Graphs

sushmakadge@somaiya.edu

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Graph- Concept
- Graph terminology: vertex, edge, adjacent, incident, degree, cycle, path, connected component, spanning tree
- Types of graphs: undirected, directed, weighted
- Graph representations: adjacency matrix, array adjacency lists, linked adjacency lists
- Graph search methods: breath-first, depth-first search

Somaiya TRUST

Graphs

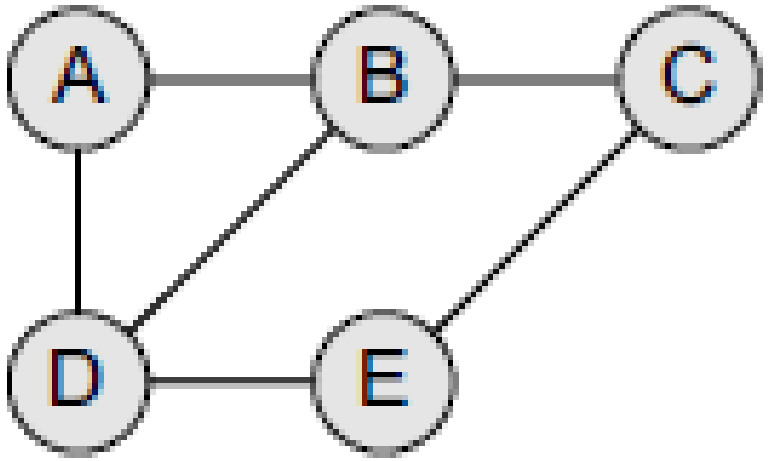
- A graph is an abstract data structure that is used to implement the mathematical concept of graphs.
- It is basically a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

Why are Graphs 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.

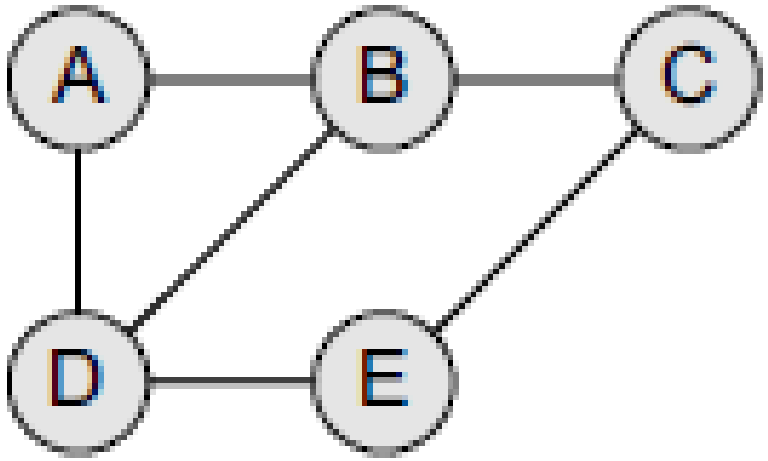
Definition

- A graph G is defined as an ordered set (V, E), where
- V(G) represents the set of vertices
- E(G) represents the edges that connect these vertices.
- Figure 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 five vertices or nodes and six edges in the graph.



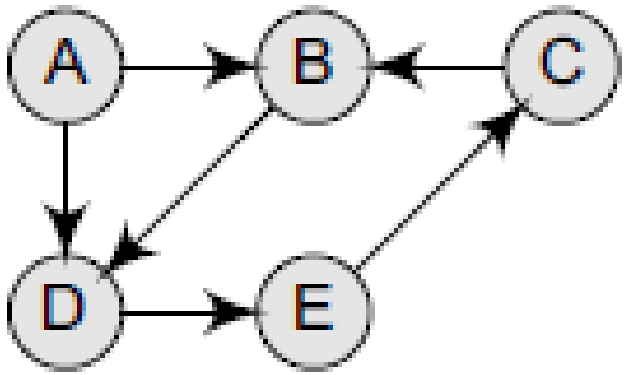
Definition

- A graph can be directed or undirected.
- In an undirected graph, 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.
- Figure shows an undirected graph because it does not give any information about the direction of the edges.



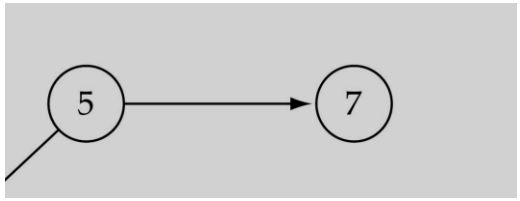
Definition

- Look at Fig. which 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).



Graph Terminology

- Adjacent nodes or neighbors: two nodes are adjacent if they are connected by an edge. 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.

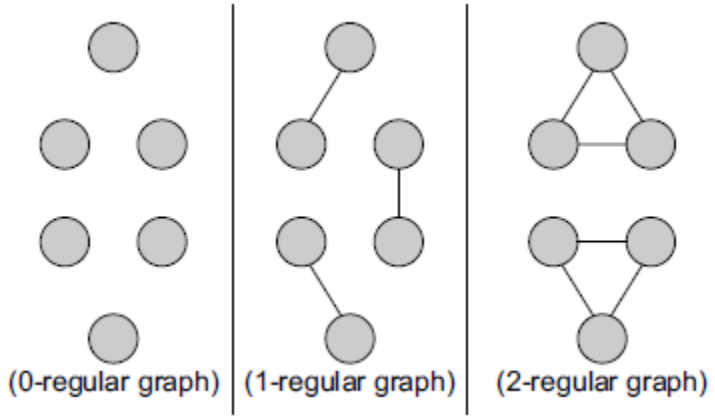


5 is adjacent to 7
7 is adjacent from 5

- Path: a sequence of vertices that connect two nodes in a graph

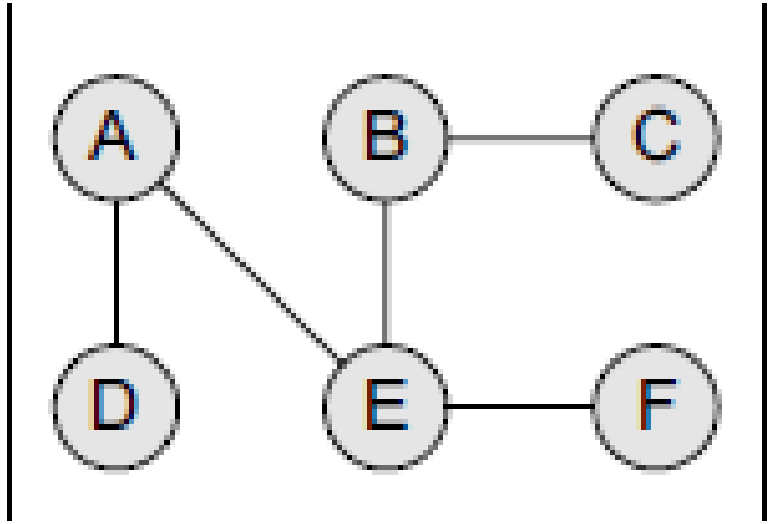
Graph Terminology

- ***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*** It 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 a regular graph of degree k. Figure shows regular graphs.



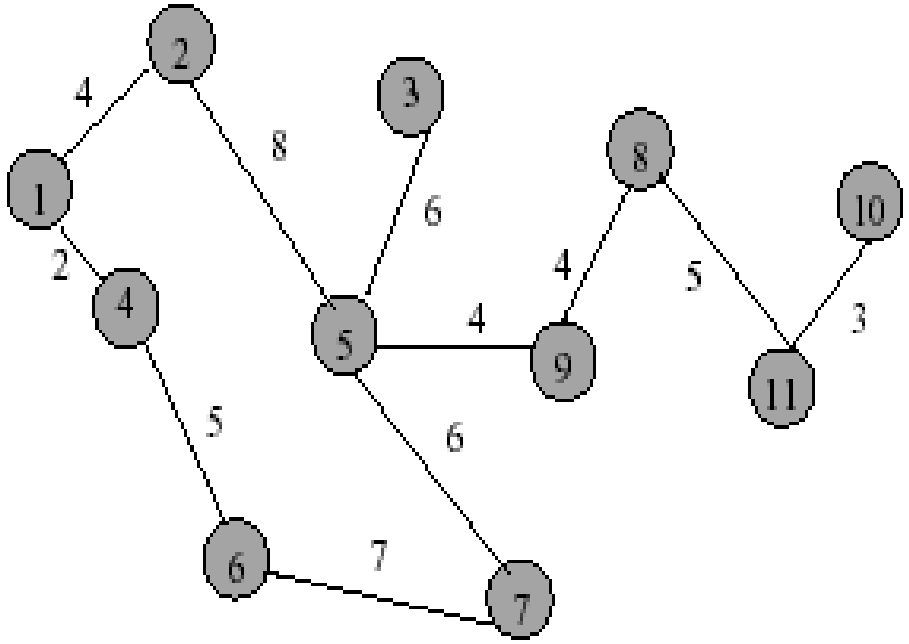
Graph Terminology

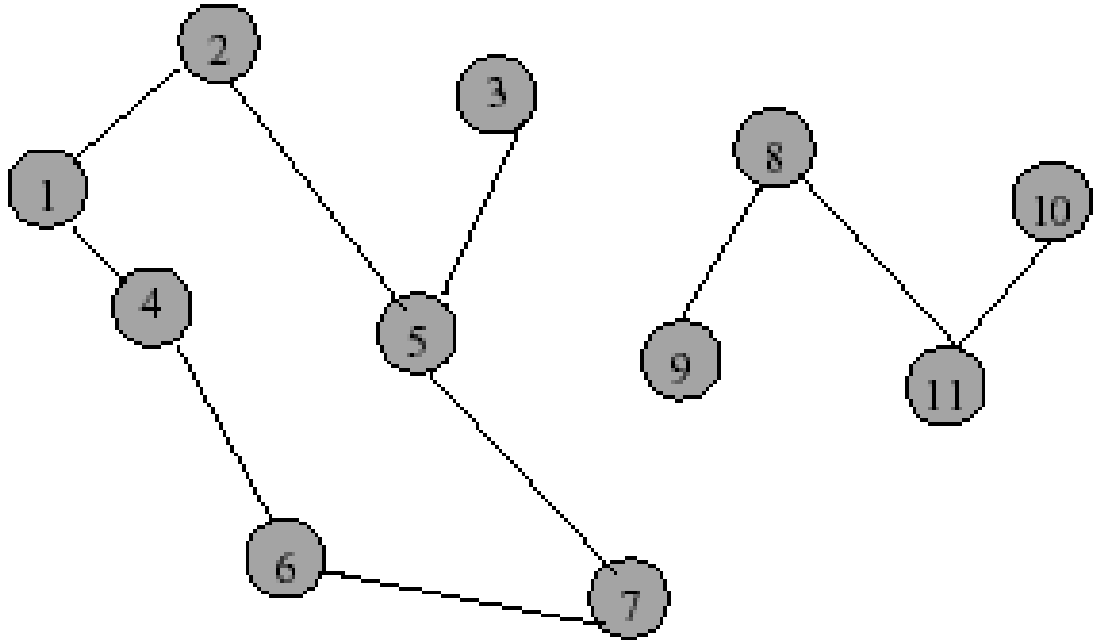
- ***Cycle*** A path in which the first and the last vertices are same.
- ***Connected graph*** A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph.
- A tree is treated as a special graph (Refer Fig.).



Graph Terminology

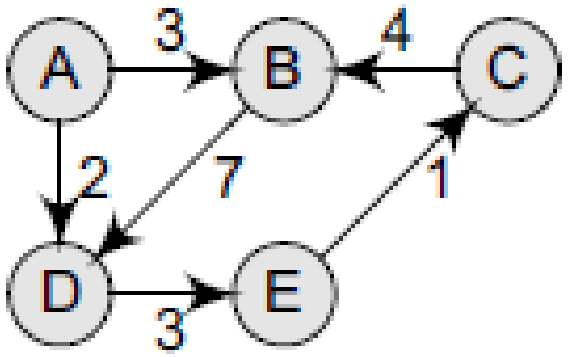
Example of Connected and Not Connected





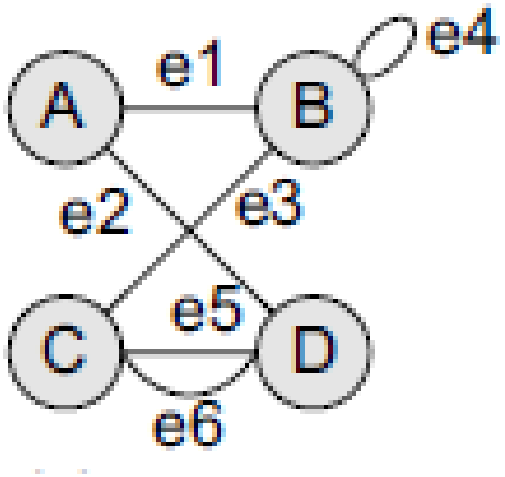
Graph Terminology

- ***Labelled graph or weighted graph*** A graph is said to be labelled 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. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge. Figure shows a weighted graph.



Graph Terminology

- ***Multi-graph*** A graph with multiple edges and/or loops is called a multi-graph. Figure shows a multi-graph.

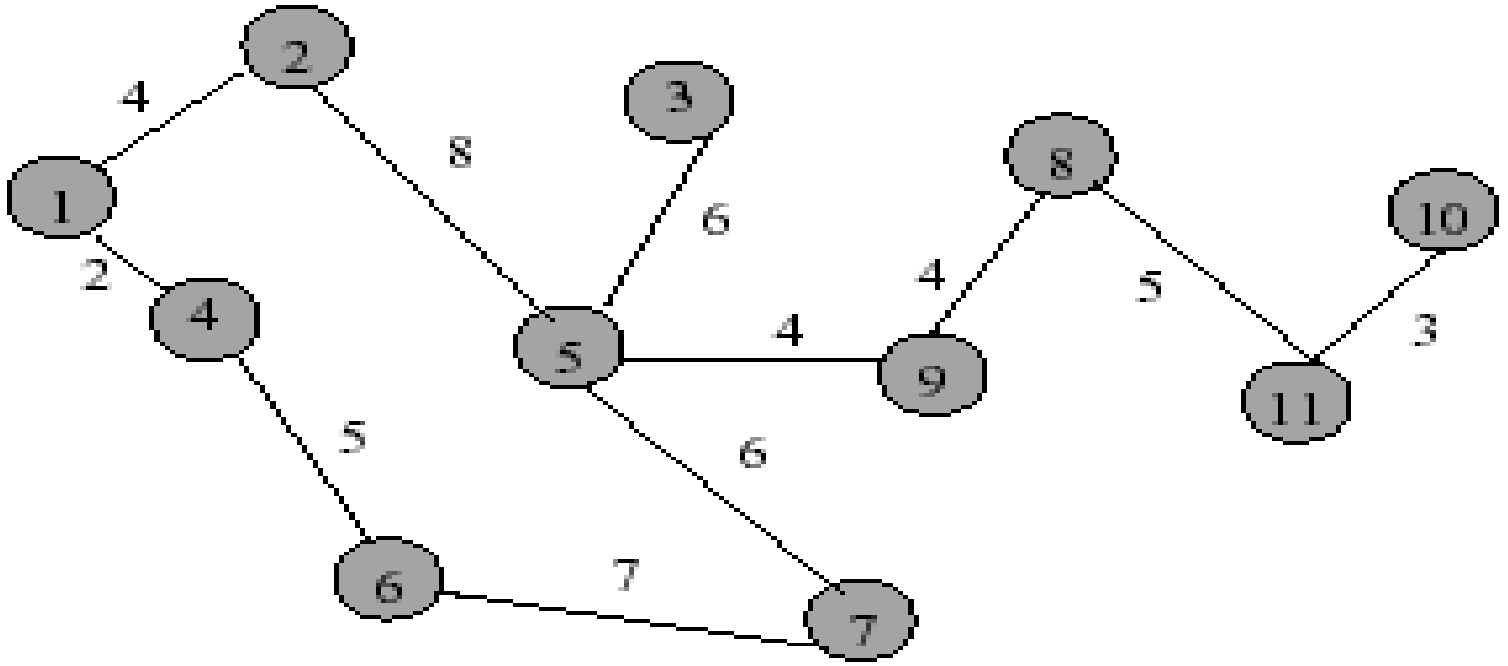


Graph Terminology

- ***Complete graph*** : A graph in which every vertex is directly connected to every other vertex. A graph G is said to be 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.

Applications

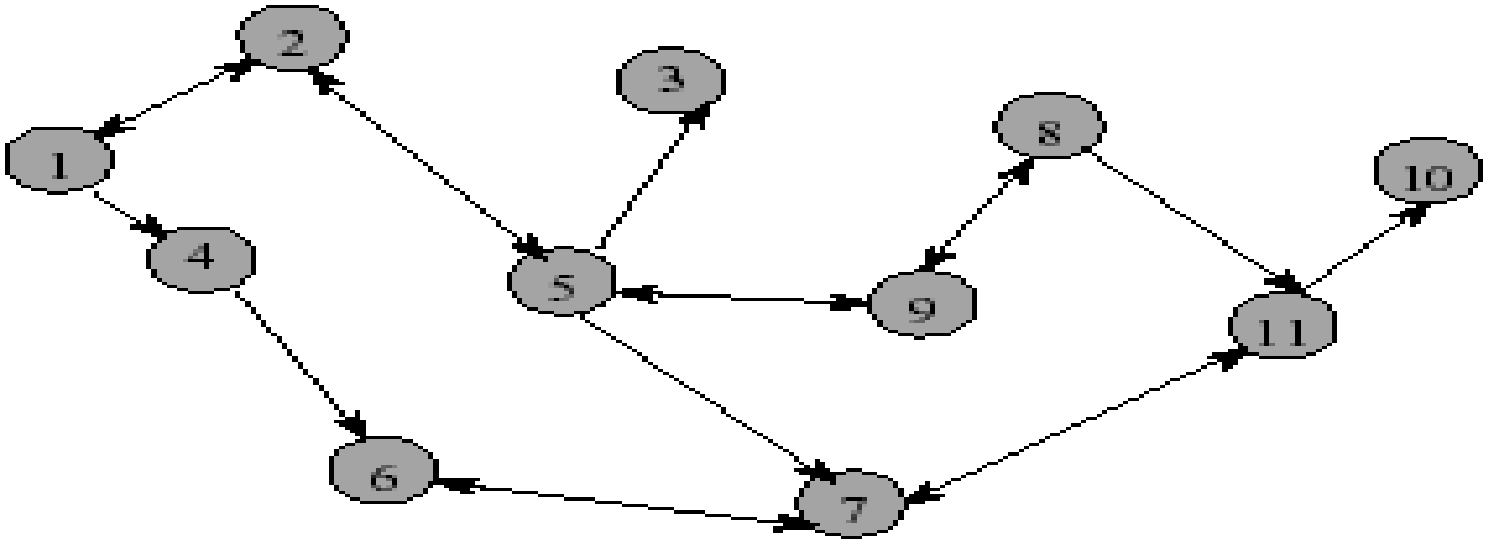
- Driving Distance/Time Map



vertex = city
edge weight = driving distance/time

Applications

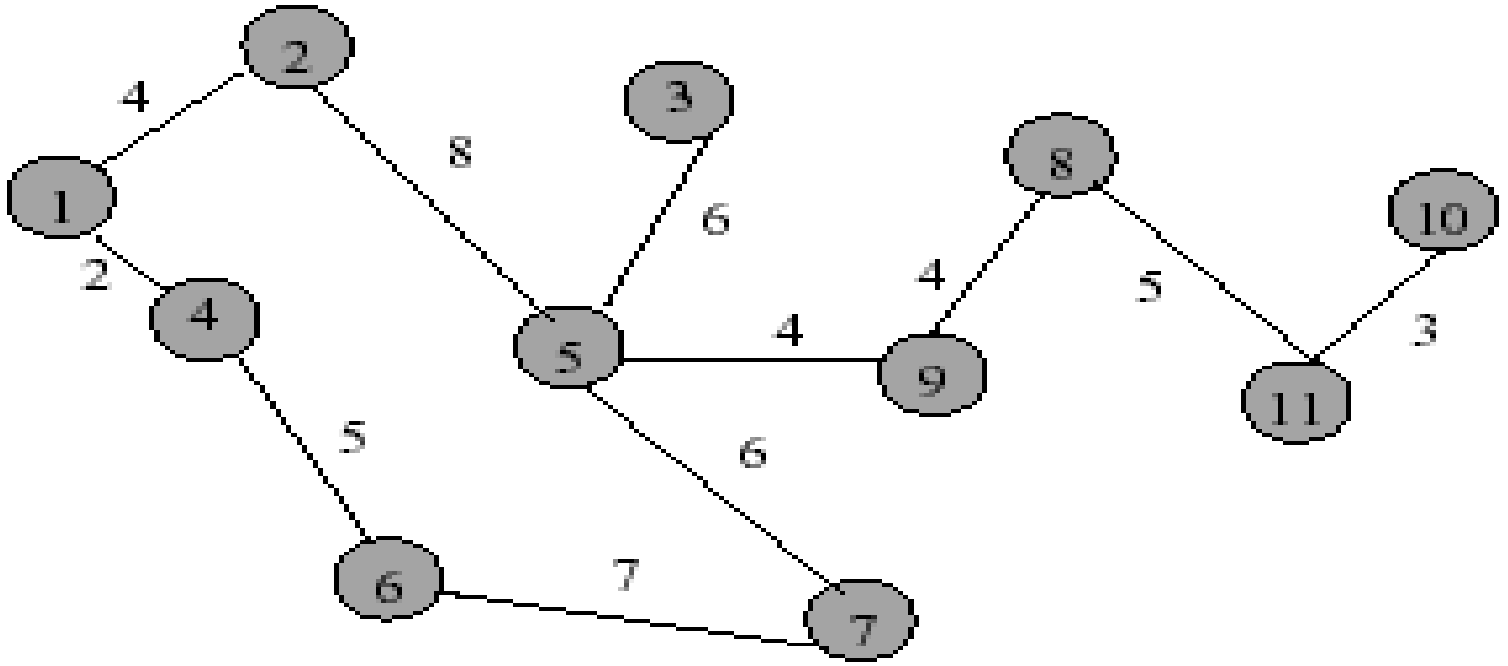
- Street Map



•Streets are one- or two-way.
• A single directed edge denotes a one-way street
• A two directed edge denotes a two-way street

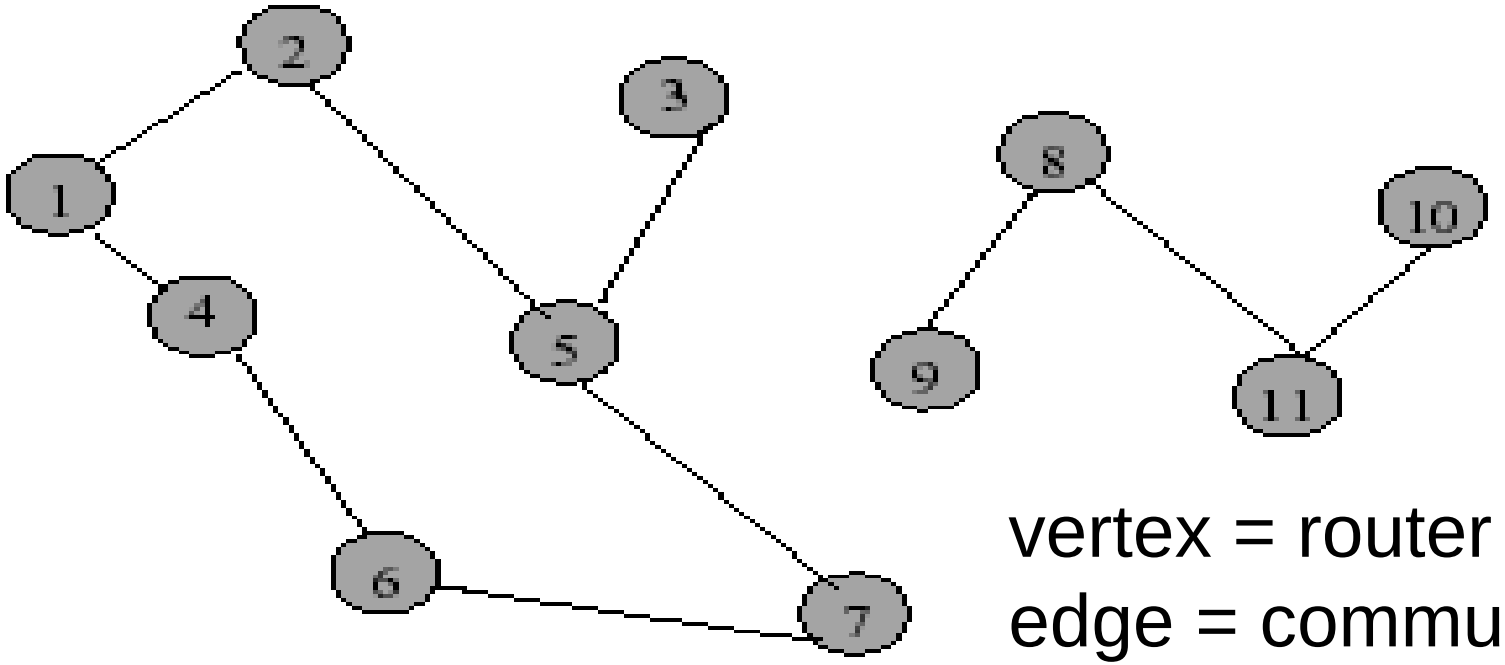
Applications

- Driving Distance/Time Map



vertex = city
edge weight = driving distance/time

Applications – Communication Network



vertex = router
edge = communication link

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*** The 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).

REPRESENTATION OF GRAPHS

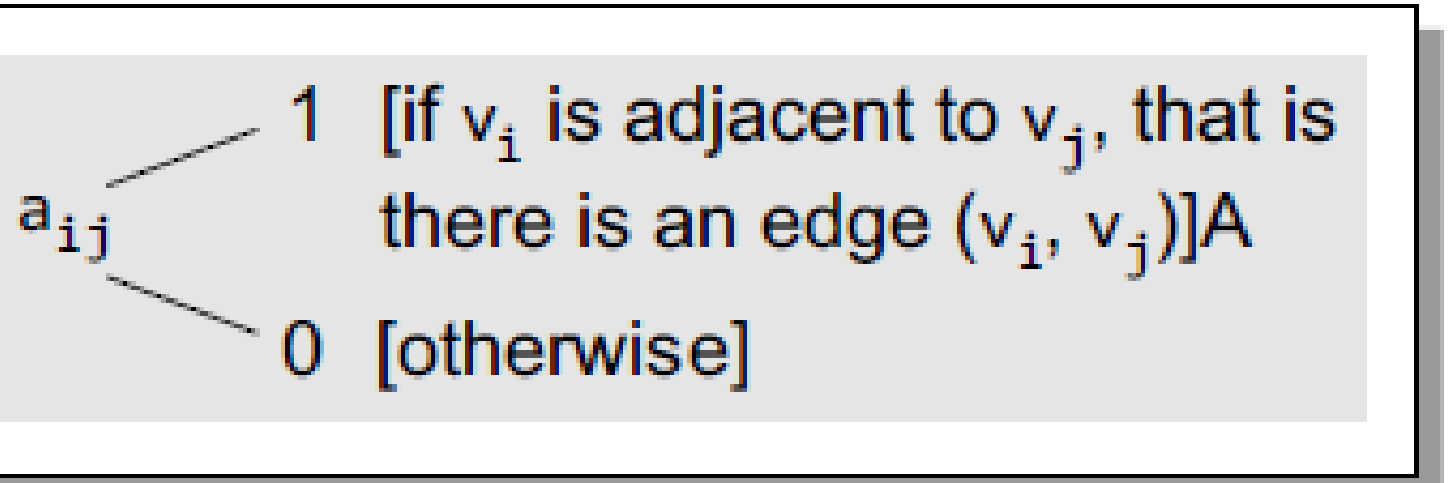
- Three common ways of storing graphs in the computer's memory.
- *Sequential representation* by using an adjacency matrix.
- *Linked representation* by using an adjacency list that stores the neighbor's of a node using a linked list.
- *Adjacency multi-list* which is an extension of linked representation

Adjacency Matrix Representation

- An adjacency matrix is used to represent which nodes are adjacent to one another.
- By definition, two nodes are said to be adjacent if there is an edge connecting them.
- For any graph G having n nodes, the adjacency matrix will have the dimension of n *n.

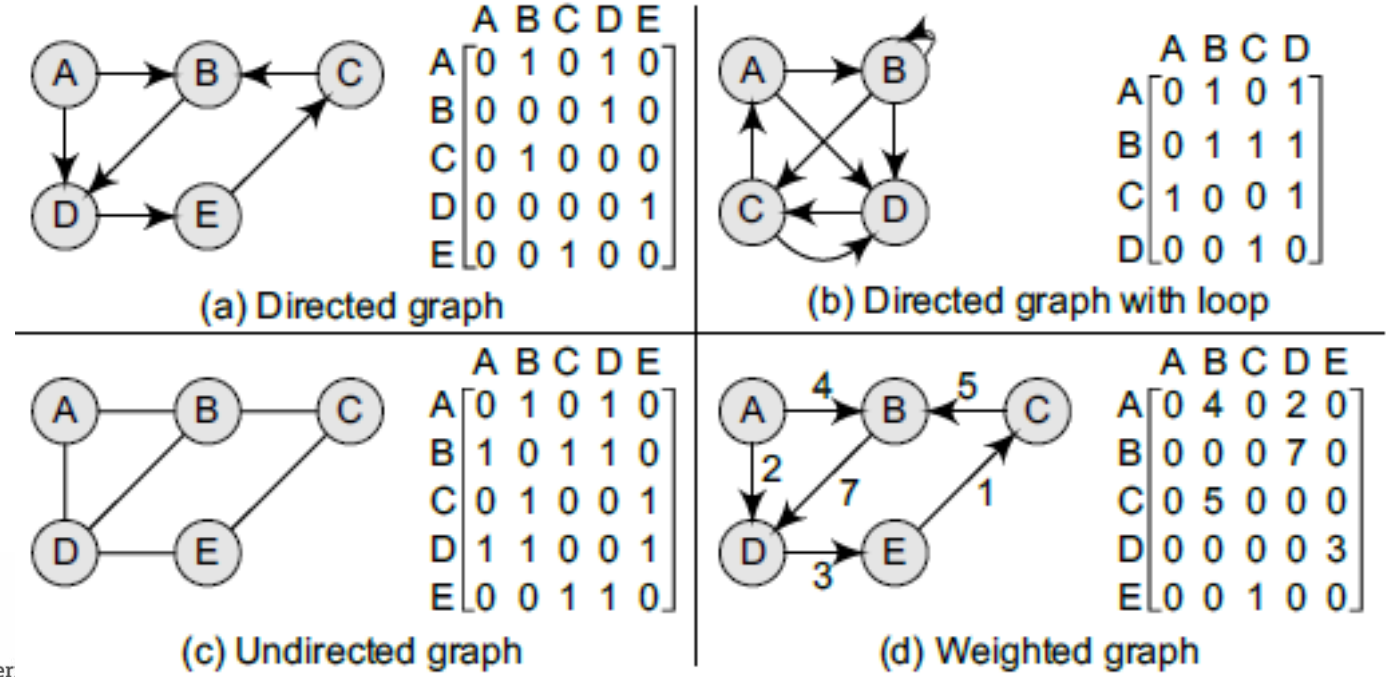
Adjacency Matrix Representation

- In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero. It is summarized in Fig.



Adjacency Matrix Representation

- 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. Fig. shows graphs and their corresponding adjacency matrices.



Adjacency Matrix Representation

- we can draw the following conclusions:
- A simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Adjacency List Representation

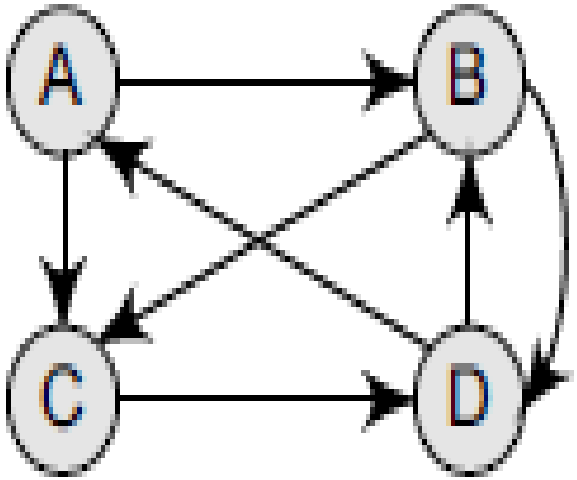
- An adjacency list is another way in which graphs can be represented in the computer's memory.
- This structure consists of a list of all nodes in G. Every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

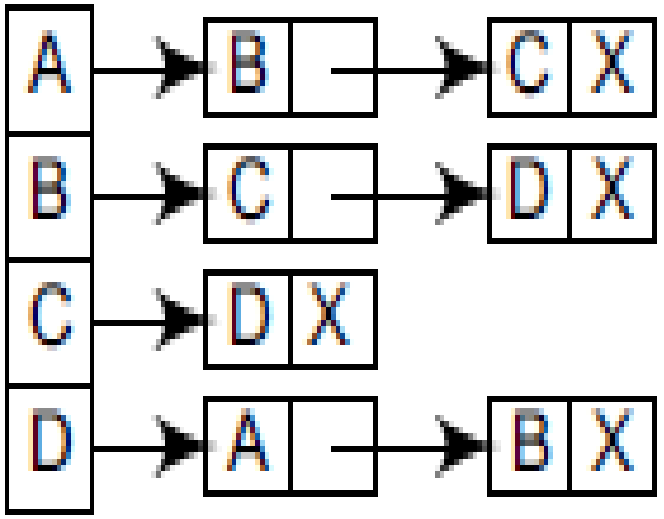
Adjacency List Representation

- The key advantages of using an adjacency list are:
- 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. i.e. an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

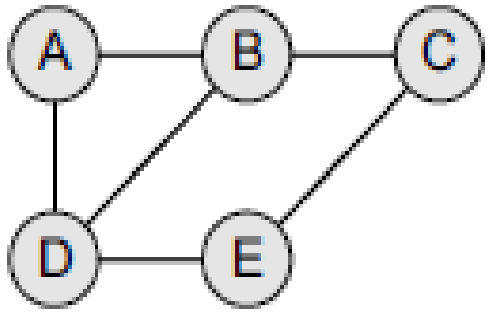
Adjacency List Representation

- Consider the graph given in Fig. and see how its adjacency list is stored in the memory. For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G. However, for an undirected graph, the sum of the 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 from v to u. Adjacency lists can also be modified to store weighted graphs.

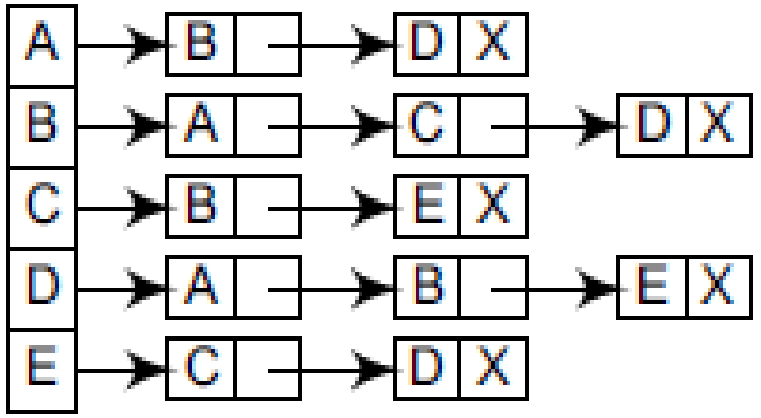


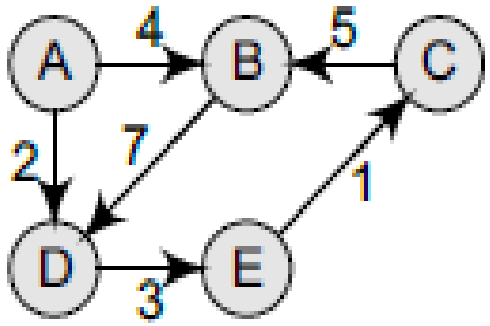


Adjacency List Representation

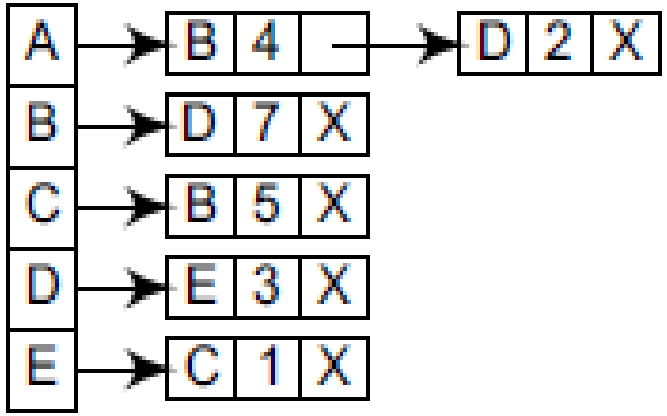


(Undirected graph)





(Weighted graph)

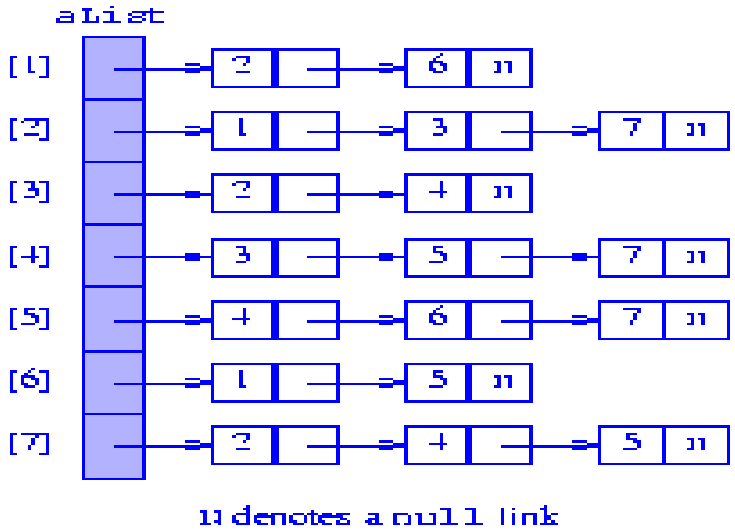


Representation

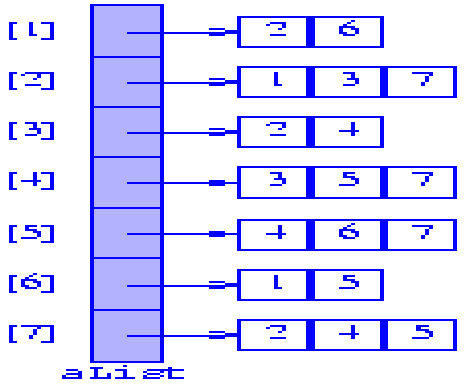
- (a) adjacency matrix

	1	2	3	4	5	6	7
1	0	1	0	0	0	1	0
2	1	0	1	0	0	0	1
3	0	1	0	1	0	0	0
4	0	0	1	0	1	0	1
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	1	0	1	1	0	0

(b) Linked adjacency list



(c) Array adjacency list



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

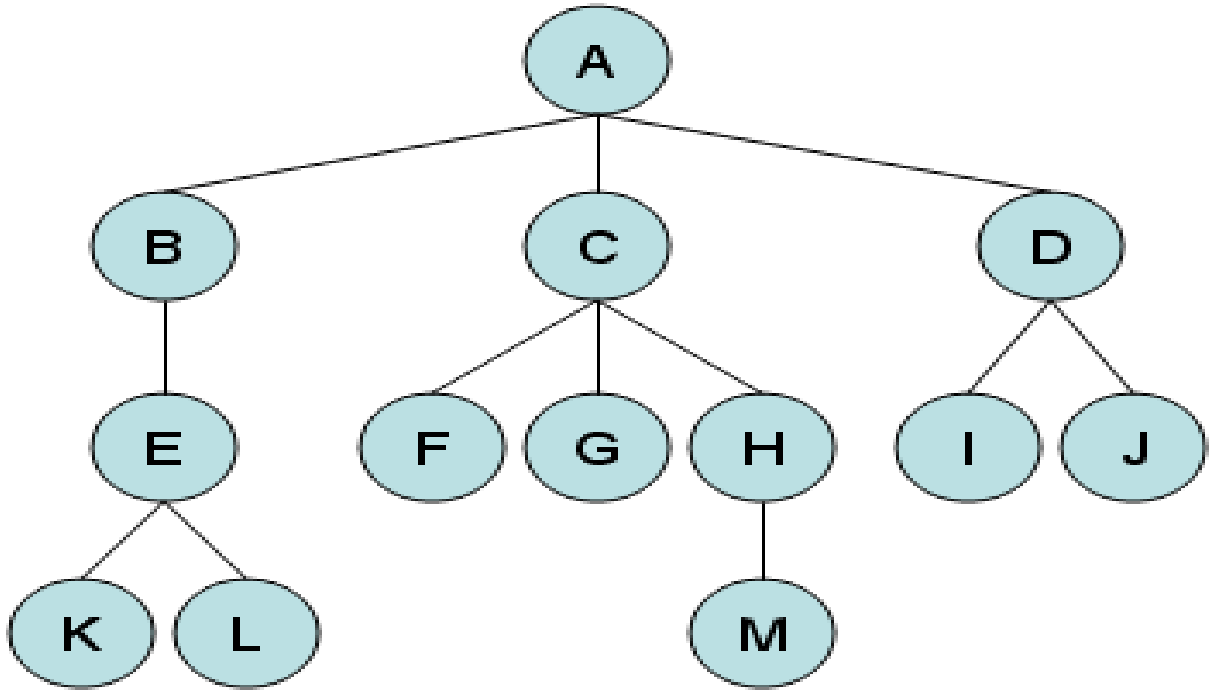
Somaiya TRUST

GRAPH TRAVERSAL ALGORITHMS

- Two methods are:
    1. Breadth-first search(BFS)
    2. Depth-first search (DFS)

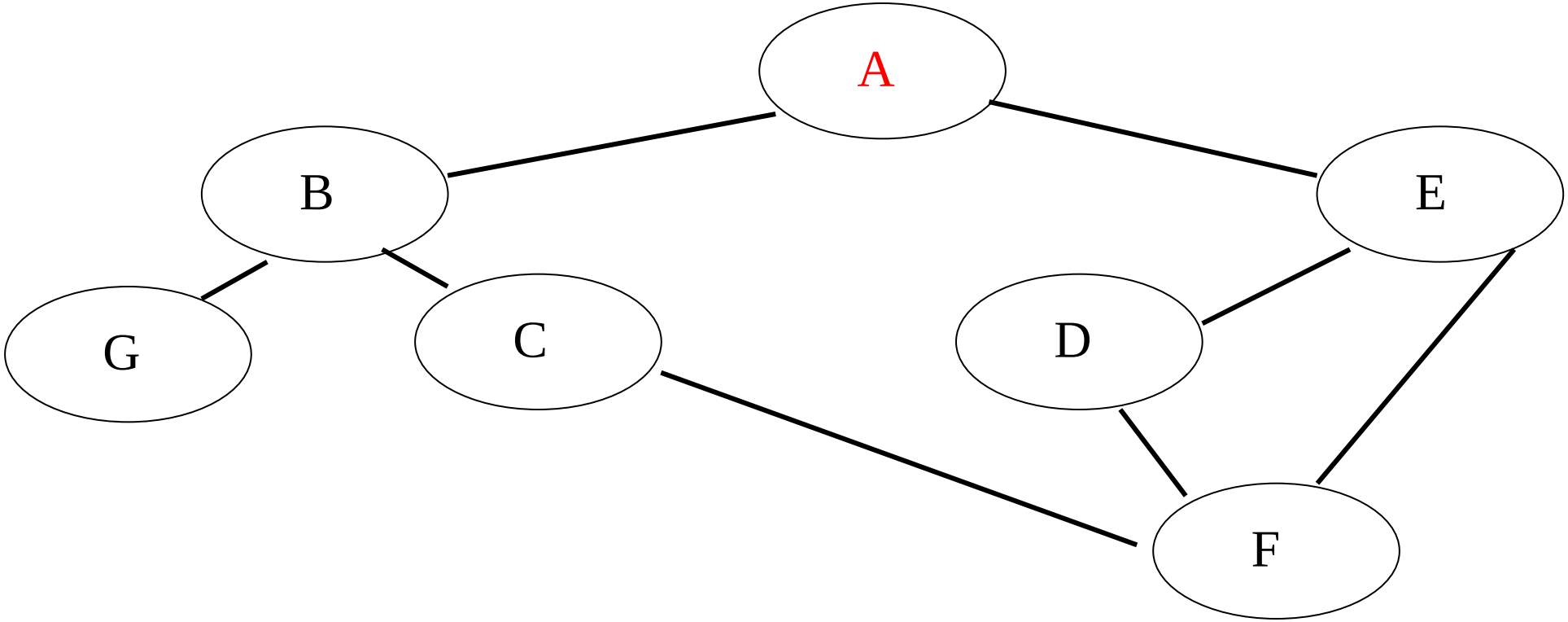
BFS in a tree

BFS: visit all siblings before their descendants



A B C D E F G H I J K L M

BFS: Graph



A B E G C D F

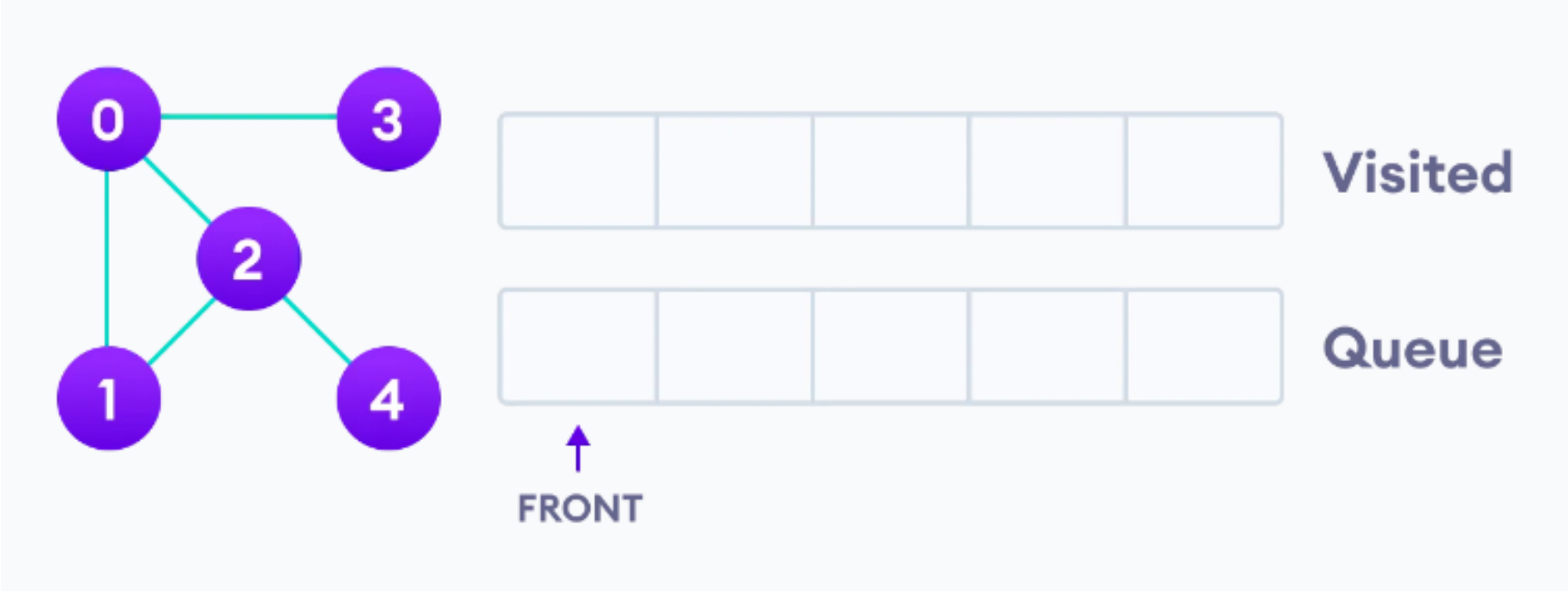
GRAPH TRAVERSAL ALGORITHMS

- While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

Breadth-first search(BFS)

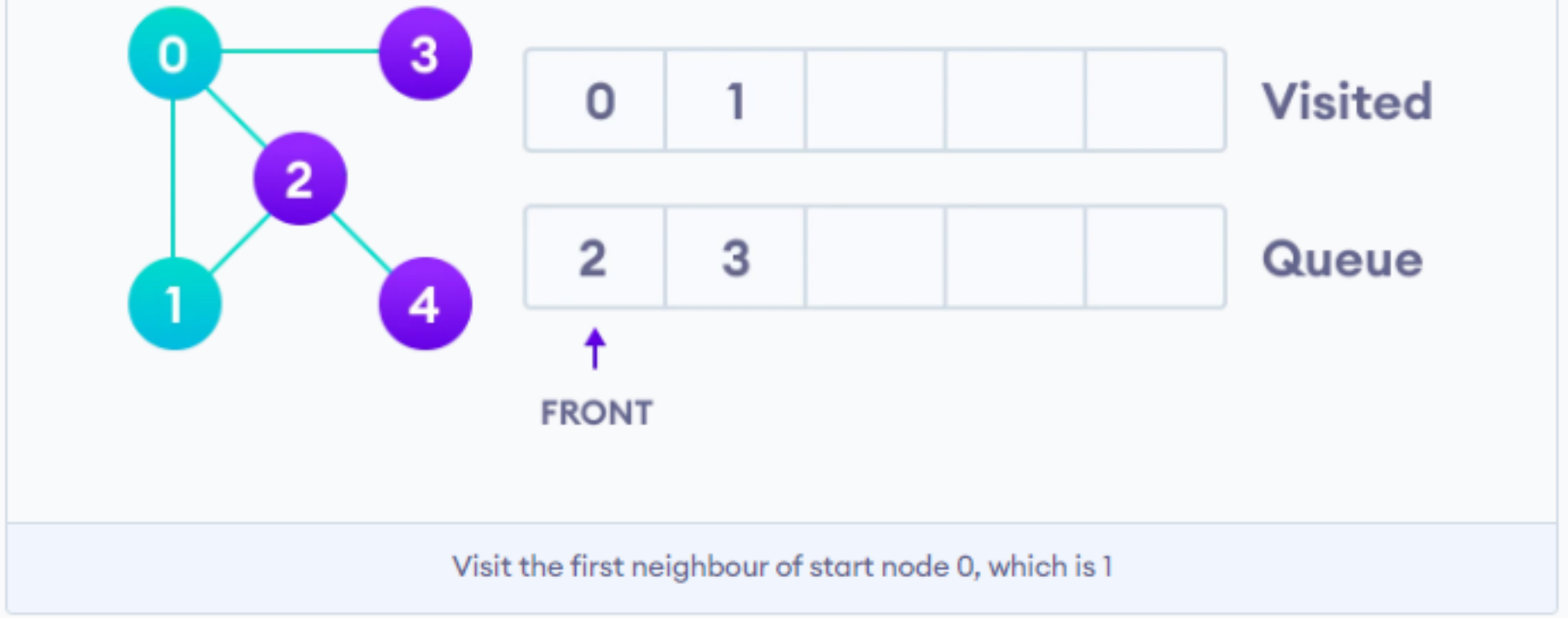
- A standard BFS implementation puts each vertex of the graph into one of two categories:
- Visited
- Not Visited
- The algorithm works as follows:
- Start by putting any one of the graph's vertices at the back of a queue.
- Take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
- Keep repeating steps 2 and 3 until the queue is empty.

Breadth-first search(BFS)



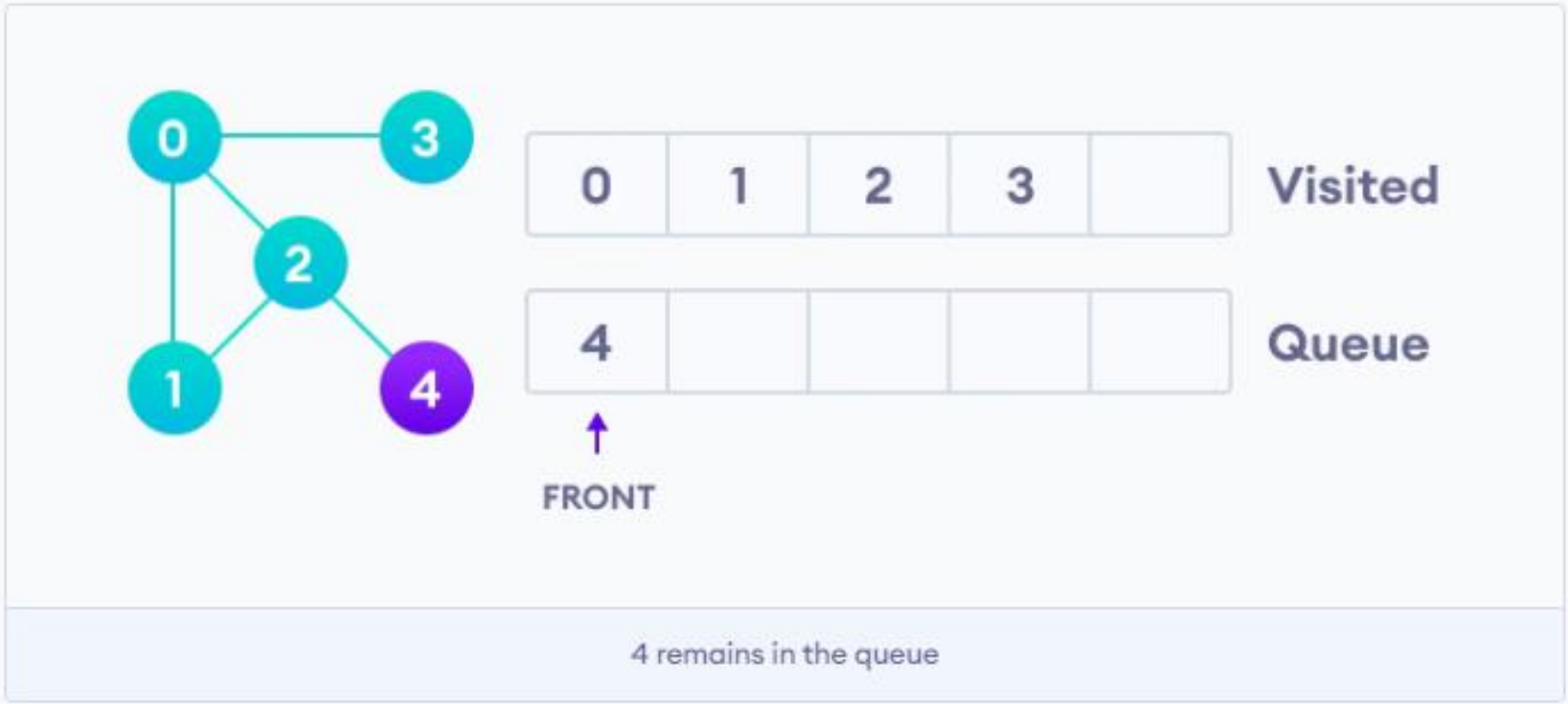
Breadth-first search(BFS)

Breadth-first search(BFS)



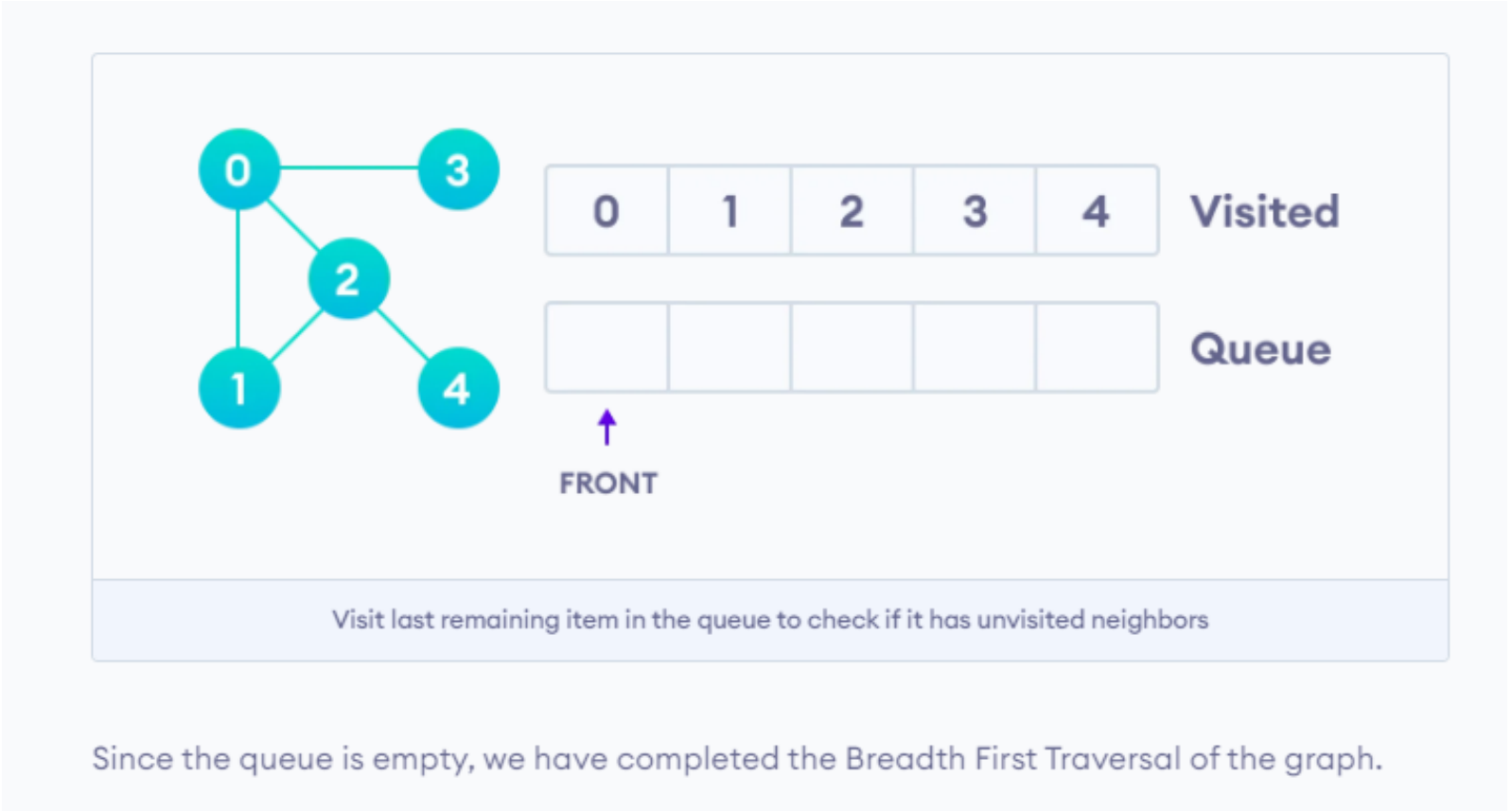
Breadth-first search(BFS)

Breadth-first search(BFS)



Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.

Breadth-first search(BFS)



Breadth-first search(BFS)

- **BFS pseudocode**
- Create a queue Q
- Mark v as visited and put v into Q
- While Q is non-empty

  remove the head u of Q

  mark and enqueue all (unvisited) neighbours of u

Breadth-first search(BFS)

- **BFS Algorithm Complexity**
- The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is O(V).

Breadth-first search(BFS)

- **BFS Algorithm Applications**
- To build index by search index
- For GPS navigation
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph
- In minimum spanning tree

Depth-first search(DFS)

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node.
- Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.
- During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

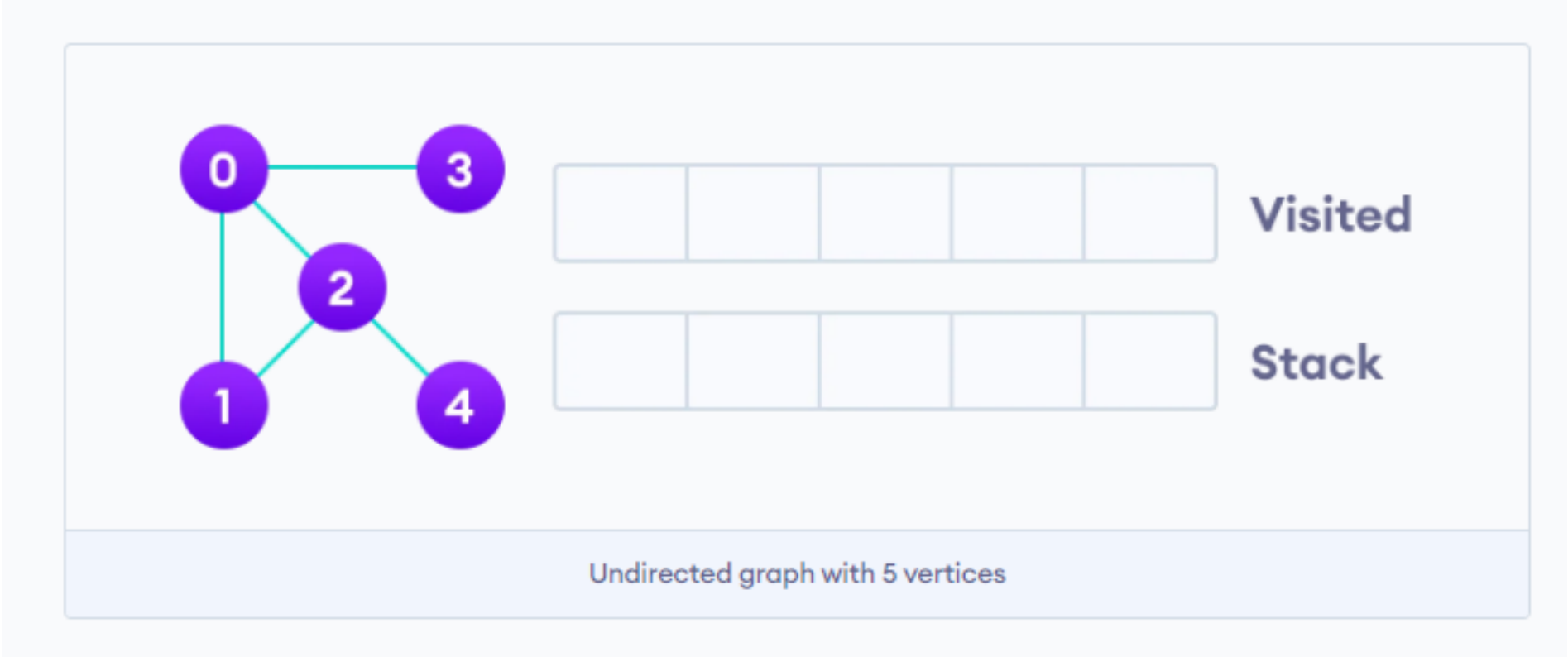
Depth-first search(DFS)

- A standard DFS implementation puts each vertex of the graph into one of two categories:
- Visited
- Not Visited

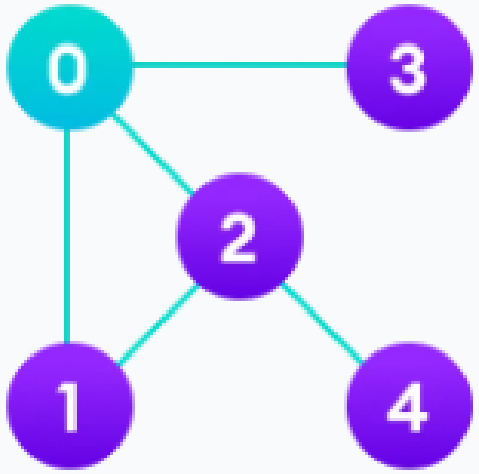
Depth-first search(DFS)

- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The DFS algorithm works as follows:
- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Keep repeating steps 2 and 3 until the stack is empty.

Depth-first search(DFS)



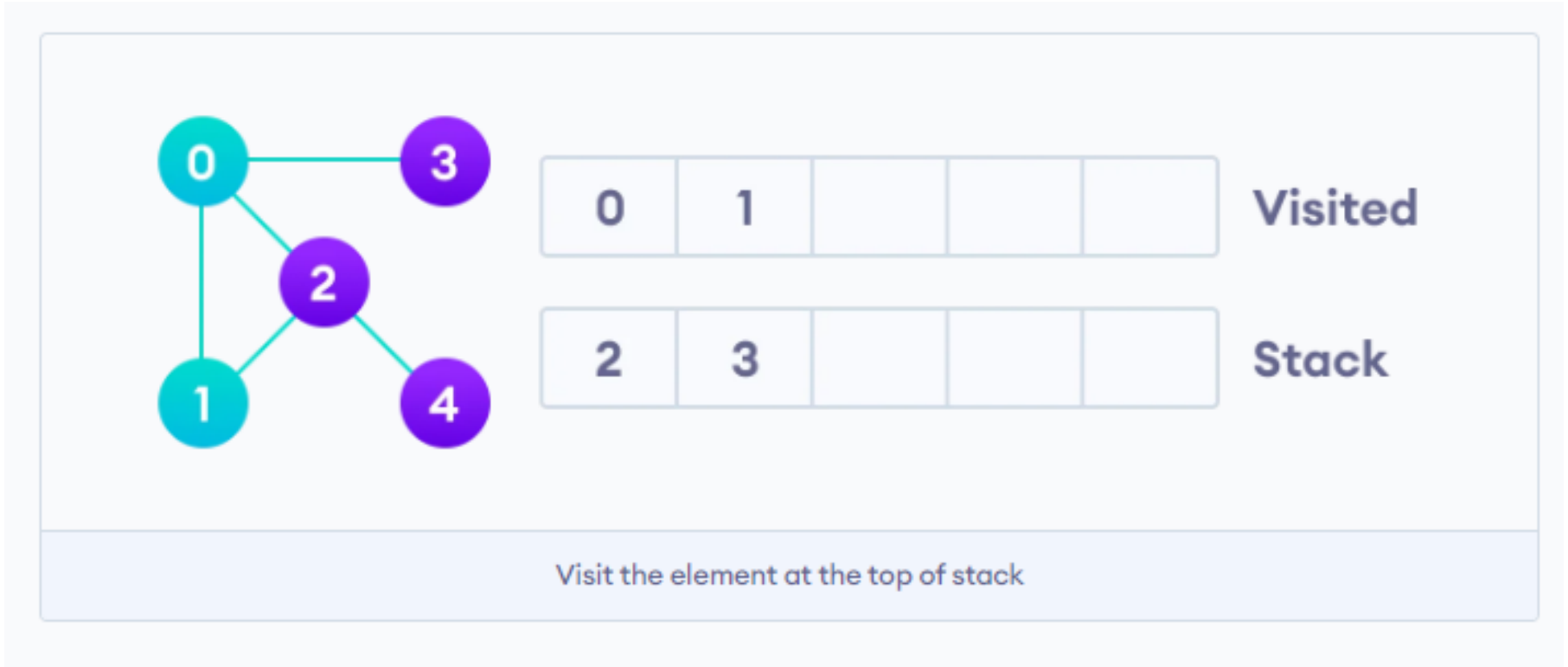
Depth-first search(DFS)



0					Visited
1	2	3			Stack

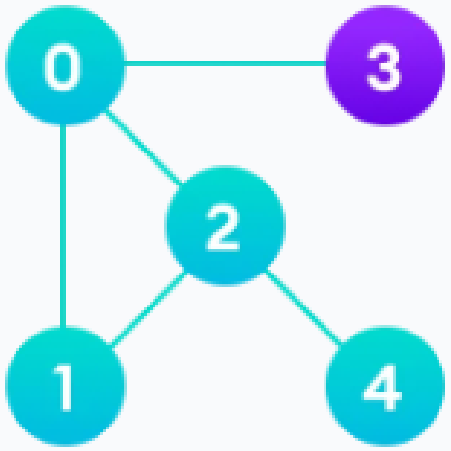
Visit the element and put it in the visited list

Depth-first search(DFS)



Depth-first search(DFS)

Depth-first search(DFS)

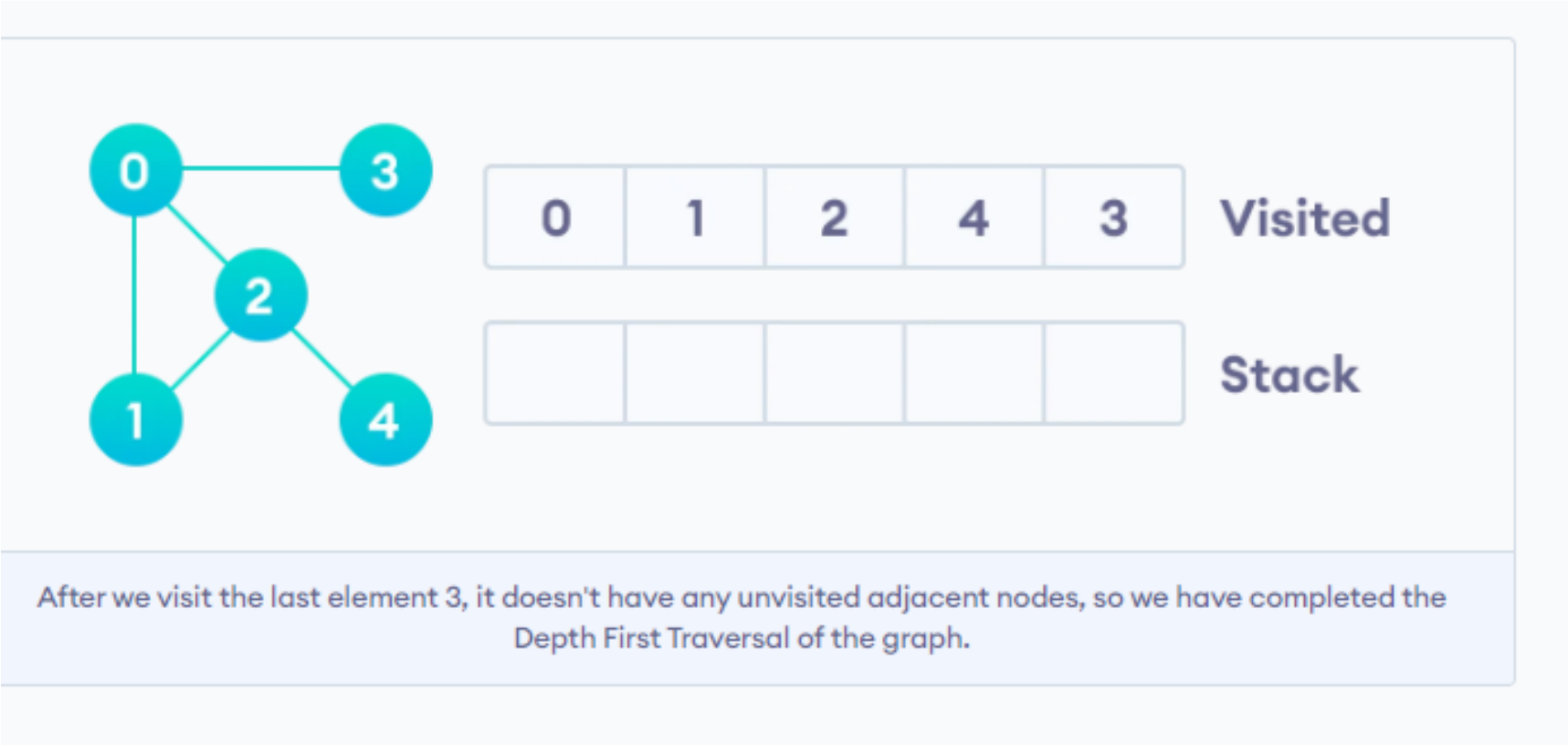


0	1	2	4		Visited

3					Stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

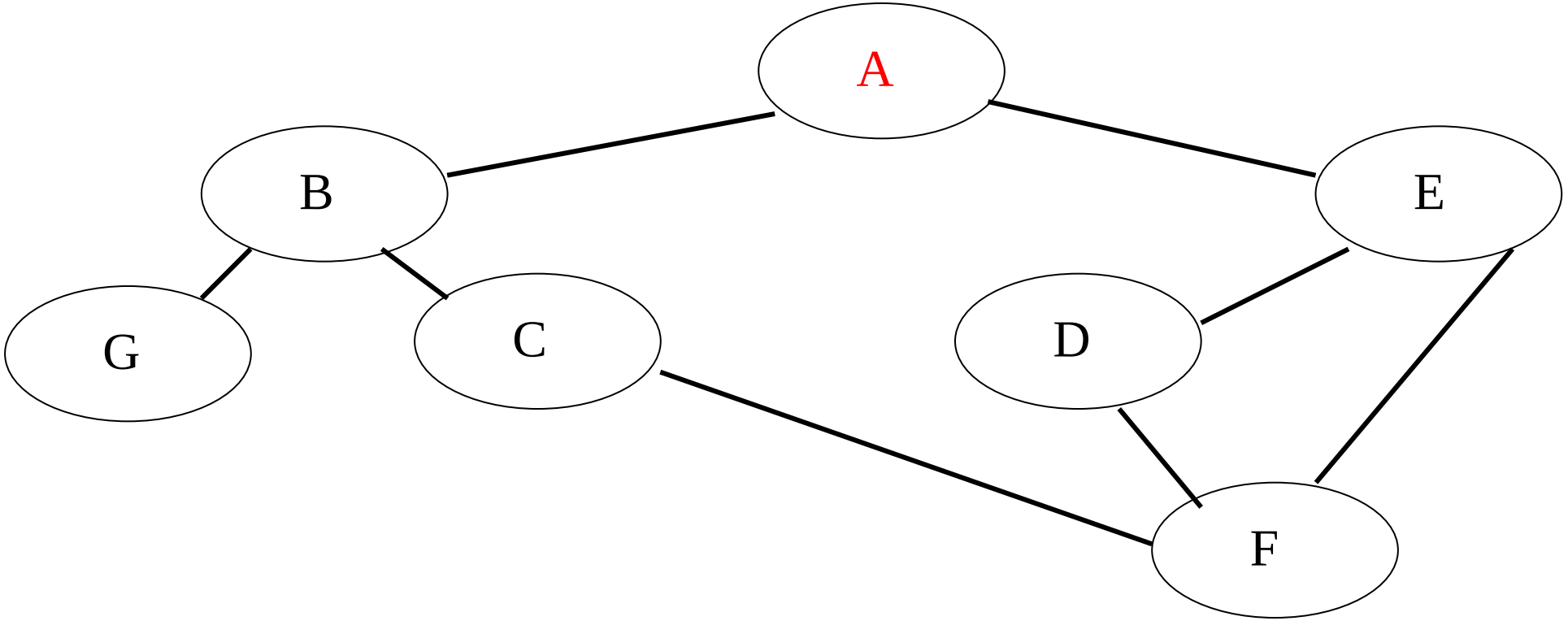
Depth-first search(DFS)



DFS(graph g, vertex s)

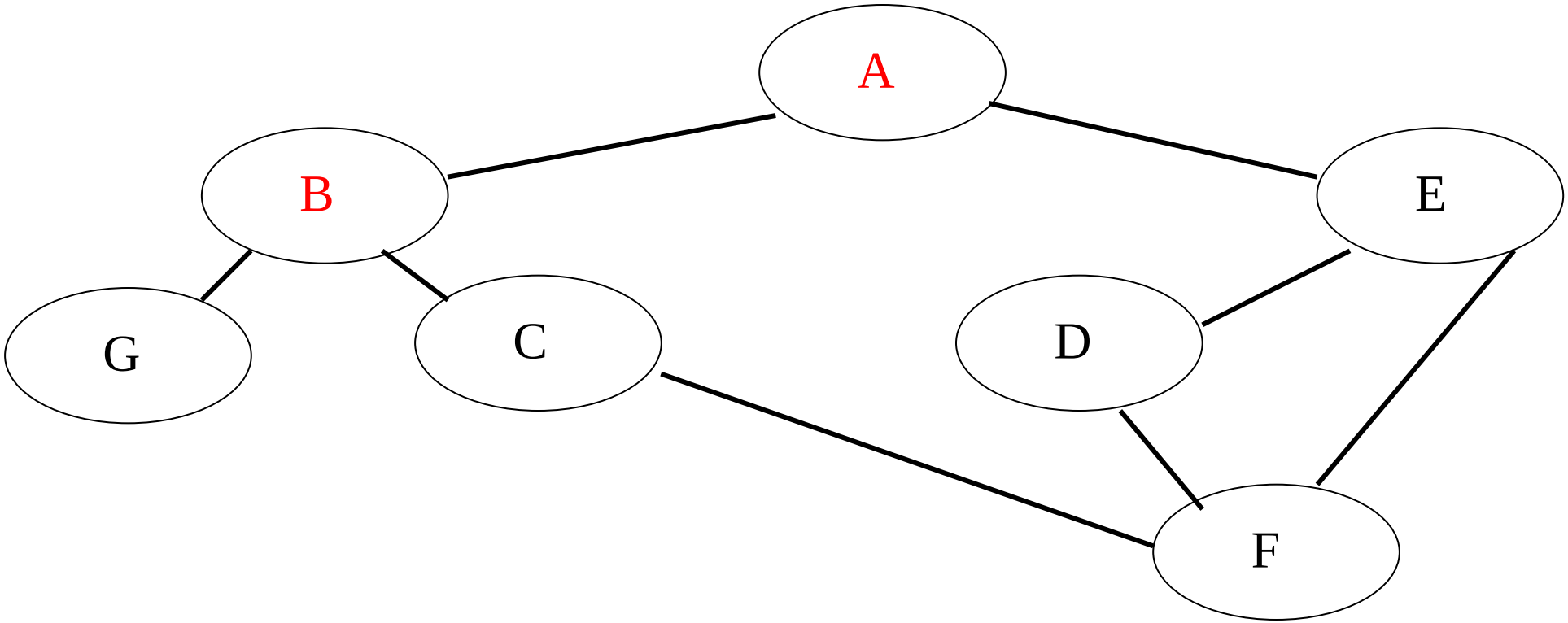
```
1. unmark all vertices in G
2. Stack ← new stack
3. mark s
4. Push(stack, s)
5. while (not empty(stack))
6.    curr ← pop(stack)
7.    visit curr // e.g., print its data
8.    for each edge <curr, V>
9.       if V is unmarked
10.         mark V
11.         push(stack, V)
  1.  Print curr
```

Current vertex: A



Start with A. Mark it.

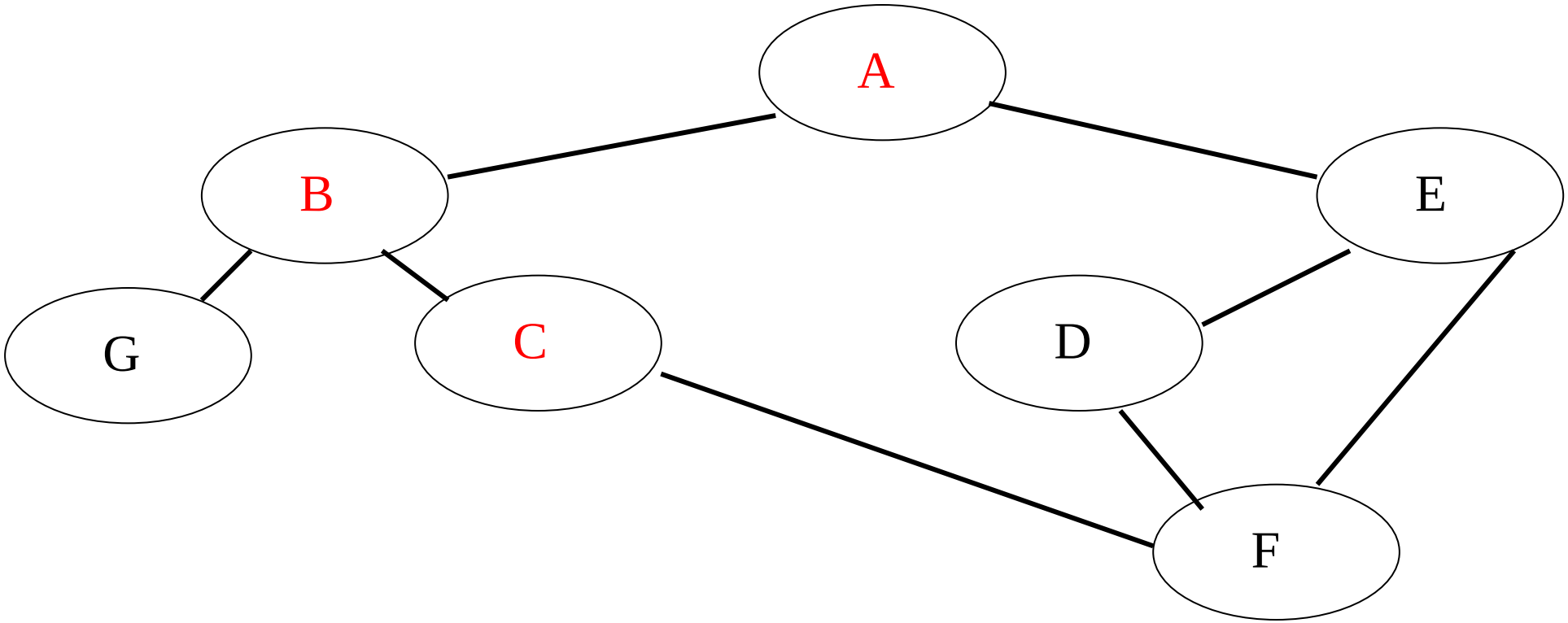
Current: B



Expand A’s adjacent vertices. Pick one (B).

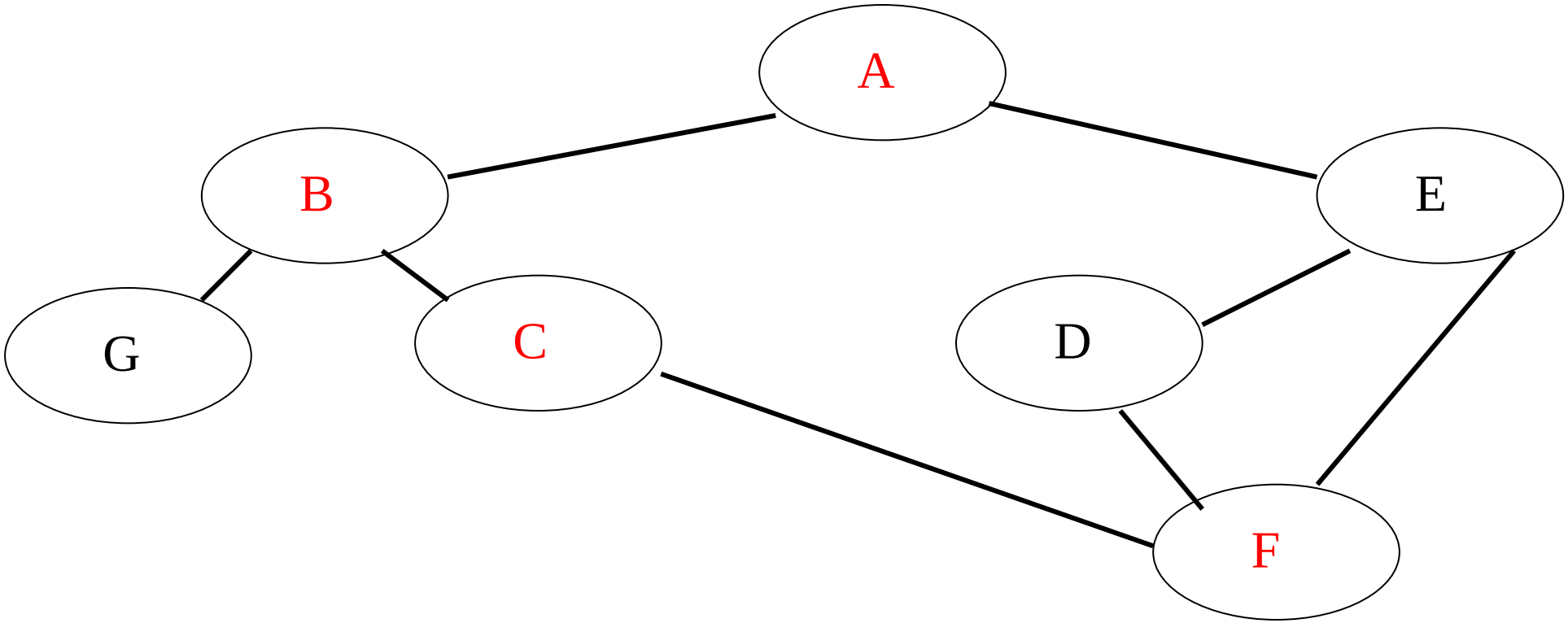
Mark it and re-visit.

Current: C



Now expand B, and visit its neighbor, C.

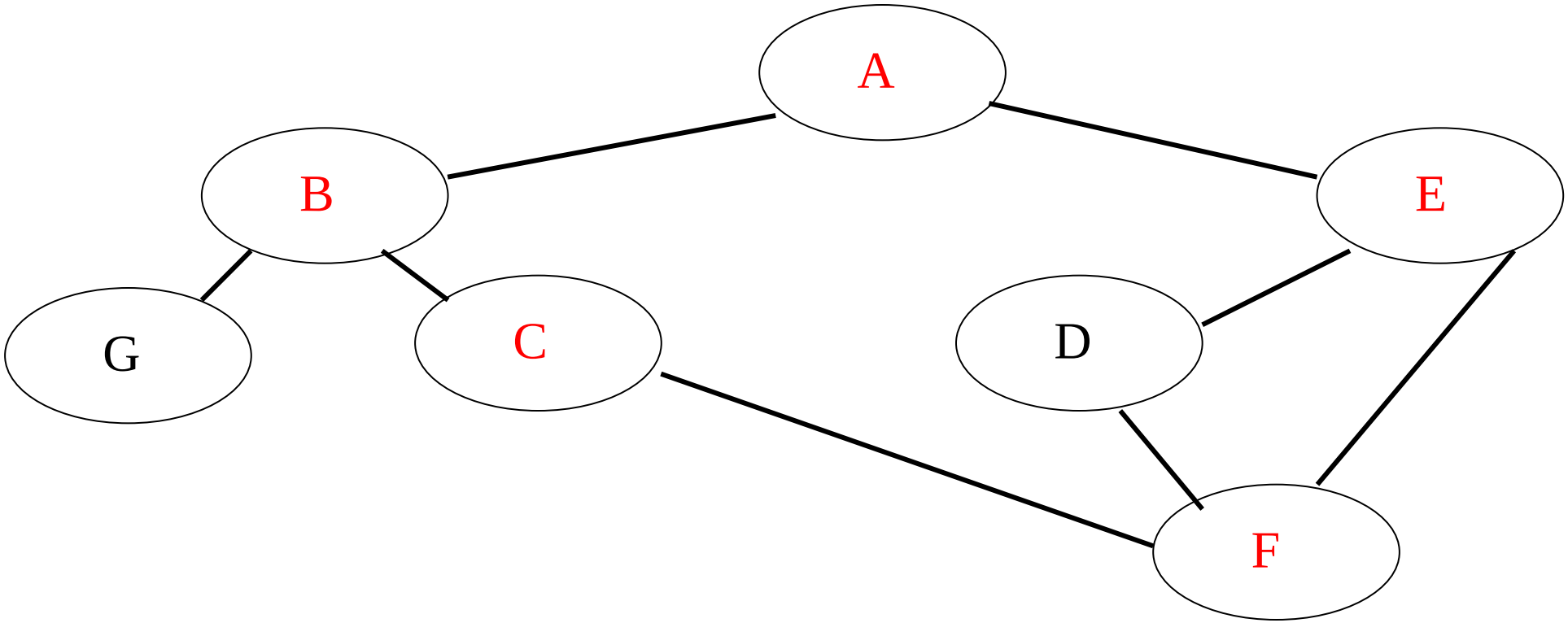
Current: F



Visit F.

Pick one of its neighbors, E.

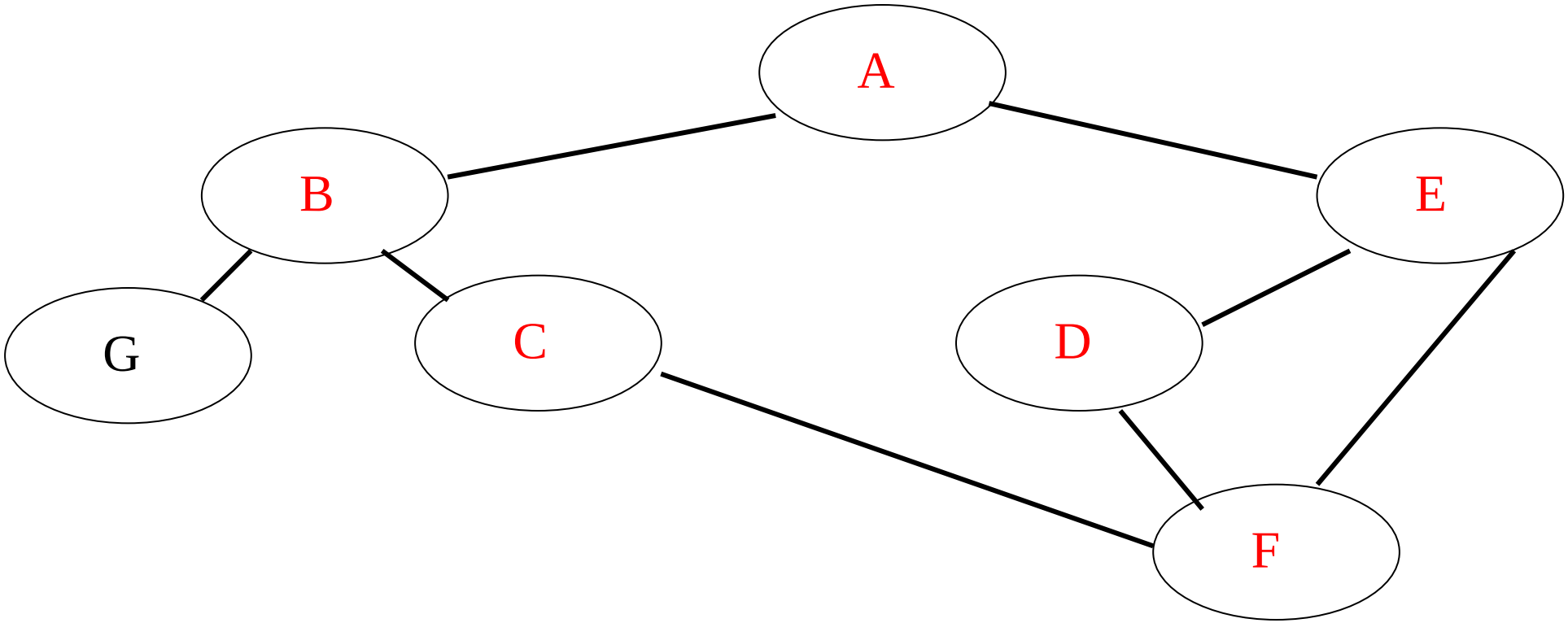
Current: E



E’s adjacent vertices are A, D and F.

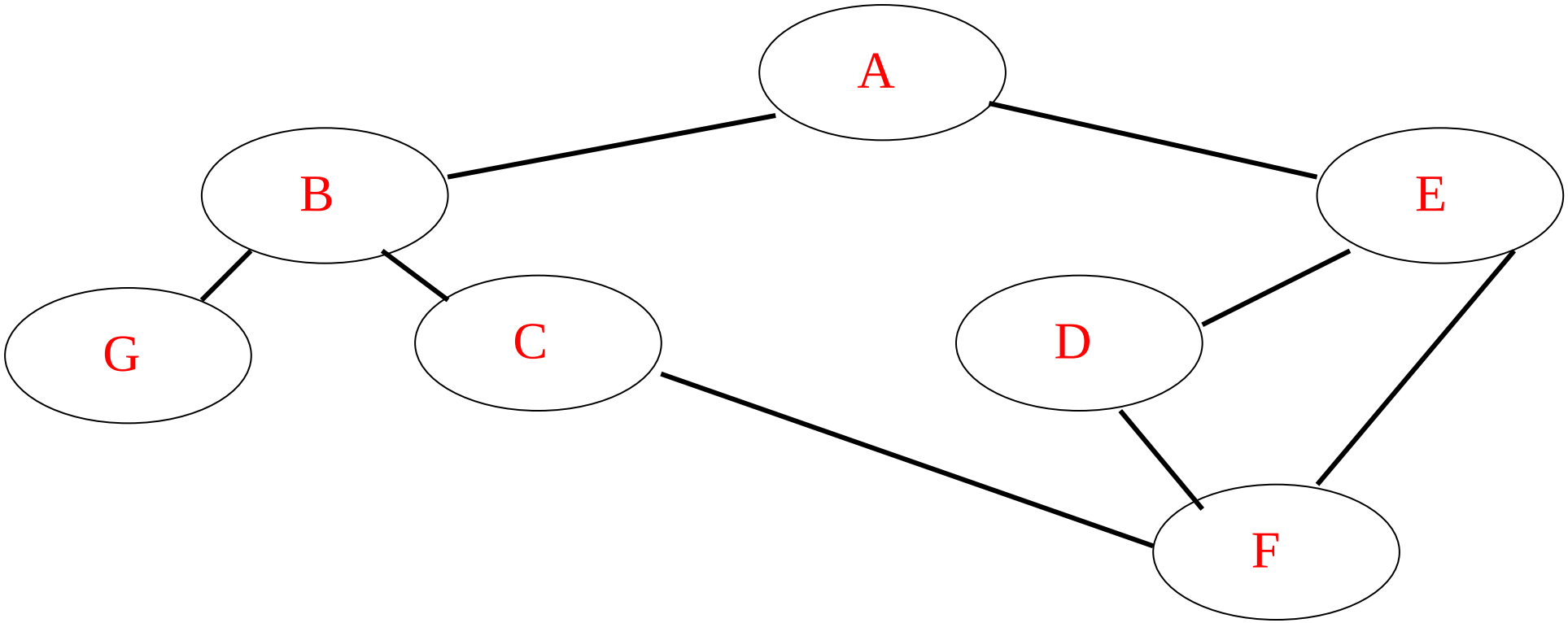
A and F are marked, so pick D.

Current: D



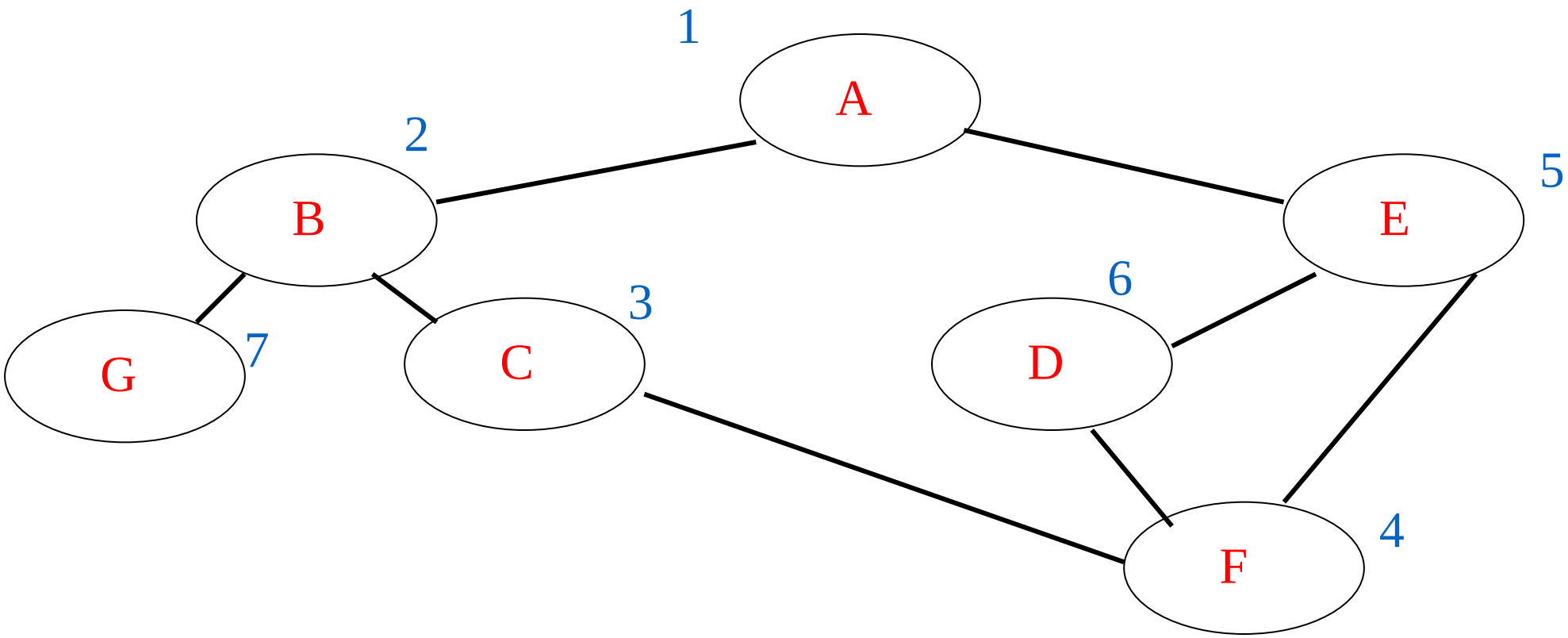
Visit D. No new vertices available. Backtrack to E. Backtrack to F. Backtrack to C. Backtrack to B

Current: G



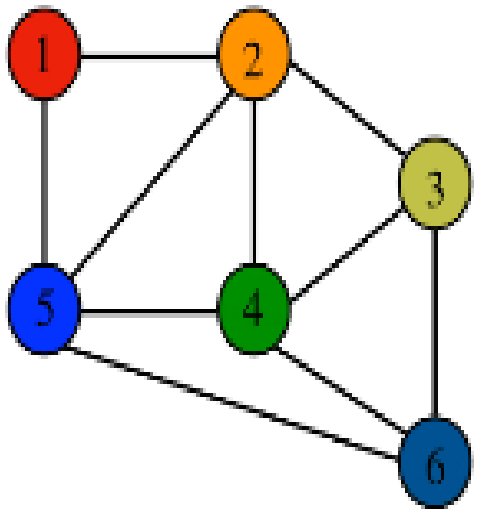
Visit G. No new vertices from here. Backtrack to B. Backtrack to A. E already marked so no new.

Current:

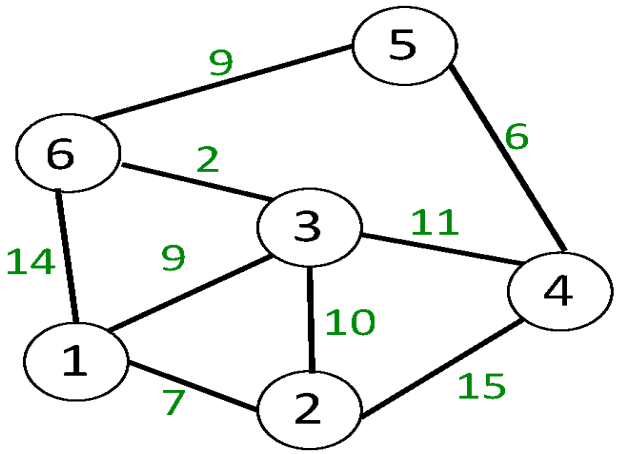


Done. We have explored the graph in order:

A B C F E D G



stack	Marked	curr	DFS
{}	{}	-	-
~~{1}~~	{1}	1	1
~~{2,5}~~	{1,2,5}	2	1,2
~~{3,4,5}~~	{1,2,5,3,4}	3	1,2,3
~~{6,4,5}~~	{1,2,5,3,4,6}	6	1,2,3,6
~~{4,5}~~	{1,2,5,3,4,6}	4	1,2,3,6,4
~~{5}~~	{1,2,5,3,4,6}	5	1,2,3,6,4,5
Empty	{1,2,5,3,4,6}	-	DFS: 1,2,3,6,4,5



Stack	Marked	Curr	DFS
{}	{}		
~~{1}~~	{1}	1	1
~~{2,3,6}~~	{1,2,3,6}	2	1,2
~~{4,3,6}~~	{1,2,3,6,4}	4	1,2,4
~~{5,3,6}~~	{1,2,3,6,4,5}	5	1,2,4,5
~~{3,6}~~	{1,2,3,6,4,5}	3	1,2,4,5,3
~~{6}~~	{1,2,3,6,4,5}	6	1,2,4,5,3,6
empty	algo terminates		DFS: 1,2,4,5,3,6

Depth-first search(DFS)

- **Complexity of Depth First Search**
- The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is O(V).

Depth-first search(DFS)

- **Application of DFS Algorithm**
- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

THANK YOU