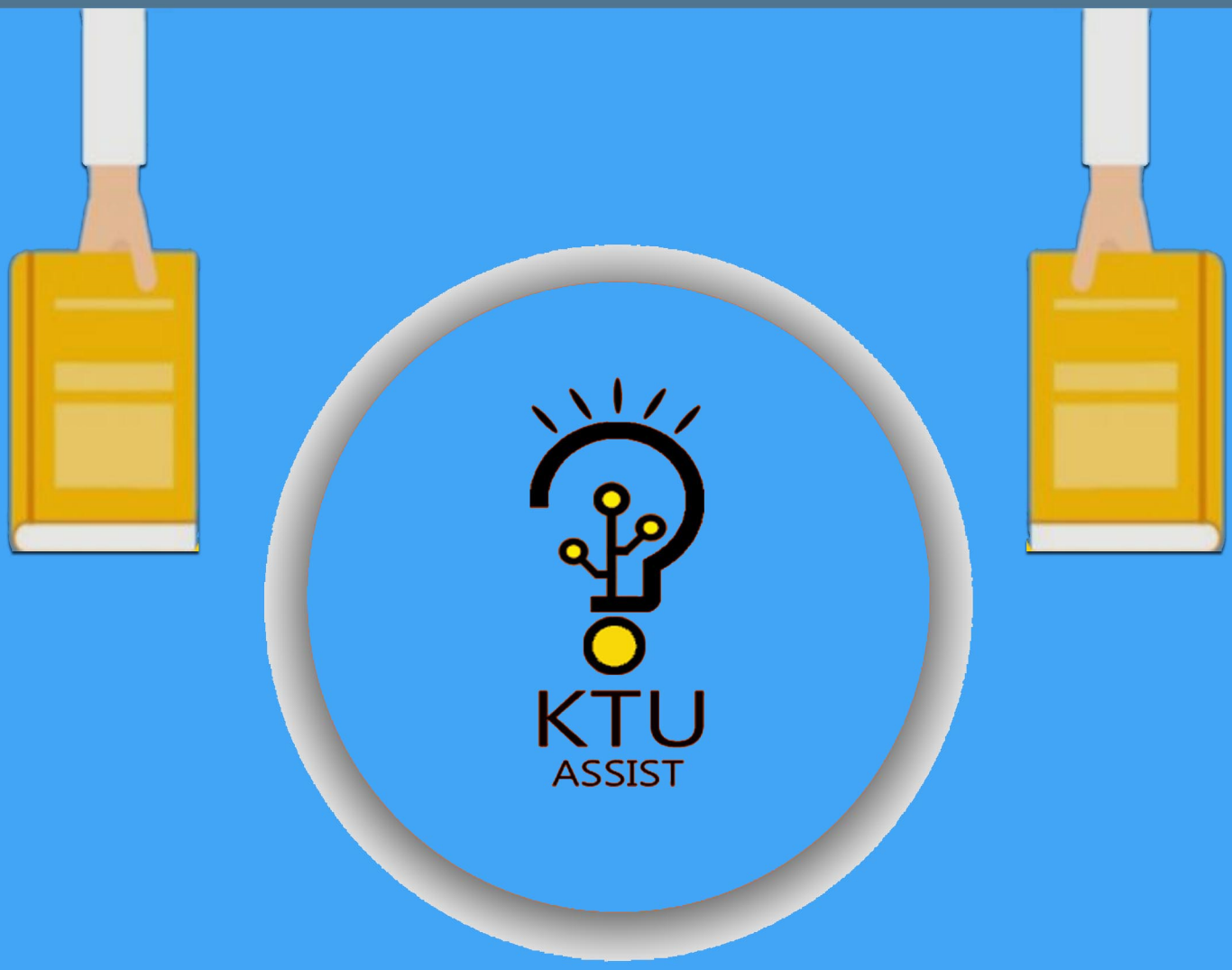
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



Module 3

Graphs

GRAPHS

Ques 1) What is graph? Write its definition.

Ans: Graphs

A graph is a representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by mathematical abstractions called **vertices**, and the links that connect some pairs of vertices are called **edges**.

Definition

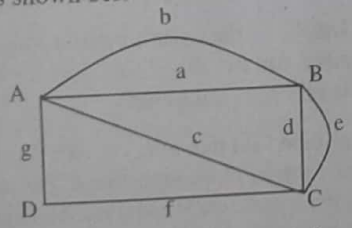
A graph G =(V, E, ϕ) consists of a set of objects V={v_1, v_2, v_3,.....} called **vertices** and another set E={ e_1, e_2, e_3,.....}, whose elements are called **edges**, such that each edge e_k is identified with an unordered pair (v_i, v_j,) of vertices under the mapping ϕ.

For example, the representation of the graph, G = (V, E,ϕ) where:

V = {A, B, C, D}, E = {a, b, c, d, e, f, g} and

$$\phi = \begin{pmatrix} a & b & c & d & e & f & g \\ \{A,B\} & \{A,B\} & \{A,C\} & \{B,C\} & \{B,C\} & \{C,D\} & \{D,A\} \end{pmatrix}.$$

This graph is shown below:



The graph is commonly represented by means of a diagram, in which vertices are represented as points and each edge as a line segment joining its end vertices. Simply a graph can be written as G = (V, E).

Ques 2) Discuss the terminologies used in graph.

Ans: Terminologies of Graph

1) **Vertex (Node):** A node v is a terminal point or an intersection point of edges in a graph.

2) **Edge (Link):** An edge e is a link between two nodes. The link (i, j) is of initial extremity i and of terminal extremity j. In the **figure 3.1** e is the edge and i and j are the vertices.

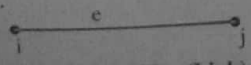


Figure 3.1: Edge (Link)

3) **Directed and Undirected Edge:** In a graph G (V, E) an edge, which is directed from one node to another is called a "directed edge", while an edge, which has no specific direction is called an "undirected edge".

4) **Loop:** An edge of a graph which joins a node to itself is called a "**loop**".

5) **Degree:** The number of edges incident on a vertex is called the degree of that particular vertex.

6) **Incidence and Degree:** An edge e of a graph g is said to be incident with the vertex v if v is an end vertex of e or v is incident with e. The number of edges incident on a vertex v_i with self-loop contend twice, is called the degree, d (v_1) of the vertex v_1. In **figure 3.2**, the degree of vertex u is:

d(u) = 4

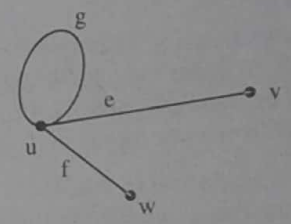


Figure 3.2: Graph G

7) **Adjacent Vertices:** If there exists at least one edge between two vertices v_i and v_j in a graph G, then vertices v_i and v_j are called adjacent vertices.

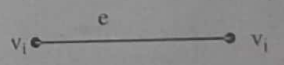


Figure 3.3: Adjacent Vertices

8) **Adjacent Edges:** If two edges e_i and e_j have one end vertex common in a graph G, then e_i and e_j are called adjacent edges.

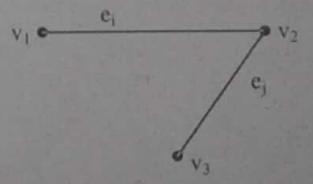


Figure 3.4: Graph G of Adjacent Edges

9) **Parallel Edges:** If two or more edges of a graph g have the same end vertices then these edges are called parallel edges. **For example,** in the **figure 3.5** e_1, e_2, e_3 are parallel edges in the graph. This is also called multiple edges.

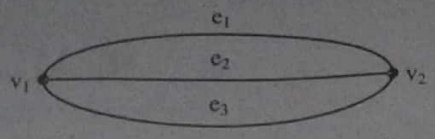


Figure 3.5: Graph G of Parallel Edges

10) **Odd or Even Vertex:** If the degree of a vertex in the graph is even, the vertex is said to be even vertex. If the degree of a vertex in the graph is odd, the vertex is said to be odd vertex.

For example, in the **figure 3.6** $d(v_1) = d(v_3) = 3$ and $d(v_2) = d(v_4) = 2$. So the vertices v_1 & v_3 are odd vertices and v_2, v_4 are even vertices.

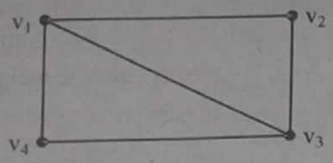


Figure 3.6: Odd or Even Vertex

11) **Isolated Vertex:** A vertex of graph g, which is not the end of any edge is called isolated vertex or in other words, isolated vertices are vertices with zero degree.

12) **Pendant Vertex:** A vertex of degree one is called a pendant vertex.

13) **Path:** A path in a graph represents a way to get from an origin to a destination by traversing edges in the graph. **For example,** ((6, 4), (4, 5), (5, 1)) is also a path in G from node 6 to node 1.

14) **Cycle:** A chain where the initial and terminal nodes are the same and that does not use the same link more than once is a cycle.

15) **Circuit:** A path in where the initial and terminal nodes correspond. It is a cycle where all the links are traveled in the same direction.

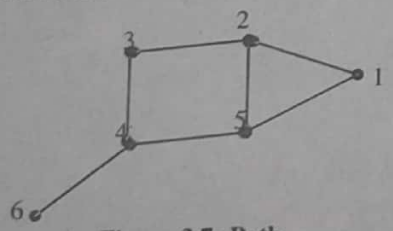


Figure 3.7: Path

Ques 3) Consider the graph G = G (V, E) shown in the figure 3.8.

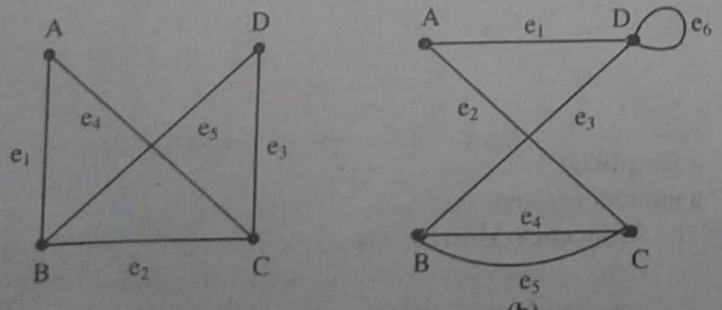


Figure 3.8

1) **Find the number of vertices and edges.**
2) **Are there any multiple edges or loops? If so, what are they?**

Ans: In case of **figure 3.8(a)**

1) G contains four vertices A, B, C, D; and five edges, $e_1, e_2, \ldots\ldots, e_5$
2) There are no multiple edges

In case of **figure 3.8(b)**

1) G contains four vertices A, B, C, D; and six edges, $e_1, e_2, \ldots\ldots, e_6$ (although the edges e_2 and e_3 across at a point, the diagram does not indicate that the intersection point is a vertex of G).
2) The edges e_4 and e_5 are multiple edges since they both have the same endpoints B and C. The edge e_6 is a loop.

Ques 1) Describe the various types of graphs.
Or
Define directed and complete graph.

Ans: Types of Graph

There are various types of graph. They are:

1) **Simple Graph:** A graph that has neither self-loops nor parallel edges are called a simple graph. A simple graph G = (V, E) consists of V, a non-empty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges.

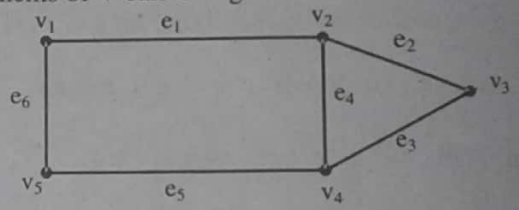


Figure 3.9: Simple Graph

2) **Null Graph:** In the definition of a graph G= (V, E), it is possible for the edge set E to be empty. Such a graph is said to be null graph.

It can also be said that every vertex in a null graph is a isolated vertex. Although the edge set E may be empty, the vertex set V must not be empty; otherwise, there is no graph. In other words, a graph must have at least one vertex. A null graph is also known as empty or trivial graph.

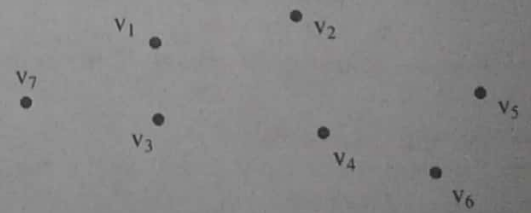


Figure 3.10: Null Graph of Seven Vertices

3) **Directed and Undirected Graph:** A graph in which every edge is directed is called a directed graph or diagraph. A graph in which every edge is undirected is called an undirected graph. Both graphs are shown in **figures 3.11 & 3.12.**

4) **Multiple Graphs:** A graph that contains some multiple edges (parallel edges) is called a multiple graph. In a multiple graph self loops are not allowed.

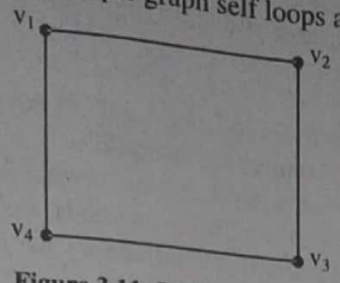


Figure 3.11: Undirected Graph

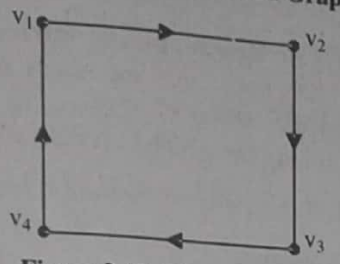


Figure 3.12: Directed Graph

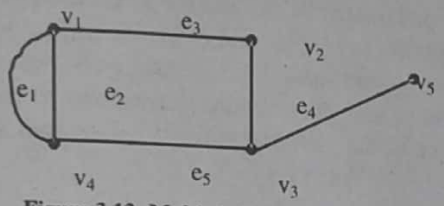


Figure 3.13: Multiple Graph

5) **Regular Graph:** If for some positive integer k, degree (v) = k for every vertex v of the graph g, then g is called k-regular. Or a graph in which all vertices are of equal degree is called a regular graph.

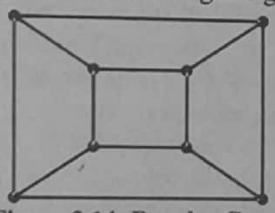


Figure 3.14: Regular Graph

6) **Subgraph:** A graph g is said to be a subgraph of a graph G if all the vertices and all the edges of g are in G, and each edge of g has the same end vertices in g as in G. **For example,** the graph in **figure 3.15(b)** is a subgraph of the one in **figure 3.15 (a)**.

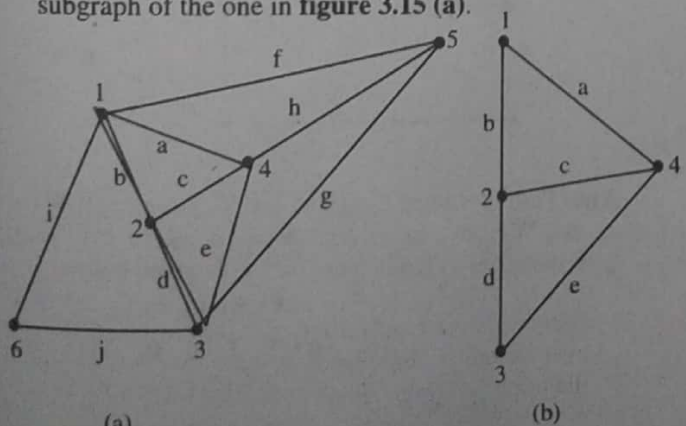


Figure 3.15: Graph (a) and One of its Subgraph (b)

7) **Complete Graph:** A complete graph is a simple graph in which each pair of distinct vertices is joined by an edge. Or in other words a complete graph is a simple graph $G = (V,E)$ such that every pair of distinct vertices in G are connected by exactly one edge—so, for each pair of distinct vertices, either (x, y) or (y, x) (but not both) is in E.

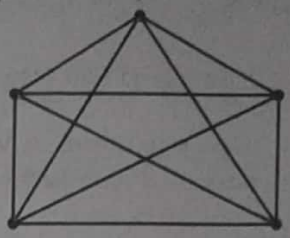


Figure 3.16: Complete Graph

8) **Weighted Graph:** A graph in which weights are assigned to every edge is called a weighted graph.

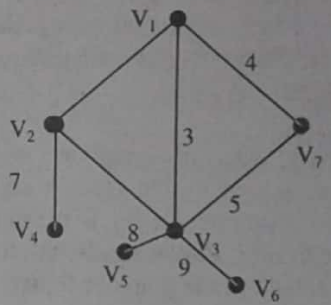


Figure 3.17: Weighted Graph

9) **Bipartite Graph:** A bipartite graph, also called a bi-graph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. A graph is bipartite if all its cycles are of even length.

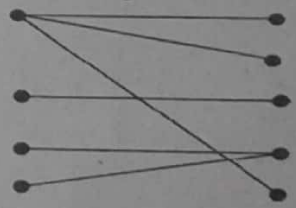


Figure 3.18: Bipartite Graph

Ques 4) In how many ways a graph can be represented?

Or

How adjacency matrix and adjacency list are used for graph representation?

Ans: Representation of Graph

In topological graph theory, the intuitive viewpoint is that a graph is a network of nodes and curved arcs from some nodes to others or to themselves. A graph is called "simplicial" if it has no self-loops or multiple edges, because, in the standard sense of topology, such a graph is a **simplicial 1-complex**.The graphs can be represented by a number of ways like:

1) **Adjacency Matrix:** The adjacency matrix of a graph G with n vertices is an n×n matrix AG such that each entry a_{ij} is the number of edges connecting v_i and v_j. Thus, $a_{ij} = 0$ if there is no edge from v_i to v_j.

The adjacency matrix of a graph G with n vertices and parallel edges/self-loops is an n × n matrix

$A(G) = [a_{ij}]$

Given by

a_{ij} = N, where N is the number of edges between i^{th} and j^{th} vertices

And $a_{ij} = 0$, if there is no edge between them.

Let G be a graph with n vertices $V_1, V_2, \ldots V_n$. The adjacency matrix of G with respect to this particular listing of n vertices is the n x n matrix A (g) $=a_{ij}$, where a_{ij} is the number of edges joining the vertex v_i to v_j. If G has no loops then all the entries of the main diagonal will be 0 and if G has no parallel edges then the entries of A(G) are either 0 or 1. If the graph has no self-loops and no parallel edges, the degree of a vertex equals the number of ones in the corresponding row or column of A(G).

The adjacency matrix of a graph is a matrix with rows and column labeled by the vertices and such that its entry in row i, column j, $i \neq j$, is the number of edges incident on i and j. **For example**, the following is the adjacency matrix of the graph of **figure 3.19.**

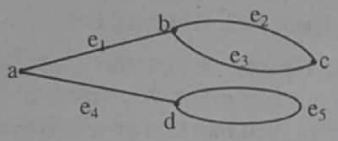


Figure 3.19

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

One of the uses of the adjacency matrix A of a simple graph G is to compute the number of paths between two vertices, namely entry (i, j) of A^n is the number of paths of length n from i to j.

2) **Adjacency List:** In this representation, the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in graph G. The nodes in list i represent the vertices that are adjacent from vertex i.

Each node has two fields, vertex and link. The vertex field contains the indices of the vertices adjacent to vertex i. Each node has a head node. Head nodes are sequential providing easy random access to the adjacency list for any particular vertex.

3) **Adjacency Multi-List:** In the adjacency list representation of an undirected graph each edge (V_i, V_j) is represented by two entries, one on the list for Vi and the other on the list for V_j. In some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined.

This can be accomplished easily if the adjacency lists are actually maintained as multilist (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node, but this node will be in two lists, i.e., the adjacency list for each of the two nodes it is incident to.

4) **Path Matrix or Reach-ability Matrix:** Suppose G is a graph with n nodes and the nodes of G are being ordered and are called $v_1, v_2, v_3, \ldots, v_n$ then the Path matrix P = (p_{ij}) of the graph G is defined as,

$$p_{ii} = \begin{cases} 1, \text{ if there is a path between } v_i \text{ and } v_i \\ 0, \text{ otherwise} \end{cases}$$

Or

A path matrix is defined for a specific pair of vertices in a graph, say (x, y), and is written as P(x, y). The rows in P(x, y) correspond to different paths between vertices x and y, and the columns correspond to the edges in G. That is, the path matrix for (x, y) vertices are:

$P(x, y) = [p_{ij}]$,

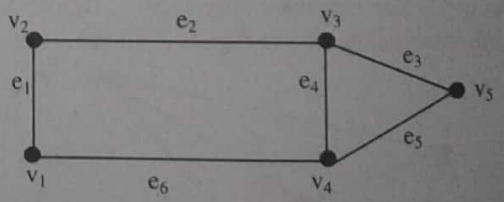
Where

p_{ij} = 1, if j^{th} edge lies in i^{th} path, and
= 0, otherwise.

Some of the observations one can make at once about a path matrix P(x, y) of a graph G are:

i) A column of all 0's corresponds to an edge that does not lie in any path between x and y.
ii) A column of all 1's corresponds to an edge that lies in every path between x and y.
iii) There is no row with all 0's.
iv) The ring sum of any two rows in P(x, y) corresponds to a circuit or an edge-disjoint union of circuits.

Ques 5) Find the path graph between v_2 and v_5 for the following graph:



Ans: Paths between v_2 and v_5 are $W_1 = \{e_2, e_3\}$, $W_2 = \{e_2, e_4, e_5\}$, $W_3 = \{e_1, e_6, e_5\}$ and $W_4 = \{e_1, e_6, e_4, e_3\}$. Thus, the path matrix of G between v_2 and v_5 is given below:

$$W(v_2, v_5) = \begin{array}{c} \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Ques 6) Find the adjacency matrix M of the graph D with four vertices d_1, d2, d3 and d_4 shown in Figure 3.20.

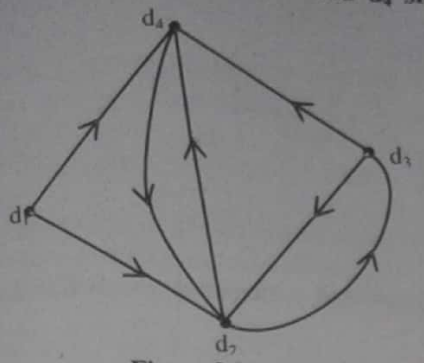


Figure 3.20: Digraph

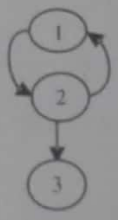
Ans: The adjacency matrix M of graph D.

$$M = \begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{array} \begin{array}{c} \begin{array}{cccc} d_1 & d_2 & d_3 & d_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

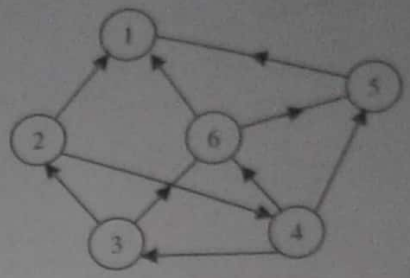
Ques 7) Calculate the adjacency matrix of following graph:

Ans: The following is the adjacency matrix of given graph:

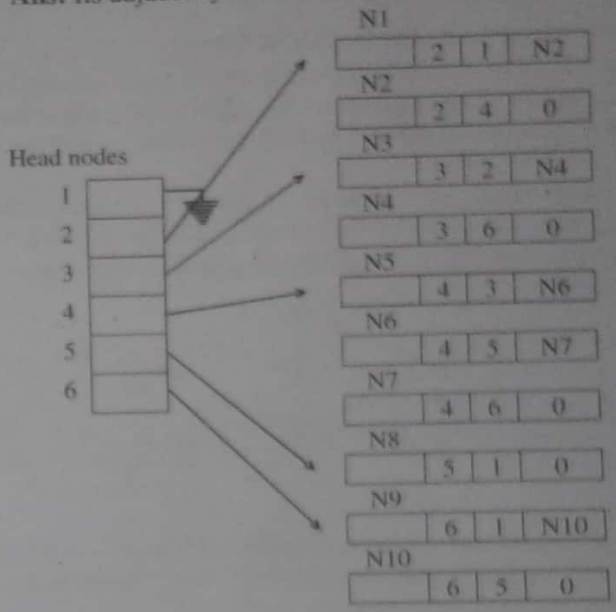
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$



Ques 8) Find the adjacency multi-list for the given graph:



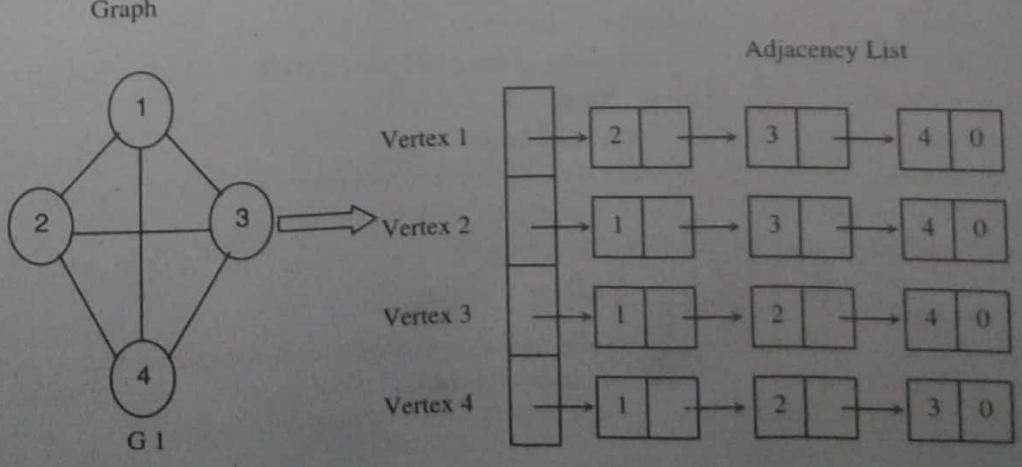
Ans: Its adjacency multi-list representation

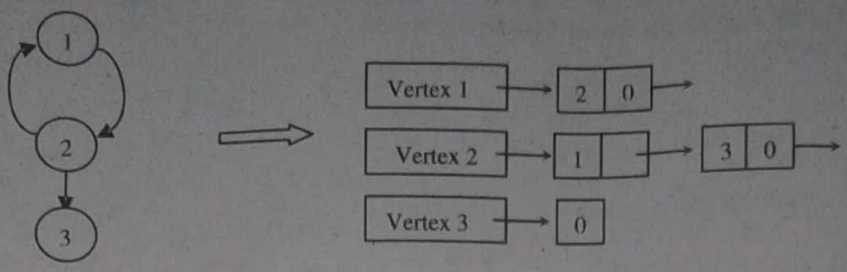


Ques 9) Find the adjacency list of the adjacency matrix given below:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Ans: The adjacency list is as below:





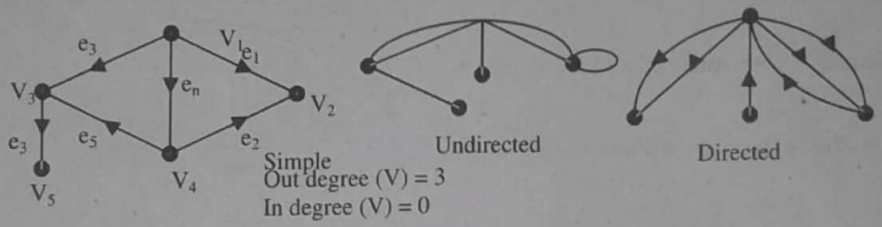
G 2

Ques 10) What do you mean by in degree and out degree of a graph? Proof that a directed graph has an even number of vertices of odd degree.

Ans: In Degree and Out Degree of Graph

In a graph with directed edges the in-degree of a vertex v, denoted by $\deg^-(v)$, is the number of edges with v as their terminal vertex. The out-degree of v, denoted by $\deg^+(v)$, is the number of edges with v as their initial vertex. (Note that a loop at a vertex contributes 1 to both the in-degree and the out-degree of this vertex).

Or in other words, in a directed graph for any node v the number of edges, which have v as initial node, is called the **outdegree** of the node v. The number of edges to have v as their terminal node is called the **indegree** of v and Sum of outdegree and indegree of a node v is called its total degree.



Directed Graph has an Even Number of Vertices of Odd Degree

Let V_1 and V_2 be the set of vertices of even degree and the set of vertices of odd degree, respectively, in an undirected graph G = (V, E). Then,

$$2e = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v)$$

Since deg(v) is even for $v \in V_1$, the first term in the right-hand side of the last equality is even. Furthermore, the sum of the two terms on the right-hand side of the last equality is even, since this sum is 2e. Hence, the second term in the sum is also even. Since all the terms in this sum are odd, there must be an even number of such terms. Thus, there is an even number of vertices of odd degree.

Ques 11) What do you mean by graph traversal?

Or

Illustrate the importance of various traversing techniques in graph.

Ans: Graph Traversal

The general technique for graph traversing is given below:

Step 1) Initially all the nodes of the graph are marked as 'unreached'.

Step 2) After selecting a start node from where the traversing of the graph starts, mark 'reached'.

Step 3) The node marked as 'reached' is placed on the 'ready list'.

Step 4) Now, the node from the 'the ready list' is selected and processed.

Step 5) The processed node is deleted, and the adjacent node to the deleted node is marked as 'reached' only if they are marked as 'un-reached'. These 'reached' nodes are then added to the 'ready list'.

Step 6) The whole process continues until the 'ready list' becomes empty.

Importance of Graph Traversal

The notion of graph traversal is of fundamental importance to solving many computational problems. In many modern applications involving graph traversal such as those arising in the domain of social networks, Internet based services, fraud detection in telephone calls etc., the underlying graph is very large and dynamically evolving.

The graph traversal is used to decide the order of vertices to be visit in the search process. A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

The advantage (and application) of BFS is that in unweighted graphs it can be used to construct a shortest path from u to v. The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph.

Ques 12) Write DFS algorithm to traverse a graph.

Ans: DFS (Depth-First Search Algorithm)

The idea of this algorithm is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible. This algorithm uses a stack (initially empty) to store vertices of the graph.

Depth-first search is a generalization of traversing trees in preorder. The starting vertex may be determined by the problem or may be chosen arbitrarily.

While traversing vertices starting from the initial vertex, a dead end may be reached. A dead end is a vertex such that all its neighbours, that is, vertices adjacent to it, have already been visited. At a dead end we back up along the last edge traversed and branch out in another direction.

Algorithm: Depth First Search

Step 1) Choose an arbitrary node in the graph, designate it as the search node, and mark it as visited.

Step 2) From the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited as yet. Mark it as visited new search node.

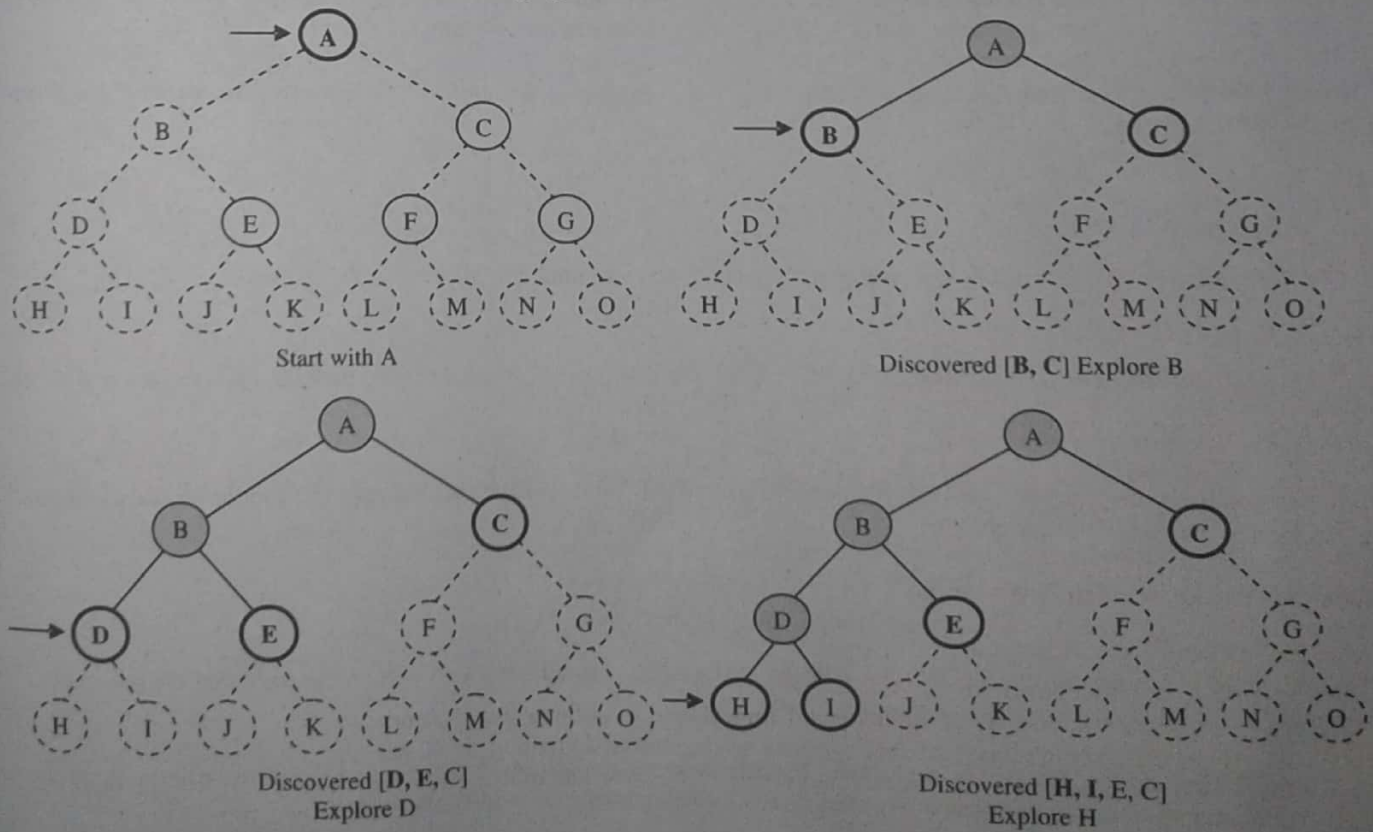
Step 3) Repeat Step 2 using the new search node it there are no nodes satisfying on Step 2, return to the previous search node and continue the process from there.

Step 4) When a return to the previous search node in Step 3 is impossible, the search from the original chosen search node is complete.

Step 5) If there are any nodes in the graph which are still unvisited, choose any node that has not been visited and repeat Step 1 through Step 4.

Step 6) Stop

The process of DFS is illustrated in the following **figure 3.21**:



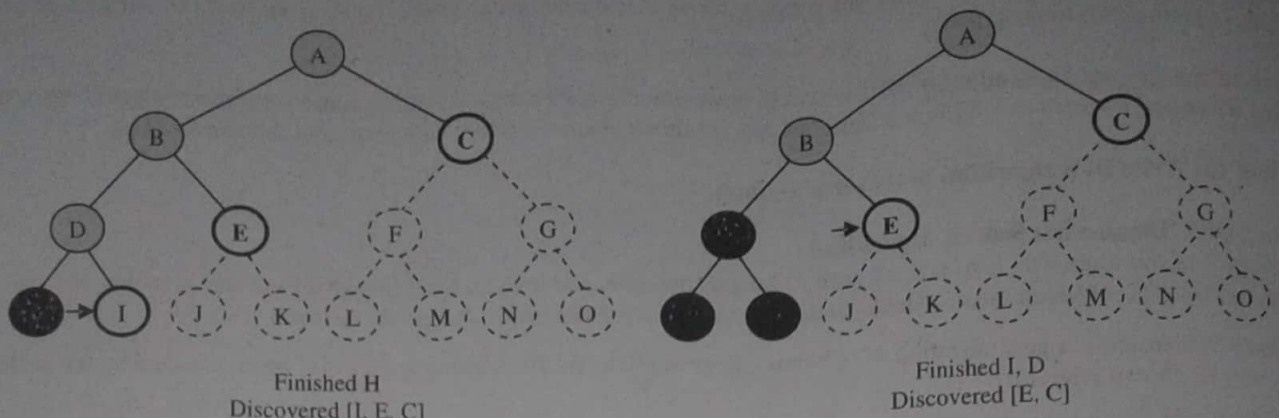
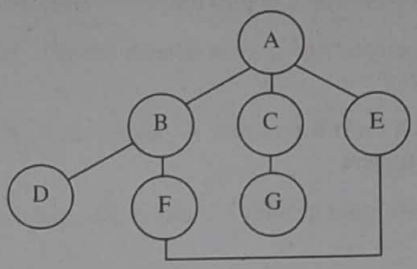


Figure 3.21

Similarly proceed for other nodes.

Ques 13) Apply DFS (Depth-First Search) for the following graph and find the shortest path:



Ans: Starting DFS (Depth-First Search) at node A, assuming that the left edges in the **above graph** are chosen before right edges and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), visiting of the nodes will occur in the following order: A, B, D, F, E, C, G.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc., forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it precedes left-to-right as above:

0: A

1: A (repeated), B, C, E

(Note that iterative deepening has now seen C, when a conventional depth-first search did not.)

2: A, B, D, F, C, G, E, F

(Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice).

3: A, B, D, F, E, C, G, E, F, B

For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

Ques 14) Define breadth first search.

Ans: BFS (Breadth-First Search)
The idea is to start with vertex V_1 as root, add the vertices that are adjacent to V_1, then the ones that are adjacent to the latter and have not been visited yet, and so on.

In a breadth-first search, the graph is searched breadth wise by exploring all the nodes adjacent to a node. A queue is a convenient data structure to keep track of nodes that are visited during a breadth-first search.

Algorithm: Breadth First Search

Step 1) Start with any vertex and mark it as visited.

Step 2) Using the adjacency matrix of the graph, find a vertex adjacent to the vertex in step 1. Mark it as visited.

Step 3) Return to the vertex in Step 1 and move along an edge towards an unvisited vertex, and mark the new vertex as visited.

Step 4) Repeat Step 3 until all vertices adjacent to the vertex, as selected in Step 2, have been marked as visited.

Step 5) Repeat Step 1 through Step 4 starting from the vertex visited in Step 2, then starting from the nodes visited to Step 3 in the order visited. If all vertices have been visited, then continue to next step.

Step 6) Stop

Ques 15) Apply BFS of on following graph G (Figure 3.22)

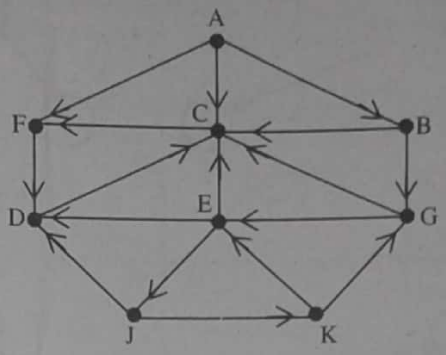


Figure 3.22: Digraph G

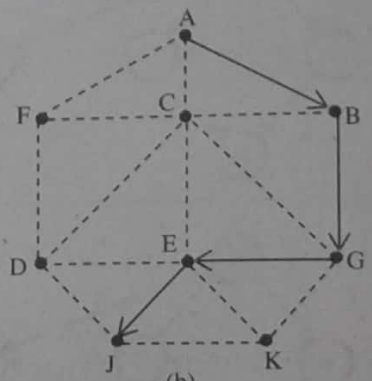
Ans: Starting at the vertex A, and stop as soon as J is encountered, i.e., added to the waiting list. **Table below** shows the sequence of waiting lists in QUEUE and the vertices being processed up to the time vertex J is encountered. Working backwards from J to obtain the desired path,

$$E^J \leftarrow G^E \leftarrow B^G \leftarrow A^B \leftarrow A \quad \text{or} \quad A \rightarrow B \rightarrow G \rightarrow E \rightarrow J$$

This sequence is pictured in **figure 3.23** Thus a flight from city A to city J will make three intermediate stops, at B, G and E.

Vertex	Queue
	A
A	A^B, A^C, A^F
A^F	F^D, A^B, A^C
A^C	F^D, A^B
A^B	B^G, F^D
F^D	B^G
B^G	G^E
G^E	E^J

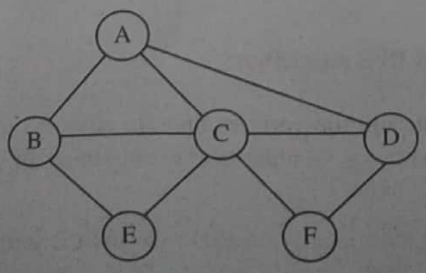
(a)



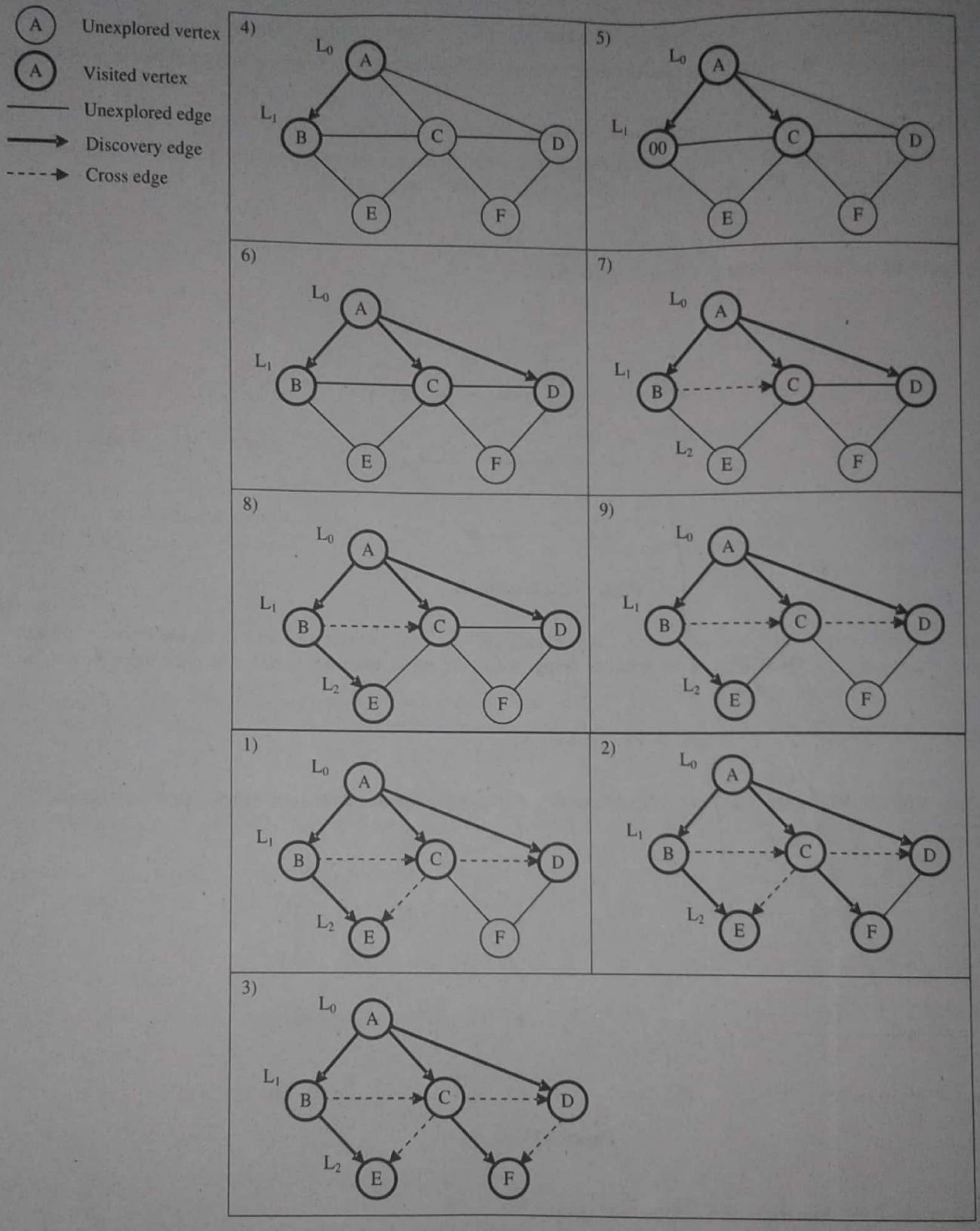
(b)

Figure 3.23

Ques 16) Apply the BFS algorithm on the following graph:



Ans:



Ques 17) What is the complexity of DFS and BFS algorithm?

Ans: Complexity of DFS and BFS Algorithm

1) **DFS:** If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to V is O(n). Since at most n vertices are visited, the total time is $O(n^2)$.

2) **BFS:** Each vertex visited gets into the queue exactly once, so the loop is iterated at most n times. If an adjacency matrix is used then the for loop takes O(n) times for each vertex visited. The total time is therefore $O(n^2)$.

Ques 18) Define applications of BSF and DFS methods.

Ans: Applications of Depth First Search

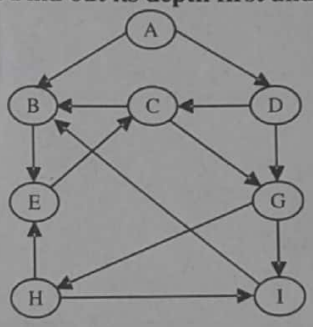
Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2) Detecting cycle in a graph. A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)
3) Path Finding. We can specialize the DFS algorithm to find a path between two given vertices u and z.
4) Topological Sorting. Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.

Applications of Breadth First Traversal

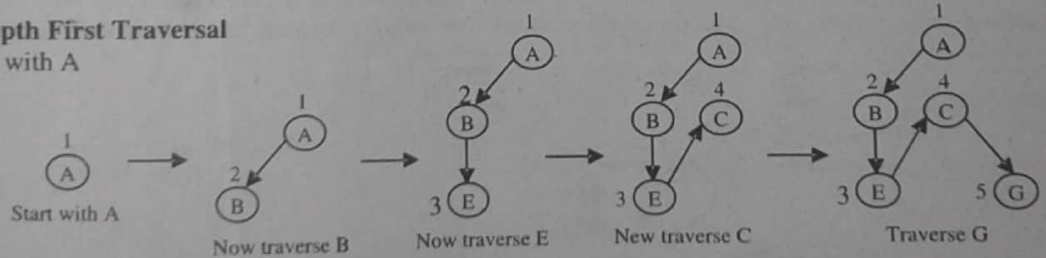
1) Shortest Path and Minimum Spanning Tree for unweighted graph. In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3) Crawlers in Search Engines. Crawlers build index using Breadth First The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4) Social Networking Websites. In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5) GPS Navigation systems. Breadth First Search is used to find all neighboring locations.
6) Broadcasting in Network. In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7) In Garbage Collection. Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8) Cycle detection in undirected graph. In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

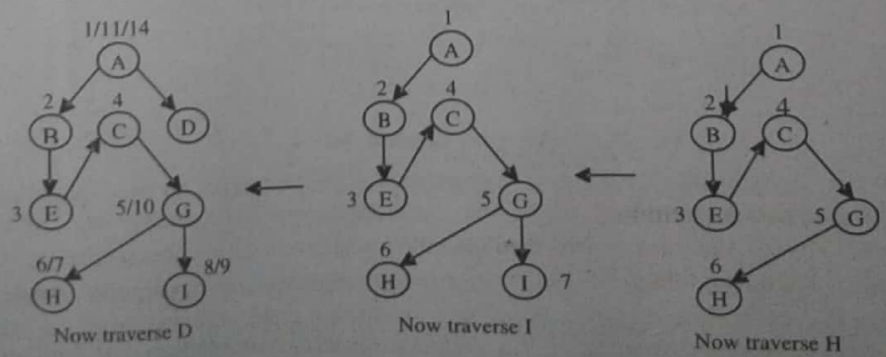
Ques 19) Consider the graph given below. Find out its depth first and breadth first traversal scheme.



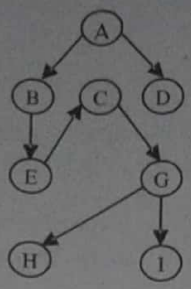
Ans: Depth First Traversal

Let start with A

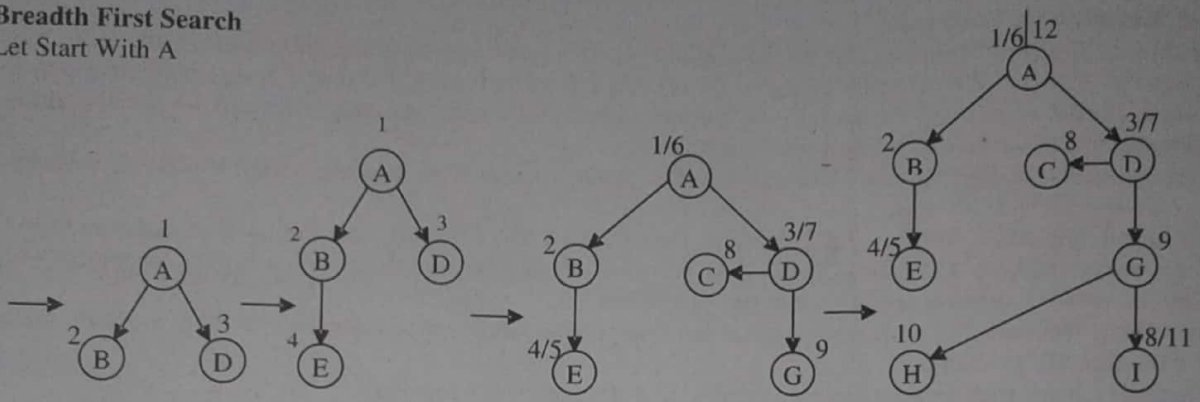




Hence the DFS is

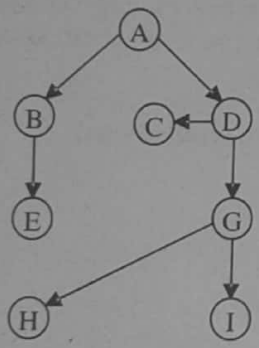


Breadth First Search
Let Start With A



Now traverse B and D | Now traverse E | Now traverse C and G | Now traverse H and I

Hence the BFS is



Ques 20) What are the various applications of graph?

Ans: Applications of Graphs

1) **Model of WWW:** The model of World Wide Web (www) can be represented by a graph (directed) wherein nodes denote the documents, papers, articles, etc., and the edges represent the outgoing hyperlinks between them as shown in **figure 3.24**.

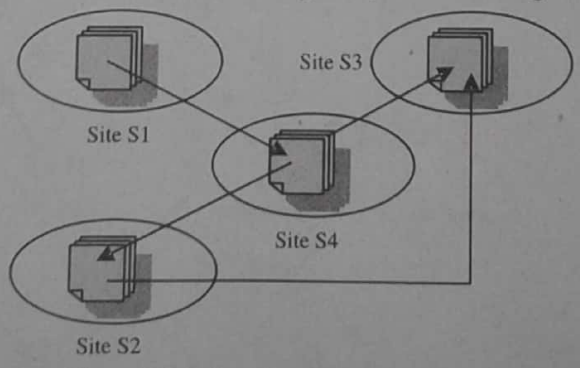


Figure 3.24: Graphic Representation of www

It may be noted that a page on site S1 has a link to page on site S4. The pages on S4 refers to pages on S2 and S3, and so on. This hyperlink structure is the basis of the web structure similar to the web created by a spider and, hence, the name world wide web.

2) **Resource Allocation Graph:** In order to detect and avoid deadlocks, the operating system maintains a resource allocation graph for processes that are active in the system. In this graph, the processes are represented as rectangles and resources as circles. Number of small circles within a resource node indicates the number of instances of that

resource available to the system. An edge from a process P to a resource R, denoted by (P, R), is called a **request edge**. An edge from a resource R to a process P, denoted by (R, P), is called an **allocation edge** as shown in **figure 3.25**:

It may be noted from the resource allocation graph of **figure 3.25** that the process P1 has been allocated one instance of resource R1 and is requesting for an instance of resource R2. Similarly, P2 is holding an instance of both R1 and R2. P3 is holding an instance of R2 and is asking for an instance of R1. The edge (R1, P1) is an allocation edge whereas the edge (P1, R2) is a request edge.

An operating system maintains the resource allocation graph of this kind and monitors it from time to time for detection and avoidance of deadlock situations in the system.

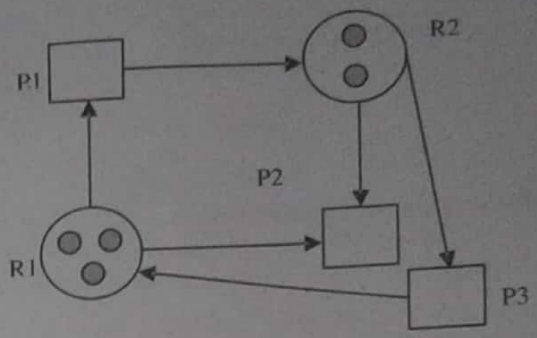


Figure 3.25: Resource Allocation Graph

3) **Coloring of Maps:** It is an interesting problem wherein it is desired that a map has to be colored in such a fashion that no two adjacent countries or regions have the same color. The constraint is to use minimum number of colours.

A map (**figure 3.26**) can be represented as a graph wherein a node represents a region and an edge between two regions denote that the two regions are adjacent.

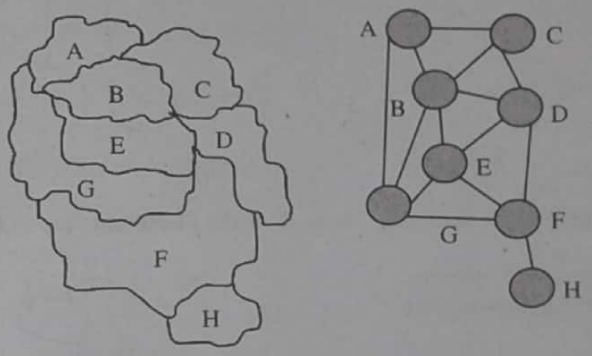


Figure 3.26: Graphic Representation of a Map

4) **Scene Graphs:** The contents of a visual scene are also managed by using graph data structure. Virtual Reality Modeling Language (VRML) supports scene graph programming model. This model is used by MPEG-4 to represent multimedia scene composition.

Ques 21) What is spanning tree and minimum cost spanning tree?

Ans: Spanning Tree

A tree T is said to be spanning tree of a connected graph G if T is a subgraph of G and T contains all vertices of G. i.e. if $G = (V_1, E_1)$ is a connected graph, then the tree $T = (V_2, E_2)$ is a spanning tree of G if $V_1 = V_2$. Since spanning trees are the largest (with maximum number of edges) trees among all trees in G, it is also quite appropriate to call a spanning tree a maximal tree subgraph or maximal tree of G.

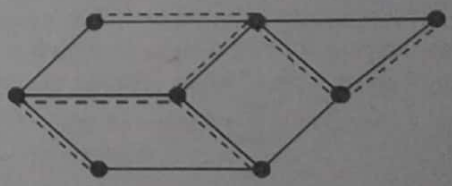


Figure 3.27: Spanning Tree of a Graph G

A spanning tree is defined only for connected graph, because a tree is always connected, and in a disconnected graph of n vertices one cannot find a connected subgraph with n vertices. Each component of a disconnected graph, however, does have a spanning tree. Thus disconnected graph with k components has a spanning forest consisting of k spanning trees.

Following **figure 3.27** and **figure 3.28** shows a connected graph and its spanning tree.

Every connected graph has a spanning tree which can be obtained by removing edges until the resulting graph becomes acyclic i.e. where there are no cycles.

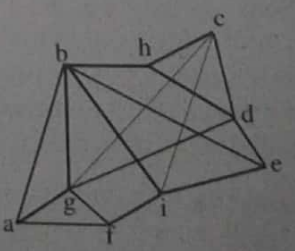


Figure 3.28: Spanning Tree

Figures 3.29 and 3.30 shows a connected graph and its spanning tree.

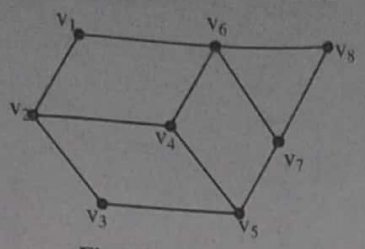


Figure 3.29: Graph G

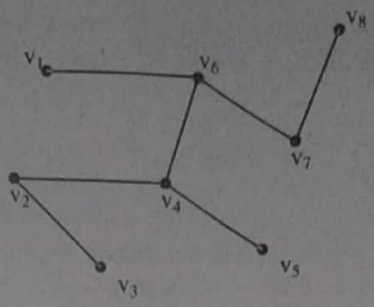


Figure 3.30: Spanning Tree of Graph G

Minimum Cost Spanning Tree

The minimum-cost spanning tree is defined on a weighted, undirected, connected graph. A minimum-cost spanning tree in which the sum of the weights on the edges is a minimum.

It is a tree from the set of spanning trees, which has minimum weight. A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.

For example, the **figure 3.31** shows the minimum spanning trees.

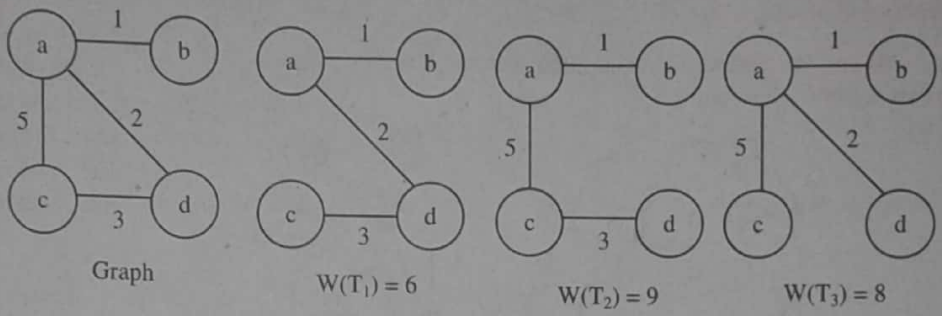


Figure 3.31: Graph and Its Spanning Trees; T_1 is the Minimum Spanning Tree

Ques 22) What do you understand by single-source shortest path? Also discuss about the existence of single-source shortest path.

Ans: Single-Source Shortest Path

If there is a path from vertex u to vertex v in a network G, any path of minimum length from u to v is a **shortest path** from u to v, and its weight is the Shortest Distance (SD) from u to v. The problem of finding shortest paths in networks is called **shortest path problem**.

In a shortest path problem, one have an undirected graph G = (V,E), a weight function w. E→R, mapping every edge on to an item of R. The weight of a path p = $(p_0,\ldots,p_n)$ is the sum of the weight of edges forming the path:

$$w(p) = \sum_{i=1,\ldots,n} w(v(i-1), v(i))$$

The shortest path weight from vertex u to vertex v is defined as follows:

Definition: Given a weighted, directed graph G = (V,E) and a pair of vertices n, m in V for which a path exists in G, a shortest path weight between n and m, δ (m, n), is the minimum weight w(p) of the weight of the paths from n to m, if a path from n to m exists, ∞ otherwise.

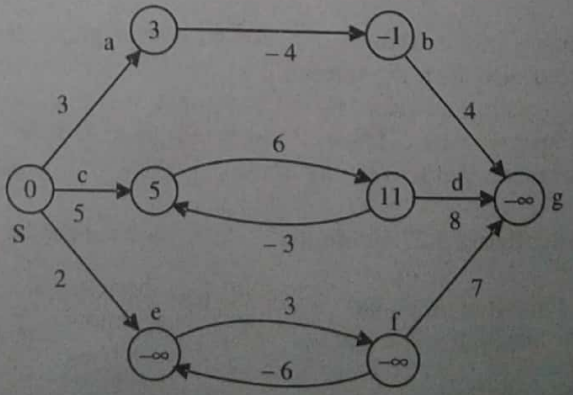
Existence of Shortest Path

If some path from s to v contains a **negative cost cycle** then, there does not exist a shortest-path. Otherwise, there exists a shortest s – v that is simple.

Cost of C < 0

If there is a negative-weight cycle on some path from s to v, we define δ(s, v) = –∞.

For example, consider the figure. There are infinitely many paths from s to c: ⟨s,c⟩, ⟨s,c,d,c⟩, ⟨s,c,d,c,d,c⟩, and so on. Because the cycle ⟨c,d,c⟩ has weight 6+ (–3) = 3 > 0, the shortest-path from s to c is ⟨s,c⟩, with weight δ(s,c) = 5. Similarly, there are infinitely many paths from s to e: ⟨s,e⟩, ⟨s,e,f,e⟩, ⟨s,e,f,e,f,e⟩, and so on. Since the cycle ⟨e,f,e⟩ has weight 3 + (–6) = –3 < 0, however, there is no shortest-path from s to e. By traversing the negative-weight cycle ⟨e,f,e⟩ arbitrarily many times, we can find paths from s to e with arbitrarily large negative weights, and so δ(s,e) = –∞.



Ques 23) Explain how shortest path can be represented.

Ans: Representing Shortest Path

Given a graph G = (V,E), we maintain for each vertex $v \in V$ a predecessor $\pi[v]$ that is either another vertex or NIL. During the execution of a shortest-paths algorithm, however, the π values need not indicate shortest-paths.

As in breadth-first search, we shall be interested in the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by the values π. Here again, we define the vertex set V_π, to be the set of vertices of G with non-NIL predecessors, plus the source s:

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}\ \}$$

The directed edge set E_π is the set of edges induced by the π values for vertices in V_π:

$$E_\pi = \{(\pi[v],v) \in E : v \in V_\pi - \{s\}\ \}$$

A shortest-paths tree rooted at s is a directed sub graph G′ = (V′, E′), where $V' \subseteq V$ and $E' \subseteq E$, such that

1) V′ is the set of vertices reachable from s in G,
2) G′ forms a rooted tree with root s, and
3) For all $v \in V'$, the unique simple path from s to v in G′ is a shortest-path from s to v in G.

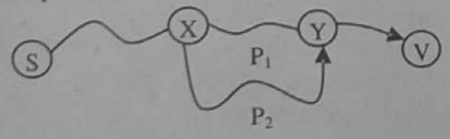
Shortest-paths are not necessarily unique, and neither are shortest-paths trees.

Ques 24) What are the main properties of shortest path?

Ans: Properties of Shortest Path

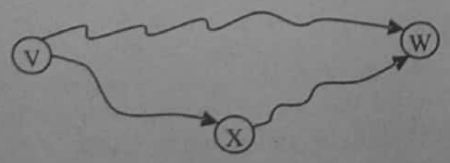
The properties of shortest path are given below:

1) **Optimal Substructure Property:** All sub-paths of shortest-paths are shortest-paths.



Let P_1 be x – y sub-path of shortest s – v path P. Let P_2 be any x – y path. Then cost of $P_1 \leq$ cost of P_2, otherwise P not shortest s – v path.

2) **Triangle Inequality:** Let d(v,w) be the length of the shortest-path from v to w. Then,
$d(v,w) \leq d(v,x) + d(x,w)$



3) **Relaxation:** The single-source shortest-paths algorithms are based on a technique known as relaxation, a method that repeatedly decreases an upper bound on the actual shortest-path weight of each vertex until the upper bound equals the shortest-path weight. For each vertex $v \in V$, we maintain an attribute d[v], which is an upper bound on the weight of a shortest-path from source s to v. We call d[v] a shortest-path estimate. We initialize the shortest-path estimates and predecessors by the following procedure:

Algorithm: INITIALIZE-SINGLE-SOURCE (G,s)
Step 1: for each vertex $v \in V[G]$
Step 2: do $d[v] \leftarrow \infty$
Step 3: $\pi[v] \leftarrow NIL$
Step 4: $d[s] \leftarrow 0$

After initialization, $\pi[v] = NIL$ for all $v \in V$, $d[v] = 0$ for v = s, and $d[v] = \infty$ for $v \in V - \{s\}$.

The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest-path to v found so far by going through u and, if so, updating d[v] and $\pi[v]$. A relaxation step may decrease the value of the shortest-path estimate d[v] and update v's predecessor field $\pi[v]$. The following code performs a relaxation step on edge (u,v).

Algorithm: RELAX (u,v,w)
Step 1: if $d[v] > d[u] + w(u,v)$
Step 2: then $d[v] \leftarrow d[u] + w(u,v)$
Step 3: $\pi[v] \leftarrow u$

In Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once. In the Bellman-Ford algorithm, each edge is relaxed several times

Ques 25) What are the different variants of single-source shortest path? What are the different methods of single-source shortest path? List them.

Ans: Variants of Single Source Problem

The algorithm for the single source problem includes the following variants given below:

1) **Single-Source Shortest Path Problem**, in which we have to find shortest paths from a source vertex *v* to all other vertices in the graph.
2) **Single-Destination Shortest Path Problem**, in which we have to find shortest paths from all vertices in the graph to a single destination vertex *v*. This can be reduced to the single-source shortest path problem by reversing the edges in the graph.
3) **All-Pairs Shortest Path Problem**, in which we have to find shortest paths between every pair of vertices v, v' in the graph.

Methods for Single Source Shortest Path Problem

Two famous algorithms for solving the single source shortest path problem are:

1) Bellman-Ford Algorithms
2) Dijkstra's Algorithm

Ques 26) Describe the Dijkstra's algorithm in detail. What is the runtime complexity of Dijkstra's algorithm.

Ans: Dijkstra's Algorithm

Let G = (V, E), where V = {1,2, . . . , n} is a directed network in which the weight of every arc is non-negative. This algorithm can be used to find the SP(Shortest Path) and SD(Shortest destination) from any fixed vertex (say vertex 1) to any vertex i if there is a directed path from 1 to i.

Let a(i,j) be the weight of the arc from i to j. If there is no arc from i to j, a(i,j) is + ∞. Each vertex i is assigned a label that is either permanent or tentative.

The permanent label L(i) is the SD from 1 to i. The tentative label L′(i) is an upper bound of L(i). At each stage of the algorithm, P is the set of vertices with permanent labels and T is the set of vertices with tentative labels. Initially, P is the set {1} with L(1) = 0 and L′(j) = a(1,j) for all j. The procedure terminates when P = V.

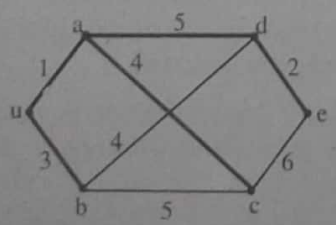
Iterations in Dijkstra's Algorithm

Each iteration consists of two steps:

Step 1: Find a vertex k in T for which L′(k) is finite and minimum. If there is no k, stop; there is no path from 1 to any unlabeled vertex. Otherwise, declare k to be permanently labeled, and adjoin k to P. Stop if P = V. Label the arc (i, k), where i is a labeled vertex that determines the minimum value of L′(k).

Step 2: Replace L′(j) by the smaller value of L′(j) and L(k) + a(k, j) for every j in T. Go to step 1. If G has n vertices and it is possible to obtain the SD from the starting vertex v to every other vertex, the set of (n – 1) arcs obtained by this method will form an arborescence rooted at v that will give both the SD and SP from v to every other vertex.

For example, in the weighted graph below, shortest paths from u are found to the other vertices in the order a, b, c, d, e, with distances 1, 3, 5, 6, 8 respectively. To reconstruct the paths, there is only need the edge on which each shortest path arrives at its destination, because the earlier portion of a shortest u, z-path that reaches z on the edge vz is a shortest u, v-path.



The algorithm can maintain this information by recording the identity of the "selected vertex" whenever the tentative distance to z is updated. When z is selected, the vertex that was recorded when t(z) was last updated is the predecessor of z on the u, z-path of length d(u, z). In this example, the final edges on the paths to a, b, c, d, e generated by the algorithm are ua, ub, ac, ad, de, respectively, and these are the edges of the spanning tree generated from u.

Dijkstra's Algorithm works also for digraphs, generating an out-tree rooted at u if every vertex is reachable from u. The proof works for graphs and for digraphs. The technique of proving a stronger statement in order to make an inductive proof work is called "loading the induction hypothesis".

Algorithm: Dijkstra

Dijkstra's algorithm computes a solution to the single source shortest path problem for a weighted graph G = (V,E), provided that each edge in E has a non-negative weight.

To keep track of progress, the algorithm colors each vertex white or black. At first, all vertices are white. When the shortest path from the source s to a vertex v has been found, v is blackened. That is, for every black vertex v, d[v] is the weight for a shortest path from s to v. The algorithm iteratively selects the white vertex u which has the least shortest-path estimate, blackens u, and relaxes all edges leaving from u.

The graph is represented by an adjacency lists. Other data structures have been employed. In particular,

1) color[u], storing the color of the vertex u.
2) pred[u], containing the parent of the vertex u.
3) d[u], storing the shortest-path estimate for u.
4) Q, a priority queue on the d[u] values, storing the white vertices.

Algorithm: Single-source-Dijkstra(G,w,s)

Step 1: for each vertex u in V
Step 2: d[v] = infinity
Step 3: pred[v] = NULL
Step 4: color[v] = white
Step 5: d[s] = 0
Step 6: Add all the vertices in V to Q
Step 7: While (! = ISEMPTY(Q))
Step 8: Extract from Q the vertex u such that d[u] is the minimum
Step 9: color[u] = black
Step 10: for each node v adjacent to u with color[v] = white do
Step 11: if d[v] > d[u] + w(u,v) then
Step 12: d[v] = d[u] + w(u,v)
Step 13: pred[v] = u

Explanation

1) **Lines 1-4** initialize the date structures: for each vertex, except the source, the color is set to white, d[u] is set to infinity and parent to NULL.
2) **Line 5** sets d[u] to 0 for the source.
3) **Line 6** inserts all the vertices into the priority queue.
4) **Lines 7-13** extract the white vertex u with the smallest, shortest path estimate, blacken it and relax all edges outgoing from it.

5) **Lines 10-13** execute the relaxation process, decreasing d[v] and setting pred[v] to u.

Analysis of Dijkstra Algorithm

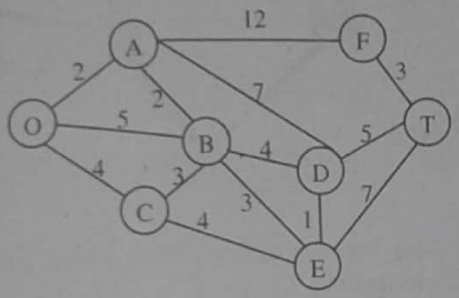
1) Lines 1-4 take $O(V)$ time.

2) If we implement the priority queue with a Fibonacci heap, the $|V|$ applications of the extraction from Q of the vertex with the minimum d take $O(|V|\lg|V|)$ amortized time.

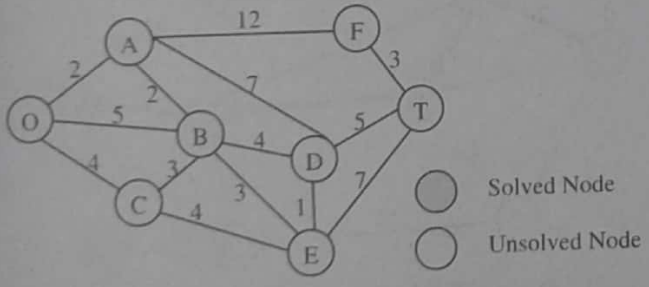
3) Each vertex v is whitened once, and for loop of lines 10-13 examine each edge in the adjacency lists exactly once.

4) Since the total number of edges is $|E|$, lines 10-13 are executed $|E|$ times, each one taking O(1). Thus, the total running time is $O(|V|\lg|V|+|E|)$.

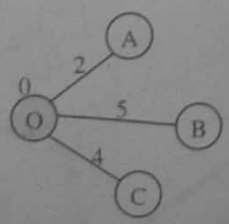
Ques 27) Find the shortest path for the following network using Dijkstra's algorithm.



Ans: Figure below shows a network whose arcs are labelled with distances between the two nodes, it is connecting. We will be finding the shortest route from origin, O to the destination, T using Dijkstra's Algorithm.

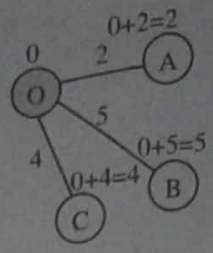


Initialise by displaying the origin as solved. We will label it with O, since it is 0 units from the origin.

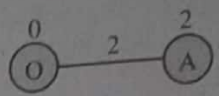


Identify all unsolved nodes connected to any solved node. For each arc connecting a solved and unsolved node, calculate the distance.

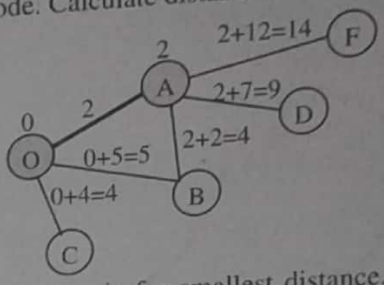
Distance = Distance to the solved node + Length of arc



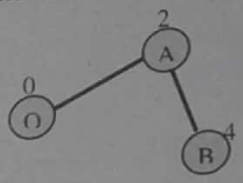
Choose the smallest distance. Here A is on the smallest distance; hence change node A to solved and labels it with the distance. Add the arc to the arc set.



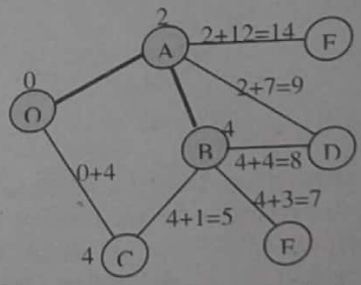
Repeat these steps until we have reached the destination node. Identify all unsolved nodes directly connected to a solved node. Calculate distance of each connecting arc.



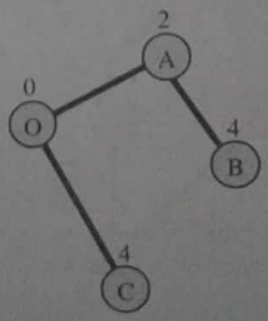
Now we have a tie for smallest distance. In this case we will choose one of them arbitrarily. We will change node B to solved and label it with the distance. Add the arc to the arc set.



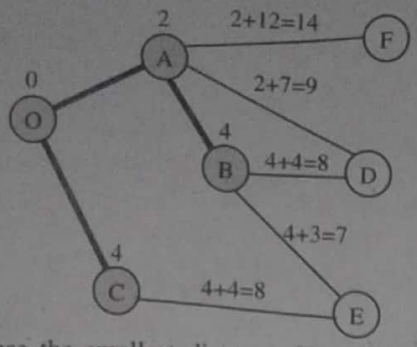
This is not our destination, so we will continue. Identify all unsolved nodes that are directly connected to a solved node. Calculate distance of each connecting arc.



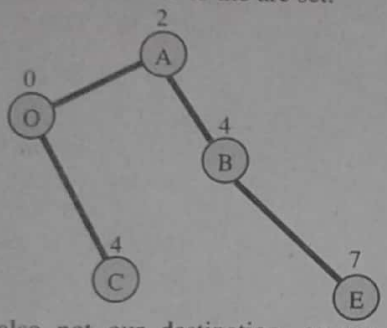
Choose the smallest distance. Now C is on smallest distance, hence change the node C to solved and labels it with the distance. Add the arc to the arc set.



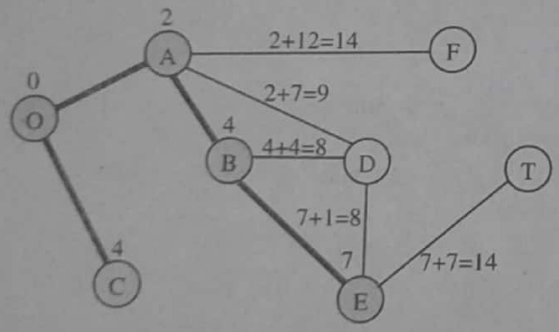
This is also not our destination, so we will continue. Identify all unsolved nodes that are directly connected to a solved node. Calculate distance of each connecting arc.



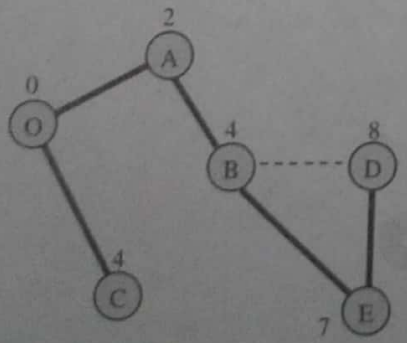
Choose the smallest distance. Here E is on the smallest distance; hence change node E to solved and labels it with the distance. Add the arc to the arc set.



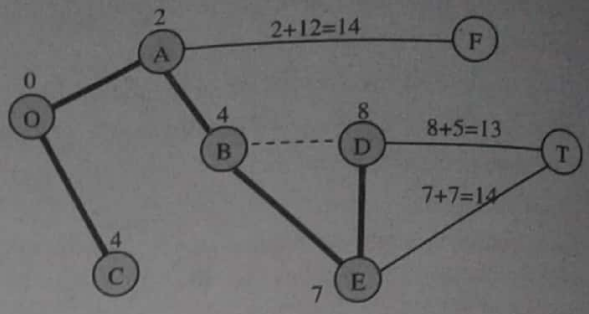
This is also not our destination, so we will continue. Identify all unsolved nodes directly connected to a solved node. Calculate the distance of each connecting arc.



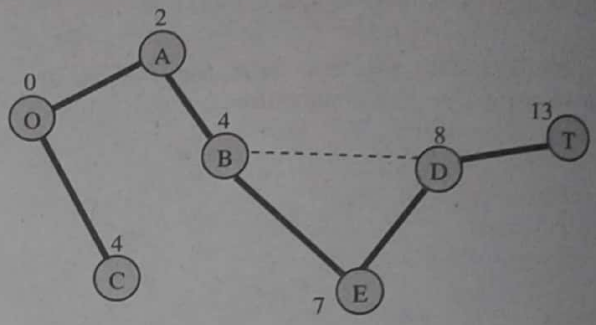
Choose the smallest distance. But here we have a tie for smallest distance. Since both arcs are connecting to the same unsolved node, we will choose one to add to the arc set and mark the alternate using a dotted line. Change node D to solve and labels it with the distance.



This is also not our destination, so we will continue. Identify all unsolved nodes directly connected to a solved node. Calculate the distance of each connecting arc.



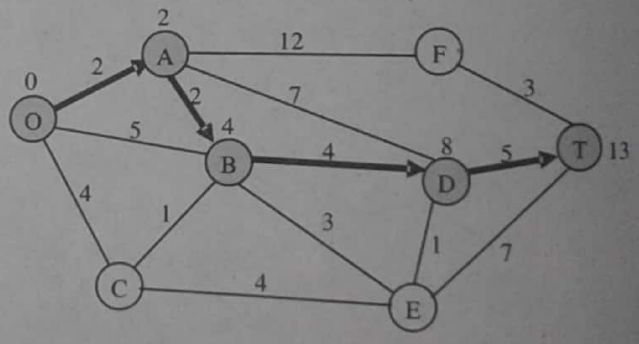
Choose the smallest distance. Now the T is on the smallest distance, hence we have to change node T to solved and label it with the distance. Add arc to the arc set.



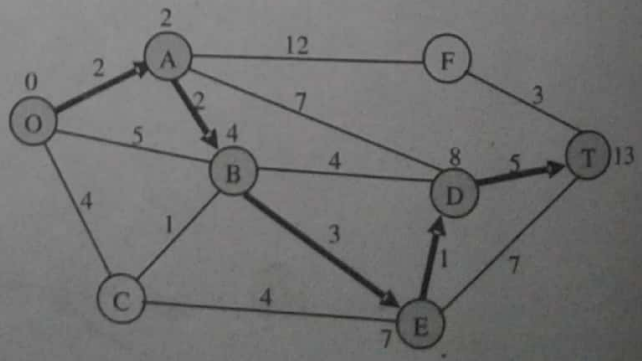
All the nodes are not solved here, hence there is an **incomplete** route O to F.

The two shortest paths from O to T are:

1) O– A – B – D – T



2) O – A – B – E – D – T



Ques 28) What is topological sorting? Discuss in detail.
Or
Discuss the complexity analysis of single-source shortest paths in directed acyclic graphs.

Ans: Topological Sort

Topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that, for every edge uv, u comes before v in the ordering. **For example,** the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks.

A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Topological Sort Algorithm

Step1) Create an array of length equal to the number of vertices.

Step2) While the number of vertices is greater than 0, repeat:

i) Find a vertex with no incoming edges ("no prerequisites").
ii) Put this vertex in the array.
iii) Delete the vertex from the graph.

Topological Sort for Single-Source Shortest Paths in Directed Acyclic Graphs (DAG)

A topological sorting of a directed acyclic graph is a linear ordering of its nodes where each node comes before all nodes to which it has edges.

By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, one can compute shortest paths from a single source in Θ(V + E) time. Shortest paths are always well defined in DAG, since even if there are negative-weight edges, no negative-weight cycles can exist. Negative edges are admitted, because the graph has no cycle. If there exists a path from u to v then u precedes v in the topological sort.

For example, consider **figure 3.32** which shows an execution of the shortest-path-DAG algorithm. The vertex of G is topologically ordered from left to right. Nodes are blackened when they are visited by the algorithm and the predecessor of a node is connected to it by a dashed edge.

Figure 3.32 (a) shows the result of the initialization of lines 2-6. **Figures 3.32 (b)-(g)** show the result of the execution of the loop in lines 8-11. The final result is given in **figure 3.32(g)**.

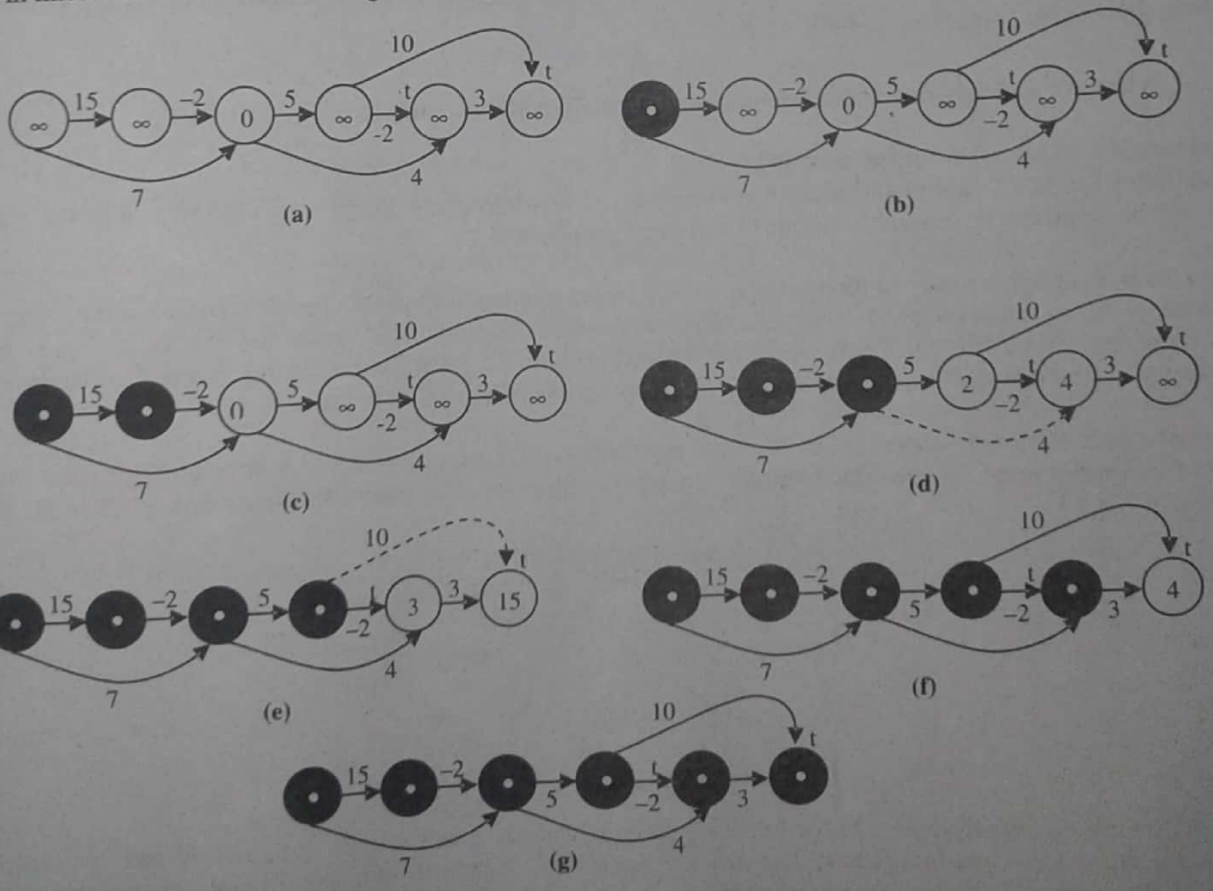


Figure 3.32 Execution of Single-Source Shortest Path in a DAG Algorithm

The graph is represented by adjacency lists. Other data structures have been employed. In particular:
1) color[u], storing the color of the vertex u.
2) pred[u], containing the parent of the vertex u.
3) d[u], storing the shortest-path estimate for u.

The algorithm starts by topologically sorting the DAG to impose a linear ordering on the vertices.

Algorithm: SHORTEST-PATH _DAG (G, w, s)
TOPOLOGICAL- SORT (G)
Step 1) for each vertex u in V[G]
Step 2) d[v] = infinity
Step 3) pred[u] = NULL
Step 4) color[v]= white
Step 5) d[s] = 0
Step 6) for each vertex u, selected by following the topological order do
Step 7) for each vertex v adjacent to u do
Step 8) if d[v] > d[u] + w(u,v)
Step 9) d[v] = d[u] + w(u,v)
Step 10) pred[v] = u

Analysis Single-Source Shortest Paths in Directed Acyclic Graphs
TOPOLOGICAL- SORT (G) calls the topological sort procedure on the graph G. **Lines 1-4** initialize the data structures: for each vertex, except for the source, the color is set to white, d[u] is set to infinity and parent to NULL. **Line 5** sets d[s] to 0 from the source. **Lines 7-11**: For each vertex selected by following the topological ordering, the algorithm relaxes the edges that are in its adjacency lists.

The topological sort of G takes $\Theta(|V|+|E|)$. In fact, **lines 1-4** require $\theta(V)$. For each vertex, the algorithm considers the edges in its adjacency lists. Thus, **lines 8-10** are executed $O(|E|)$ times and take O(1) time. Thus, the time complexity of the shortest-path_DAG algorithm is $\Theta(V+E)$.

Ques 29) What are strongly connected components? Explain with an example.

Ans: Strongly Connected Components (SCC)
Connectivity in an undirected graph means that every vertex can reach every other vertex via any path. If the graph is not connected the graph can be broken down into Connected Components.

Strong connectivity applies only to directed graphs. A directed graph is strongly connected if there is a directed path from any vertex to every other vertex. This is same as connectivity in an undirected graph, the only difference being strong connectivity applies to directed graphs and there should be directed paths instead of just paths. Similar to connected components, a directed graph can be broken down into Strongly Connected Components.

A directed graph is strongly connected if there is a path between all pairs of vertices. A Strongly Connected Component (SCC) of a directed graph is a maximal strongly connected sub-graph. **For example,** there are 3 SCCs in the following **figure 3.33**:

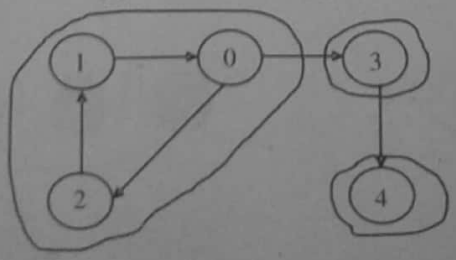


Figure 3.33

A directed graph is strongly connected it for every pain of vertices u and v there is a directed path between any two vertices, i.e., from u to v and from v to u. The strongly connected components of a directed graph are its maximal strongly connected sub-graphs. There form a partition of the graph.

Example: In the following **figure**, there is a directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labelled with its discovery and finishing times.

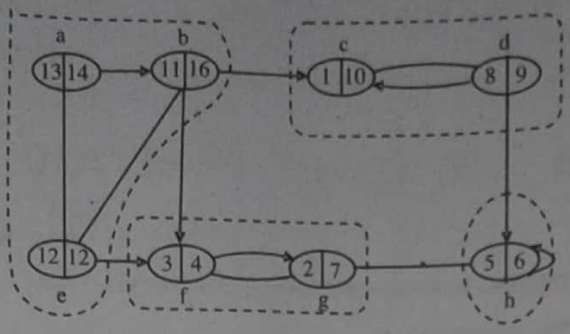


Figure 3.34

The following algorithm computes the strongly connected components of a directed graph.

Strongly-Connected Components (G)

1) Call DFS (G) to calculate the finishing time f[u] for each vertex u.
2) Compute G^T.
3) Call DFS (G^T), but in the main loop of DFS, consider the vertices in order of decreasing f[u] (as computed in line 1).
4) Output the vertices at each tree in the depth-first forest formed in line 3 as a separate strongly connected component.

Analysis: A linear-time algorithm, if the graph is represented as an adjacency list θ (V + E), as a matrix it is an 0 $(V)^2$) algorithm, computes the strongly connected components of a directed graph G = (V, E) using two Depth-First Search (DFS), one on G and one on G^T, the transpose grap. Breadth First Search (SFS) can be used instead of DFS.

Ques 30) Give a linear time algorithm alongwith proof of correctness to find strongly connected components of a graph.

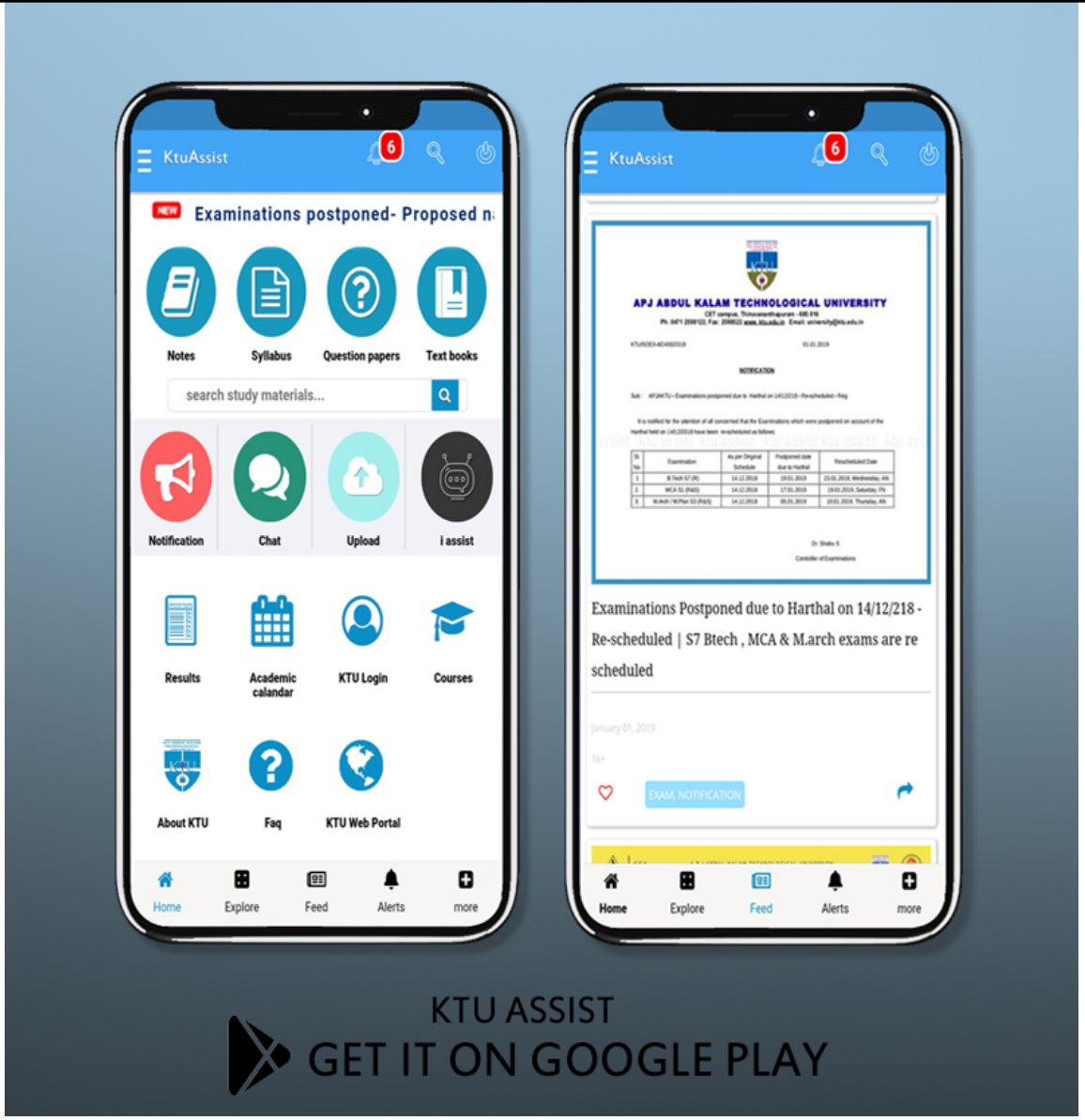
Ans: The following linear time θ (V + E) algorithm computes the strongly connected components of a directed graph G = (V, E) using two-depth first searches, one on G and on G^T.

Strongly-Connected-Components (G)

1) Call DFS (G) to compute finishing time f [u] for each vertex u.
2) Compute G^T.
3) Call DFS (GT), but in the main loop of DFS, consider vertices in order of decreasing f(u).
4) Output the vertices of each tree in the depth first forest formed in line 3 as a separate strongly connected component.

Ques 31) How can the number of strongly connected components of a graph change if a new edge is added?

Ans: If a new edge is added then the number of strongly connected components can be reduced because the new added edge makes two strongly connected components combined into one. Otherwise, the number will be same.



END

facebook.com/ktuassist

instagram.com/ktu_assist