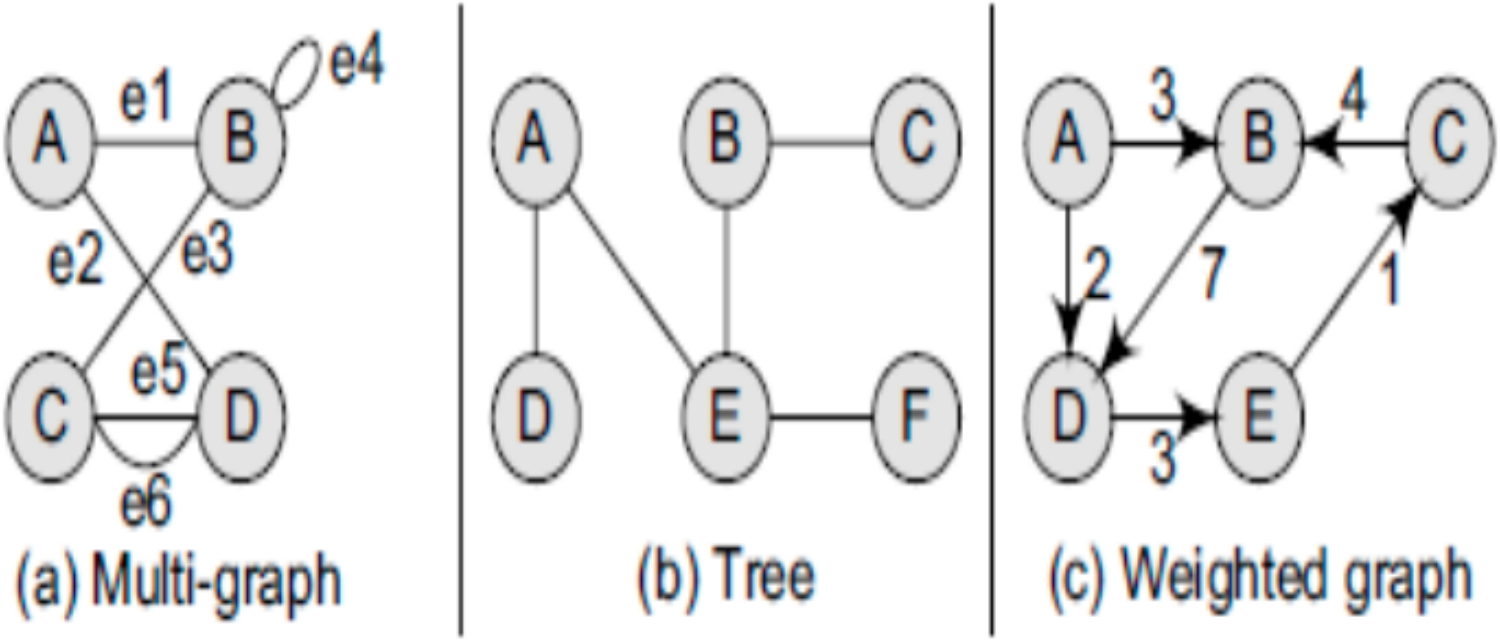


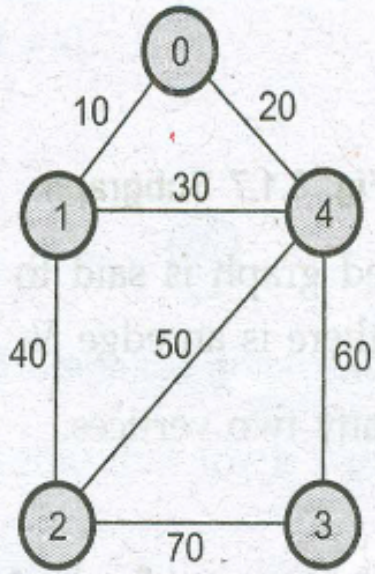
Figure 13.4 Multi-graph, tree, and weighted graph

Graph Terminology

Weighted graph

A weighted graph is a graph which consists of weights along its edges.

For example :



Graph Terminology

Path : A path is denoted using sequence of vertices and there exists an edge from one vertex to the next vertex.

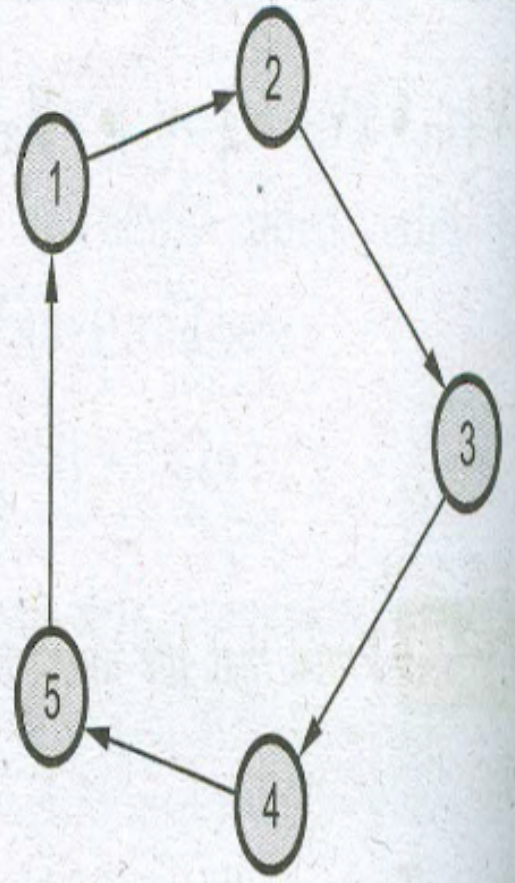


Fig. 3.1.10 Path of G is 1-2-3-4-5

Graph Terminology

Cycle : A closed walk through the graph with repeated vertices, mostly having the same starting and ending vertex is called a cycle.

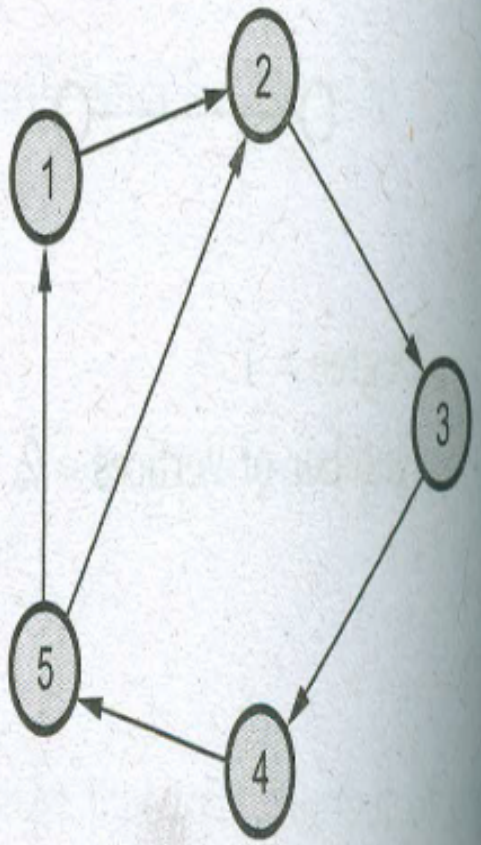


Fig. 3.1.11 Cycle 2-3-4-5-2 or 1-2-3-4-5-1

Graph Terminology

Add figure 15

Cycle : A closed walk through the graph with repeated vertices, mostly having the same starting and ending vertex is called a cycle.

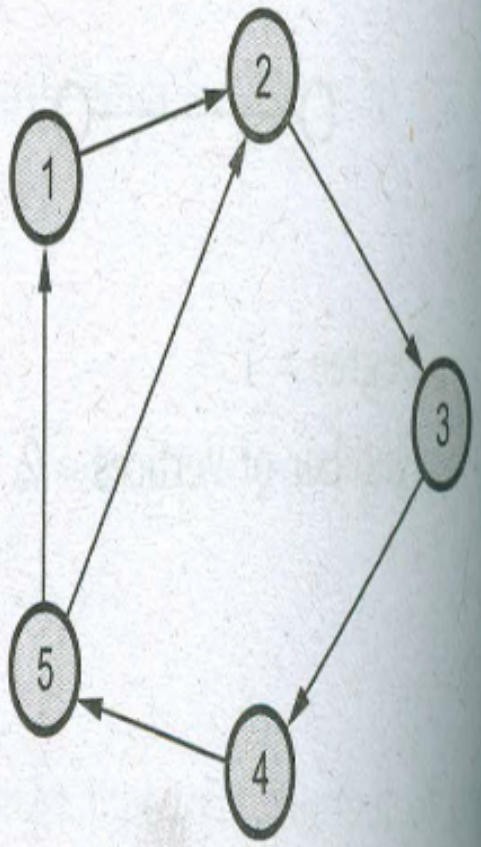


Fig. 3.1.11 Cycle 2-3-4-5-2 or 1-2-3-4-5-1

Directed Graph

- A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v)-
- The edge begins at u and terminates at v
- U is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e
- U is the predecessor of v. Correspondingly, v is the successor of u nodes u and v are adjacent to each other.

Terminology of a Directed Graph

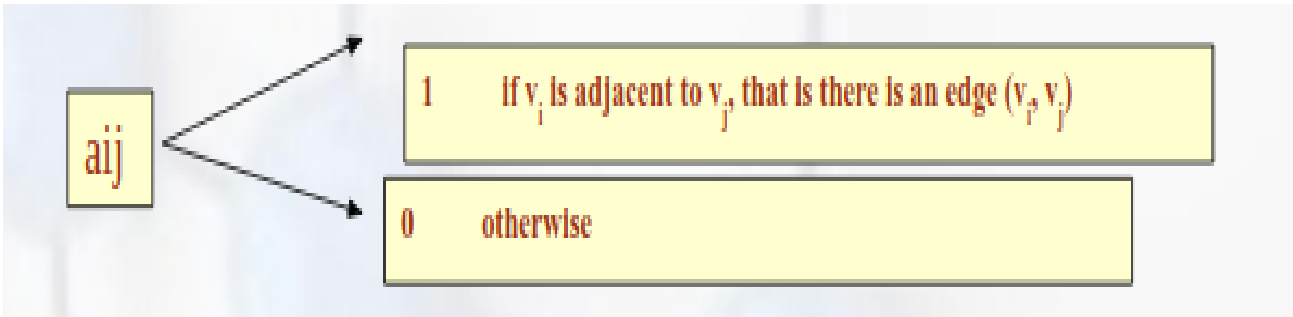
- *Out-degree of a node:* The out degree of a node u, written as outdeg(u), is the number of edges that originate at u.
- *In-degree of a node:* The in degree of a node u, written as indeg(u), is the number of edges that terminate at u.
- *Degree of a node:* Degree of a node written as deg(u) is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u)
- *Source:* A node u is known as a source if it has a positive out-degree but an in-degree = 0.
- *Sink:* A node u is known as a sink if it has a positive in degree but a zero out-degree.
- *Reachability:* A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

Terminology of a Directed Graph

- *Strongly connected directed graph*: A digraph is said to be strongly connected if and only if there exists a path from every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.
- *Unilaterally connected graph*: A digraph is said to be unilaterally connected if there exists a path from any pair of nodes u, v in G such that there is a path from u to v or a path from v to u but not both.
- *Parallel/Multiple edges*: Distinct edges which connect the same end points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.
- *Simple directed graph*: A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycle with an exception that it cannot have more than one loop at a given node

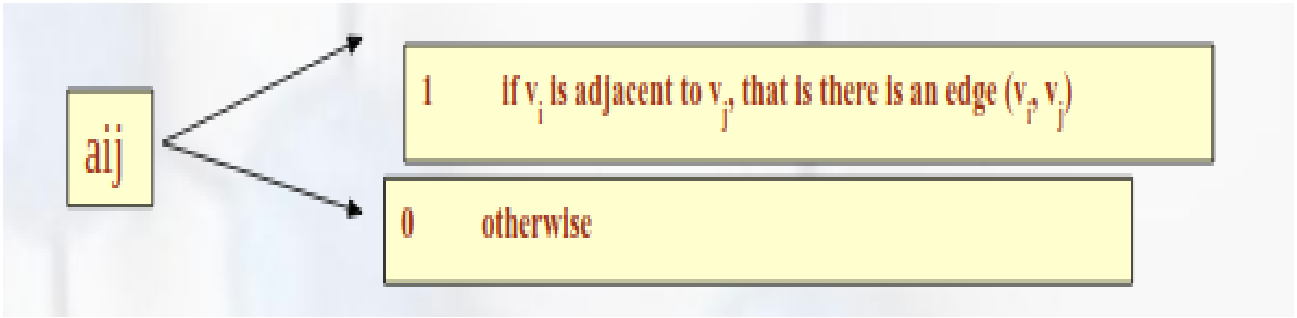
Adjacency Matrix Representation

- An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, we have learnt that, two nodes are said to be adjacent if there is an edge connecting them.
- In a directed graph G, if node v is adjacent to node u, then surely there is an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have dimensions of n X n.
- In an adjacency matrix, the rows and columns are labeled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other. However, if the nodes are not adjacent, a_{ij} will be set to zero.



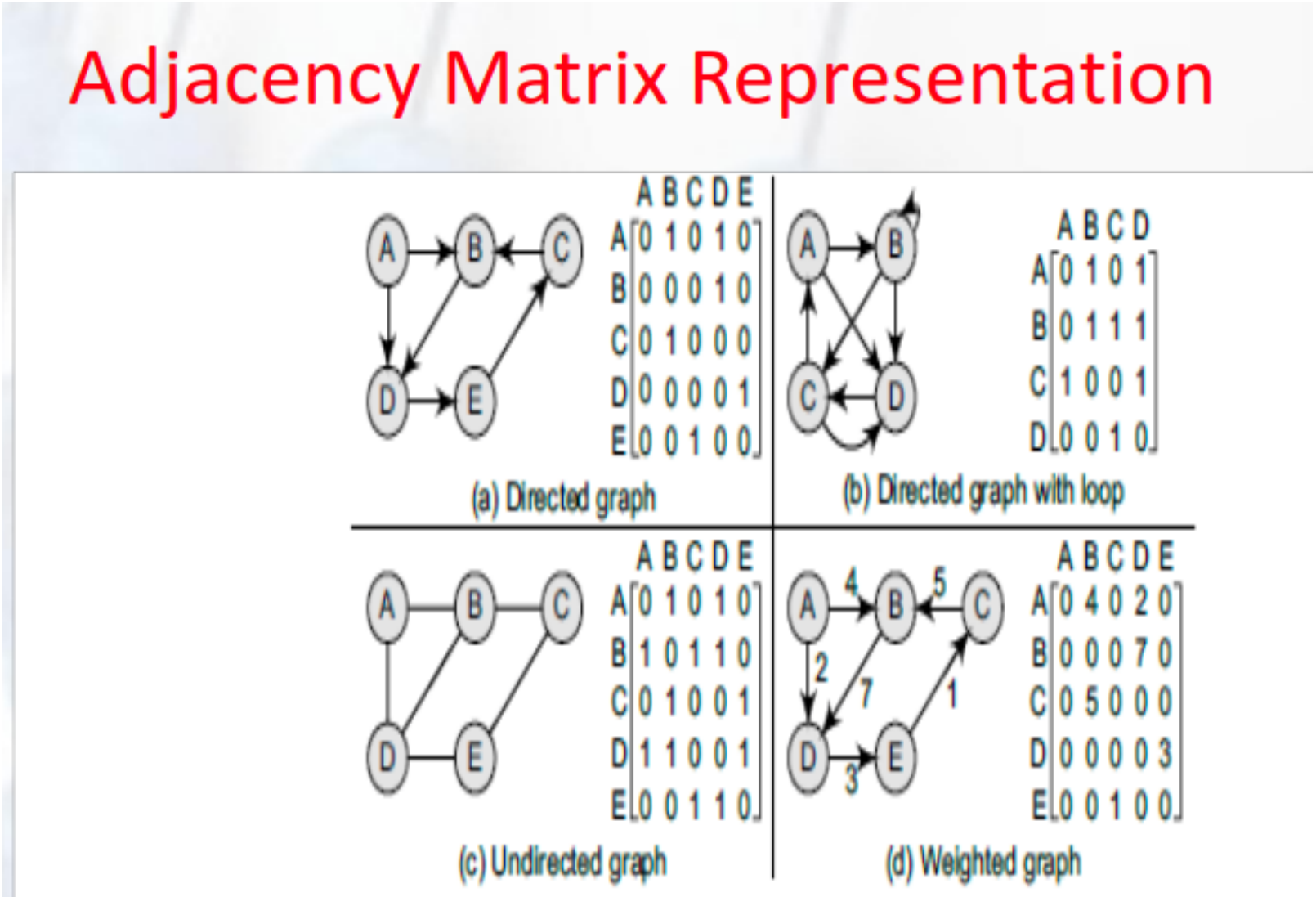
- Since an adjacency matrix contains only 0s and 1s, it is called a bit

Adjacency Matrix Represnetation

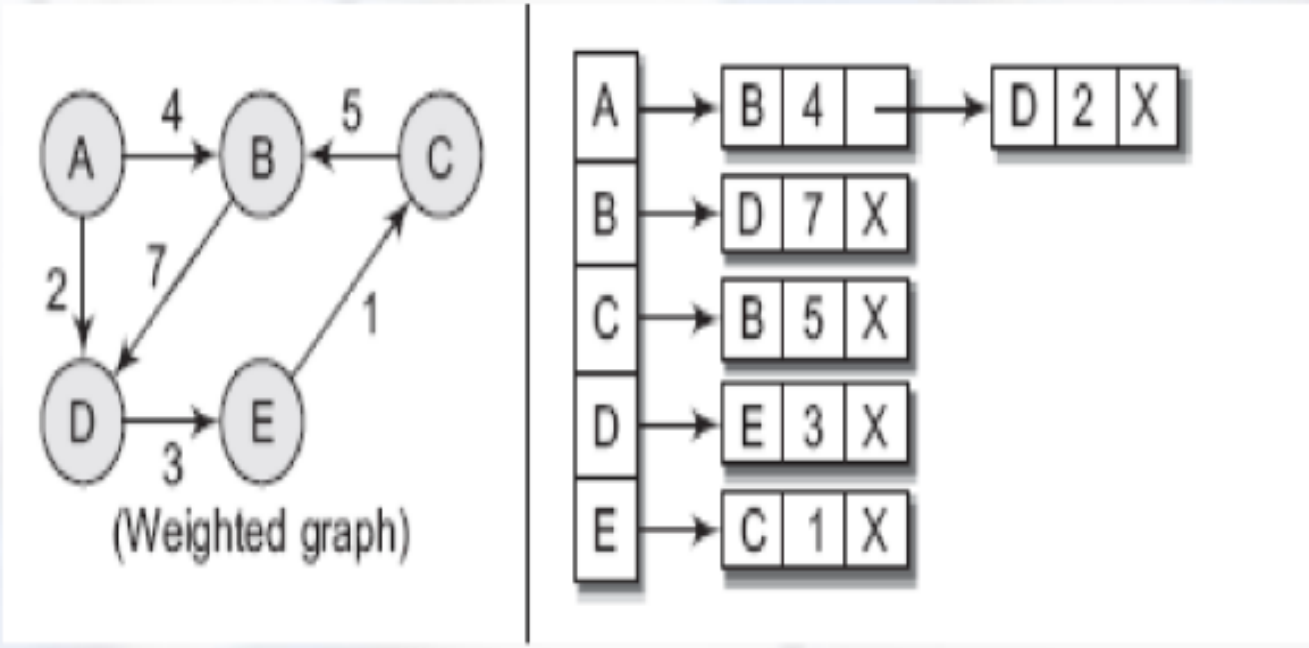


Since, an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. therefore, a change in the order of nodes will result in a different adjacency matrix.

Adjacency Matrix Representation



Adjacency List Representation

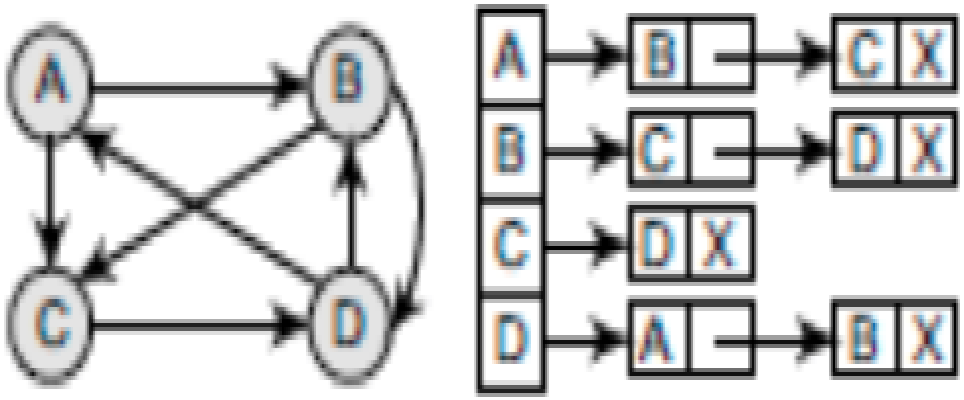


Adjacency List

The adjacency list is another way in which graphs can be represented in computer's memory.
This structure consists of a list of all nodes in G.
Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to itself.
The key advantage of using an adjacency list includes:

- It is easy to follow, and clearly shows the adjacent nodes of a particular node
- It is often used for storing graphs that have a small to moderate number of edges.

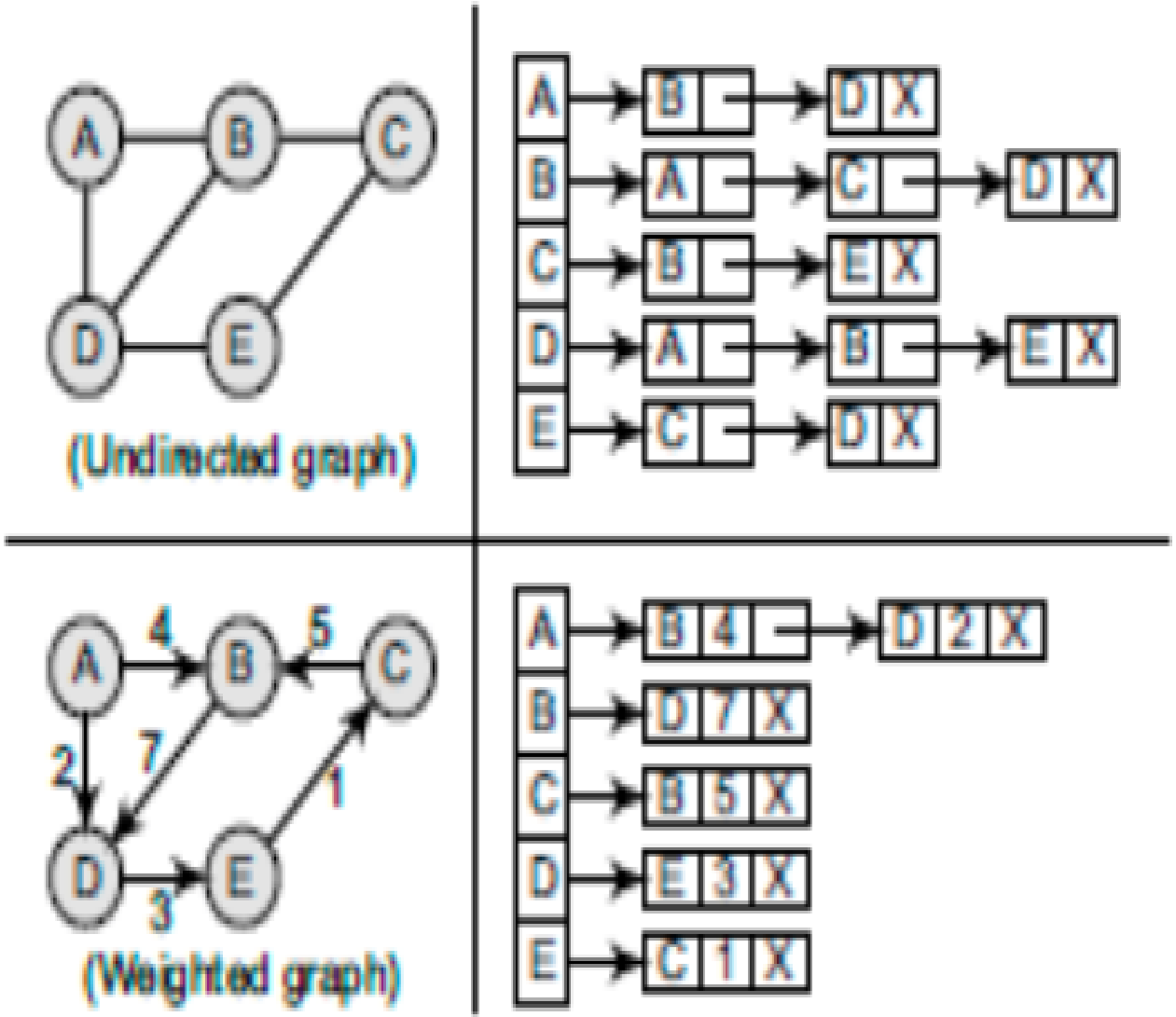
Adjacency List of Graph



Graph G and its adjacency list:

- For a directed graph, the sum of lengths of all adjacency lists is equal to the number of edges in G.
- However, for an undirected graph, the sum of lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge v to u.
- The adjacency list can also be modified to store weighted graphs.

Adjacency List of Graph



Comparison between Tree and Graph

S.No	Graph	Tree
1	Graph is a non-linear data structure.	Tree is a non-linear data structure.
2	It is collection of node/vertices and edges.	It is collection of node and edges.
3	Each node can have any number of edges.	general tree consist of the node having any number of child node But in case of Binary tree every node can have at most two child
4	There is no unique node called root in Graph.	There is a unique node called root in tree.
5	A cycle can be formed.	There will not be any cycle.
6	Application: Finding shortest path in network.	Application: game tree, decision tree, etc.

Traversal of Graph

- There is no first vertex or root in the graph, hence the traversal can start from any vertex.
- We can choose any arbitrary vertex as the starting vertex.
- A traversal algorithm will produce different sequences for different starting vertices.
- In the tree or list, when we start traversing from the first vertex, all the elements are visited but in graph only those vertices will be visited which are reachable from the starting vertex.
- In tree or list while traversing, we never encounter a vertex more than once while in graph we may reach a vertex more than once.

Traversal of Graph

- In the tree or list we have unique traversals. For example- if we are traversing a binary tree in inorder there can be only one sequence in which vertices are visited. But in graph, for the same technique of the traversal there can be different sequences in which the vertices can be visited.
- Like binary trees, in graphs also there can be many methods by which a graph can be traversed but two of them are standard and are known as *breath first search* and *depth first search*.

Traversal of Graph

By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section. These two methods are-

- Breadth first search
- Depth first search
- While **breadth first search** will use a *queue* as an auxiliary data structure to store nodes for further processing, the **depth-first search** scheme will use a *stack*. But both these algorithms will make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1, 2 or depending on its current state. Table I shows the value of status and its significance.

Traversal of Graph

STATUS	STATE OF THE NODE	DESCRIPTION
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Breadth First Search

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.
- That is, we start examining the node A and then all the neighbors of A are examined. In the next step we examine the neighbors of neighbors of A, so on and so forth

Breadth First Search

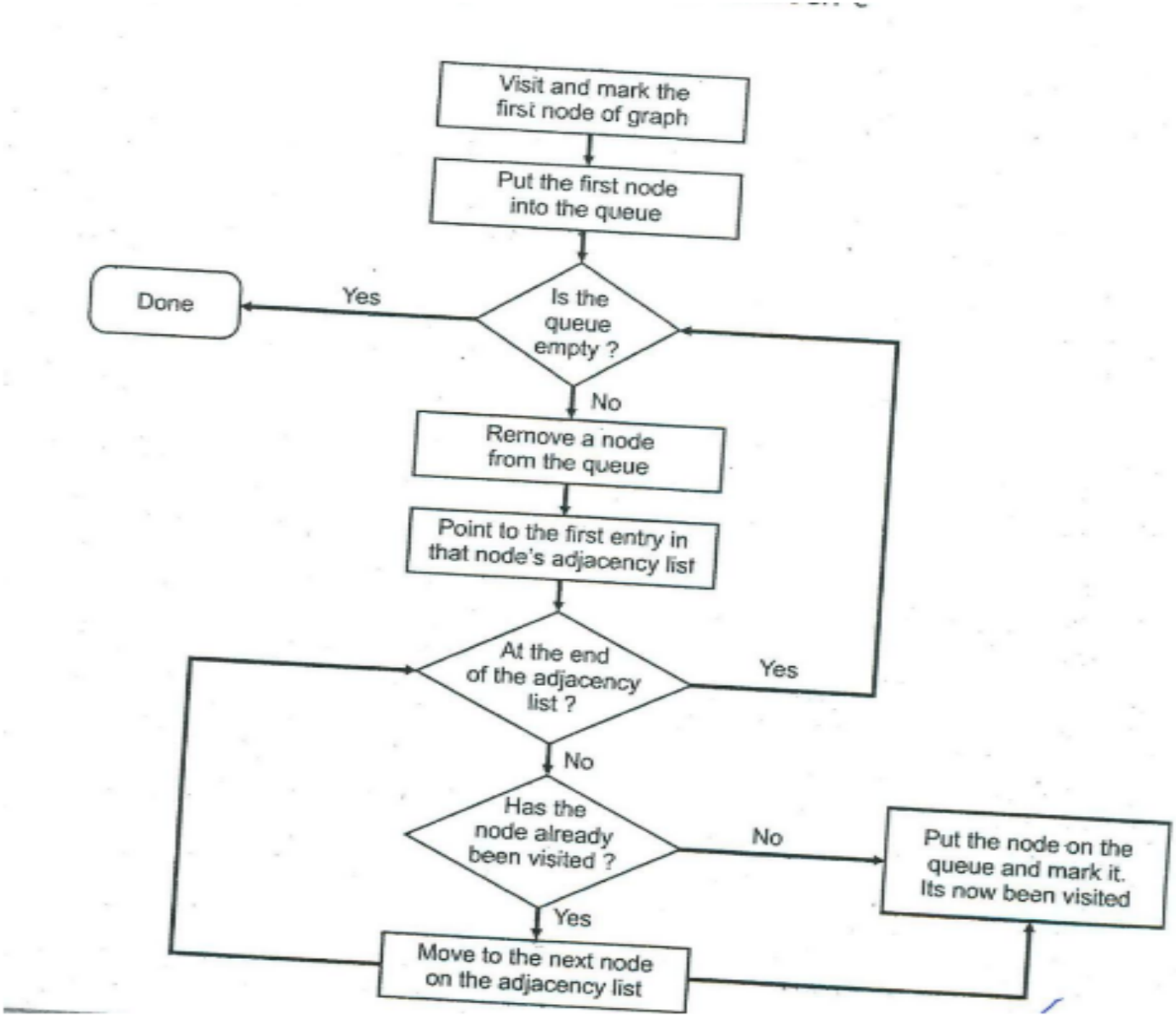
Algorithm for breadth-first search in a graph G beginning at a starting node A

Step 1: SET STATUS = 1 (ready state) for each node in G.
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbors of N that are in the ready state
(whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT

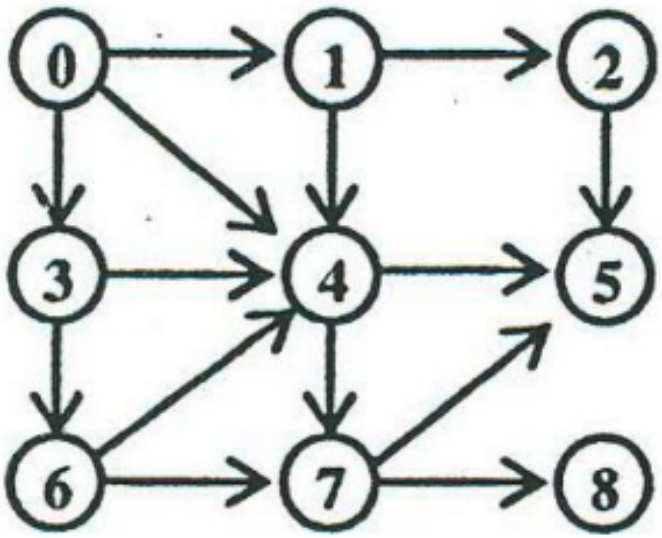
BFS

- Visit the starting Vertex
- Visit all the vertices Adjacent to it
- Pick adjacent vertices one by one and visit their adjacent vertices and so on.
- Equivalent to level order Traversal of trees

Breadth First Search

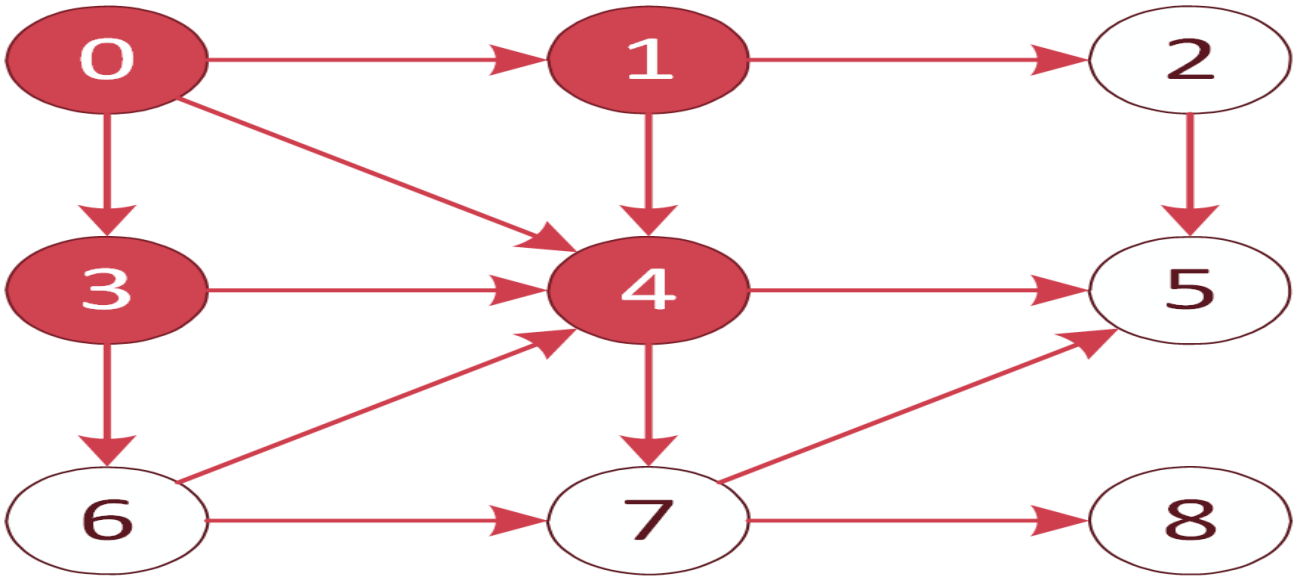


Breadth First Search



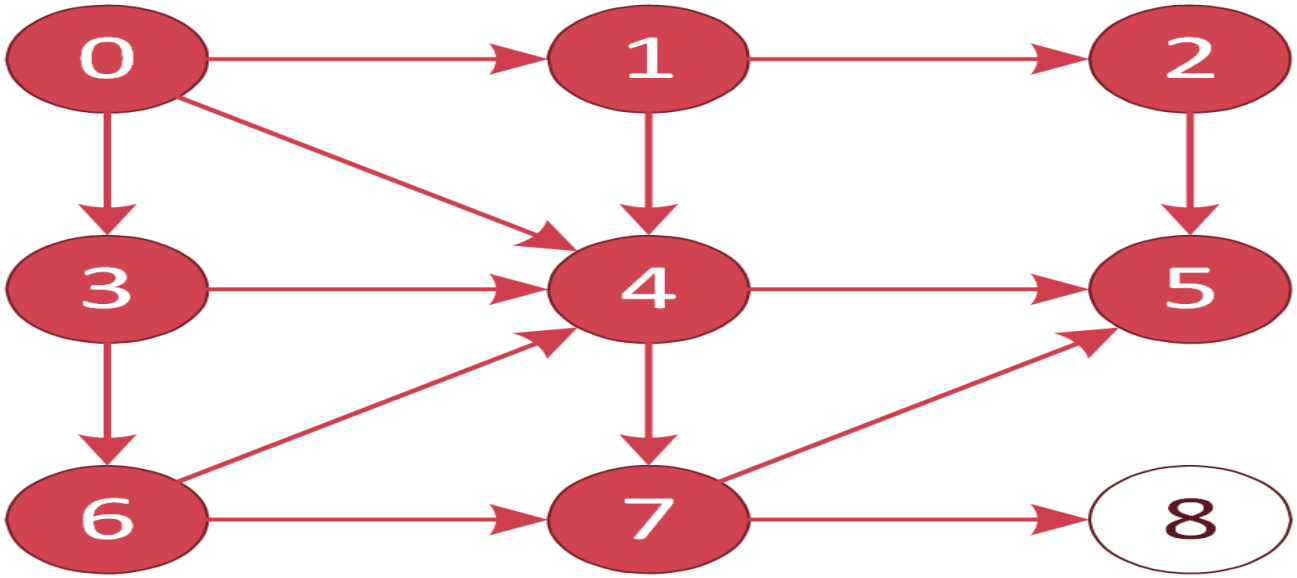
First we will visit Vertex 0 ad
then all vertices adjacent to it
Now TRAVERSAL is 0134

Breadth First Search



Now, first we visit all the vertices adjacent to 1 then all the vertices adjacent to 3 and then all the vertices adjacent to 4, So first we visit 2,then 6 and then 5,7. Note that vertex 4 is adjacent to 1 and 3, but it has already been visited so we ignore it. Now the traversal is: **01342657**

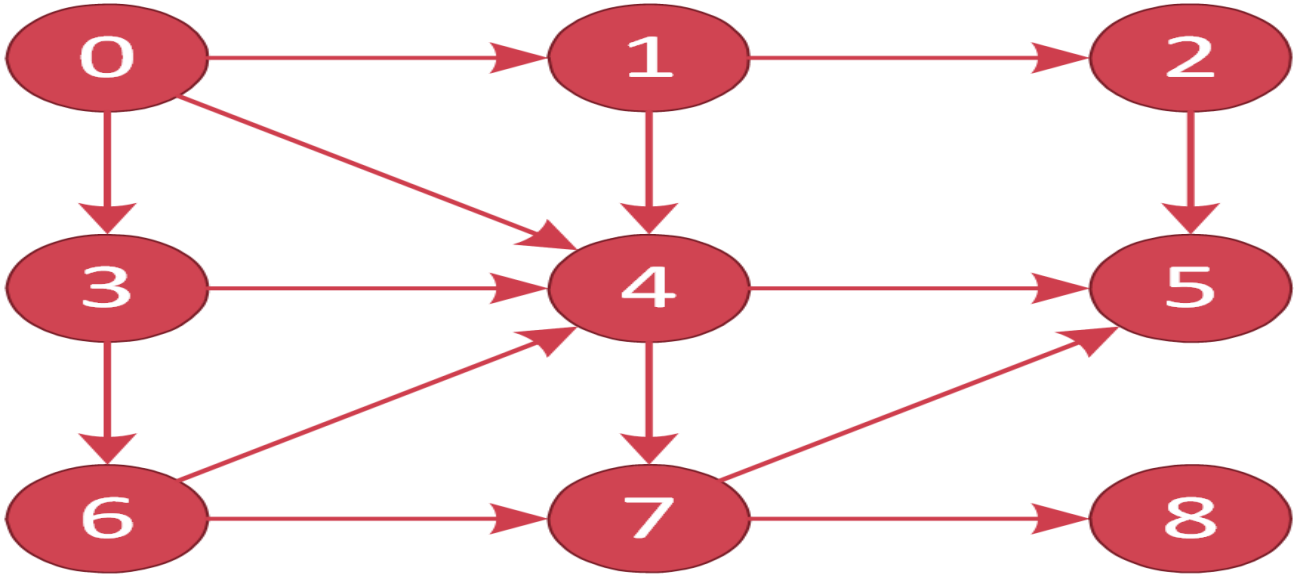
Breadth First Search



Now, we visit one by one all the vertices adjacent to vertices 2,5,6,7. We can see that 5 is adjacent to 2, but it has already been visited so will just ignore it proceed further. Now vertices adjacent to vertex 6 are vertices 4 and 7 which have already been visited so ignore then also. Vertex 5 has no adjacent vertices. Vertex 7 has vertices 5 and 8 adjacent to it out of which 8 has not been visited, so visit vertex 8.

Now the traversal is *013426578*

Breadth First Search



Now, we have to visit vertices adjacent to 8 but there is no vertex adjacent to 8 So our procedure stops.
This was the traversal when we take 0 as starting vertex.
Suppose we take 1 as starting vertex. Then applying above technique we get the following traversal:- *1,2,4,5,7,8*

Breadth First Search

Here are different traversals when we take different starting vertices.

Start Vertex	Traversal
0	0 1 3 4 2 6 5 7 8
1	1 2 4 5 7 8
2	2 5
3	3 4 6 5 7 8
4	4 5 7 8
5	5
6	6 4 7 5 8
7	7 5 8
8	8

Implementation of BFS using Queue

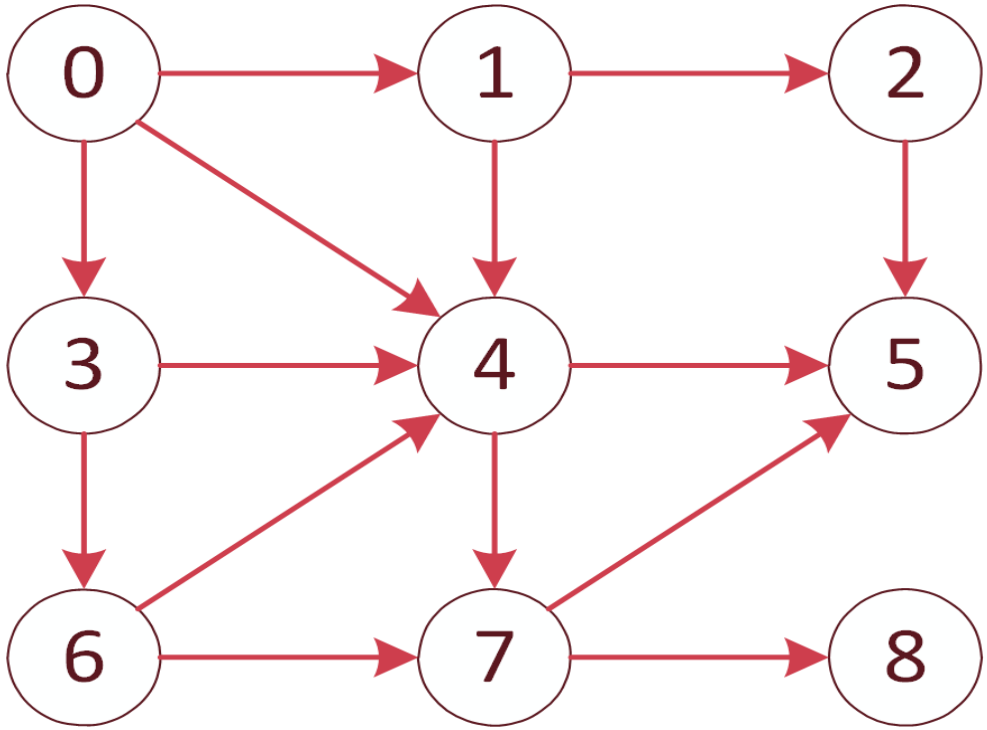
During the algorithm, any vertex will be in one of three state-**initial,waiting** and **visited**. At the start of the algorithm, all the vertices will be in initial stage. When vertex will be insert in the *queue* its state will change from initial to waiting. When a vertex will be deleted from queue and visited, its state will change from waiting to visited. This procedure is as:-

Implementation of BFS using Queue

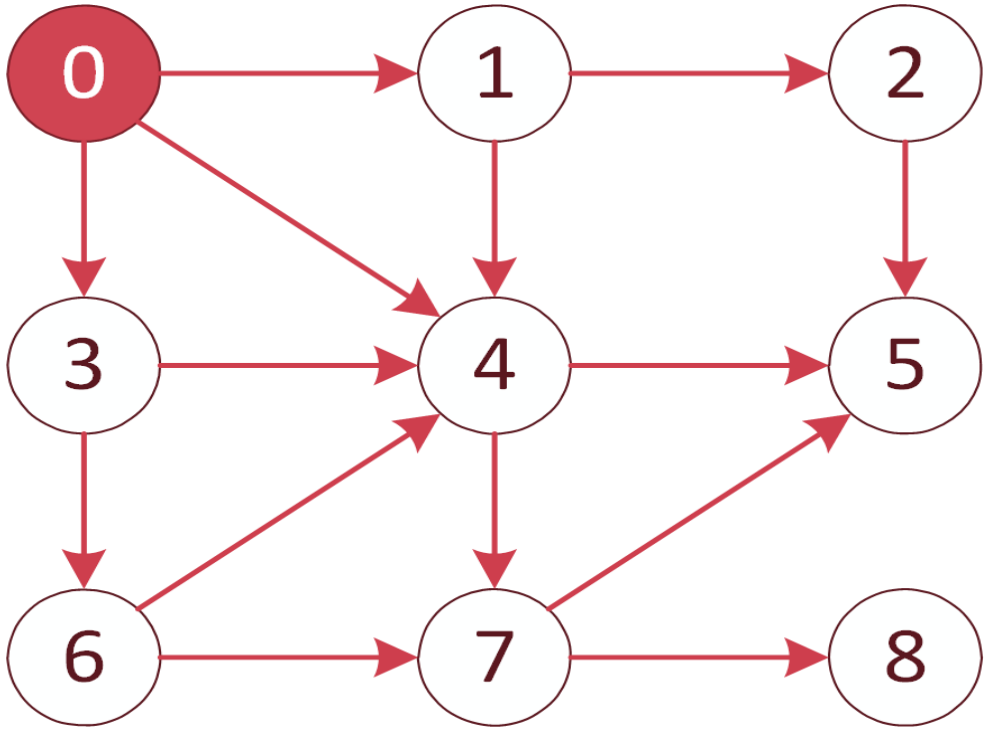
Initially queue is empty, and all vertices are in initial state.

- Insert the starting vertex into the *queue*, change its state to *waiting*
- Delete front element from the *queue* and visit it, change its state to *visited*.
- Look for the adjacent vertices of the deleted element, and from these insert only those which are in the initial state, Change the state of the all these vertices from initial to waiting.
- Repeat step 2,3 until queue is empty.

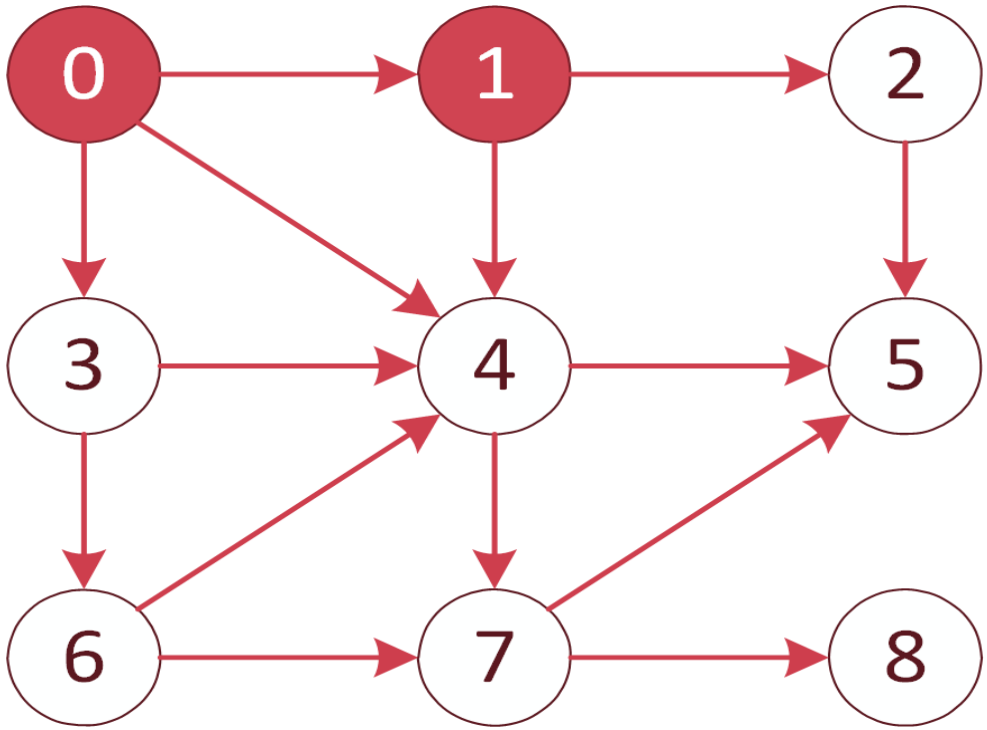
Insert 0 in to the *Queue*
QUEUE : 0



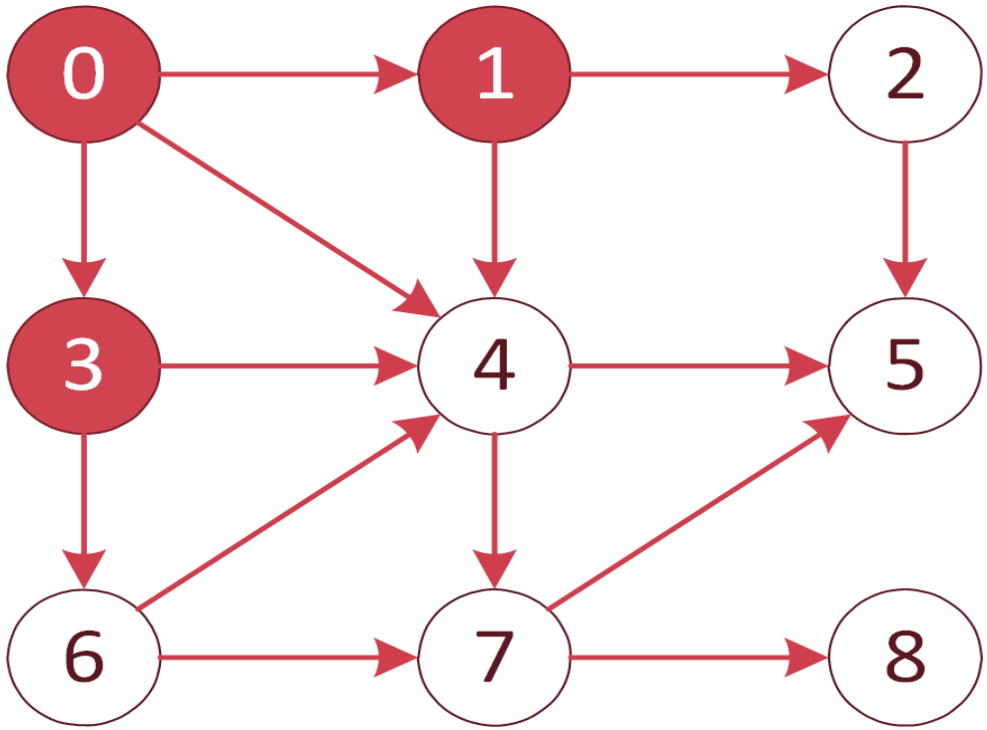
Delete vertex 0 from the *Queue* and visit it.
TRAVERSAL : 0
vertex adjacent to 0 are 1,3,4 all these are in initial state. So inset them into *queue*.
QUEUE: 1,3,4



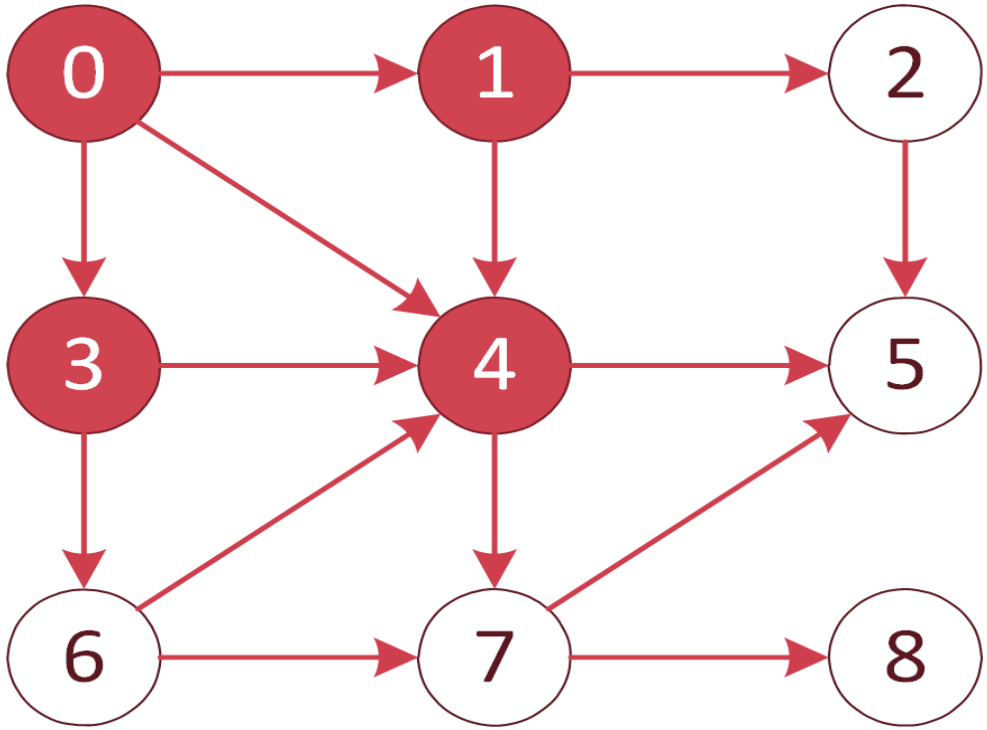
Delete vertex 1 from queue, and visit it.
TRAVERSAL: 0,1
vertices adjacent to 1 are 2,4. Vertex 4 is in waiting state because it is in the *queue*, So it is not inserted in the queue. Vertex 2 is in initial state, So insert it into queue.
QUEUE : 3,4,2



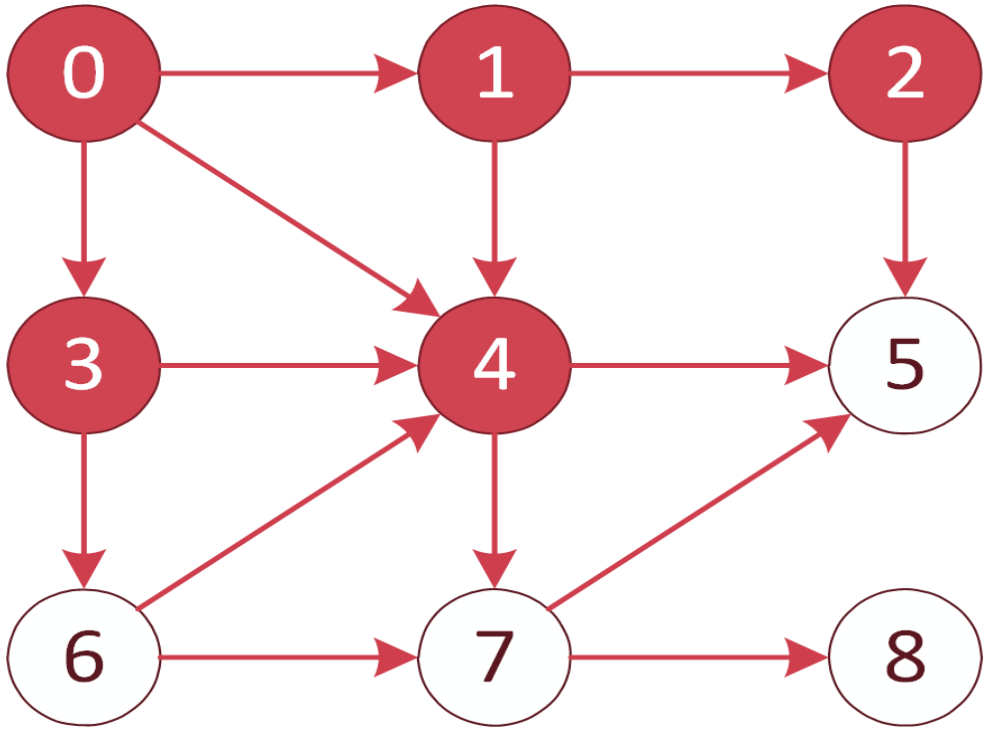
Delete vertex 3 from queue, and visit it.
TRAVERSAL : 0,1,3
Vertices adjacent to 3 are 4 and 6. Vertex 4 is in waiting state because it is in the *queue*, So it is not inserted in the queue. Vertex 6 is in initial state, So insert it into queue.
QUEUE : 4,2,6



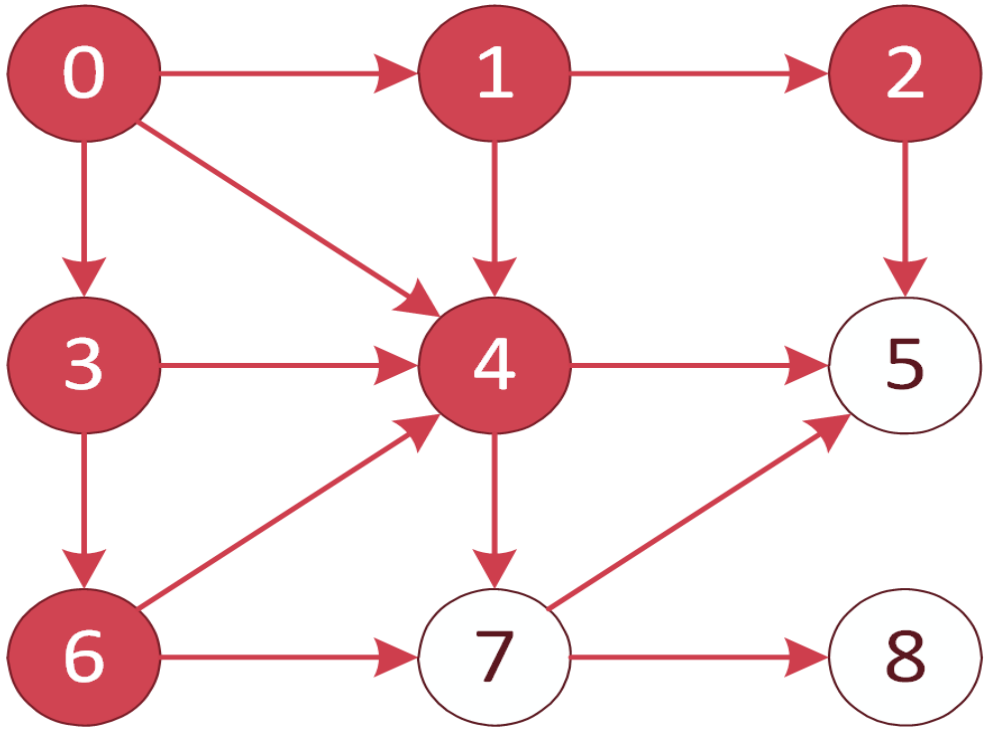
Delete vertex 4 from queue, and visit it.
TRAVERSAL : 0,1,3,4
Vertices adjacent to 4 are 5 and 7. Both are in initial state, So insert it into queue.
QUEUE : 2,6,5,7



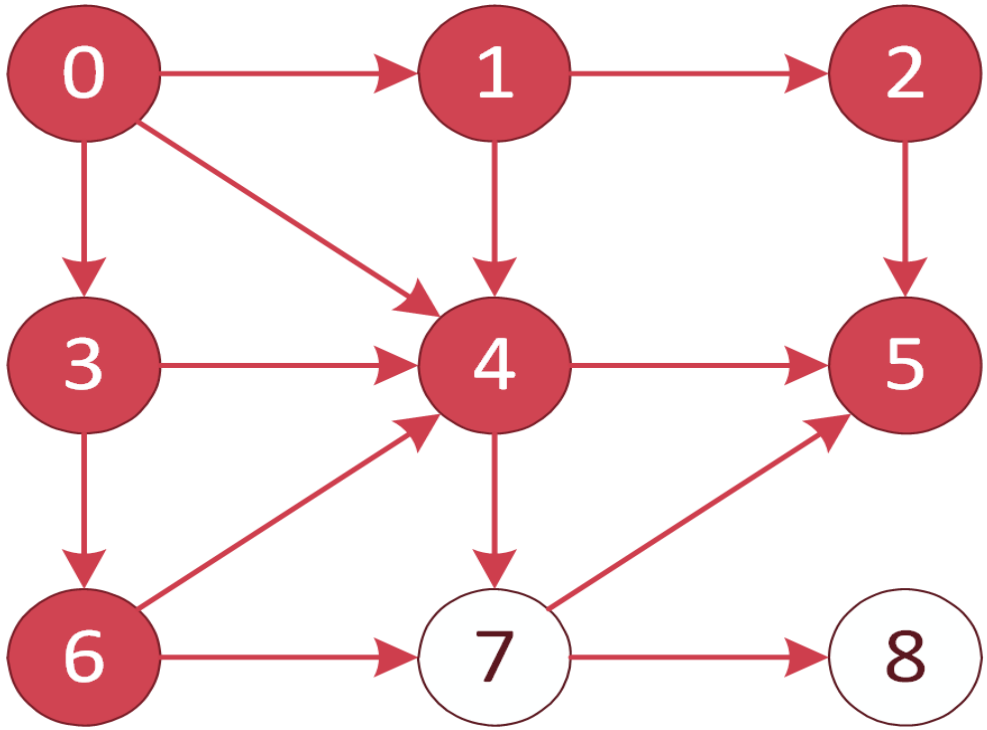
Delete vertex 2 from queue, and visit it.
TRAVERSAL : 0,1,3,4,2
Vertices adjacent to 2 is 5. Vertex 6 is in waiting state because it is already in queue. So it is not inserted into queue.
QUEUE : 6,5,7



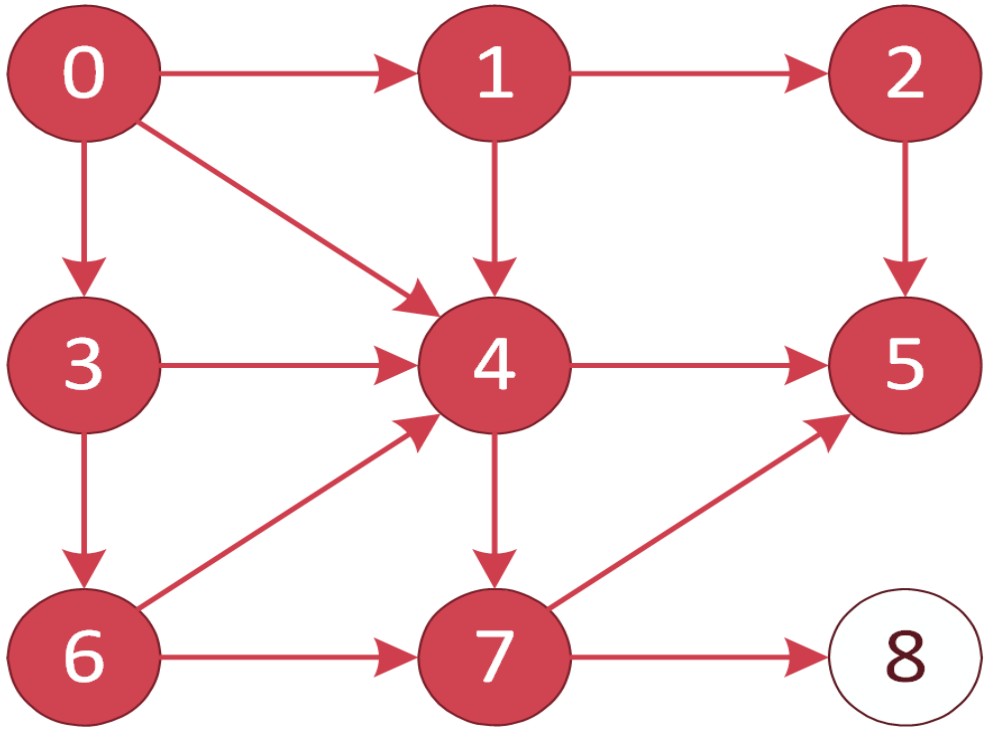
Delete vertex 6 from queue, and visit it.
TRAVERSAL : 0,1,3,4,2,6
Vertices adjacent to 6 are 7. Vertex 7 is in waiting state because it is already in queue, So it is not inserted into queue.
QUEUE : 5,7



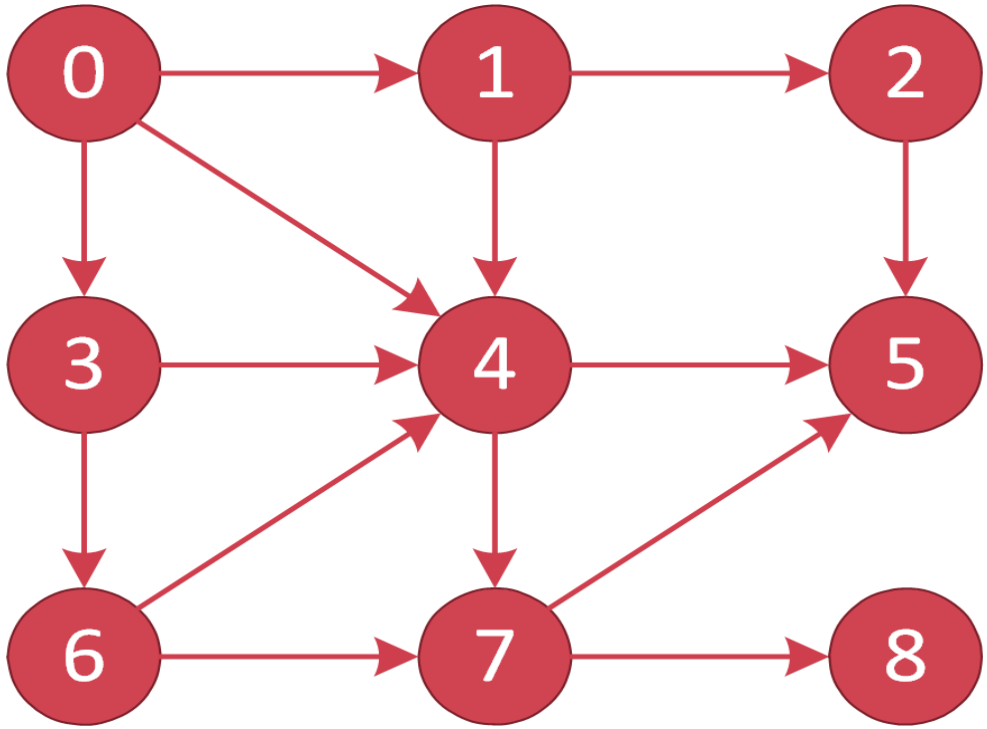
Delete vertex 5 from queue, and visit it.
TRAVERSAL : 0,1,3,4,2,6,5
Vertex 5 has no adjacent vertex.
QUEUE : 7



Delete vertex 7 from queue, and visit it.
TRAVERSAL : 0,1,3,4,2,6,7
Vertices adjacent to 7 is 8. Vertex 8 is in initial state, So it is inserted into queue.
QUEUE : 8

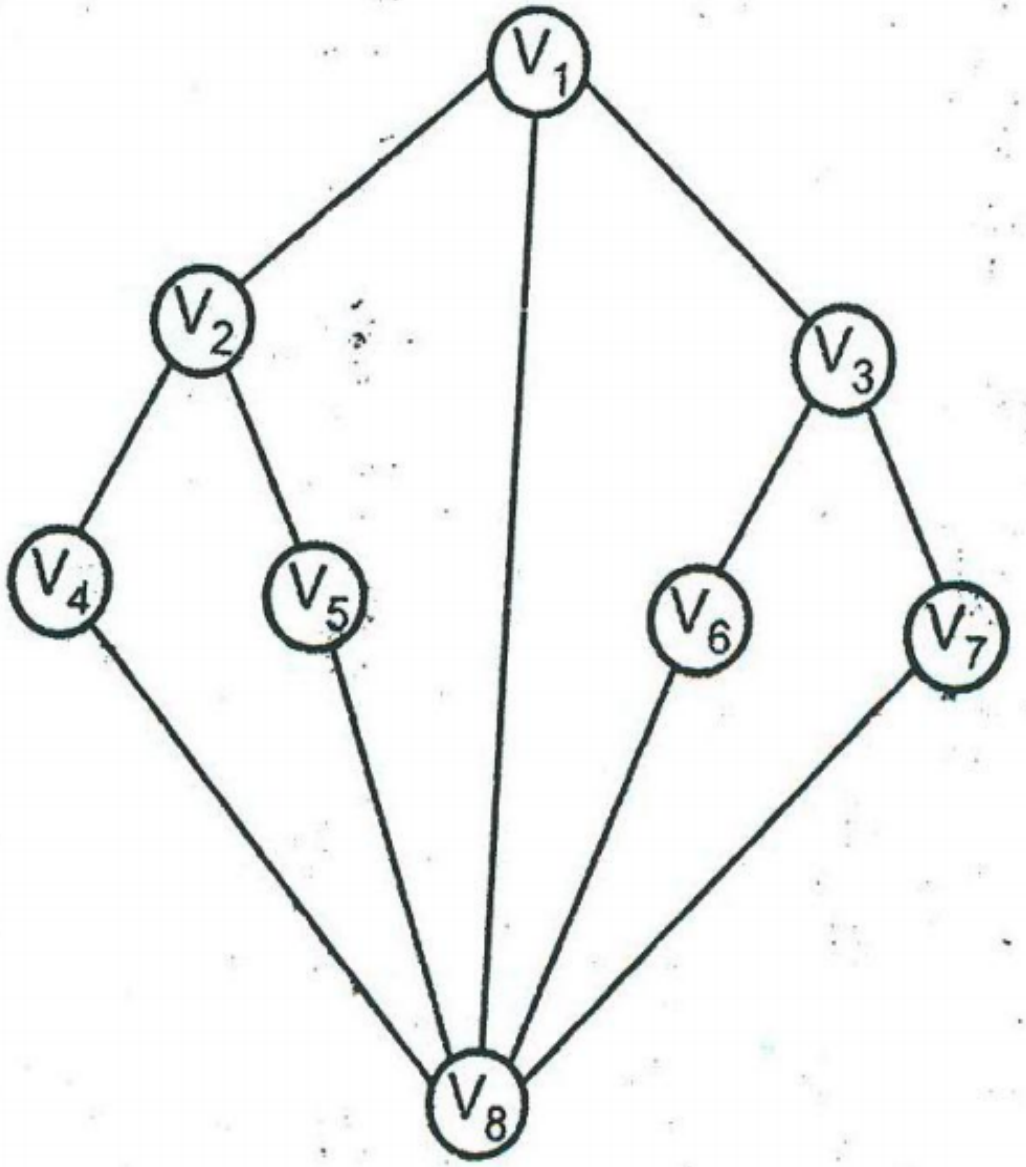


Delete vertex 8 from queue, and visit it.
TRAVERSAL : 0,1,3,4,2,6,7,8
Vertex 5 has no adjacent vertex. QUEUE : EMPTY

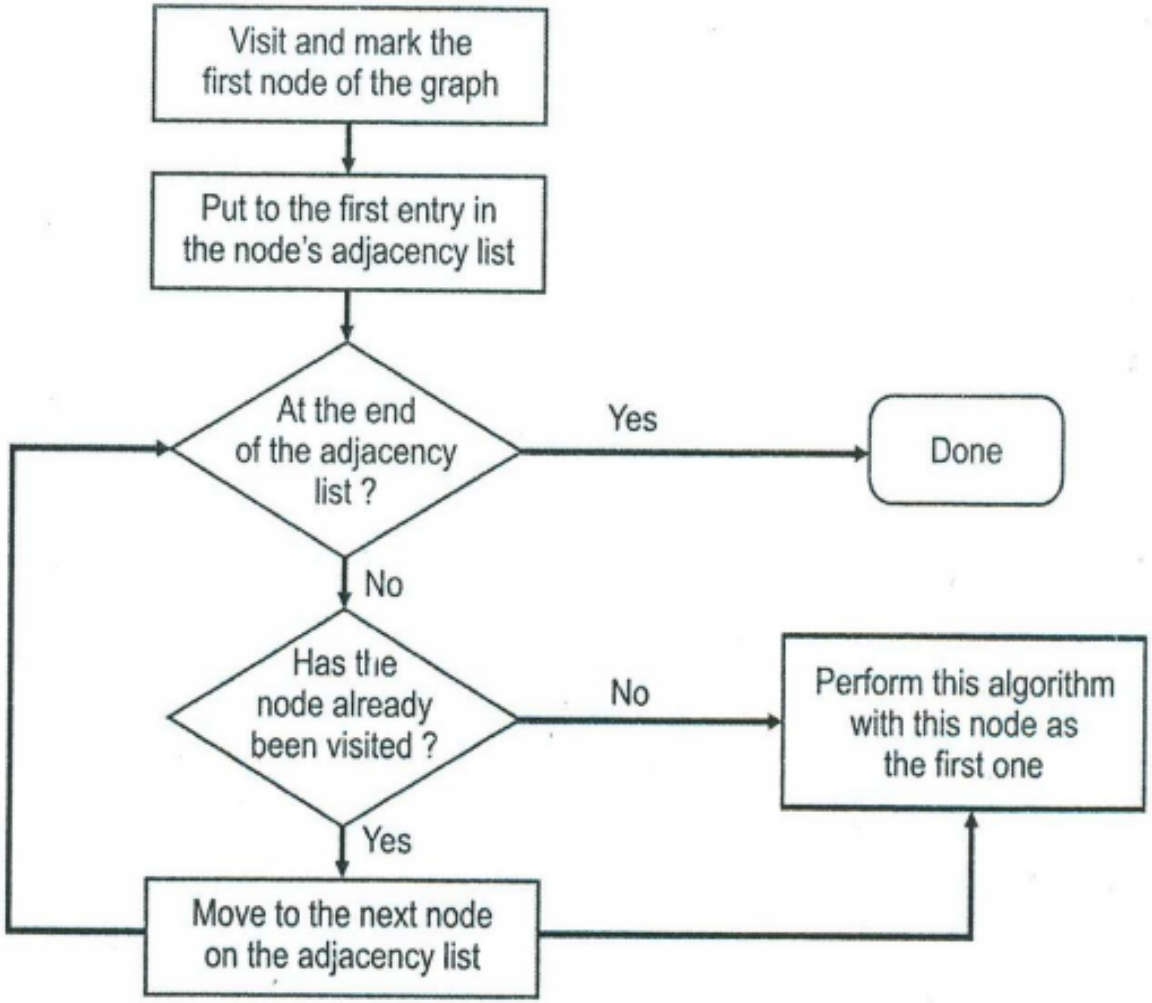


Applications

- Finding all the connected components in a graph G.
- Finding all the nodes within individual connected component.
- Finding the shortest path between two node u and v of an unweighted graph.
- Finding the shortest path between two node u and v of a weighted graph.


V1
V2
V3
V4
V5
V6
V7
V8

Depth First Search



DFS

- We Travel Along a path in the graph and when dead end comes we back track.
- Starting vertex is visited and we pick up any path in the graph that starts say(1) ad any vertex adjacent to it (say 2).
- Now if 2 has any vertex adjacent to it, which has not been visited then visit it and so on till dead end comes.

DFS

- *Dead End* : when we reach a Vertex which does not have any adjacent vertex or all of its adjacent vertex have been visited.
- After reaching the dead End we will back track along the path that we have visited till now. Move backwards till reach a vertex that has any unvisited adjacent vertex
- This procedure finishes when we reach the starting vertex and there are no vertices adjacent to it which have to be Visited.

Depth First Search

Algorithm for depth-first search in a graph G beginning at a starting node A

Step 1: SET STATUS = 1 (ready state) for each node in G.

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

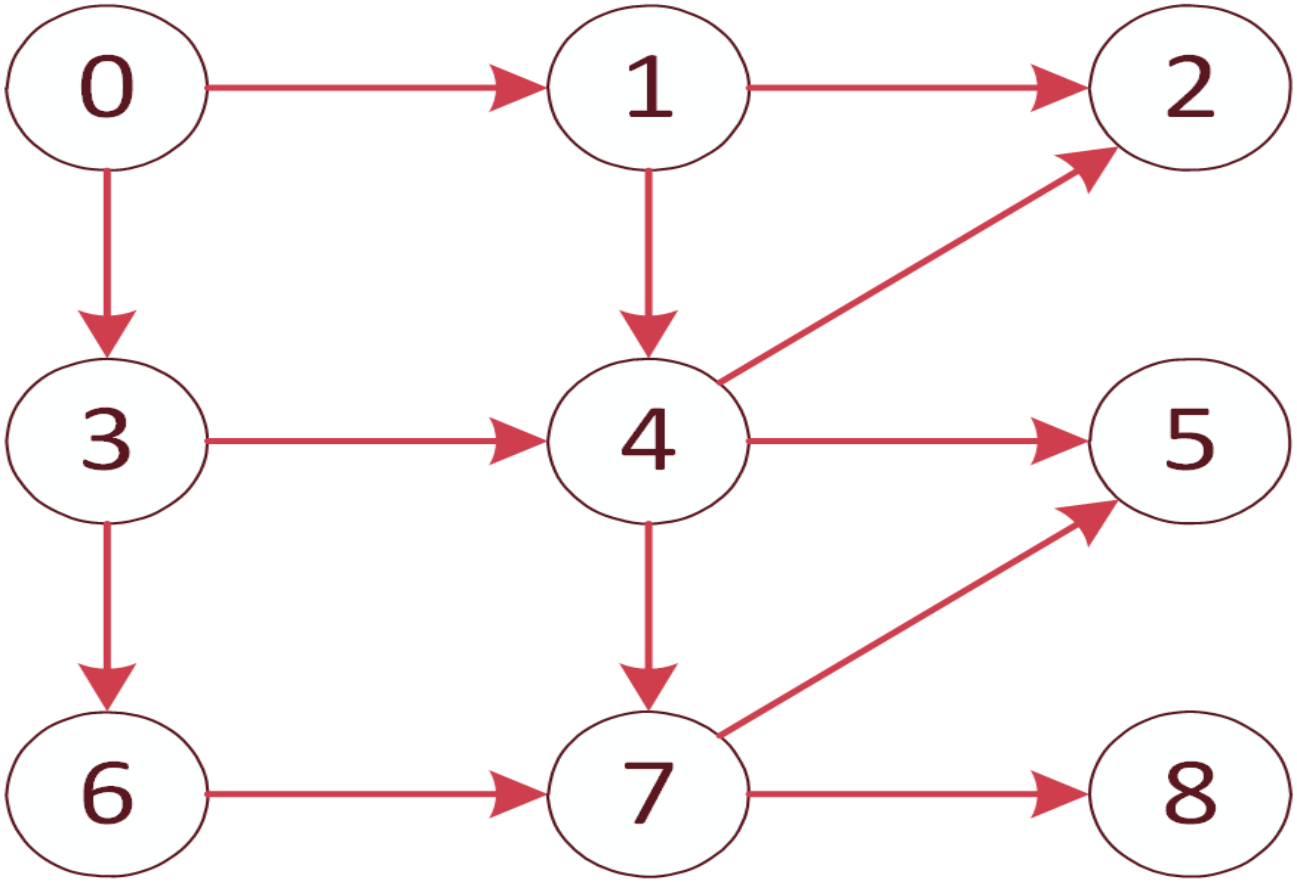
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state).

Step 5: Push on to the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

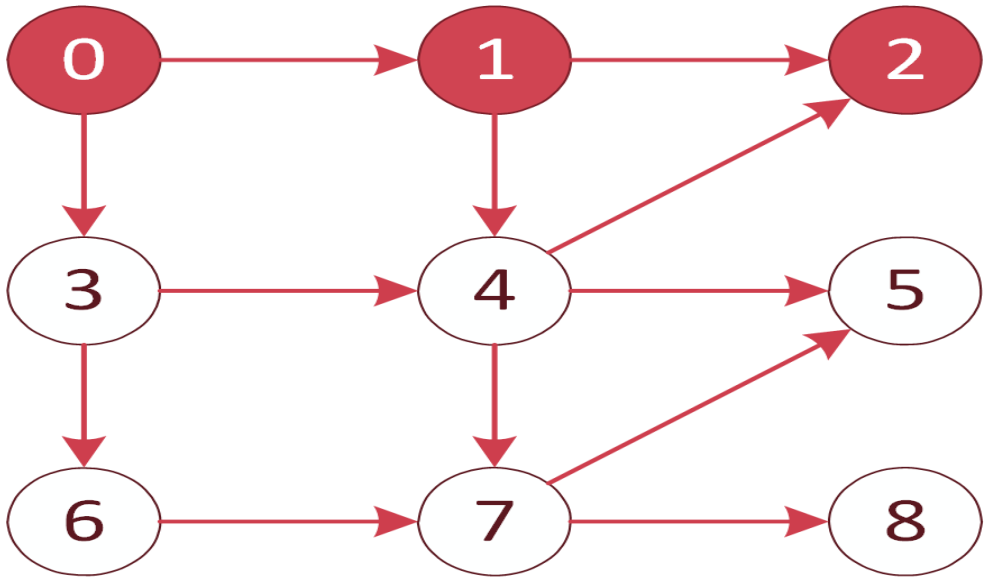
[END OF LOOP]

Step 6: EXIT

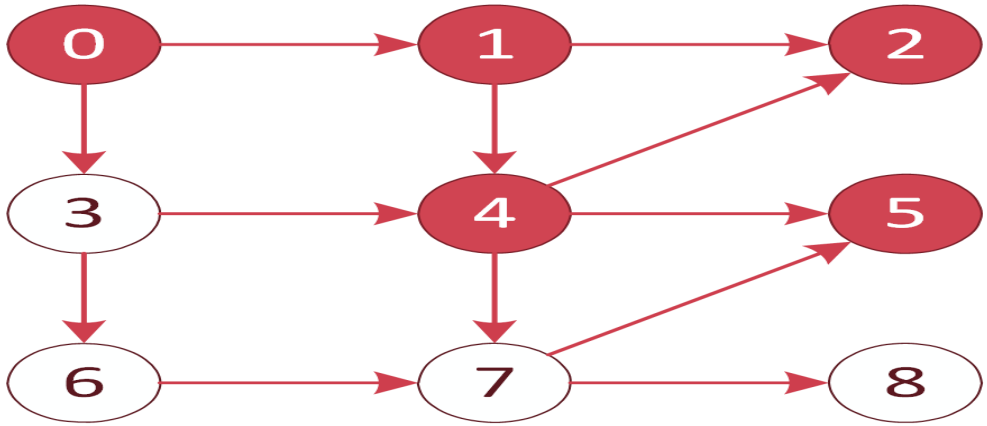
Depth First Search



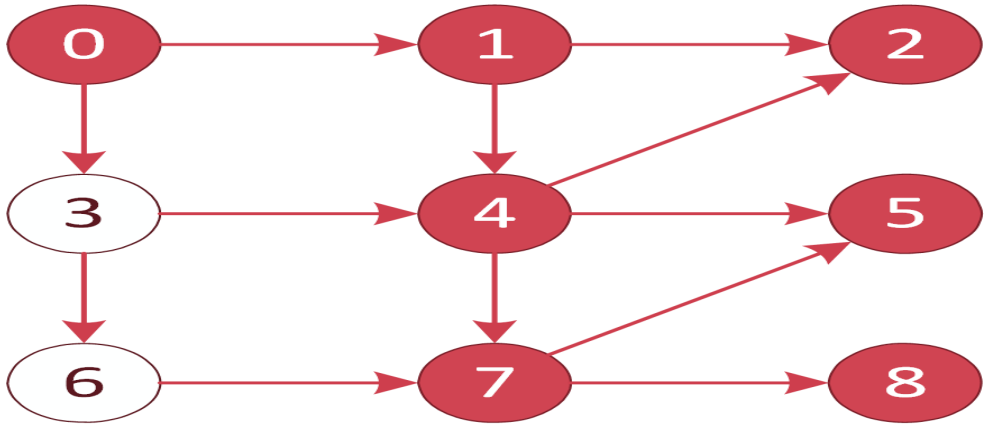
First we visit to vertex 0. Vertex 0 is adjacent to 1,3. Suppose we visit 1. Now we look adjacent of vertices 1. from the two adjacent vertices 2 and 4 we choose 2. till now the *Traversal* is : **0,1,2**

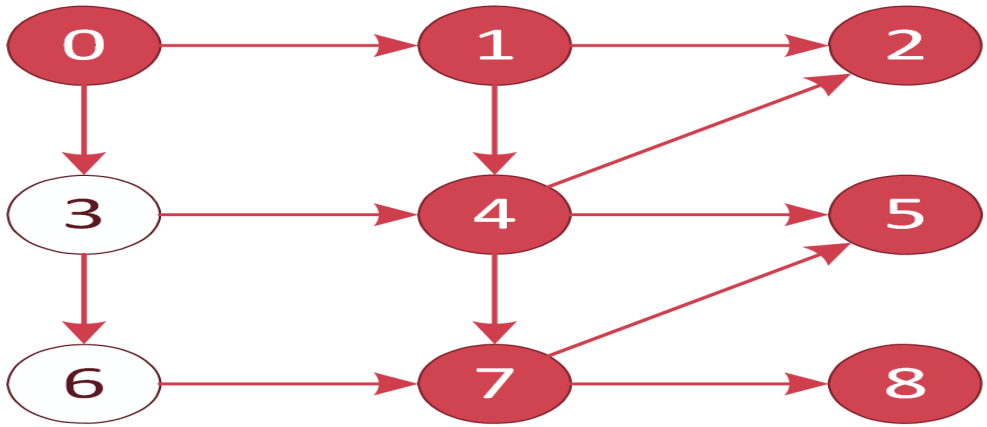


There is no adjacent of vertices 2, means we have reached the end of path or dead from where we can't go forward. So we move backward We reach vertex 1 and see if there is any vertex adjacent to it, and not yet visit. Vertex 4 is such a vertex and therefore we visit it. Now vertices 5 and 7 are adjacent to 4 and unvisited, and from we choose vertex 5. till now the *Traversal* is: **0,1,2,4,5**

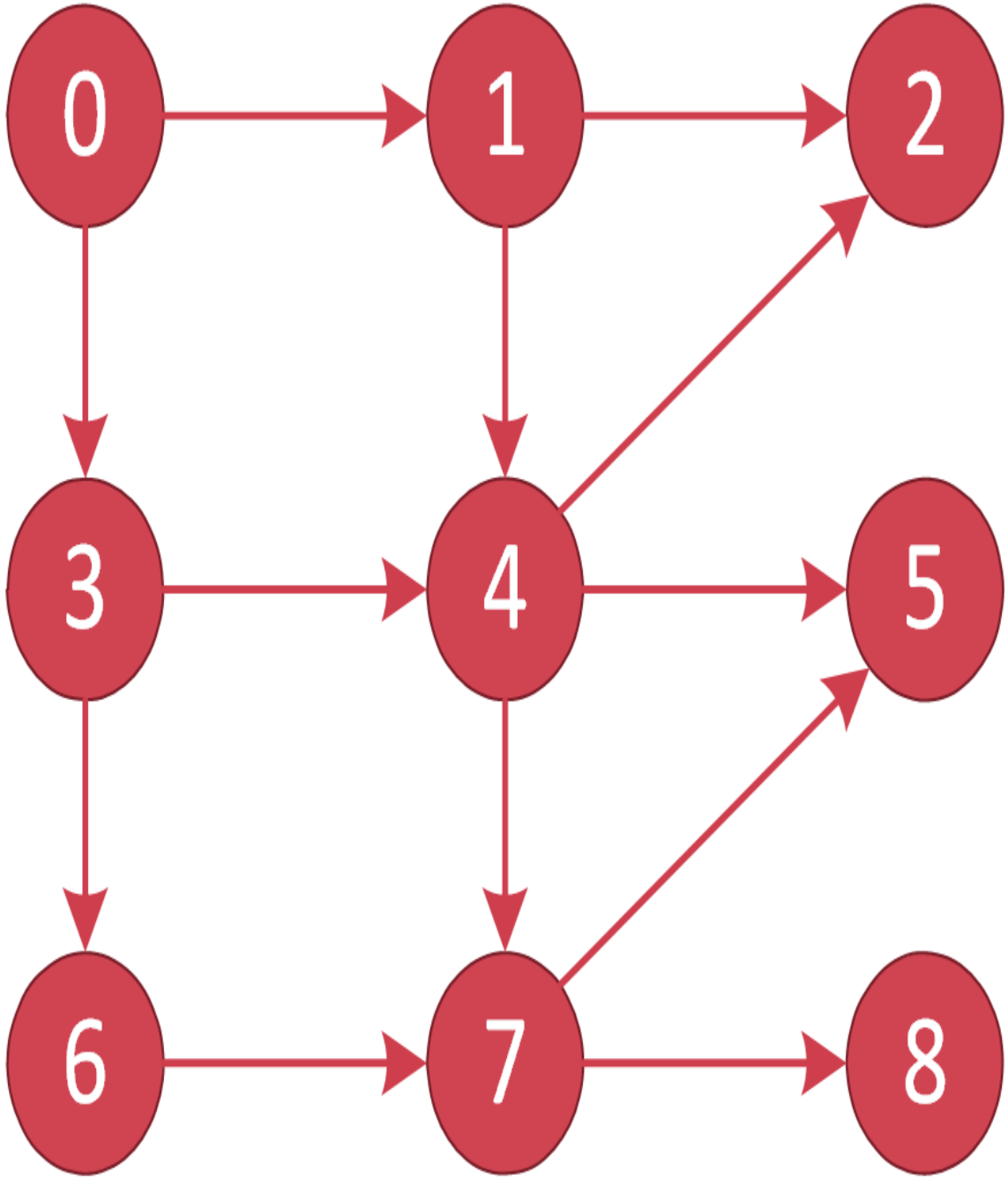


There is no vertex adjacent to 5 so we will backtrack. We reach vertex 4, and its unvisited vertex 7 so we visit it. Now 8 is the only unvisited vertex adjacent to 7 so we visit it. till now *traversal* is: **0,1,2,4,5,7,8**





Vertex 8 has no unvisited adjacent vertex so we backtrack and backtrack to 7. Now vertex 7 has also has no unvisited vertex and reach to vertex 4. Vertex 4 also has no unvisited vertex so we backtrack to 1. Vertex 1 also has no unvisited so we backtrack and reach to vertex 0. Vertex 3 is adjacent to vertex 0 and unvisited so we visit vertex 3. Vertex 6 is adjacent to vertex 3 and unvisited so we visit vertex 6. till now the traversal is:**0,1,2,4,5,7,8,3,6**


0
1
2
3
4
5
6
7
8

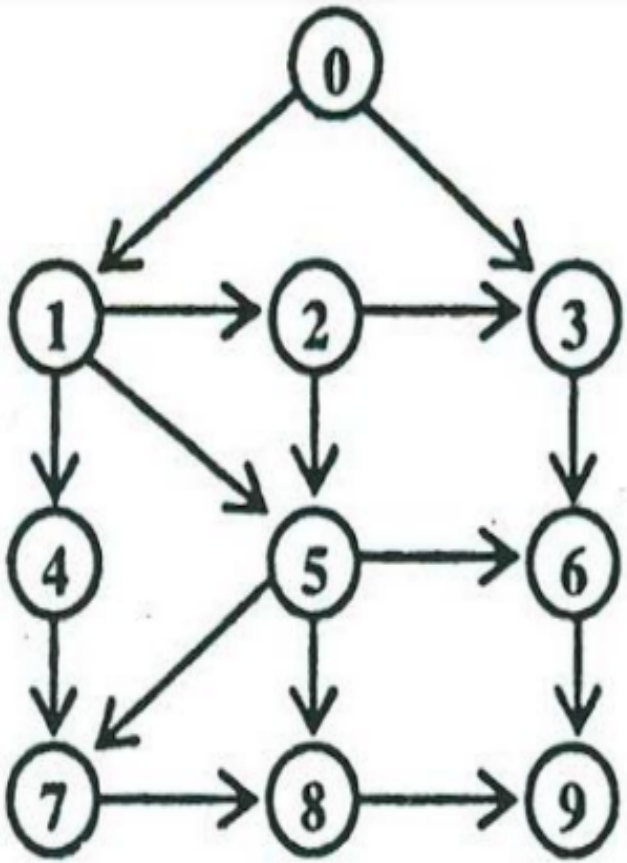
Now vertex 6 has no unvisited adjacent vertex so we will backtrack and reach to vertex 3. Vertex 3 has no unvisited adjacent vertex so we will backtrack and rech to vertex 3. vertex 0 has no unvisited adjacent vertex, Now can not backtrack.
so our traversal is finish here. So final *traversal* is:- **0,1,2,4,5,7,8,3,6**

Implementation of DFS

During the algorithm any vertex will be in one of two state- initial or visited. At the start of the algorithm, all vertices will be in initial state, and when a vertex will be popped from stack its state will change to visited. The procedure is as-
Initially stack is empty. and all vertices are in initial state.

- Push starting vertex on the stack.
- Pop a vertex from the stack.
- If popped vertex is in initial state, visit it and change its state to visited. Push all unvisited vertices adjacent to the popped vertex.
- Repeat steps 2 and 3 until stack is empty.

Implementation of DFS



Implementation of DFS

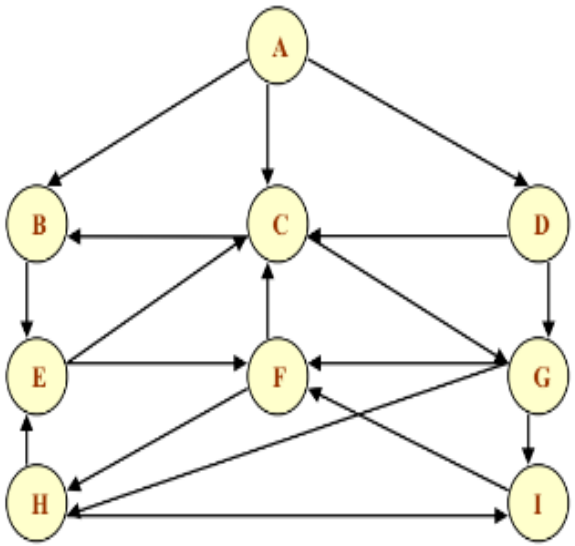
Start vertex is 0, so initially push vertex 0 on the stack.

Pop 0: Visit 0	Push 3, 1	Stack : 3 1
Pop 1: Visit 1	Push 5, 4, 2	Stack : 3 5 4 2
Pop 2: Visit 2	Push 5, 3	Stack : 3 5 4 5 3
Pop 3: Visit 3	Push 6	Stack : 3 5 4 5 6
Pop 6: Visit 6	Push 9	Stack : 3 5 4 5 9
Pop 9: Visit 9		Stack : 3 5 4 5
Pop 5: Visit 5	Push 8, 7	Stack : 3 5 4 8 7
Pop 7: Visit 7	Push 8	Stack : 3 5 4 8 8
Pop 8: Visit 8		Stack : 3 5 4 8
Pop 8:		Stack : 3 5 4
Pop 4: Visit 4		Stack : 3 5
Pop 5:		Stack : 3
Pop 3:		Stack : Empty

Depth first traversal is : 0 1 2 3 6 9 5 7 8 4

Depth First Search Algorithm

Example: Consider the graph G given below. The adjacency list of G is also given. Suppose we want to print all nodes that can be reached from the node H (including H itself). One alternative is to use a Depth-First Search of G starting at node H. the procedure can be explained as below.



Adjacency Lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Implementation of DFS

a)Push H on to the stack

STACK: H

Pop and Print the top element of the STACK, that is, H. Push all the neighbors of H on to the stack that are in the ready state. The STACK now becomes:

STACK: E, I

Pop and Print the top element of the STACK, that is, I. Push all the neighbors of I on to the stack that are in the ready state. The STACK now becomes:

PRINT: I

STACK: E, F

Pop and Print the top element of the STACK, that is, F. Push all the neighbors of F on to the stack that are in the ready state. (Note F has two neighbors C and H. but only C will be added as H is not in the ready state). The STACK now becomes:

PRINT: F

STACK: E, C

Pop and Print the top element of the STACK, that is, C. Push all the neighbors of C on to the stack that are in the ready state. The STACK now becomes:

Implementation of DFS

PRINT: C

STACK: E, B, G

Pop and Print the top element of the STACK, that is, G. Push all the neighbors of G on to the stack that are in the ready state. Since there are no neighbors of G that are in the ready state no push operation is performed. The STACK now becomes:

PRINT G:

STACK: E, B

Pop and Print the top element of the STACK, that is, B. Push all the neighbors of B on to the stack that are in the ready state. Since there are no neighbors of B that are in the ready state no Push operation is performed. The STACK now becomes:

PRINT e:

STACK: E

Pop and Print the top element of the STACK, that is, E. Push all the neighbors of E on to the stack that are in the ready state. Since there are no neighbors of E that are in the ready state no Push operation is performed. The STACK now becomes empty: PRINT: E. Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are-H, I, F, C, G, B E. These are the nodes which are reachable from the node H.