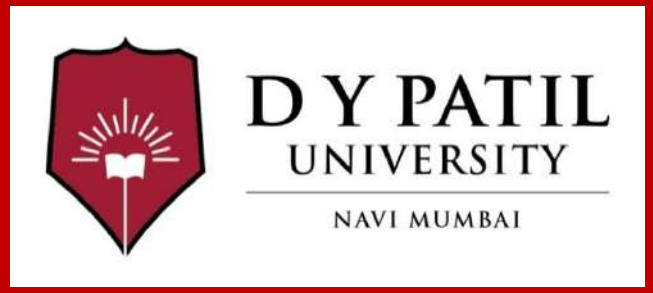
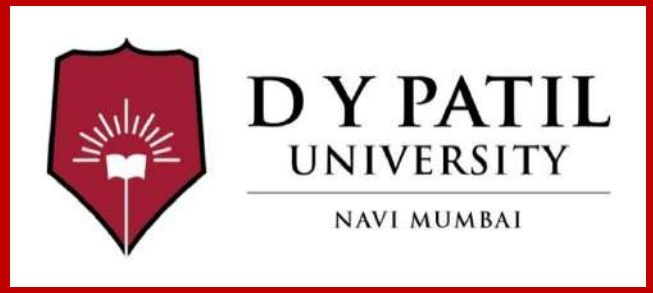


Subject Name : Data Structure

Unit No: 6

Unit Name : Graph

Faculty Name :Pooja Mehta,Swati Kadu



Unit No.:6

Faculty Name : Pooja Mehta,Swati kadu

Index

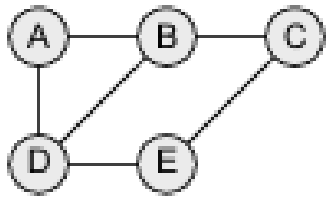
Contents

- Definition of Graph, Basic Concepts of Graph
- Representation of Graph
- Adjacency Matrix
- Adjacency List
- Graph Traversal:
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

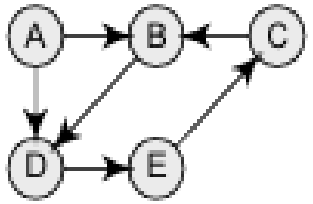
Graph

- 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.
- **Why are Graphs Useful?**
- Graphs are widely used to model any situation where entities or things are related to each other in pairs.
  - 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.

- 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.
- 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.



Undirected Graph



Directed graph

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.

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

- **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, v , 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
- **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.

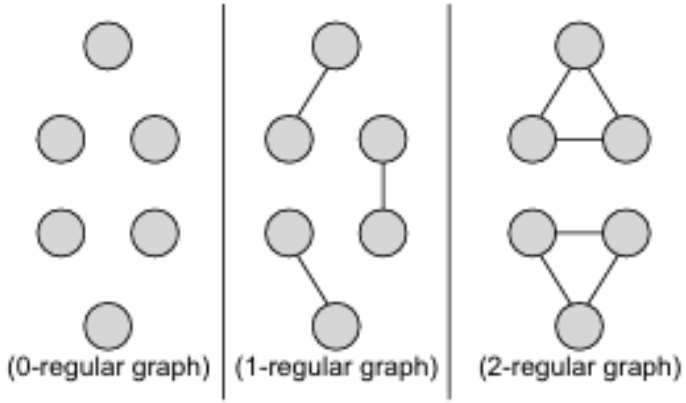


Figure 13.3 Regular graphs

Graph Terminology

- **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.
- **Clique** : In an undirected graph G = (V, E), clique is a subset of the vertex set C Õ V, such that for every two vertices in C, there is an edge that connects two vertices.
- **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. Figure 13.4(a) shows a multi-graph.
- **Size of a graph**: The size of a graph is the total number of edges in it

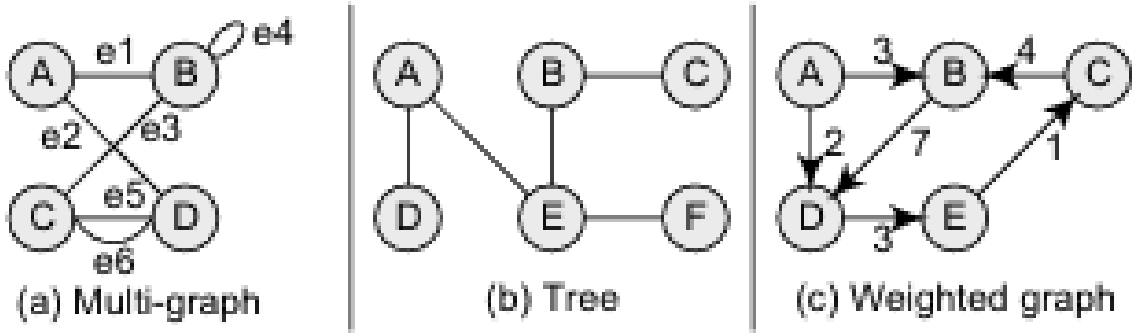
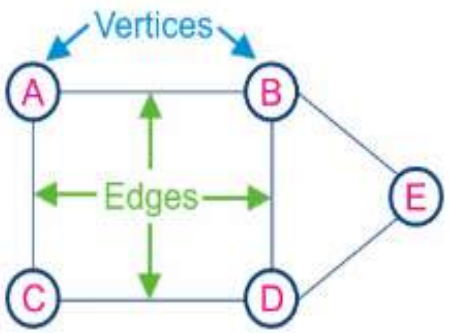


Figure 13.4 Multi-graph, tree, and weighted graph

Definition of Graph

A graph, in the context of mathematics and computer science, is a data structure used to represent relationships between entities. It consists of two sets:

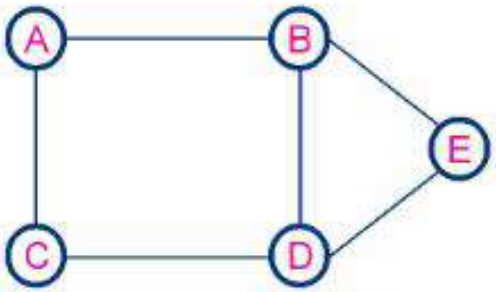
- **Vertices (V):** Represent the entities themselves, also called nodes or points.
- **Edges (E):** Represent the relationships between the entities, also called links or lines.



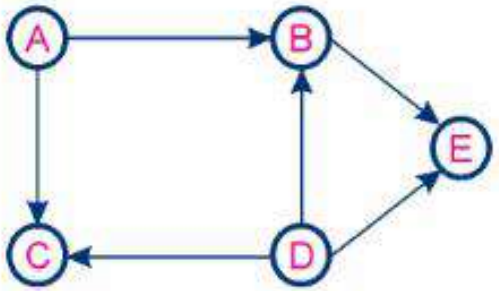
Definition

A graph is denoted as G = (V, E), where:
-V is a set of vertices.
-E is a set of edges.

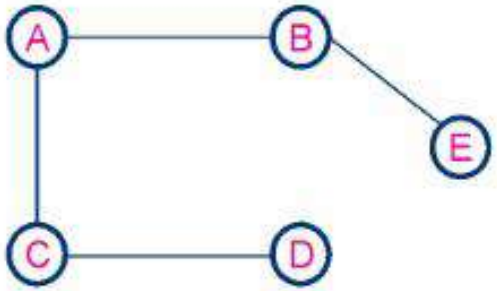
D Y PATIL UNIVERSITY
NAVI MUMBAI



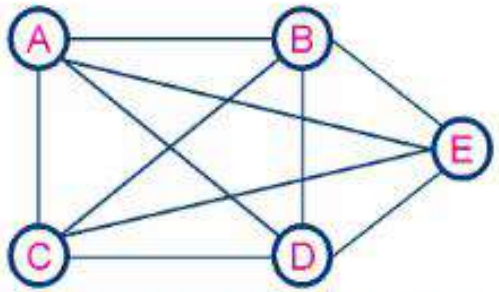
Undirected Graph



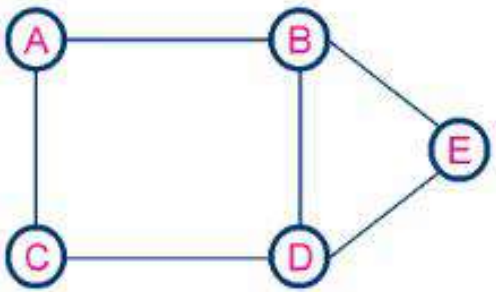
Directed Graph



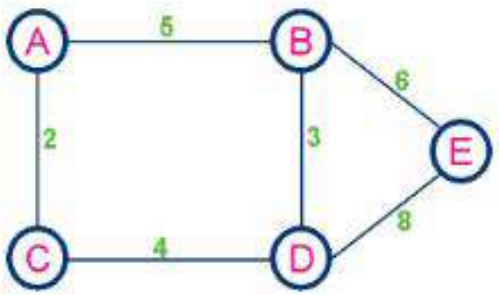
Sparce Graph



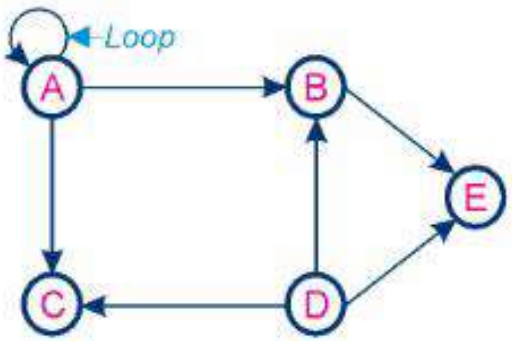
Complete Graph (Dense)



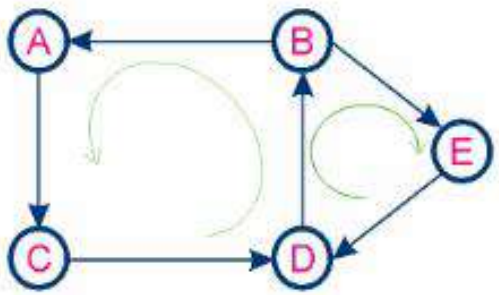
Unweighted Graph



Weighted Graph



Acyclic Graph



Cyclic Graph

- **Graph:** A collection of nodes (or vertices) and edges that connect pairs of nodes.
- **Vertex (Node):** The fundamental unit by which graphs are formed. A vertex represents a point in the graph.
- **Edge:** A line connecting two vertices in a graph. It can be directed (having a direction) or undirected (no direction).
- **Directed Graph (Digraph):** A graph in which the edges have a direction, indicating a one-way relationship between vertices.
- **Undirected Graph:** A graph in which the edges do not have a direction, indicating a two-way relationship.
- **Weighted Graph:** A graph in which edges have weights assigned to them, typically representing costs, lengths, or capacities.
- **Unweighted Graph:** A graph in which edges do not have weights.
- **Adjacent (Neighbors):** Two vertices are adjacent if there is an edge connecting them.
- **Degree:** The degree of a vertex is the number of edges connected to it. For directed graphs, there are **in-degree** (edges coming into the vertex) and **out-degree** (edges going out from the vertex).
- **Path:** A sequence of edges that allows you to go from one vertex to another.
- **Cycle:** A path that starts and ends at the same vertex without traversing any edge more than once.

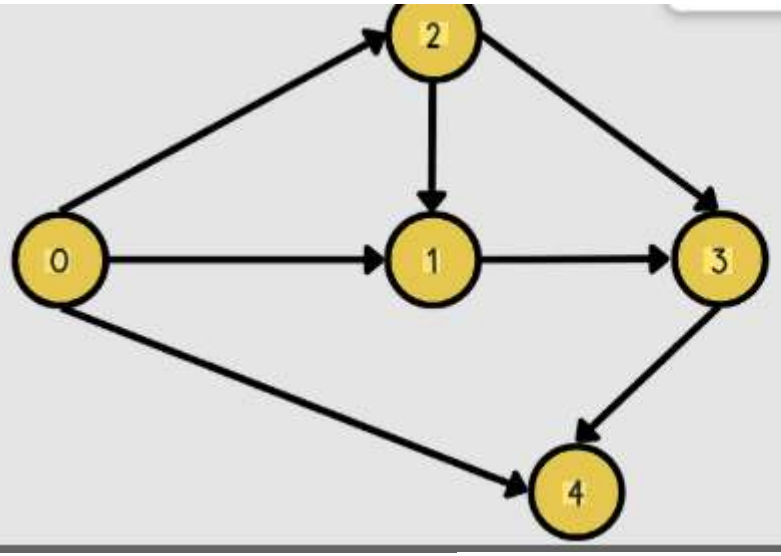
- **Loop:** An edge that connects a vertex to itself.
- **Subgraph:** A graph formed from a subset of the vertices and edges of another graph.
- **Connected Graph:** An undirected graph is connected if there is a path between every pair of vertices.
- **Disconnected Graph:** A graph is disconnected if it is not connected, i.e., if there are at least two vertices with no path between them.
- **Complete Graph:** A graph in which there is an edge between every pair of vertices.
- **Bipartite Graph:** A graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set.
- **Tree:** A connected, undirected graph with no cycles.
- **Acyclic Graph:** A graph with no cycles. In the context of directed graphs, it is often called a Directed Acyclic Graph (DAG).
- **Graph Isomorphism:** Two graphs are isomorphic if there is a one-to-one correspondence between their vertex sets that preserves edge connectivity.

Directed Graph

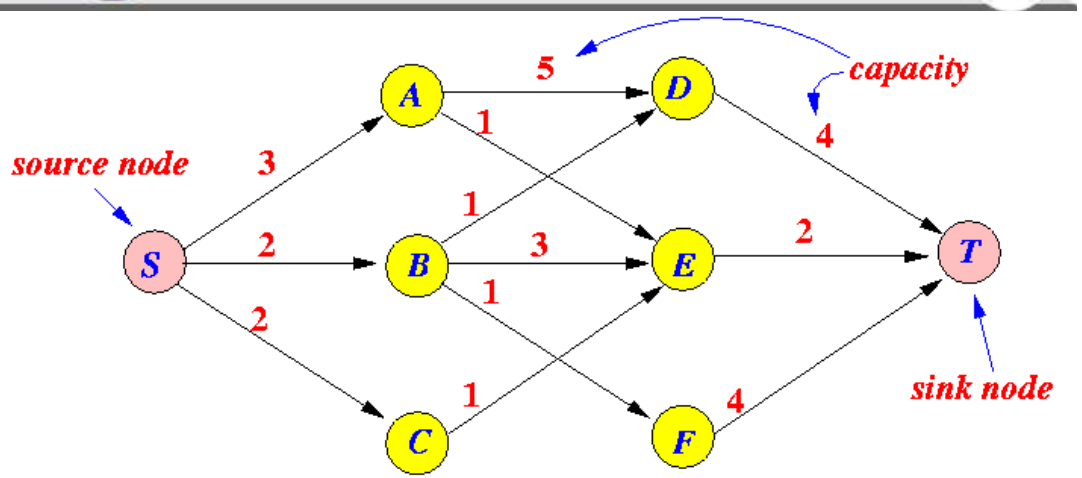
- A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),
- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. 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** : 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).
- **Isolated vertex** : A vertex with degree zero. Such a vertex is not an end-point of any edge.
- Pendant vertex :(also known as leaf vertex) A vertex with degree one.
- **Cut vertex** : A vertex which when deleted would disconnect the remaining graph.
- **Source** : A node u is known as a source if it has a positive out-degree but a zero in-degree.
- **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. For example, if you consider the directed graph given in Fig. 13.5(a), you will observe that node D is reachable from node A.
- **Strongly connected directed graph** : A digraph is said to be strongly connected if and only if there exists a path between 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 between 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.



nodes	indegree	outdegree
0	0	3
1	2	1
2	1	2
3	2	1
4	2	0



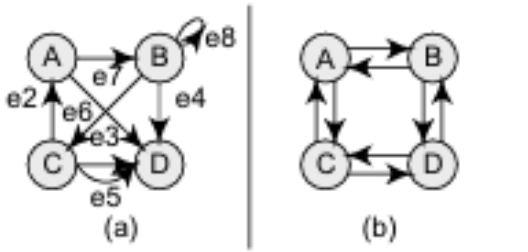


Figure 13.5 (a) Directed acyclic graph and (b) strongly connected directed acyclic graph

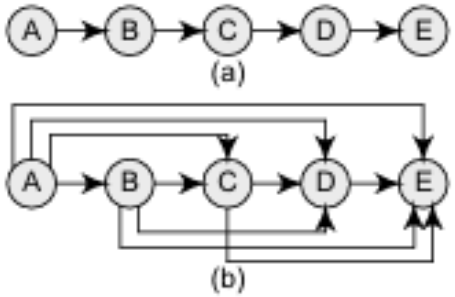


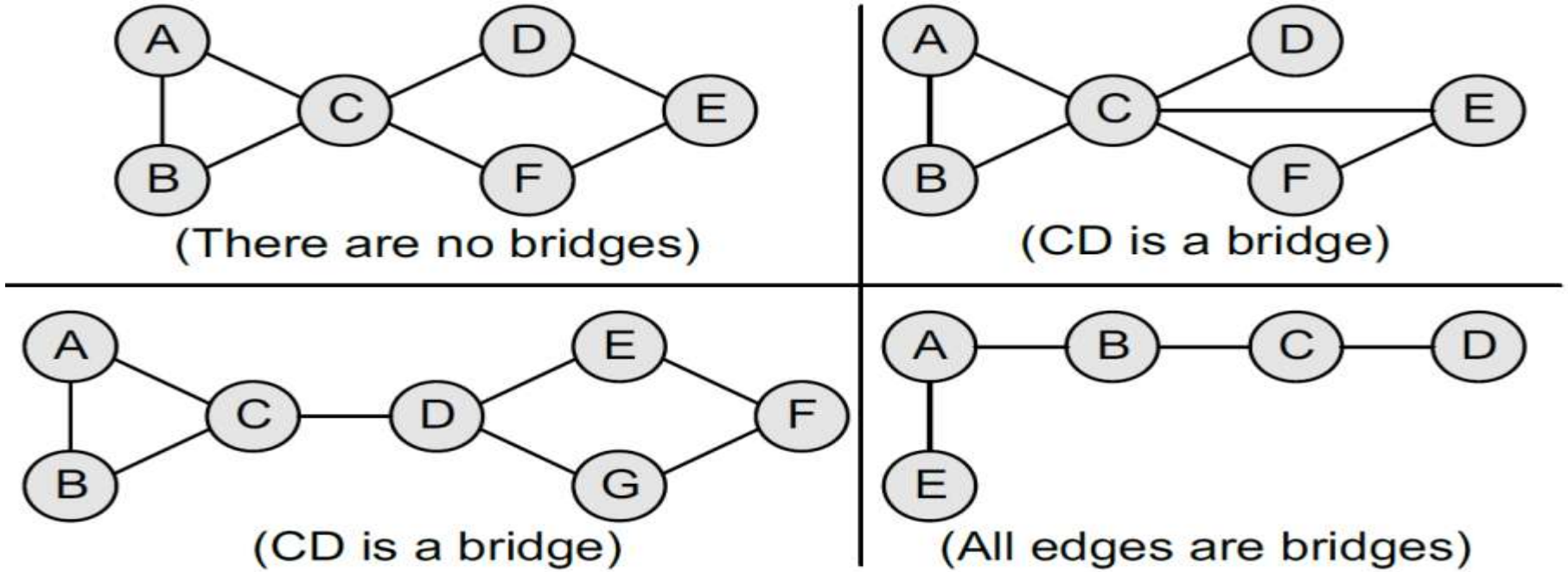
Figure 13.6 (a) A graph G and its (b) transitive closure G*

Weakly connected digraph : A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

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. In Fig. 13.5(a), e3 and e5 are multiple edges connecting nodes C and D.

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 cycles with an exception that it cannot have more than one loop at a given node.

- Bridge:
- An edge in a graph is called a bridge if removing that edge results in a disconnected graph



Representation of Graphs

- There are three common ways of storing graphs in the computer's memory. They are:

(1) Sequential representation by using an adjacency matrix.

(2) Linked representation by using an adjacency list that stores the neighbors of a node using a linked list.

(3) 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. In a directed graph G, if node v is adjacent to node u, then there is definitely 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 the dimension of n ¥ n.

$$a_{ij} \begin{cases} 1 & [\text{if } v_i \text{ is adjacent to } v_j, \text{ that is there is an edge } (v_i, v_j)]A \\ 0 & [\text{otherwise}] \end{cases}$$

Figure 13.13 Adjacency matrix entry

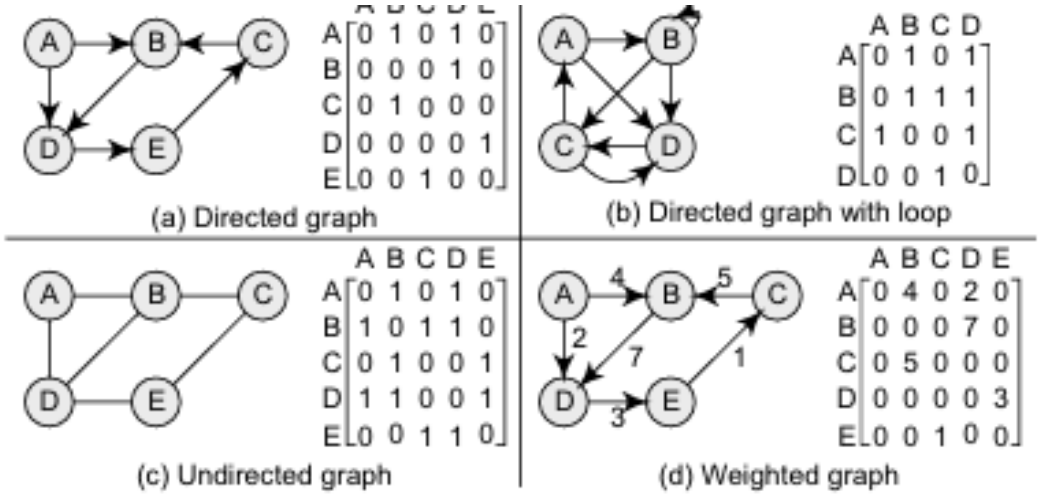


Figure 13.14 Graphs and their corresponding adjacency matrices

Sequential Representation

- For any graph GG having nn nodes, the adjacency matrix will have the dimension of $nn \times nn$
- – The rows and columns are labelled by graph vertices
- – An entry $aaiiii$ in the adjacency matrix will contain 1, if vertices

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

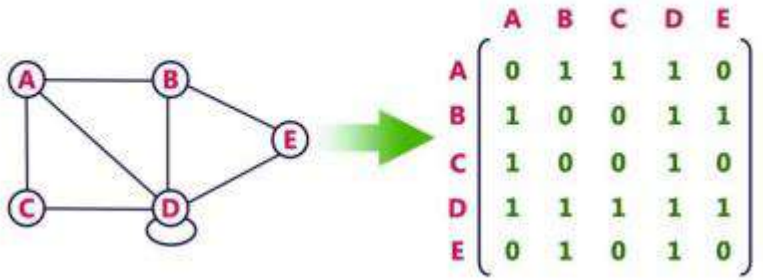
In this matrix, rows and columns both represent vertices.

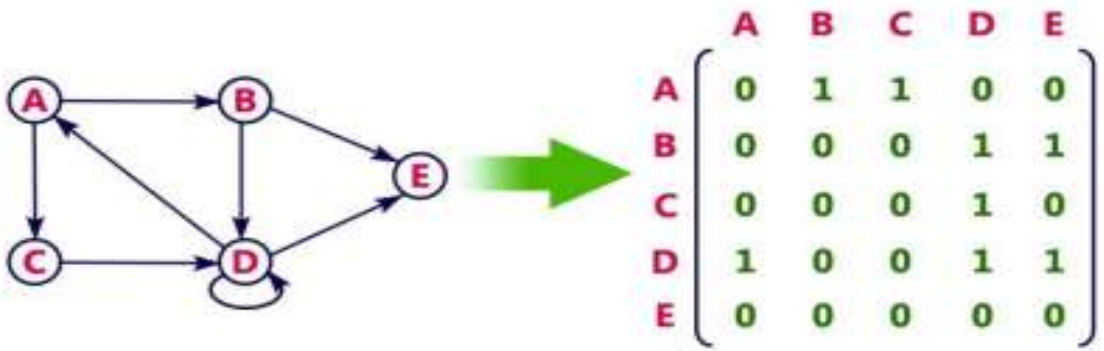
This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff $(v_i, v_j) \in E(G)$ ($\langle v_i, v_j \rangle$ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph

example : for undirected graph





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

The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

Merits of Adjacency Matrix:

From the adjacency matrix, to determine the connection of vertices is easy

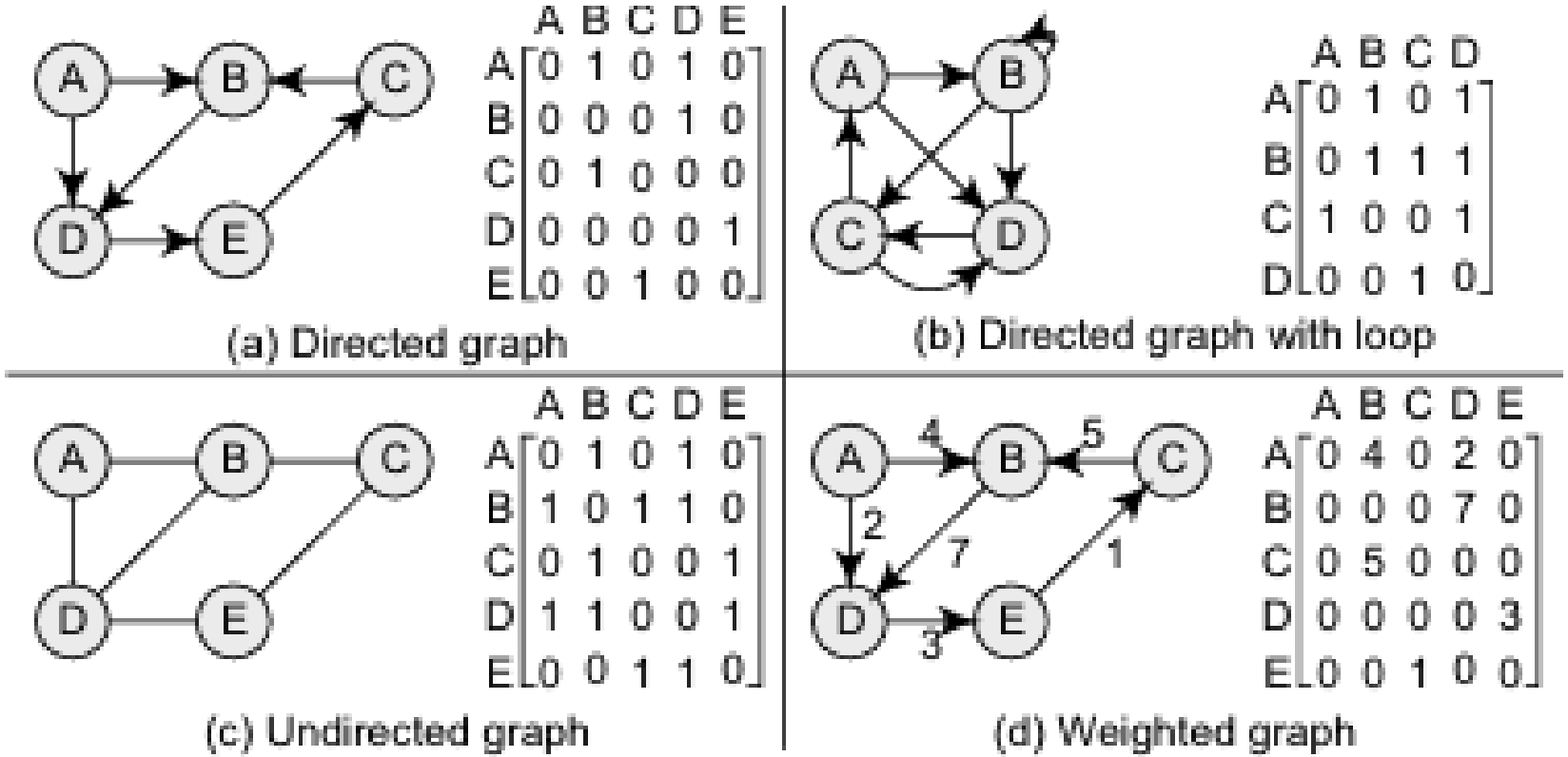
The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

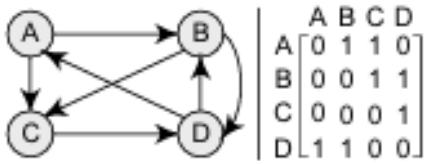
The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

Graphs and their adjacency matrices



Graphs and their corresponding adjacency matrices

$$A^2 = A^1 \times A^1$$



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

Figure 13.15 Directed graph with its adjacency matrix

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

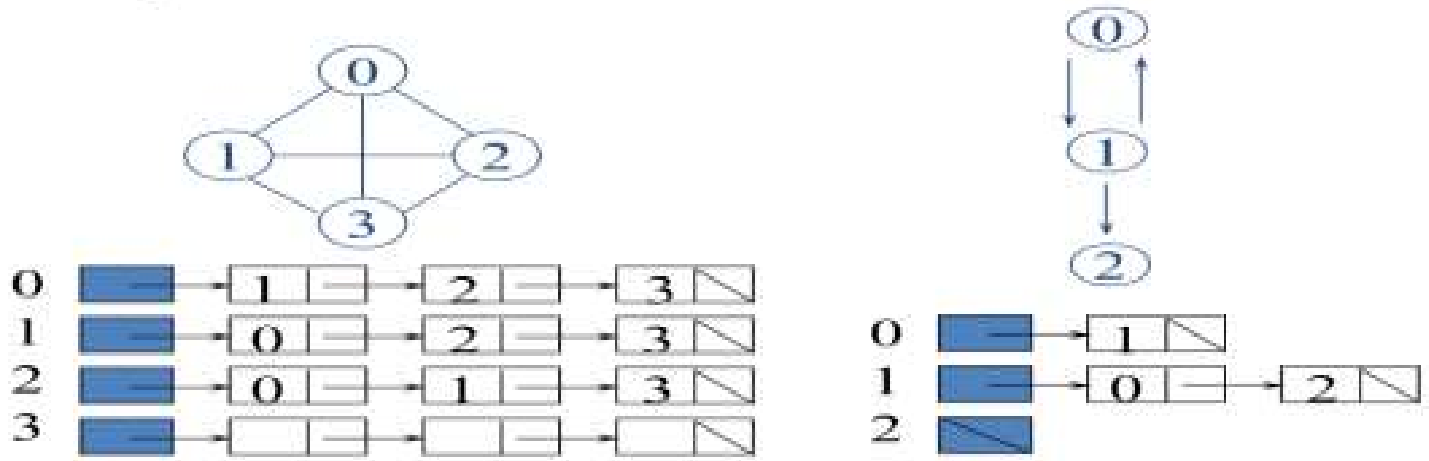
$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

based on the above calculations, we define matrix B as: Br = A1 + A2 + A3 + ... + Ar

Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:



So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. Structure is

Adjacency list for an undirected graph and a weighted graph

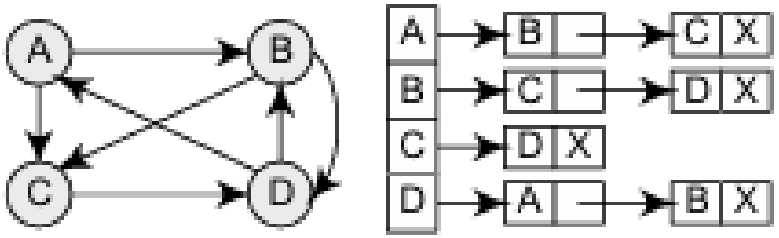
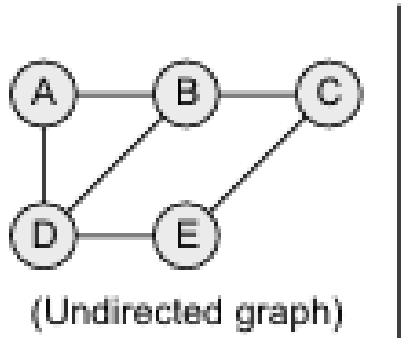
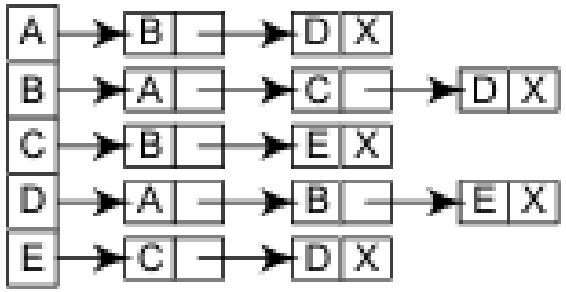
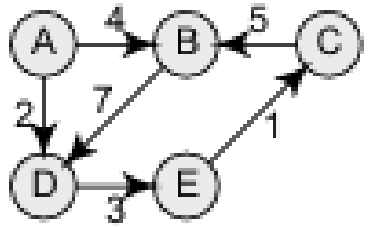


Figure 13.17 Graph G and its adjacency list

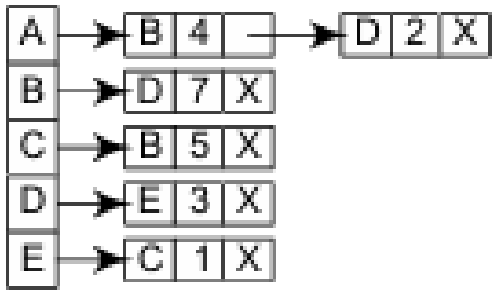


(Undirected graph)





(Weighted graph)



Advantages

o Adjacency list saves lot of space.

o We can easily insert or delete as we use linked list.

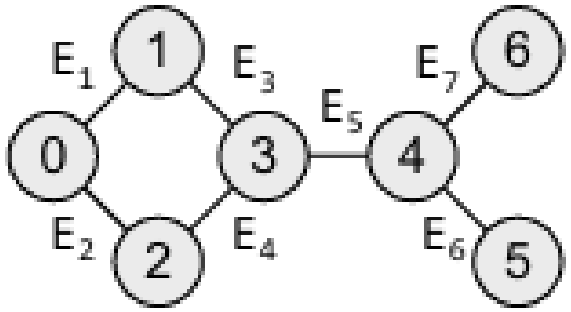
o Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Disadvantages

o The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

Adjacency MultiLists

- Adjacency Multi-lists are an edge, rather than vertex based, graph representation.
- In the Multilist representation of graph structures; these are two parts, a directory of Node information and a set of linked list of edge information.
- There is one entry in the node directory for each node of the graph.
- The directory entry for node i points to a linked adjacency list for node i. each record of the linked list area appears on two adjacency lists: one for the node at each end of the represented edge.



A single bit field to indicate whether the edge has been examined or not.
vi : A vertex in the graph that is connected to vertex vj by an edge.
vj : A vertex in the graph that is connected to vertex vi by an edge

Link i for vi : A link that points to another node that has an edge incident on vi . Link j for vi : A link that points to another node that has an edge incident on vj .

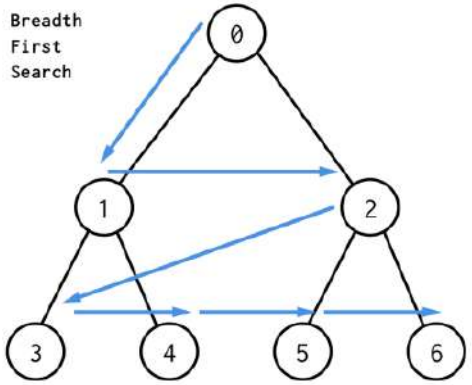
Edge 1		0	1	Edge 2	Edge 3
Edge 2		0	2	NULL	Edge 4
Edge 3		1	3	NULL	Edge 4
Edge 4		2	3	NULL	Edge 5
Edge 5		3	4	NULL	Edge 6
Edge 6		4	5	Edge 7	NULL
Edge 7		4	6	NULL	NULL

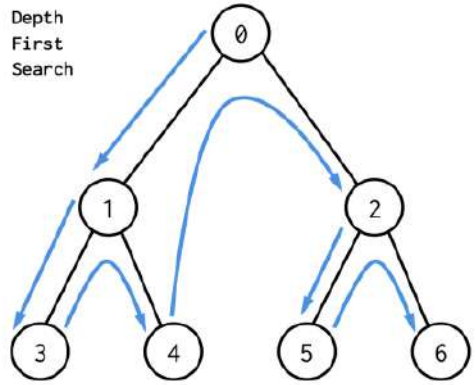
VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

Graph Traversal Algorithms

- There are two standard methods of graph traversal 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.





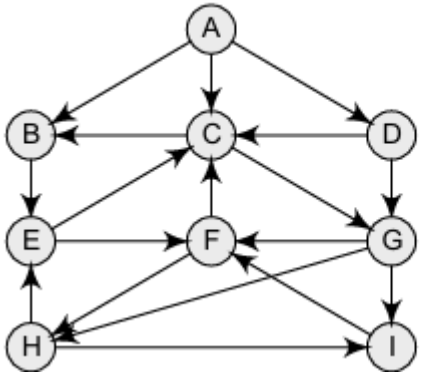
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Breadth-First search algorithm

- 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.
- we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once.
- This is accomplished by using **a queue** that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

Algorithm for breadth-first search

```
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
```



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

Graph G and its adjacency list

Consider the graph G given in Fig. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1. The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have Pas A -> C -> G -> I.

Applications of Breadth-First Search Algorithm

- Breadth-first search can be used to solve many problems such as:
  - Finding all connected components in a graph G.
  - Finding all nodes within an individual connected component.
  - Finding the shortest path between two nodes, u and v, of an unweighted graph.
  - Finding the shortest path between two nodes, u and v, of a weighted graph.

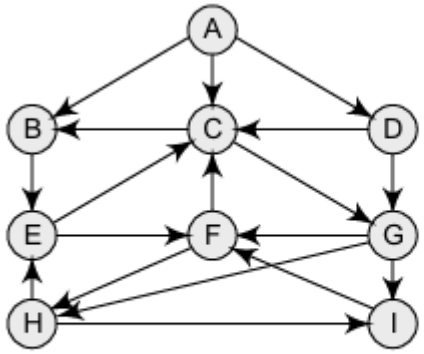
- **Time complexity:** In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd). , the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.
- **Completeness**: Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.
- **Optimality:** Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

```
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
```

Figure 13.22 Algorithm for depth-first search

Example

- 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.



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 13.23 Graph G and its adjacency list

(a) Push H onto the stack.

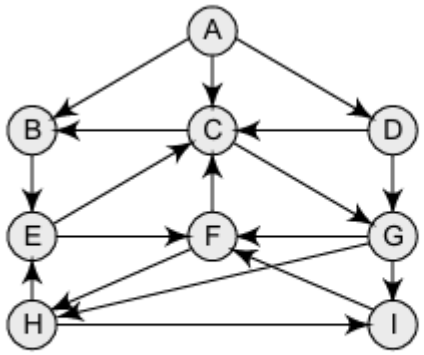
STACK: H

(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



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 13.23 Graph G and its adjacency list

(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:

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.

Features of Depth-First Search Algorithm

- **Space complexity**: The space complexity of a depth-first search is lower than that of a breadth first search.
- **Time complexity** : The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).
- **Completeness :** Depth-first search is said to be a complete algorithm. If there is a solution, depth first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

- Depth-first search is useful for:
  - Finding a path between two specified nodes, u and v, of an unweighted graph.
  - Finding a path between two specified nodes, u and v, of a weighted graph.
  - Finding whether a graph is connected or not.
  - Computing the spanning tree of a connected graph.

Thank You

D Y PATIL
UNIVERSITY
NAVI MUMBAI