CHAPTER 10: GRAPHS

BIBHA STHAPIT
ASST. PROFESSOR
IoE, PULCHOWK CAMPUS

Introduction

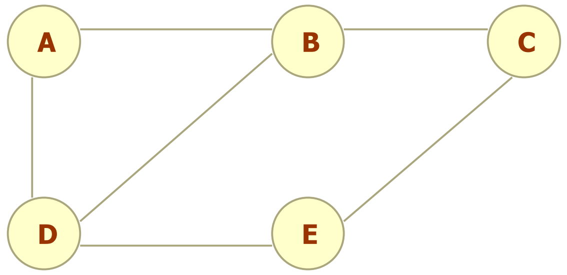
- A graph is an abstract data structure which is basically, a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes, any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

- <u>Why graphs are useful?</u>
- Graphs are widely used to model any situation where entities or things are related to each other in pairs; for example, the following information can be represented by graphs:
- *Family trees* in which the member nodes have an edge from parent to each of their children.
- *Transportation networks* in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Definition

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

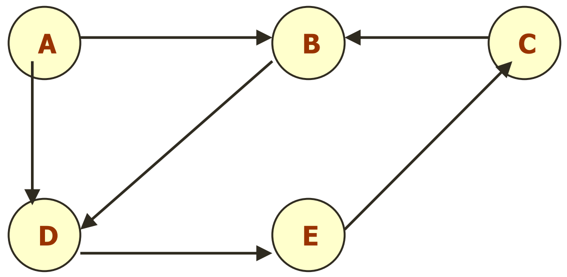
- The figure given shows a graph with V(G) = { A, B, C, D and E} and E(G) = { (A, B), (B, C), (A, D), (B, D), (D, E), (C, E) }. Note that there are 5 vertices or nodes and 6 edges in the graph.



Definition

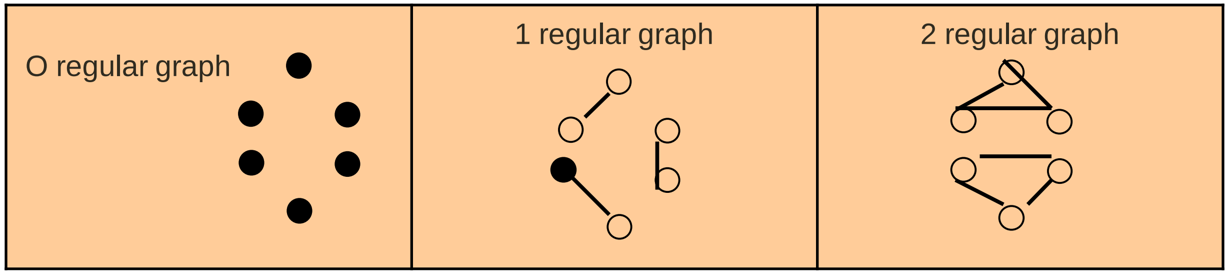
- A graph can be directed or undirected. In an undirected graph, the edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A. The above figure shows an undirected graph because it does not gives any information about the direction of the edges.

- The given figure shows a directed graph. In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).



Graph terminology

- ***Adjacent Nodes or Neighbors:*** For every edge, e = (u, v) that connects nodes u and v; the nodes u and v are the end-points and are said to be the adjacent nodes or neighbors.
- ***Degree of a node:*** Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.
- ***Regular graph:*** Regular graph is a graph where each vertex has the same number of neighbors. That is every node has the same degree. A regular graph with vertices of degree *k* is called a *k*-regular graph or regular graph of degree *k*.



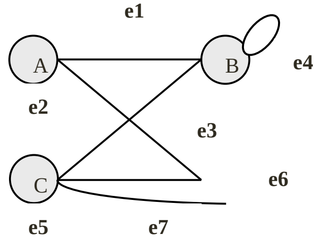
Graph terminology

- ***Path****:* A path P, written as P = {v0, v1, v2,….., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi-1 is adjacent to vi for i = 1, 2, 3, …, n.
- ***Closed path:*** A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn.
- ***Simple path:*** A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.
- ***Cycle****:* A closed simple path with length 3 or more is known as a cycle. A cycle of length *k* is called a *k* – cycle.
- ***Connected graph:*** A graph in which there exists a path between any two of its nodes is called a connected graph. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.
- ***Complete graph:*** A graph G is said to be a complete, if all its nodes are fully connected, that is, there is a path from one node to every other node in the graph. A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.

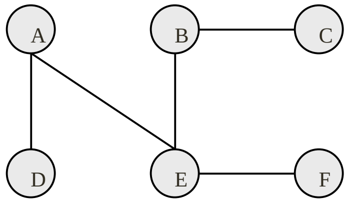
Graph terminology

- ***Labeled graph or weighted graph:*** A graph is said to be labeled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. Weight of the edge, denoted by w(e) is a positive value which indicates the cost of traversing the edge.
- ***Multiple edges:*** Distinct edges which connect the same end points are called multiple edges. That is, e = {u, v) and e' = (u, v) are known as multiple edges of G.
- ***Loop****:* An edge that has identical end-points is called a loop. That is, e = (u, u).
- ***Multi- graph****:* A graph with multiple edges and/or a loop is called a multi-graph.
- ***Size of the graph:*** The size of a graph is the total number of edges in it.

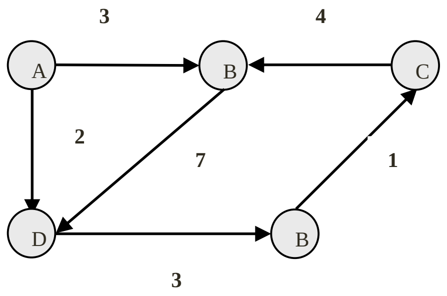
Graph terminology



Multigraph



Tree



Weighted graph

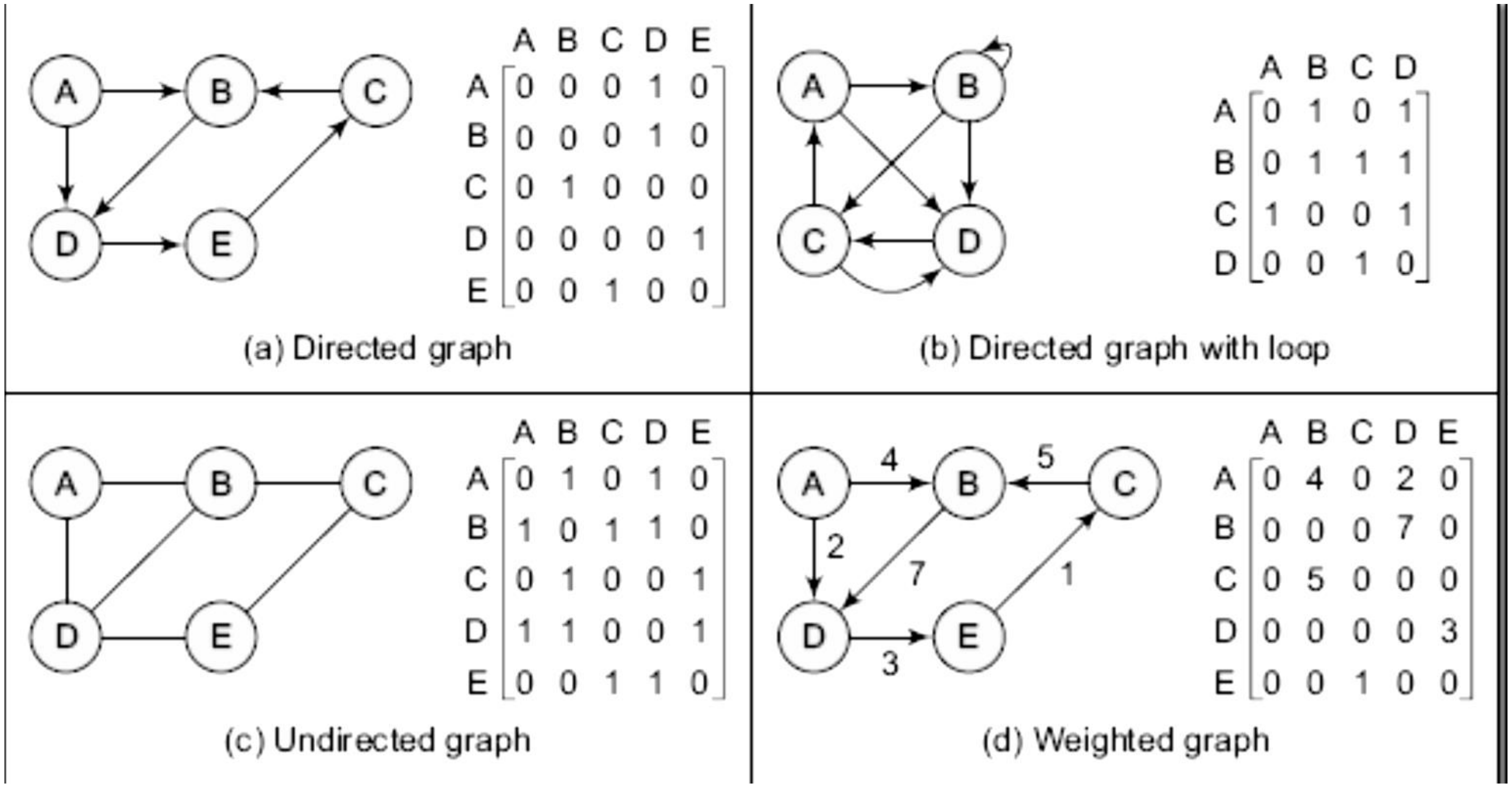
Adjacency matrix representation

- An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, we have learnt that, two nodes are said to be adjacent if there is an edge connecting them.
- In a directed graph G, if node v is adjacent to node u, then surely there is an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have dimensions of n X n.
- In an adjacency matrix, the rows and columns are labeled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero.

$$\begin{tikzcd}
 & \boxed{\textbf{1} \quad \textbf{if } v_i \textbf{ is adjacent to } v_j\textbf{, that is there is an edge } (v_i, v_j)} \\
\boxed{\text{aij}} \arrow[ur] \arrow[dr] & \\
 & \boxed{\textbf{0} \quad \textbf{otherwise}}
\end{tikzcd}$$

Adjacency matrix representation

- Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. therefore, a change in the order of nodes will result in a different adjacency matrix.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 0 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(a) Directed graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

(b) Directed graph with loop

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

(c) Undirected graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

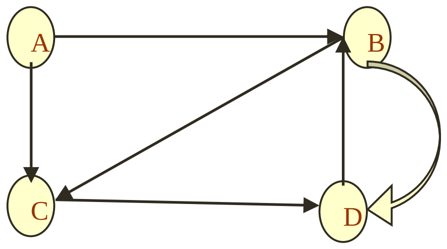
(d) Weighted graph

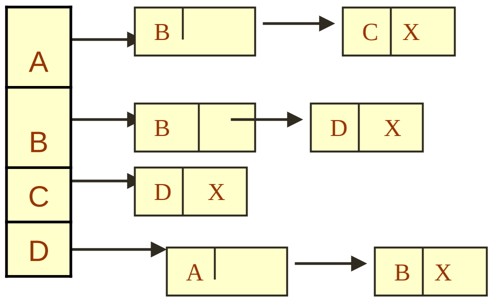
Adjacency list representation

- The adjacency list is another way in which graphs can be represented in computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to itself.
- The key advantage of using an adjacency list includes:
- It is easy to follow, and clearly shows the adjacent nodes of a particular node
- It is often used for storing graphs that have a small to moderate number of edges. That is an Adjacency list is preferred for representing sparse graphs in computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an Adjacency list. Adding new nodes in an Adjacency matrix is a difficult task as size of the matrix needs to be changed and existing nodes may have to be reordered.

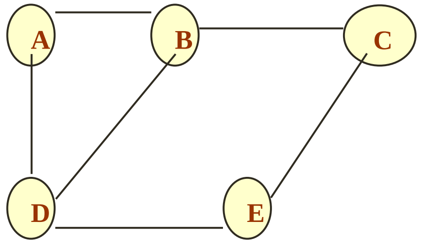
Adjacency list representation

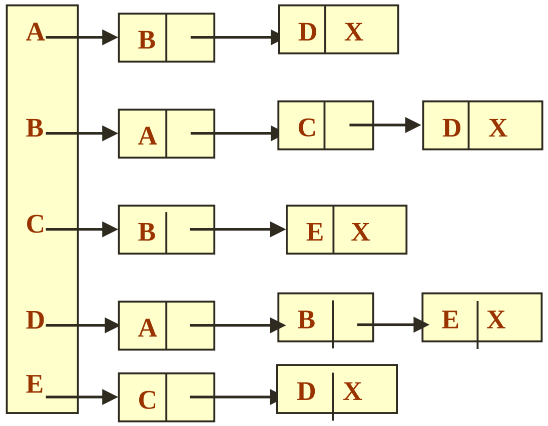
- For a directed graph, the sum of lengths of all adjacency lists is equal to the number of edges in G. However, for an undirected graph, the sum of lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge v to u. The adjacency list can also be modified to store weighted graphs.
- Graph G and its adjacency list

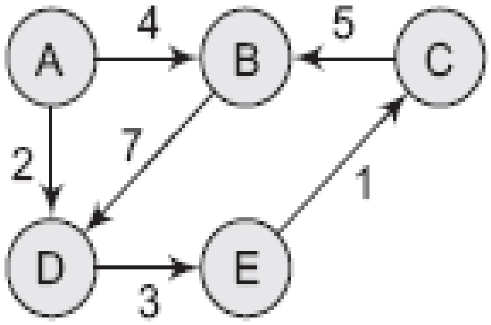




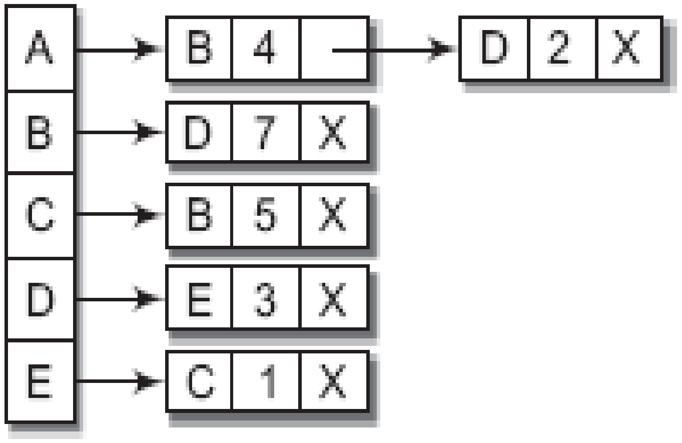
Adjacency list representation







(Weighted graph)



Graph traversal

- By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section. These two methods are-
- Breadth first search
- Depth first search
- While breadth first search will use a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme will use a stack. But both these algorithms will make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1, 2 or depending on its current state. Table I shows the value of status and its significance.

Graph traversal

STATUS	STATE OF THE NODE	DESCRIPTION
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Breadth first traversal/search

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.
- That is, we start examining the node A and then all the neighbors of A are examined. In the next step we examine the neighbors of neighbors of A, so on and so forth

Breadth first traversal/search

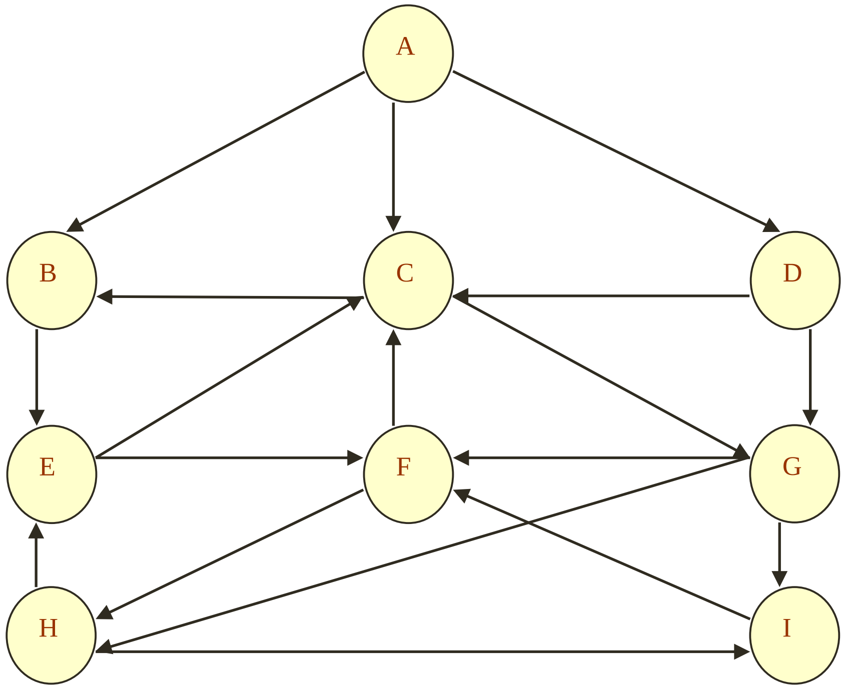
Algorithm for breadth-first search in a graph G beginning at a starting node A

- **Step 1: SET STATUS = 1 (ready state) for each node in G.**
- **Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)**
- **Step 3: Repeat Steps 4 and 5 until QUEUE is empty**
- **Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).**
- **Step 5: Enqueue all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)**

- **[END OF LOOP]**
- **Step 6: EXIT**

Breadth first search

- Example: Consider the graph G given below. The adjacency list of G is also given.



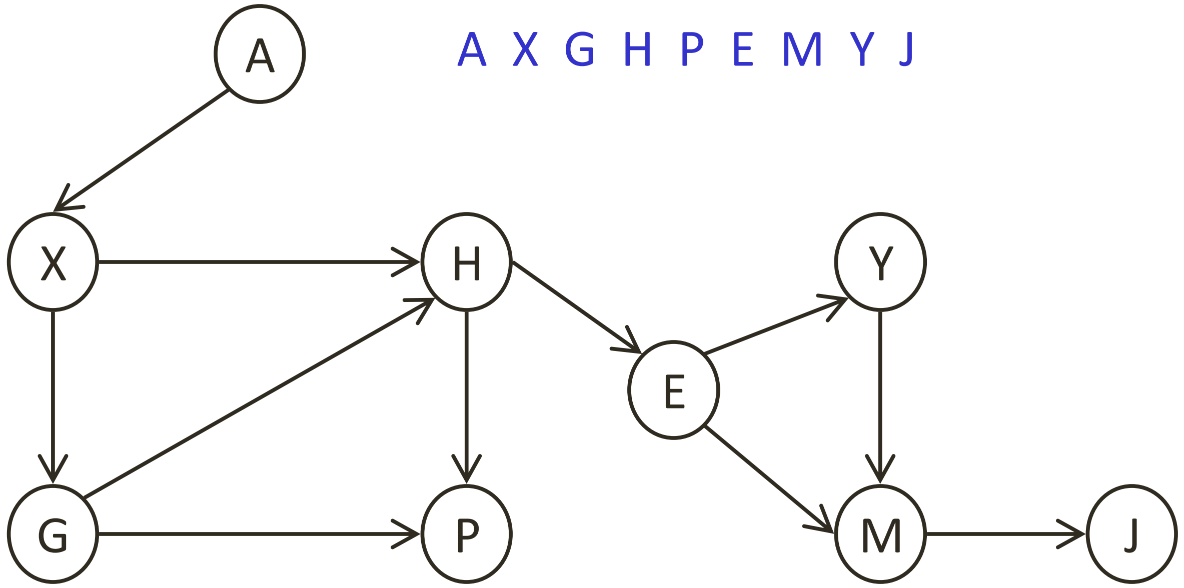
Adjacency Lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

a)Initially add A to QUEUE and add NULL to visited, so

Set of visited vertices	Queue
NULL	A
A	B, C, D
A, B	C, D, E
A, B, C	D, E, G
A, B, C, D	E, G
A, B, C, D, E	G, F
A, B, C, D, E, G	F, H
A, B, C, D, E, G, F	H
A, B, C, D, E, G, F, H	I
A, B, C, D, E, G, F, H, I	NULL

Breadth first traversal



A X G H P E M Y J

A	X	G H	H P	P E	E	M Y	Y J	J

queue

Depth first search/traversal

- The Depth First Search algorithm progresses by expanding the starting node of G and thus going deeper and deeper until a goal node is found, or until a node that has no children is encountered. When a dead- end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

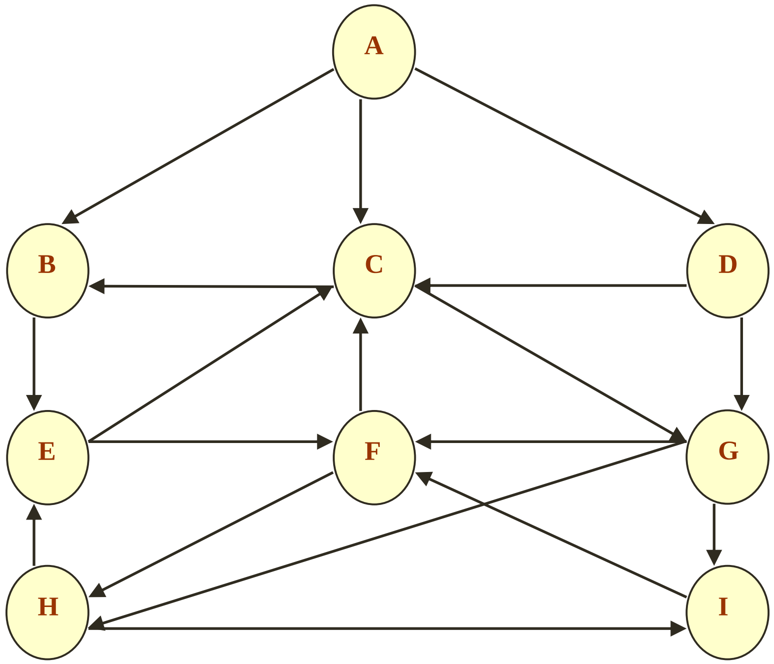
- In other words, the Depth- First Search algorithm begins at a starting node A which becomes the current node. Then it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the un-visited (un-processed node) becomes the current node.

Depth first search

```
Algorithm for depth-first search in a graph G beginning
  at a starting node A

Step 1: SET STATUS = 1 (ready state) for each node in G.
Step 2: Push the starting node A on the stack and set
      its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:Pop the top node N. Process it and set its
      STATUS = 3 (processed state).
Step 5:Push on to the stack all the neighbors of N that
  are in the ready state (whose STATUS = 1) and set their
  STATUS = 2 (waiting state)
            [END OF LOOP]
 Step 6: EXIT
```

Depth first search

- Example: Consider the graph G given below. The adjacency list of G is also given. Suppose we want to print all nodes that can be reached from the node A (including A itself).

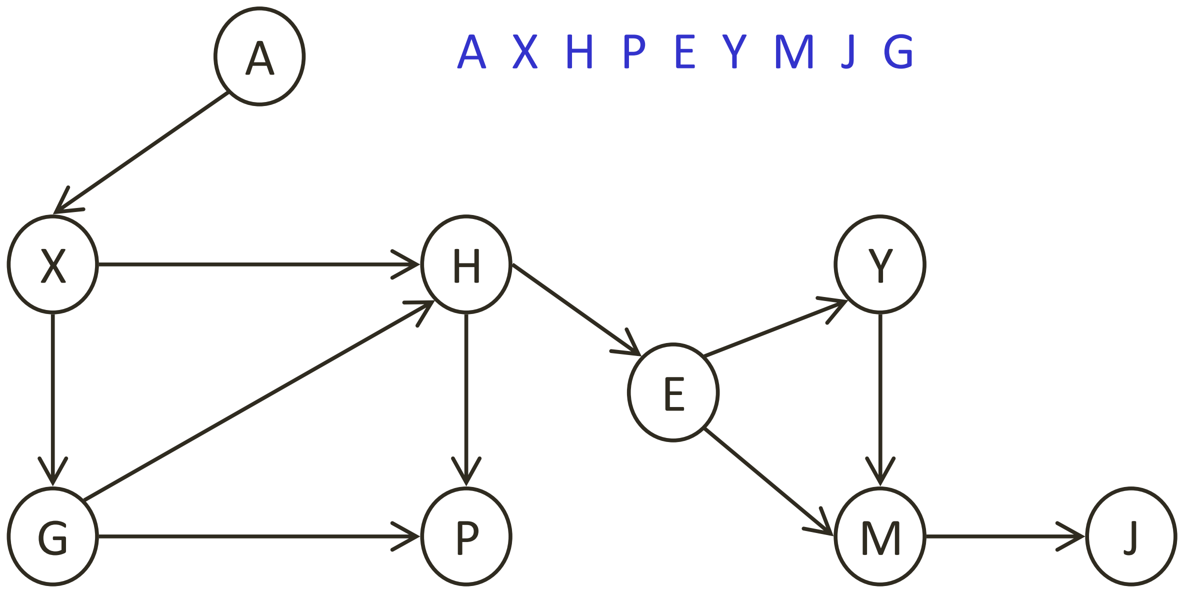


Adjacency Lists

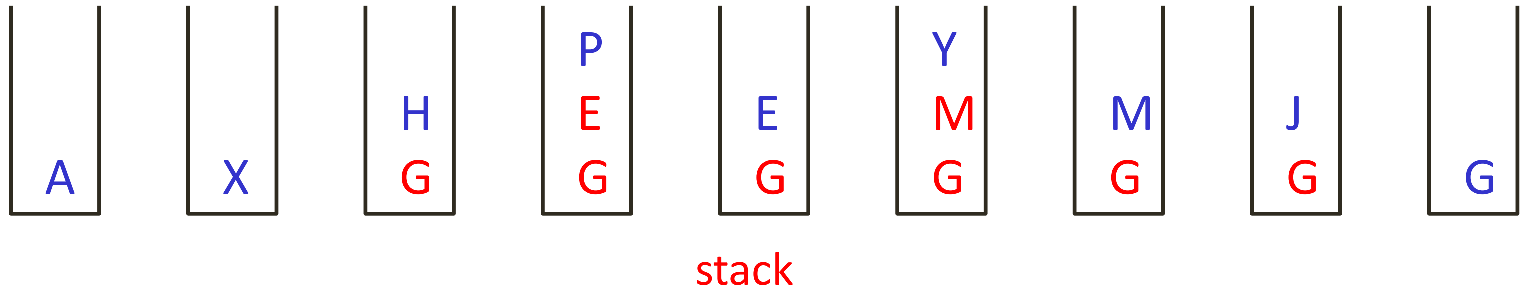
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Set of visited vertices	Stack
NULL	A
A	B, C, D
A, D	B, C, G
A, D, G	B, C, F, H, I
A, D, G , I	B, C, F, H
A, D, G , I, H	B, C, F, E
A, D, G , I, H, E	B, C, F
A, D, G , I, H, E, F	B, C
A, D, G , I, H, E, F, C	B
A, D, G , I, H, E, F, C, B	NULL

Depth first traversal



A X H P E Y M J G

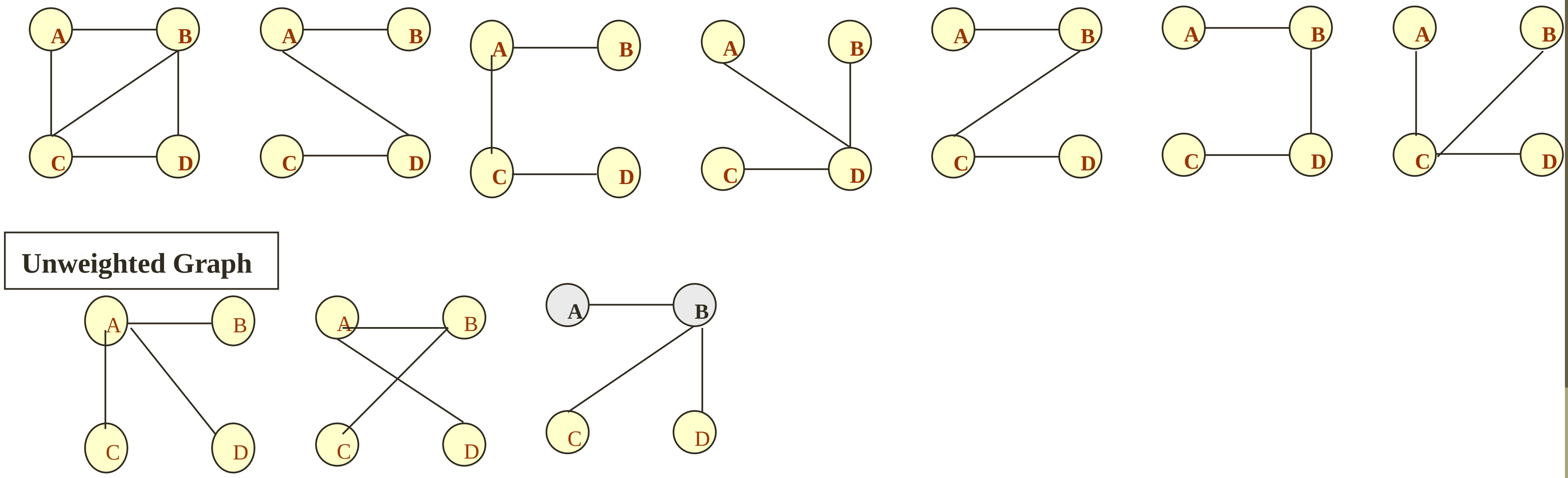


Minimum spanning tree (MST)

- A spanning tree of a connected, undirected graph G, is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees.
- When we assign *weights* to each edge (which is a number that represents how unfavorable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.
- A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a **minimum spanning tree** is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

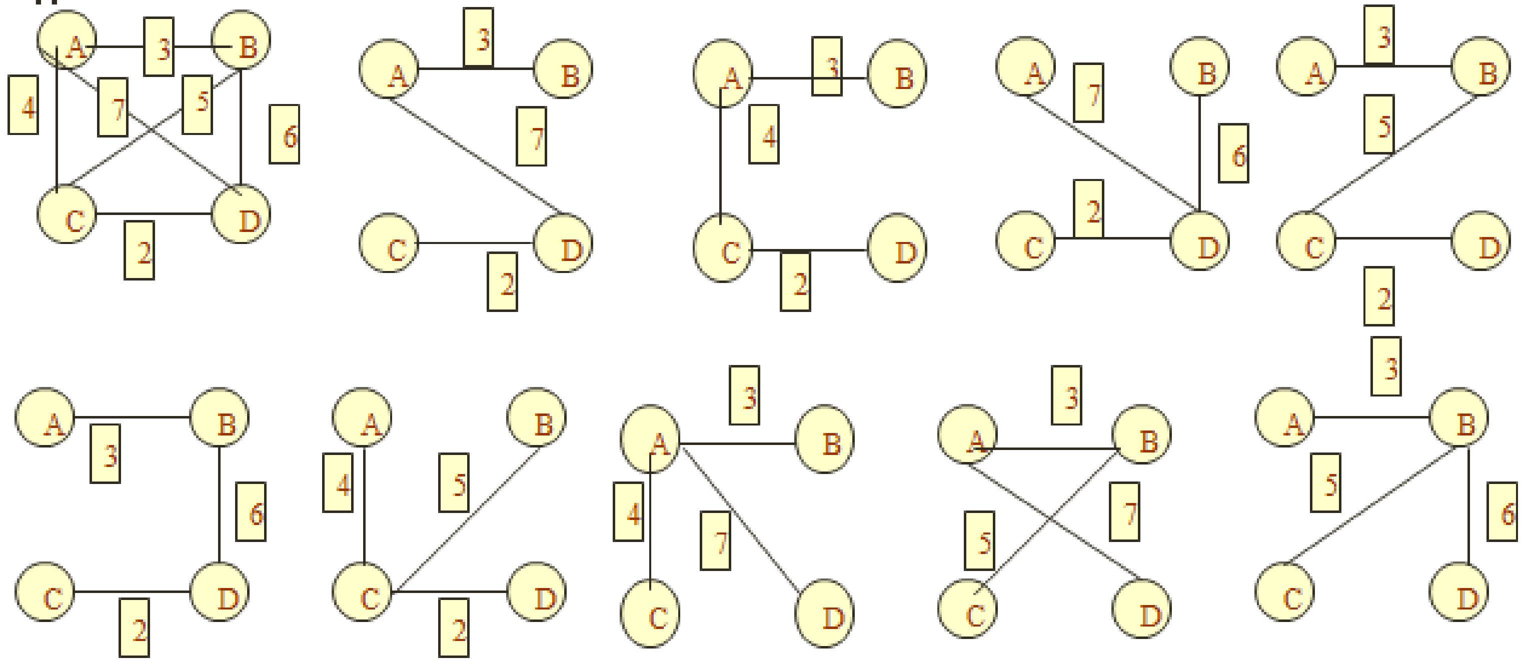
Minimum spanning tree

Example: Consider an un-weighted graph G given below. From G we can draw many distinct spanning trees. (Eight of them are given here). For an un-weighted graph, every spanning tree is a minimum spanning tree.



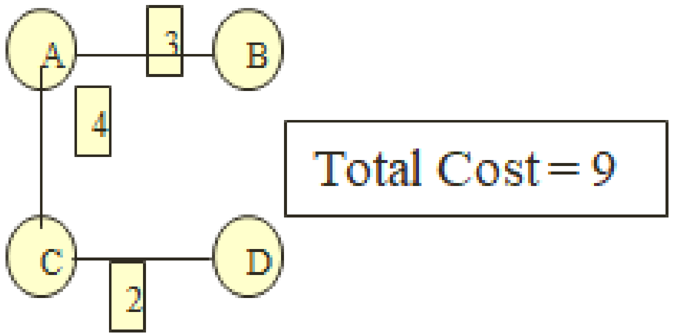
Minimum spanning tree

- Example: Consider a weighted graph G given below. From G we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has minimum weight (cost) associated with it



Minimum spanning tree

- Thus, we see that of all the given spanning trees, the tree given below is called the minimum spanning tree as it has the lowest cost associated with it.



Prim's algorithm for MST

Tree vertices: Vertices that are a part of the minimum spanning tree T.

Fringe vertices: Vertices that are currently not a part of T, but are adjacent to some tree vertex.

Unseen Vertices: Vertices that are neither tree vertices nor fringe vertices fall under this category.

The steps in the Prim algorithm can be given as follows:

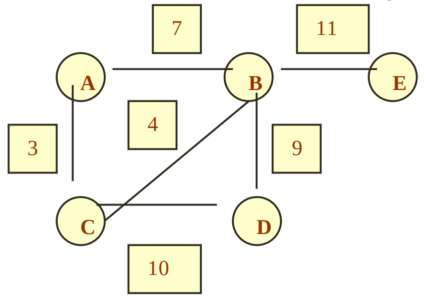
- Choose a starting vertex
- Branch out from the starting vertex and during each iteration select a new vertex and edge. Basically, during each iteration of the algorithm, we have to select a vertex from the fringe vertices in such a way that the edge connecting the tree vertex and the new vertex has minimum weight assigned to it.

Prim’s algorithm for MST

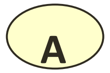
```
Prim’s Algorithm

Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are
           fringe vertices
Step 3:Select an edge e connecting the tree
           vertex and fringe vertex that has
           minimum weight
Step 4:Add the selected edge and the vertex to
           the minimum spanning    tree T
    [END OF LOOP]
  Step 5: EXIT
```

Prim's algorithm for MST

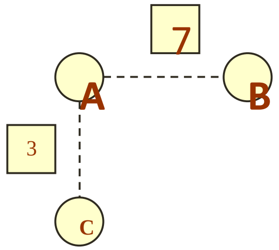
Example: Construct a minimum spanning tree of the graph given below.



Choose a starting vertex, A for example.

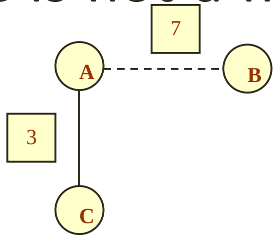


Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.

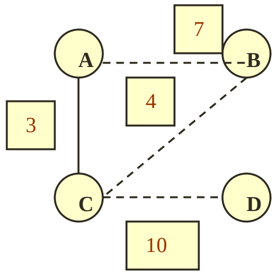


Prim's algorithm for MST

- Select an edge connecting the tree vertex and fringe vertex that has minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, so add C to the tree. Now C is not a fringe vertex but a tree vertex.

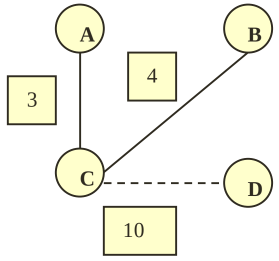


- Add the fringe vertices (that are adjacent to C).

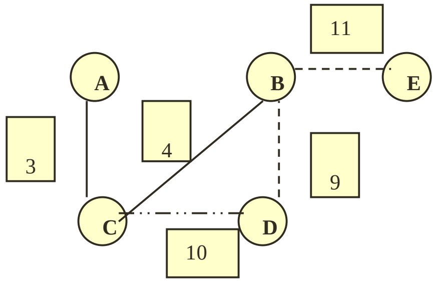


Prim’s algorithm for MST

- Select an edge connecting the tree vertex and fringe vertex that has minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, so add B to the tree. Now B is not a fringe vertex but a tree vertex.

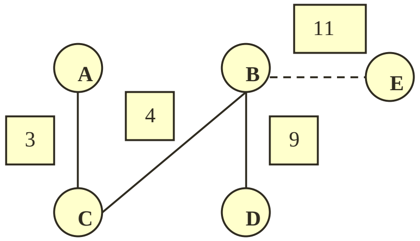


- Add the fringe vertices (that are adjacent to B).

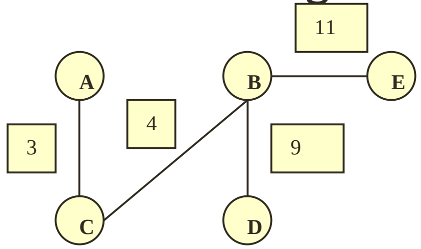


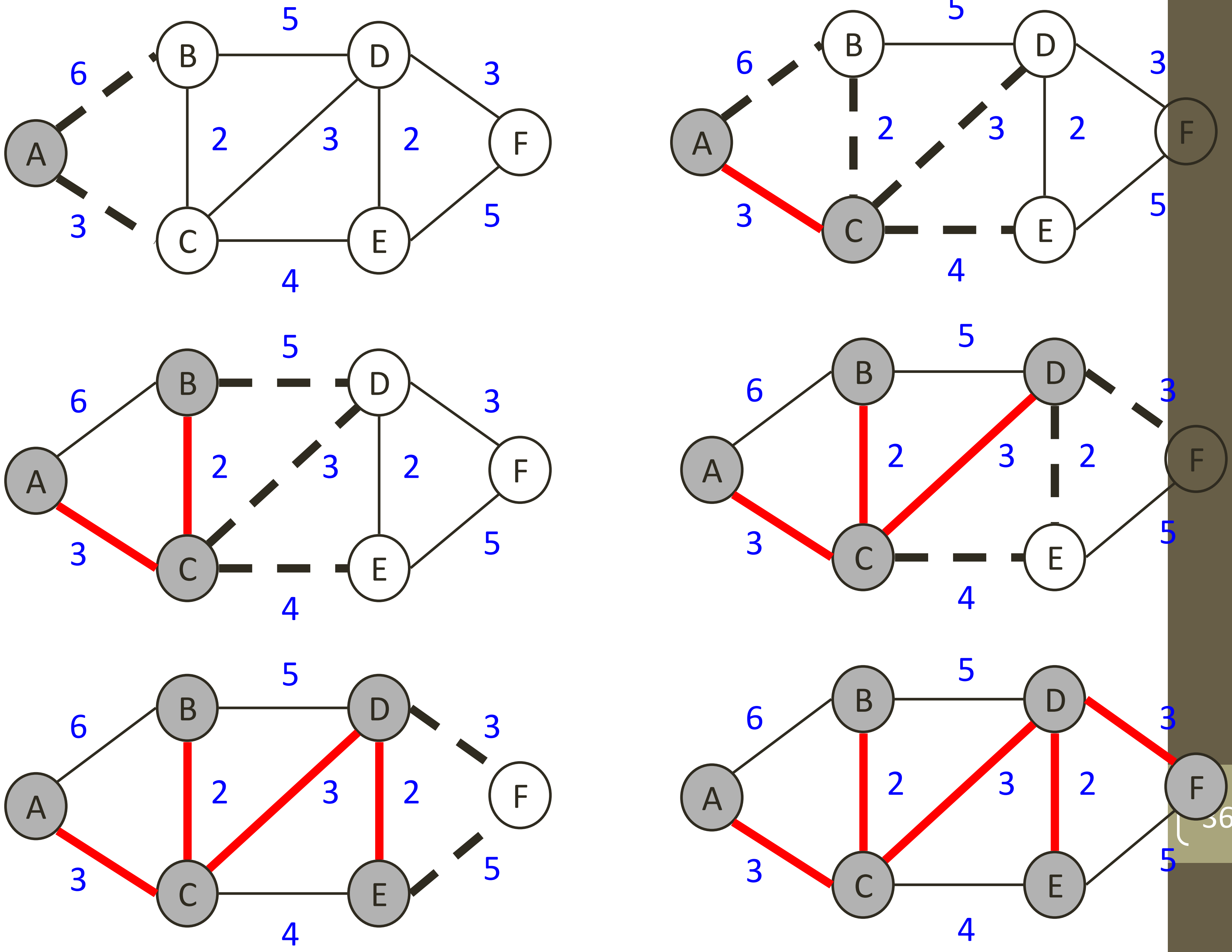
Prim's algorithm for MST

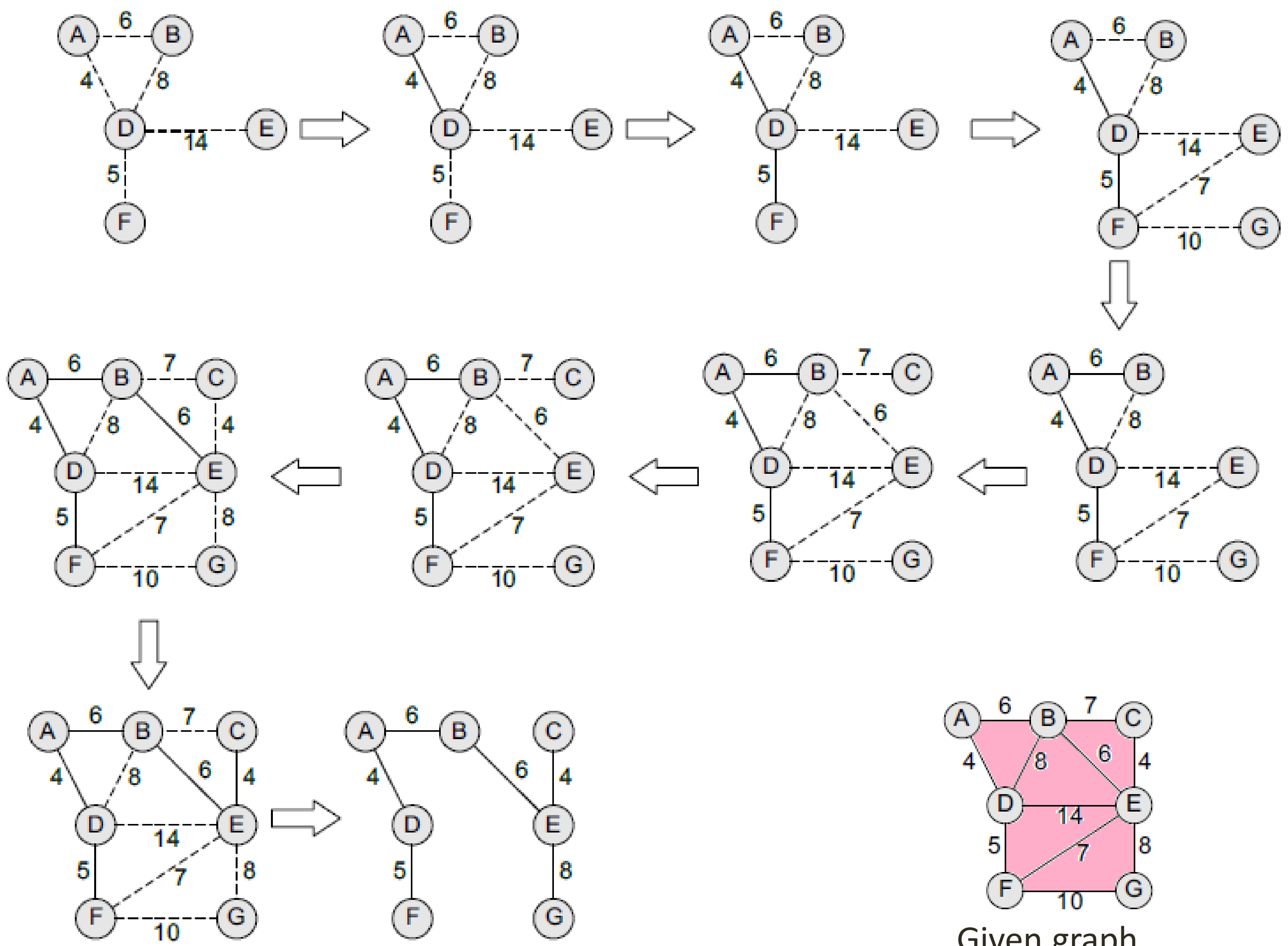
- Select an edge connecting the tree vertex and fringe vertex that has minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, so add D to the tree. Now D is not a fringe vertex but a tree vertex.

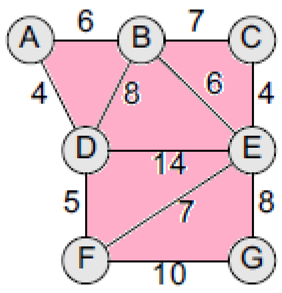


- Note, now node E is not connected, so we will add it in the tree because a minimum spanning tree is one in which all n nodes are connected with n-1 edges that has minimum weight. So the minimum spanning tree can now be given as,




A
B
C
D
E
F
6
5
3
2
3
2
3
4
5




Given graph

Kruskal's algorithm for MST

- Like the Prim's algorithm, the Kruskal's algorithm is used to find a minimum spanning tree for a connected weighted graph. That is, the algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if initially, the graph is not connected then it finds a *minimum spanning forest* (Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees).

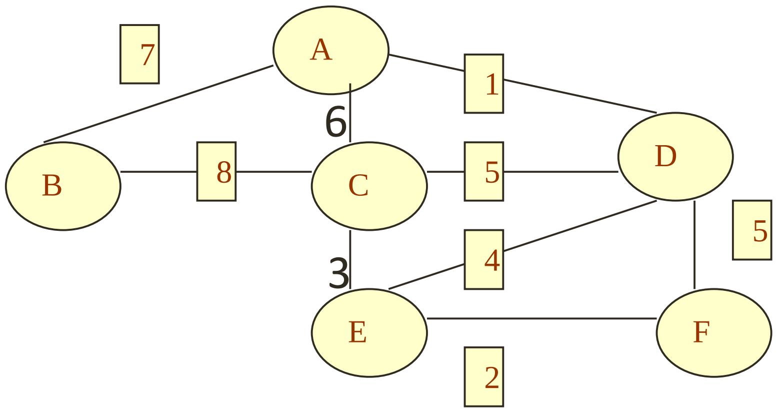
- Kruskal's algorithm is an example of a greedy algorithm as it makes the locally optimal choice at each stage with the hope of finding the global optimum. The algorithm can be given as shown in figure.

Kruskal's algorithm for MST

```
KRUSKAL'S ALGORTIHM

Step 1: Create a forest in such a way that each graph
            is a separate tree.
Step 2: Create a priority queue Q that contains all
            the edges of the graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4: Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two
      different trees, then ,
            Add it to the forest (for combining two
  trees into one tree).
       ELSE
            Discard the edge
Step 6: END
```

Kruskal's algorithm for MST

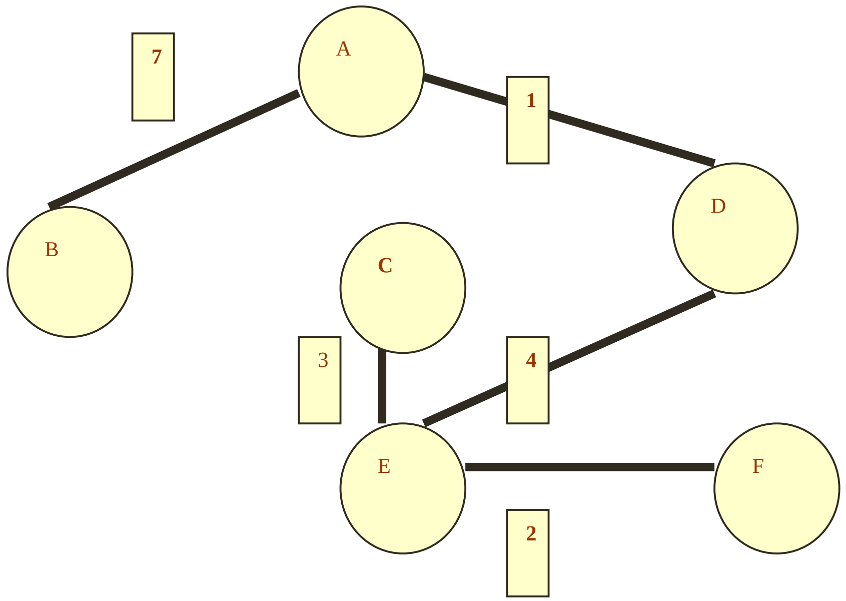
Example: Apply Kruskal's algorithm on the graph given below



Initially, we have F = {{A}, {B}, {C}. {D}, {E}, {F}}
MST = {}
Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

STEP	F	MST	Q
1	{{A, D}, {B}, {C}, {E}, {F}}	{A, D}	{(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}
2	{{A, D}, {B}, {C}, {E, F}}	{(A, D), (E, F)}	{(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}
3	{{A, D}, {B}, {C, E, F}}	{(A, D), (C, E), (E, F)}	{(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}
4	{{A, C, D, E, F}, {B}}	{(A, D), (C, E), (E, F), (E, D)}	{(C, D), (D, F), (A, C), (A, B), (B, C)}
5	{{A, C, D, E, F}, {B}}	{(A, D), (C, E), (E, F), (E, D)}	{(D, F), (A, C), (A, B), (B, C)}
6	{{A, C, D, E, F}, {B}}	{(A, D), (C, E), (E, F), (E, D)}	{(A, C), (A, B), (B, C)}
7	{{A, C, D, E, F}, {B}}	{(A, D), (C, E), (E, F), (E, D)}	{(A, B), (B, C)}
8	{A ,B, C, D, E, F}	{(A,D),(C,E),(E,F) , (E,D), (A, B)}	{(B, C)}
9	{A ,B, C, D, E, F}	{(A,D),(C,E),(E,F) , (E,D), (A, B)}	----

- The MST is obtained as,



Dijkstra’s algorithm

- Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree.
- Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node.
- Moreover, Dijkstra’s algorithm is also used for finding costs of shortest paths from a source node to a destination node.
- That means, it solves single source shortest path problem for a directed graph G(V,E) with non-negative edge weights i.e. w(u,v)>=0

Dijkstra's algorithm

Dijkastra's Algorithm

1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with 0, and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labeled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. a. If the node that is not in N has no labeled then SET the label of the node = the label of the newly inserted node + the length of the edge.

   b. Else if the node that is not in N was already labeled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)

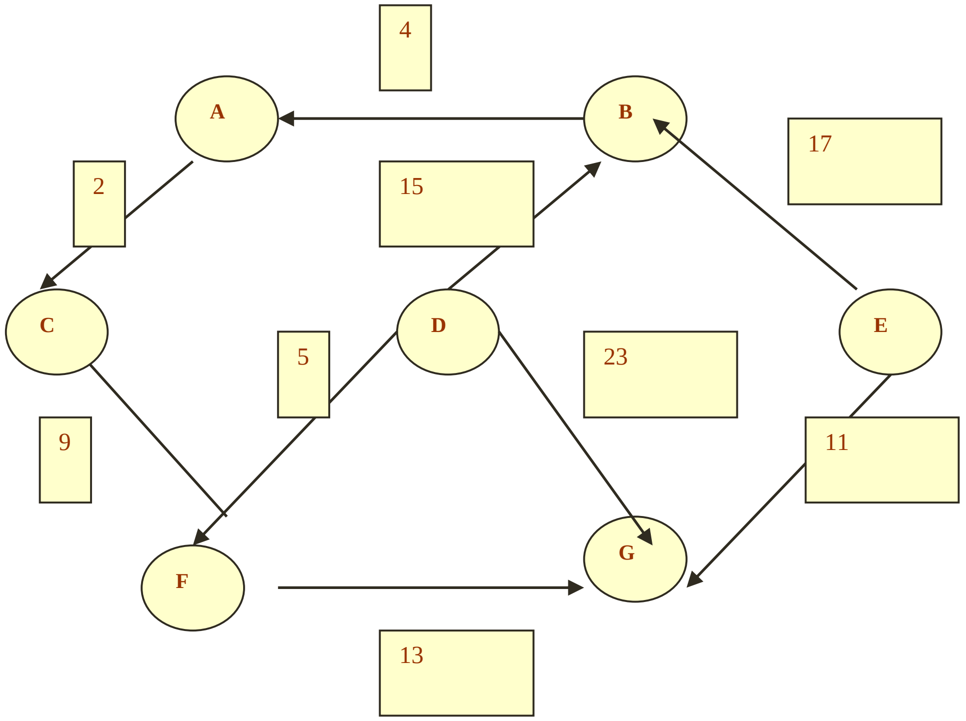
7. Pick a node not in N that has the smallest label assigned to it and add it to N

Dijkstra’s algorithm

- Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: temporary and permanent. While temporary labels are assigned to nodes that have not been reached, permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known.
- The execution of this algorithm will produce either of two results-
  - If the destination node is labeled, then the label will in turn represent the distance from the source node to the destination node.
  - If it the destination node is not labeled, then it means that there is no path from the source to the destination node.

Dijkstra's algorithm

- Example: Consider the graph G given below. Taking D as the initial node, execute the Dijkastra's algorithm on it.



Dijkstra's algorithm

- Step 1: SET the label of D = 0 and **N = {D**}.
- Step 2: Label of D = 0, B = 15, G = 23 and F = 5. Therefore, **N = {D, F}.**
- Step 3: Label of D = 0, B = 15, G has been re-labeled 18 because minimum (5 + 13, 23) = 18, C has been re-labeled 14 (5 + 9). Therefore, **N = {D, F, C}.**
- Step 4: Label of D = 0, B = 15, G = 18. Therefore, **N = {D, F, C, B}.**
- Step 5: Label of D = 0, G = 18 and A = 19 (15 + 4). Therefore, **N = {D, F, C, B, G}.**
- Step 6: Label of D = 0 and A = 19. Therefore,**N = {D, F, C, B, G, A}**
- Note that we have no label for node E, this means that E is not reachable from D. only the nodes that are in N are reachable from B.

Warshall's algorithm

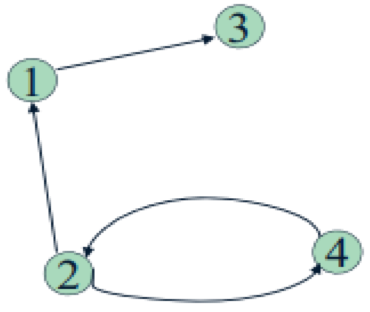
- Computes the transitive closure of a relation (Alternatively: all paths in a directed graph)
- Warshall's algorithm defines matrices P0, P1, P2, …. Pn as given as

$P_k[i][j]$

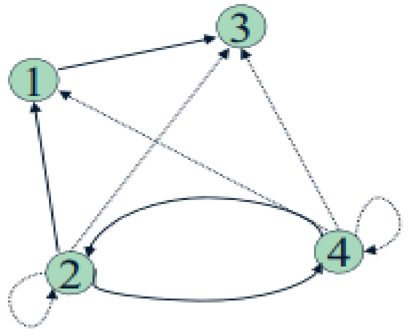
0 otherwise

1 if there is a path from v_i to v_j. The path should not use any other nodes except v_1, v_2, …., v_k

- Example of transitive closure:
-



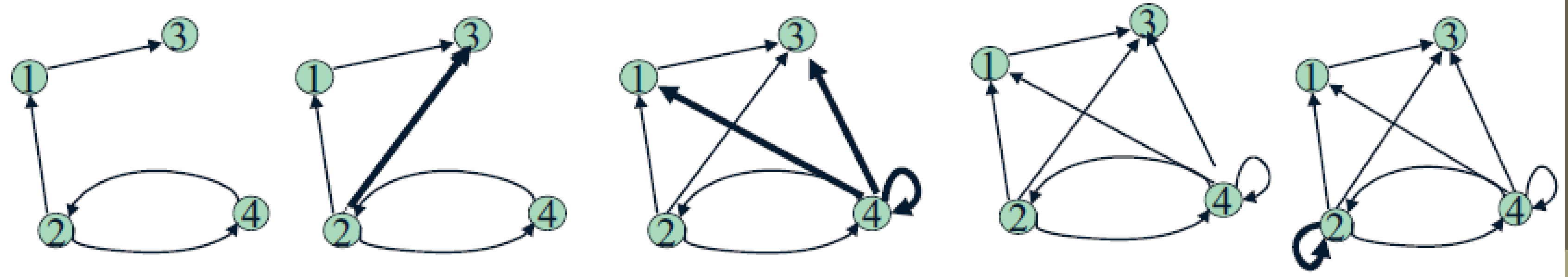
$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



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

Warshall’s algorithm

- This means, if P0[i][j] = 1, then there exists an edge from node vi to vj.
- If P1[i][j] = 1, then there exists an edge from vi to vj that does not use any other vertex except v1.
  - If P2[i][j] = 1, then there exists an edge from vi to vj that does not use any other vertex except v1 and v2.
- Note P0 is equal to the adjacency matrix of G. If G contains n nodes then Pn = P which is the path matrix of the graph G.



P_0	P_1	P_2	P_3	P_4
0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0
1 0 0 1	1 0 **1** 1	1 0 1 1	1 0 1 1	1 **1** 1 1
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
0 1 0 0	0 1 0 0	**1** 1 **1** **1**	1 1 1 1	1 1 1 1

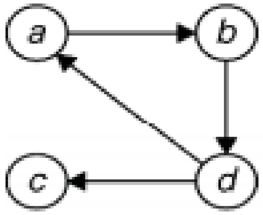
Warshall’s algorithm

- Therefore, we can conclude that, Pk[i][j] is equal to 1 only when either of the two cases occur:
  - There is a path from vi to vj that does not use any other node except v1, v2, …, vk-1. Therefore, **Pk-1[i][j] = 1**.
  - There is a path from vi to vk and a path from a path from vk to vj where every edge does not use any other node except v1, v2, …, vk-1. Therefore,
    - **Pk-1[i][k] = 1 AND Pk-1[k][j] = 1**

- Hence, the path matrix Pn can be calculated with the formula given as,
- **Pk[i][j] = Pk-1[i][j] V (Pk-1[i][k] Λ Pk-1[k][j])**

Warshall's algorithm

```
Warshall's algorithm to find path matrix P

Step 1: [Initialize the Path Matrix] Repeat Step
  2 for I = 0 to n-1, where n is the number of
  nodes in the graph
Step 2: Repeat Step 3 for J = 0 to n-1
Step 3: IF A[I][J] = 0, then SET P[I][J] = 0
           ELSE P[I][J] = 1
     [END OF LOOP]
  [END OF LOOP]
Step 4: [Calculate the path matrix P] Repeat
           Step 5 for K = 0 to n-1
Step 5: Repeat Step 6 for I = 0 to n-1
Step 6: Repeat Step 7 for J=0 to n-1
Step7: SET $P_K[I][J] = P_{K-1}[I][J] \vee (P_{K-1}[I][K]$
                        $\wedge P_{K-1}[K][J])$
Step 8: EXIT
```



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

8-7

Application of Warshall's algorithm to the digraph shown. New ones are in bold.

Floyd -Warshall's algorithm

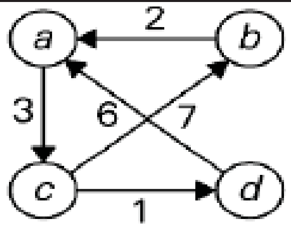
- The modified Warshall's algorithm is used to obtain a matrix that gives the shortest paths between the every nodes in weighted graph G with positive and negative edges.
- As an input to the algorithm, we will take the Adjacency matrix A of G and replace all those values of A which are zero by infinity (∞). Infinity (∞) denotes a very large number and indicates that there no path between the vertices.
- In Warshall's modified algorithm, we will obtain a set of matrices Q0, Q1, Q2,…., Qm using the formula given below.

  Qk[i][j] = Minimum(Mk-1[i][j], Mk-1[i][k] + Mk-1[k][j])
- Q0 is exactly the same as A with a little difference that every element that has a zero value in A is replaced by (∞) in Q0. Using the formula given above the matrix Qn will give the path matrix that has the shortest path between the vertices of the graph.

Floyd - Warshall's algorithm

```
Floyd-Warshall's algorithm to find the shortest path matrix P

Step 1: [Initialize the Shortest Path Matrix, Q] Repeat Step 2
  for I = 0 to n-1, where n is the number of nodes in the graph
Step 2: Repeat Step 3 for J = 0 to n-1
Step 3: IF A[I][J] = 0, then SET Q[I][J] = Infinity (or 9999)
              ELSE Q[I][J] = A[I][j]
       [END OF LOOP]
  [END OF LOOP]
Step 4: [Calculate the shortest path matrix Q] Repeat Step 5 for
              K = 0 to n-1
Step 5: Repeat Step 6 for I = 0 to n-1
Step 6: Repeat Step 7 for J=0 to n-1
Step7: IF Q[I][J] <= Q[I][K] + Q[K][J]
              SET Q[I][J] = Q[I][J]
       ELSE SET Q[I][J] = Q[I][K] + Q[K][J]
       [END OF IF]
       [END OF LOOP]
  [END OF LOOP]
  [END OF LOOP]
Step 8: EXIT
```

Floyd - Warshall’s algorithm



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

8-12

Application of Floyd’s algorithm to the graph shown. Updated elements are

THANK YOU!