MC4101

UNIT III GRAPHS

Elementary Graph Algorithms:

Introduction to Graphs

- Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

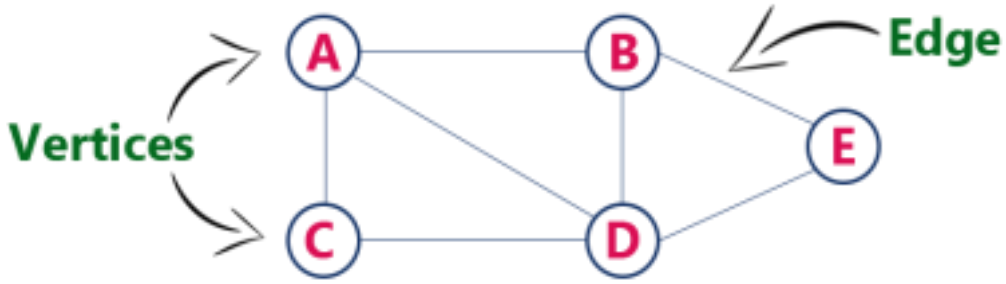
Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example

- ✓ The following is a graph with 5 vertices and 6 edges.
- ✓ This graph G can be defined as G = (V , E)Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

We use the following terms in graph data structure...

Vertex (NODE) :

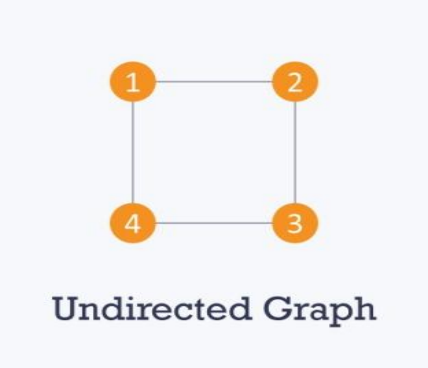
Individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge

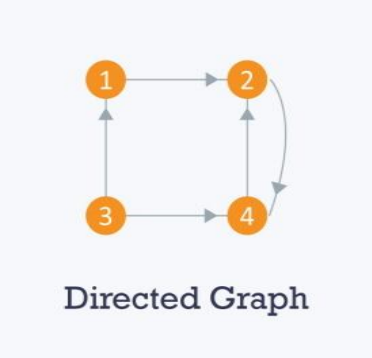
- An edge is a connecting link between two vertices.
- Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex).
- For example, in above graph the link between vertices A and B is represented as (A,B).
- In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

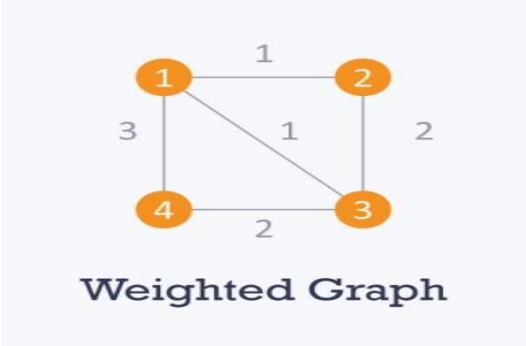
1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).



2. **Directed Edge -** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).



3. **Weighted Edge -** A weighted egde is a edge with value (cost) on it.



Undirected Graph

- A graph **with only undirected edges** is said to be undirected graph.

Directed Graph

- A graph with only directed edges is said to be directed graph.

Mixed Graph

- A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

- The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin

- If a edge is directed, its **first endpoint** is said to be the origin of it.

Destination

- If a edge is directed, its first endpoint is said to be the origin of it and the other **endpoint** is said to be the destination of that edge.

Adjacent

- If there is an edge between vertices A and B then both A and B are said to be adjacent.
- In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident

- Edge is said to be incident on a vertex if the vertex is **one of the endpoints of that edge.**

Outgoing Edge

- A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

- A directed edge is said to be incoming edge on its destination vertex.

Degree

- Total number of edges connected to a vertex is said to be degree of that vertex.

In degree

- Total number of incoming edges connected to a vertex is said to be in degree of that vertex.

Out degree

- Total number of outgoing edges connected to a vertex is said to be out degree of that vertex.

Parallel edges or Multiple edges

- If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

- Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

- A graph is said to be simple if there are no parallel and self-loop edges.

Path

- A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Representations of Graphs

Graph data structure is represented using following representations...

1. **Adjacency Matrix**

2. Incidence Matrix
3. Adjacency List

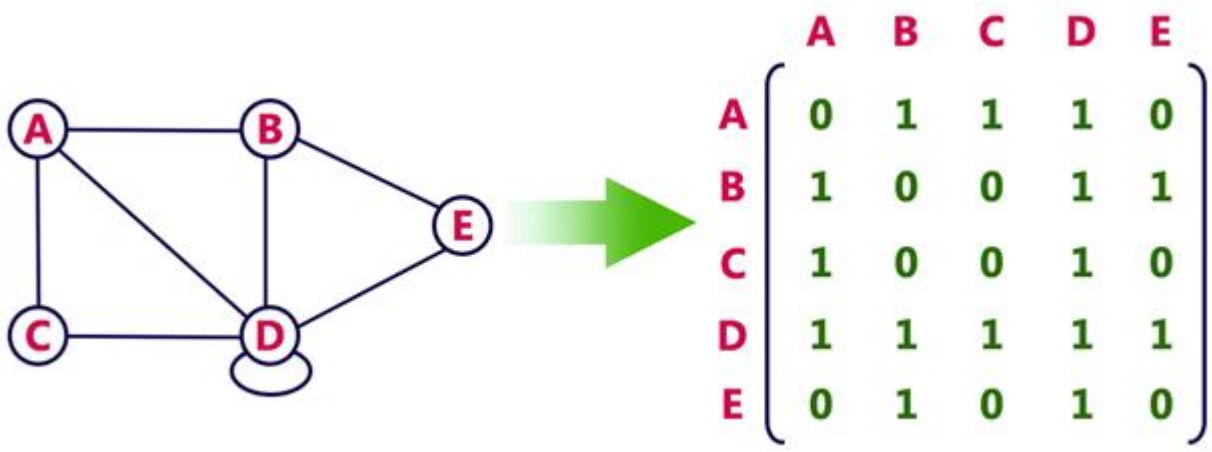
1. Adjacency Matrix

- ✓ Adjacency matrix is a sequential representation.
- ✓ It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- ✓ In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$.

Example

Consider the following undirected graph representation:

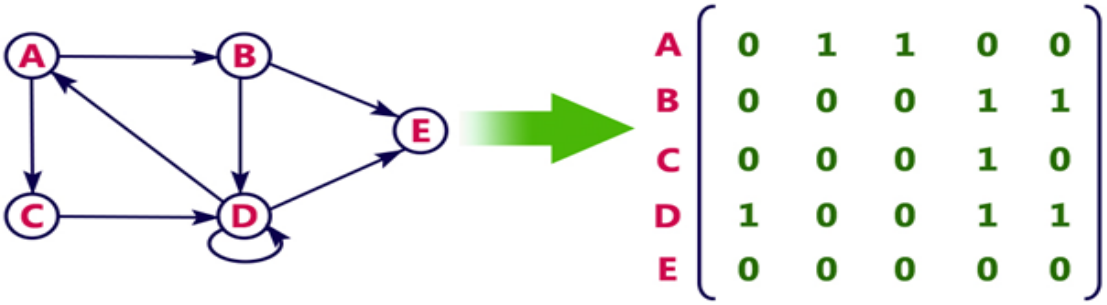
Undirected graph representation



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Directed graph represenation

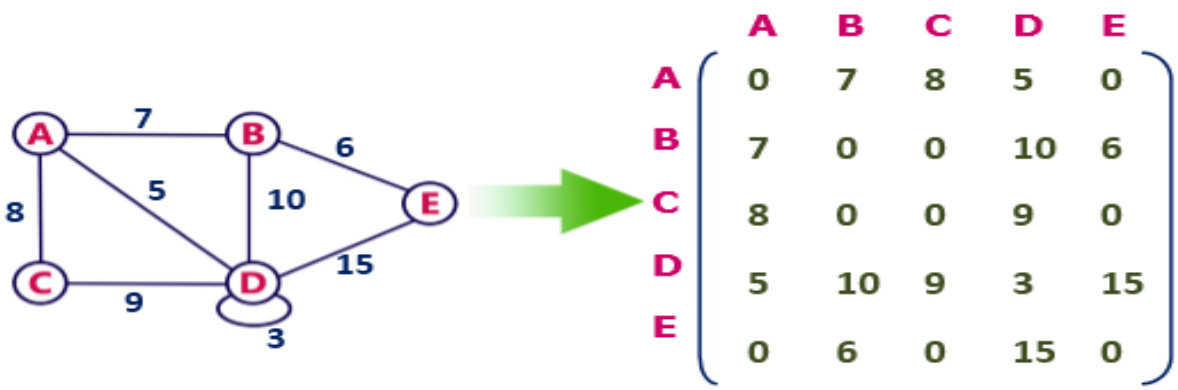
See the directed graph representation:



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

Undirected weighted graph represenation



Pros: Representation is easier to implement and follow.

Cons: It takes a lot of space and time to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

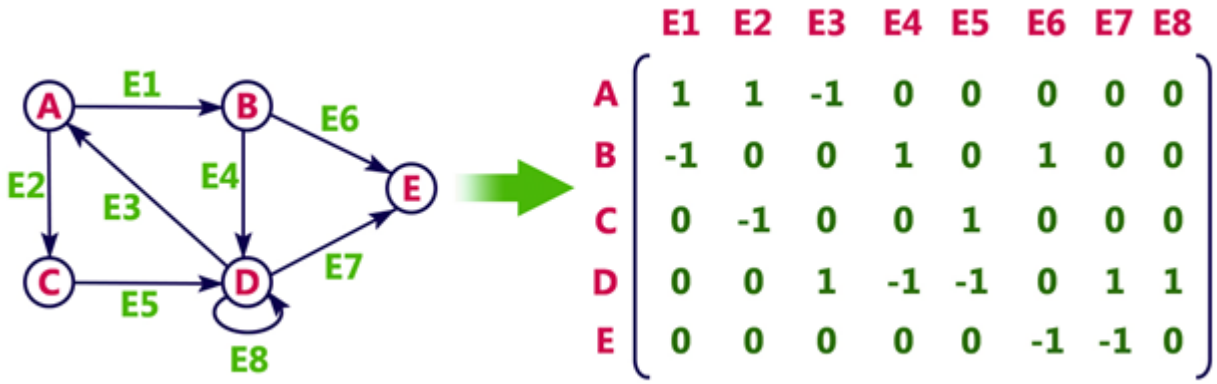
2) Incidence Matrix

In Incidence matrix representation, graph can be represented using a matrix of size:

- Total number of vertices by total number of edges.
- It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class. In this matrix, columns represent edges and rows represent vertices.
- This matrix is filled with either 0 or 1 or -1. Where,
- 0 is used to represent row edge which is not connected to column vertex.
- 1 is used to represent row edge which is connected as outgoing edge to column vertex.
- -1 is used to represent row edge which is connected as incoming edge to column vertex.

Example

Consider the following directed graph representation.

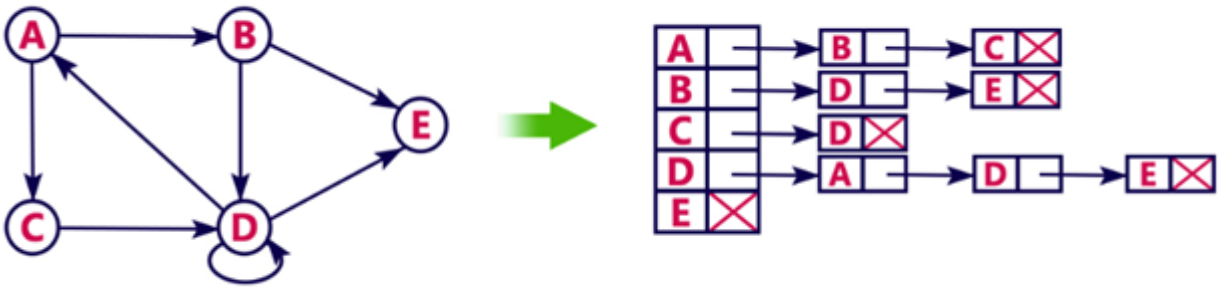


3) Adjacency List

- **Adjacency list is a linked representation.**
- **In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.**
- **We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbors of v.**

Example

Let's see the following directed graph representation implemented using linked list:



Pros:

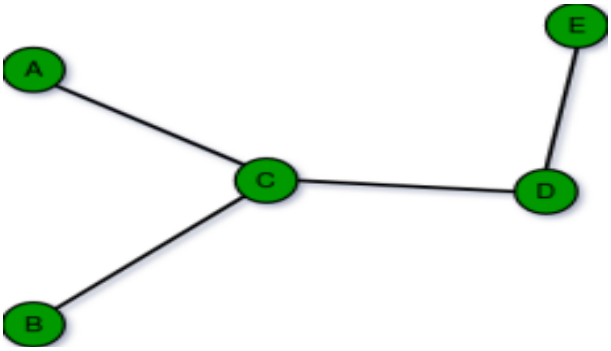
- **Adjacency list saves lot of space.**
- **We can easily insert or delete as we use linked list.**
- **Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.**

Cons:

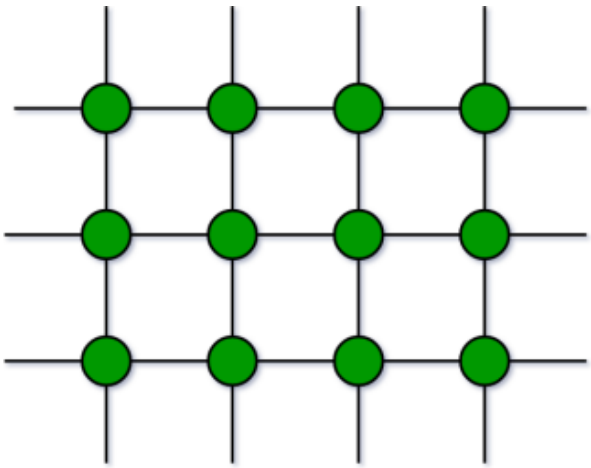
- **The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.**

Graph Types

- **Finite Graphs: A graph is said to be finite if it has finite number of vertices and finite number of edges.**

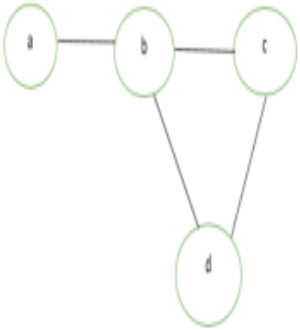


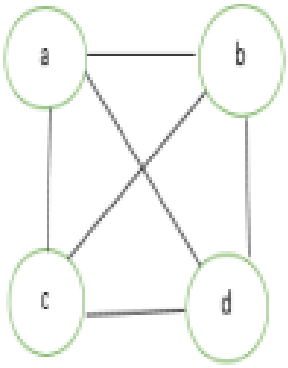
- **Infinite Graph: A graph is said to be infinite if it has infinite number of vertices as well as infinite number of edges.**



- **Trivial Graph: A graph is said to be trivial if a finite graph contains only one vertex and no edge.**

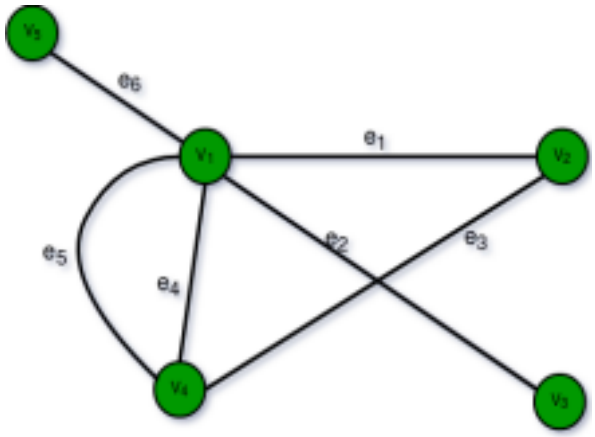
- **Simple Graph: A simple graph is a graph which does not contains more than one edge between the pair of vertices. A simple railway tracks connecting different cities is an example of simple graph.**





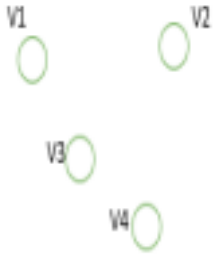
- **Multi Graph: Any graph which contain some parallel edges but doesn't contain any self-loop is called multi graph. For example A Road Map.**

Loop: An edge of a graph which join a vertex to itself is called loop or a self-loop.

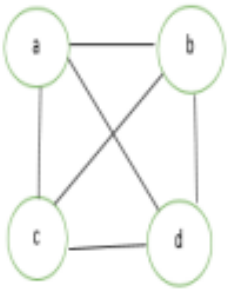


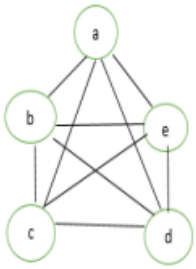
- **Parallel Edges: If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.**

- **Null Graph: A graph of order n and size zero that is a graph which contain n number of vertices but do not contain any edge.**

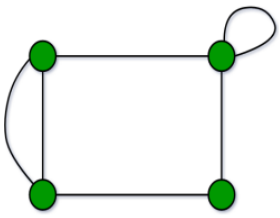


- **Complete Graph: A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.**





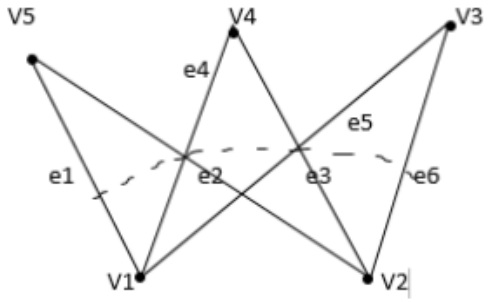
- **Pseudo Graph: A graph G with a self loop and some multiple edges is called pseudo graph.**



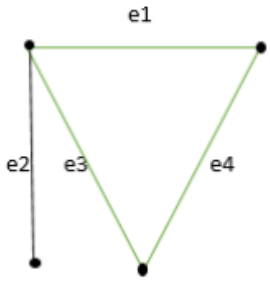
Regular Graph: A simple graph is said to be regular if all vertices of a graph G are of equal degree. All complete graphs are regular but vice versa is not possible.

Bipartite Graph: A graph G = (V, E) is said to be bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets.

V1(G) and V2(G) in such a way that each edge e of E(G) has its one end in V1(G) and other end in V2(G). The partition V1 U V2 = V is called Bipartite of G.Here in the figure: V1(G)={V5, V4, V3} V2(G)={V1, V2}

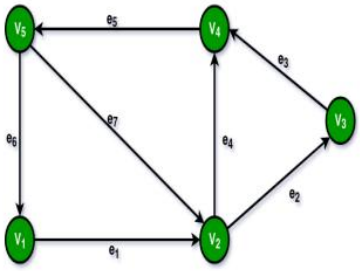


Labelled Graph: If the vertices and edges of a graph are labelled with name, data or weight then it is called labelled graph. It is also called *Weighted Graph*.

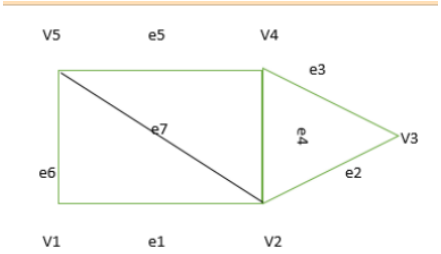


Digraph Graph: A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj.

Here in the figure:e1 = (V1, V2)e2 = (V2, V3)e4 = (V2, V4)

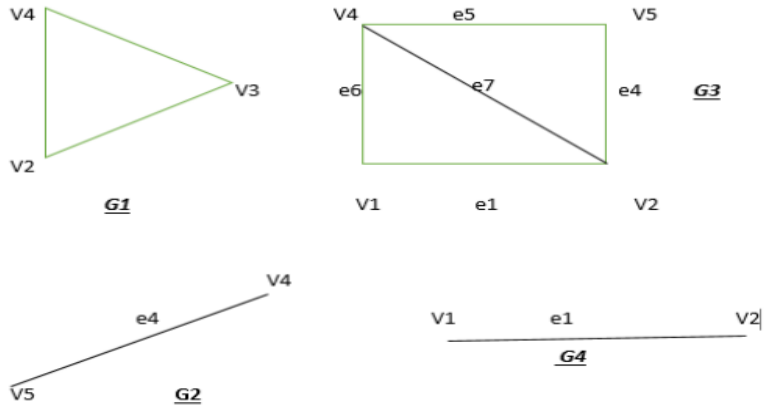


Subgraph: A graph G = (V1, E1) is called subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



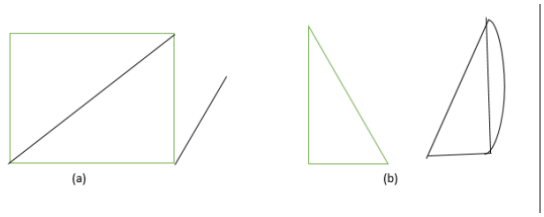
Types of Subgraph:

- **Vertex disjoint subgraph: Any two graph G1 = (V1, E1) and G2 = (V2, E2) are said to be vertex disjoint of a graph G = (V, E) if V1(G1) intersection V2(G2) = null. In figure there is no common vertex between G1 and G2.**
- **Edge disjoint subgraph: A subgraph is said to be edge disjoint if E1(G1) intersection E2(G2) = null. In figure there is no common edge between G1 and G2.**
- **Note: Edge disjoint subgraph may have vertices in common but vertex disjoint graph cannot have common edge, so vertex disjoint subgraph will always be an edge disjoint subgraph.**

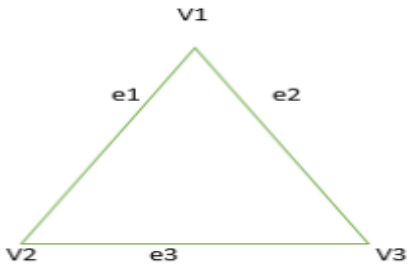


- **Connected or Disconnected Graph: A graph G is said to be connected if for any pair of vertices (Vi, Vj) of a graph G are reachable from one another. Or a graph is said to be connected if there exist atleast one path between each and every pair of vertices in graph G, otherwise it is disconnected. A null graph with n vertices is disconnected graph**

- **consisting of n components. Each component consist of one vertex and no edge.**



Cyclic Graph: A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – – – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – – – – – — -(Vn, V1) are called cyclic graph.



Application of Graphs:

- **Computer Science: In computer science, graph is used to represent networks of communication, data organization, computational devices etc.**
- **Physics and Chemistry: Graph theory is also used to study molecules in chemistry and physics.**
- **Social Science: Graph theory is also widely used in sociology.**
- **Mathematics: In this, graphs are useful in geometry and certain parts of topology such as knot theory.**
- **Biology: Graph theory is useful in biology and conservation efforts.**

Breadth-First Search – Depth-First Search

Graph Traversal - BFS

- **Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process.**
- **A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.**

There are two graph traversal techniques and they are as follows.

1. **BFS (Breadth First Search)**
2. **DFS (Depth First Search)**

BFS (Breadth First Search)

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

ALGORITHM

We use the following steps to implement BFS traversal...

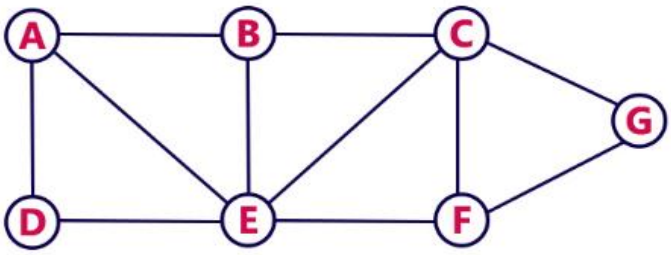
- **Step 1 - Define a Queue of size total number of vertices in the graph.**
- **Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.**
- **Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.**
- **Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.**
- **Step 5 - Repeat steps 3 and 4 until queue becomes empty.**
- **Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph**

Applications of Breadth First Traversal

- ✓ **Shortest Path and Minimum Spanning Tree** for unweighted graph
- ✓ Ford-Fulkerson algorithm
- ✓ In Garbage Collection - **Cheney's algorithm.**
- ✓ to find all **neighbor nodes in Peer to Peer** Networks (like BitTorrent
- ✓ **Crawlers** in Search Engines
- ✓ **Social Networking Websites** - find people within a given distance 'k'
- ✓ **GPS** Navigation systems:
- ✓ **Broadcasting** in Network
- ✓ **Cycle detection** in undirected graph
- ✓ To test if a graph is **Bipartite**
- ✓ Path Finding
- ✓ Finding all nodes within one connected component:

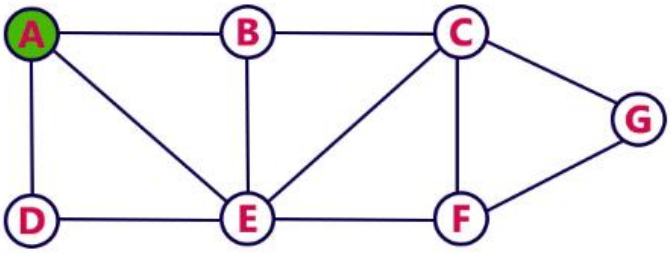
EXAMPLE

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

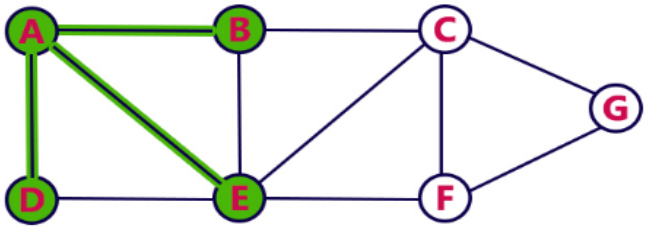


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

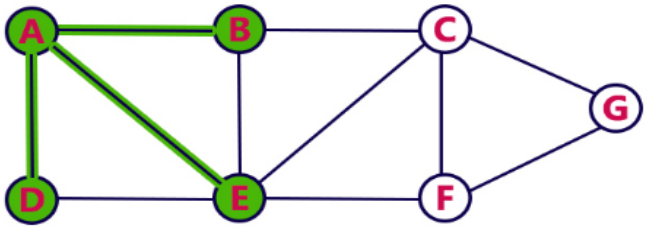


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

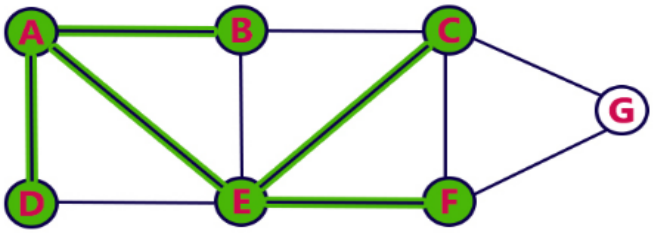


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

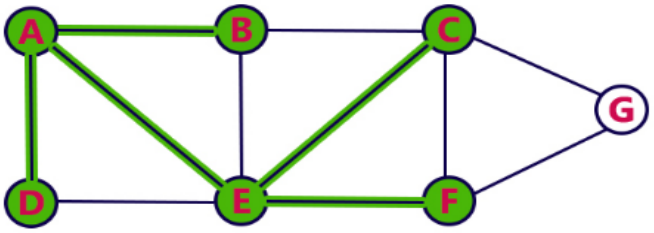


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

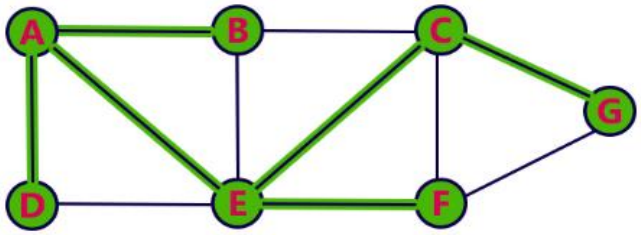


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

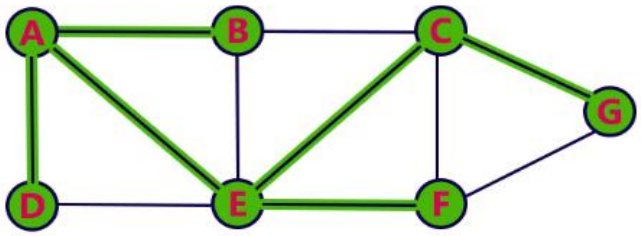


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

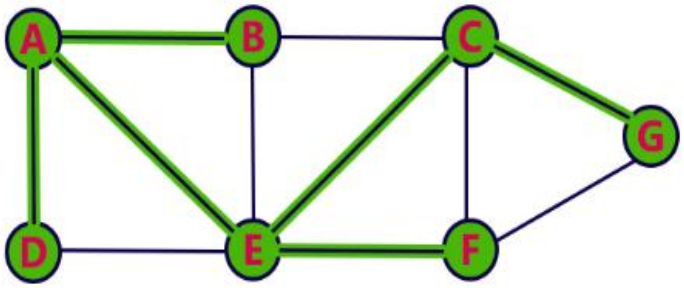


Queue

						G

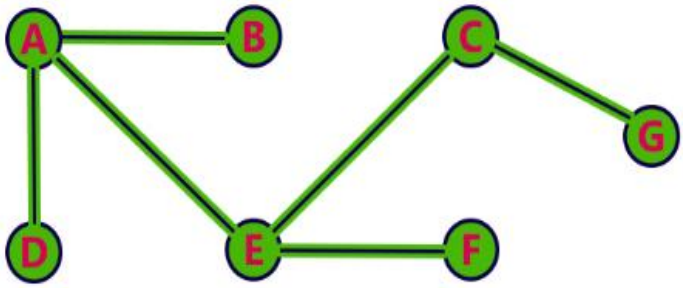
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

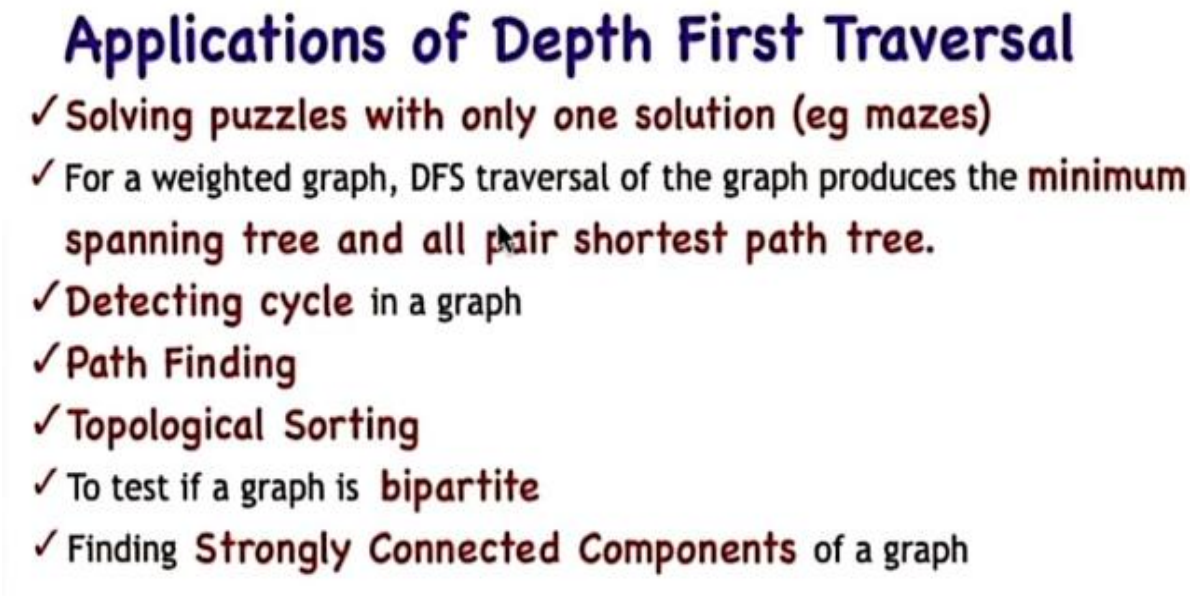


DFS (Depth First Search)

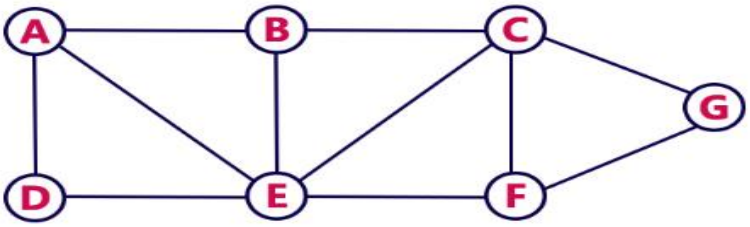
DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

- **Step 1 - Define a Stack of size total number of vertices in the graph.**
- **Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.**
- **Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.**
- **Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.**
- **Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.**
- **Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.**
- **Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph**

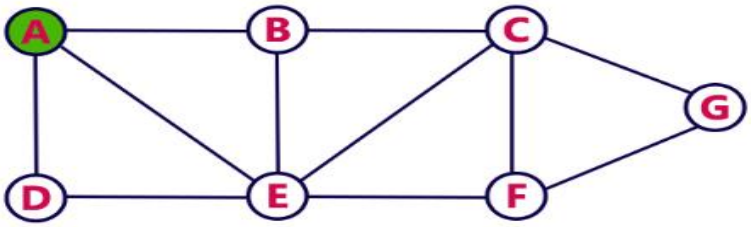


Consider the following example graph to perform DFS traversal



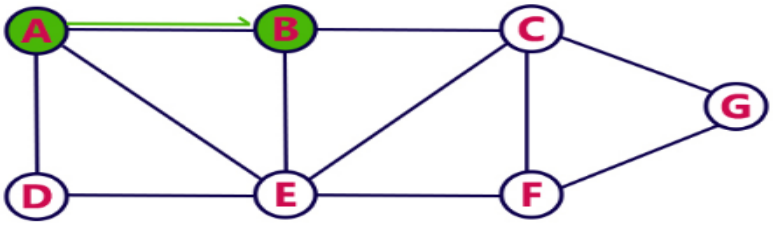
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



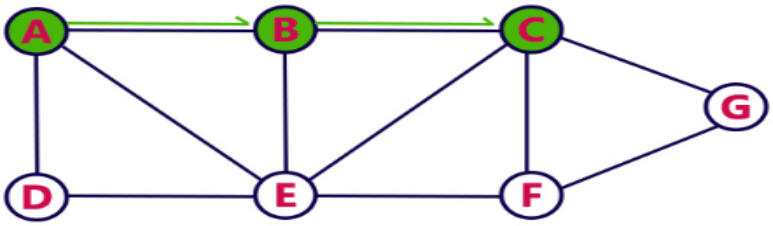
Step 2:

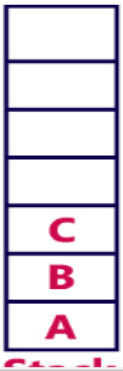
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

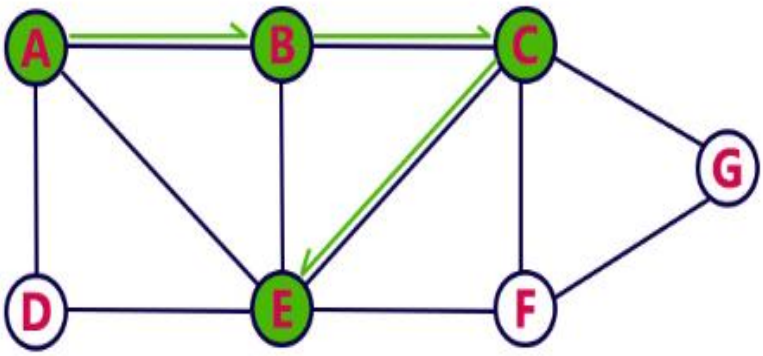
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





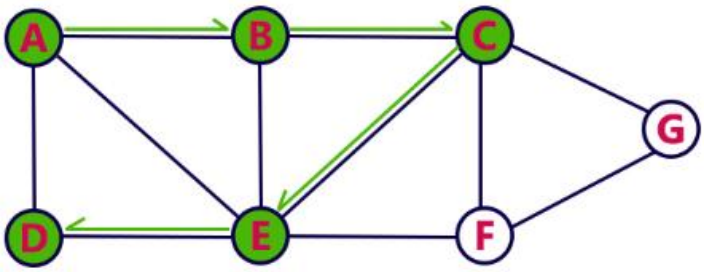
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



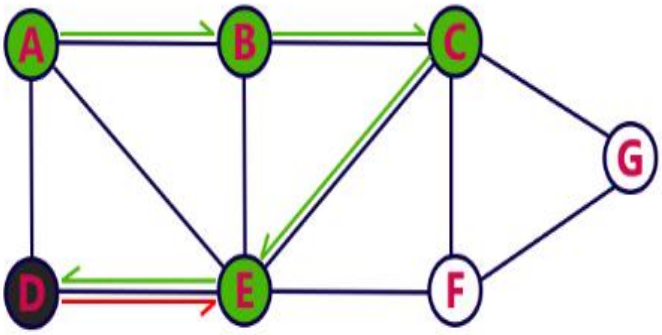
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



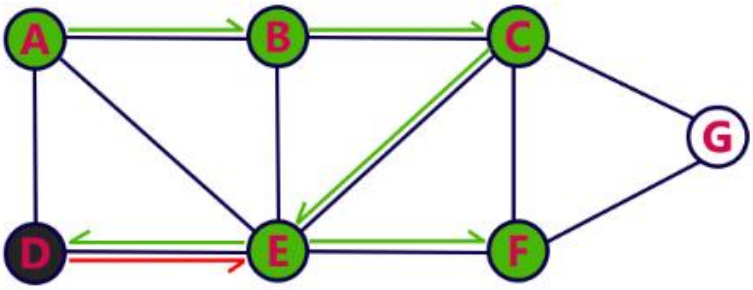
Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



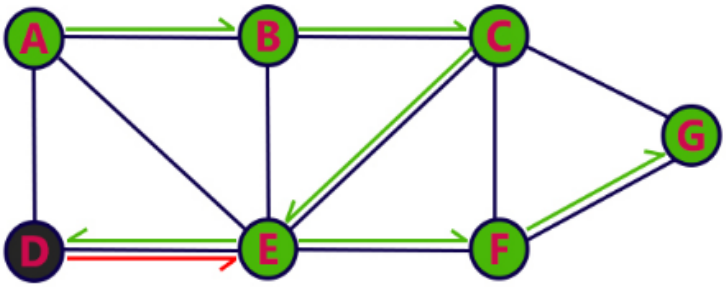
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



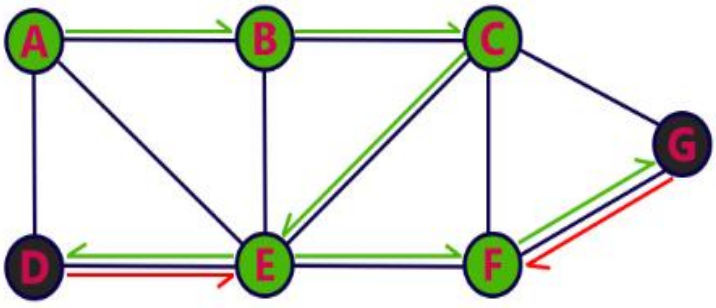
Step 8:

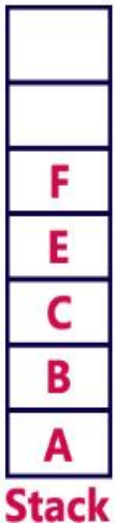
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



Step 9:

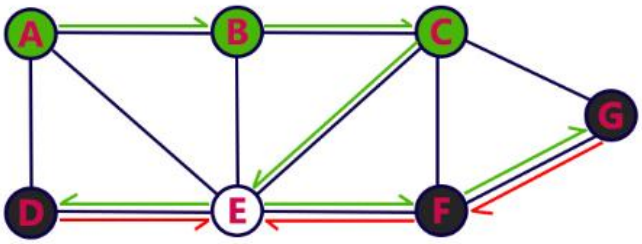
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

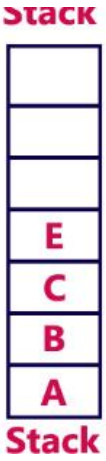




Step 10:

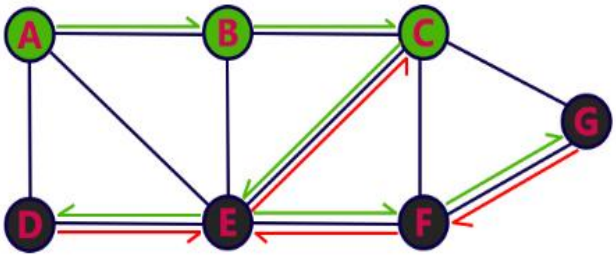
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

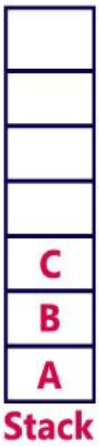




Step 11:

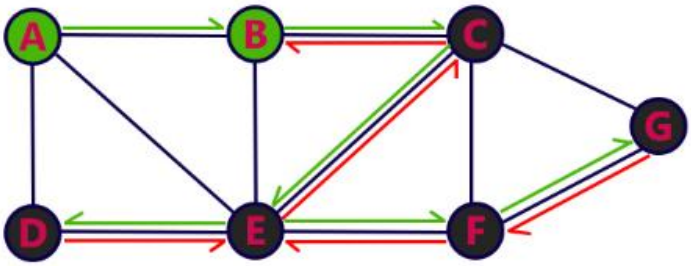
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





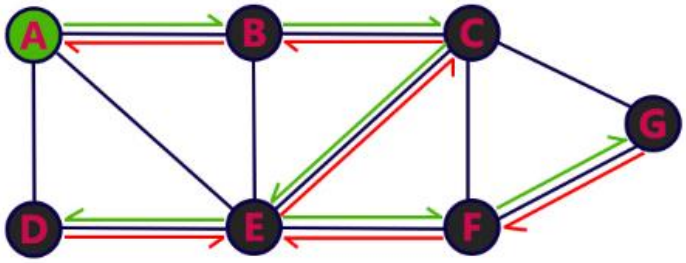
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



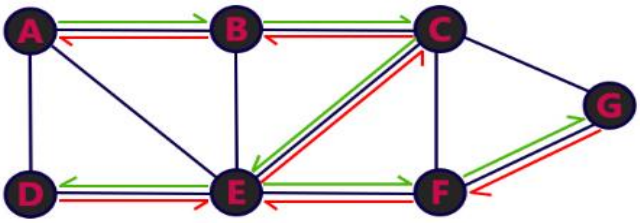
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

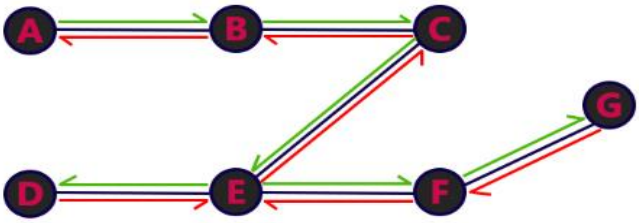


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



BFS Vs DFS

Breadth First Search (BFS) Vs Depth First Search (DFS)

BFS	DFS
Level based search	Depth Based Search
vertex based technique	edge based technique
Queue Data Structure is used	Stack Data Structure is used
BFS is more suitable for searching target vertices which are closer to the given source.	DFS is more suitable when there are solutions or targets away from source.
Not suitable for decision making trees (games, puzzles) since neighbors are considered	Suitable for decision making trees (games, puzzles)
Higher memory requirement than DFS	lower memory requirements because it's not necessary to store all of the child pointers at each level
BFS is slower	DFS is faster
Time complexity of BFS & DFS is O(V + E), where V is vertices & E is edges.	

Topological Sort-

- ✓ **Topological sort : of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.**
- ✓ **For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; topological sorting is just a valid sequence for the tasks.**
- ✓ **In topological sorting, we need to print a vertex before its adjacent vertices.**
- ✓ **Topological Sorting for a graph is not possible if the graph is not a DAG(Directed Acyclic Graph).**

Algorithm

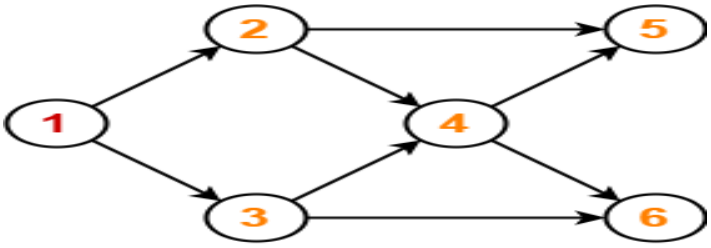
1. **Store each vertex's In-Degree in an array D**
2. **Initialize queue with all "in-degree=0" vertices**
3. **While there are vertices remaining in the queue:**
   - **(a) Dequeue and output a vertex**
   - **(b) Reduce In-Degree of all vertices adjacent to it by 1**

(c) Enqueue any of these vertices whose In-Degree became zero

4. If all vertices are output then success, otherwise there is a cycle.

Topological Sort Example-

Consider the following directed acyclic graph-



Topological Sort Example

For this graph, following 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

Applications of Topological Sort-

Few important applications of topological sort are-

- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in makefiles
- Data Serialization

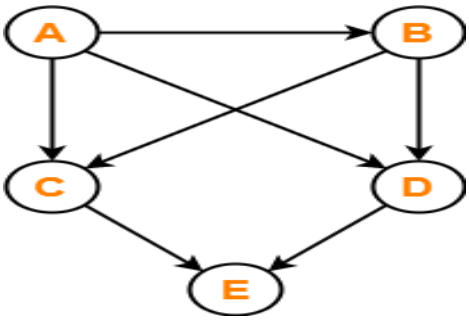
Advantages of Topological Sorting

- Topological Sorting is mostly used to schedule jobs based on their dependencies. Instruction scheduling, ordering formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linker are all examples of applications of this type in computer science.

- **Finding cycle in a graph: Only directed acyclic graphs may be ordered topologically (DAG). It is impossible to arrange a circular graph topologically.**
- **Operation System deadlock detection: A deadlock occurs when one process is waiting while another holds the requested resource.**
- **Dependency resolution: Topological Sorting has been proved to be very helpful in Dependency resolution.**
- **Critical Path Analysis: A project management approach known as critical route analysis. It's used to figure out how long a project should take and how dependent each action is on the others. There may be some preceding actions before an activity. Before beginning a new activity, all previous actions must be completed.**
- **Course Schedule problem: Topological Sorting has been proved to be very helpful in solving the Course Schedule problem.**
- **Other applications like manufacturing workflows, data serialization, and context-free grammar.**

Problem-01:

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

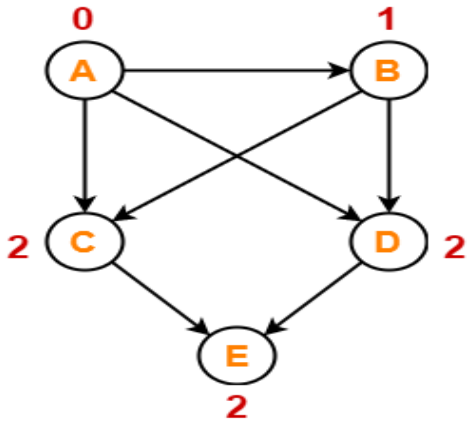


Solution-

The topological orderings of the above graph are found in the following steps-

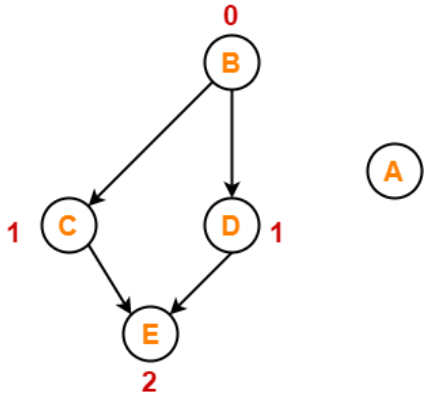
Step-01:

Write in-degree of each vertex-



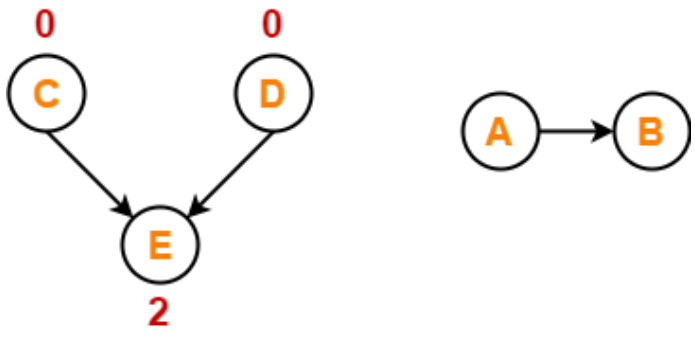
Step-02:

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



Step-03:

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



Step-04:

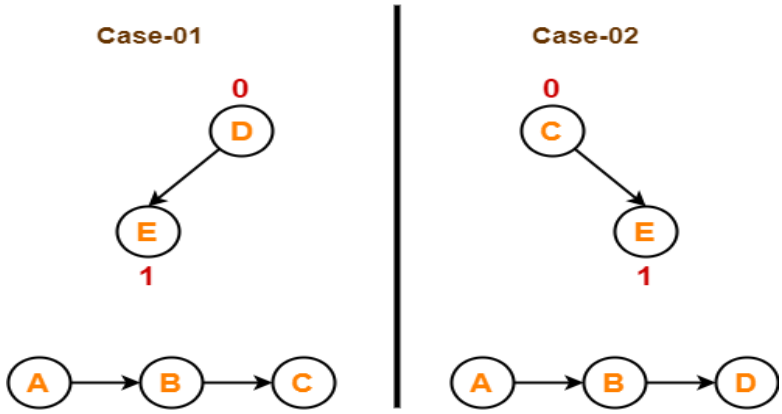
There are two vertices with the least in-degree. So, following 2 cases are possible-

In 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.**



Step-05:

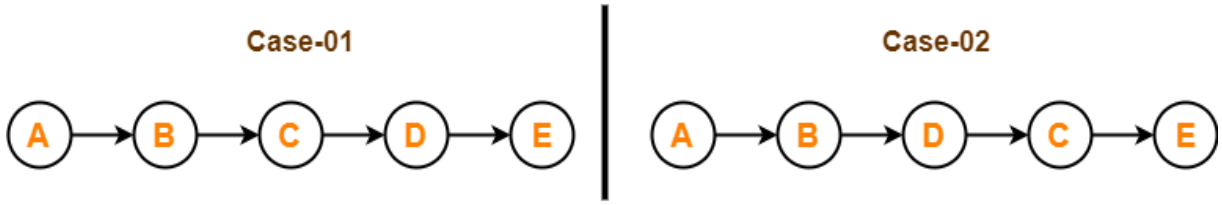
Now, the above two cases are continued separately in the similar manner.

In case-01,

- **Remove vertex-D since it has the least in-degree.**
- **Then, remove the remaining vertex-E.**

In case-02,

- **Remove vertex-C since it has the least in-degree.**
- **Then, remove the remaining vertex-E.**



Conclusion-

For the given graph, following 2 different topological orderings are possible-

- **A B C D E**
- **A B D C E**

PROGRAM:

```
include<stdio.h>

#include<conio.h>

int main(){

int

i,j,k,n,a[10][10],indeg[10],flag[10],count=

0;  printf("Enter the no of vertices:\n");

scanf("%d",&n);

printf("Enter the adjacency matrix:\n");
for(i=0;i<n;i++){

printf("Enter row %d\n",i+1);

for(j=0;j<n;j++)

scanf("%d",&a[i][j]);

}

for(i=0;i<n;i++){

 indeg[i]=0;

 flag[i]=0; }

 for(i=0;i<n;i++)

 for(j=0;j<n;j++)
```

```
 indeg[i]=indeg[i]+a[j][i];

printf("\nThe topological order

is:");   while(count<n){

 for(k=0;k<n;k++){

 if((indeg[k]==0) && (flag[k]==0)){

printf("%d ",(k+1));

 flag [k]=1;

 }

 for(i=0;i<n;i++){

 if(a[i][k]==1)

 indeg[k]--;

 } }

 count++;
 } return 0;}
```

OUTPUT:

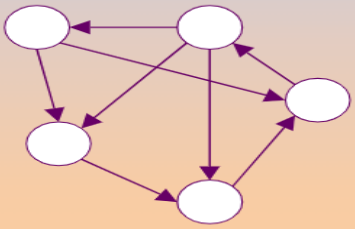
```
Enter the no of vertices:
4
Enter the adjacency matrix:
Enter row 1
0 1 1 0
Enter row 2
0 0 0 1
Enter row 3
0 0 0 1
Enter row 4
0 0 0 0

The topological order is:1 2 3 4
```

Strongly Connected Components-

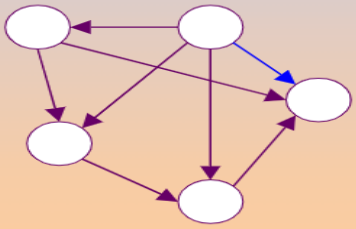
Strongly Connected Components

- Every pair of vertices are reachable from each other
- Graph *G* is ***strongly connected*** if, for every *u* and *v* in *V*, there is some path from *u* to *v* and some path from *v* to *u*.

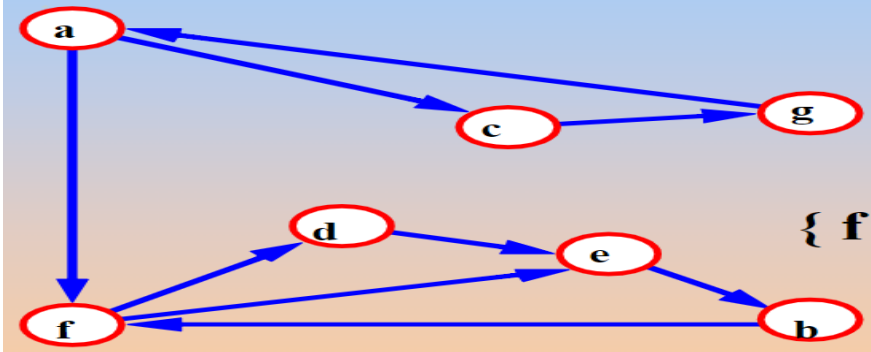
Strongly Connected



Not Strongly Connected



Example



{ a , c , g }

{ f , d , e , b }

Finding Strongly Connected Components

- Input: A directed graph G = (V,E)
- Output: a partition of V into disjoint sets so that each set defines a strongly connected component of G

Algorithm

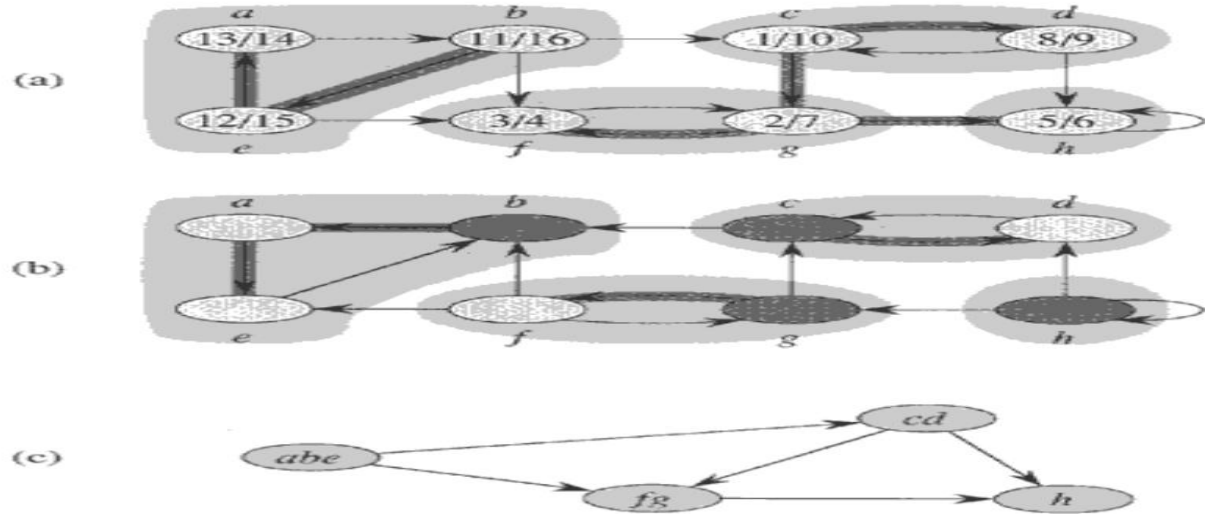
Strongly-Connected-Components(G)

1. call DFS(G) to compute finishing times f[u] for each vertex u. Cost: O(E+V)
2. compute G^T Cost: O(E+V)
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f[u] Cost: O(E+V)
4. output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component.

The graph G^T is the transpose of G, which is visualized by reversing the arrows on the digraph.

- Cost: O(E+V)

Example



Minimum Spanning Trees: Growing a Minimum Spanning Tree

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 sub graph of G (every edge in the tree belongs to G).

Minimum Spanning:

- **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.**
- **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:**

1. **Cluster Analysis**
2. **Handwriting recognition**
3. **Image segmentation**

Application of Minimum Spanning Tree

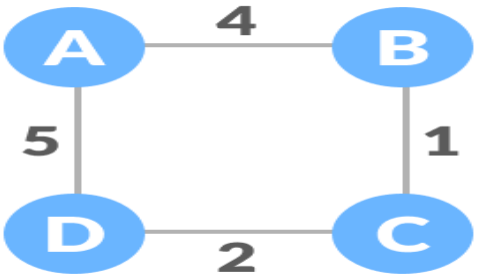
1. **Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost. The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.**
2. **Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.**
3. **Designing Local Area Networks.**
4. **Laying pipelines connecting offshore drilling sites, refineries and consumer markets.**
5. **Suppose you want to apply a set of houses with**
    - **Electric Power**
    - **Water**
    - **Telephone lines**
    - **Sewage lines**

To reduce cost, you can connect houses with minimum cost spanning trees.

Example of a Spanning Tree

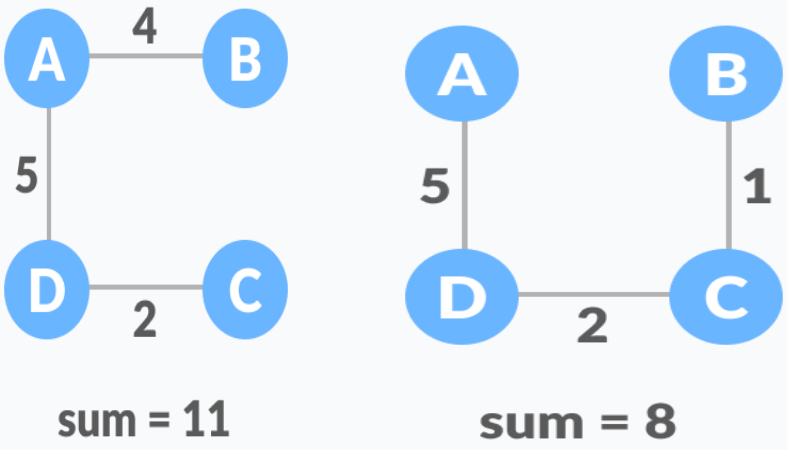
Let's understand the above definition with the help of the example below.

The initial graph is:

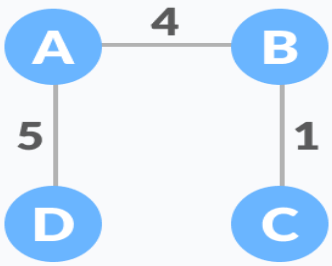


Weighted graph

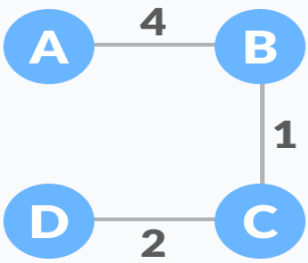
The possible spanning trees from the above graph are:



Minimum spanning tree – 1 **Minimum spanning tree – 2**



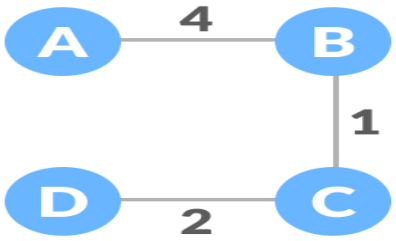
sum = 10



sum = 7

Minimum spanning tree – 3 **Minimum spanning tree - 4**

The minimum spanning tree from the above spanning trees is:



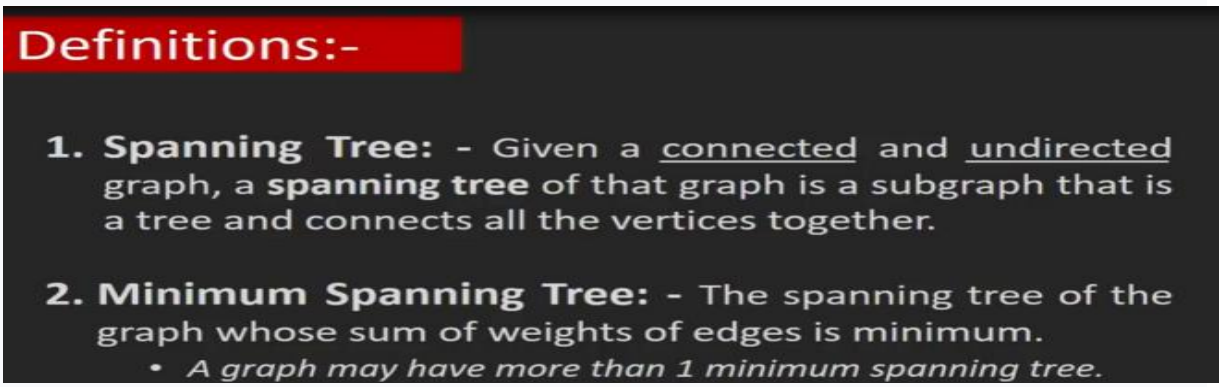
sum = 7

Minimum spanning tree

The minimum spanning tree from a graph is found using the following algorithms:

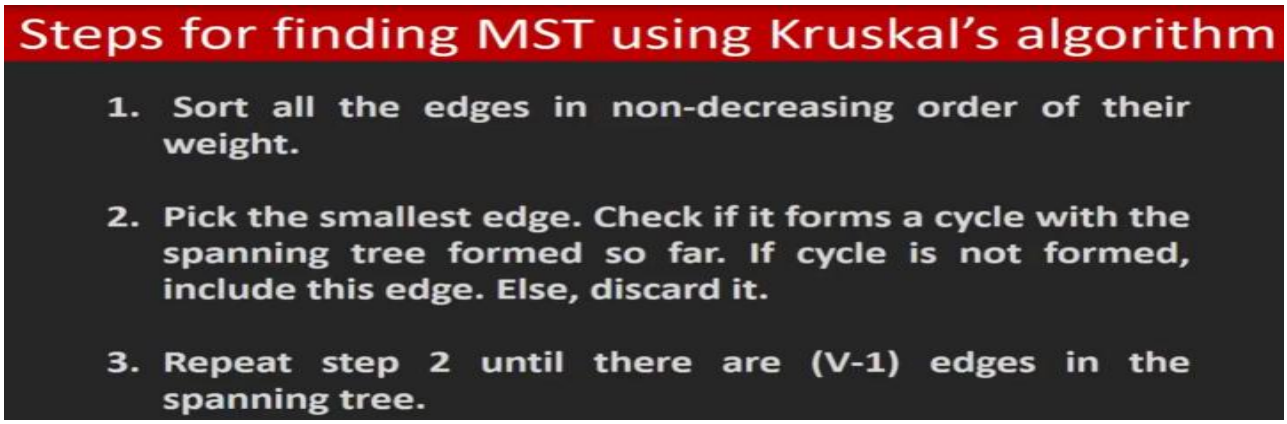
1. Prim's Algorithm
2. Kruskal's Algorithm

Kruskal's Algorithm



- ✔ Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.
- ✔ The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.
- ✔ Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.
- ✔ Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph.

The Kruskal's algorithm is given as follows.



How Kruskal's algorithm works

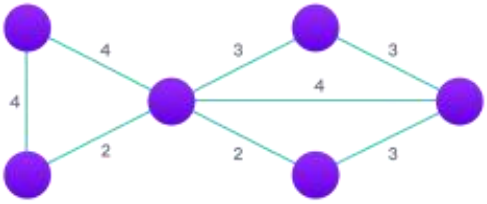
- ❖ It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
- ❖ We start from the edges with the lowest weight and keep adding edges until we reach our goal.
- ❖ The steps for implementing Kruskal's algorithm are as follows:

o Sort all the edges from low weight to high

o Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

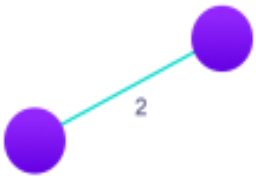
o Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



Step: 1

Start with a weighted graph

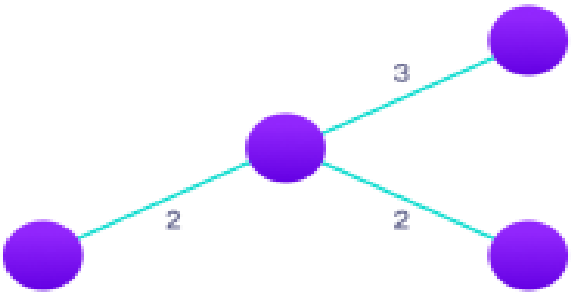


Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone

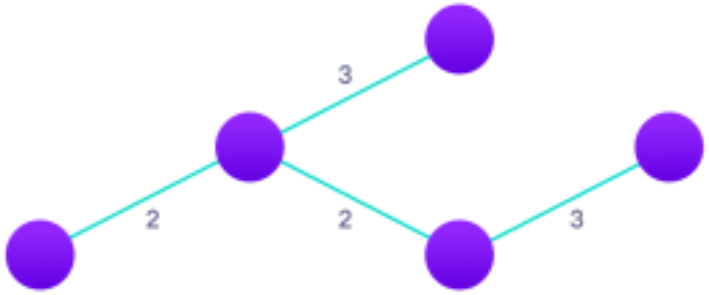
Step: 3

Choose the next shortest edge and add it



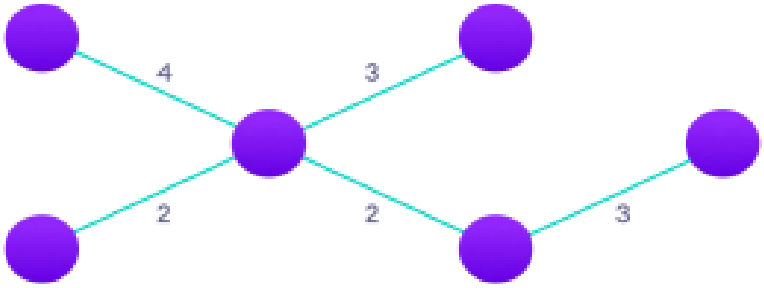
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

- Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.
- The most common way to find this out is an algorithm called Union Find. The Union Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

KRUSKAL(G):

A = ∅

For each vertex v ∈ G.V:

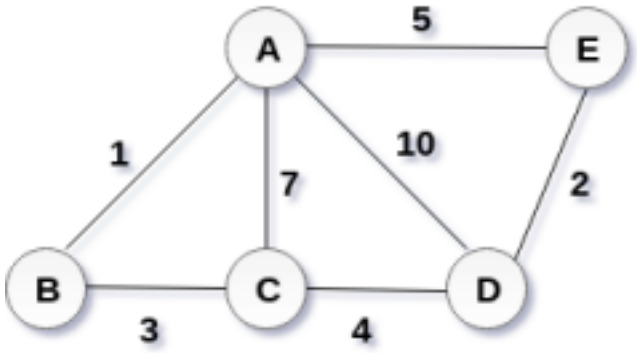
MAKE-SET(v)

For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v): if FIND-SET(u) ≠ FIND-SET(v):

A = A ∪ {(u, v)}

UNION(u, v)
return A

Example : Apply the Kruskal's algorithm on the graph given as follows.



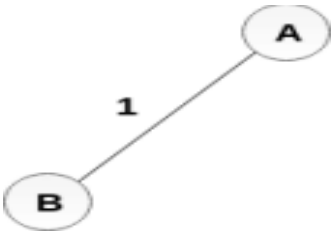
Solution: the weight of the edges given as :

Edge	AE	AD	AC	AB	BC	CD	DE
Weight	5	10	7	1	3	4	2

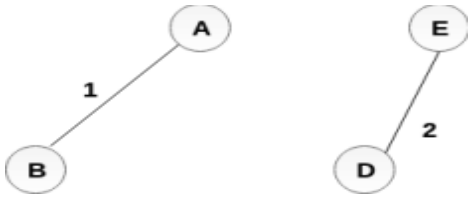
Sort the edges according to their weights.

Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

<u>Start constructing the tree;</u>

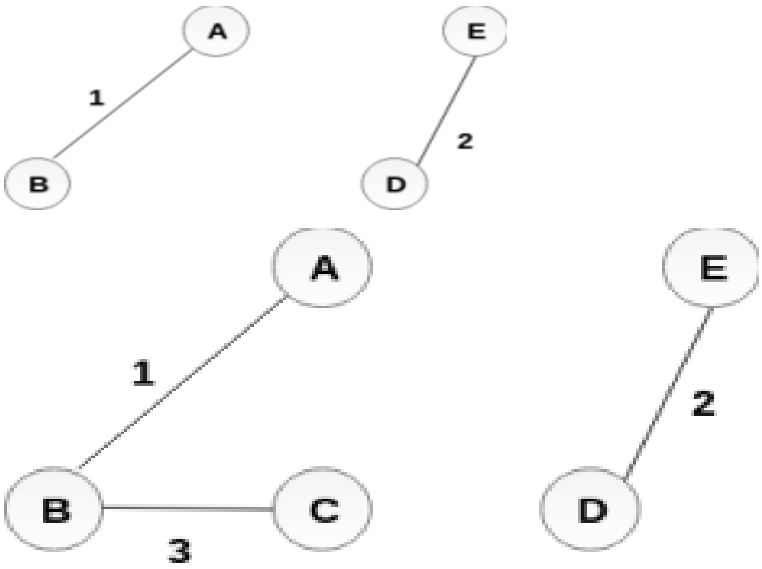


Step1: Add AB to the MST;



Step2: Add DE to the MST;

.



Step3: Add BC to the MST

- **The next step is to add AE, but we can't add that as it will cause a cycle.**
- **The next edge to be added is AC, but it can't be added as it will cause a cycle.**
- **The next edge to be added is AD, but it can't be added as it will contain a cycle.**

Hence, the final MST is the one which is shown in the step 4.

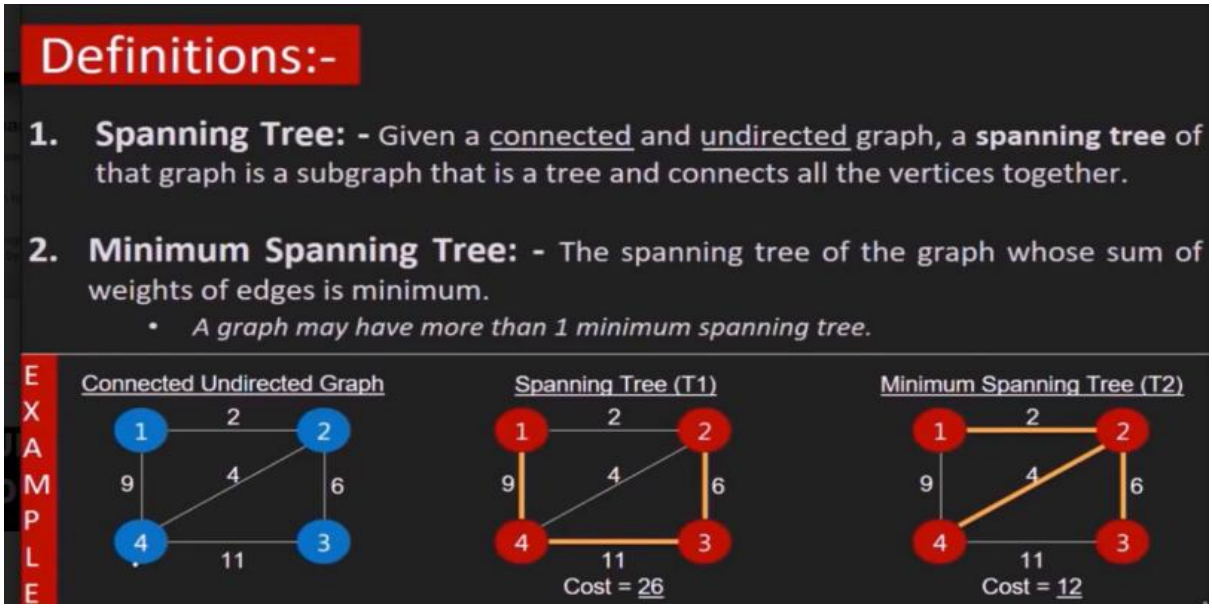
Edge	AB	DE	BC	CD
Weight	1	2	3	4

Kruskal's Algorithm Applications

- ✔ In order to layout electrical wiring
- ✔ In computer network (LAN connection)

Prim's Algorithm

Prim's Algorithm:

- **Prim's Algorithm is a famous greedy algorithm.**
- **It is used for finding the Minimum Spanning Tree (MST) of a given graph.**
- **To apply Prim's algorithm, the given graph must be weighted, connected and undirected.**
- 

Prim's Algorithm Implementation-

The implementation of Prim's Algorithm is explained in the following steps-

Step1

- **Randomly choose any vertex.**
- **The vertex connecting to the edge having least weight is usually selected.**

Step-2

- **Find all the edges that connect the tree to new vertices.**
- **Find the least weight edge among those edges and include it in the existing tree.**
- **If including that edge creates a cycle, then reject that edge and look for the next least weight edge.**

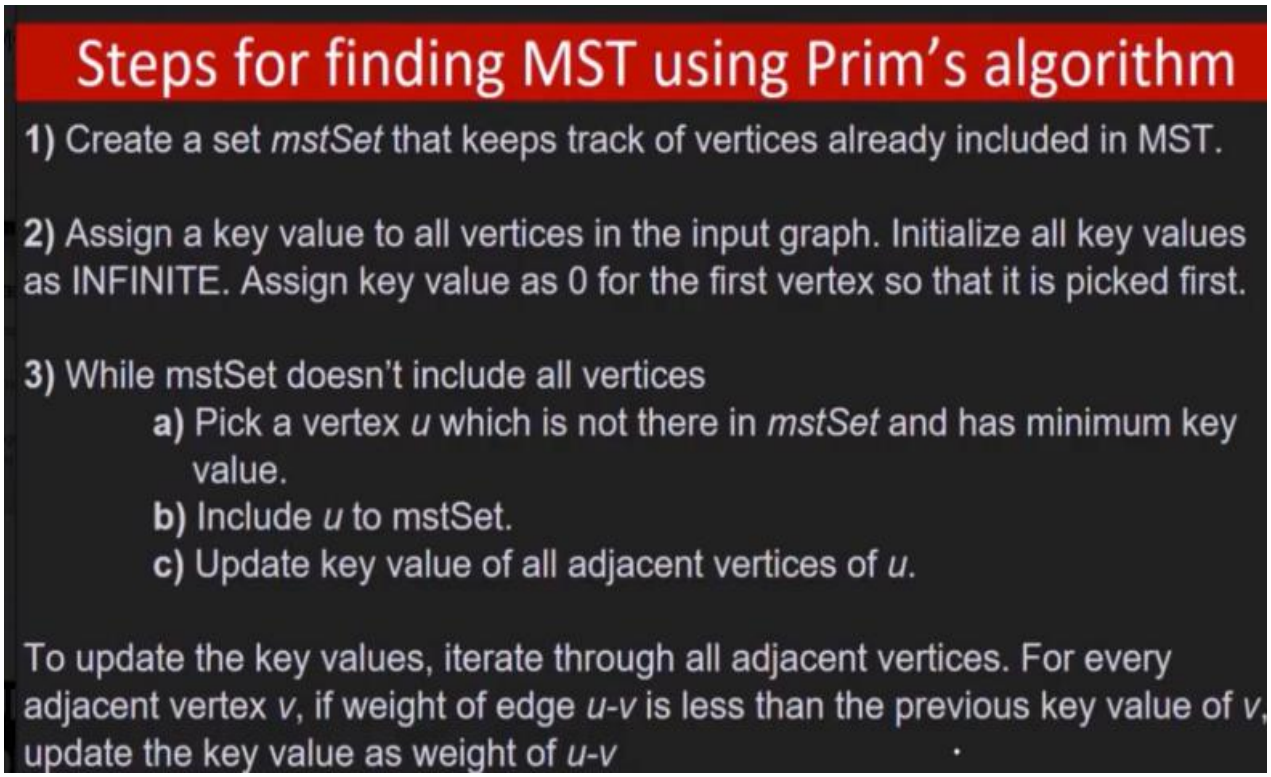
Step-03:

- **Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.**

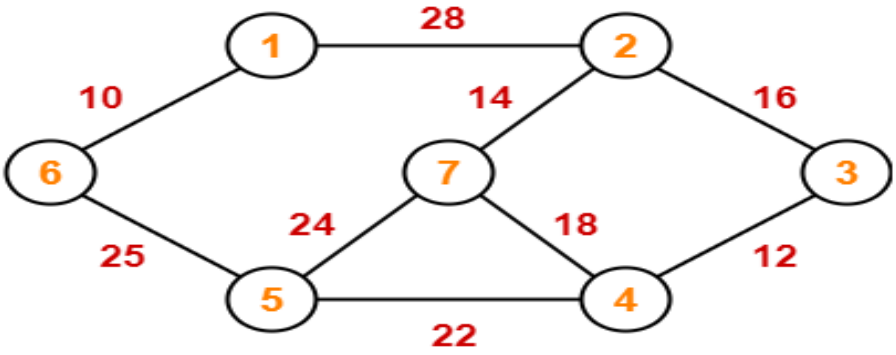
Prim's Algorithm Time Complexity-

Worst case time complexity of Prim's Algorithm is-

- **O(ElogV) using binary heap**
- **O(E + VlogV) using Fibonacci heap**

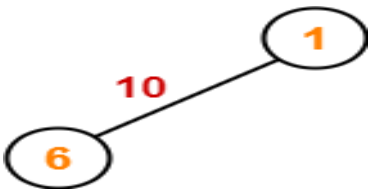


EXAMPLE: Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-

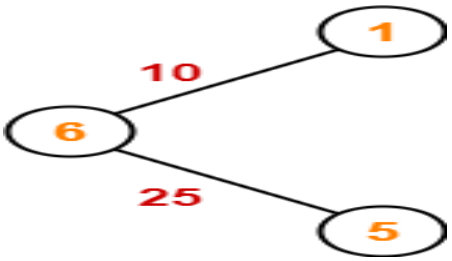


Solution- **The above discussed steps are followed to find the minimum cost spanning tree using Prim's Algorithm-**

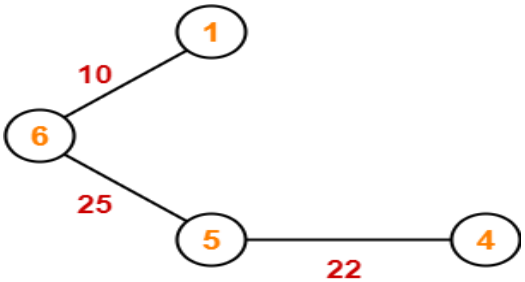
Step-01:



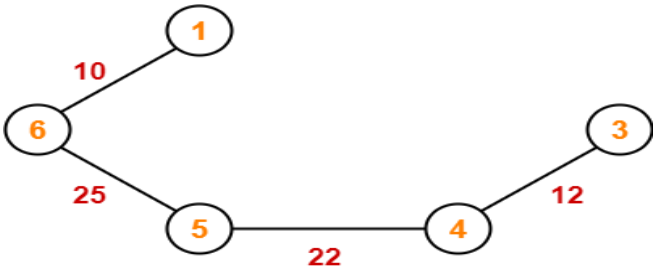
Step-02:



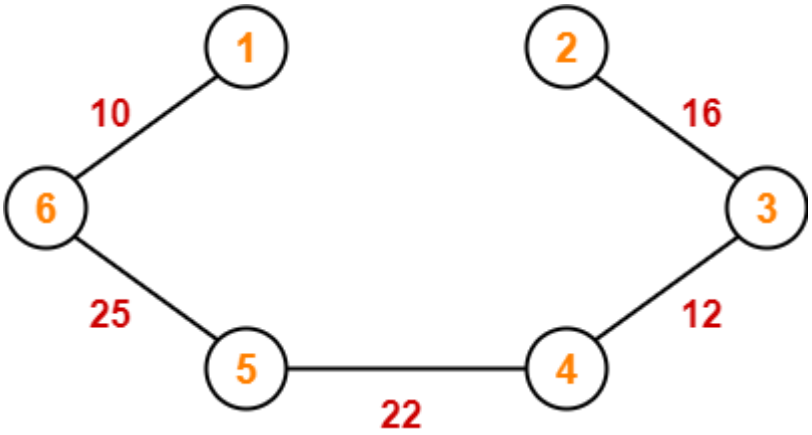
Step-03:



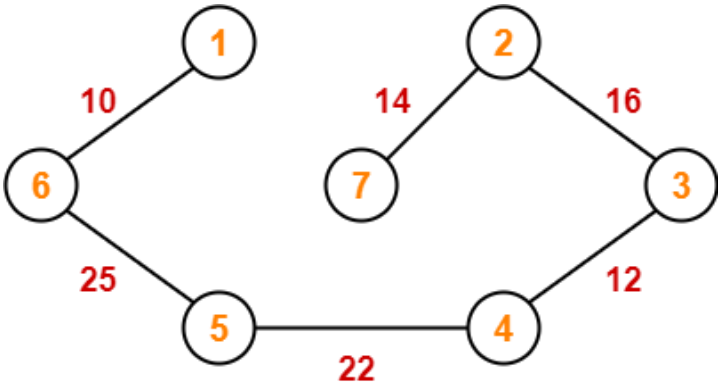
Step-04:



Step-05:



Step-06:



Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

3.11.3 Difference between Prim's And Kruskal's Algorithm

Sr. No.	Prims Algorithm	Kruskal's Algorithm
1.	Prim's algorithm initializes with node	Kriskal's algorithm initializes with edge
2.	In Prim's algorithm the next node is always selected from previously selected node.	In Kruskal's algorithm the minimum weight edge is selected independent of earlier selection.
3.	In Prim's algorithm graph must be connected.	The Kruskal's algorithm can work with disconnected graph.
4.	The time complexity of Prim's algorithm is V^2.	The time complexity of Kruskal's algorithm is O(logV).

To gain better understanding about Difference between Prim's and Kruskal's Algorithm.

Single-Source Shortest Path Problem-

- **It is a shortest path problem where the shortest path from a given source vertex to all other remaining vertices is computed.**

- **Dijkstra's Algorithm and Bellman Ford Algorithm are the famous algorithms used for solving single-source shortest path problem.**

Introduction:

In a shortest- paths problem, we are given a weighted, directed graphs G = (V, E), with weight function w: E → R mapping edges to real-valued weights. The weight of path p = (v_0,v_1,..... v_k) is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The shortest path from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The breadth-first- search algorithm is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

In a Single Source Shortest Paths Problem, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.

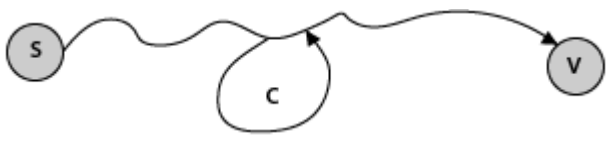
Variants:

There are some variants of the shortest path problem.

- **Single- destination shortest - paths problem: Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.**
- **Single - pair shortest - path problem: Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also.**
- **Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.**
- **All - pairs shortest - paths problem: Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.**

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



Cost of C < 0

The Bellman-Ford algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

- ✔ **It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.**

Why would one ever have edges with negative weights in real life?

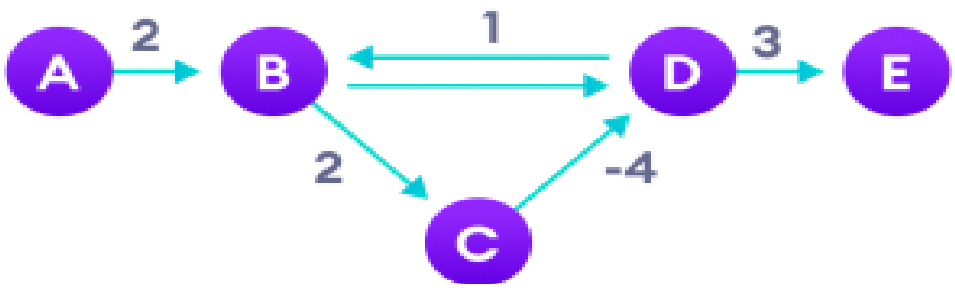
- ✔ **Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.**

- ✔ **For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.**

- ✔ **If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.**

Why do we need to be careful with negative weights?

- ✔ **Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.**



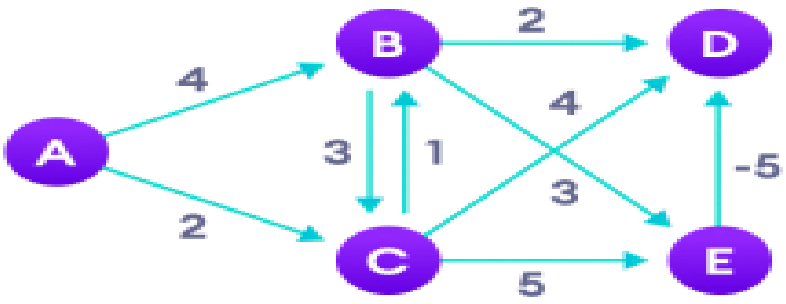
- ✔ Negative weight cycles can give an incorrect result when trying to find out the shortest path
- ✔ Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

How Bellman Ford's algorithm works

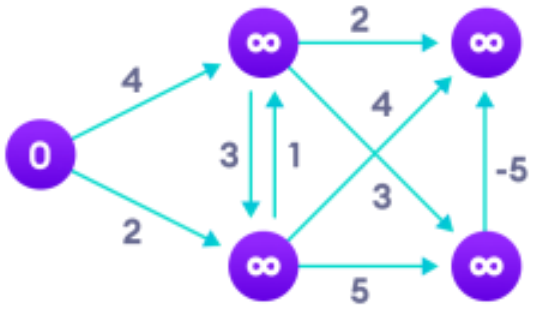
✔ Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

✔ By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

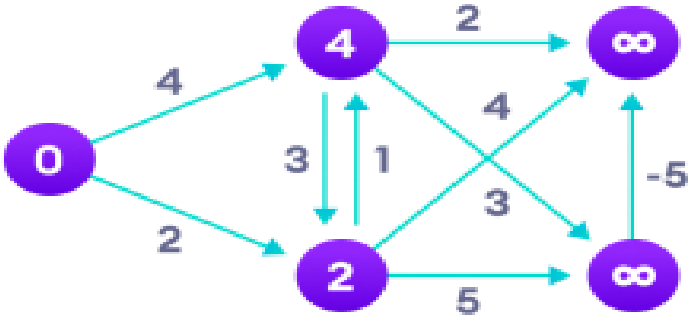
Step 1: Start with the weighted graph



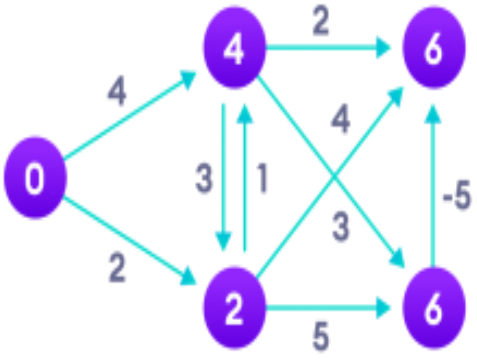
Step 2: Choose a starting vertex and assign infinity path values to all other vertices



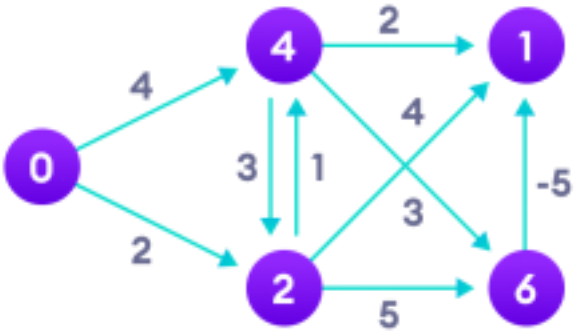
Step 3: Visit each edge and relax the path distances if they are inaccurate



Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



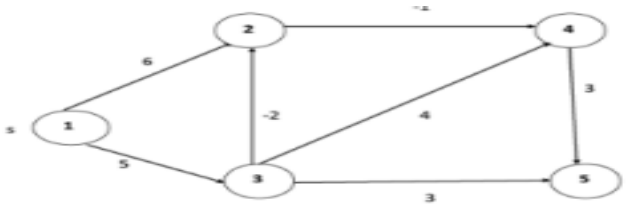
Step 5: Notice how the vertex at the top right corner had its path length adjusted



Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

EXAMPLE 2:



Solution:

$dist^k[u] = [\min[dist^{k-1}[u], \min[dist^{k-1}[i] + cost[i,u]]]$ as $i \neq u$.

Vertices

No of Edges Traversed	1	2	3	4	5
1	0	6	5	∞	∞
2	0	3	5	5	8
3	0	3	5	2	8
4	0	3	5	2	5

Bellman Ford Pseudocode

- ✔ We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.
- ✔ We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.
- ✔ Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```
function bellmanFord(G, S)
 for each vertex V in G
 distance[V] <- infinite
 previous[V] <- NULL
 distance[S] <- 0

 for each vertex V in G
 for each edge (U,V) in G
```

tempDistance <- distance[U] + edge_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

for each edge (U,V) in G

If distance[U] + edge_weight(U, V) < distance[V}

Error: Negative Cycle Exists

return distance[], previous[]

Bellman Ford vs Dijkstra

Bellman Ford's algorithm and Dijkstra's algorithm are very similar in structure. While Dijkstra looks only to the immediate neighbors of a vertex, Bellman goes through each edge in every iteration.

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL

    distance[S] <- 0

    for each vertex V in G

        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q

    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    return distance[], previous[]
```

Single Source Shortest Path in a directed Acyclic Graphs

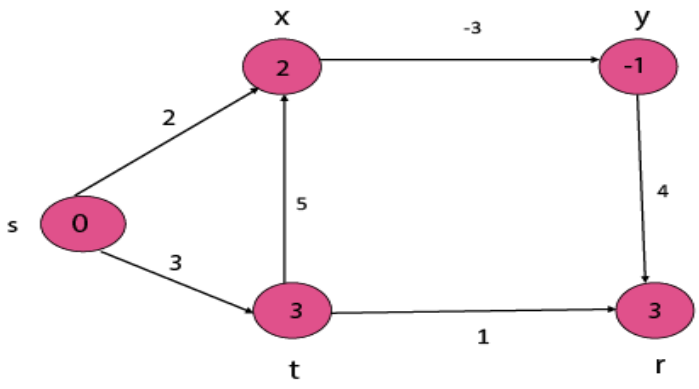
- **By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, we can figure out shortest paths from a single source in ∅(V+E) time.**
- **Shortest paths are always well described in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.**

DAG - SHORTEST - PATHS (G, w, s)

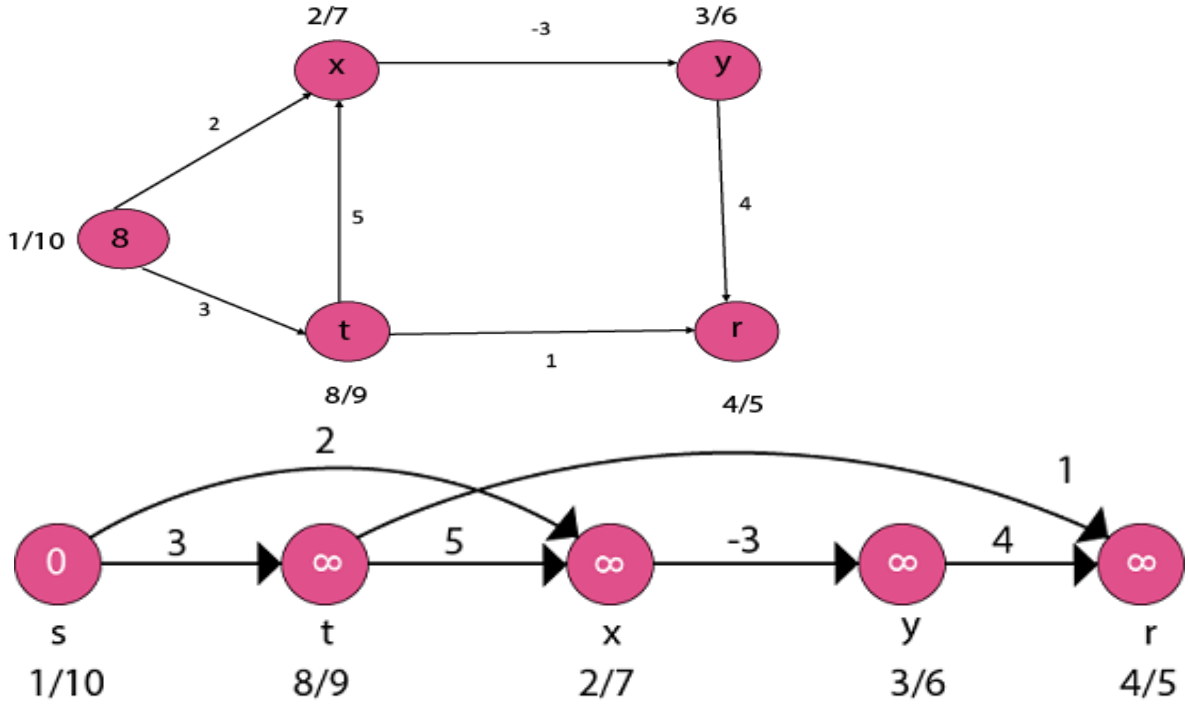
1. **Topologically sort the vertices of G.**
2. **INITIALIZE - SINGLE- SOURCE (G, s)**
3. **for each vertex u taken in topologically sorted order**
4. **do for each vertex v ∈ Adj [u]**
5. **do RELAX (u, v, w)**

The running time of this data is determined by line 1 and by the for loop of lines 3 - 5. The topological sort can be implemented in ∅ (V + E) time. In the for loop of lines 3 - 5, as in Dijkstra's algorithm, there is one repetition per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, we use only O (1) time per edge. The running time is thus ∅ (V + E), which is linear in the size of an adjacency list depiction of the graph.

Example:

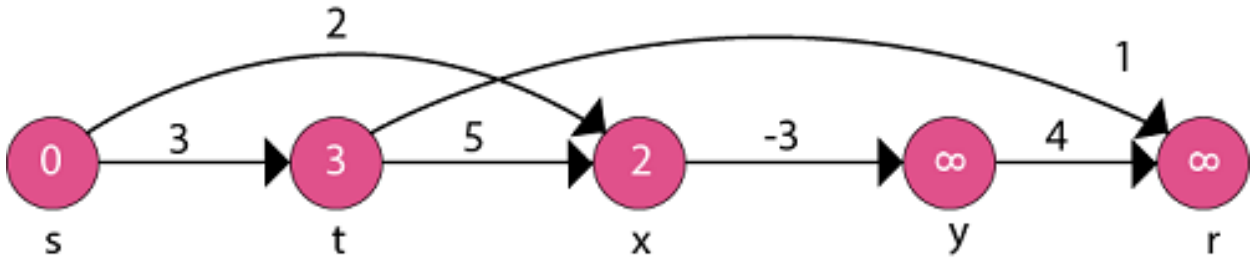


Step1: To topologically sort vertices apply DFS (Depth First Search) and then arrange vertices in linear order by decreasing order of finish time.



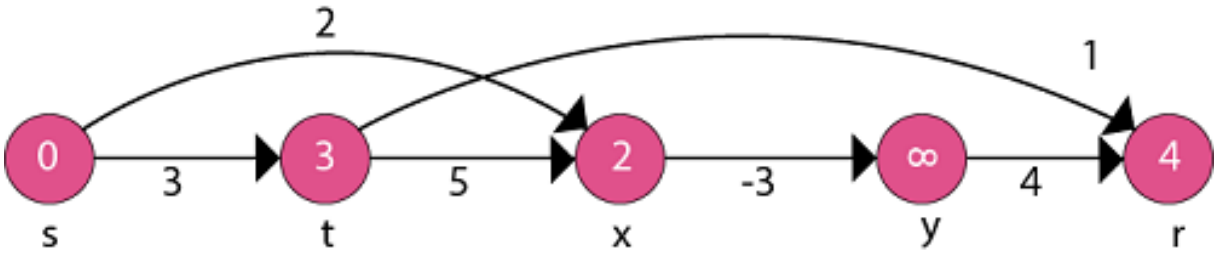
Initialize Single Source

Now, take each vertex in topologically sorted order and relax each edge.

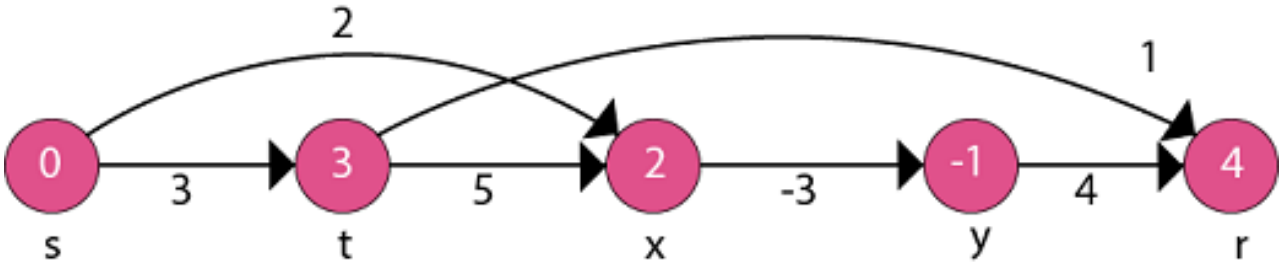


1. **adj [s] →t, x**
2. **0 + 3 < ∞**
3. **d [t] ← 3**
4. **0 + 2 < ∞**
5. **d [x] ← 2**

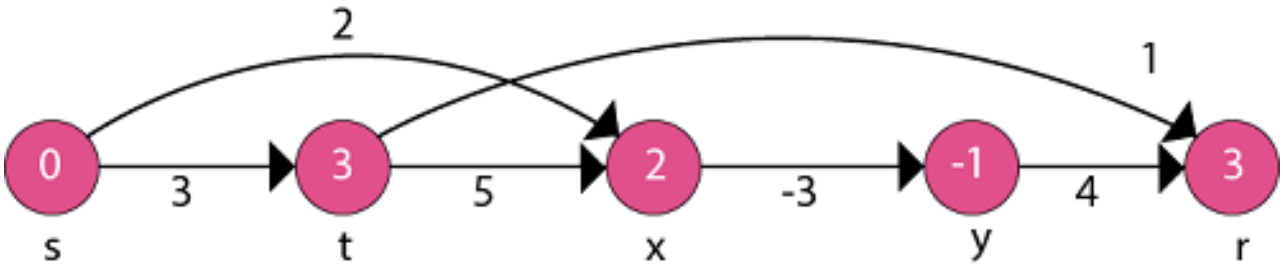
MC4101



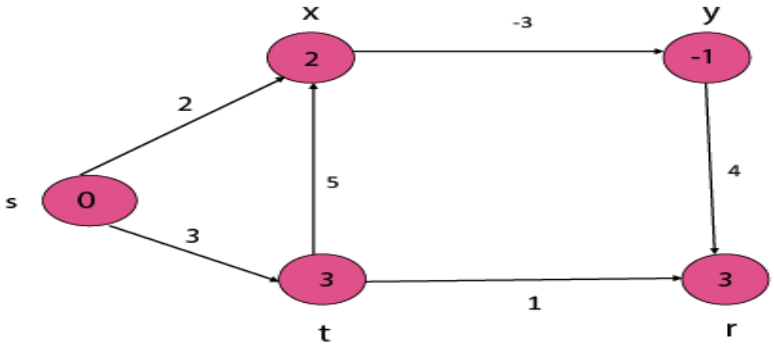
1. adj [t] → r, x
2. 3 + 1 < ∞
3. d [r] ← 4
4. 3 + 5 ≤ 2



1. adj [x] → y
2. 2 - 3 < ∞
3. d [y] ← -1



1. adj [y] → r
2. -1 + 4 < 4
3. 3 <4
4. d [r] ← 3



Thus the Shortest Path is:

1. **s to x is 2**
2. **s to y is -1**
3. **s to t is 3**
4. **s to r is 3**

Dijkstra's Algorithm;

- **Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.**

- **Dijkstra's Algorithm allows you to calculate the shortest path between one node (you pick which one) and *every other node in the graph*.**
- **Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.**
- **It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.**
- **The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.**
- **For a given source node in the graph, the algorithm finds the shortest path between that node and every other**
- **It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.**

For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

How Dijkstra's Algorithm works

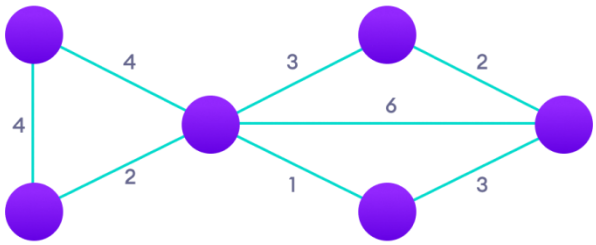
Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

Each subpath is the shortest path

- ✔ **Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex.**
- ✔ **Then we visit each node and its neighbors to find the shortest subpath to those neighbors.**
- ✔ **The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.**

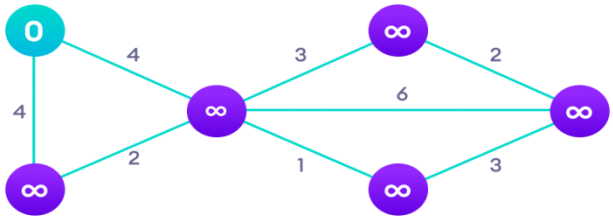
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.



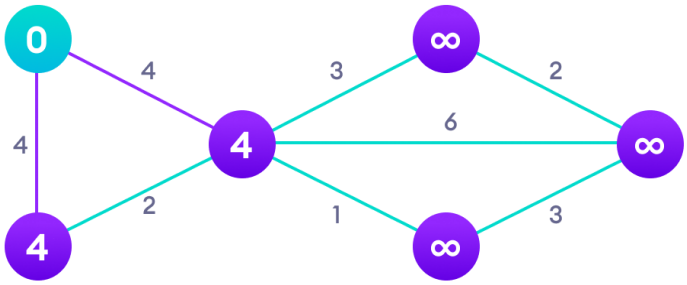
Step: 1

Start with a weighted graph



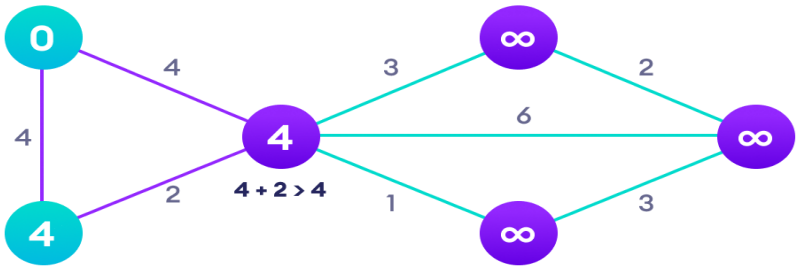
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



Step: 3

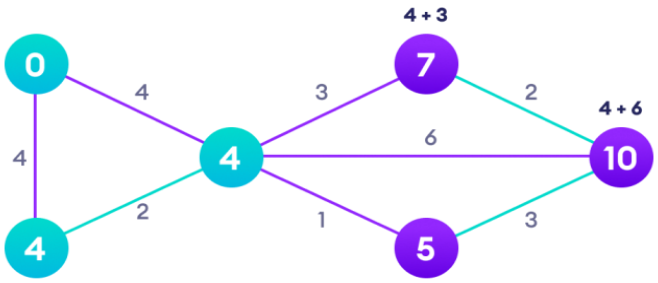
Go to each vertex and update its



Step: 4

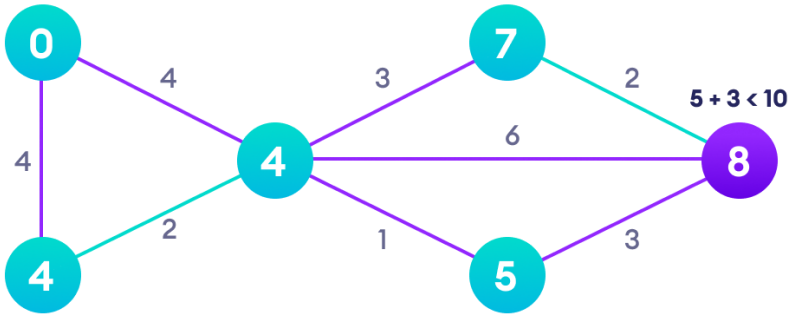
If the

path length of the adjacent vertex is lesser than new path length, don't update it



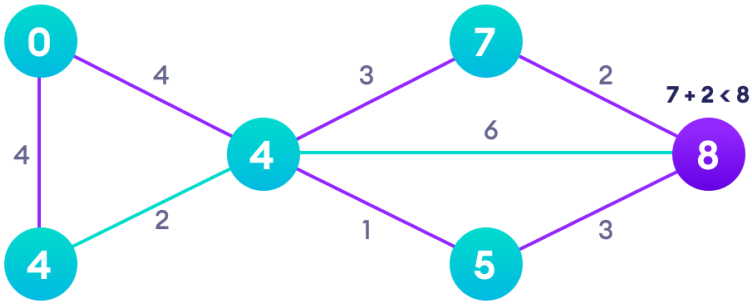
Step: 5

Avoid updating path lengths of already visited vertices



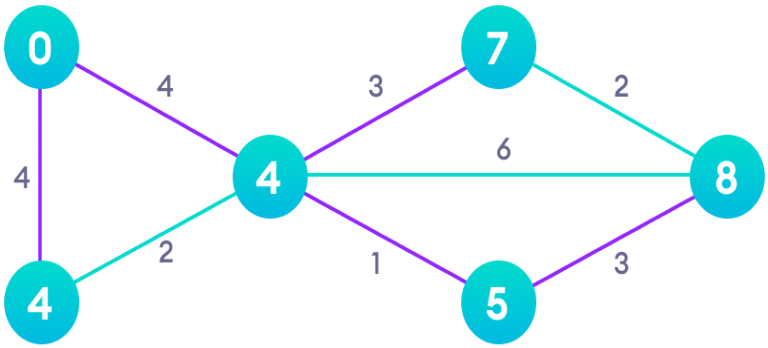
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

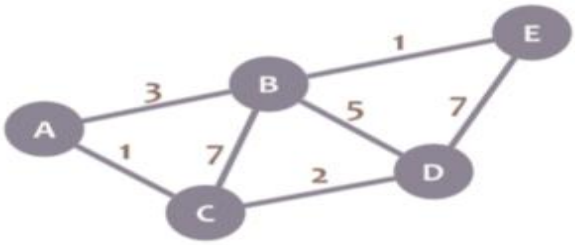
- ✔ We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.
- ✔ We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.
- ✔ Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path. A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
for each vertex V in G
distance[V] <- infinite
previous[V] <- NULL
If V != S, add V to Priority Queue Q
distance[S] <- 0
```

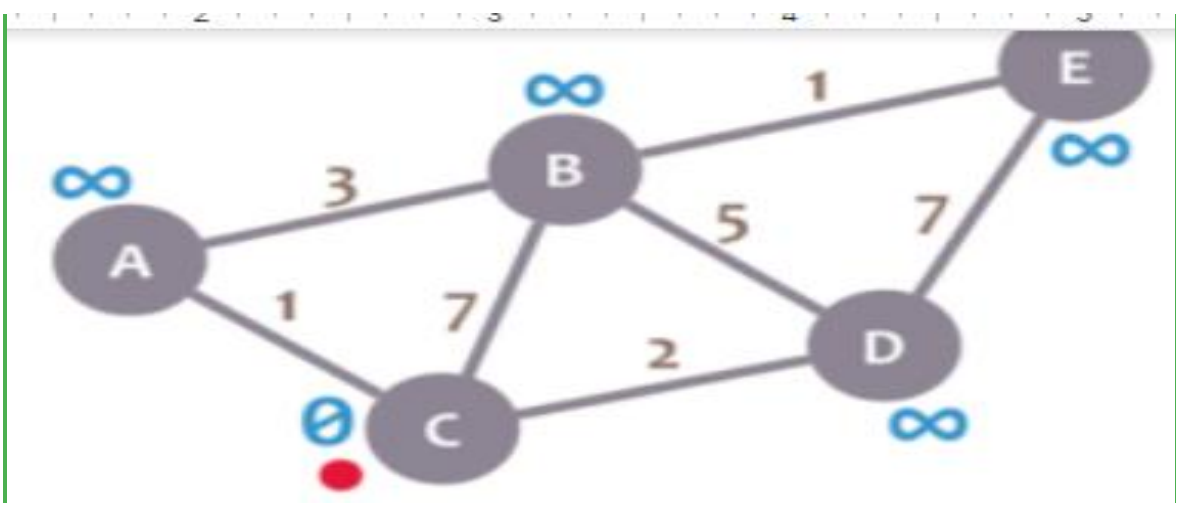
```
while Q IS NOT EMPTY
U <- Extract MIN from Q

for each unvisited neighbour V of U

tempDistance <- distance[U] + edge_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

return distance[], previous[]
```

Example 2:

Let's calculate the shortest path between node C and the other nodes in our graph:



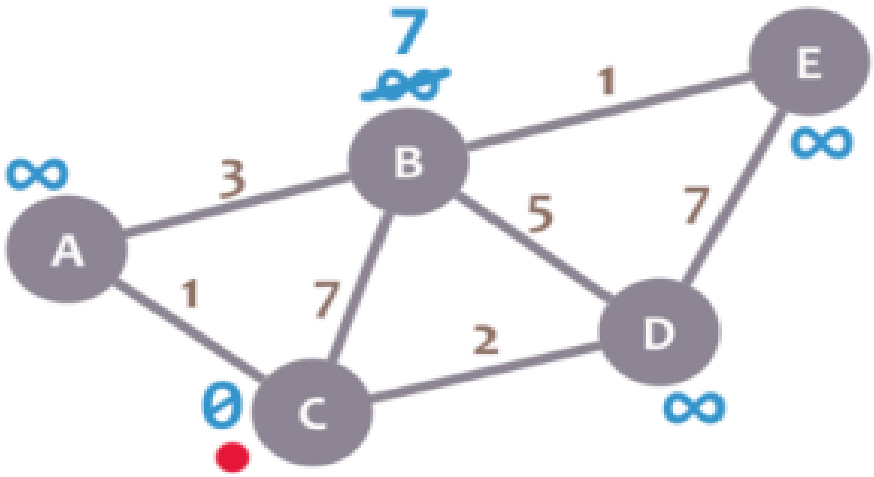
During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For **node C, this distance is 0.** For the rest of nodes, as we still don't know that minimum distance, **it starts being infinity (∞):**



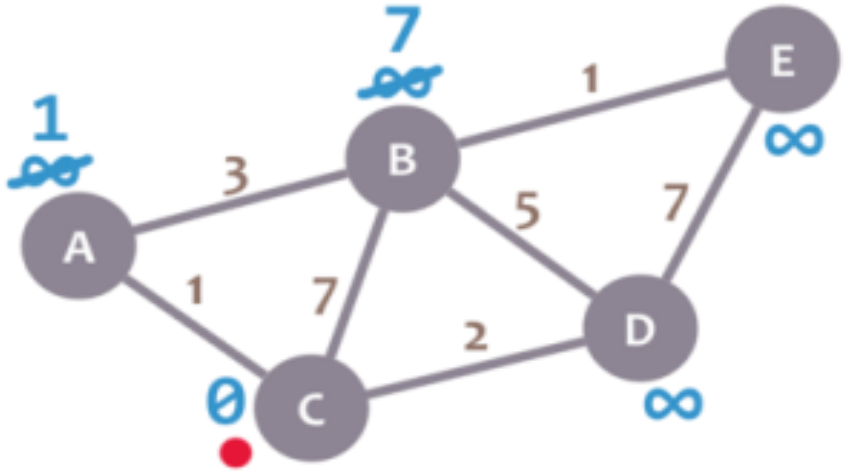
We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

✔ Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7.

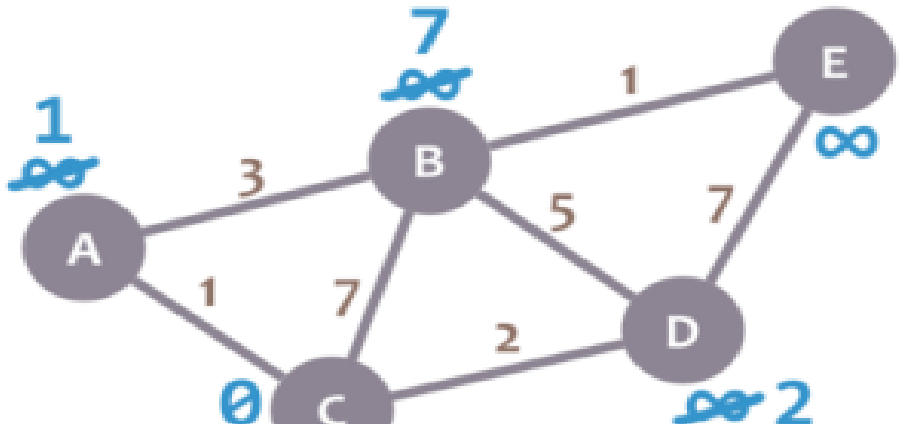
✔ We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



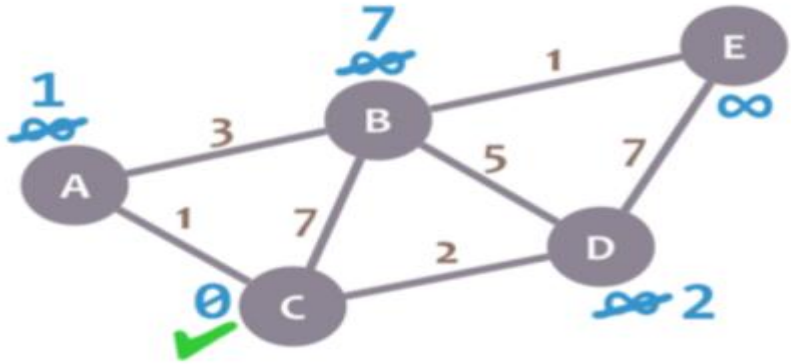
✔ So far, so good. Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



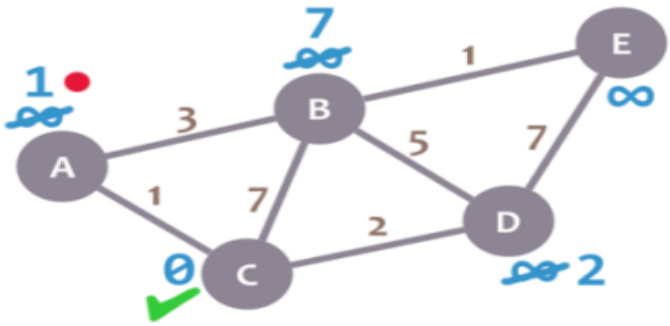
OK. Repeat the same procedure for D:



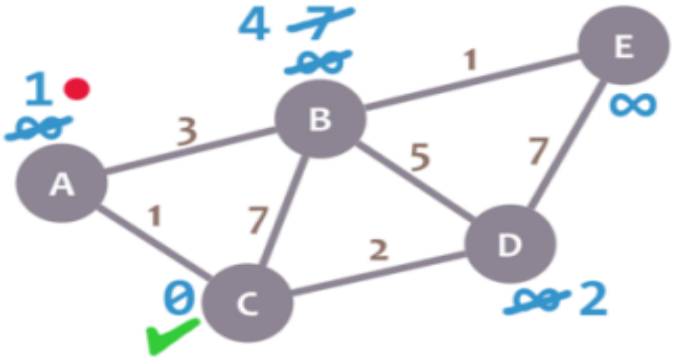
✔**We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:**



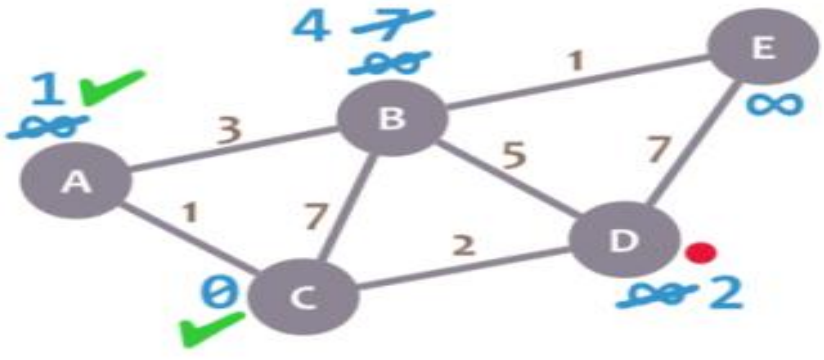
We now need to pick a new *current node*. That **node must be the unvisited** node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:



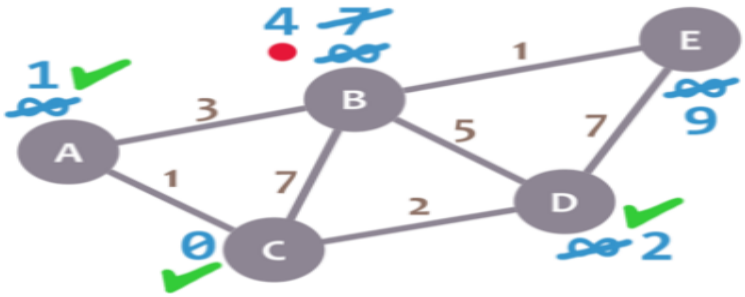
- ✓ And now we repeat the algorithm. We check the neighbours of our current node, **ignoring the visited nodes**. This means we only check B.
- ✓ For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



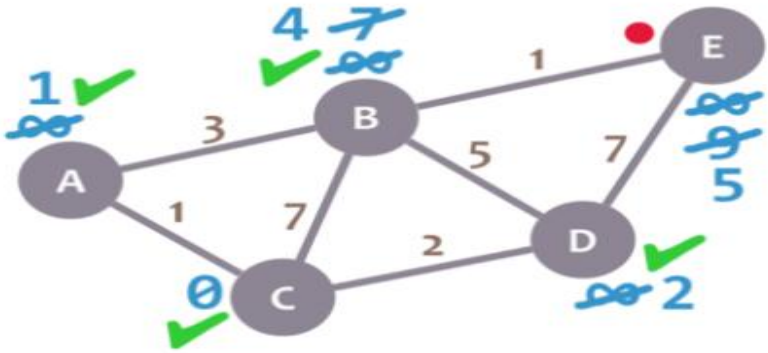
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



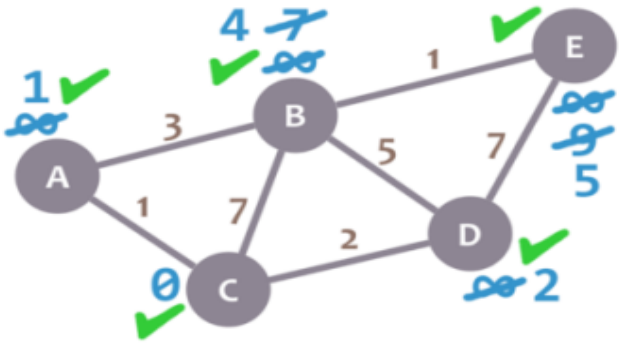
- ✓ We repeat the algorithm again. This time, we check B and E.
- ✓ For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4).
- ✓ For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).
- ✓ We mark D as visited and set our current node to B.



- ✓ Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!

Limitations of Dijkstra Algorithm

The following are some limitations of the Dijkstra Algorithm:

1. **The Dijkstra algorithm does not work when an edge has negative values.**
2. **For cyclic graphs, the algorithm does not evaluate the shortest path. Hence, for the cyclic graphs, it is not recommended to use the Dijkstra Algorithm.**

Usages of Dijkstra Algorithm

A few prominent usages of the Dijkstra algorithm are:

1. **The algorithm is used by Google maps.**
2. **The algorithm is used to find the distance between two locations.**
3. **In IP routing also, this algorithm is used to discover the shortest path.**

Dynamic Programming - All-Pairs Shortest Paths:

Introduction

The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be

solved using. applications of Dijkstra's algorithm or all at once using the Floyd-Warshall algorithm.

This is often impractical regarding memory consumption, so these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in u's row and v's column should be the weight of the shortest path from u to v.

Three approaches for improvement:

Algorithm	Cost
Matrix Multiplication	O (V^3 logV)
Floyd-Warshall	O (V^3)
Johnson O	(V^2 log?V+VE)

Unlike the single-source algorithms, which assume an adjacency list representation of the graph, most of the algorithm uses an adjacency matrix representation. (Johnson's Algorithm for sparse graphs uses adjacency lists.) The input is a n x n matrix W representing the edge weights of an n-vertex directed graph G = (V, E). That is, W = (w_{ij}), where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

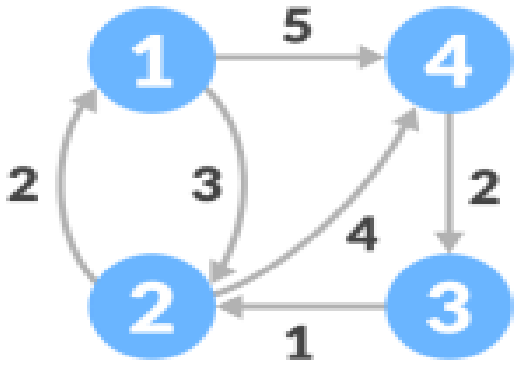
The Floyd-Warshall Algorithm

- **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.**
- **But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).**
- **A weighted graph is a graph in which each edge has a numerical value associated with it.**
- **Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.**
- **This algorithm follows the dynamic programming approach to find the shortest paths. ✔Let the vertices of G be V = {1, 2.....n} and consider a subset {1, 2.....k} of vertices for some k.**
- **For any pair of vertices i, j ∈ V, considered all paths from i to j whose intermediate vertices are all drawn from {1, 2.....k}, and let p be a minimum weight path from amongst them.**
- **The Floyd-Warshall algorithm exploits a link between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2.....k-1}.**
- **The link depends on whether or not k is an intermediate vertex of path p.**
- **If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2.....k-1}.**
- **Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k-1} is also the shortest path i to j with all intermediate vertices in the set {1, 2.....k}.**

➢ **✔If k is an intermediate vertex of path p, then we break p down into i → k → j.**

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Initial graph

- ✓ Follow the steps below to find the shortest path between all the pairs of vertices.
- ✓ Create a matrix A^0 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 i^{th} vertex to the j^{th} vertex.
- ✓ If there is no path from i^{th} vertex to j^{th} 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
- ✓ Now, create a matrix A^1 **using matrix** A^0.
- ✓ The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.
- ✓ 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]).

- ✓ That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].
- ✓ In this step, **k is vertex 1**. We calculate the distance from source vertex to destination vertex through this vertex k.

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

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

  For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

- ✓ Similarly, A^2 is created using A^3. The elements in the second column and the second row are left as they are.
- ✓ In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

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

- ✓ Calculate the distance from the source vertex to destination vertex through this vertex 2. Similarly, A^3 and A^4 is also created.

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

- ✓ Calculate the distance from the source vertex to destination vertex through this vertex 3

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

- ✓ Calculate the distance from the source vertex to destination vertex through this vertex 4.A^4 gives the shortest path between each pair of vertices.

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

         A^k[i, j] = min (A^(k-1)[i, j], A^(k-1)[i, k] + A^(k-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 $O(n^3)$.

Space Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Floyd Warshall Algorithm Applications

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

Example: Apply Floyd-Warshall algorithm for constructing the shortest path. Show that matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.

Python Example:

```
# Floyd Warshall Algorithm in python

# The number of vertices
nV = 4

INF = 999

# Algorithm implementation
def floyd_warshall(G):
  distance = list(map(lambda i: list(map(lambda j: j, i)), G))
  # Adding vertices individually
  for k in range(nV):
    for i in range(nV):
      for j in range(nV):
        distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
```

```
    print_solution(distance)
# Printing the solution
def print_solution(distance):
    for i in range(nV):
        for j in range(nV):
            if(distance[i][j] == INF):
                print("INF", end=" ")
            else:
                print(distance[i][j], end="  ")
        print(" ")
G = [[0, 3, INF, 5],
```