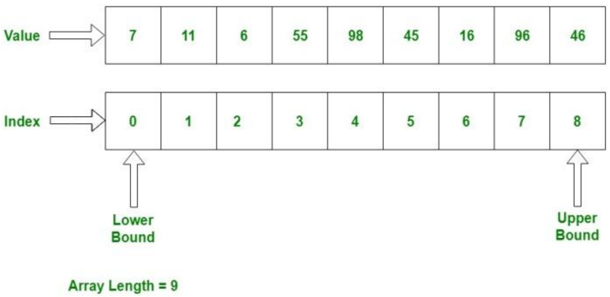
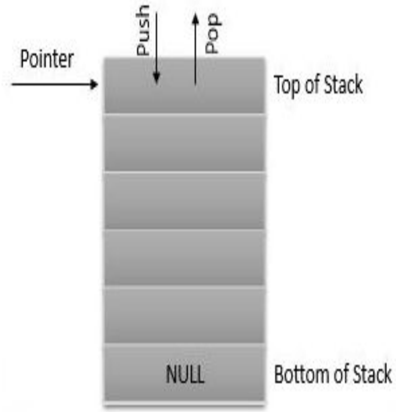
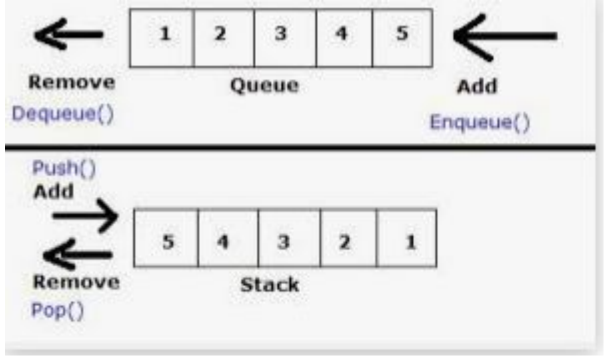
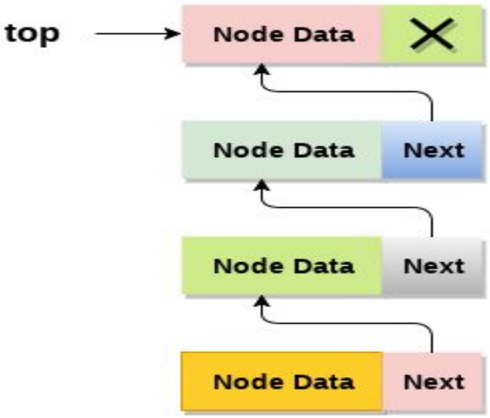
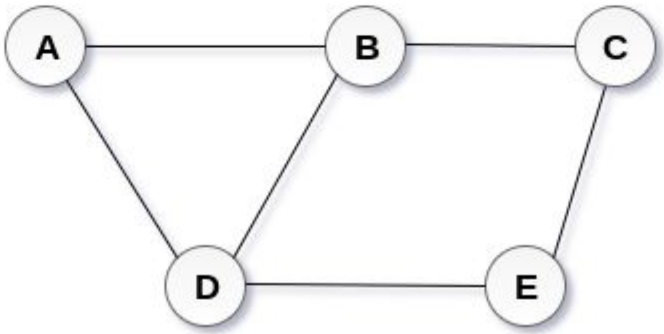


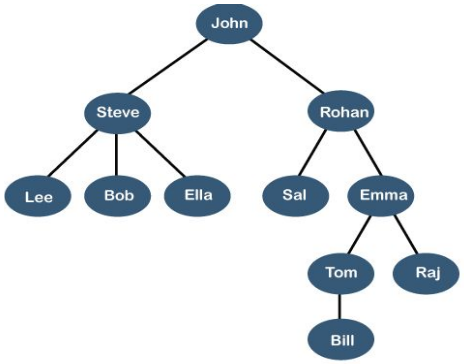




Non Linear Data Structure Part 2[Graph]





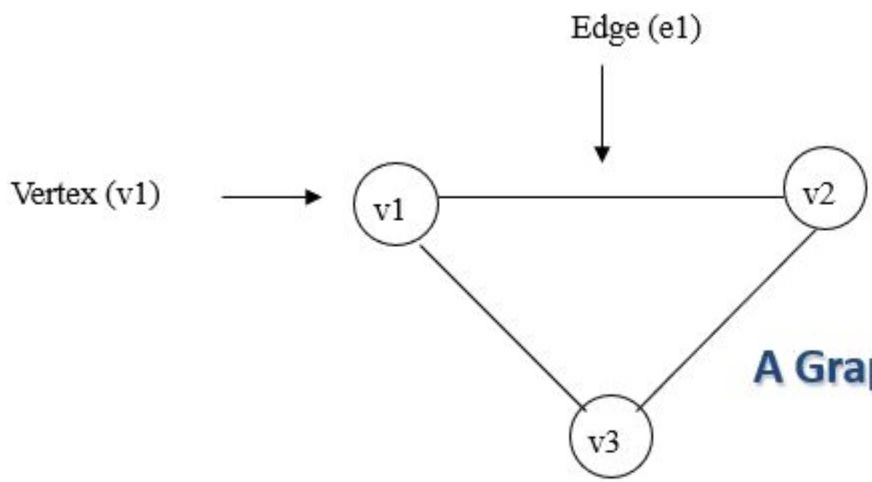


Outline

- Definition of Graph
- Representation of Graphs
- Types of Graph
- Graph Traversal [Depth First Search, Breadth First Search]
- Graph Traversal and Spanning Forest
- Minimum Spanning Tree [Prim’s Algorithm and Kruskal’s Algorithm]
- Finding the Shortest Path [Warshall’s Algorithm, Dijkstra’s Technique]

Basic Terminology

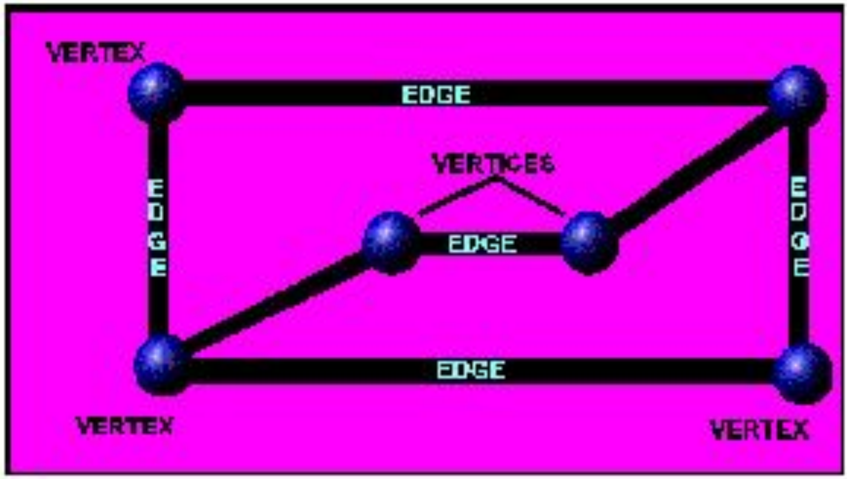
- A graph G = (V(G),E(G)) consist of two finite sets :
- A graph is a nonempty set of nodes (vertices) and a set of arcs (edges) such that each arc connects two nodes. Here V(G) represents vertices set and E(G) represents edge.

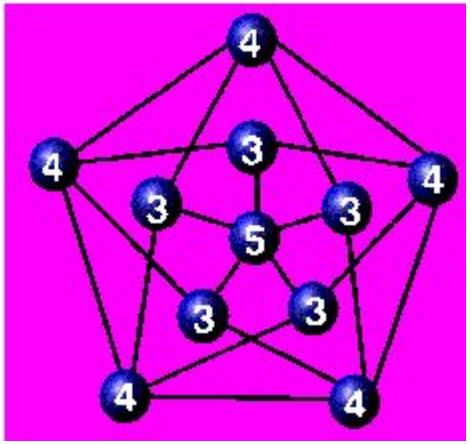


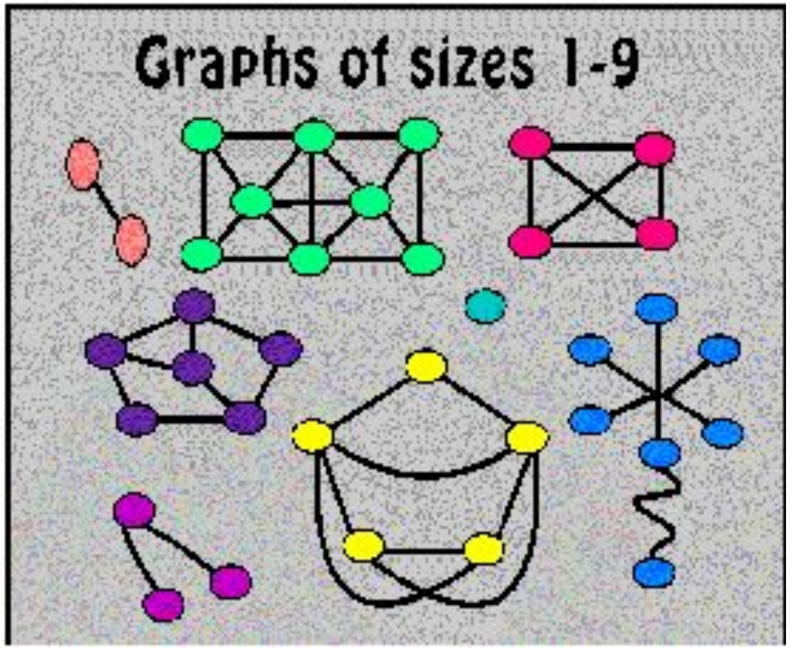
A Graph with 3 vertices and 3 edges

Edge, Vertex, Degree, Size

- Edge – any line drawn from one dot to another
- Vertex – Each dot/point/nodes that appears in a collaboration of dots
- Degree – (of a vertex) The number of edges that touch a vertex
- Size – of a graph is the number of vertices that the graph has

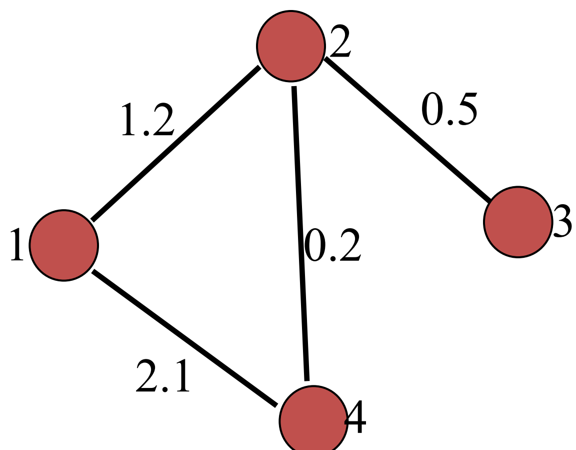


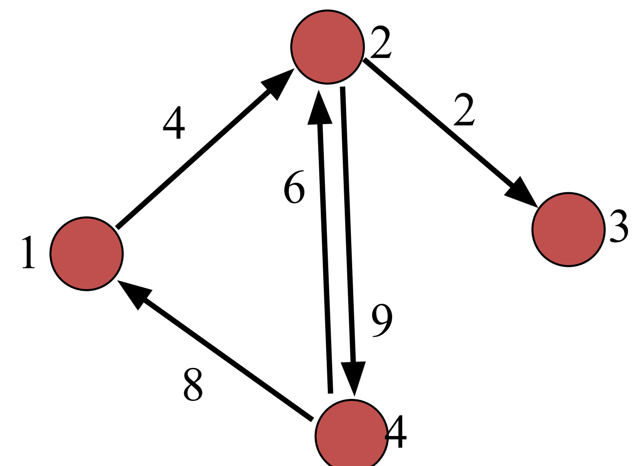




Weighted Graph

- A weighted graph is a graph for which each edge has an associated weight, usually given by a weight function w: E → R
- This could represent distance, energy consumption, cost, etc
- A graph where each edge has a weight is termed an ***weighted* graph**





- A graph where edges do not have any associated weights is termed an ***unweighted* graph**
- An *unweighted graph* may be considered to be a weighted graph where all edges have weight 1

Representation of Graph

- **Matrix :** With help of Adjacency Matrix.
- **Linked list** : With help of Adjacency List
- Comparing the two representation
  - Space Complexity
    - Adjacency matrix is $O(n^2)$
    - Adjacency list is O(n+E) where E is no. of edges.
  - Static versus dynamic representation
    - An adjacency matrix is static representation : the graph is built in one go and is difficult to alter once built.
    - An adjacency list is a dynamic representation : the graph is built incrementally, thus is more easily altered during run-time

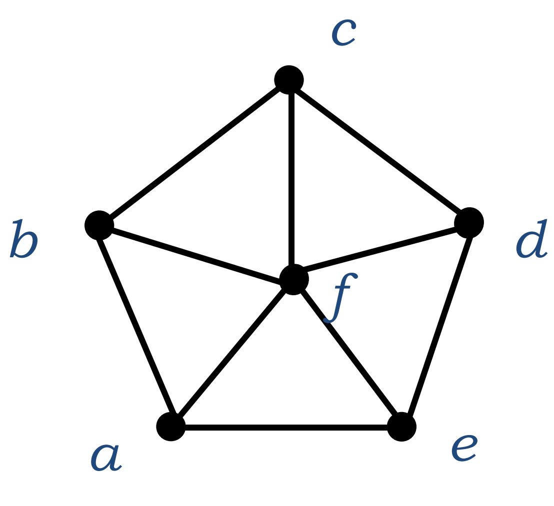
Adjacency Matrix

- A simple graph G = (V,E) with n vertices can be represented by its adjacency matrix, A, where the entry aij in row i and column j is:

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge in } G \\ 0 & \text{otherwise} \end{cases}$$

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

Adjacency Matrix

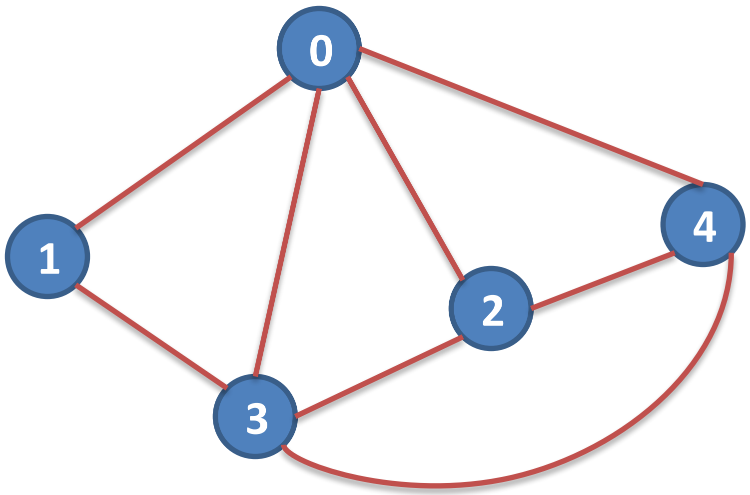


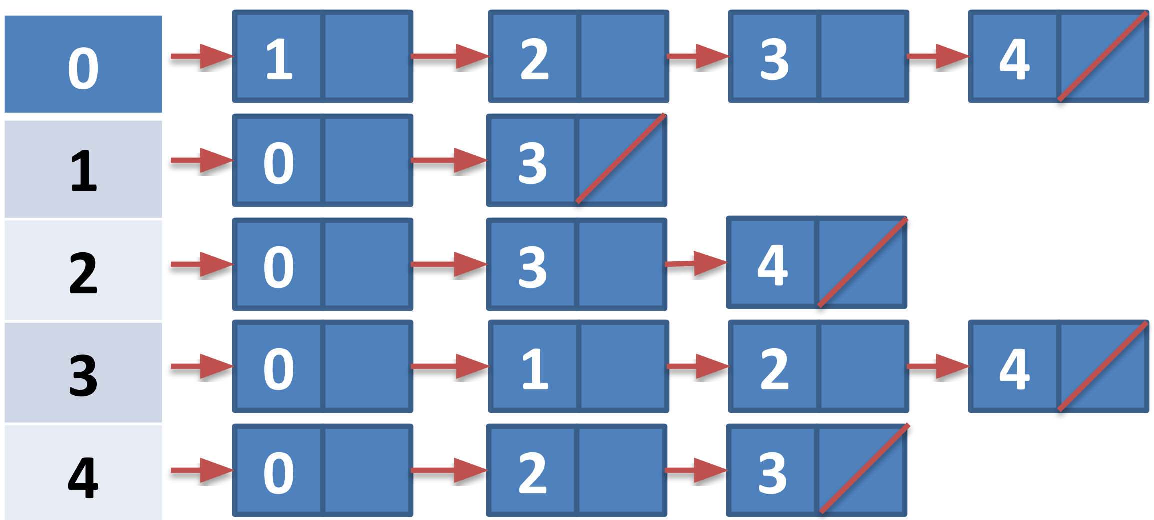
W_5

$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 1 & 0 & 0 & 1 & 1 \\ b & 1 & 0 & 1 & 0 & 0 & 1 \\ c & 0 & 1 & 0 & 1 & 0 & 1 \\ d & 0 & 0 & 1 & 0 & 1 & 1 \\ e & 1 & 0 & 0 & 1 & 0 & 1 \\ f & 1 & 1 & 1 & 1 & 1 & 0 \end{array}$$

(Rows: From; Columns: To)

Adjacency List Representation





Traversing a Graph

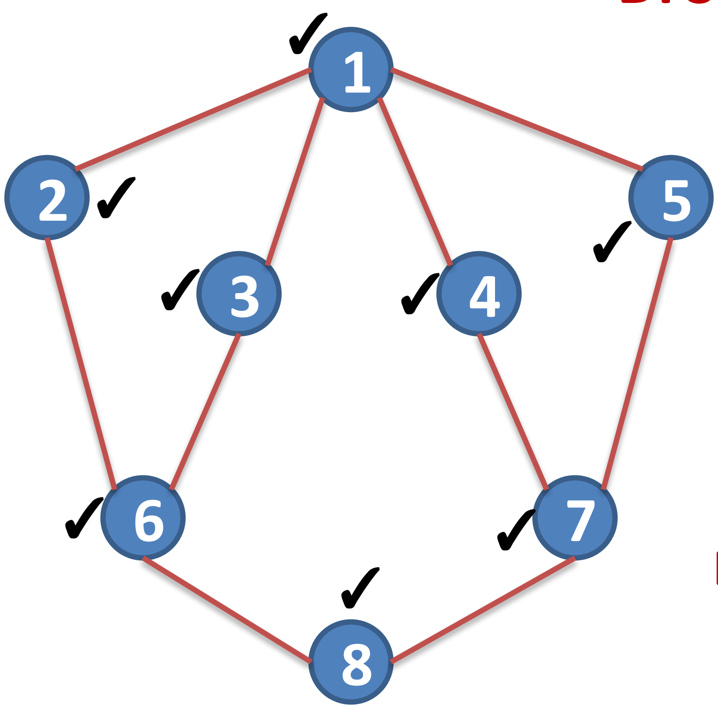
- Traversing a graph means visiting all the vertices of the graph exactly once.
- It can be started from any vertex of the graph.
- Graph traversing algorithms are :
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- Common Traversal Steps
  1. Start from any vertex of a graph
  2. From this starting vertex, traverse as deep as you can go. Whenever you cannot go further, then backtrack one vertex and do the same traversal from this vertex until you cannot traverse further, and so on.
  3. Process the information contained in that vertex.
  4. Then move along an edge to process a neighbor (adjacent vertex)
  5. When the traversal finishes, all the above vertices that can be reached from the start vertex are processed.

Depth-First Search [DFS]

- Once a possible path is found, continue the search until the end of the path
- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node
  3. If the current node has no unvisited neighbors, backtrack to its parent, and make that parent the new current node
  4. Repeat steps 3 and 4 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

Depth-First Search [DFS]

- It is like preorder traversal of tree [Root,Left,Right]
- Traversal can start from any vertex Vi
- Vi is visited and then all vertices adjacent to Vi are traversed recursively using DFS



DFS (G, 1) is given by

Step 1: Visit (1)

Step 2: DFS (G, 2) → **DFS (G, 2):**
DFS (G, 3)
DFS (G, 4)
DFS (G, 5)

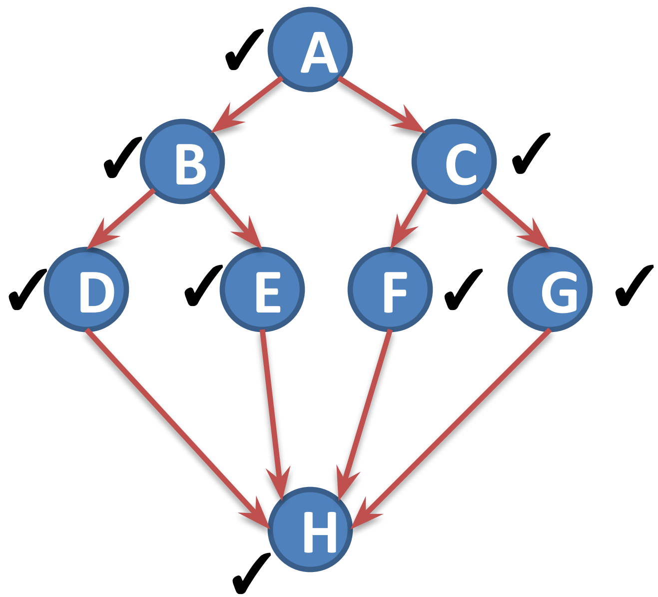
Step1: Visit(2)
Step 2: DFS (G, 6) → **DFS (G, 6):**

Step1: Visit(6)
Step 2: DFS (G, 3)
DFS (G, 8)

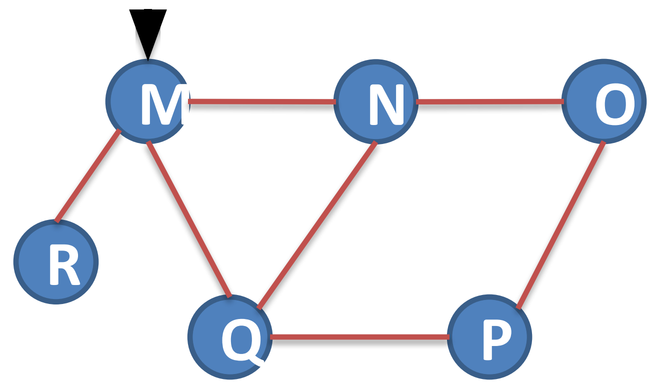
DFS of given graph starting **from** node **1** is given by

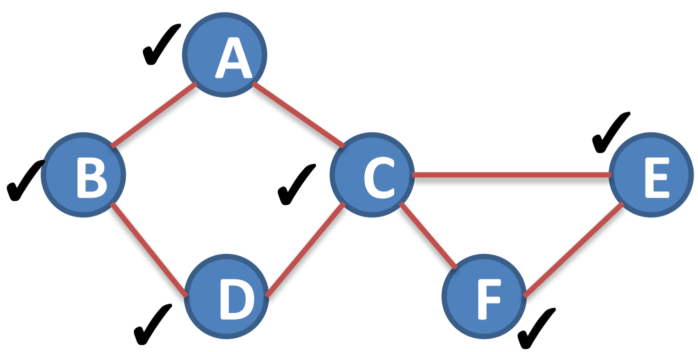
1 2 6 3 8 7 4 5

Depth-First Search [DFS]



A B D H E C F G





A B D C F E

Depth-First Search Program

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start){
        int stack[MAX];
        int top = -1, i;
        printf("%c–",start + 65);
        visited[start] = 1;
        stack[++top] = start;
        while(top != -1)       {
                start = stack[top];
                for(i = 0; i < MAX; i++) {
                        if(adj[start][i] && visited[i] == 0) {
                                stack[++top] = i;
                                printf("%c–", i + 65);
                                visited[i] = 1;
                                break;
                        }
                }
                if(i == MAX)
                        top--;
        }
}
```

Depth-First Search Program

```
int main()
{
        int adj[MAX][MAX] = {{0,1,0,1,0},{1,0,1,1,0},{0,1,0,0,1},{0,0,1,1,0}};
        int visited[MAX] = {0}, i, j;
       //printf("\n Enter the adjacency matrix: ");
        //for(i = 0; i < MAX; i++)
        //    for(j = 0; j < MAX; j++)
                   //scanf("%d", &adj[i][j]);
       printf("DFS Traversal: ");
       depth_first_search(adj,visited,0);
       printf("\n");
       return 0;
}
```

Output

DFS Traversal: A–B–C–E–D–

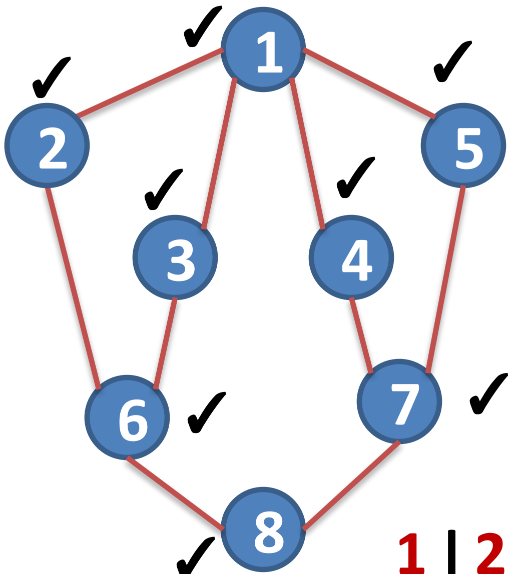
Breadth-First Search [BFS]

- Start several paths at a time, and advance in each one step at a time
- BFS follows the following rules:
  1. First choose a starting vertex
  2. Find all the vertices which are connected to the starting vertex.
  3. Then choose one of the connected vertices that are connected to this vertex.
  4. Continue this procedure until all the vertices are visited.
- Implementation of BFS
  - Observations: The first node visited in each level is the first node from which to proceed to visit new nodes.
  - This suggests that a queue is the proper data structure to remember the order of the steps.

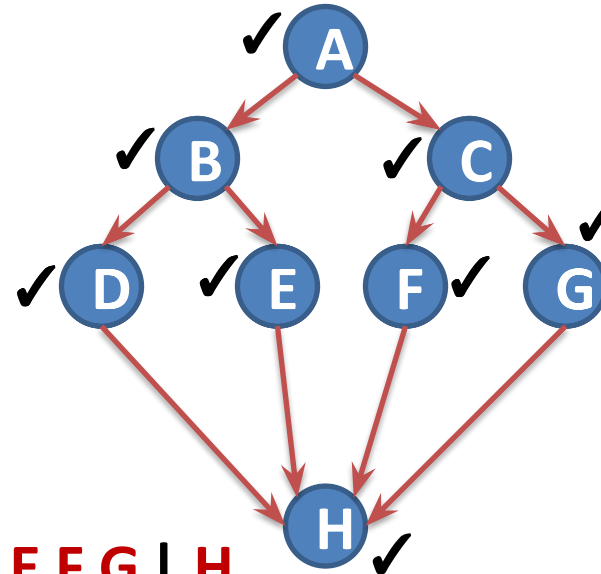
Breadth-First Search [BFS]

- This methods starts from vertex V_0
- V_0 is marked as visited. All vertices adjacent to V_0 are visited next
- Let vertices adjacent to V0 are V_1, V_2, V_3, V_4
- V_1, V_2, V_3 and V_4 are marked visited
- All unvisited vertices adjacent to V_1, V_2, V_3, V_4 are visited next
- The method continuous until all vertices are visited
- The algorithm for BFS has to maintain a list of vertices which have been visited but not explored for adjacent vertices
- The vertices which have been visited but not explored for adjacent vertices can be stored in queue

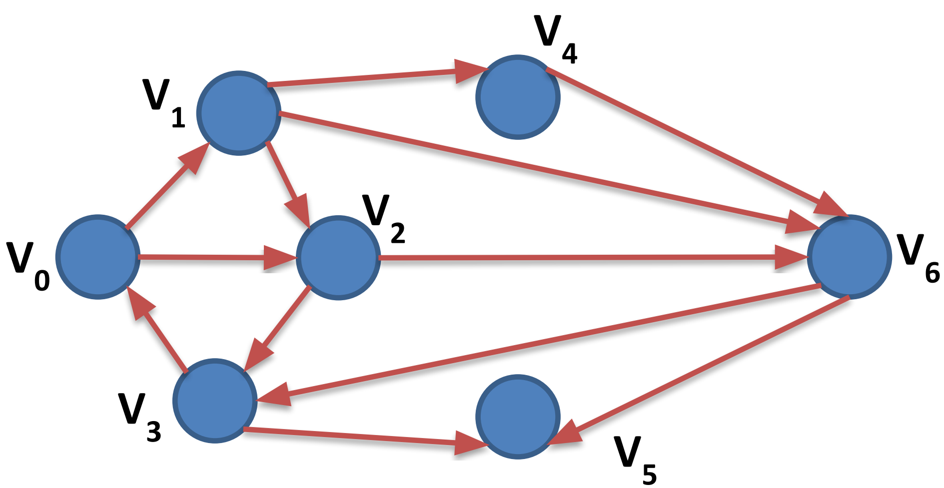
Breadth-First Search [BFS]



1 | 2 3 4 5 | 6 7 | 8



A | B C | D E F G | H



V_0 | V_1 V_2 | V_4 V_6 V_3 | V_5

Breadth-First Search Program

```
#include <stdio.h>
#define MAX 5
void breadth_first_search(int adj[][MAX],int visited[],int start){
	int queue[MAX],rear = -1,front = -1, i;
	queue[++rear] = start;
	visited[start] = 1;
	while(rear != front){
		start = queue[++front];
		if(start == 5)
			printf("5\t");
		else
			printf("%c \t",start + 65);
		for(i = 0; i < MAX; i++) {
			if(adj[start][i] == 1 && visited[i] == 0) {
				queue[++rear] = i;
				visited[i] = 1;
			}
		}
	}
}
```

Breadth-First Search Program

```
int main()
{
	int visited[MAX] = {0};
	int i,j;
	//int adj[MAX][MAX];
	int adj[MAX][MAX] = {{0,1,0,1,0},{1,0,1,1,0},{0,1,0,0,1},{0,0,1,1,0}};
	//printf("\n Enter the adjacency matrix: ");
	 /*for(i = 0; i < MAX; i++)
		for(j = 0; j < MAX; j++)
		{ printf("\n Enter % d x %d :",i,j);
				scanf("%d", &adj[i][j]);}*/
	printf("\n BFS Traversal: ");
	breadth_first_search(adj,visited,0);
	return 0;
}
```

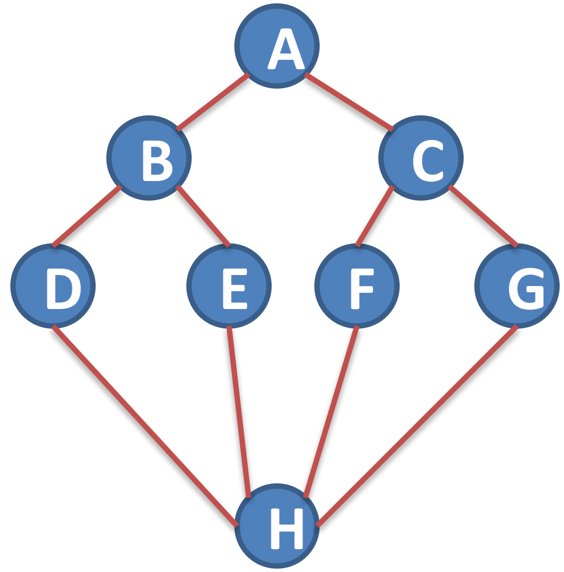
Output

BFS Traversal: A B D C E

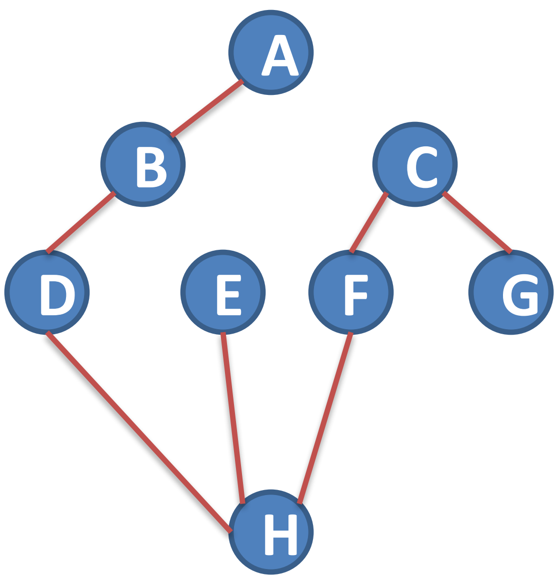
Minimal Spanning Tree(MST)

- **Weighted Spanning Tree:** If graph G is weighted graph, then the weight of a spanning tree T of G is defined as the sum of the weights of all the branches in T then T is called the weighted spanning tree.
- **Minimal Spanning Tree:** A spanning tree with the smallest weight in a weighted graph is called a shortest spanning tree or shortest-distance spanning tree or minimal spanning tree.
- Two techniques for Constructing minimum cost spanning tree
  - Prim’s Algorithm
  - Kruskal’s Algorithm

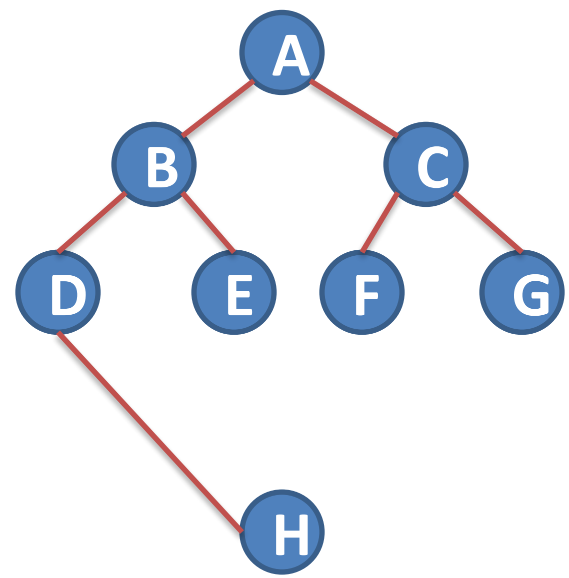
Construct Spanning Tree



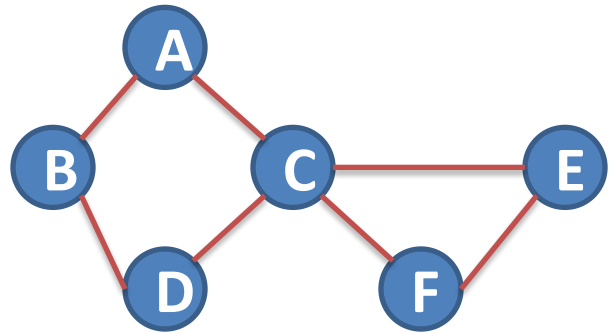
DFS Spanning Tree



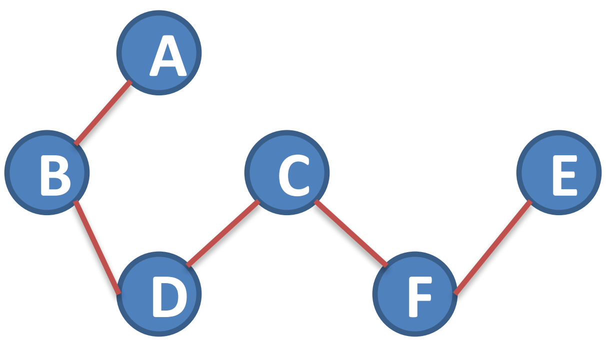
BFS Spanning Tree



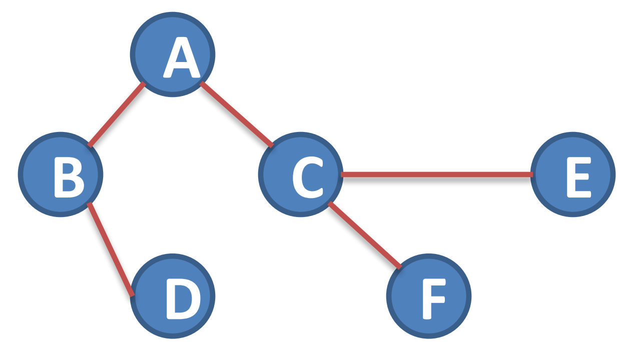
Construct Spanning Tree



DFS Spanning Tree



BFS Spanning Tree

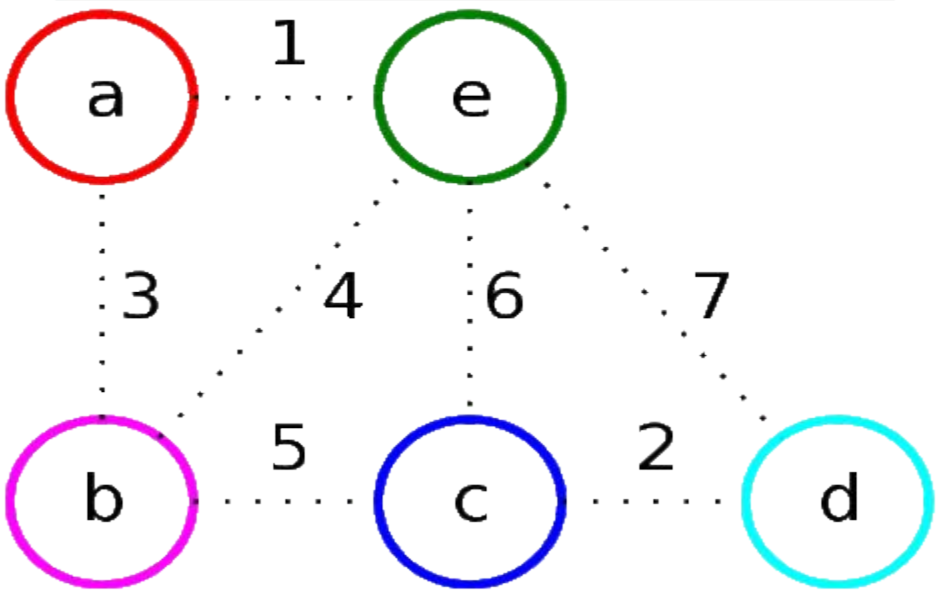


Minimal Spanning Tree Algorithms

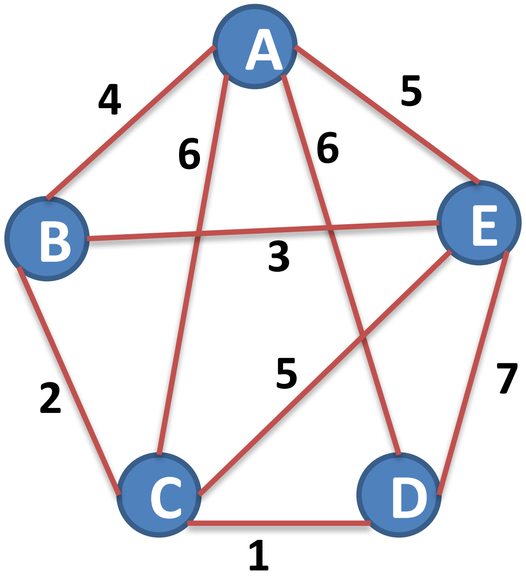
Kruskal's algorithm

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6

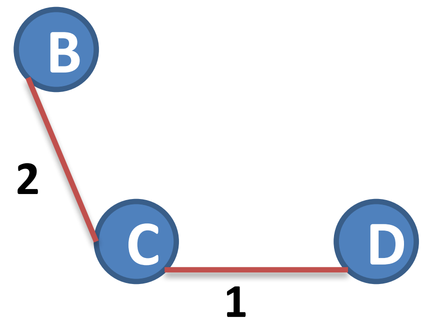


Kruskal’s Algorithms

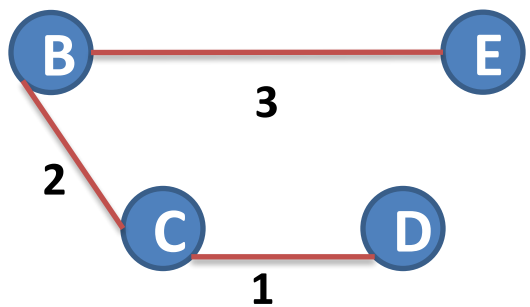


Step 1: Taking min edge (C,D)

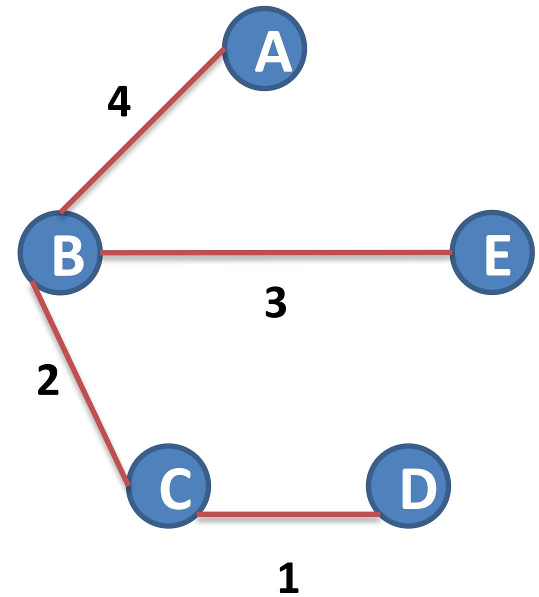
Step 2: Taking next min edge (B,C)



Step 3: Taking next min edge (B,E)



Step 4: Taking next min edge (A,B)



so we obtained minimum spanning tree of cost:
4 + 2 + 1 + 3 = 10

Implementing Kruskal Algorithms

```
#include<stdio.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
int main()
{
   printf("\nEnter the no. of vertices:");
   scanf("%d",&n);
   printf("\nEnter the cost adjacency matrix\n");
   for(i=1;i<=n;i++)
  {  printf("\n");
     for(j=1;j<=n;j++)
     {
         printf("a[%d][%d]: ",i,j);
         scanf("%d",&cost[i][j]);
         if(cost[i][j]==0)
              cost[i][j]=999;
       }
   }
```

Implementing Kruskal Algorithms

```
  printf("\nThe edges of Minimum Cost Spanning Tree
    are\n\n");
  while(ne<n)  {
      for(i=1,min=999;i<=n;i++)      {
       for(j=1;j<=n;j++)   {
            if(cost[i][j]<min)    {
                 min=cost[i][j];
                  a=u=i;
                  b=v=j;
             }
         }
      }
      u=find(u);
      v=find(v);
      if(uni(u,v))  {
         printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
         mincost +=min;  }
     cost[a][b]=cost[b][a]=999; }//while ended
  printf("\n\tMinimum cost = %d\n",mincost);
  return 0;
}
```

Implementing Kruskal Algorithms

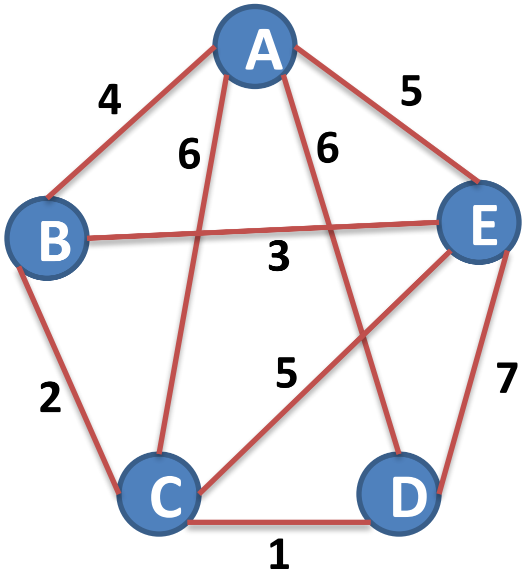
```
int find(int i)
{
    while(parent[i])
        i=parent[i];
    return i;
}

int uni(int i,int j)
{
     if(i!=j)
     {
         parent[j]=i;
         return 1;
      }
      return 0;
}
```

Minimal Spanning Tree Algorithms

Prim's algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

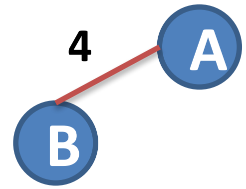
Prims Algorithms



A – B | 4	A – D | 6	C – E | 5
A – E | 5	B – E | 3	C – D | 1
A – C | 6	B – C | 2	D – E | 7

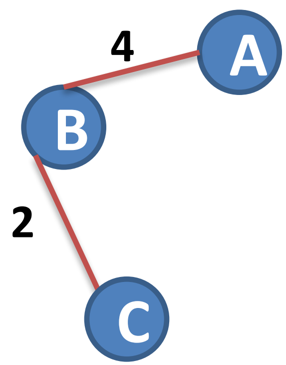
Let X be the set of nodes explored, initially X = { A }

Step 1: Taking minimum Weight edge of all Adjacent edges of X={A}



X = { A , B }

Step 2: Taking minimum weight edge of all Adjacent edges of X = { A , B }

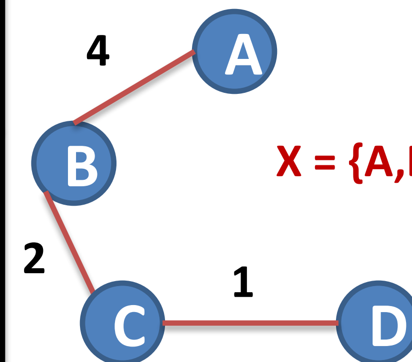


X = {A ,B,C}

We obtained minimum spanning tree of cost:

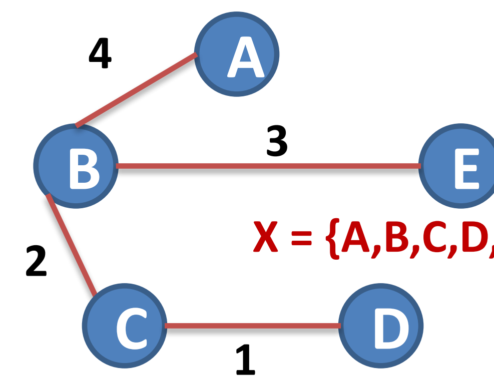
4 + 2 + 1 + 3 = 10

Step 3: Taking minimum weight edge of all Adjacent edges of X = { A , B , C }



X = {A,B,C,D}

Step 4: Taking minimum weight edge of all Adjacent edges of X = {A ,B ,C ,D }



X = {A,B,C,D,E}

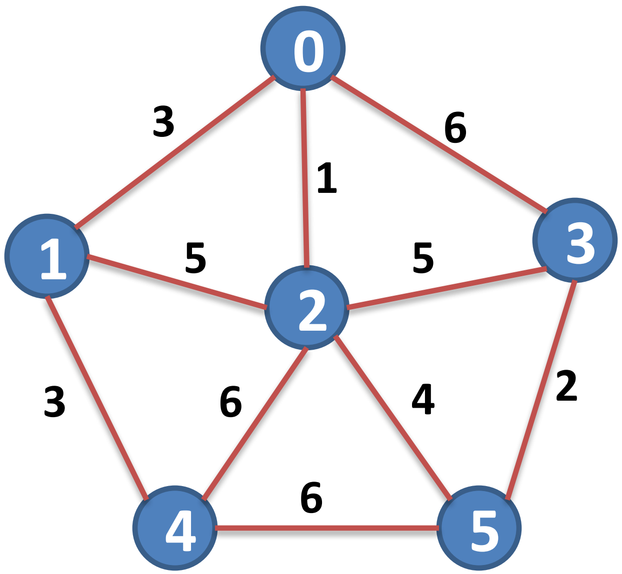
Implementing Prim’s Algorithm

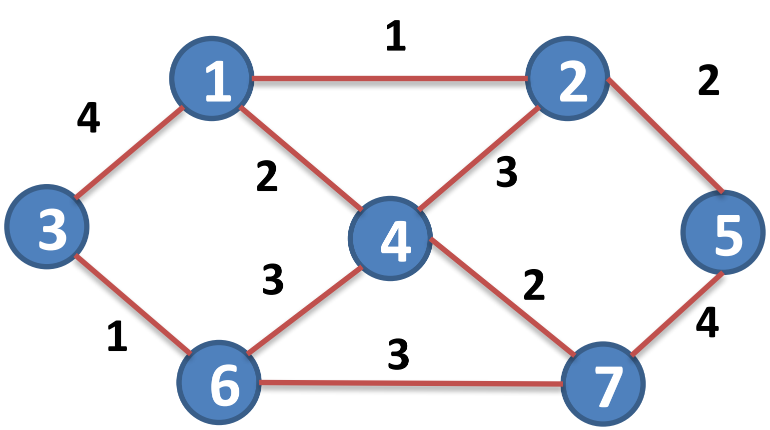
```
# include<stdio.h>
int G[50][50],select[50], i, j, k, n, min_dist,total=0,u, v;
/*  This function finds the minimal spanning tree by Prim's Algorithm   */
void Prim()
{
  printf("\n\n The Minimal Spanning Tree Is :\n");
  select[0] = 1;
  for (k=1 ; k<n ; k++) {
      min_dist = 32767;
      for (i=0 ; i<n ; i++)
          for (j=0 ; j<n ; j++)
              if (G[i][j] && ((select[i] && !select[j]) || (!select[i] && select[j])))
                 if (G[i][j] < min_dist)      {
                     min_dist = G[i][j];
                     u = i;
                     v = j;                   } //end of if
      printf("\n Edge (%d %d )and weight = %d",u,v,min_dist);
      select[u] = select[v] = 1;
      total =total+min_dist;     }//end of for
   printf("\n\n\t Total Path Length Is = %d",total);
}
```

Implementing Prim’s Algorithm

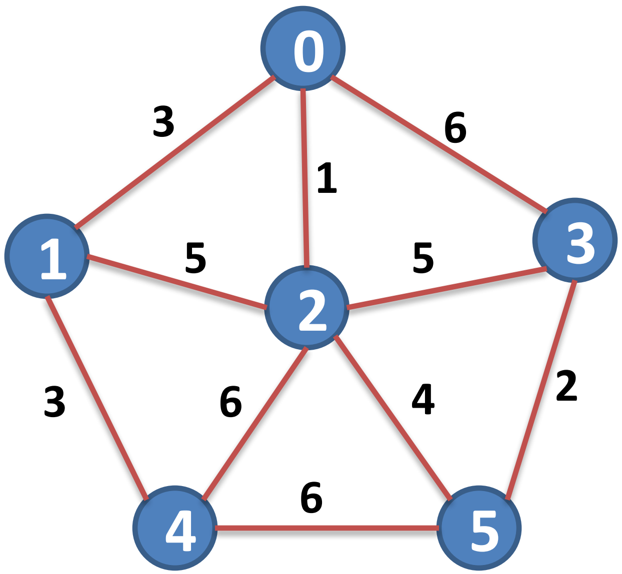
```
int main()
{
  printf("\n Enter Number of Nodes in The Graph: ");
  scanf("%d",&n);
  //entering weighted graph
  printf("\nEnter the cost adjacency matrix\n");
 for(i=0;i<n;i++)
  {  printf("\n");
    for(j=0;j<n;j++)
     {
       printf("a[%d][%d]: ",i,j);
       scanf("%d",&G[i][j]);
     }
  }
  Prim();
  return 0;
}
```

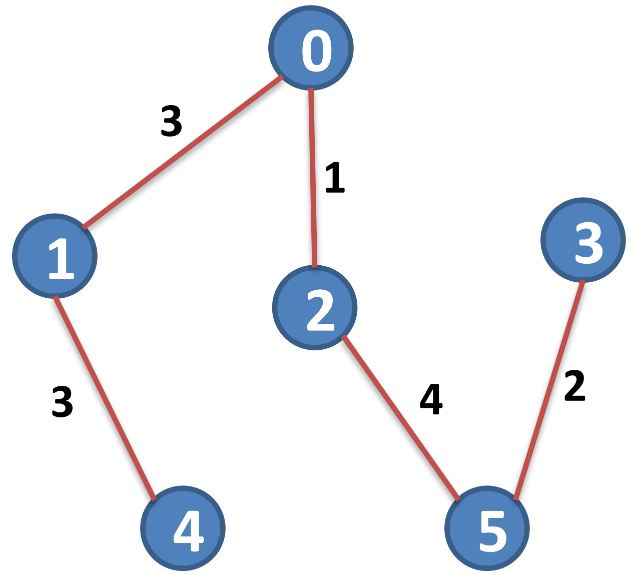
Construct Minimum Spanning Tree

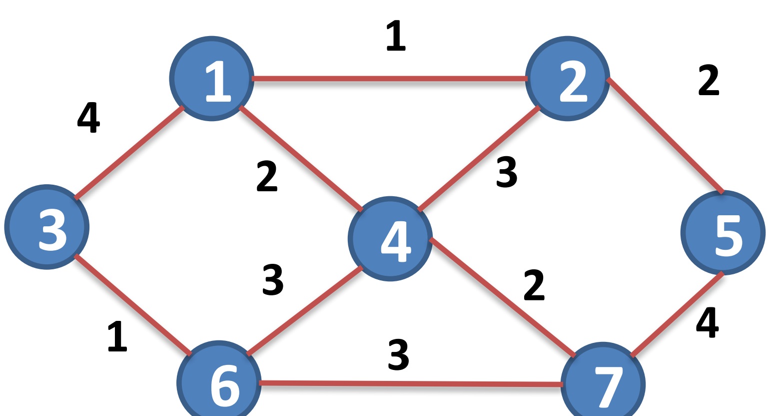


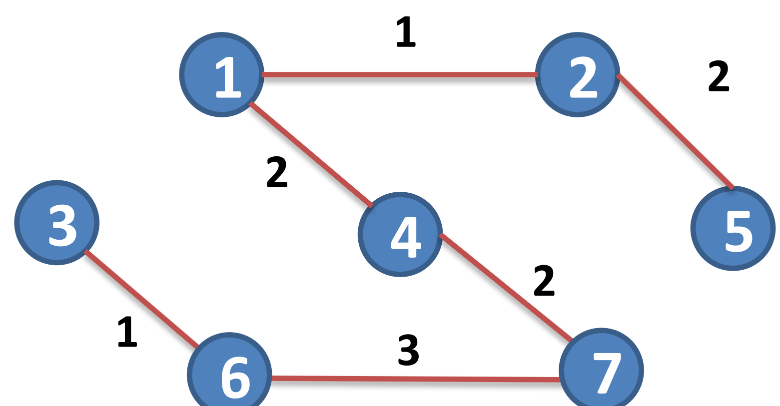


Construct Minimum Spanning Tree







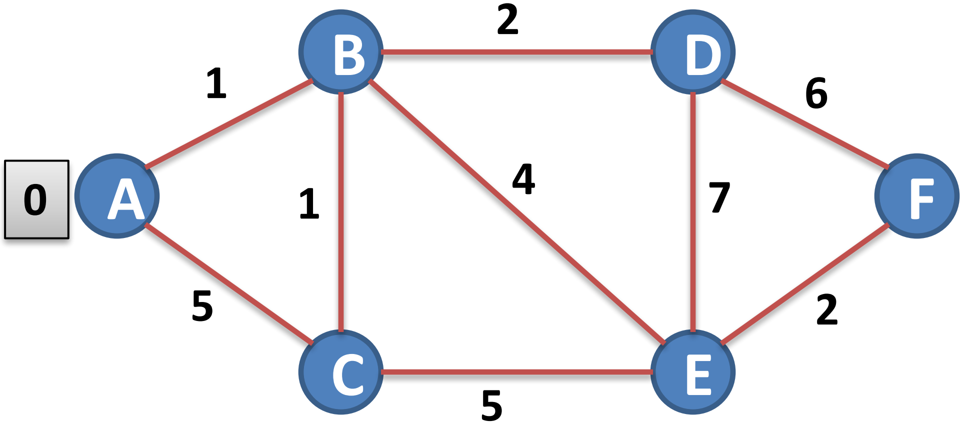


Finding the Shortest Path

- A weighted graph has values (weights)assigned to its edges.
- The length of a path = the sum of the weights of the edges in the path w(i,j) = weight of edge (i,j)
- The shortest path between two vertices is the path having the minimum length.

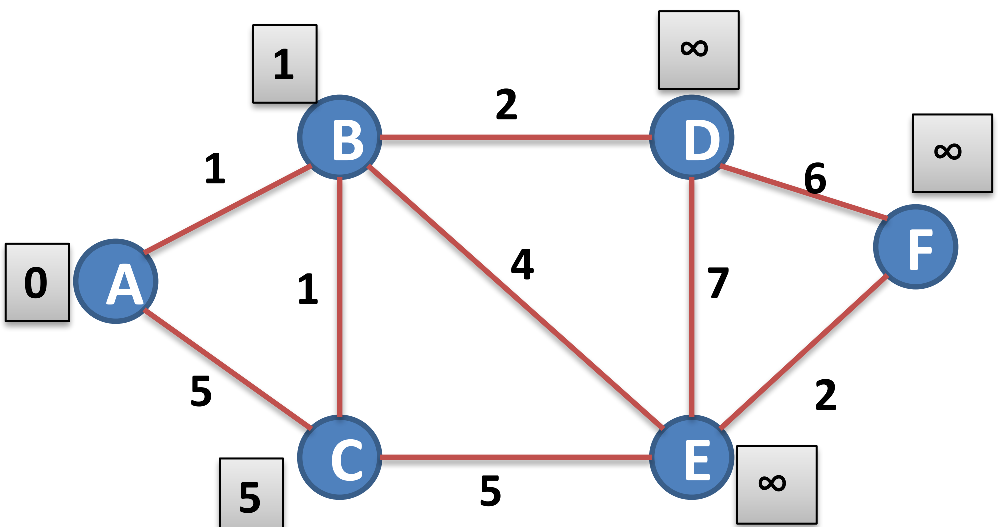
Edsger Wybe Dijkstra

Dijkstra's Algorithm



	A	B	C	D	E	F
Distance	0	∞	∞	∞	∞	∞
Visited	0	0	0	0	0	0

1st Iteration: Select **Vertex A** with minimum distance



	A	B	C	D	E	F
Distance	0	1 ∞	5 ∞	∞	∞	∞
Visited	1	0	0	0	0	0

Dijkstra’s Algorithm

2nd Iteration: Select **Vertex B** with minimum distance

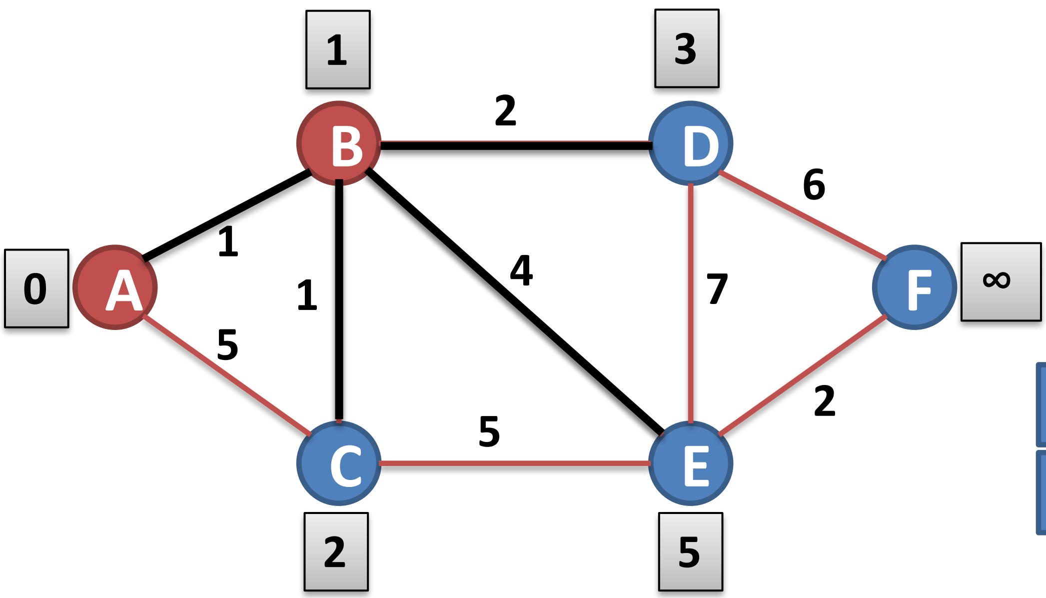
Cost of going to C via B = dist[B] + cost[B][C] = 1 + 1 = 2

Cost of going to D via B = dist[B] + cost[B][D] = 1 + 2 = 3

Cost of going to E via B = dist[B] + cost[B][E] = 1 + 4 = 5

Cost of going to F via B = dist[B] + cost[B][F] = 1 + ∞ = ∞

	A	B	C	D	E	F
Distance	0	1	5	∞	∞	∞
Visited	1	0	0	0	0	0



	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	0	0	0	0

Dijkstra's Algorithm

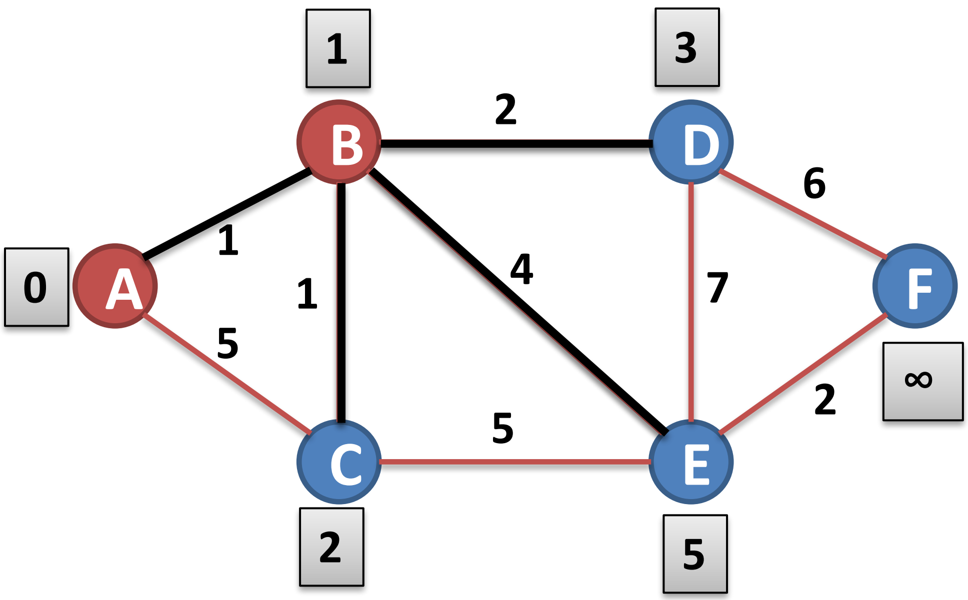
3rd Iteration: Select **Vertex C** via B with minimum distance

Cost of going to D via C = dist[C] + cost[C][D] = 2 + ∞ = ∞

Cost of going to E via C = dist[C] + cost[C][E] = 2 + **5** = **7**

Cost of going to F via C = dist[C] + cost[C][F] = 2 + ∞ = ∞

	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	0	0	0	0



	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	1	0	0	0

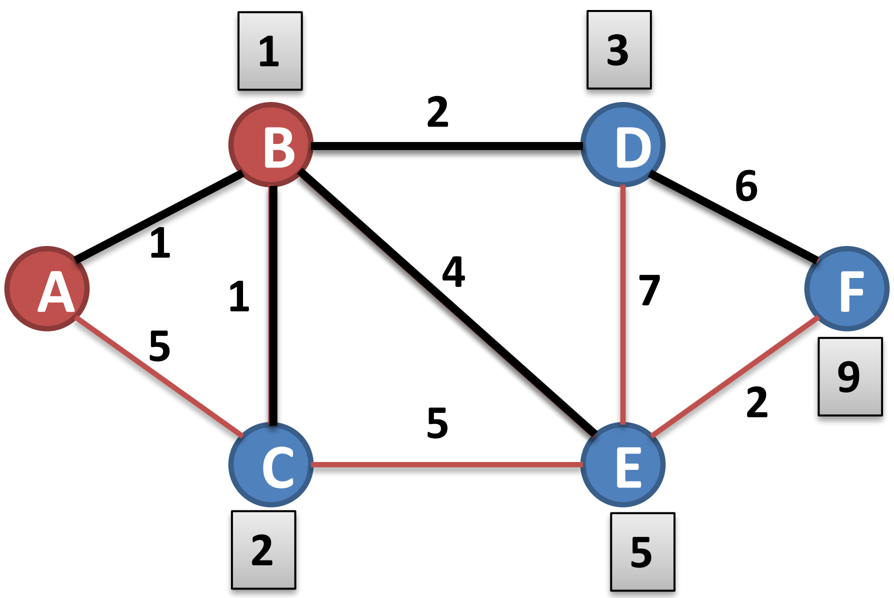
Dijkstra's Algorithm

4th Iteration: Select **Vertex D** via path A - B with minimum distance

Cost of going to E via D = dist[D] + cost[D][E] = 3 + **7** = **10**

Cost of going to F via D = dist[D] + cost[D][F] = 3 + **6** = **9**

	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	1	0	0	0



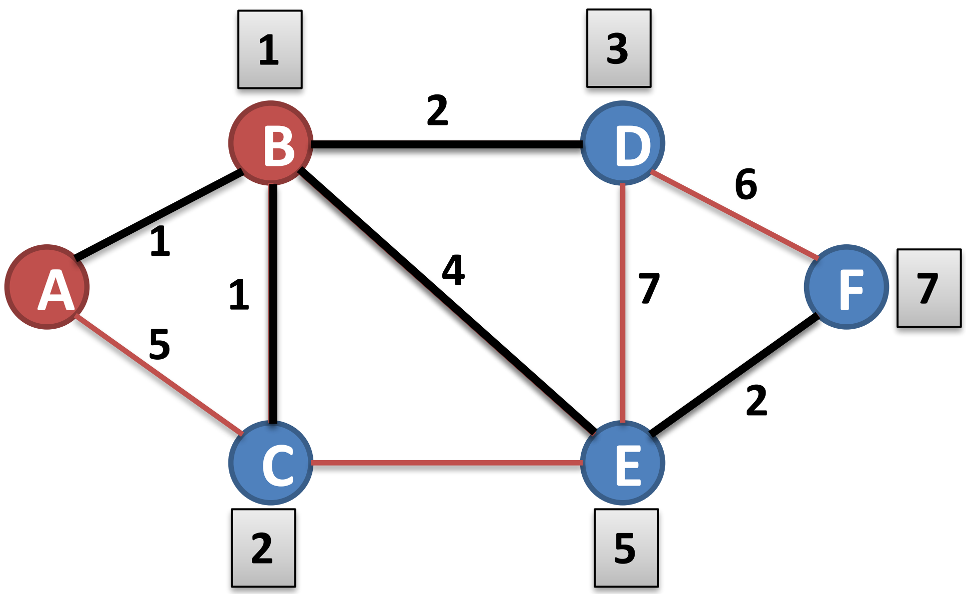
	A	B	C	D	E	F
Distance	0	1	2	3	5	9
Visited	1	1	1	1	0	0

Dijkstra’s Algorithm

4th Iteration: Select **Vertex E** via path A – B – E with minimum distance

Cost of going to F via E = dist[E] + cost[E][F] = 5 + **2** = **7**

	A	B	C	D	E	F
Distance	0	1	2	3	5	9
Visited	1	1	1	1	0	0

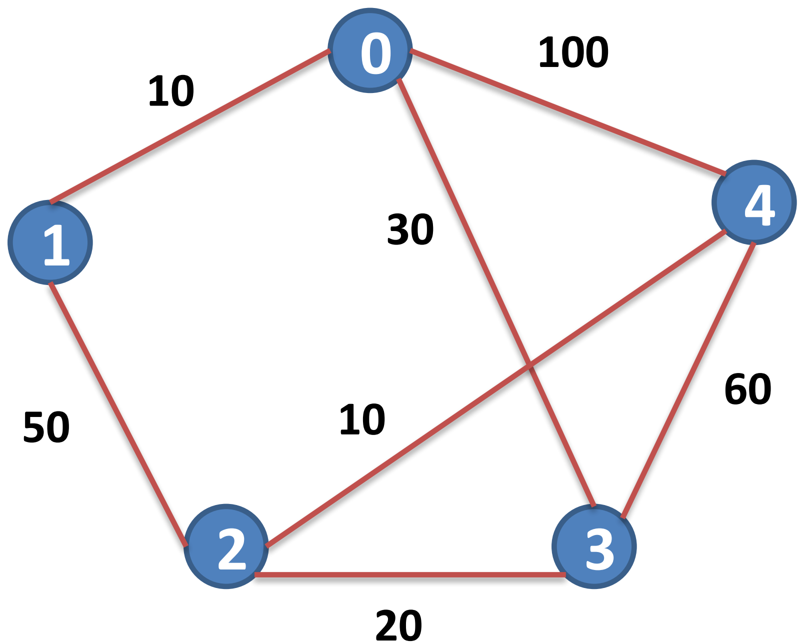


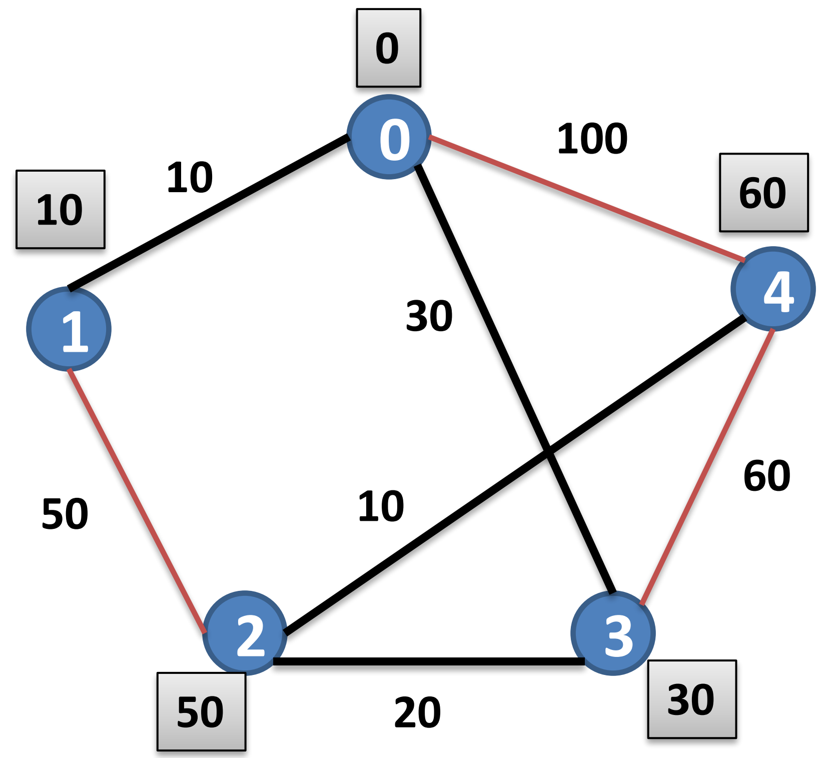
	A	B	C	D	E	F
Distance	0	1	2	3	5	7
Visited	1	1	1	1	1	0

Shortest Path from A to F is
A □ B □ E □ F = 7

Dijkstra’s Algorithm

Find out shortest path from node 0 to all other nodes using Dijkstra Algorithm





Warshall's algorithm of path matrix

Step 1: [Initialization of path matrix]

Repeat through step 2 for i=0,1,2,…n-1

Repeat through step 2 for j=0,1,2,…n-1

Step 2: [Test the condition and assign accordingly value to path matrix accordingly]

If (a[i][j] == 0)

p[i][j]=0

Else

p[i][j]=1

Step 3: [Evaluate path matrix]

Repeat through step 4 for k=0,1,2,….,n-1

Repeat through step 4 for i=0,1,2,….,n-1

Repeat through step 4 for j=0,1,2,….,n-1

Step 4:p[i][j]=p[i][j] V (p[i][k] **Λ** p[k][j]);

Step 5: Exit

V- OR **Λ - AND**

Modified Warshall’s algorithm

Step 1: [Initialization matrix m]

    Repeat through step 2 for i=0,1,2,…n-1
    Repeat through step 2 for j=0,1,2,…n-1

Step 2: [Test the condition and assign the required value to matrix m]

    If (a[i][j] == 0)
        m[i][j]=Infinity
    Else
        m[i][j]=a[i][j]

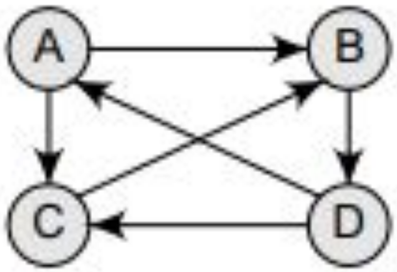
Step 3: [Shortest path Evaluation]

    Repeat through step 4 for k=0,1,2,….,n-1
    Repeat through step 4 for i=0,1,2,….,n-1
    Repeat through step 4 for j=0,1,2,….,n-1

Step 4: If m[i][j] < m[i][k] + m[k][j]

        m[i][j]=m[i][j]
    Else
        m[i][j] = m[i][k] + m[k][j]

Step 5: Exit



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

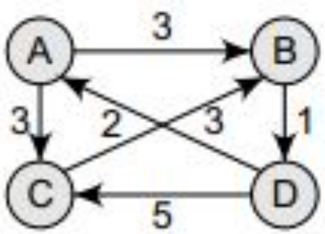
Figure 13.41 Graph G

$$\mathbf{Q0} = \begin{bmatrix} 9999 & 1 & 1 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 9999 \\ 1 & 9999 & 1 & 9999 \end{bmatrix}$$

$$\mathbf{Q1} = \begin{bmatrix} 9999 & 1 & 1 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 9999 \\ 1 & 2 & 1 & 9999 \end{bmatrix}$$

$$\mathbf{Q2} = \begin{bmatrix} 9999 & 1 & 1 & 2 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 2 \\ 1 & 2 & 9999 & 3 \end{bmatrix}$$

$$\mathbf{Q3} = \begin{bmatrix} 9999 & 1 & 1 & 2 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 1 & 9999 & 2 \\ 1 & 2 & 9999 & 3 \end{bmatrix} \qquad \mathbf{Q4} = \mathbf{Q} \begin{bmatrix} 3 & 1 & 1 & 2 \\ 2 & 3 & 2 & 1 \\ 3 & 1 & 3 & 2 \\ 1 & 2 & 1 & 3 \end{bmatrix}$$



$$\begin{pmatrix} 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & 5 & 0 \end{pmatrix}$$

Figure 13.42 Graph G

$$\mathbf{Q0} = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 9999 & 5 & 9999 \end{bmatrix}$$

$$\mathbf{Q1} = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 5 & 5 & 9999 \end{bmatrix}$$

$$\mathbf{Q2} = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$\mathbf{Q3} = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 6 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$\mathbf{Q4} = \mathbf{Q}\begin{bmatrix} 6 & 3 & 3 & 4 \\ 3 & 6 & 6 & 1 \\ 6 & 3 & 9 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$


thank you!