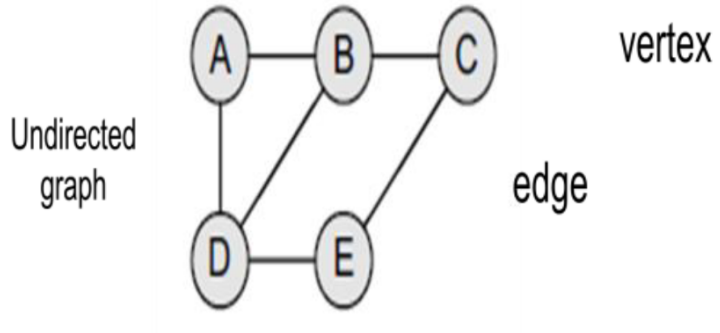
Graphs

Graph Basic Concepts

Definition

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

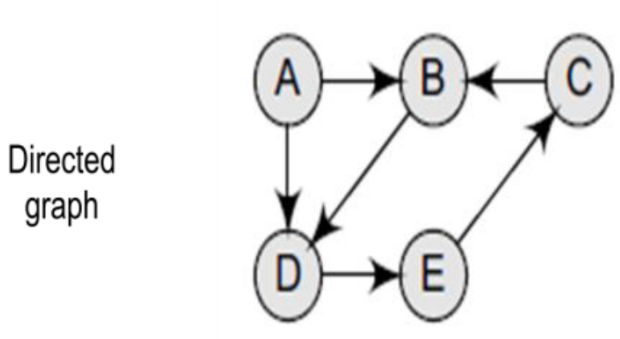


A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them.

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.

A graph G with V(G) = {A, B, C, D and E} and

E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}.



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).

A graph G with V(G) = {A, B, C, D and E} and

E(G) = {(A, B), (C, B), (A, D), (B, D), (D, E), (E, C)}.

Terminologies

Adjacent nodes or **neighbours** 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 neighbours.

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 neighbours. 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.

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.

Terminologies….2

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 path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

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 connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph.

Complete graph 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.

Termonologies..3

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.

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 loops is called a multi-graph.

Size of a graph The size of a graph is the total number of edges in it.

BI-CONNECTED components

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected.

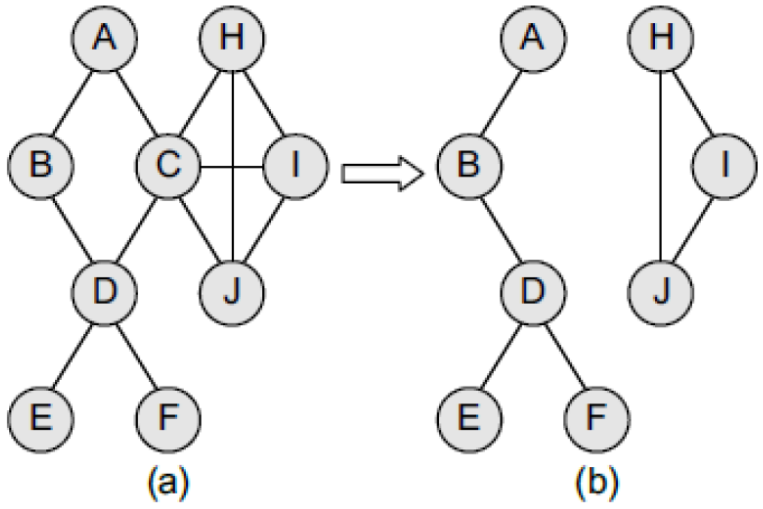
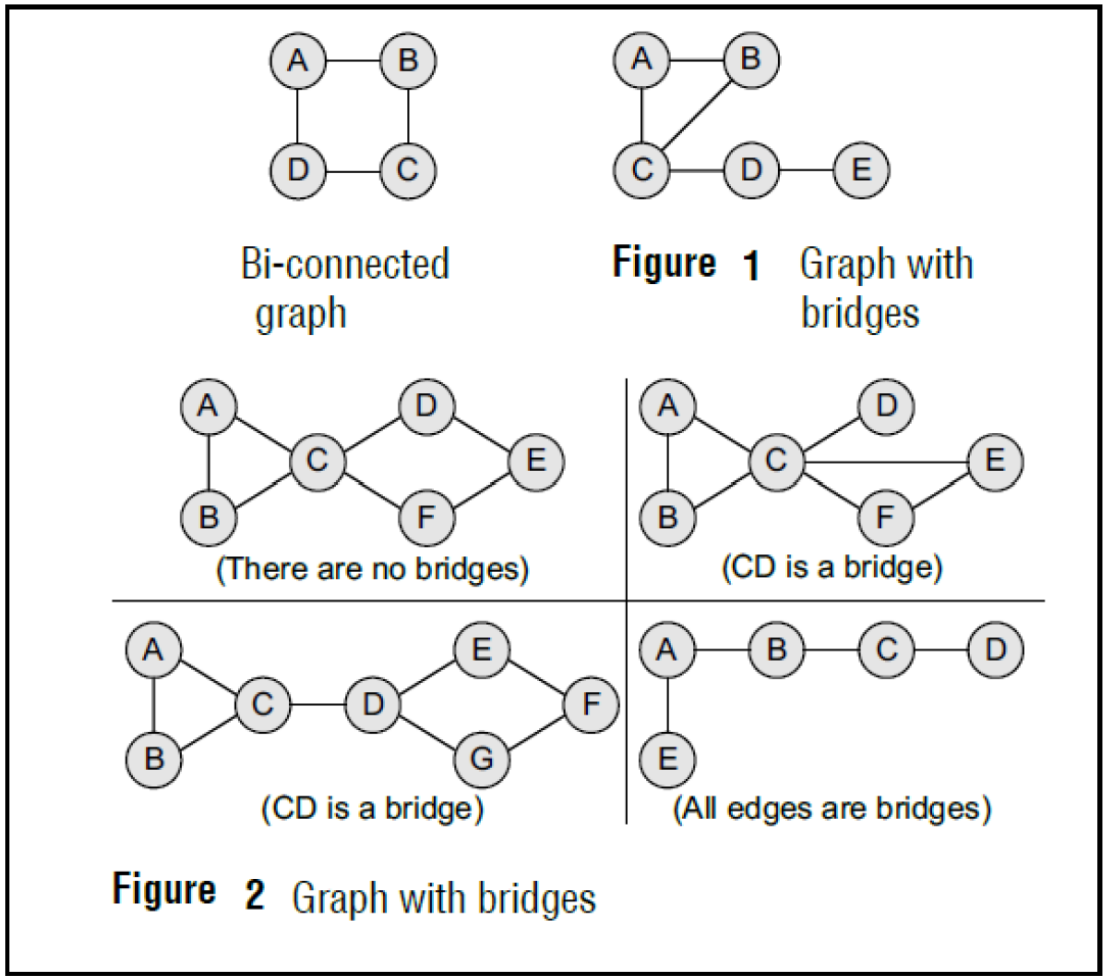


Figure Non bi-connected graph

A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.

In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

Note that the graph shown in Fig. (a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph (Fig. (b)).



As for vertices, there is a related concept for edges. An edge in a graph is called a bridge if removing that edge results in a disconnected graph.

Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge. Look at the graph shown in Fig.1.

In the graph, CD and DE are bridges. Consider some more examples shown in Fig. 2.

Adjacency Matrix Representation

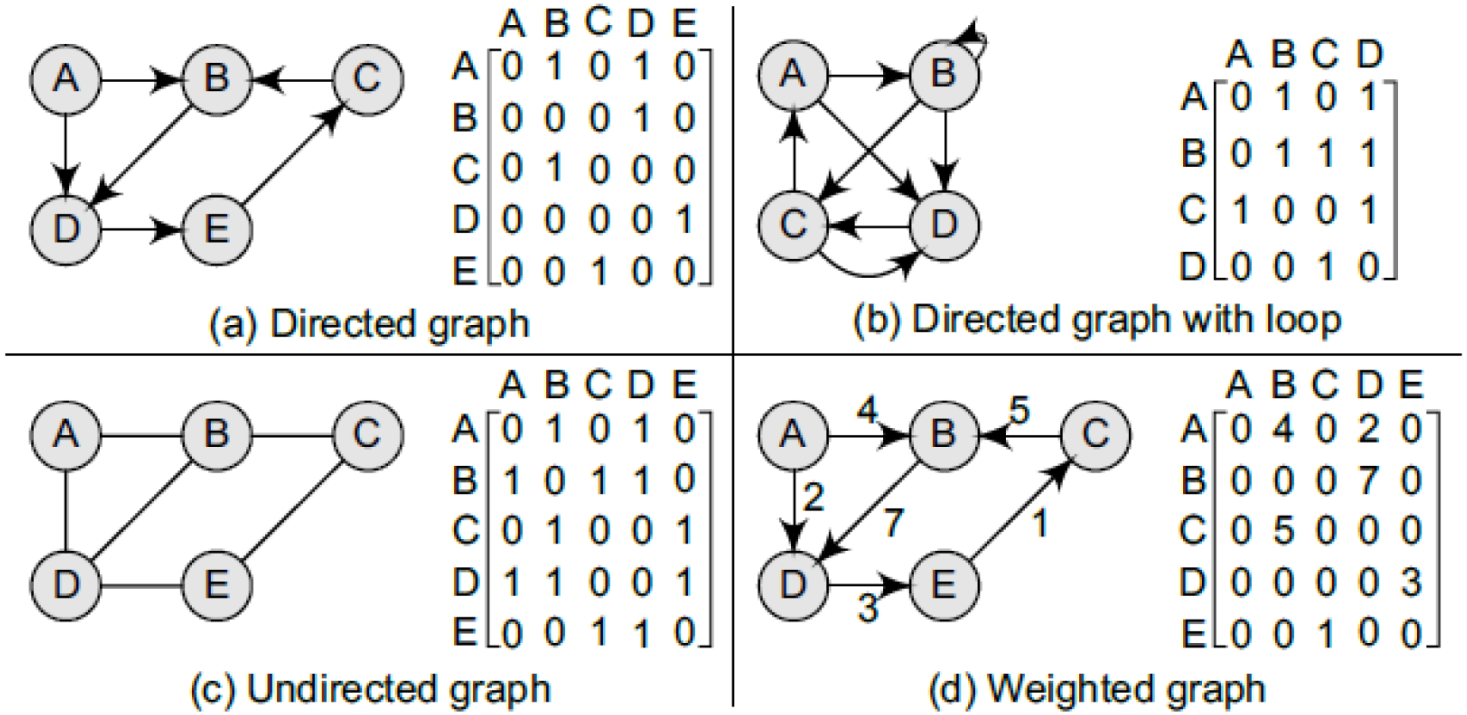


Figure Graphs and their corresponding adjacency matrices

Conclusion:

- ✓ For 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

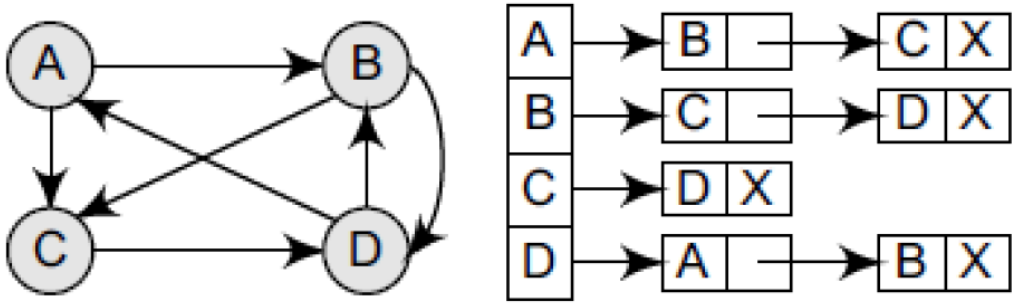
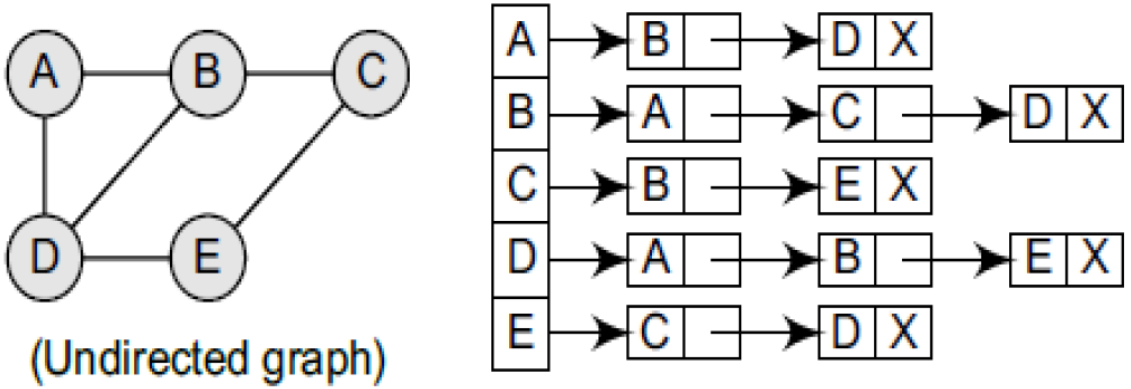


Figure Graph G and its adjacency list



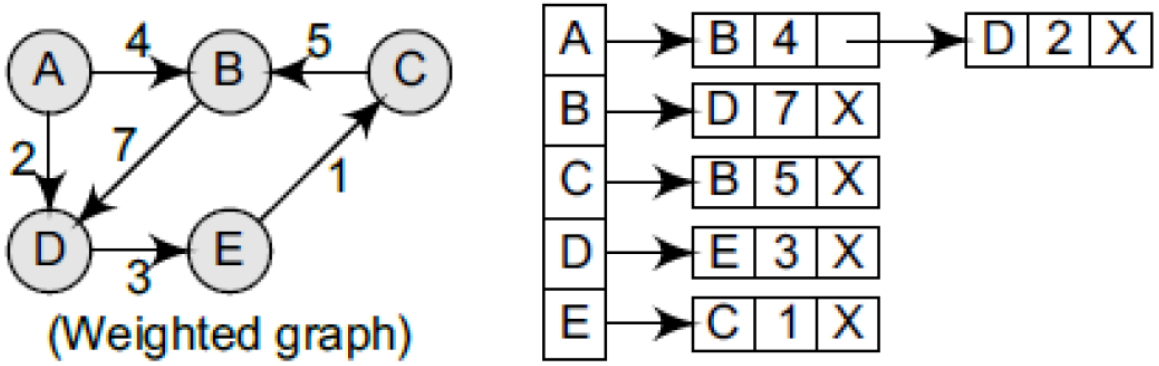


Figure Adjacency list for an undirected graph and a weighted graph

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. That is, 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.

GRAPH TRAVERSAL ALGORITHMS

These two methods are:
1. Breadth-first search
2. Depth-first search

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. But both these algorithms 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 or 2, depending on its current state.

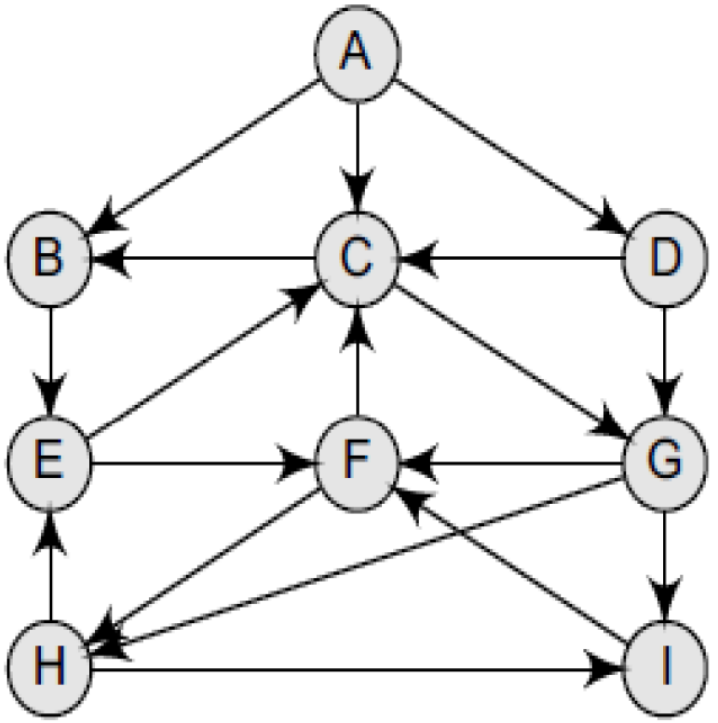
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Breadth-first search (BFS):

A queue holds the nodes that are waiting for further processing and a variable STATUS represents the current state of the node.

```
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 neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Algorithm for breadth-first search



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

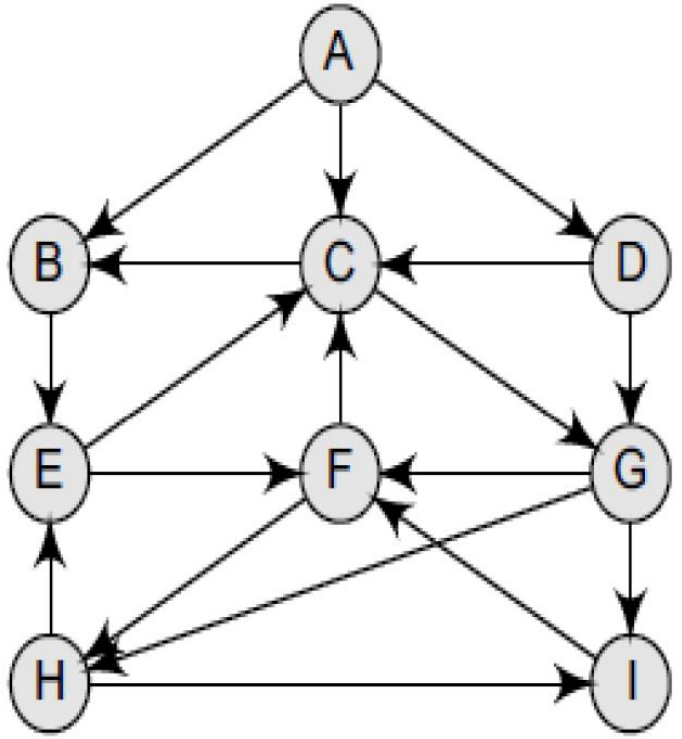
Figure Graph G and its adjacency list

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

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

```
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 the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```



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

Figure Graph G and its adjacency list

Difference between Breadth-first search and Depth-first search in Graphs

Traversal Approach

- **BFS**: Explores all the nodes at the current depth level before moving to the next depth level. It proceeds level by level in a graph.
- **DFS**: Explores as far along a branch (or path) as possible before backtracking to explore other branches.

Data Structure Used

- **BFS**: Utilizes a **queue** (FIFO - First In, First Out) to keep track of nodes to be explored.
- **DFS**: Utilizes a **stack** (LIFO - Last In, First Out). This can be implemented using recursion (implicit stack) or an explicit stack.

Order of Exploration

- **BFS**: Traverses nodes in the order they are discovered, starting with the closest (in terms of edge count) to the source node.
- **DFS**: Traverses nodes by diving deep into a branch and backtracking when needed, which can lead to exploring nodes in a less structured order.

Consider the graph G given in Figure. The adjacency list of G is also given. Suppose we want to print all the 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 here.

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, 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

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

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.

The time complexity can be given as (O(|V| + |E|)).

Assignment

Qn.1 Describe giving examples the following algorithms that calculate the shortest path between the vertices of a graph G:

a) Minimum spanning tree

b) Dijkstra's algorithm

c) Warshall's algorithm

Hint: Using ChatGPT make summary notes on the following and submit in 1 Weeks time