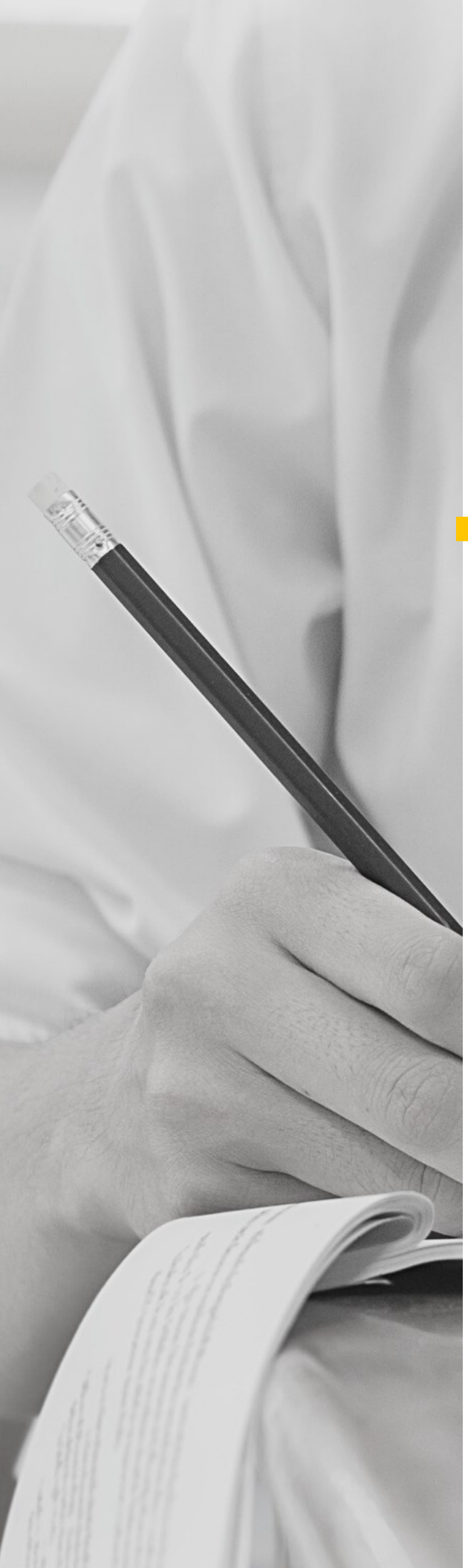


Prof. Trupesh
Patel's Notes

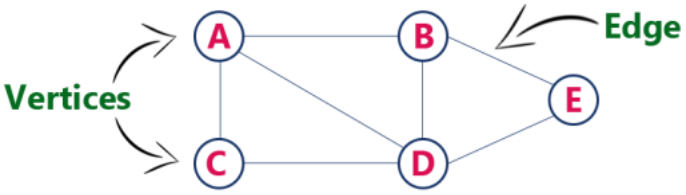
GRAPHS

Data Structures & Algorithms

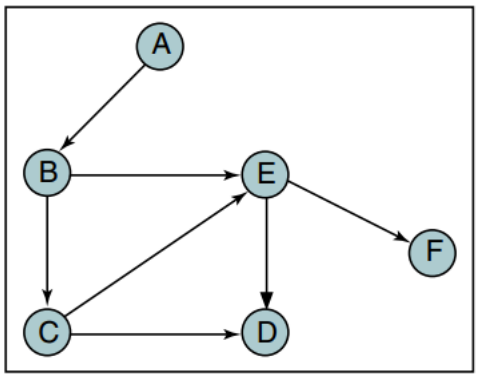
- Graph - Terminology
- Matrix representation
  - Adjacency matrix
  - Adjacency List
- Elementary graph operations
  - Breadth First Search
  - Depth First Search
- Spanning Trees & MST
- Shortest Paths

♣ Graph: Terminology

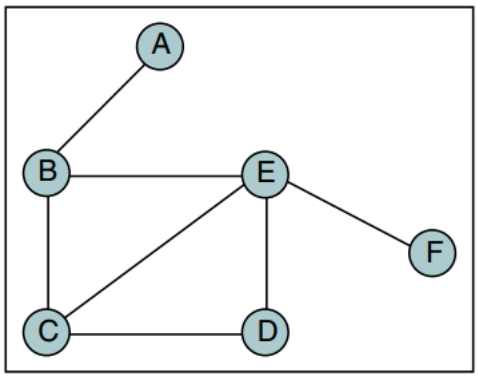
- A **graph** is a collection of nodes, called vertices, and line segments, called arcs or edges, that connect pairs of nodes.
- A **graph** is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.
  Example: graph G can be defined as G = (V , E)
  Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.
  This is a graph with 5 vertices and 6 edges



- A **directed graph**, or **digraph** for short, is a graph in which each line has a direction (arrow head) to its successor. The lines in a directed graph are known as arcs.
- An **undirected graph** is a graph in which there is no direction (arrow head) on any of the lines, which are known as edges.
- A **path** is a sequence of vertices in which each vertex is adjacent to the next one.

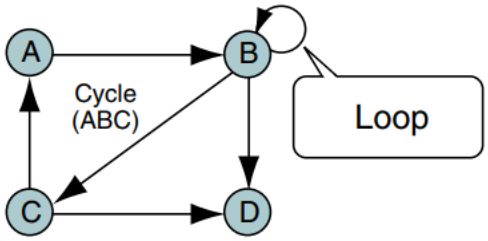


(a) Directed graph



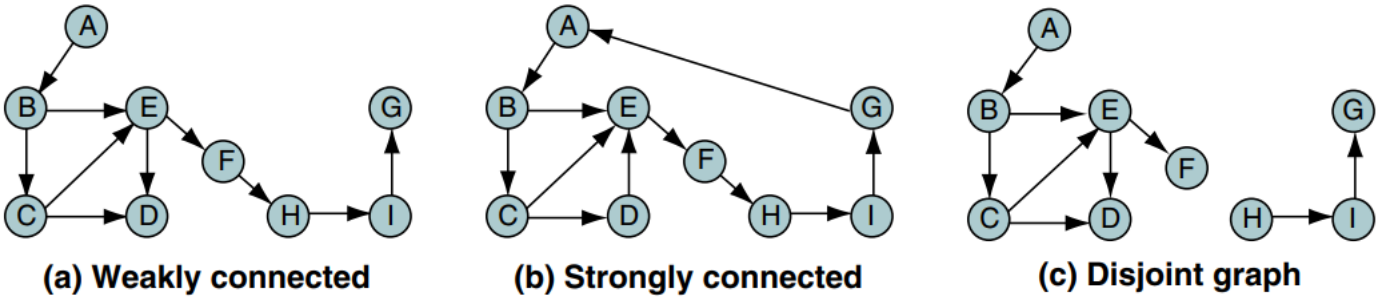
(b) Undirected graph

- Two vertices in a graph are said to be **adjacent vertices** (or neighbors) if there is a path of length 1 connecting them.
- A **cycle** is a path consisting of at least three vertices that starts and ends with the same vertex.
- A **loop** is a special case of a cycle in which a single arc begins and ends with the same vertex. In a loop the end points of the line are the same.



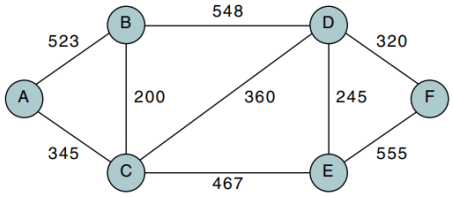
- Two **vertices** are said to be **connected** if there is a path between them.
- A **graph** is said to be **connected** if, ignoring direction, there is a path from any vertex to any other vertex.

- A **directed** graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph.
- A directed graph is **weakly connected** if at least two vertices are not connected.
- A graph is a **disjoint** graph if it is not connected.



(a) Weakly connected (b) Strongly connected (c) Disjoint graph

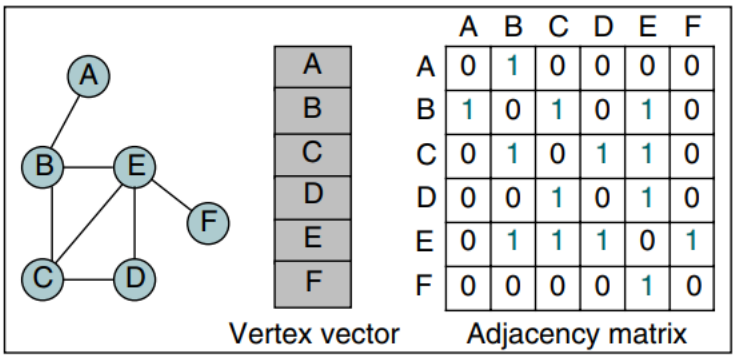
- The **degree of a vertex** is the number of lines incident to it.
- The **outdegree of a vertex** in a digraph is the number of arcs leaving the vertex.
- The **indegree** is the number of arcs entering the vertex.
- A network is a graph whose lines are weighted. It is also known as a **weighted graph**.



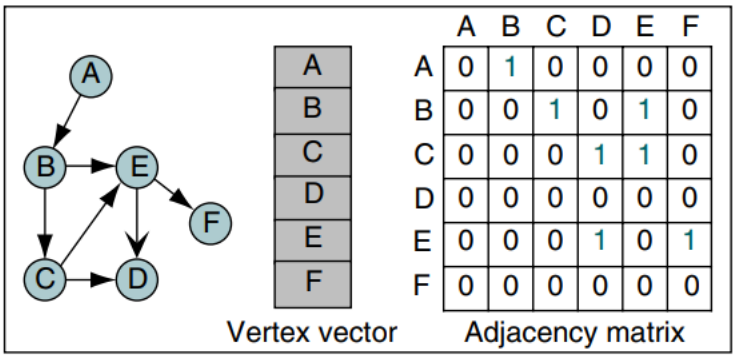
♣ Matrix Representation of Graphs

❖ Adjacency Matrix:

- To represent a graph, we need to store two sets. The first set represents the vertices of the graph, and the second set represents the edges or arcs. The two most common structures used to store these sets are **arrays** and **linked lists**.
- The adjacency matrix uses a vector (one-dimensional array) for the vertices and a matrix (two-dimensional array) to store the edges.
- If two vertices are adjacent—that is, if there is an edge between them—the matrix intersect has a value of 1; if there is no edge between them, the intersect is set to 0.
- If the graph is **directed**, the intersection in the adjacency matrix indicates the direction. For example, in Figure, there is an arc from source vertex B to destination vertex C. In the adjacency matrix, this arc is seen as a 1 in the intersection from B (on the left) to C (on the top). Because there is no arc from C to B, however, the intersection from C to B is 0. On the other hand, in Figure, the edge from B to C is bidirectional; that is, you can traverse it in either direction because the graph is nondirected. Thus, the nondirected adjacency matrix uses a 1 in the intersection from B to C as well as in the intersection from C to B. In other words, the matrix reflects the fact that you can use the edge to go either way.



(a) Adjacency matrix for nondirected graph

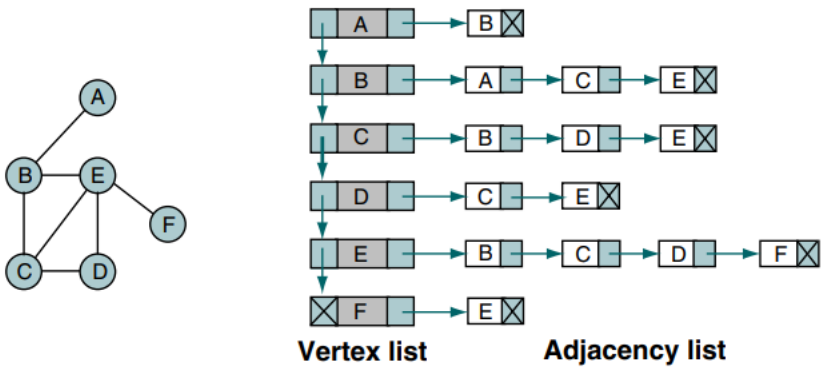


(b) Adjacency matrix for directed graph

❖ Limitations:

- ❖ The size of the graph must be known before the program starts.
- ❖ Only one edge can be stored between any two vertices.

❖ Adjacency List: The adjacency list uses a two-dimensional ragged array to store the edges.

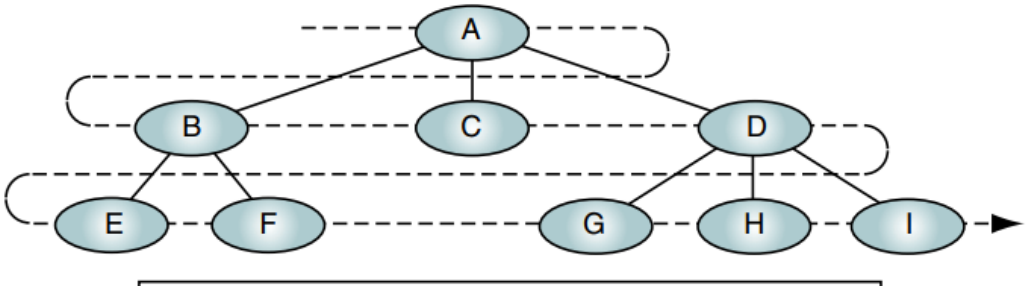


- ❖ The vertex list is a singly linked list of the vertices in the list.
- ❖ Depending on the application, it could also be implemented using **doubly linked lists** or **circularly linked lists**.
- ❖ The pointer at the left of the list links the vertex entries. The pointer at the right in the vertex is a head pointer to a linked list of edges from the vertex. Thus, in the nondirected graph on the left in Figure. there is a path from vertex B to vertices A, C, and E. To find these edges in the adjacency list, we start at B's vertex list entry and traverse the linked list to A, then to C, and finally to E.

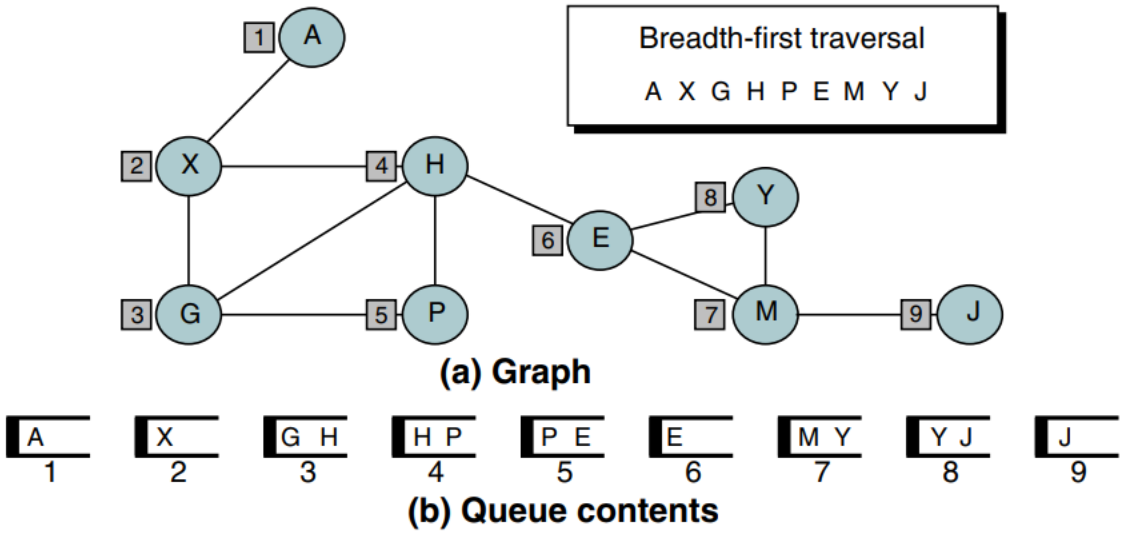
♣ Elementary Graph operations

❖ Breadth First Search:

- ❖ In the breadth-first traversal of a graph, we process all adjacent vertices of a vertex before going to the next level.
- ❖ we see that its breadth-first traversal starts at level 0 and then processes all the vertices in level 1 before going on to process the vertices in level 2.



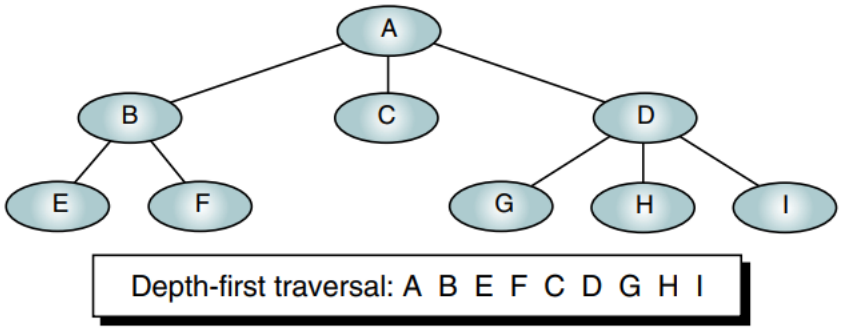
- ❖ We begin by picking a starting vertex (A); after processing it we process all of its adjacent vertices (BCD). After we process all of the first vertex's adjacent vertices, we pick its first adjacent vertex (B) and process all of its vertices, then the second adjacent vertex (C) and all of its vertices, and so forth until we are finished.



- ❖ We begin by enqueuing vertex A in the queue.
- ❖ We then loop, dequeuing the queue and processing the vertex from the front of the queue. After processing the vertex, we place all of its adjacent vertices into the queue. Thus, at step 2 in Figure 11-12(b), we dequeue vertex X, process it, and then place vertices G and H in the queue. We are then ready for step 3, in which we process vertex G.
- ❖ When the queue is empty, the traversal is complete.

❖ Depth First Search:

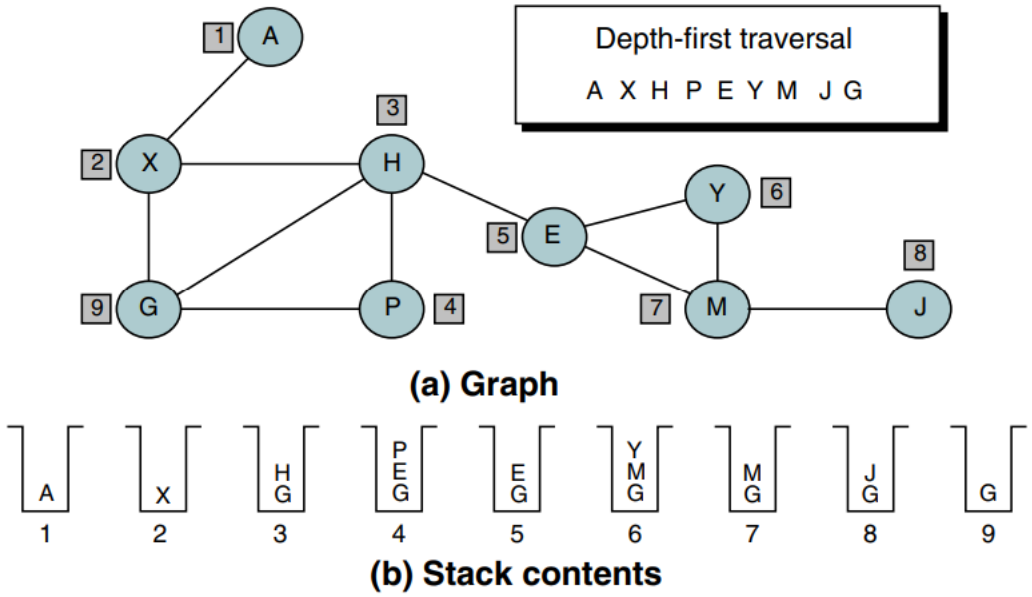
In the depth-first traversal, we process all of a vertex's descendants before we move to an adjacent vertex.



- ❖ The depth-first traversal of a graph starts by processing the first vertex of the graph.
- ❖ After processing the first vertex, we select any vertex adjacent to the first vertex and process it.
- ❖ As we process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries. This is similar to reaching a leaf in a tree.

- ❖ Let's trace a depth-first traversal through the graph in Figure.
- ❖ The number in the box next to a vertex indicates the processing order.
- ❖ The stacks below the graph show the stack contents as we work our way down the graph and then as we back out.

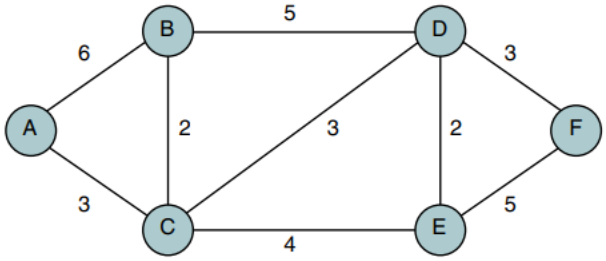
1. We begin by pushing the first vertex, A, into the stack.
2. We then loop, pop the stack, and, after processing the vertex, push all of the adjacent vertices into the stack. To process X at step 2, therefore, we pop X from the stack, process it, and then push G and H into the stack, giving the stack contents for step 3 as shown in Figure —H G.
3. When the stack is empty, the traversal is complete.



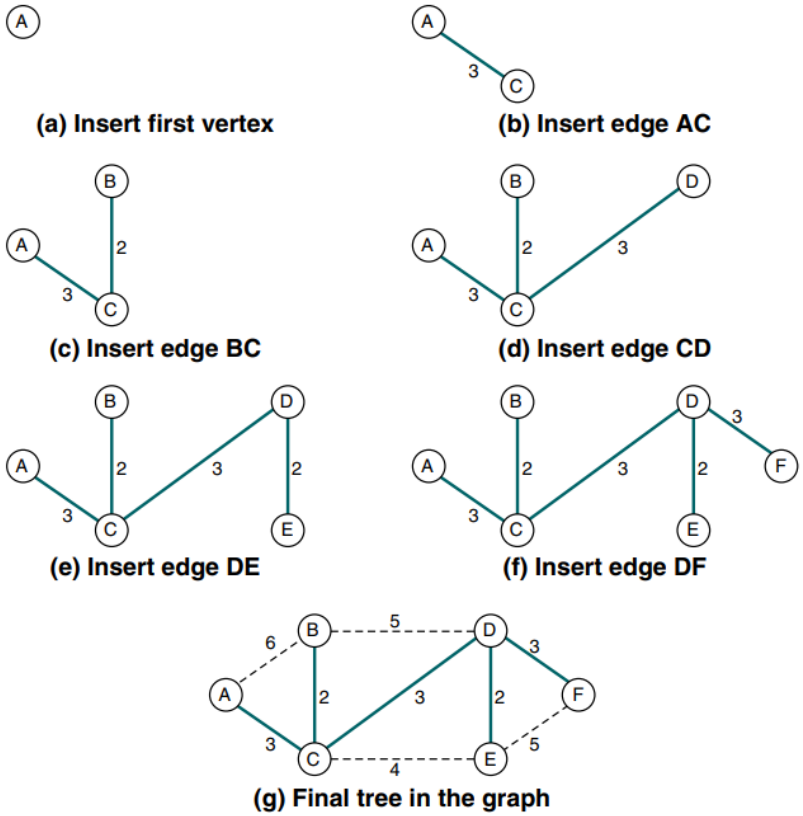
❖ Spanning Trees & Minimum Spanning Trees

- ❖ A spanning tree contains all of the vertices in a graph. A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.
- ❖ A minimum spanning tree is a spanning tree in which the total weight of the lines is guaranteed to be the minimum of all possible trees in the graph.
- ❖ Algorithm derives the minimum spanning tree of a network such that the sum of its weights is guaranteed to be minimal. If the weights in the network are unique, there is only one minimum spanning tree. If there are duplicate weights, there may be one or more minimum spanning trees.
- ❖ To create a minimum spanning tree in a strongly connected network— that is, in a network in which there is a path between any two vertices—the edges for the minimum spanning tree are chosen so that the following properties exist:

  1. Every vertex is included.
  2. The total edge weight of the spanning tree is the minimum possible that includes a path between any two vertices.



Spanning Tree



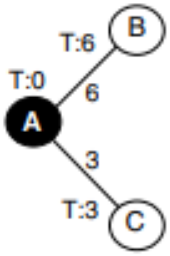
Minimum Spanning Tree

- We can start with any vertex. Because the vertex list is usually key sequenced, let's start with A.
- Then we add the vertex that gives the minimum weighted edge with A, AC.
- From the two vertices in the tree, A and C, we now locate the edge with the minimum weight. The edge AB has a weight of 6, the edge BC has a weight of 2, the edge CD has a weight of 3, and the edge CE has a weight of 4. The minimum-weighted edge is therefore BC.
- Note that in this analysis we do not consider any edge to a vertex that is already in the list. Thus, we did not consider the edge AC.
- To generalize the process, we use the following rule: from all of the vertices currently in the tree, select the edge with the minimal value to a vertex not currently in the tree and insert it into the tree.
- Using this rule, we add CD (weight 3), DE (weight 2), and DF (weight 3) in turn.

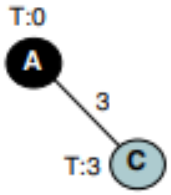
Shortest Paths

- The Dijkstra algorithm is used to find the shortest path between any two nodes in a graph.
- Generalized steps for shortest paths:

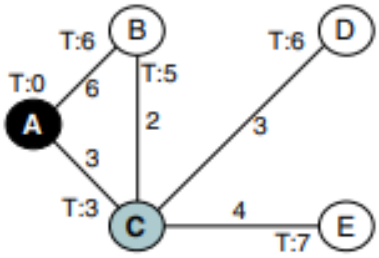
  1. Insert the first vertex into the tree.
  2. From every vertex already in the tree, examine the total path length to all adjacent vertices not in the tree. Select the edge with the minimum total path weight and insert it into the tree.
  3. Repeat step 2 until all vertices are in the tree.



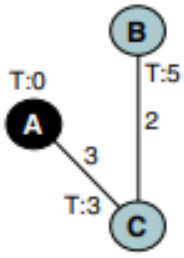
(a1) Possible paths from A1



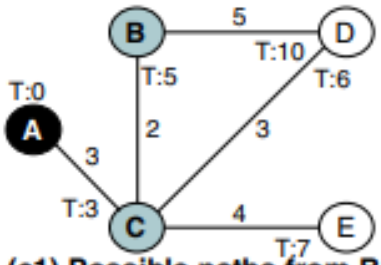
(a2) Tree after insertion of C



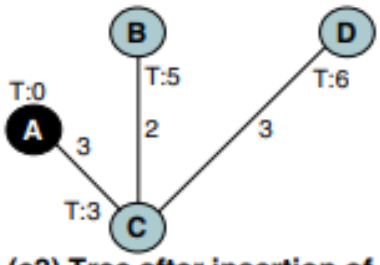
(b1) Possible paths from A and C



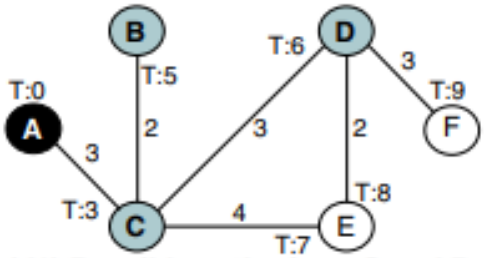
(b2) Tree after insertion of B



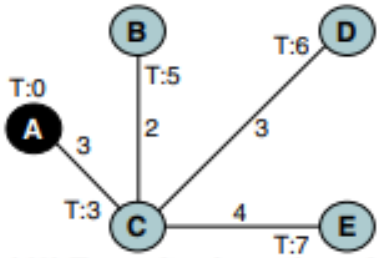
(c1) Possible paths from B and C



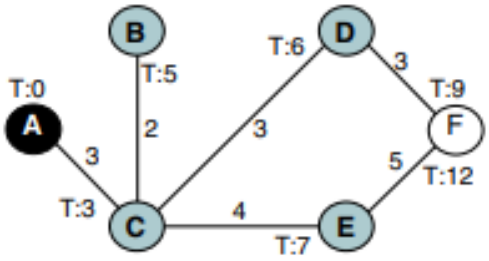
(c2) Tree after insertion of D



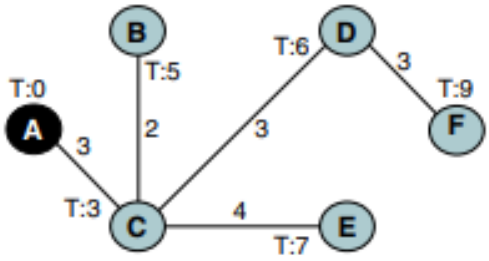
(d1) Possible paths from C and D



(d2) Tree after insertion of E



(e1) Possible paths from D and E



(e2) Tree after insertion of F

T:*n* Total path length from A to node

Determining Shortest Path