Unit-III

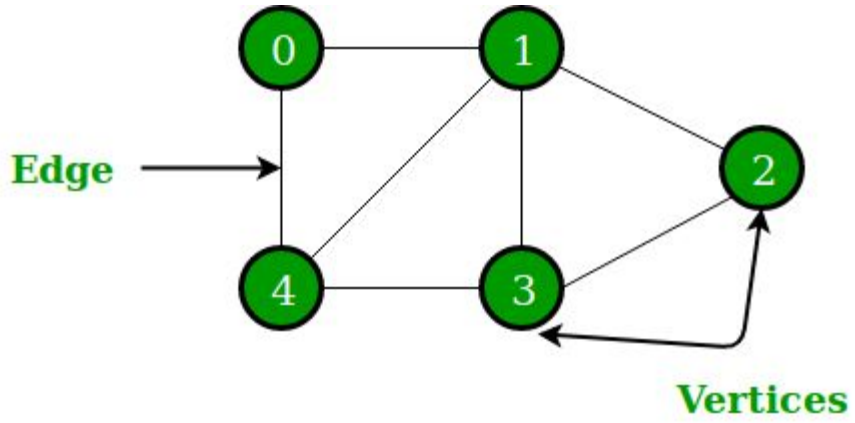
Graphs

Unit III	Graphs	(07 Hours)
Basic Concepts, Storage representation, Adjacency matrix, adjacency list, adjacency multi list, inverse adjacency list. **Traversals**-depth first and breadth first, Minimum spanning Tree, Greedy algorithms for computing minimum spanning tree- Prims and Kruskal Algorithms, Dikjtra's Single source shortest path, All pairs shortest paths- Flyod-Warshall Algorithm Topological ordering.		
#Exemplar/Case Studies	Data structure used in Webgraph and Google map	
Mapping of Course Outcomes for Unit III	CO2,CO3, CO4	

Basic Concepts

- A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- ***A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.***

Basic Concepts



the set of vertices V = {0,1,2,3,4}
the set of edges E = {01, 12, 23, 34, 04, 14, 13}

In **Computer science**
Graphs are used to represent the flow of computation

- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

Graphs are used to solve many real-life problems

- Graphs are used to represent networks.
- The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like linkedIn, Facebook.
- For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc

Storage representation

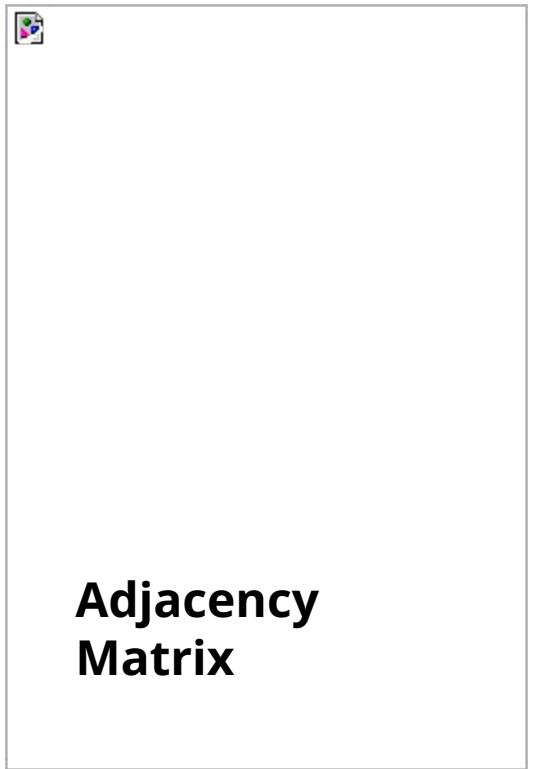
- **Graph and its representations**
- Graph is a data structure that consists of following two components:
  1. A finite set of vertices also called as nodes.
  2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
- Following two are the most commonly used representations of a graph.
  1. Adjacency Matrix
  2. Adjacency List

Adjacency matrix

- **Adjacency Matrix:**
  Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Example : Undirected graph with 5 vertices.

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).



Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List

- An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph.

Adjacency multi list

- In adjacency multi-list, node structure is as follows:
- <M, V1, V2, Link1, Link2>
- M: Mark
- V1: Source Vertex of Edge
- V1: Destination Vertex of Edge
- Link1: Address of other node (i.e. Edge) incident on V1
- Link2: Address of other node (i.e. Edge) incident on V2
- Every graph can be represented as list of such EDGE NODEs.

Adjacency multi list-Example

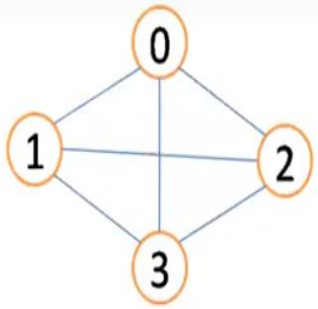


Fig: Undirected graph

List of Edges	Numbe r
(0 , 1)	N0
(0 , 2)	N1
(0 , 3)	N2
(1 , 2)	N3
(1 , 3)	N4
(2 , 3)	N5

Edges	M	Vertex-1	Vertex-2	List-1	List-2
N0		0	1		
N1		0	2		
N2		0	3		
N3		1	2		
N4		1	3		
N5		2	3		

Fig: Adjacency multi-list

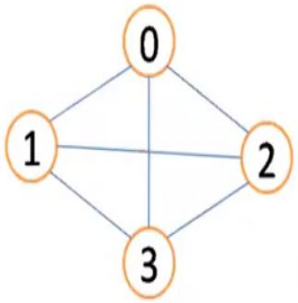


Fig: Undirected graph

List of Edges	Numbe r
(0 , 1)	N0
(0 , 2)	N1
(0 , 3)	N2
(1 , 2)	N3
(1 , 3)	N4
(2 , 3)	N5

Edges	M	Vertex-1	Vertex-2	List-1	List-2
N0		0	1	N1	N3
N1		0	2		
N2		0	3		
N3		1	2		
N4		1	3		
N5		2	3		

Fig: Adjacency multi-list

Adjacency multi list-Example

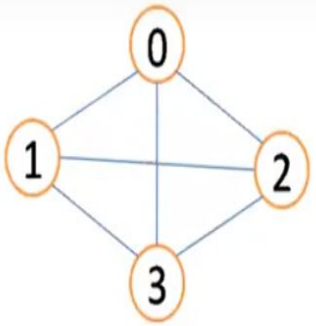


Fig: Undirected graph

List of Edges	Numbe r
(0 , 1)	N0
(0 , 2)	N1
(0 , 3)	N2
(1 , 2)	N3
(1 , 3)	N4
(2 , 3)	N5

Edges	M	Vertex-1	Vertex-2	List-1	List-2
N0		0	1	N1	N3
N1		0	2	N2	N3
N2		0	3	NIL	N4
N3		1	2		
N4		1	3		
N5		2	3		

Fig: Adjacency multi-list

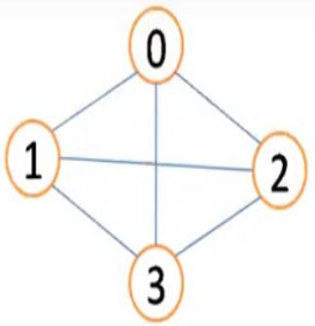


Fig: Undirected graph

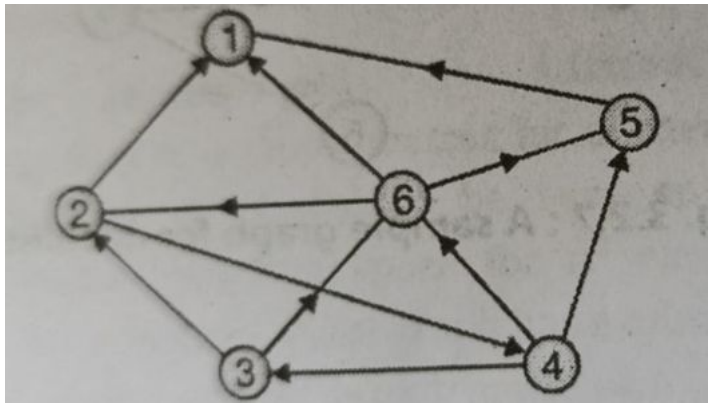
List of Edges	Numbe r
(0 , 1)	N0
(0 , 2)	N1
(0 , 3)	N2
(1 , 2)	N3
(1 , 3)	N4
(2 , 3)	N5

Edges	M	Vertex-1	Vertex-2	List-1	List-2
N0		0	1	N1	N3
N1		0	2	N2	N3
N2		0	3	NIL	N4
N3		1	2	N4	N5
N4		1	3	NIL	N5
N5		2	3	NIL	NIL

Fig: Adjacency multi-list

Inverse adjacency list

- Used to determine the in-degree of a vertex
- Keeps track of edges coming into a vertex.

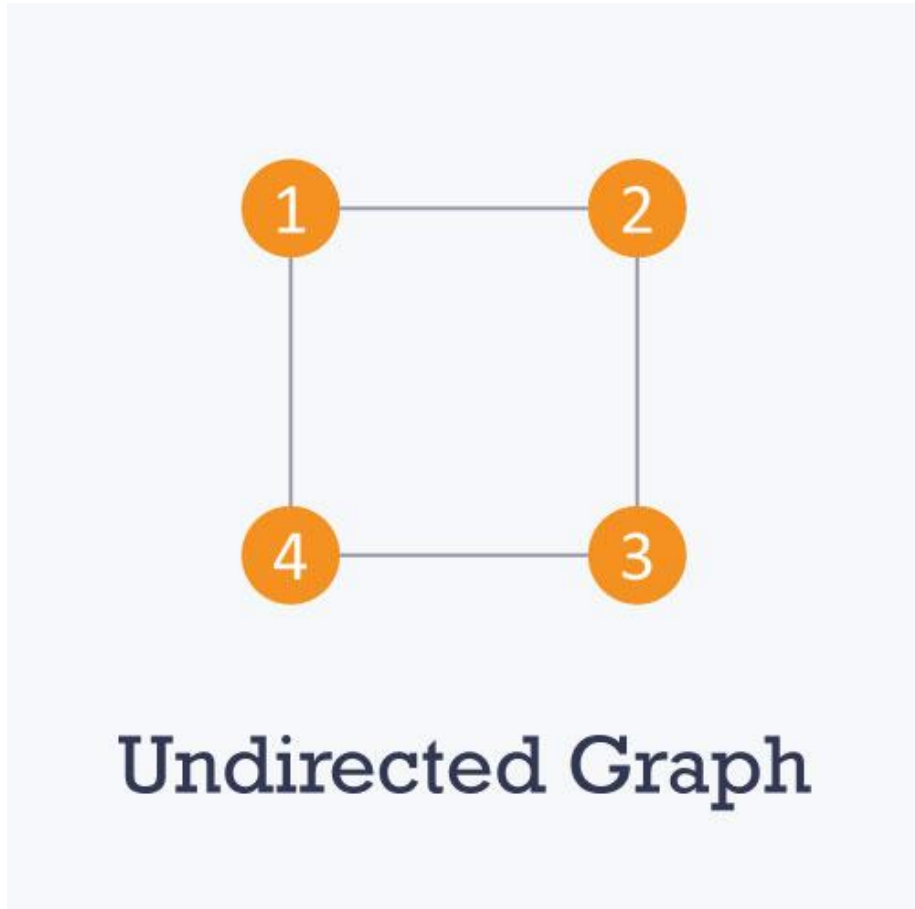


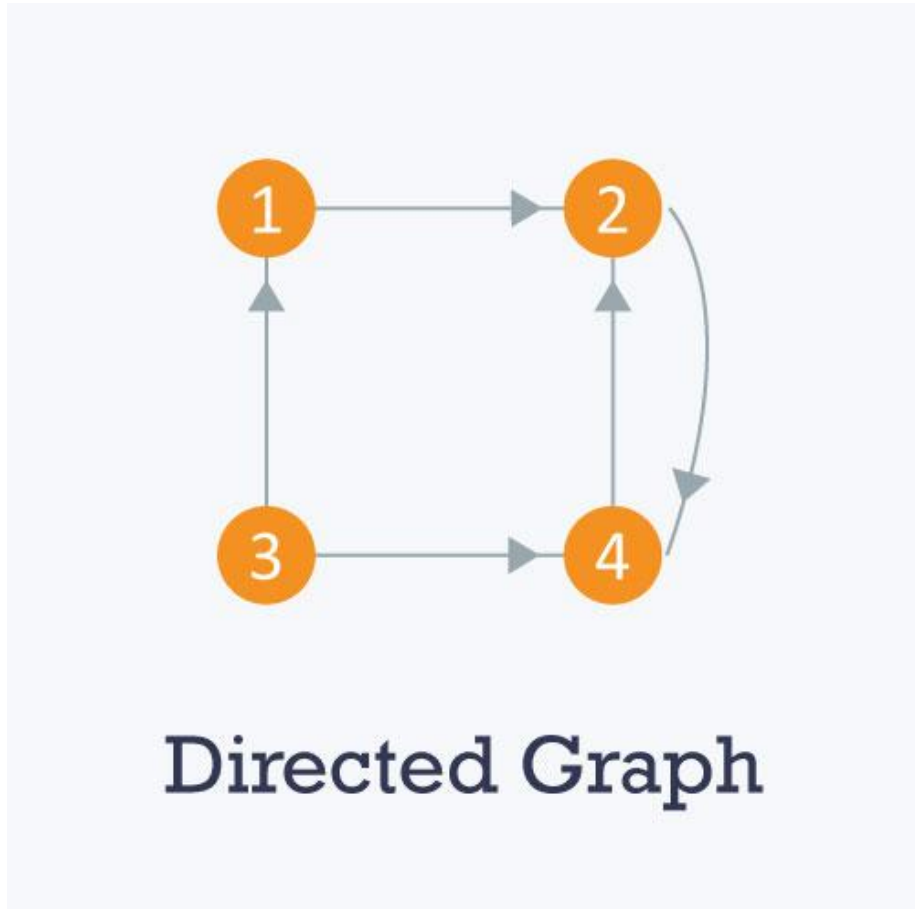
1-> {2,5,6)
2->{3,6}
3->{4}
4->{2}

Inverse Adjacency list

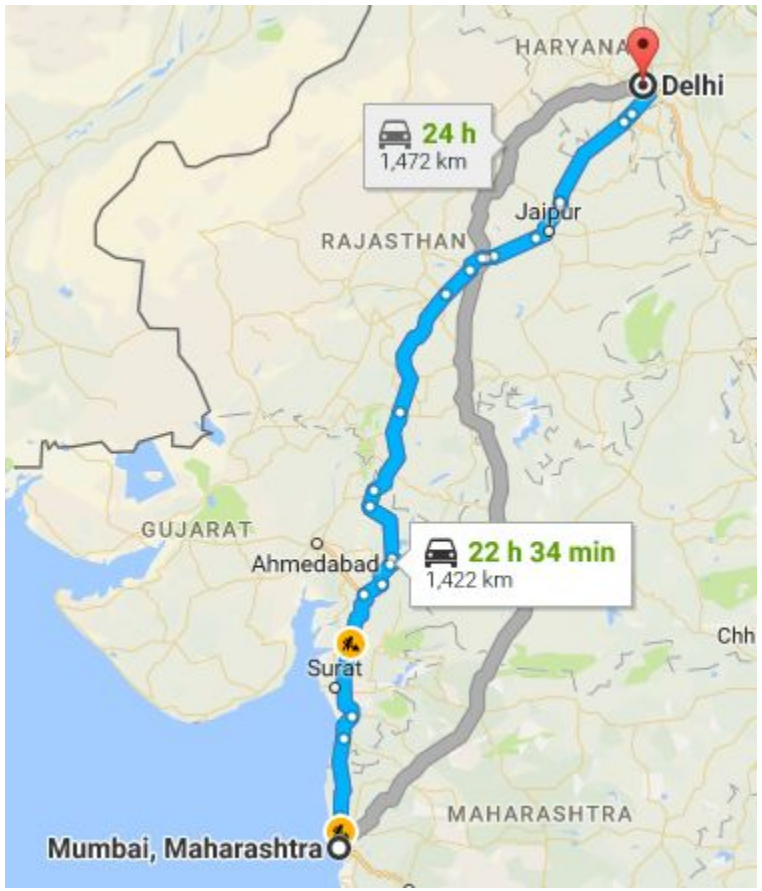
Types of graphs

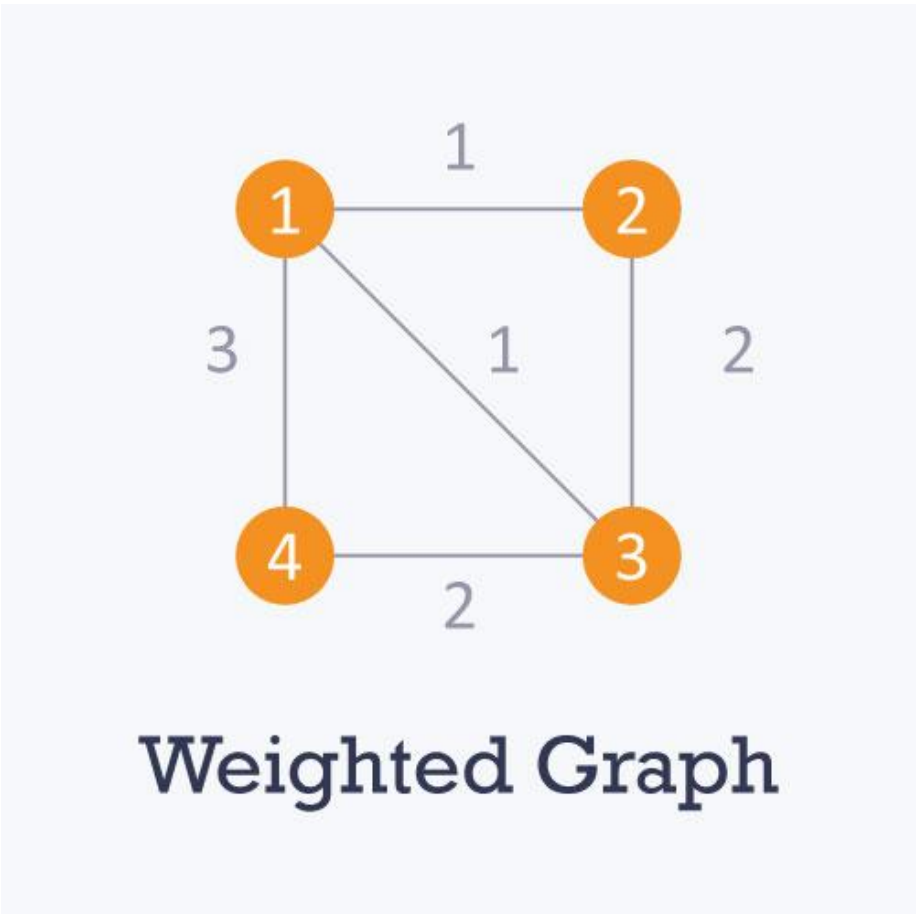
- Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.
- Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.
- Weighted: In a weighted graph, each edge is assigned a weight or cost.


1
2
4
3
Undirected Graph


1
2
3
4
Directed Graph

Example : Google Maps





- graphs can show almost any type of relationship with just data and edges.
- This is why graphs have become so widely used by companies like LinkedIn, Google, and Facebook.

Traversals-depth first and breadth first

- Traversal means visiting all the nodes of a graph.

DFS algorithm

- A standard DFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

The BFS algorithm works as follows:

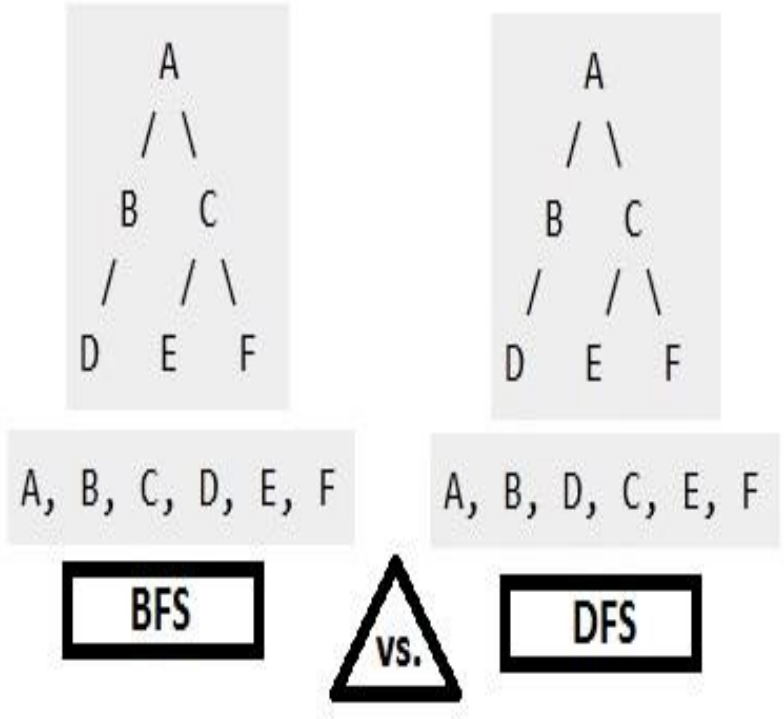
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

S. No.	Breadth First Search (BFS)	Depth First Search (DFS)
1.	BFS visit nodes **level by level** in Graph.	DFS visit nodes of graph **depth wise**. It visits nodes until reach a leaf or a node which doesn't have non-visited nodes.
2.	A node is fully explored before any other can begin.	Exploration of a node is suspended as soon as another unexplored is found.
3.	Uses Queue data structure to store Un-explored nodes.	Uses Stack data structure to store Un-explored nodes.
4.	BFS is slower and require more memory.	DFS is faster and require less memory.
5.	**Some Applications:** •Finding all connected components in a graph. •Finding the shortest path between two nodes. •Finding all nodes within one connected component. •Testing a graph for bipartiteness.	**Some Applications:** •Topological Sorting. •Finding connected components. •Solving puzzles such as maze. •Finding strongly connected components. •Finding articulation points (cut vertices) of the graph.

Difference between BFS and DFS

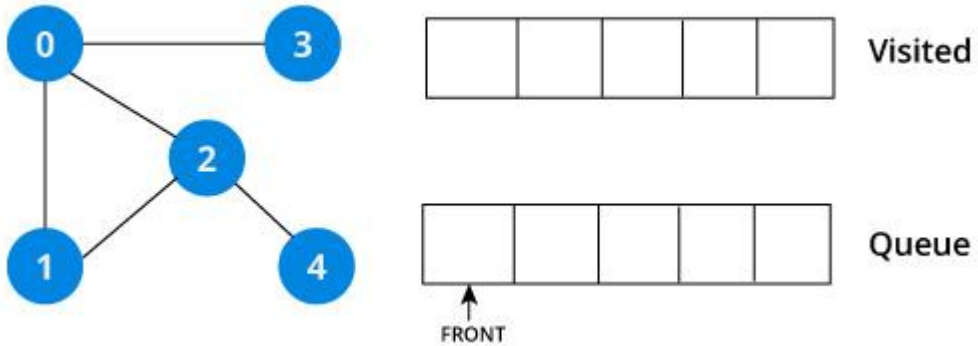
Example

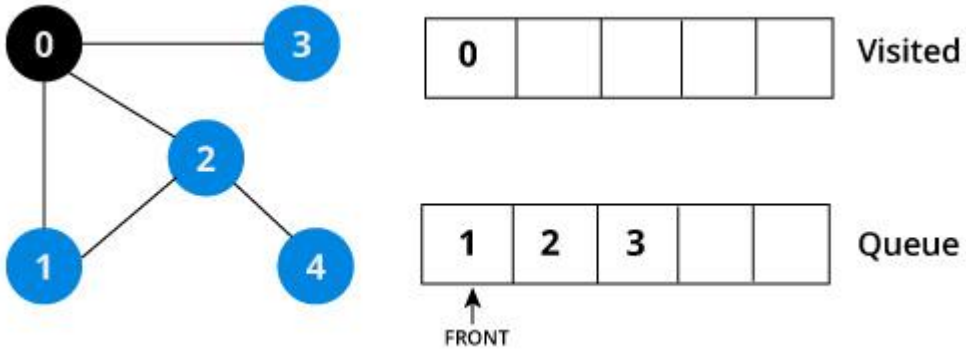
Considering A as starting vertex.

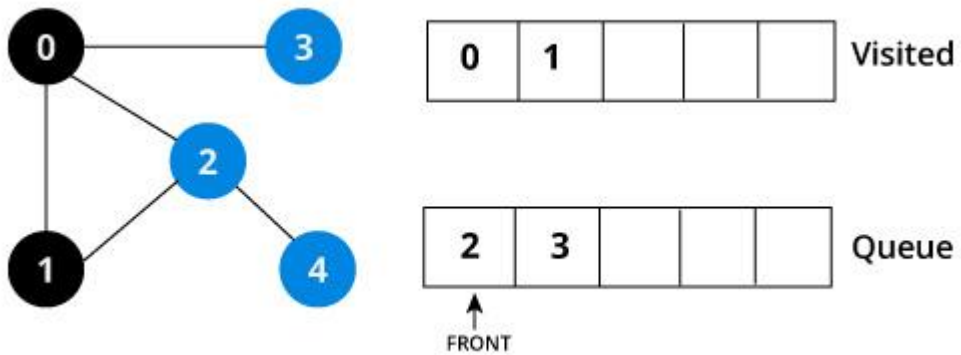


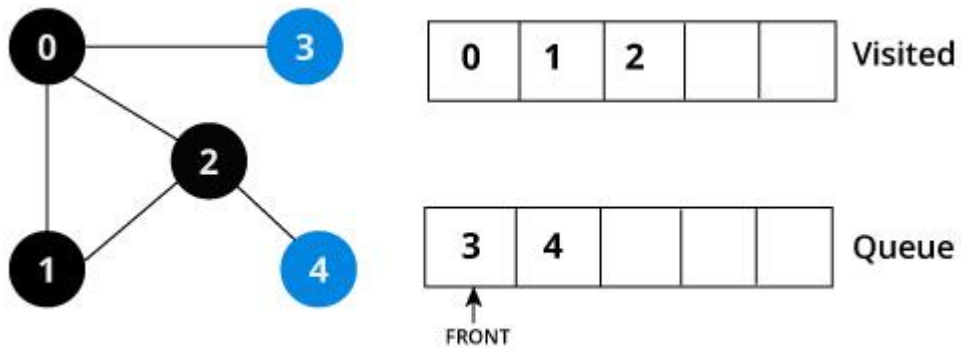
BASIS FOR COMPARISON	BFS	DFS
Basic	Vertex-based algorithm	Edge-based algorithm
Data structure used to store the nodes	Queue	Stack
Memory consumption	Inefficient	Efficient
Structure of the constructed tree	Wide and short	Narrow and long
Traversing fashion	Oldest unvisited vertices are explored at first.	Vertices along the edge are explored in the beginning.
Optimality	Optimal for finding the shortest distance, not in cost.	Not optimal
Application	Examines bipartite graph, connected component and	Examines two-edge connected graph, strongly connected graph, acyclic graph and

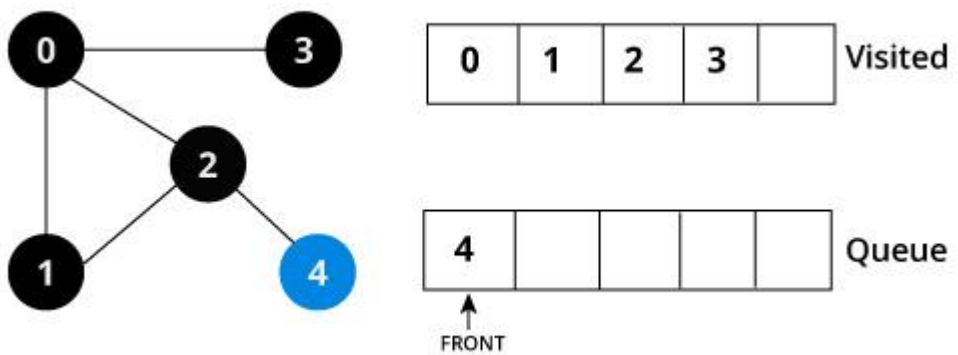
BFS example

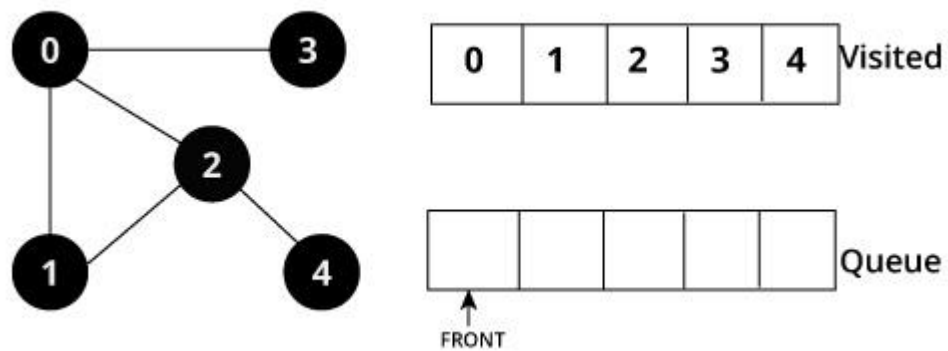

0
3
2
1
4
Visited
Queue
FRONT


0
3
2
1
4
0
Visited
1 2 3
Queue
FRONT


0
3
2
1
4
0 1
Visited
2 3
Queue
FRONT


0
3
2
1
4
0 1 2
Visited
3 4
Queue
FRONT


0
3
2
1
4
0 1 2 3
Visited
4
Queue
FRONT


0
3
2
1
4
0 1 2 3 4
Visited
Queue
FRONT

BFS pseudocode

- create a queue Q

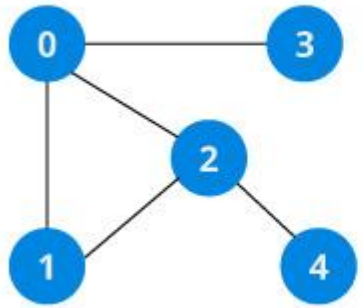
mark v as visited and put v into Q

while Q is non-empty

  remove the head u of Q

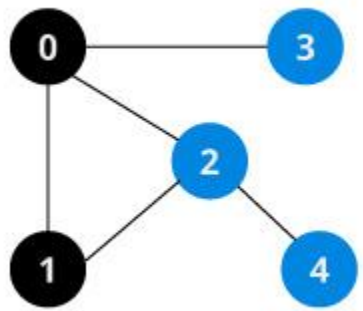
  mark and enqueue all (unvisited) neighbours of u

DFS example



Visited

Stack

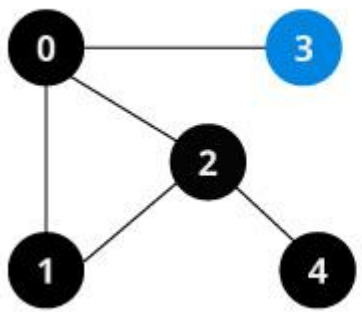


0	1			

Visited

2	3			

Stack

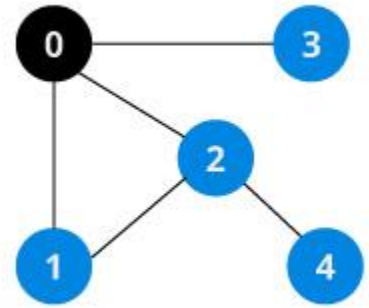


0	1	2	4	

Visited

3				

Stack

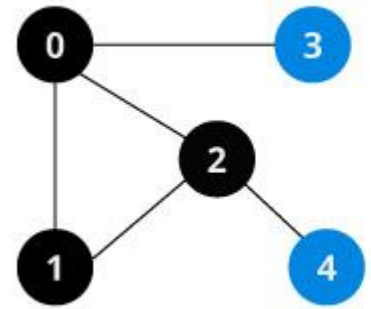


0				

Visited

1	2	3		

Stack

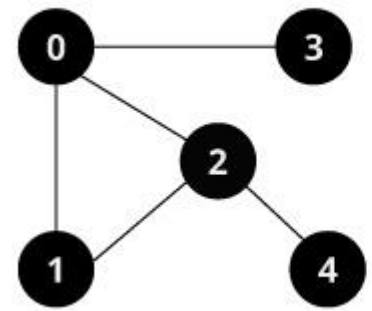


0	1	2		

Visited

4	3			

Stack



0	1	2	4	3

Visited

Stack

DFS pseudocode (recursive implementation)

- DFS(G, u)
  u.visited = true
  for each v ∈ G.Adj[u]
    if v.visited == false
      DFS(G,v)

- init() {
  For each u ∈ G
    u.visited = false
  For each u ∈ G
  DFS(G, u)
  }

Introduction to Greedy Strategy

- An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.
- Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Examples

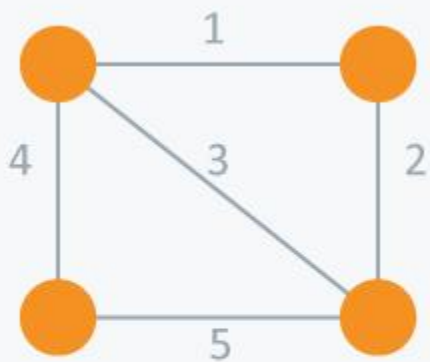
- Most networking algorithms use the greedy approach. Here is a list of few of them −
- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

Minimum spanning Tree

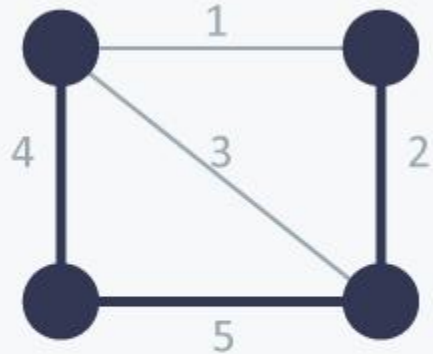
- **What is a Spanning Tree?**
- Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)
- **What is a Minimum Spanning Tree?**
- The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Applications

- Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:
- Cluster Analysis
- Handwriting recognition
- Image segmentation

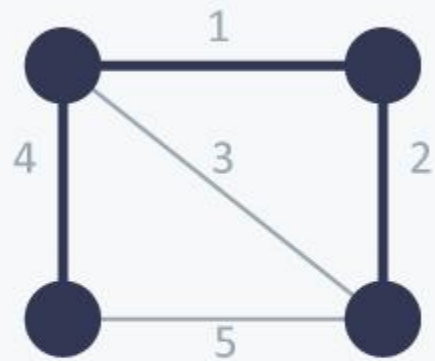


Undirected Graph



Spanning Tree

Cost = 11(=4+5+2)



Minimum Spanning Tree

Cost = 7(=4+1+2)

Greedy algorithms for computing minimum spanning tree- Prims and Kruskal Algorithms

- **Prim’s Algorithm**

- Prim’s Algorithm also use Greedy approach to find the minimum spanning tree

- **Kruskal’s Algorithm**
- Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Prims Algorithms

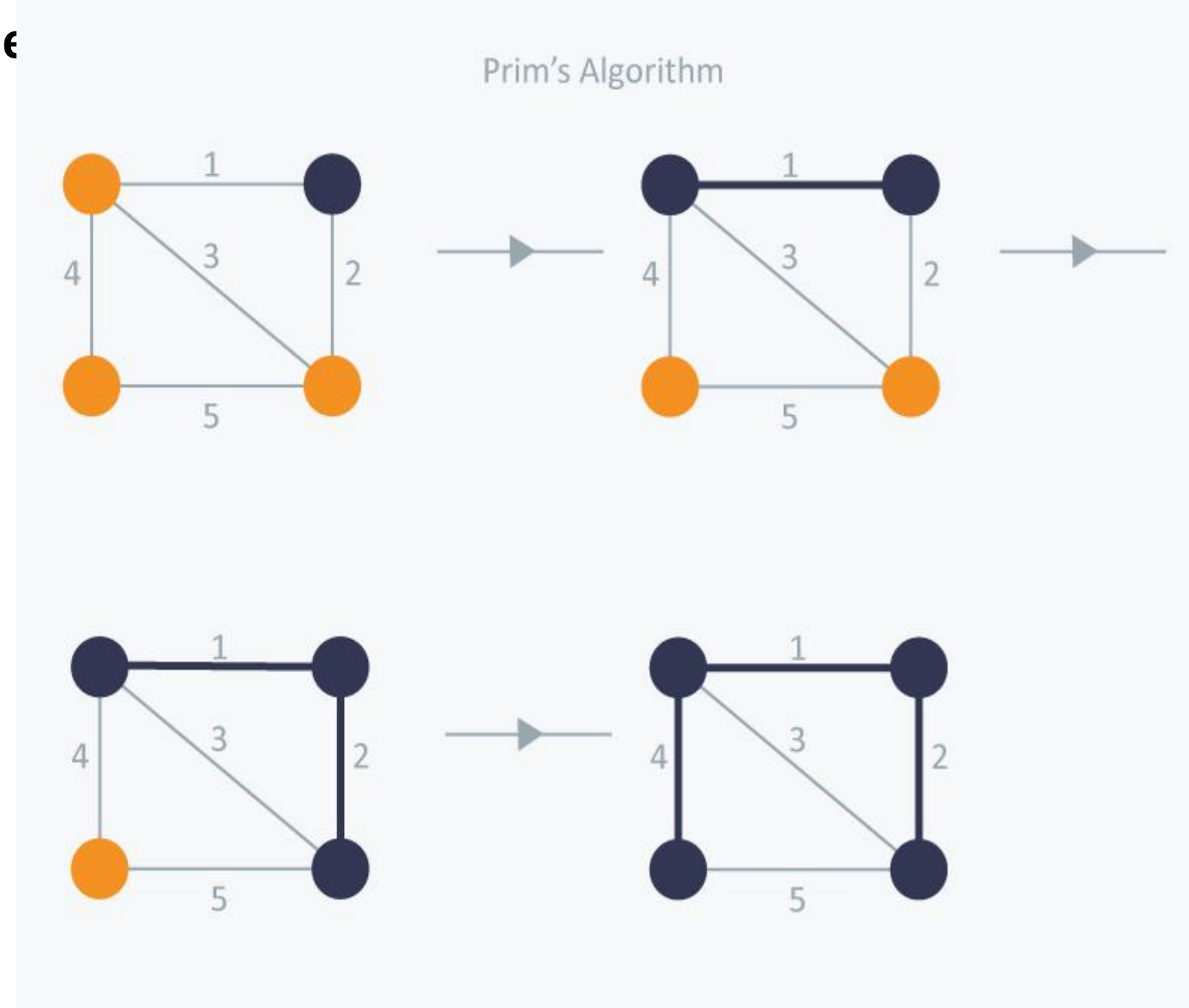
- **Algorithm Steps:**

1. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
2. Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
3. Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Prim’s Algorithm Example

minimum spanning tree of total cost 7 (= 1 + 2 +4).

Time Complexity:
The time complexity of the Prim’s Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.



Kruskal Algorithm

- Greedy Algorithm
- Used to generate a minimum spanning tree for a given graph
- Sorts all the edges in increasing order of their edge weights and keeps adding nodes to the tree only if the chosen edge does not form any cycle.
- Picks the edge with a minimum cost at first and the edge with a maximum cost at last
- Algorithm makes a locally optimal choice, intending to find the global optimal solution.

Steps – To Create MST Using Kruskal Algorithm

Step 1: Sort all edges in increasing order of their edge weights.

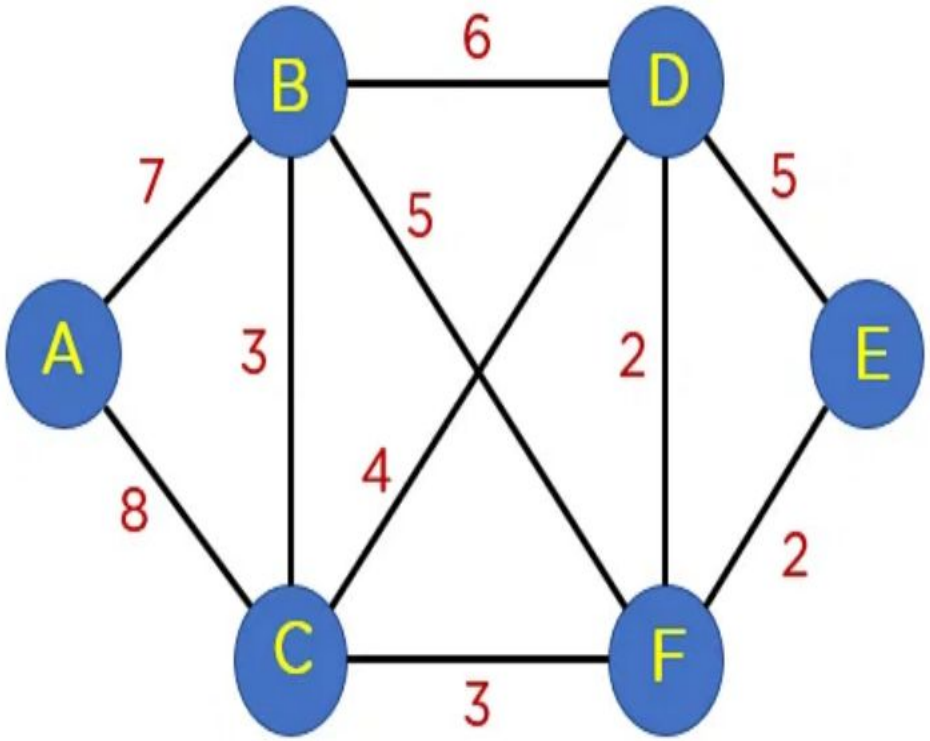
Step 2: Pick the smallest edge.

Step 3: Check if the new edge creates a cycle or loop in a spanning tree.

Step 4: If it doesn’t form the cycle, then include that edge in MST. Otherwise, discard it.

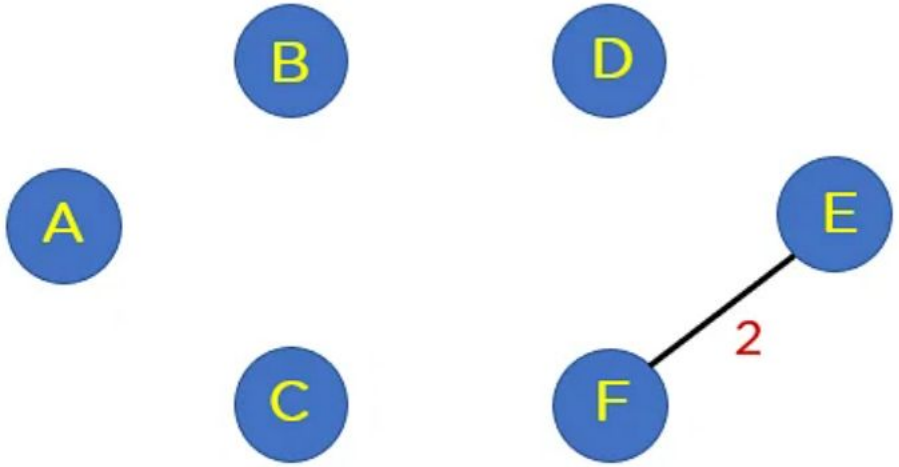
Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.

Example

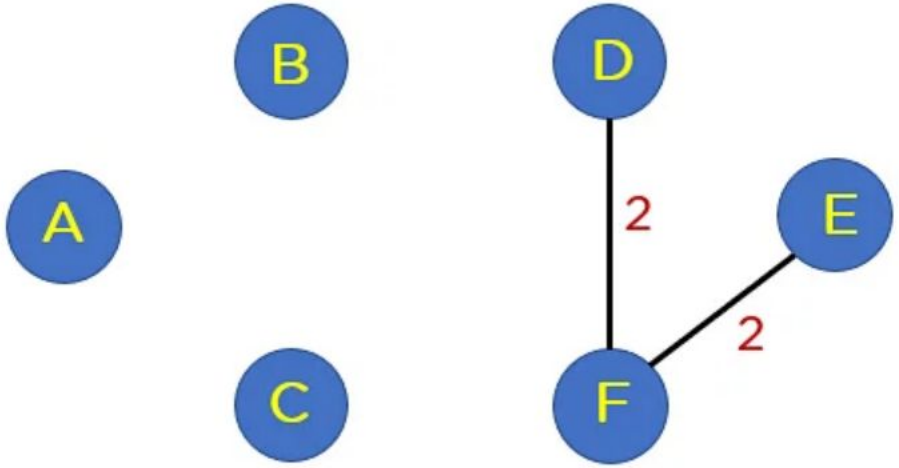


The Edges of the Graph	Edge Weight	
Source Vertex	Destination Vertex	
E	F	2
F	D	2
B	C	3
C	F	3
C	D	4
B	F	5
B	D	6
A	B	7
A	C	8

Example

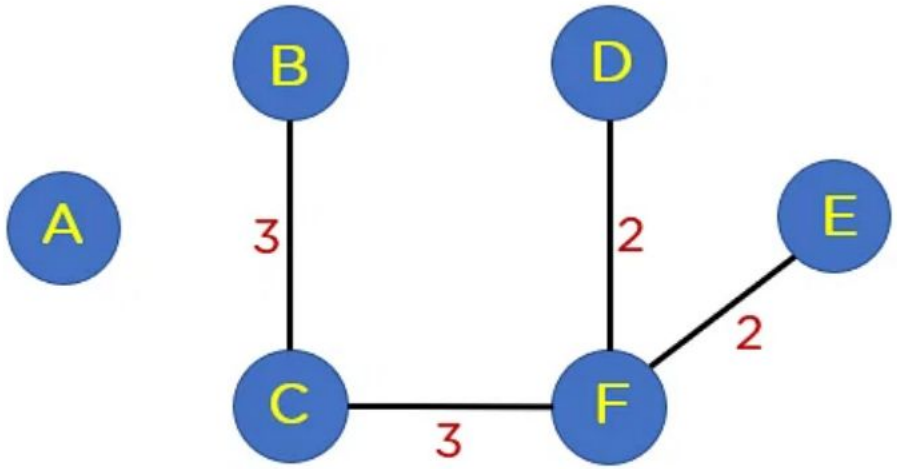


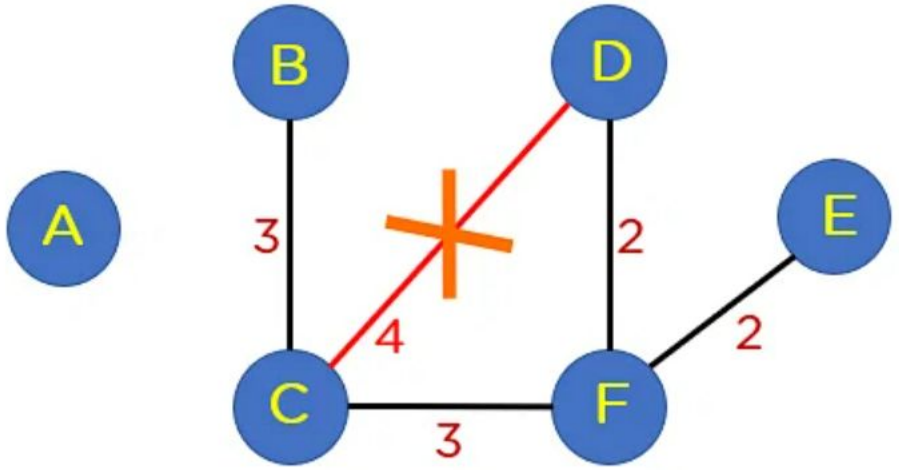
Add edge EF minimum weight edge



Add edge FD to the spanning tree.

Example

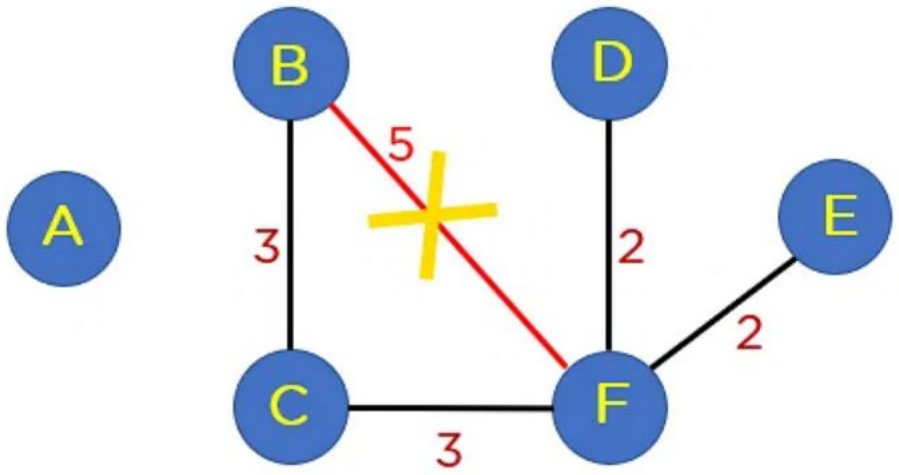




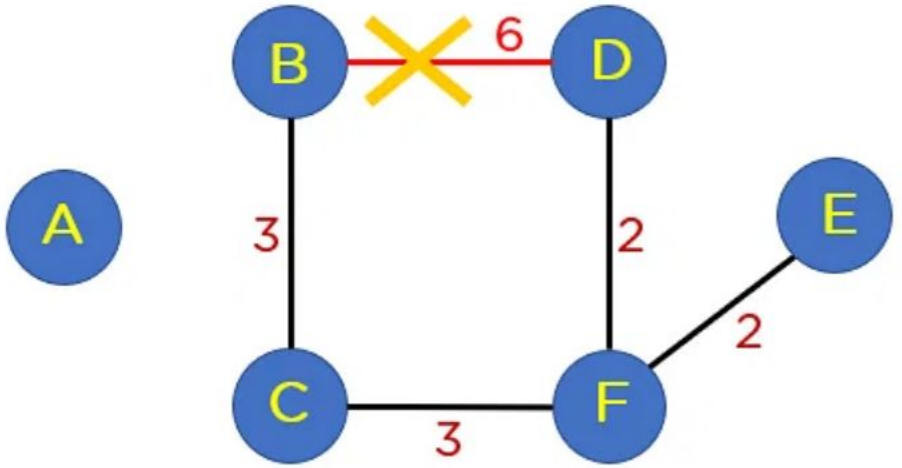
Edge CD should be discarded, as it creates loop.

Add edge BC and edge CF to the spanning tree as it does not generate any loop

Example

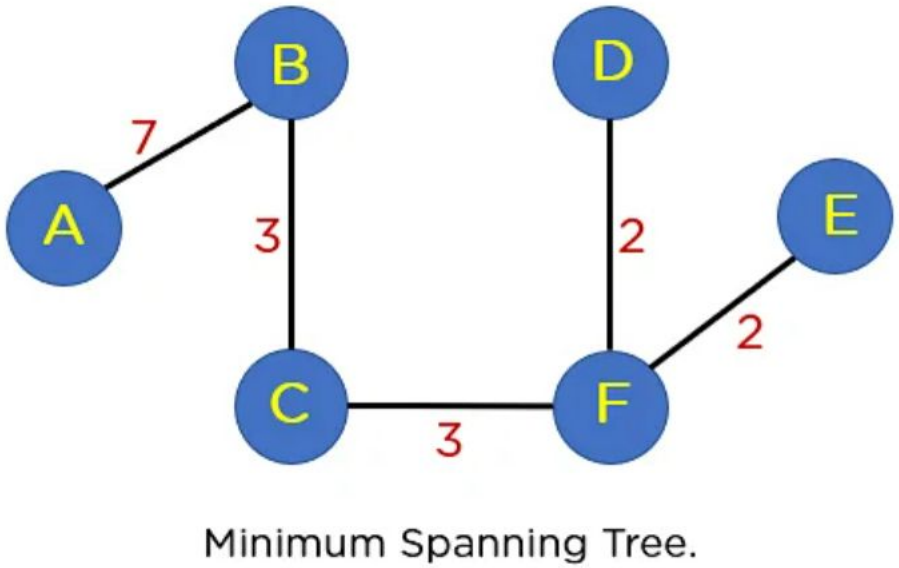


Edge BF should be discarded.



Edge BD should be discarded.

Example



Minimum Spanning Tree.

Add edge AB does not generate any cycle

The summation of all the edge weights in MST T(V', E') is equal to 17

Complexity

- The time complexity Of Kruskal's Algorithm is: $O(E \log E)$.
-

Dikjtra's Single source shortest path

- Solution to the single –source Shortest path problem in graph theory
- Both directed and undirected graph
- All edges must have non negative weight
- Graph must be connected
- Remove all self loop and parallel edge
- It is applied on weighted graph

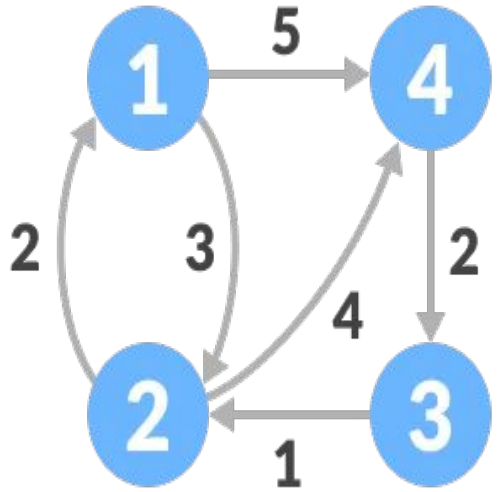
Algorithm for Dijkstra's Algorithm

- Mark the source node with a current distance of 0 and the rest with infinity.
- Set the non-visited node with the smallest current distance as the current node.
- For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
- Mark the current node 1 as visited.
- Go to step 2 if there are any nodes are unvisited.

All pairs shortest paths- Flyod-Warshall Algorithm

- Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the ***directed and undirected weighted graphs.***
- It does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
- Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm,
- follows the dynamic programming approach to find the shortest paths.

All pairs shortest paths- Flyod-Warshall Algorithm



Initial graph

Create a matrix A0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Fill each cell with the distance between ith and jth vertex

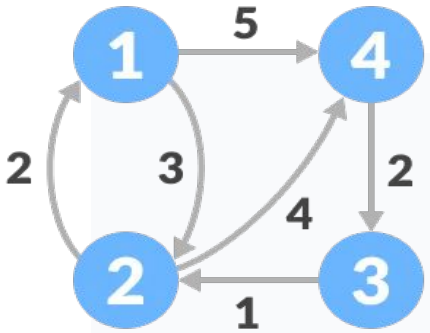
All pairs shortest paths- Flyod-Warshall Algorithm

- create a matrix A1 using matrix A0 The elements in the first column and the first row are left as they are.

Let k be the intermediate vertex in the shortest path from source to destination.
In this step, k is the first vertex.

A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

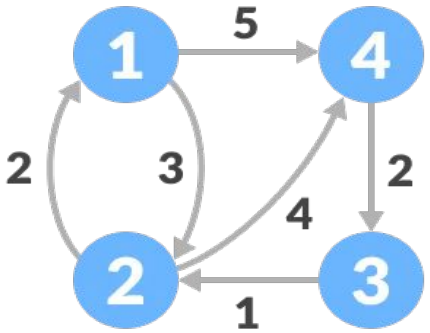
All pairs shortest paths- Flyod-Warshall Algorithm



$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

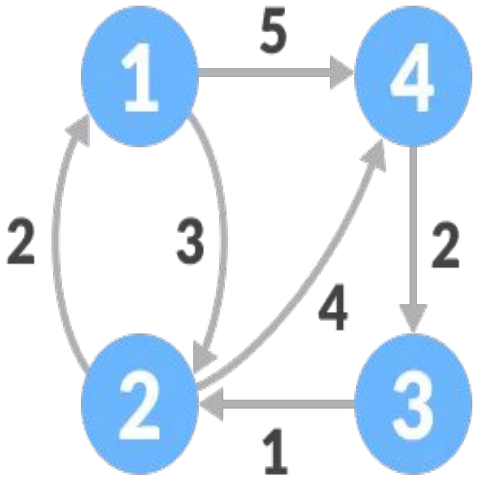
Calculate the distance from the source vertex to destination vertex through this vertex k

All pairs shortest paths- Flyod-Warshall Algorithm

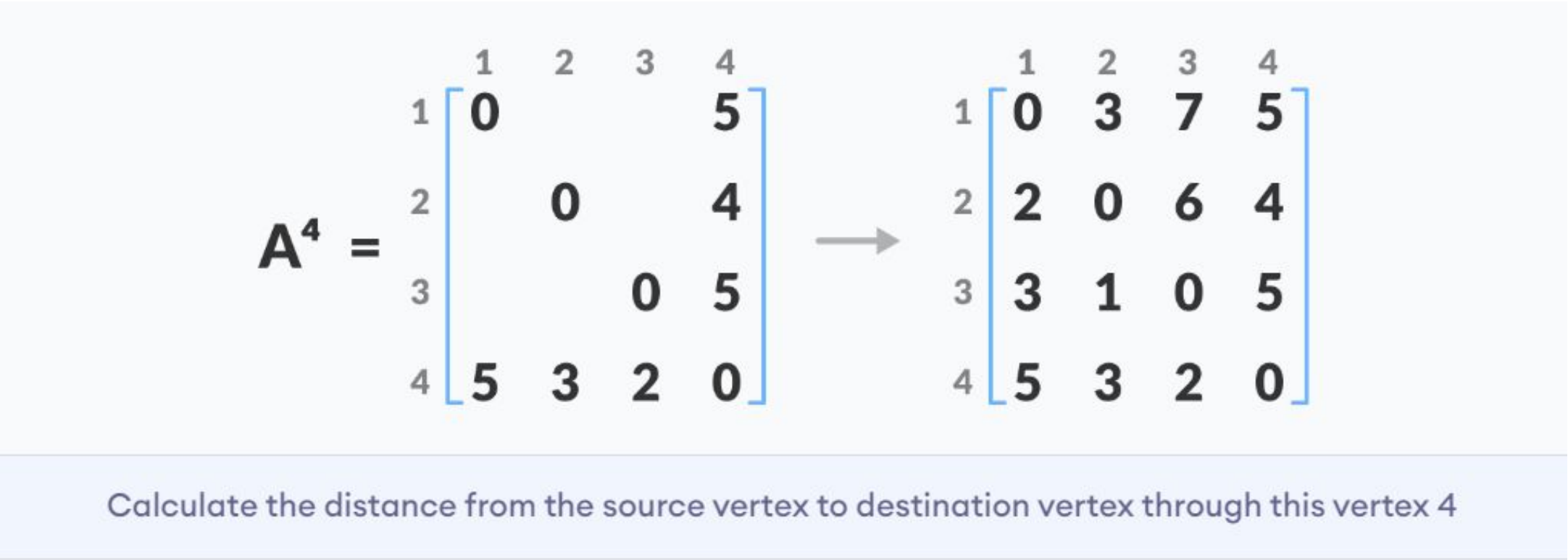


$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

All pairs shortest paths- Flyod-Warshall Algorithm



$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$



A4 gives the shortest path between each pair of vertices.

Algorithm

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.
- For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.
  - **k** is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.
  - **k** is an intermediate vertex in shortest path from **i** to **j**. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j],** if **dist[i][j] > dist[i][k] + dist[k][j]**

Floyd-Warshall Algorithm

- n = no of vertices
- A = matrix of dimension n*n

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
return A
```

Floyd Warshall Algorithm Complexity

- Time Complexity
- There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is O(n3).

- Space Complexity
- The space complexity of the Floyd-Warshall algorithm is O(n2).

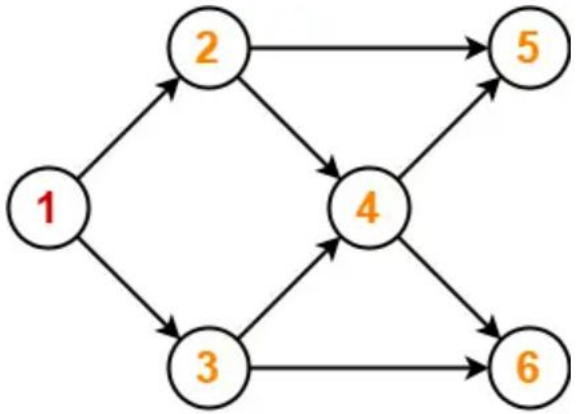
Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- For testing whether an undirected graph is bipartite

Topological Ordering(Topological Sorting)

- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.
- Topological Sorting for a graph is not possible if the graph is not a DAG.

Topological Ordering(Topological Sorting)



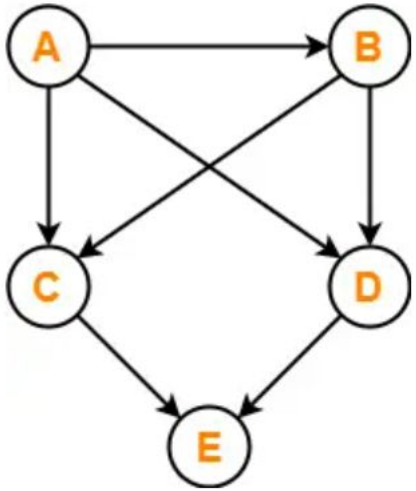
Topological Sort Example

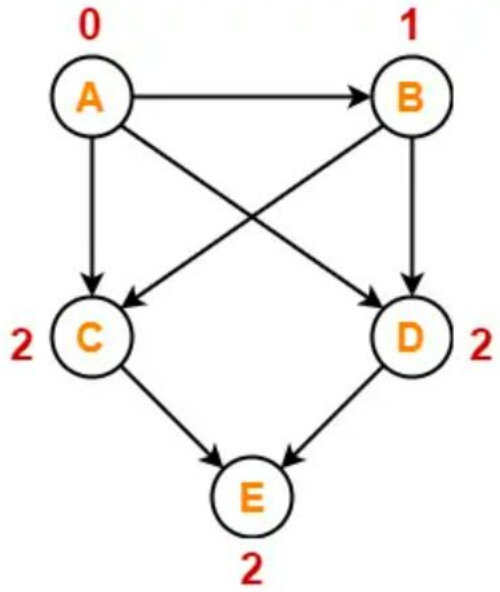
4 different topological orderings are possible-

- **1 2 3 4 5 6**
- **1 2 3 4 6 5**
- **1 3 2 4 5 6**
- **1 3 2 4 6 5**

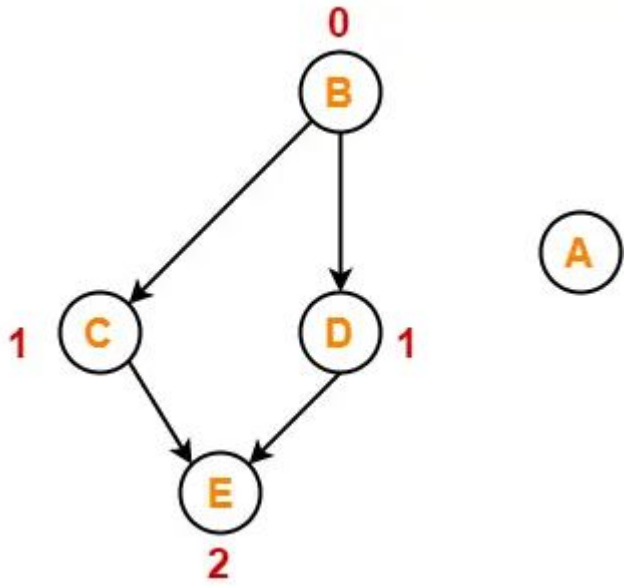
Topological Ordering -Problem

- Find the number of different topological orderings possible for the given graph-



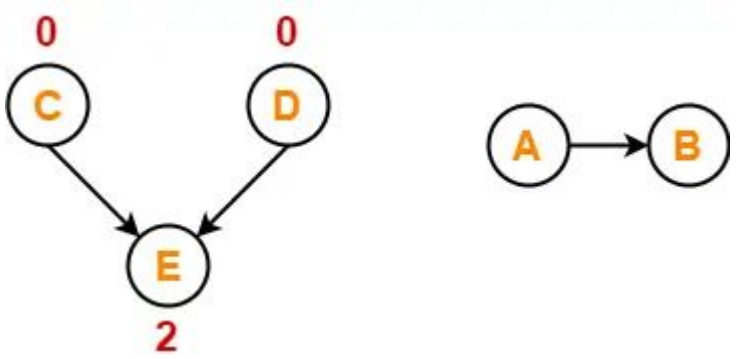


Write in-degree of each vertex-

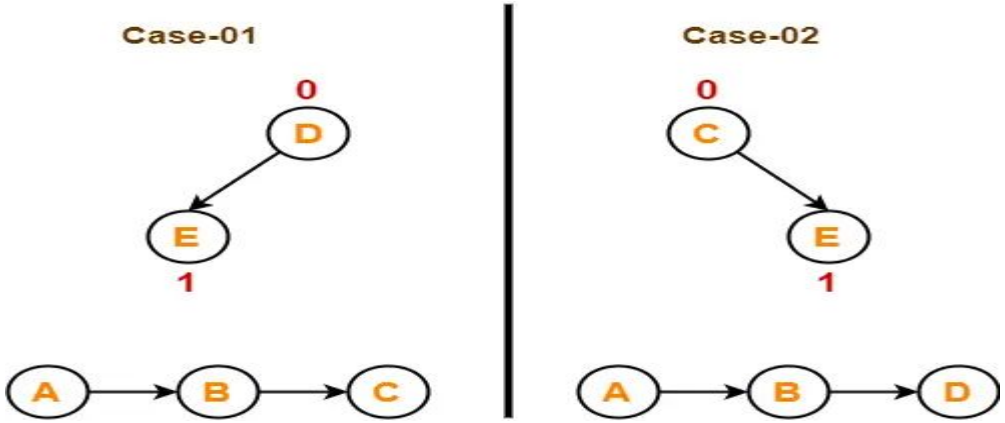


Vertex-A has the least in-degree.
So, remove vertex-A and its associated edges.
update the in-degree of other vertices.

Topological Ordering –Problem Cont…



Vertex-B has the least in-degree.
So, remove vertex-B and its associated edges.
Now, update the in-degree of other vertices.



n case-01,

Remove vertex-C and its associated edges.
Then, update the in-degree of other vertices.

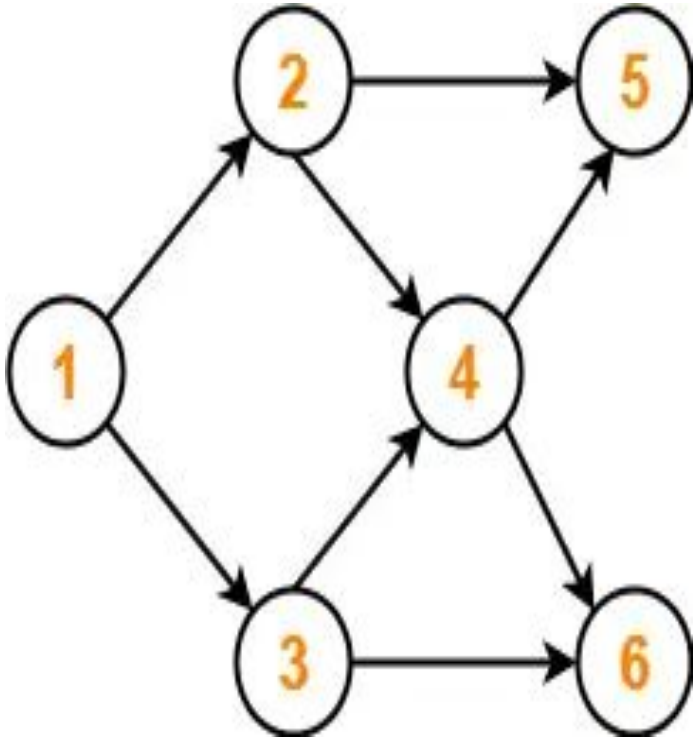
In case-02,

Remove vertex-D and its associated edges.
Then, update the in-degree of other vertices.

Topological Ordering –Problem Cont…

2 different topological orderings are possible-
A B C D E
A B D C E

Topological Ordering –Problem 2



4 different topological orderings are possible-

1 2 3 4 5 6
1 2 3 4 6 5
1 3 2 4 5 6
1 3 2 4 6 5