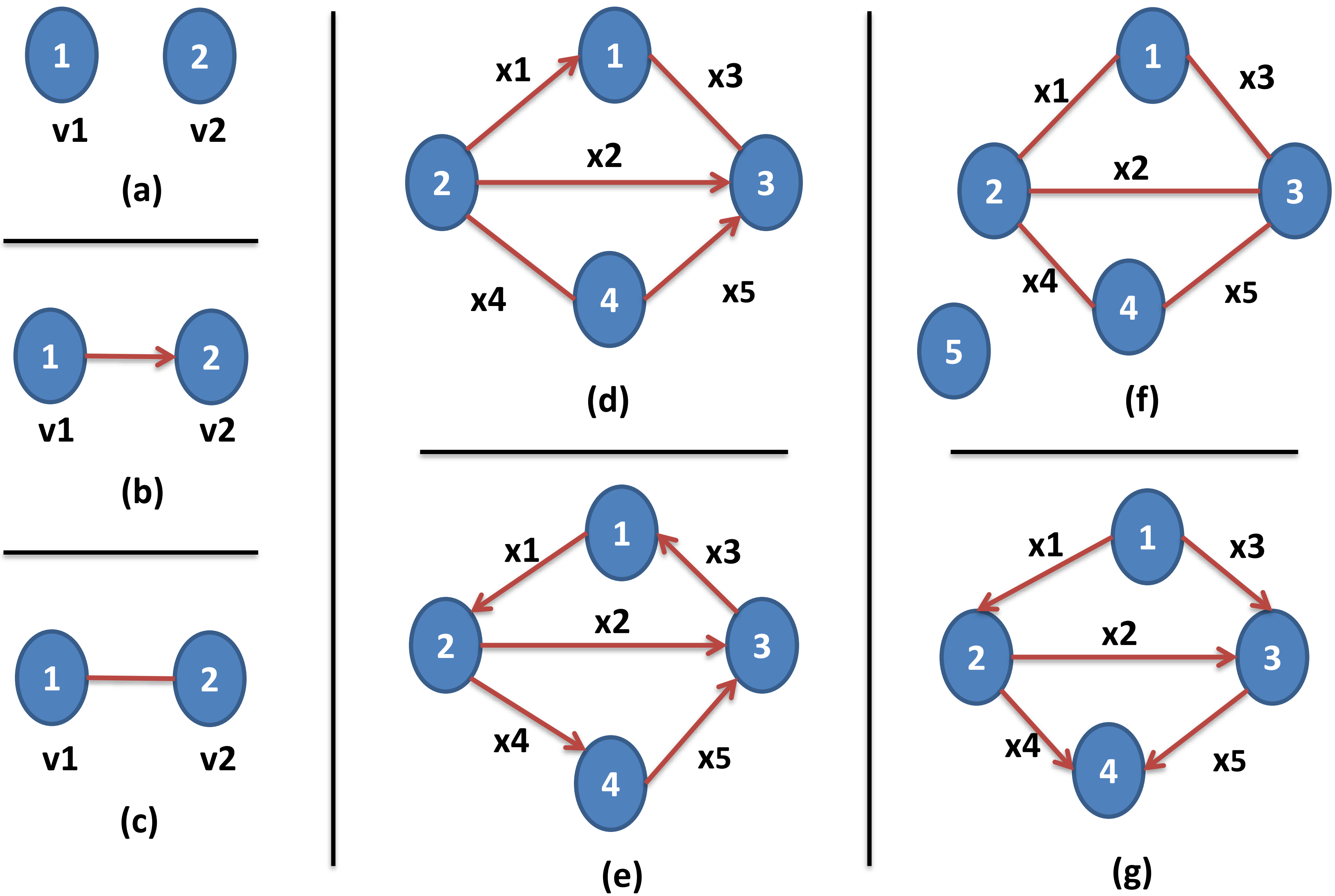
Graph

Basic Notations of Graph Theory



Basic Notations of Graph Theory

- Consider diagrams shown in above figure
- Every diagrams represent Graphs
- Every diagram consists of a **set of points** which are shown by **dots** or **circles** and are sometimes labelled V_1, V_2, V_3… OR 1,2,3…
- In every diagrams, certain pairs of such points are connected by lines or arcs
- Note that every arc start at one point and ends at another point

Basic Notations of Graph Theory

Graph

- **A graph G** consist of a **non-empty set V** called the **set of nodes** (points, vertices) of the graph, a **set E** which is the **set of edges** and a **mapping** from the set of edges **E to** a set of **pairs of elements of V**
- It is also convenient to write a graph as **G=(V,E)**
- Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X Є E is thus associated with a pair of nodes (u,v) where u, v Є V then we says that edge x connect u and v

Adjacent Nodes

- Any two nodes which are connected by an edge in a graph are called adjacent nodes

Graph – Concepts & Definitions

- **Directed & Undirected Edge**
  - In a graph **G=(V,E)** an **edge** which is **directed** from one end to another end is called a **directed edge**, while the edge which has no specific direction is called **undirected edge**
- **Directed graph (Digraph)**
  - A graph in which **every edge is directed** is called directed graph or digraph e.g. **b,e & g** are directed graphs
- **Undirected graph**
  - A graph in which **every edge is undirected** is called undirected graph e.g. **c & f** are undirected graphs
- **Mixed Graph**
  - If **some** of the **edges** are **directed** and **some are undirected** in graph then the graph is called mixed graph e.g. **d** is mixed graph

Graph – Concepts & Definitions

Loop (Sling)

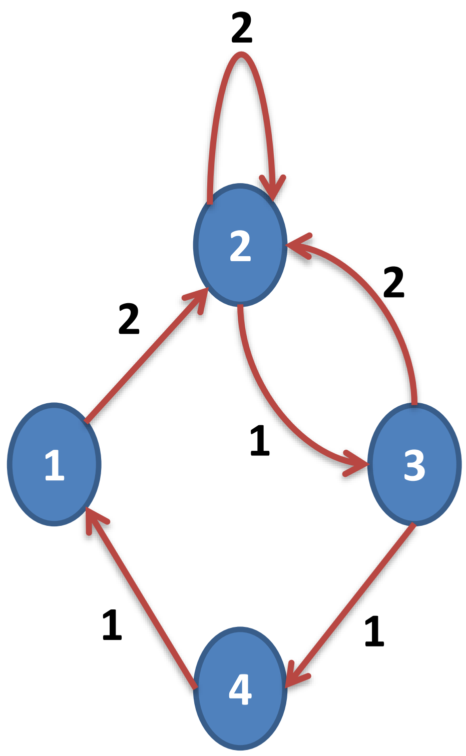
- An **edge** of a graph **which joins a node to itself** is called a loop (sling).
- The ***direction of a loop is of no significance*** so it can be considered either a directed or an undirected.

Distinct Edges

- In case of directed edges, **two possible edges** between any pair of nodes which **are opposite in direction** are considered **Distinct**.

Parallel Edges

- In some directed as well as undirected graphs, we may have **certain pairs of nodes joined by more than one edges**, such edges are called **Parallel** edges.

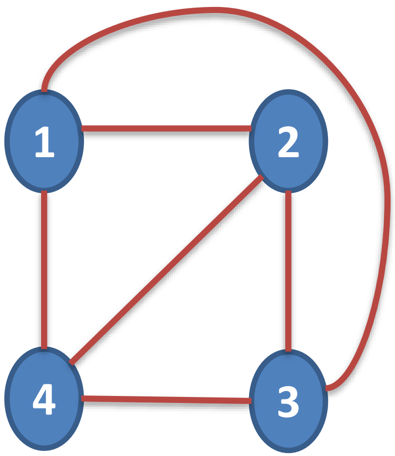


Graph – Concepts & Definitions

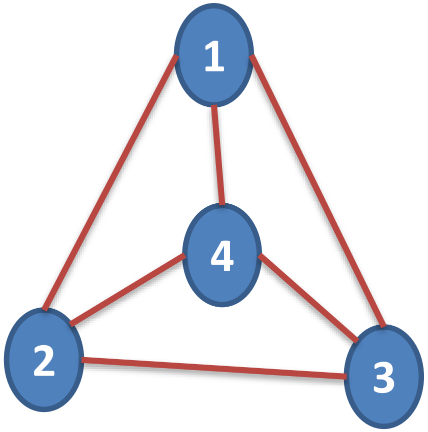
- **Multigraph**
  - Any **graph** which **contains** some **parallel edges** is called **multigraph**
  - If there is no more then one edge between a pair of nodes then such a graph is called **Simple graph**
- **Weighted Graph**
  - A graph in which **weights are assigned to every edge** is called weighted graph
- **Isolated Node**
  - In a graph a **node** which is **not adjacent to any other node** is called isolated node
- **Null Graph**
  - A graph **containing only isolated nodes** are called null graph. In other words set of edges in null graph is empty

Graph – Concepts & Definitions

- For a given **graph** there is **no unique diagram** which represents the graph.
- We can obtain a variety of diagrams by locating the nodes in an arbitrary numbers.
- Following both diagrams represents same Graph.

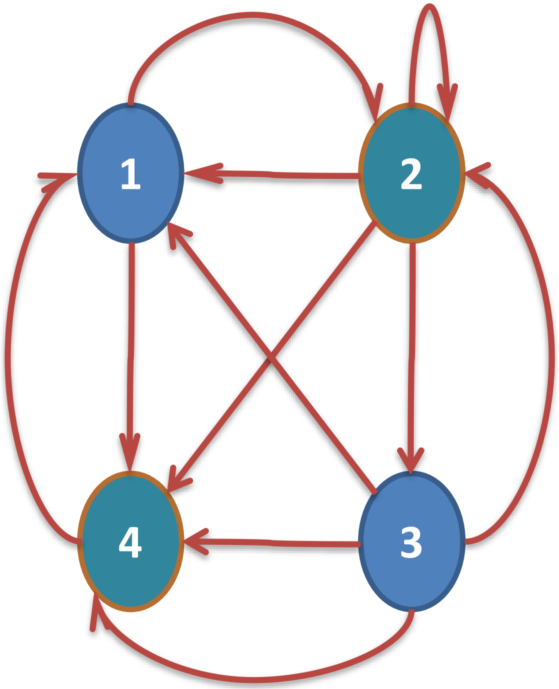


- **Indegree of Node**
  - The **no of edges** which have **V as their terminal node** is call as indegree of node V.
- **Outdegree of Node**
  - The **no of edges** which have **V as their initial node** is call as outdegree of node V.



- **Total degree of Node**
  - Sum of indegree and outdegree of node V is called its Total Degree or Degree of vertex.

Path of the Graph



Some of the path from 2 to 4 **P1 =** ((2,4))

P2 = ((2,3), (3,4))

P3 = ((2,1), (1,4))

P4 = ((2,3), (3,1), (1,4))

P5 = ((2,3), (3,2), (2,4))

P6 = ((2,2), (2,4))

- Let G=(V, E) be a simple digraph such that the terminal node of any edge in the sequence is the initial node of the edge, if any appearing next in the sequence defined as **path of the graph**.
- **Length of Path**
  - The number of edges appearing in the sequence of the path is called length of path.

Graph – Concepts & Definitions

- **Simple Path (Edge Simple)**
  - A **path** in a diagraph in which **the edges are distinct** is called simple path or edge simple
  - Path P5, P6 are Simple Paths
- **Elementary Path (Node Simple)**
  - A **path** in which **all the nodes through which it traverses** are **distinct** is called elementary path
  - Path P1, P2, P3 & P4 are elementary Path
  - Path P5, P6 are Simple but not Elementary
- **Cycle (Circuit)**
  - A **path** which **originates and ends in the same node** is called cycle (circuit)
  - E.g. C1 = ((2,2)), C2 = ((1,2),(2,1)), C3 = ((2,3), (3,1), (1,2)
- **Acyclic Diagraph**
  - A simple **diagraph which does not have any cycle** is called Acyclic Diagraph.

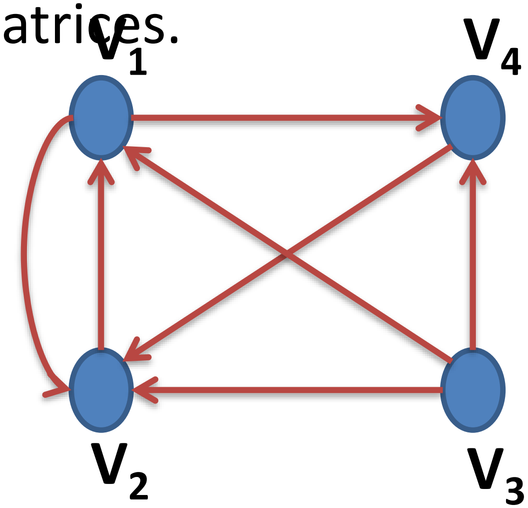
Adjacency matrix

- A **diagrammatic representation** of a **graph** may have limited usefulness. However such a representation **is not feasible** when number of **nodes** an **edges** in a graph **is large**
- It is easy to store and manipulate matrices and hence the graphs represented by them in the computer
- Let **G = (V, E)** be a simple **diagraph** in which **$V = \{v_1, v_2, \ldots, v_n\}$** and the **nodes** are assumed to be **ordered** from v_1 to v_n
- An n x n matrix **A** is called **Adjacency matrix** of the graph G whose **elements** are a_{ij} are given by

$$a_{ij} = \begin{cases} 1 & if (V_i, V_j) \in E \\ 0 & otherwise \end{cases}$$

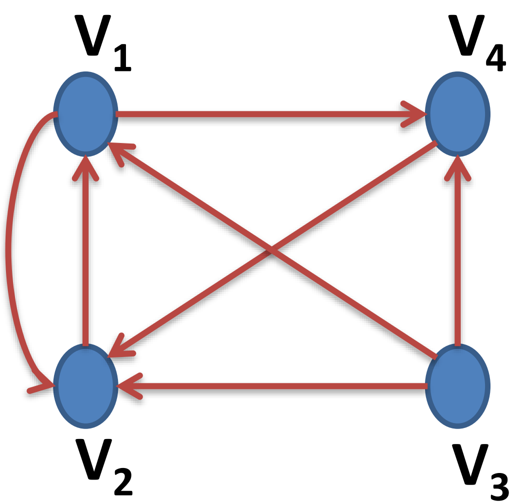
Adjacency matrix

- An **element** of the adjacency matrix is either **0** or **1**
- Any **matrix** whose **elements are either 0 or 1** is called **bit matrix** or **Boolean matrix**
- For a given graph G =m (V, E), an **adjacency matrix** depends upon the ordering of the elements of V
- For different ordering of the elements of V we get different adjacency matrices.



$$A = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Adjacency matrix



$$A = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

- The **number of elements** in the **i^{th} row** whose **value is 1** is equal to the **out-degree** of node **V_i**
- The **number of elements** in the **j^{th} column** whose **value is 1** is equal to the **in-degree** of node **V_j**
- For a **NULL graph** which consist of only n nodes but no edges, the **adjacency matrix** has **all its elements 0**. i.e. the adjacency matrix is the NULL matrix

Power of Adjacency matrix

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

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

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

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

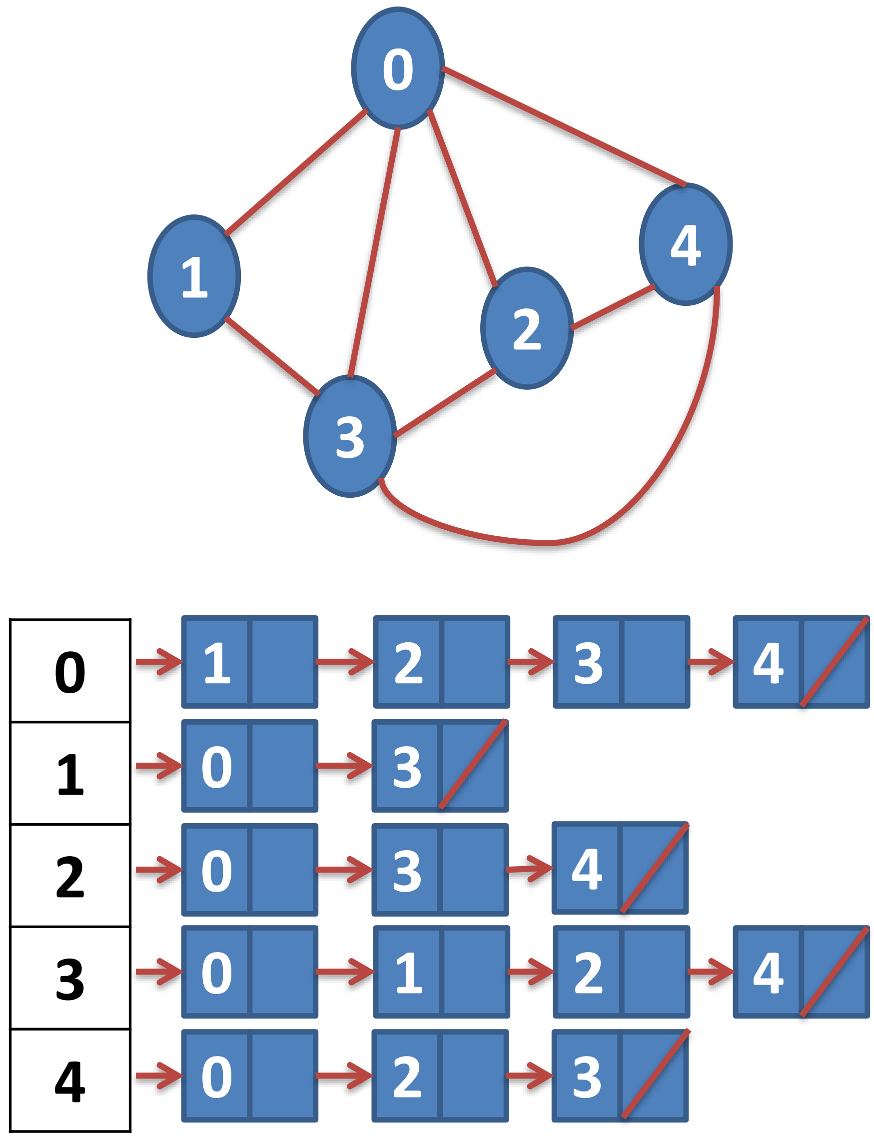
- Entry of **1** in **i^{th}** row and **j^{th}** column of **A** shows existence of an **edge (V_i, V_j)**, that is a **path of length 1**
- Entry in **A^2** shows **no of different paths** of **exactly length 2** from node **V_i** to **V_j**
- Entry in **A^3** shows **no of different paths** of **exactly length 3** from node **V_i** to **V_j**

Path matrix or reachability matrix

- Let **G = (V,E)** be a simple diagraph which contains **n nodes**that are assumed to be ordered.
- A **n x n** matrix **P** is called **path matrix**whose elements are given by

$$P_{ij} = \begin{cases} 1, if\ there\ exists\ path\ from\ node\ V_i\ to\ V_j \\ 0, otherwise \end{cases}$$

Adjacency List Representation

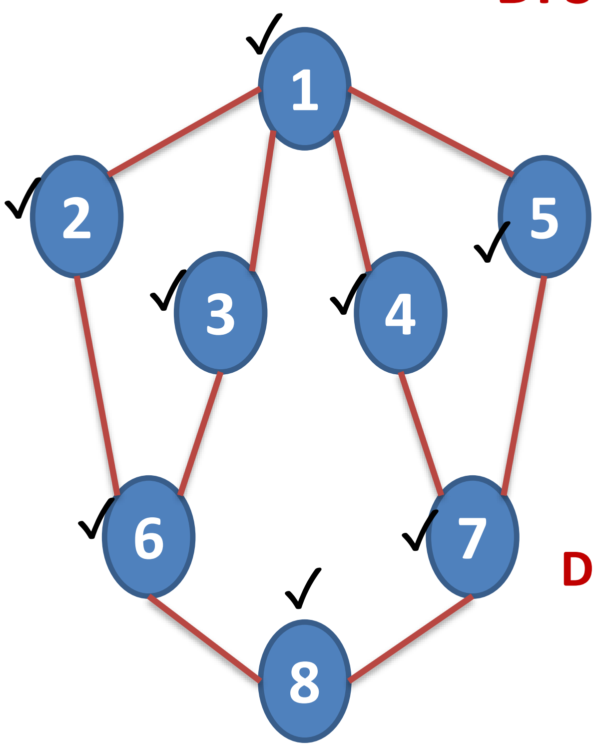


Graph Traversal

- Two Commonly used Traversal Techniques are
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

Depth First Search (DFS)

- It is like preorder traversal of tree
- Traversal can start from any vertex V_i
- V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS



DFS (G, 1) is given by

Step 1: Visit (1)

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

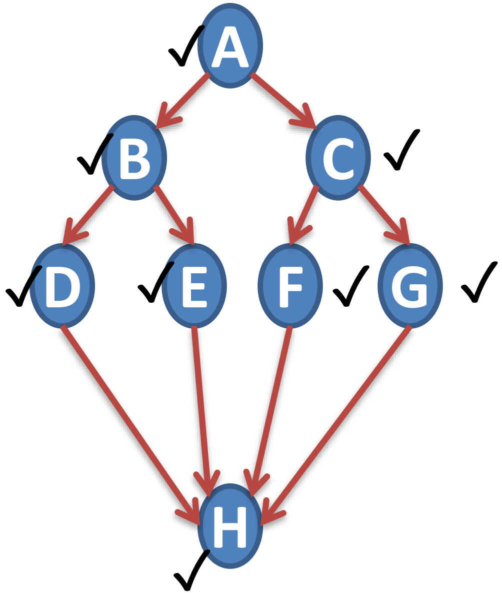
DFS (G, 2):
Step1: Visit(2)
Step 2: DFS (G, 6)

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

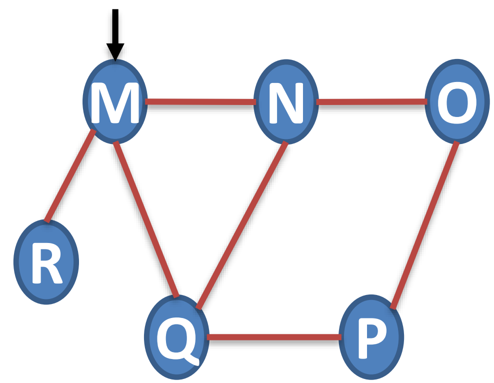
DFS of given graph starting **from** node **1** is given by

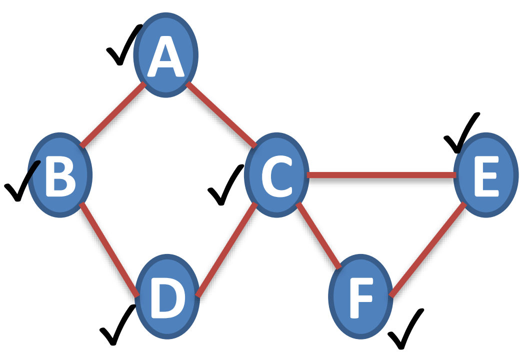
1 2 6 3 8 7 4 5

Depth First Search (DFS)



ABDHECFG



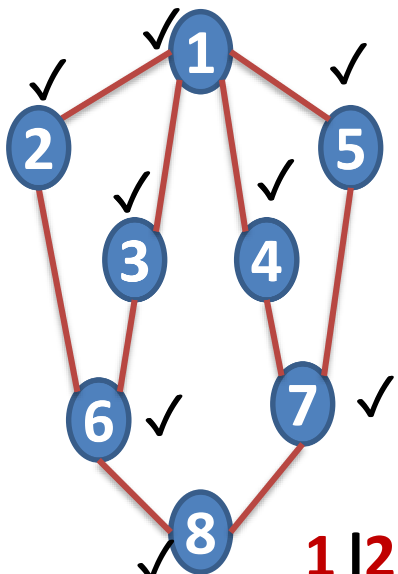


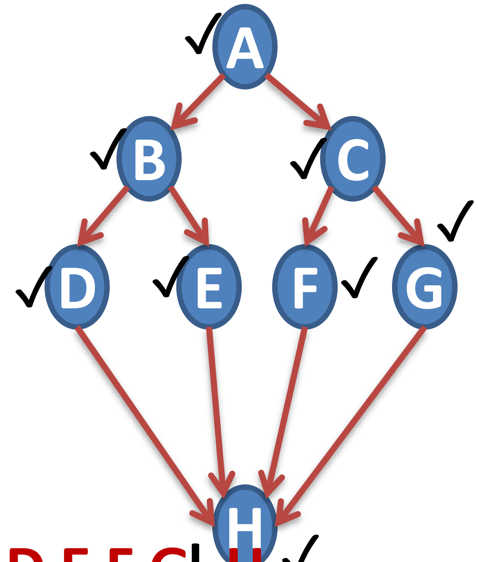
A B D C F E

Breadth First Search (BFS)

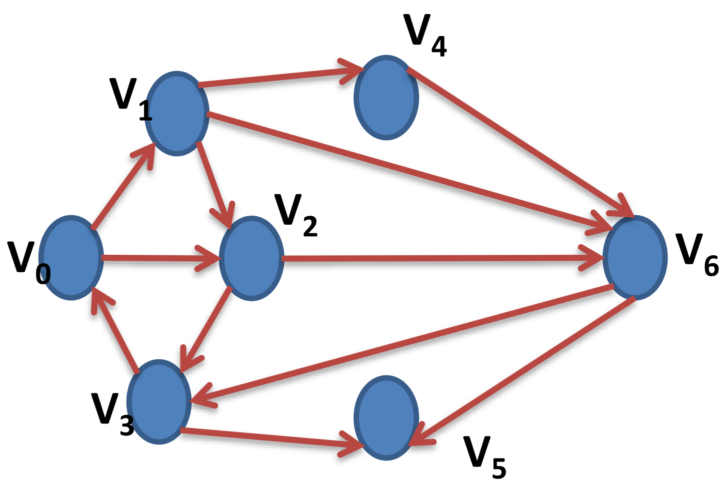
- This methods **starts** from vertex $\mathbf{V_0}$
- $\mathbf{V_0}$ is marked as **visited**. All **vertices adjacent to** $\mathbf{V_0}$ are **visited next**
- Let vertices adjacent to V_0 are V_1, V_2, V_2, V_4
- V_1, V_2, V_3 and V_4 are marked visited
- All unvisited vertices adjacent to V_1, V_2, 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)



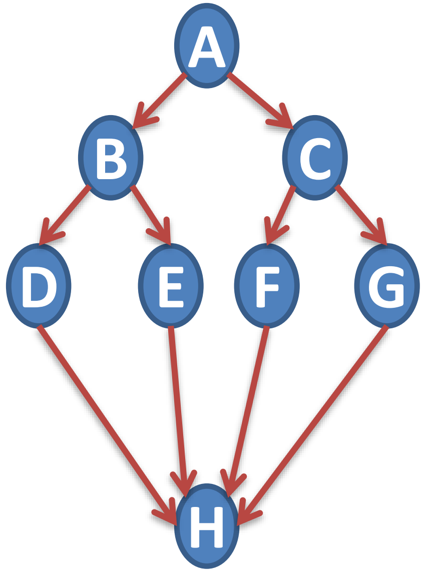


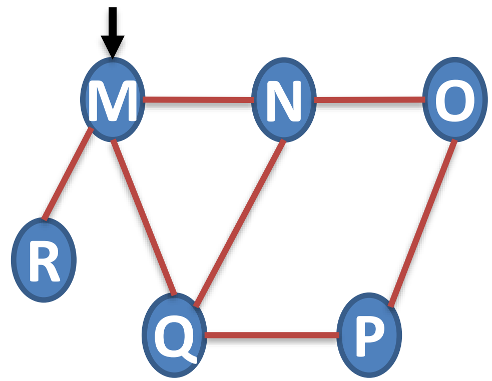
1 | 2 3 4 5 | 6 7 | 8 A | B C | D E F G | H

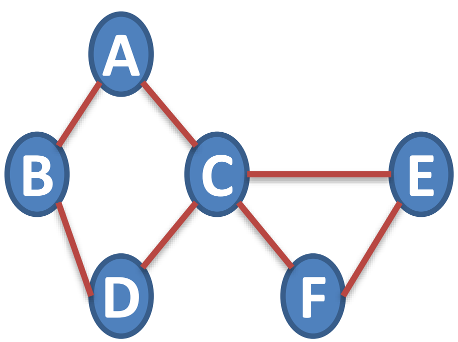


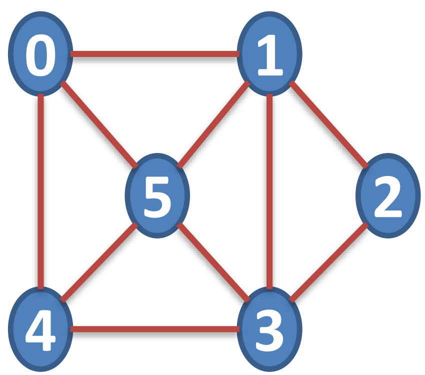
V_0 | V_1 V_2 | V_4 V_6 V_3 | V_5

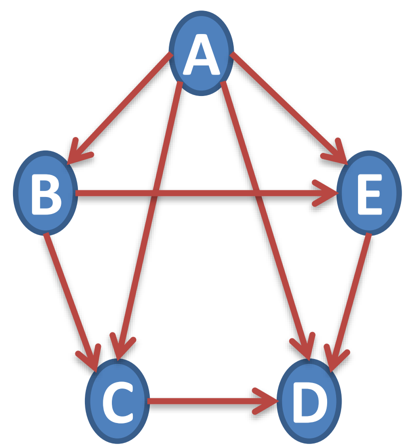
Write DFS & BFS of following Graphs









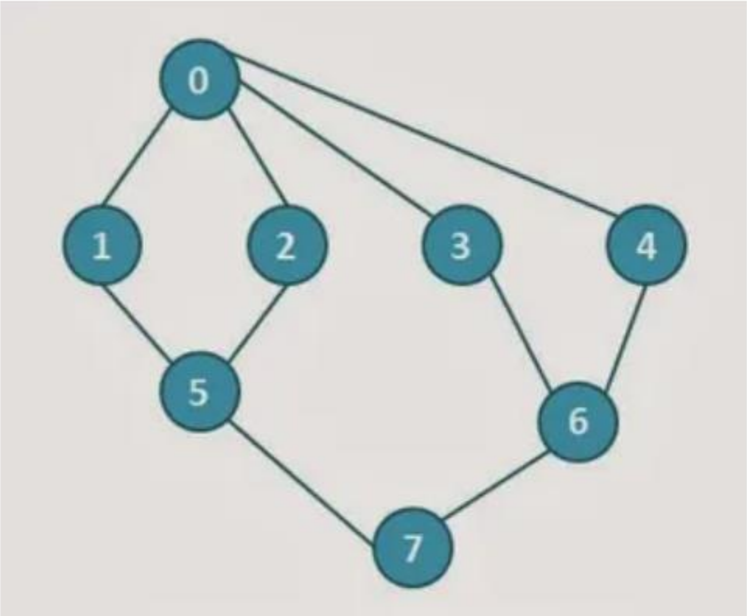


Procedure : DFS (vertex V)

- This procedure **traverse the graph G in DFS** manner.
- V is a starting vertex to be explored.
- Visited[] is an array which tells you whether particular vertex is visited or not.
- W is a adjacent node of vertex V.
- S is a Stack, PUSH and POP are functions to insert and remove from stack respectively.

Procedure : DFS (vertex V)

```
1. [Initialize TOP and Visited]
   visited[] ← 0
   TOP ← 0
2. [Push vertex into stack]
   PUSH (V)
3. [Repeat  while stack is not Empty]
   Repeat Step 3 while stack is not empty
       v  ← POP()
       if   visited[v] is 0
       then visited [v] ← 1
            for all W adjacent to v
               if   visited [w] is 0
              then PUSH (W)
            end for
       end if
```

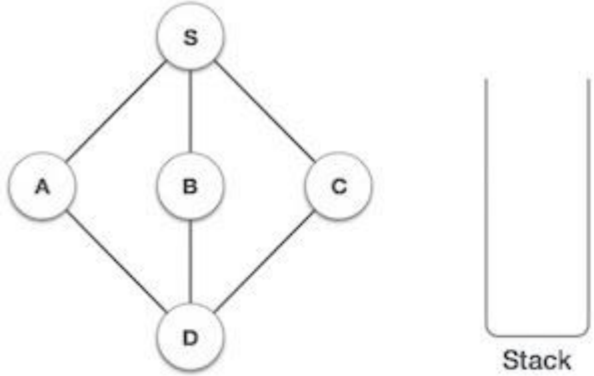


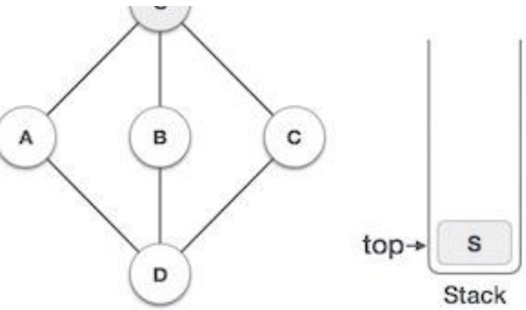
	0	1	2	3	4	5	6	7
0	0	1	1	1	1	0	0	0
1	1	0	0	0	0	1	0	0
2	1	0	0	0	0	1	0	0
3	1	0	0	0	0	0	1	0
4	1	0	0	0	0	0	1	0
5	0	1	1	0	0	0	0	1
6	0	0	0	1	1	0	0	1
7	0	0	0	0	0	1	1	0

DFS

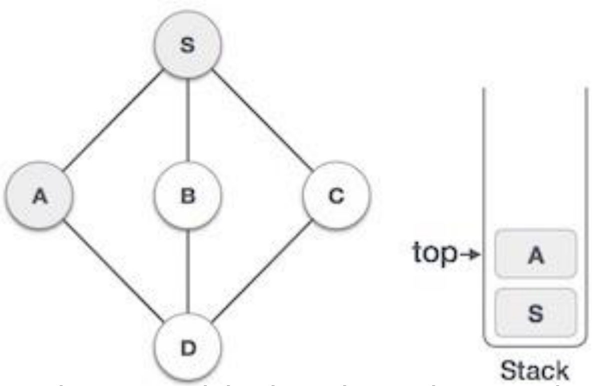
- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

DFS

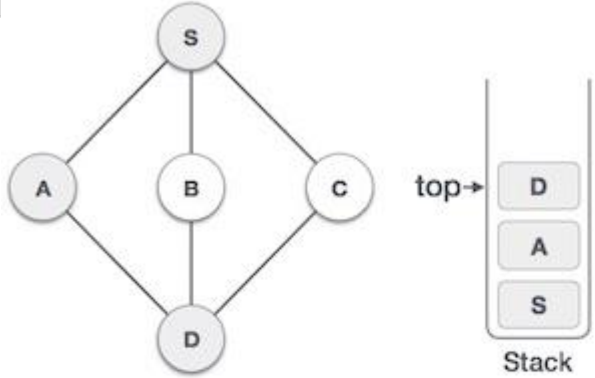




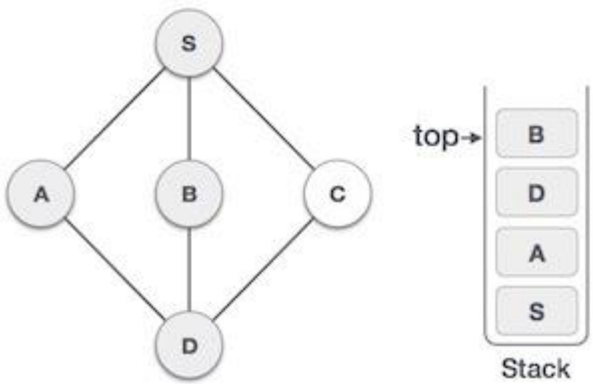
Explore any unvisited adjacent node from **S**



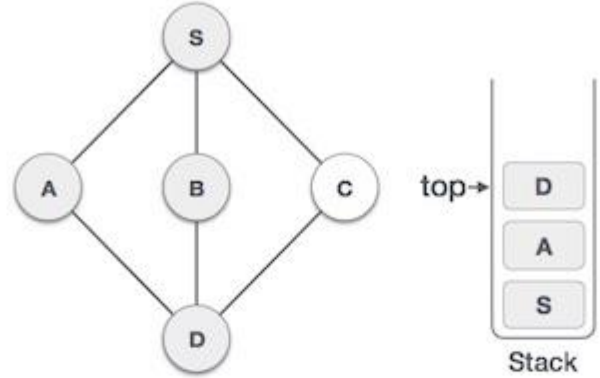
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.



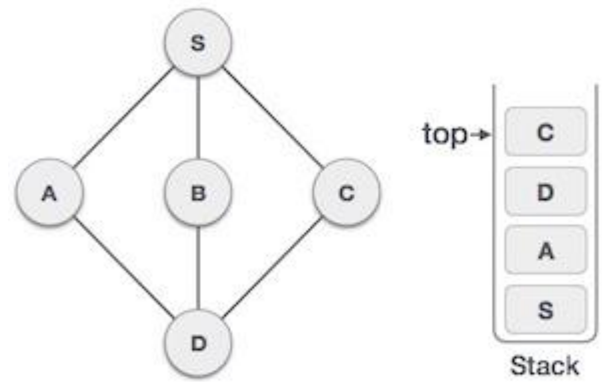
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.



We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

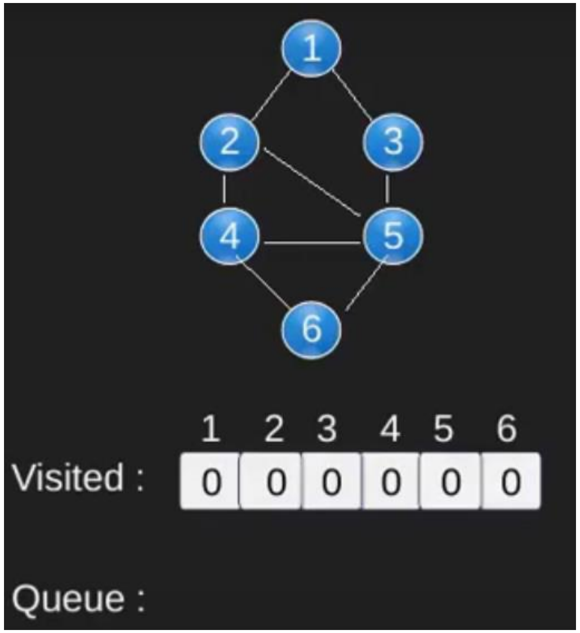
Procedure : BFS (vertex V)

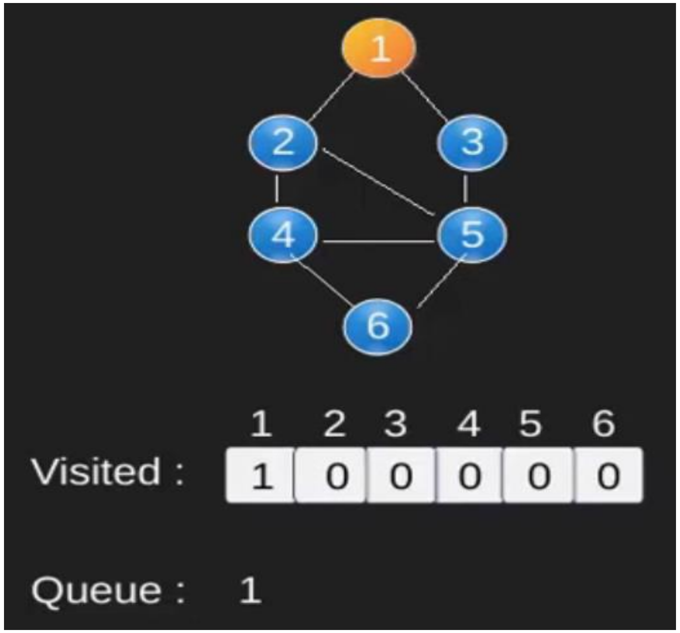
- This procedure **traverse the graph G in BFS** manner
- **V** is a **starting vertex** to be explored
- Q is a queue
- visited[] is an array which tells you whether particular vertex is visited or not
- W is a adjacent node f vertex V.

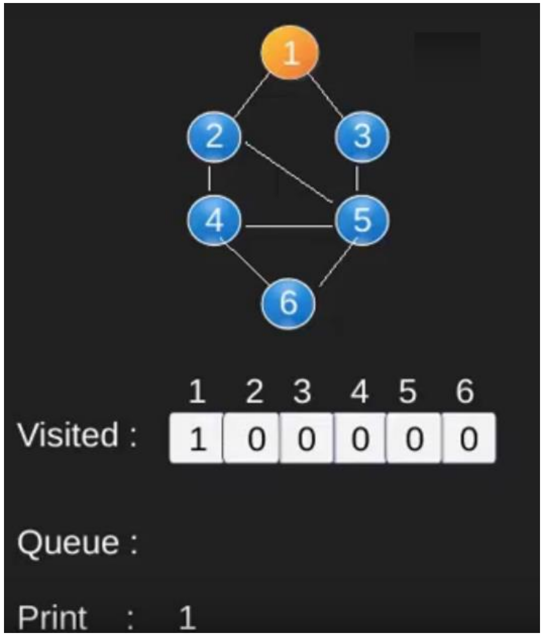
Procedure : BFS (vertex V)

```
1. [Initialize Queue & Visited]
   visited[] ← 0
   F ← R ← 0
2. [Marks visited of V as 1]
   visited[v] ← 1
3. [Add vertex v to Q]
   InsertQueue(V)
4. [Repeat while Q is not Empty]
   Repeat while Q is not empty
     v ← RemoveFromQueue()
     For all vertices W adjacent to v
       If   visited[w] is 0
      Then visited[w] ← 1
            InsertQueue(w)
```

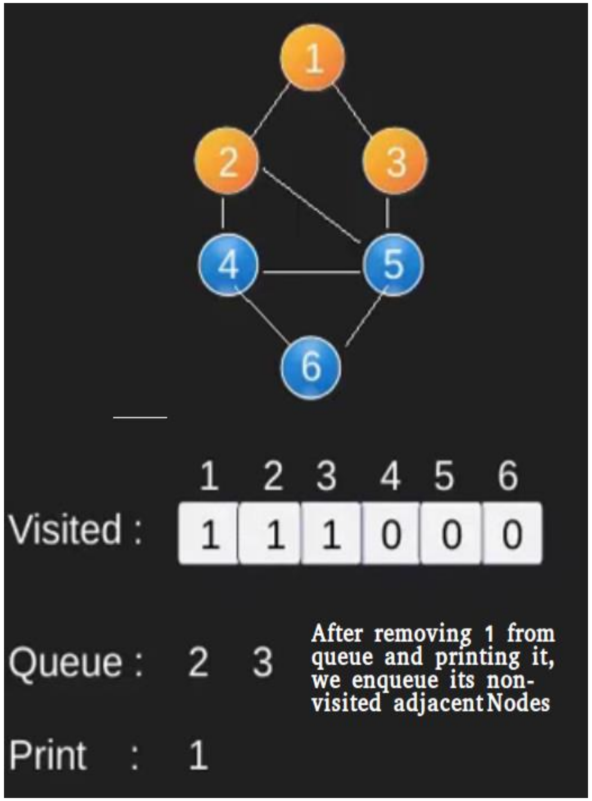
BFS

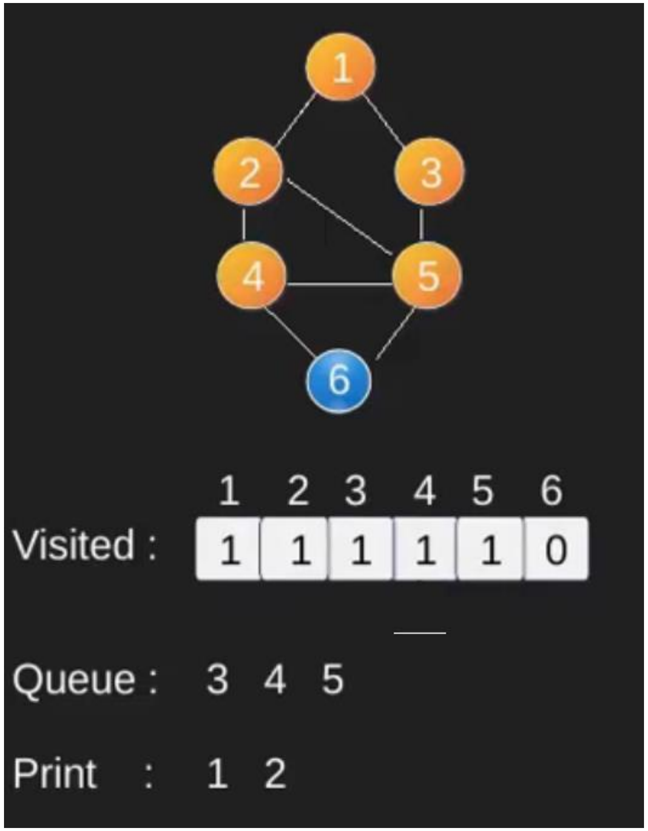

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 0 0 0 0 0 0
Queue :

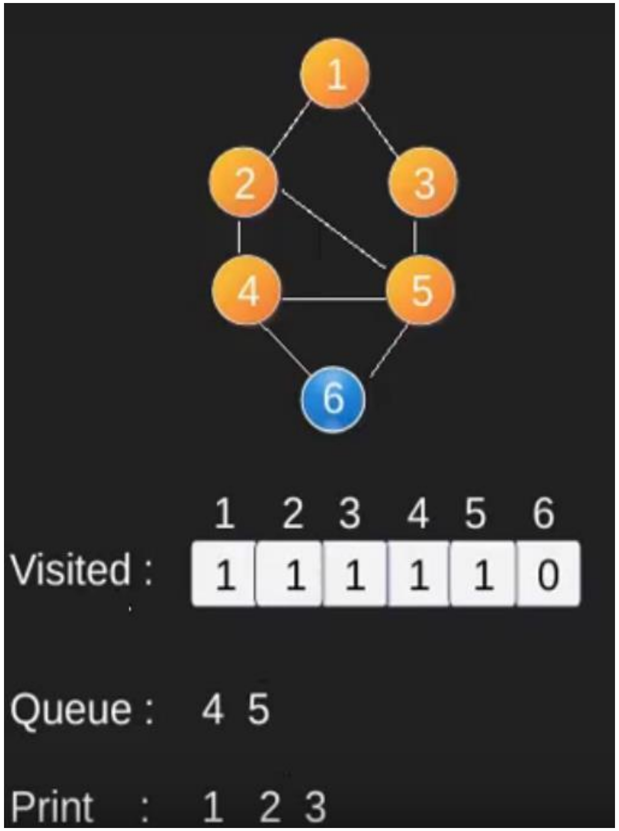

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue : 1


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue :
Print : 1

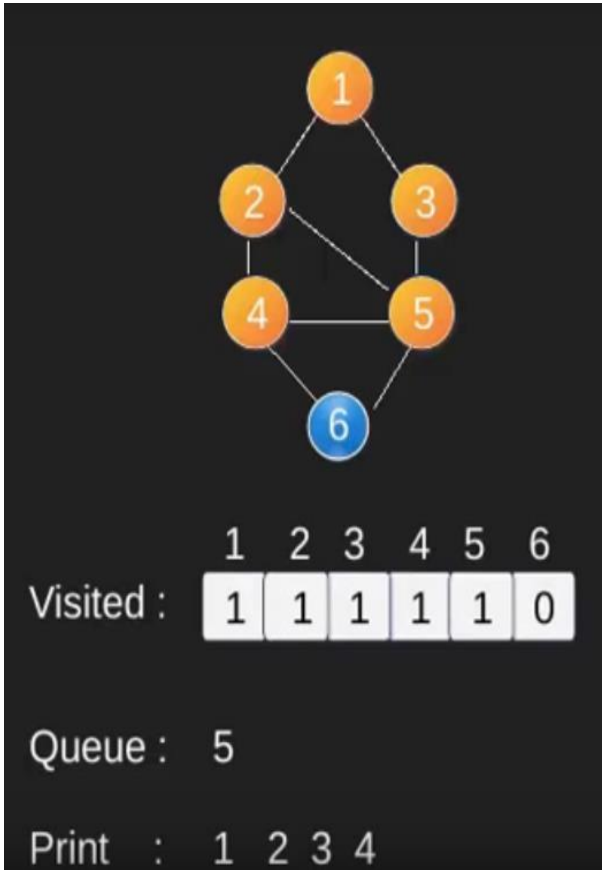
BFS

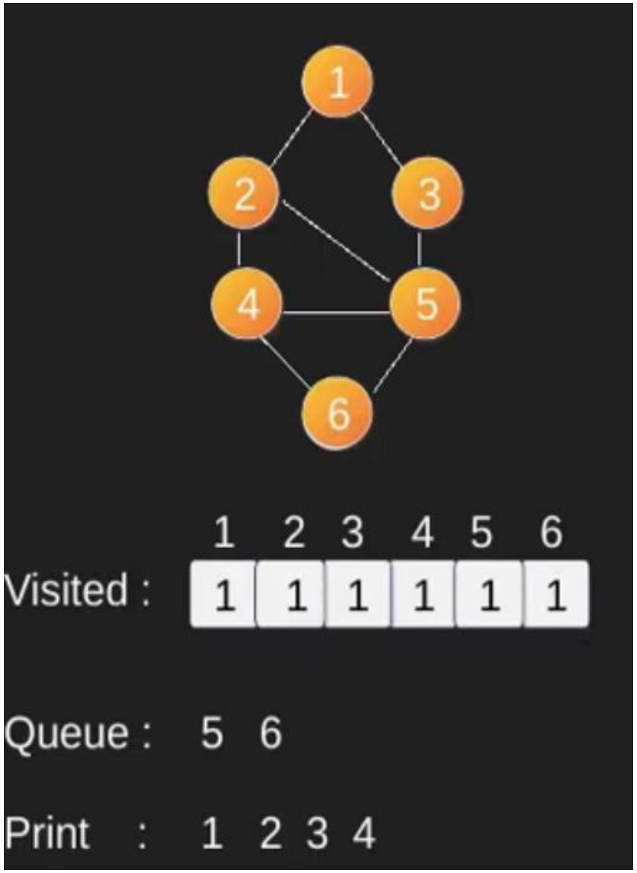

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 0 0 0
Queue : 2 3
After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes
Print : 1

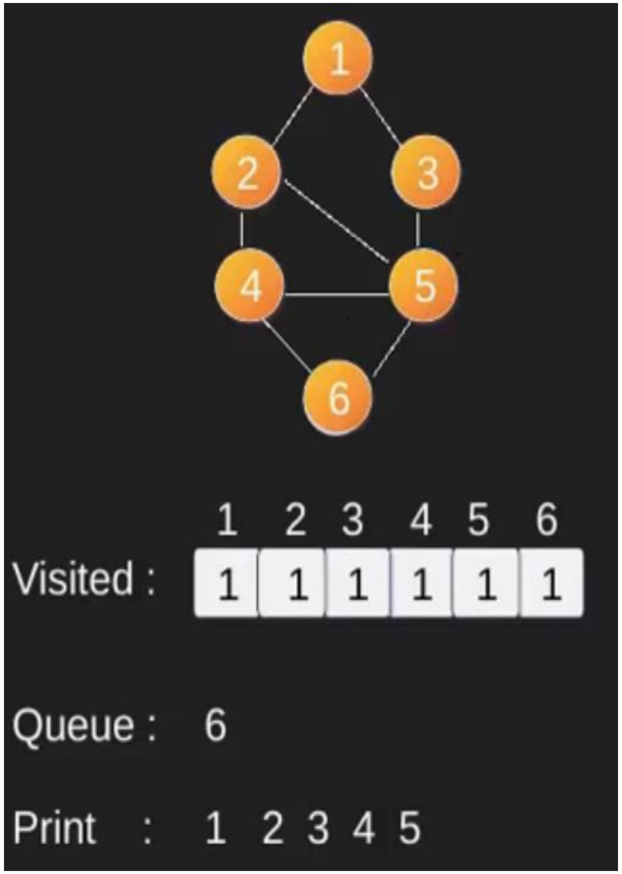

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 3 4 5
Print : 1 2


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 4 5
Print : 1 2 3

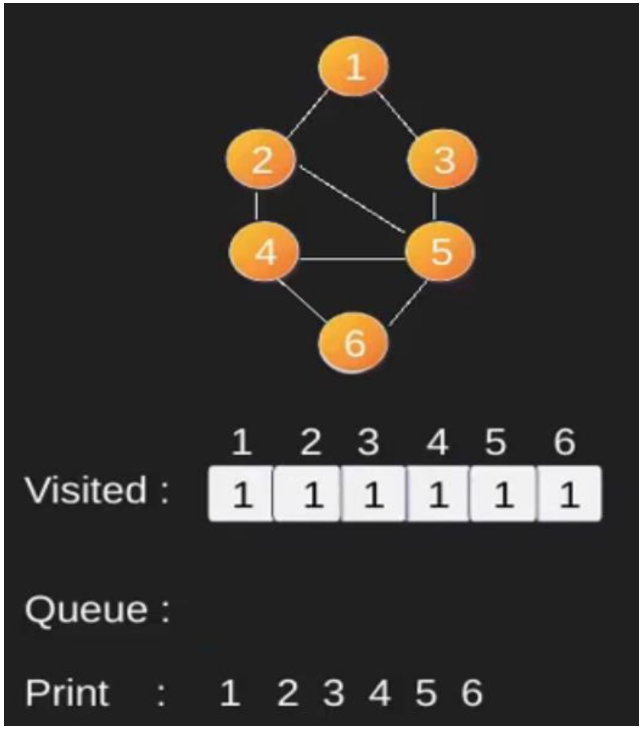
BFS


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 5
Print : 1 2 3 4


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 1
Queue : 5 6
Print : 1 2 3 4


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 1
Queue : 6
Print : 1 2 3 4 5

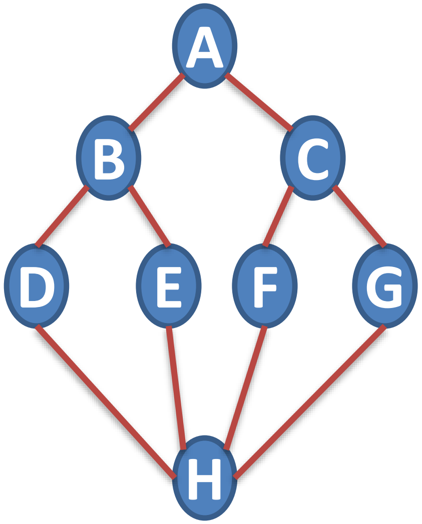
BFS

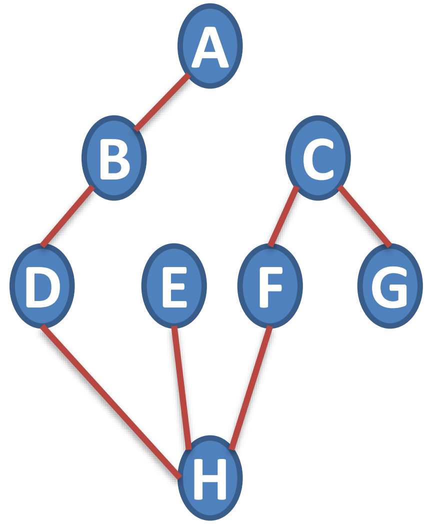


Spanning Tree

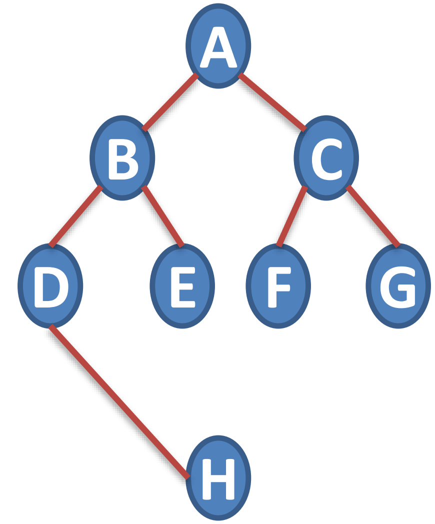
- A **Spanning tree** of a graph is an undirected tree **consisting of only those edges necessary to connect all the nodes** in the original graph
- A spanning tree has the **properties** that
  - For any **pair** of nodes there exists **only one path between them**
  - **Insertion** of any **edge** to a spanning tree **forms a unique cycle**
- The particular **Spanning for a graph** depends on the **criteria** used to **generate** it
- If **DFS search** is use, those edges traversed by the algorithm forms the edges of tree, referred to as **Depth First Spanning Tree**
- If **BFS Search** is used, the spanning tree is formed from those edges traversed during the search, producing **Breadth First Spanning tree**

Construct Spanning Tree

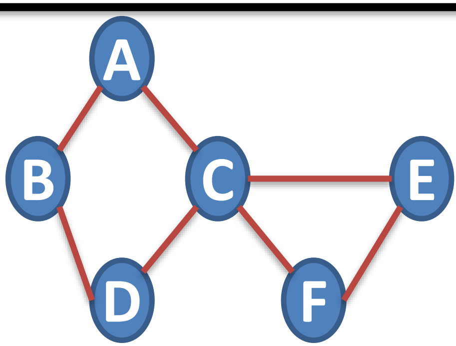


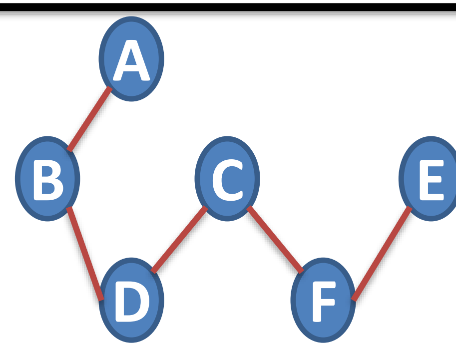


DFS Spanning Tree

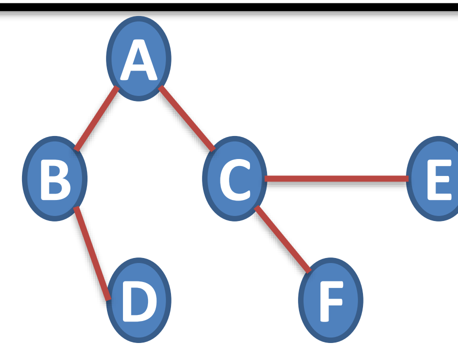


BFS Spanning Tree





DFS Spanning Tree

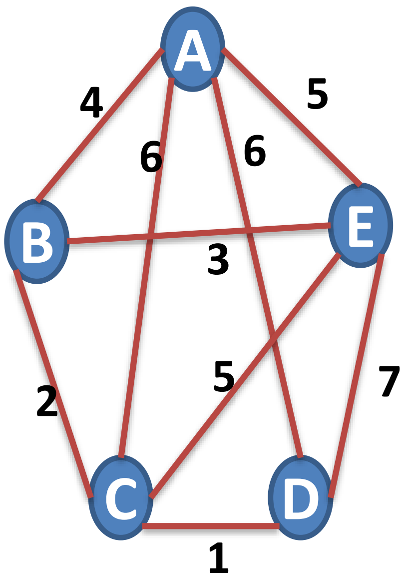


BFS Spanning Tree

Minimum Cost Spanning Tree

- The **cost of a spanning tree** of a weighted undirected graph is the sum of the costs(weights) of the edges in the spanning tree
- A **minimum cost spanning tree** is a spanning tree of least cost
- Two techniques for Constructing minimum cost spanning tree
  - Prim’s Algorithm
  - Kruskal’s Algorithm

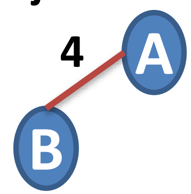
Prims Algorithm



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

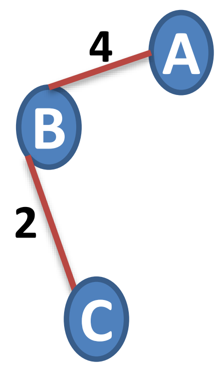
Let X be the set of nodes explored, initially X = { A }

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



X = { A , B }

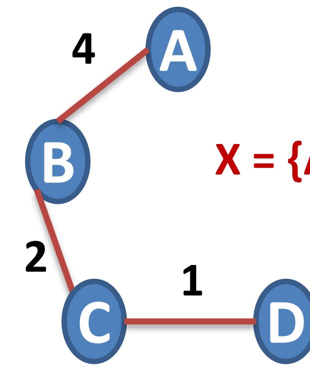
Step 2: Taking minimum weight edge of all Adjacent edges of X = { A , B }



X = {A ,B,C}

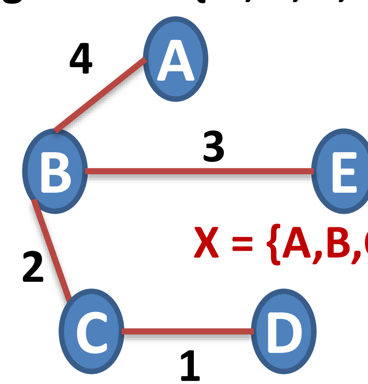
We obtained minimum spanning tree of cost:
4 + 2 + 1 + 3 = 10

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



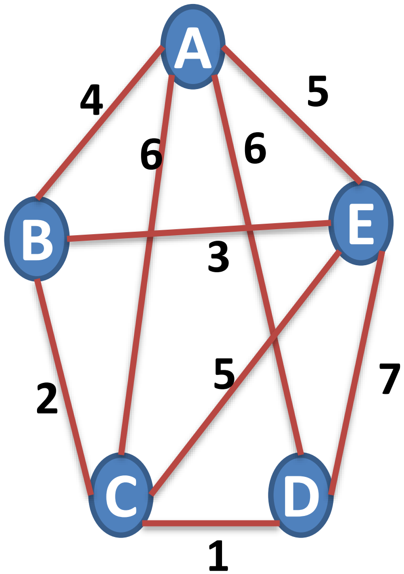
X = {A,B,C,D}

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

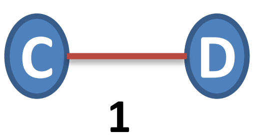


X = {A,B,C,D,E}

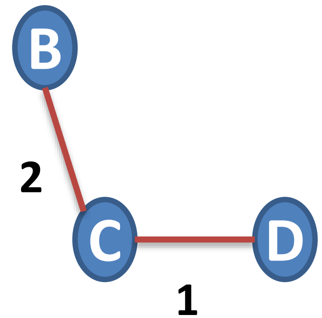
Kruskal's Algorithm



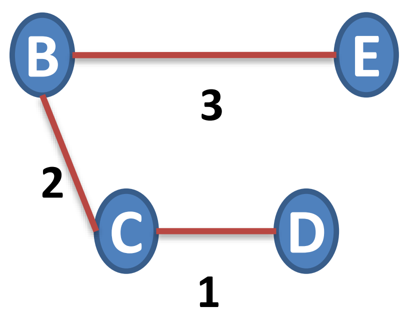
Step 1: Taking min edge (C,D)



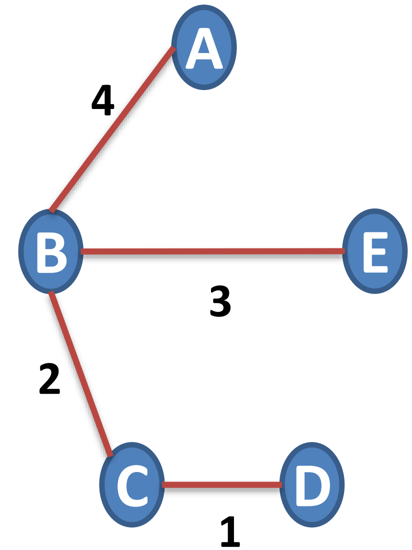
Step 2: Taking next min edge (B,C)



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



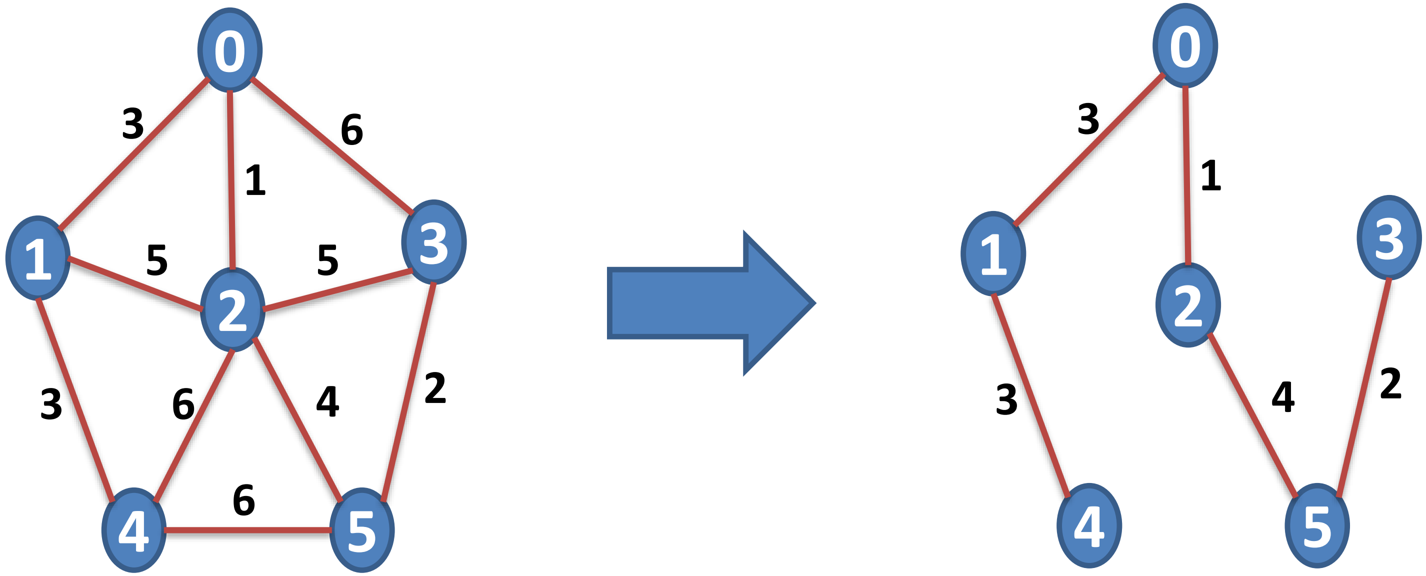
Step 4: Taking next min edge (A,B)

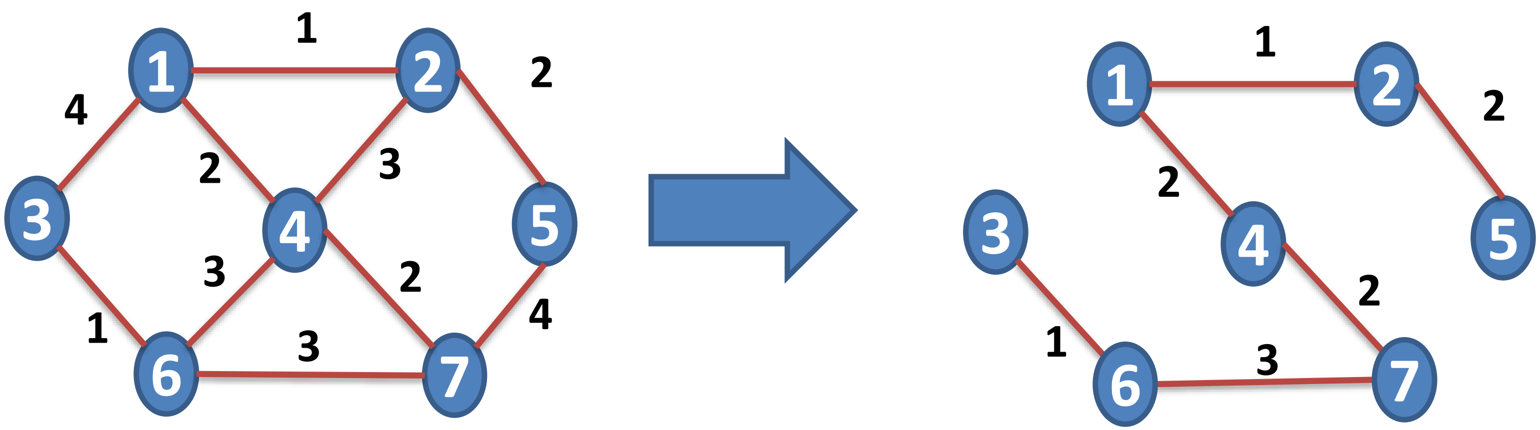


so we obtained minimum spanning tree of cost:

$4 + 2 + 1 + 3 = 10$

Construct Minimum Spanning Tree

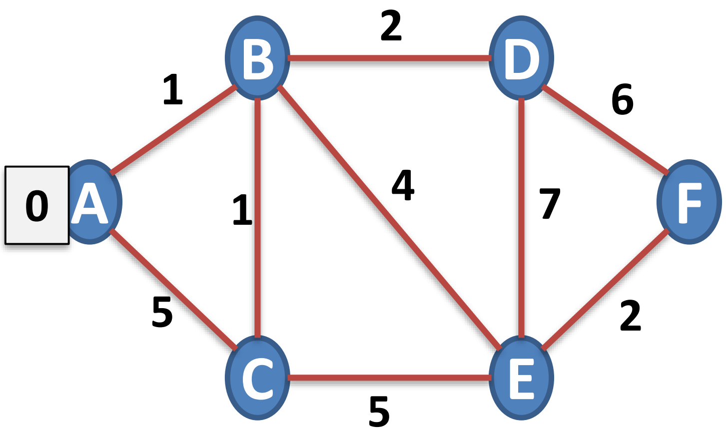




Shortest Path Algorithm

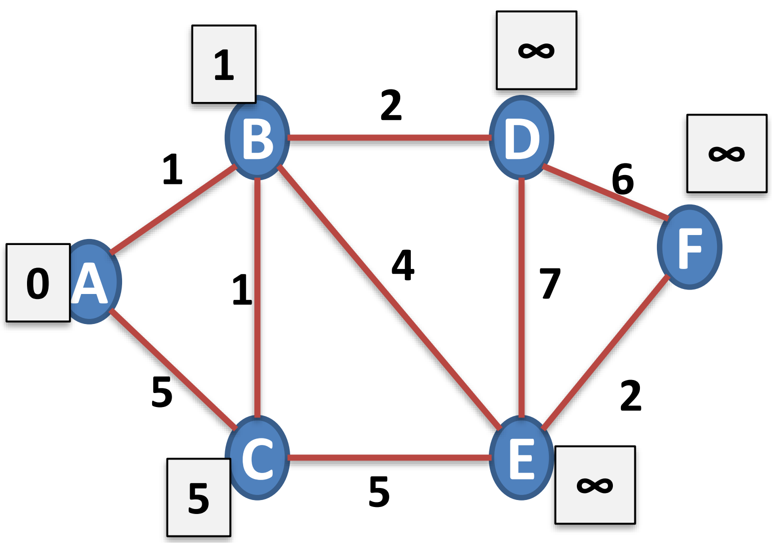
- Let **G = (V,E)** be a simple diagraph with **n vertices**
- The problem is to **find out shortest distance** from a **vertex to all other vertices** of a graph
- **Dijkstra Algorithm** – it is also called Single Source Shortest Path Algorithm

Dijkstra Algorithm – Shortest Path



	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 Algorithm – Shortest Path

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

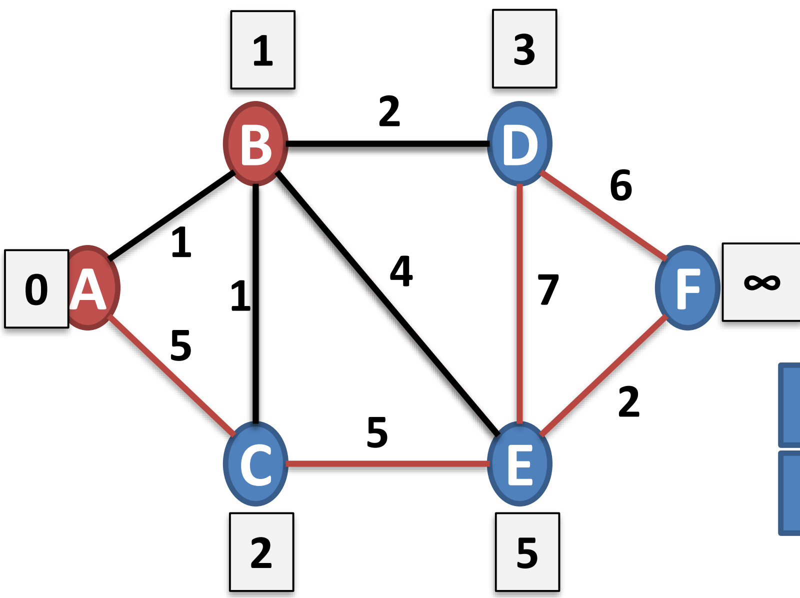
Cost of going to C via B = dist[B] + cost[B][C] = 1 + 1 = 2

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 Algorithm – Shortest Path

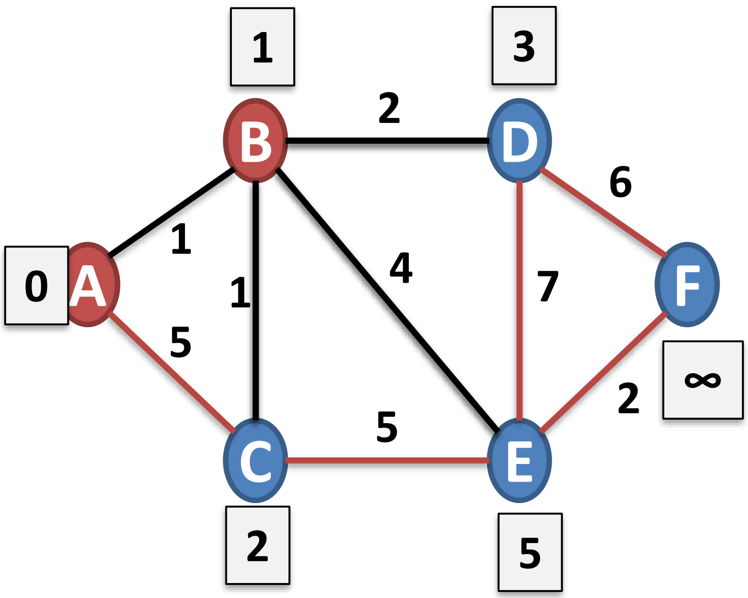
3rd Iteration: Select **Vertex C** via B with minimum distance

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

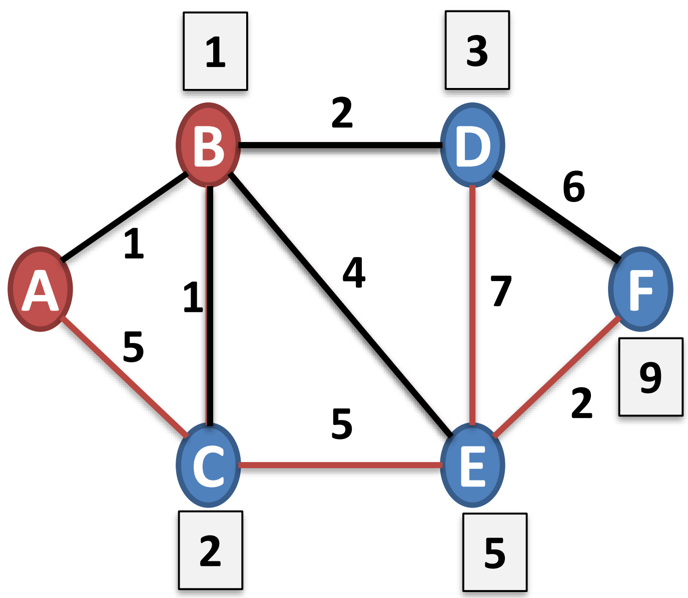
Dijkstra Algorithm – Shortest Path

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



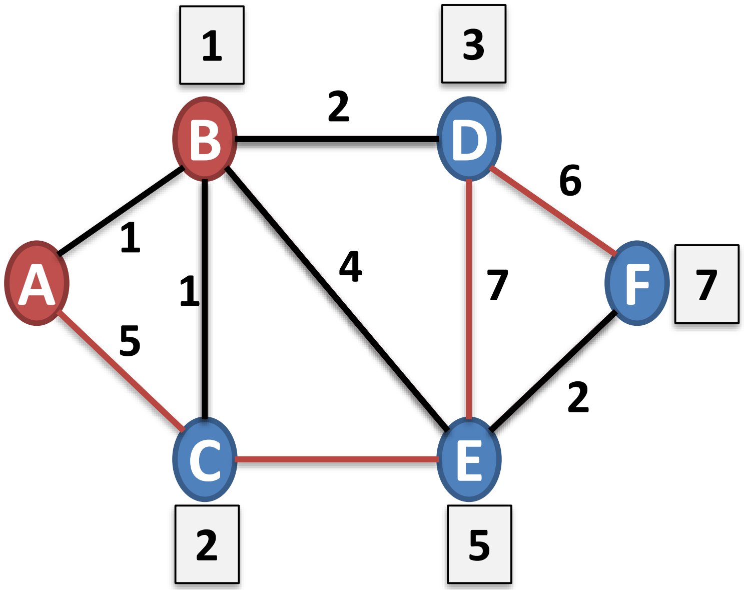
	A	B	C	D	E	F
Distance	0	1	2	3	5	9
Visited	1	1	1	1	0	0

Dijkstra Algorithm – Shortest Path

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