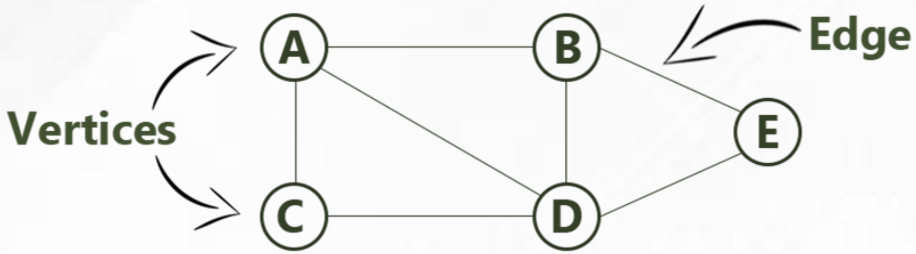
Non-Linear Data Structure
Graph



Asst. Prof. Kumar Prasun
Computer Application Department
Padmakanya Multiple Campus, Baghbazar

Kumar.prasun@pkmc.tu.edu.np
+977 9851149487

Graphs

- What is Graph?
- Representation of Graph
  - Matrix representation of Graph
  - Linked List representation of Graph
- Elementary Graph Operations
  - Breadth First Search (BFS)
  - Depth First Search (DFS)
  - Spanning Trees
  - Minimal Spanning Trees
  - Shortest Path

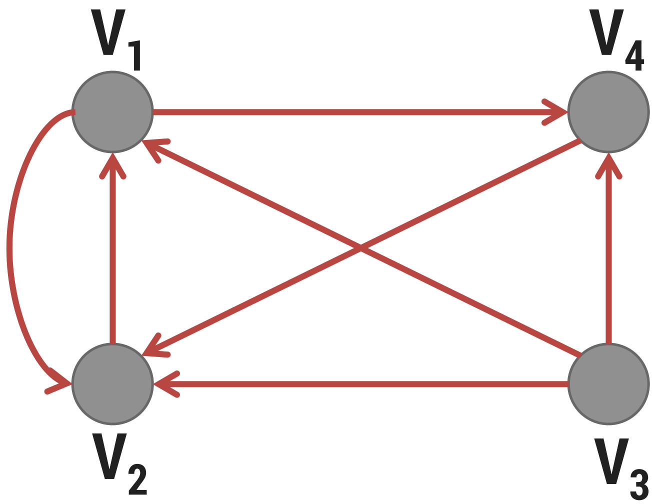
Adjacency matrix

- 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

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

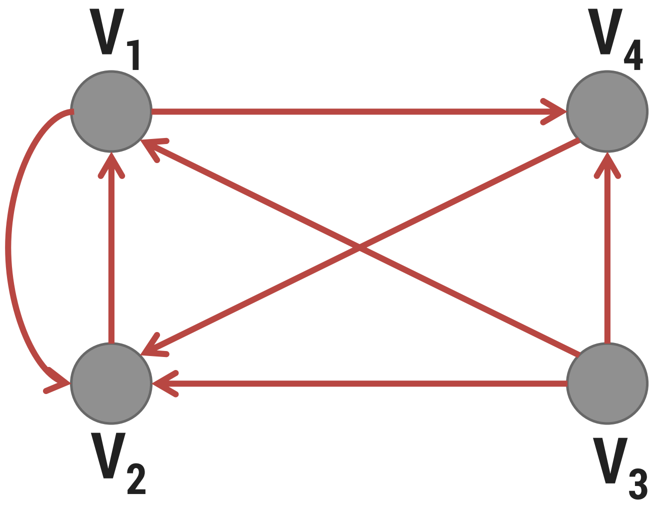
Adjacency matrix

- 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.



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

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

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

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

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

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

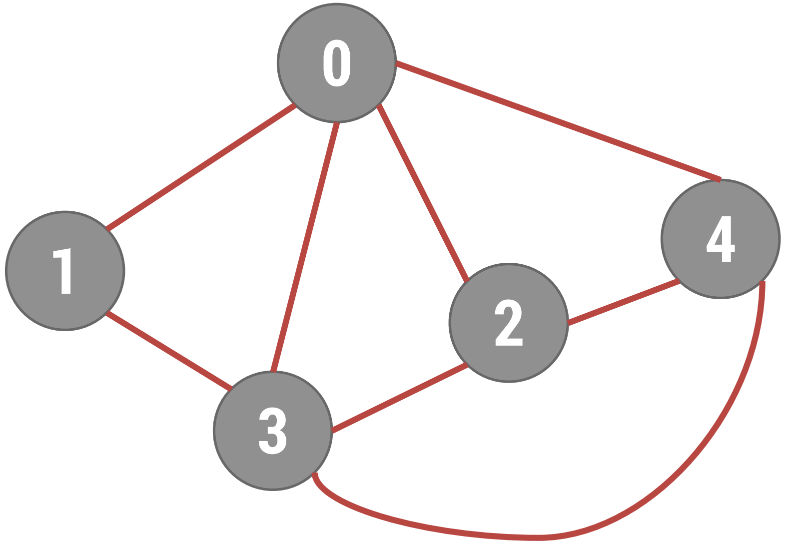
- Entry of **1** in **i^{th}** row and **j^{th}** column of **A** shows existence of an **edge (V_i, V_j)**, that is a **path of length 1**
- Entry in **A^2** shows **no of different paths** of **exactly length 2** from node **V_i** to **V_j**
- Entry in **A^3** shows **no of different paths** of **exactly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

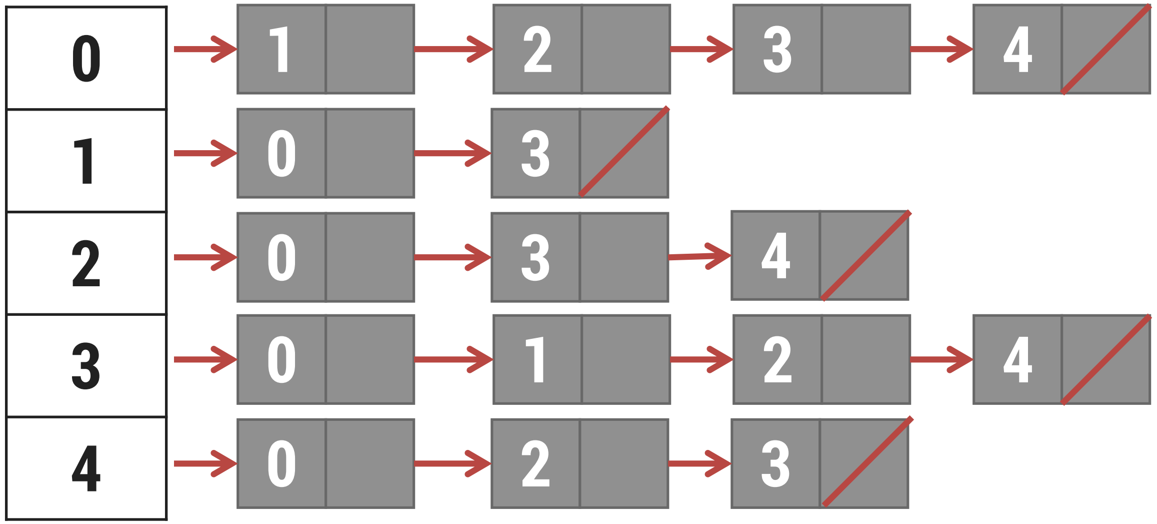
Path matrix or reachability matrix

- 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



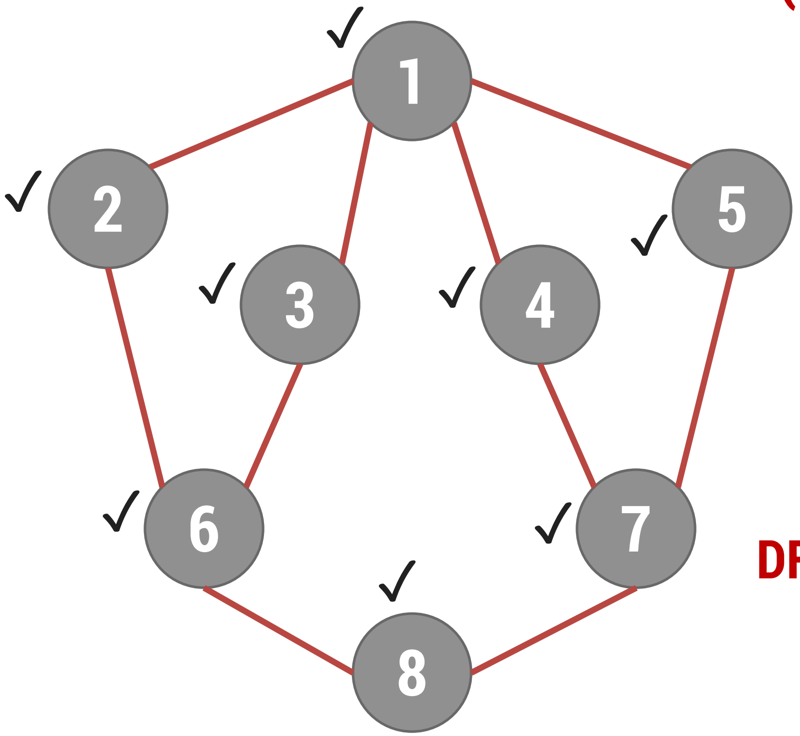


Graph Traversal

- 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

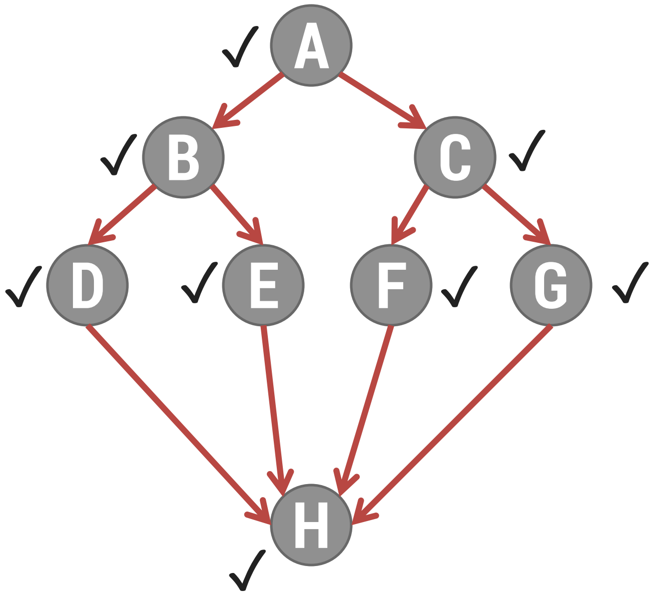
Step 1: Visit (1)

Step 2: DFS (G, 2) → **DFS (G, 2):**
DFS (G, 3) **Step1: Visit(2)**
DFS (G, 4) **Step 2:** DFS (G, 6)
DFS (G, 5) → **DFS (G, 6):**
Step1: Visit(6)
Step 2: DFS (G, 3)
DFS (G, 8)

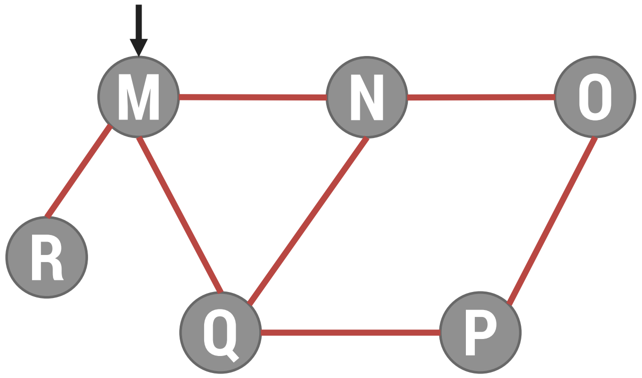
DFS of given graph starting **from** node **1** is given by

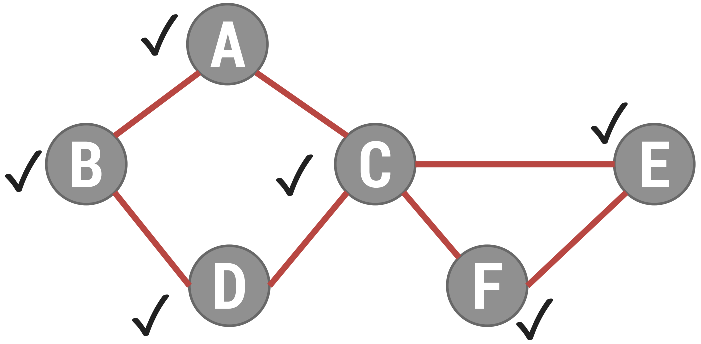
1 2 6 3 8 7 4 5

Depth First Search (DFS)



A B D H E C F G



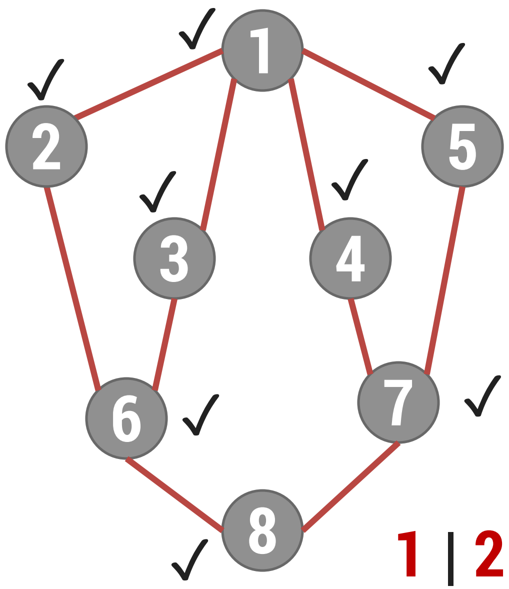


A B D C F E

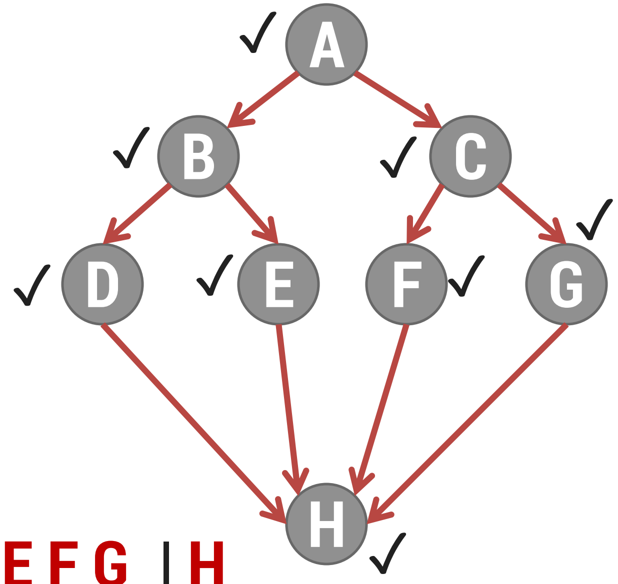
Breadth First Search (BFS)

- 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 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**

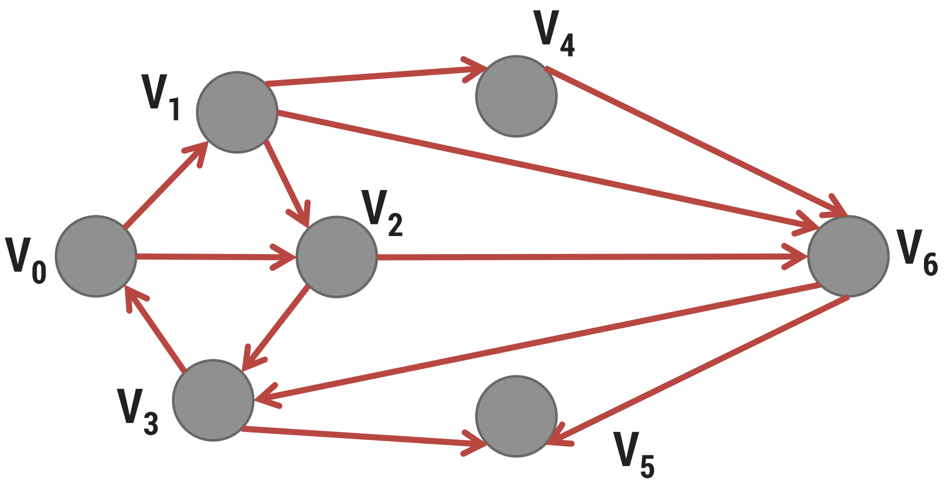
Breadth First Search (BFS)



1 | 2 3 4 5 | 6 7 | 8

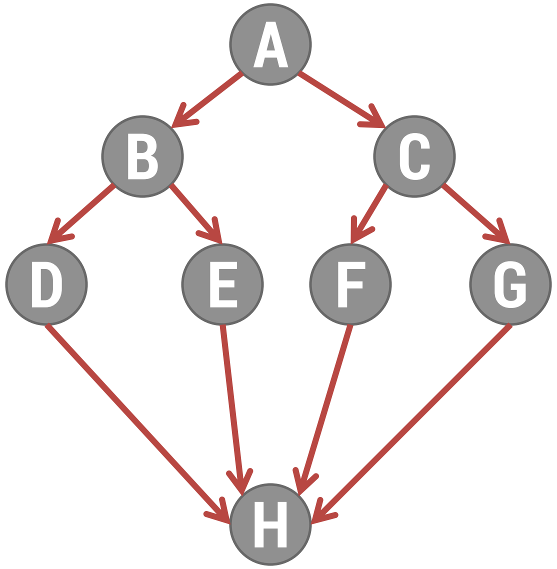


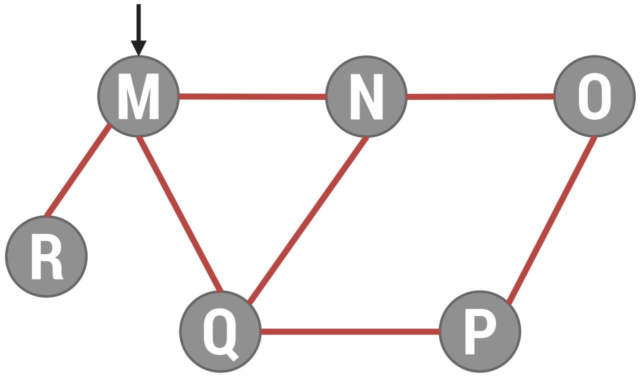
A | B C | D E F G | H

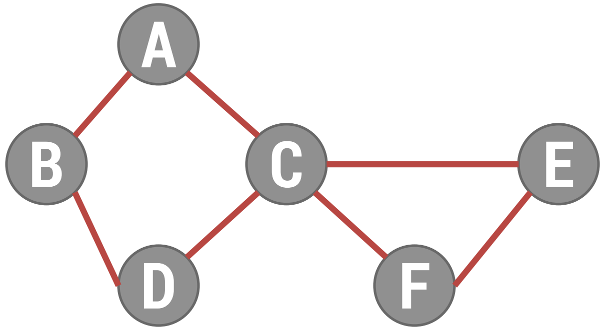


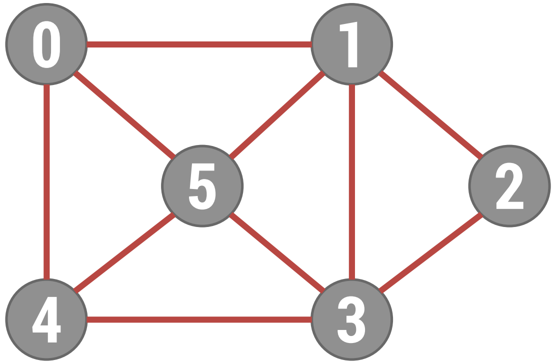
V_0 | V_1 V_2 | V_4 V_6 V_3 | V_5

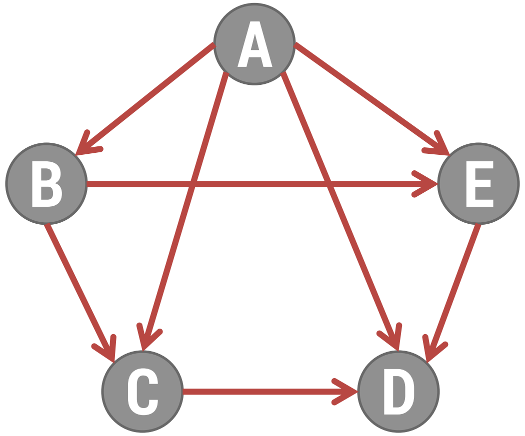
Write DFS & BFS of following Graphs











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
```

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.

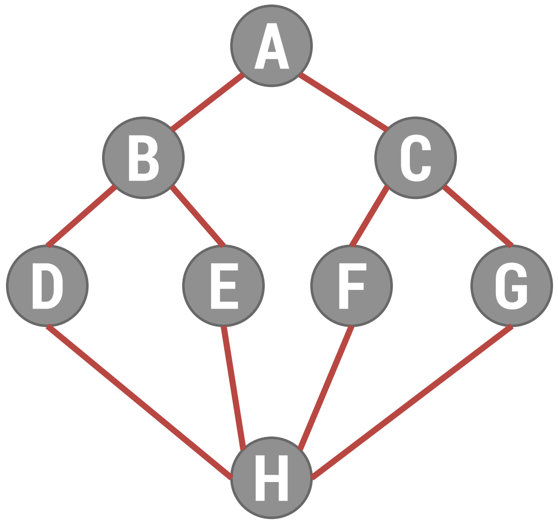
Procedure : BFS (vertex V)

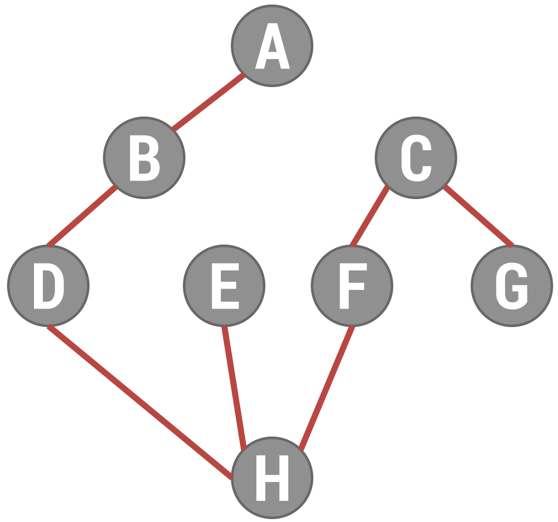
```
1. [Initialize Queue & Visited]
   visited[] ← 0
   F ← R ← 0
2. [Marks visited of V as 1]
   visited[v] ← 1
3. [Add vertex v to Q]
   InsertQueue(V)
4. [Repeat while Q is not Empty]
   Repeat while Q is not empty
     v ← RemoveFromQueue()
     For all vertices W adjacent to v
       If   visited[w]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

Spanning Tree

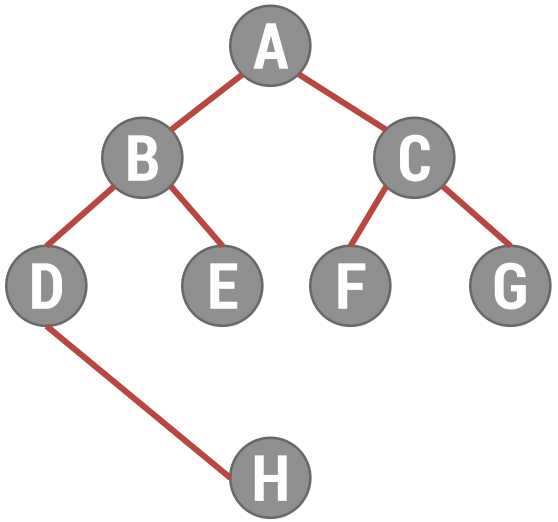
- A **Spanning tree** of a graph is an undirected tree **consisting of only those edges necessary to connect all the nodes** in the original graph
- A spanning tree has the **properties** that
  - For any **pair** of nodes there exists **only one path between them**
  - **Insertion** of any **edge** 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

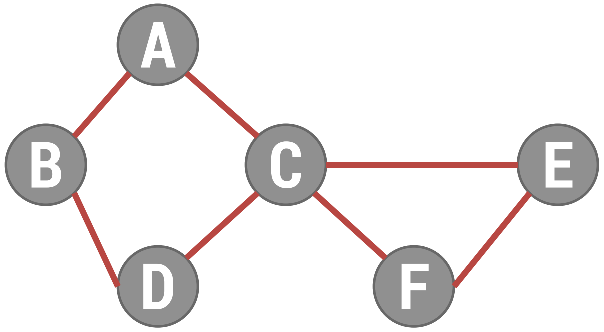


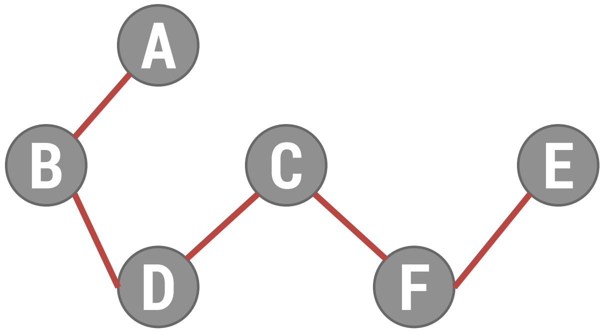


DFS Spanning Tree

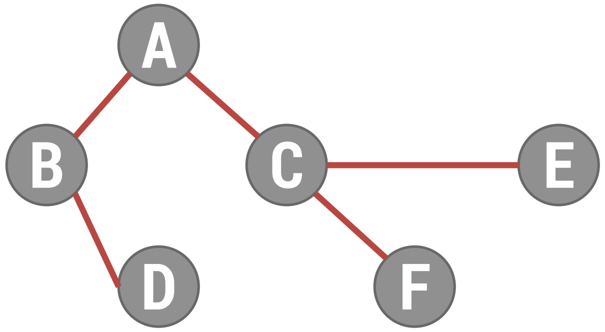


BFS Spanning Tree





DFS Spanning Tree

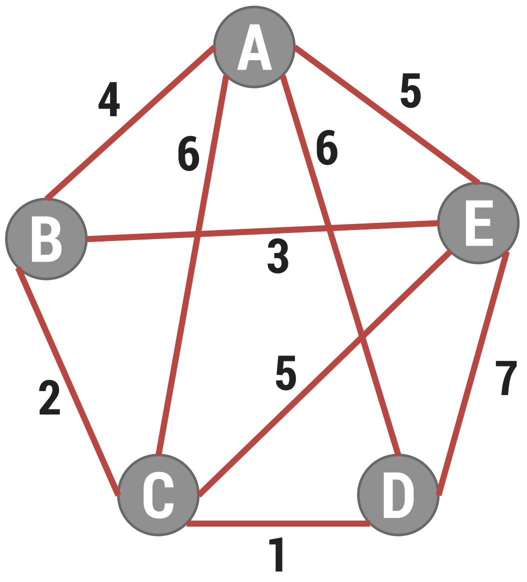


BFS Spanning Tree

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

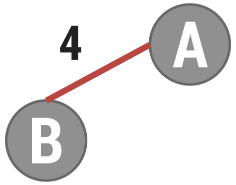
Prims Algorithm



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

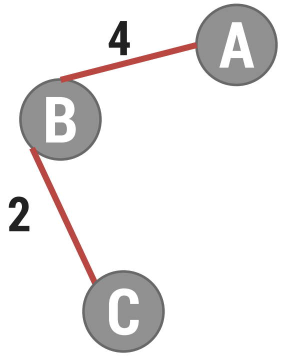
Let X be the set of nodes explored, initially X = { A }

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 }

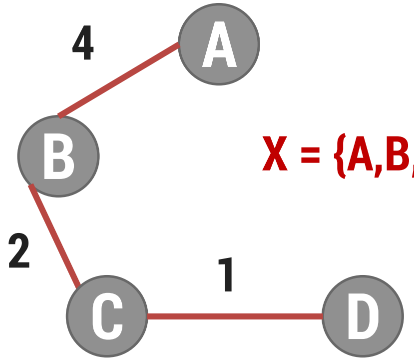


X = {A ,B,C}

We obtained minimum spanning tree of cost:

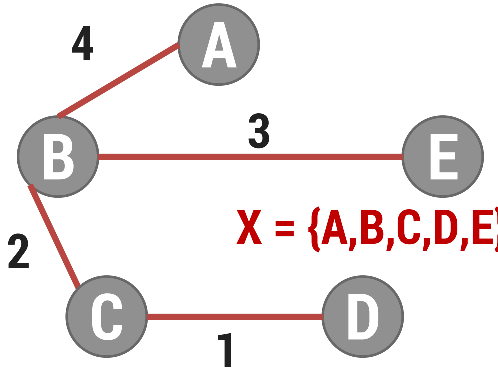
4 + 2 + 1 + 3 = 10

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



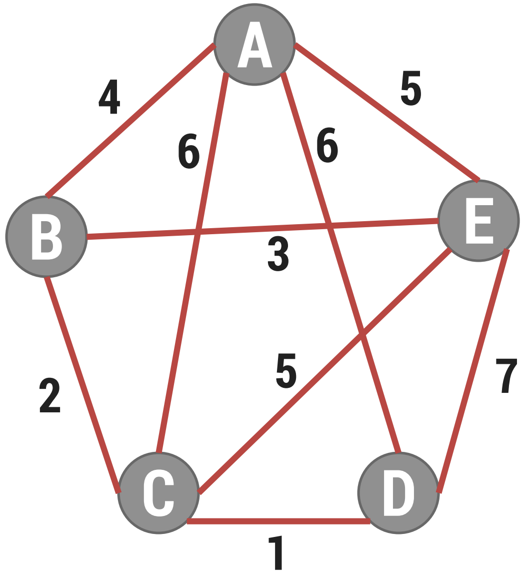
X = {A,B,C,D}

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



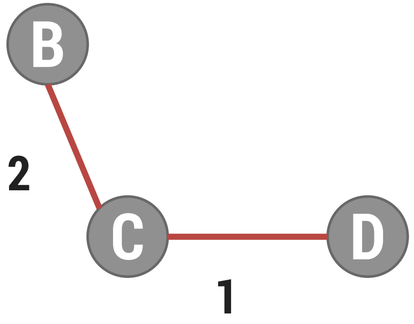
X = {A,B,C,D,E}

Kruskal's Algorithm

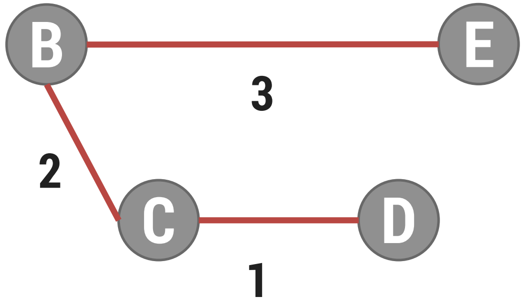


Step 1: Taking min edge (C,D)

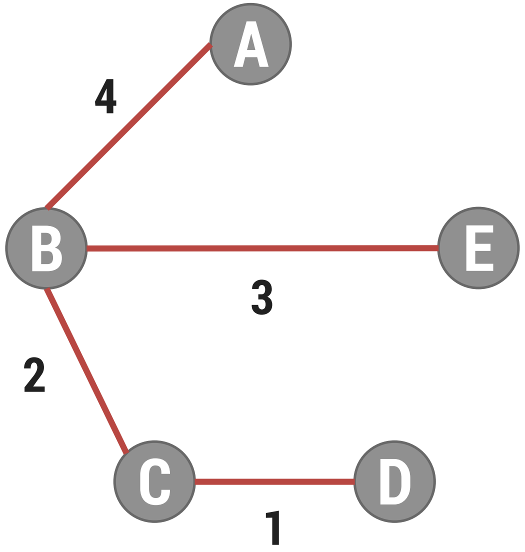
Step 2: Taking next min edge (B,C)



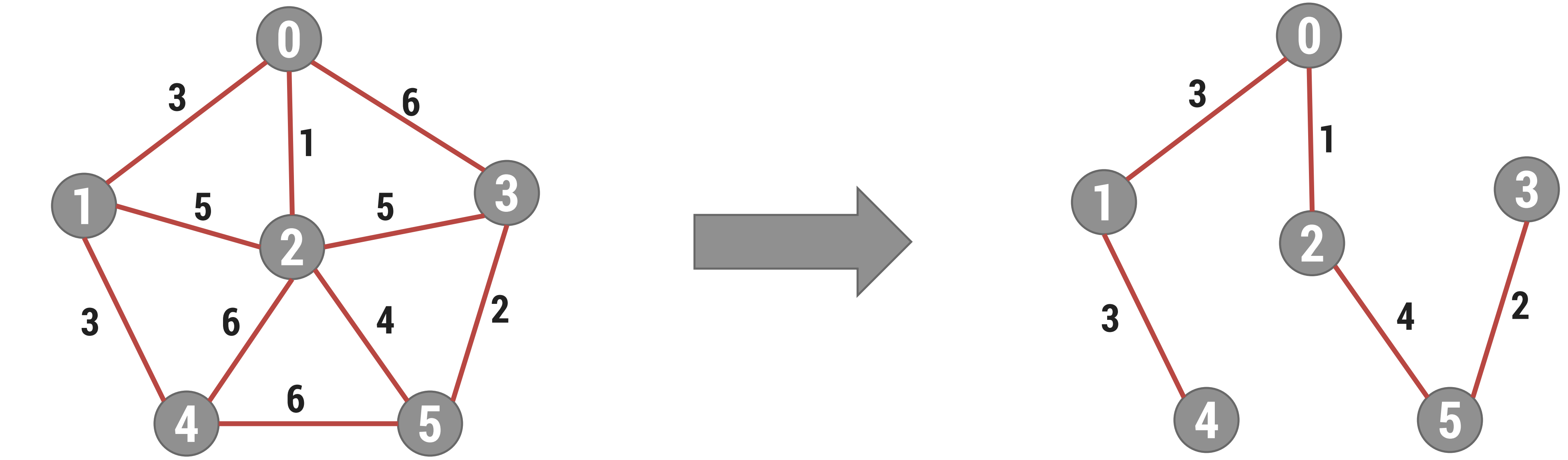
Step 3: Taking next min edge (B,E)

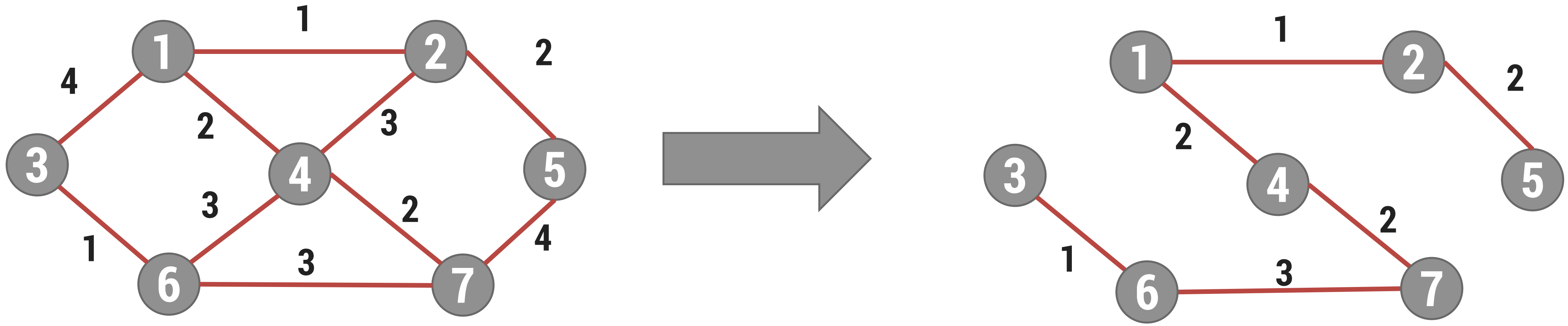


Step 4: Taking next min edge (A,B)



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

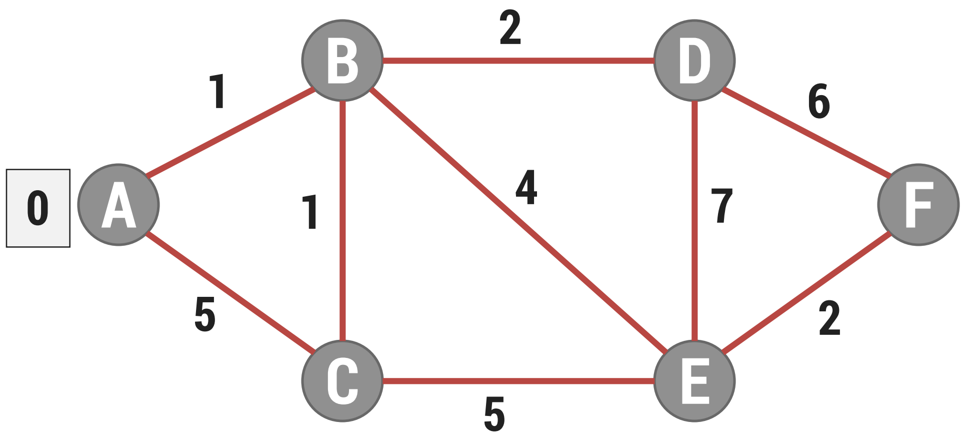




Shortest Path Algorithm

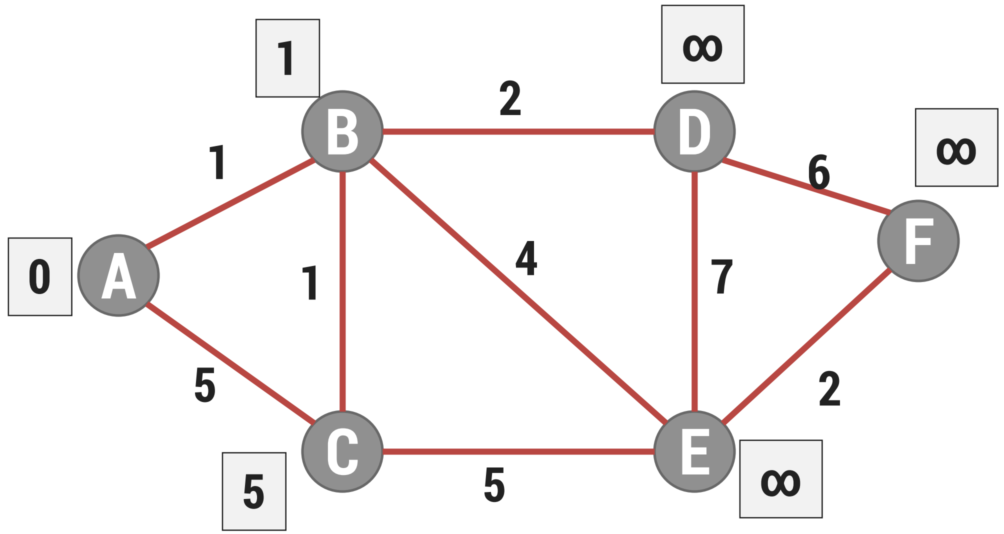
- Let **G = (V,E)** be a simple diagraph with **n vertices**
- The problem is to **find out shortest distance** from a **vertex to all other vertices** of a graph
- **Dijkstra Algorithm** – it is also called Single Source Shortest Path Algorithm

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

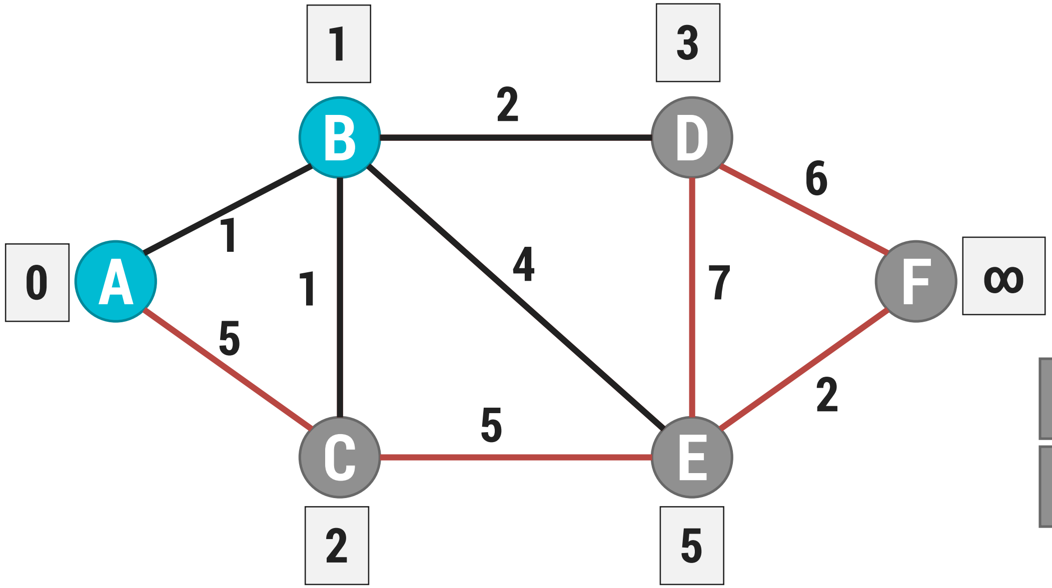
Cost of going to C via B = dist[B] + cost[B][C] = 1 + 1 = 2

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

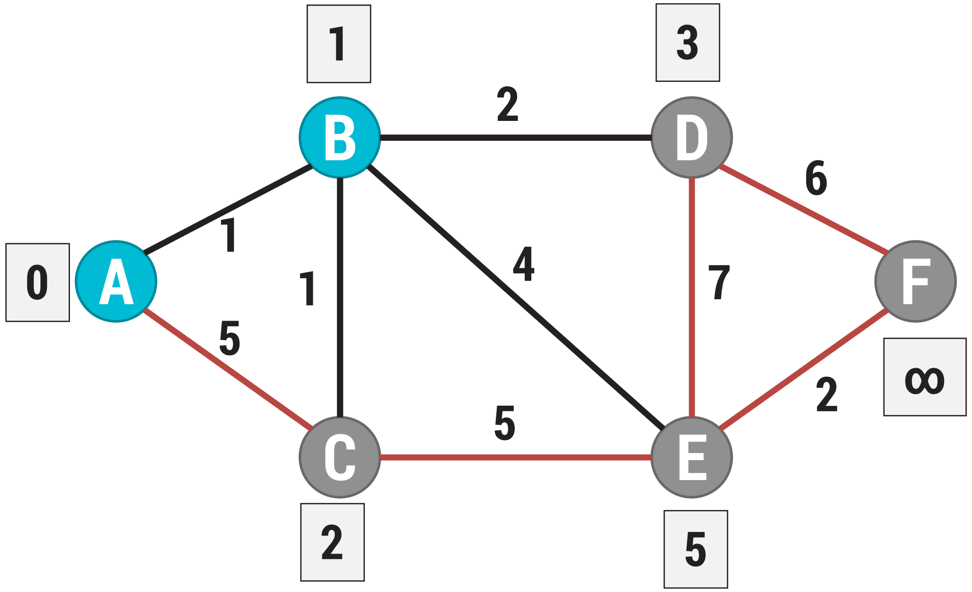
3rd Iteration: Select **Vertex C** via B with minimum distance

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



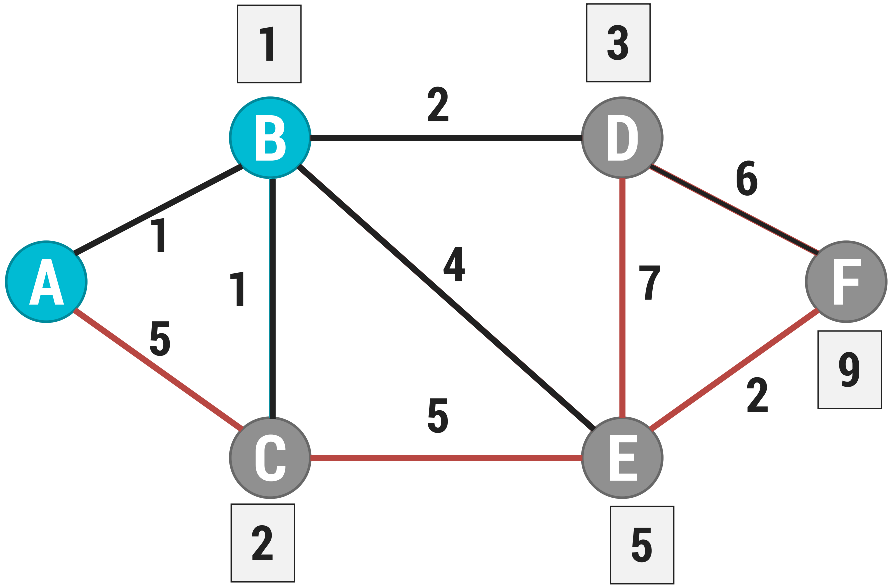
	A	B	C	D	E	F
Distance	0	1	2	3	5	∞
Visited	1	1	1	0	0	0

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

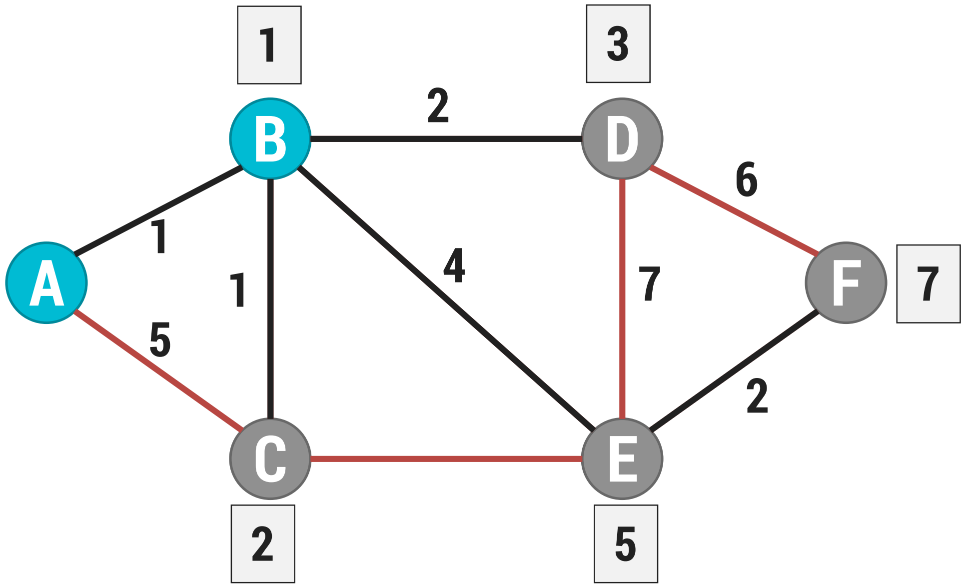


	A	B	C	D	E	F
Distance	0	1	2	3	5	9
Visited	1	1	1	1	0	0

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

Shortest Path

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