## UNIT II GRAPH ALGORITHMS

Graph algorithms: Representations of graphs - Graph traversal: DFS – BFS - applications - Connectivity, strong connectivity, bi-connectivity - Minimum spanning tree: Kruskal's and Prim's algorithm- Shortestpath: Bellman-Ford algorithm - Dijkstra's algorithm - Floyd-Warshall algorithm Network flow: Flownetworks - Ford-Fulkerson method – Matching: Maximum bipartite matching

### 2.1 Define Graph Data Structure

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are referred as nodes and edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( V ) and a set of edges( E ). The graph is denoted by G(E, V).

*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)}.

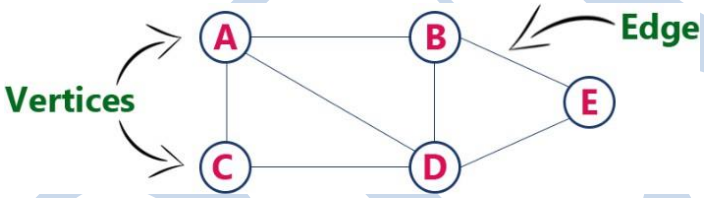



Fig 2.1 Simple Undirected Graph

### 2.1.1 Terminologies used in Graphs

*Vertex*

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.

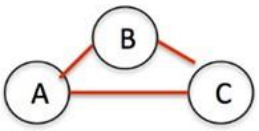
*Edge*

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). 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.*

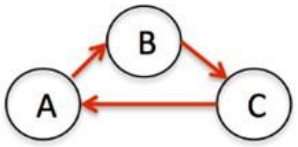
*Undirected Edge*

An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).



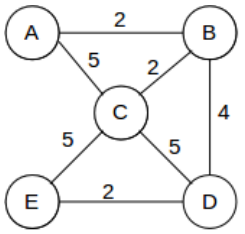

*Directed Edge*

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).

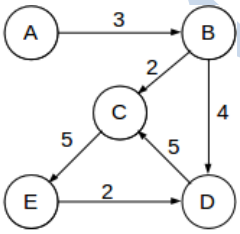



*Weighted Edge*

A weighted edge is a edge with value (cost) on it. Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.



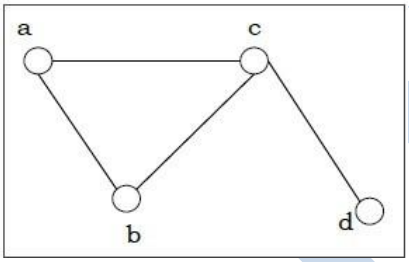

Undirected




Directed

*Connected Graph*

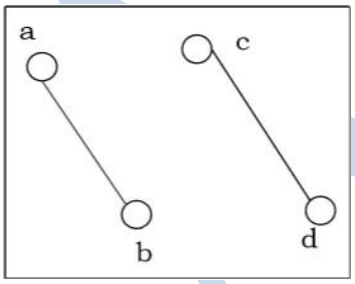
A graph is connected if any two vertices of the graph are connected by a path.




| Vertex 1 | Vertex 2 | PATH |
|---|---|---|
| a | b | a b |
| a | c | a b c, a c |
| a | d | a b c d, a c d |
| b | c | b a c, b c |
| c | d | c d |

*Disconnected Graph*

A graph is disconnected if at least two vertices of the graph are not connected by a path.




| Vertex 1 | Vertex 2 | PATH |
|---|---|---|
| a | b | a b |
| a | c | Not Available |
| a | d | Not Available |
| b | c | Not Available |
| c | d | c d |

*Degree of a vertex*

Total number of edges connected to a vertex is said to be degree of that vertex. The degree of a vertex is indicated with the help of $deg(v)$. If there is a simple graph, which contains n number of vertices, in this case, the degree of any vertex will be:

$$Deg(v) = n - 1 \; \forall \; v \in G$$

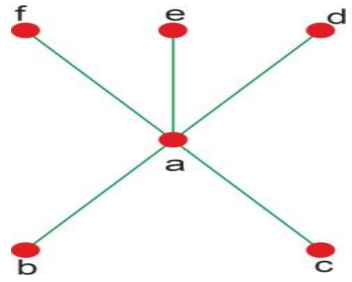
*Indegree - $deg^+(v)$*

Total number of incoming edges connected to a vertex is called as indegree of that vertex.

*Outdegree – deg⁻(v)*

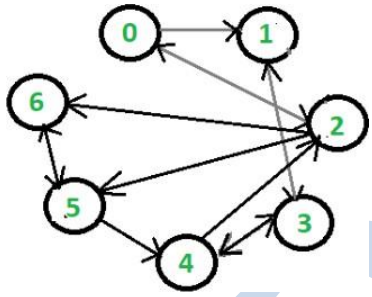
Total number of outgoing edges connected to a vertex is called as outdegree of that vertex.

*Example 1:*




In this example, we have a graph that has 6 vertices, i.e., a, b, c, d, e, and f. The vertex 'a' has degree 5, and all the other vertices have a degree 1.

*Example 2:*




| Vertex | In | Out |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 2 | 1 |
| 2 | 2 | 3 |
| 3 | 2 | 2 |
| 4 | 2 | 2 |
| 5 | 2 | 2 |
| 6 | 2 | 1 |

*Origin*

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

*Destination*

If an 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.

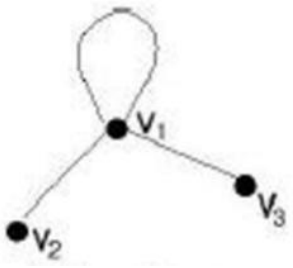
*Parallel edge*

If two vertices are connected through more than one edges then we can say there exist parallel edges in between both vertices.



*Self-loop*

If the start point and endpoint of an edge is the same vertex then there exists a self-loop.




*Simple graph*

Graph with no parallel edge and self-loop is called simple graph

*Complete graph/Clique*

A complete graph or clique is a simple graph in which every pair of vertices has an edge.n vertices complete graph is denoted by Kn.

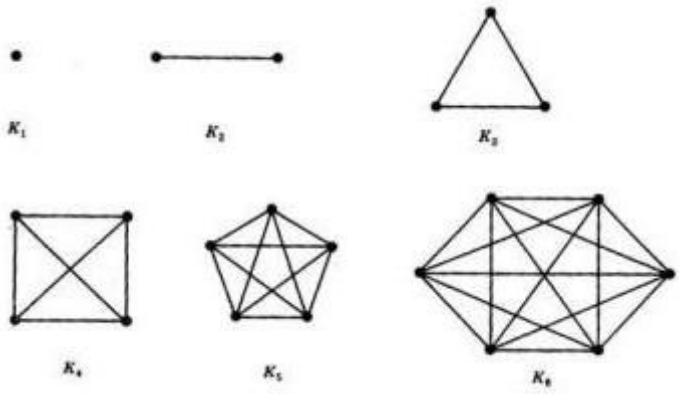
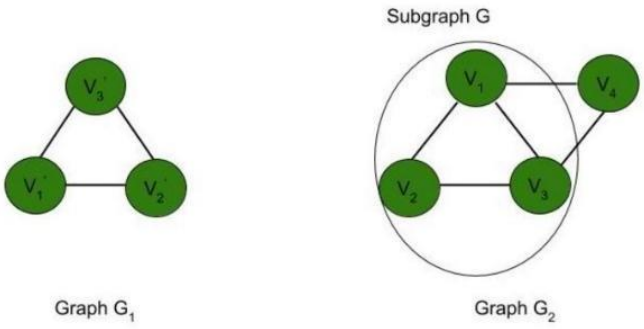


Figure2.2. Complete graph of size 1 to 6.

*Sub-graph*

A sub-graph of a graph G is a graph such that v(H) ⊆ V (G) and E(H) ⊆ E(G).



### 2.1.2 Visualize graphs generated in NetworkX using Matplotlib

NetworkX is a Python library for studying graphs and networks..

*Step 1: Import networkx and matplotlib.pyplot in the project file.*

*Step 2: Generate a graph using networkx.*

*Step 3: Now use draw() function of networkx.drawing to draw the graph.*

*Step 4: Use savefig("filename.png") function of matplotlib.pyplot to save the drawing of graph in filename.png file.*

**Simple Python program to draw a Graph**

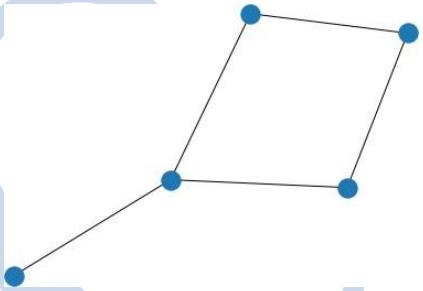
```
# importing networkx
import networkx as nx
# importing matplotlib.pyplot
import matplotlib.pyplot as plt

g = nx.Graph()

g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)
g.add_edge(1, 4)
g.add_edge(1, 5)

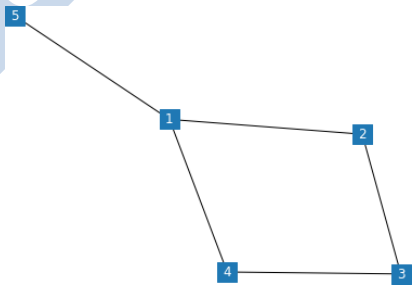
nx.draw(g)
```

Output



```
nx.draw(G, with_labels = True,
        font_color = 'white',
        node_shape='s')
```

Output



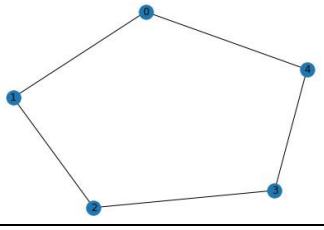

```
g = nx.cycle_graph(5)

print("Nodes in graph : ", g.nodes())
print("Edges in the graph : ", g.edges())
```

Output

Nodes in graph : [0, 1, 2, 3, 4]

Edges in the graph : [(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

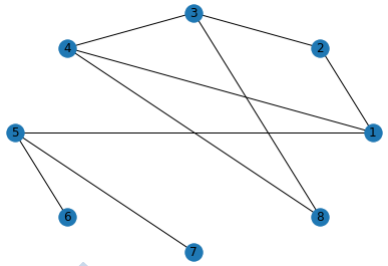


```
g = nx.Graph()
g.add_edges_from([(1,2),(2, 3),
                  (3,4),(1,4),(1,5),
                  (5,6), (5,7),
                  (4,8),(3,8)])
```

Output

```
# drawing in circular layout

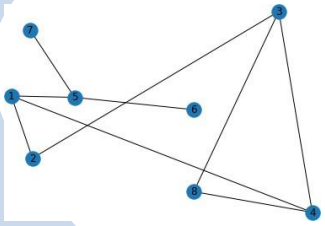
nx.draw_circular(g, with_labels = True)
```



```
# drawing in random layout

nx.draw_random(g, with_labels = True)
```

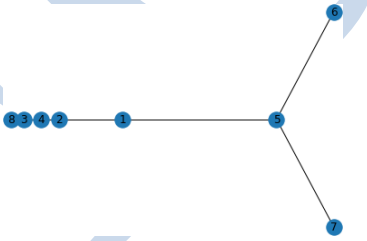
Output



```
# drawing in spectral layout

nx.draw_spectral(g, with_labels = True)
```

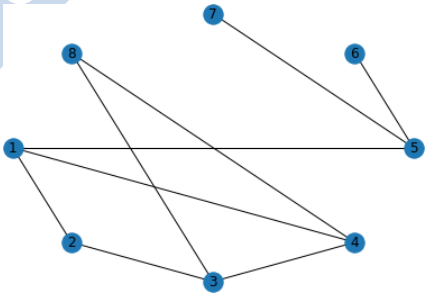
Output



```
# drawing in shell layout

nx.draw_shell(g, with_labels = True)
```

Output



**Basic Network from pandas data frame**

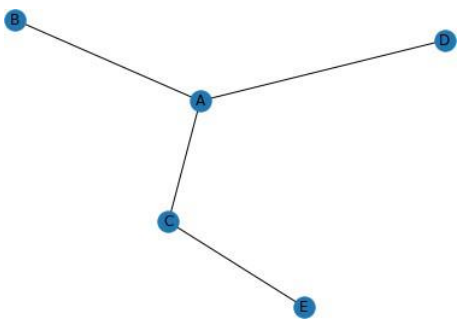
```
# Build a dataframe with 4 connections

df = pd.DataFrame({
    'from':['A', 'B', 'C','A'],
    'to':['D', 'A', 'E','C']})

# Build your graph
G = nx.from_pandas_edgelist(df,
                            'from',
                            'to')

# Plot it
nx.draw(G, with_labels=True)
plt.show()
```

Output



### 2.1.3 Real-Time Applications of Graph:

- Graphs are used to represent flow of control in computers.
- Graphs are used in social networking sites where users act as nodes and connection between them acts as edges.
- In an operating system, graphs are used as resource allocation graphs.
- Graphs are used in Google maps to find the shortest route.
- Graphs are also used in airlines system for effective route optimization.
- In-state transition diagrams, the graph are used to represent their states and their transition.
- In transportation, graphs are used to find the shortest path.
- In circuits, graphs can be used to represent circuit points as nodes and wires as edges.
- Graphs are used in solving puzzles with only one solution, such as mazes.
- Graphs are used in computer networks for Peer to peer (P2P) applications.
- Graphs basically in the form of DAG(Directed acyclic graph) are used as alternative to block chain for crypto-currency.

### 2.1.4 Merits and Demerits of Graph

*Merits*

- By using graphs we can easily find the shortest path, neighbors of the nodes, and many more.
- Graphs are used to implement algorithms like DFS and BFS.
- It is used to find minimum spanning tree which has many practical applications.
- It helps in organizing data.
- Because of its non-linear structure, helps in understanding complex problems and their visualization.

*Demerits*

- Graphs use lots of pointers which can be complex to handle.
- It can have large memory complexity.
- If the graph is represented with an adjacency matrix then it does not allow parallel edges and multiplication of the graph is also difficult.

## 2.2 Representations of graphs

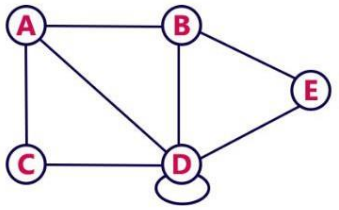
A graph can be represented using 3 data structures- **adjacency matrix, adjacency list and adjacency set.**

### 2.2.1. Adjacency Matrix

An adjacency matrix is a table with rows and columns. The row labels and column labels represent the nodes of a graph. An adjacency matrix is a square matrix where the number of rows, columns and nodes are the same. Each cell of the matrix represents an edge or the relationship between two given nodes. For example, adjacency matrix $a_{ij}$ represents the number of links from i to j, given two nodes i and j.

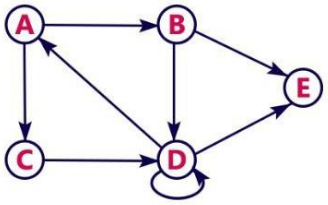
$$a_{ij} = \begin{bmatrix} 1, \text{if } \{V_i, V_j\} \text{is an edge i.e., } v_i \text{is adjacent to } v_j \\ 0, \text{if there is no edge between } v_i \text{ and } v_j \end{bmatrix}$$

*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 |

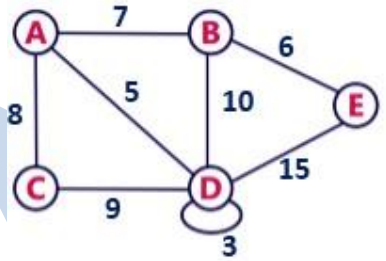
*Directed graph representation*




| | 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 representation*




| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 7 | 8 | 5 | 0 |
| B | 7 | 0 | 0 | 10 | 6 |
| C | 8 | 0 | 0 | 9 | 0 |
| D | 5 | 10 | 9 | 3 | 15 |
| E | 0 | 6 | 0 | 15 | 0 |

*Merits*

- Representation is easier to implement.
- Adding or deleting an edge takes constant time.

*Demerits*

- The size of adjacency matrix is $V^2$. Suppose there is a graph with 1000 vertices and 1 edge. The array of size $1000^2$is used for storing one edge which is a waste of memory.
- It takes a lot of space and time to visit all the neighbours of a vertex, we have to traverse all the vertices in the graph, which takes much time.

### 2.2.2. Incidence Matrix

In this representation, the graph is represented using a matrix of total number of vertices by total number of edges. A graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represent edges. The matrix is filled with 0, 1 or -1.

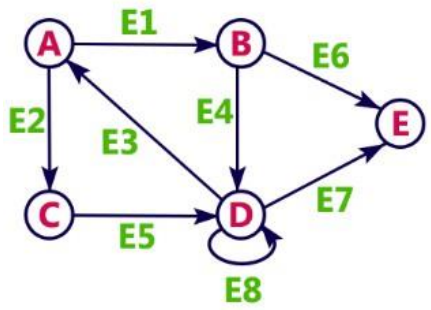
- 0→ represents that the row edge is not connected to column vertex
- 1 →represents that the row edge is connected as the outgoing edge to column vertex
- -1→ represents that the row edge is connected as the incoming edge to column vertex.

| Types of Branch | Value |
|---|---|
| Incoming branch to $k^{th}$ node | -1 |
| Outcome branch to $k^{th}$ node | +1 |
| Row edge is not connected to column vertex | 0 |

The incidence matrix is an n x m matrix C = [$c_{ij}$] and defined by

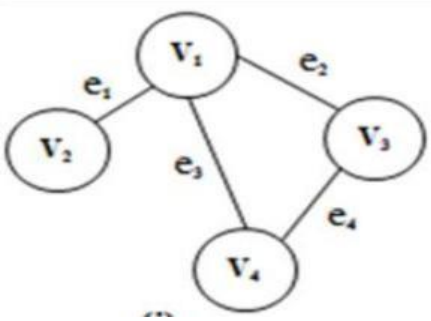
$$c_{ij} = \begin{bmatrix} 1, \text{if } V_i \text{is the initial vertex of edge } e_j \\ -1, \text{if } V_i \text{is the final vertex of edge } e_j \\ 0, V_i \text{is not incident on edge } e_j \end{bmatrix}$$

*Example 1:*



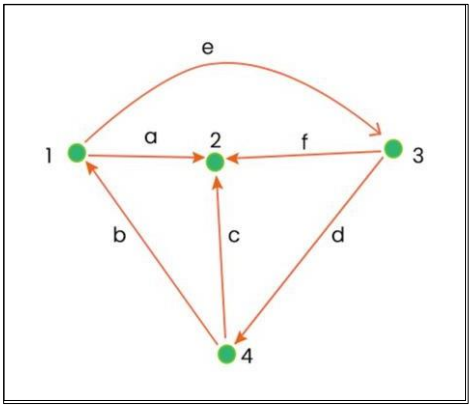

| | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | -1 | 0 | 0 | 0 | 0 | 0 |
| B | -1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| C | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | -1 | -1 | 0 | 1 | 1 |
| E | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 0 |

*Example 2:*



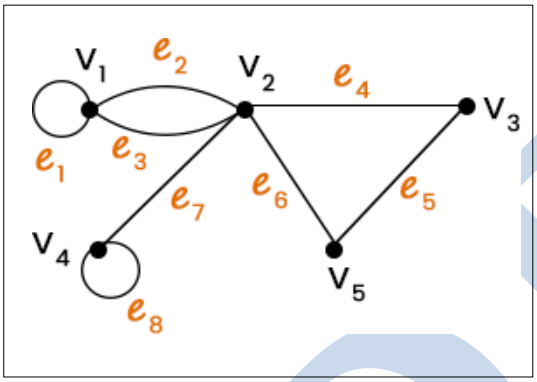

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ |
|---|---|---|---|---|
| $V_1$ | 1 | 1 | 1 | 0 |
| $V_2$ | 1 | 0 | 0 | 0 |
| $V_3$ | 0 | 1 | 0 | 1 |
| $V_4$ | 0 | 0 | 1 | 1 |

*Example 3:*




| Nodes | Branches | | | | | |
|---|---|---|---|---|---|---|
| | a | b | c | d | e | f |
| 1 | 1 | -1 | 0 | 0 | 1 | 0 |
| 2 | -1 | 0 | -1 | 0 | 0 | -1 |
| 3 | 0 | 0 | 0 | 1 | -1 | 1 |
| 4 | 0 | 1 | 1 | -1 | 0 | 0 |

*Example 4:*

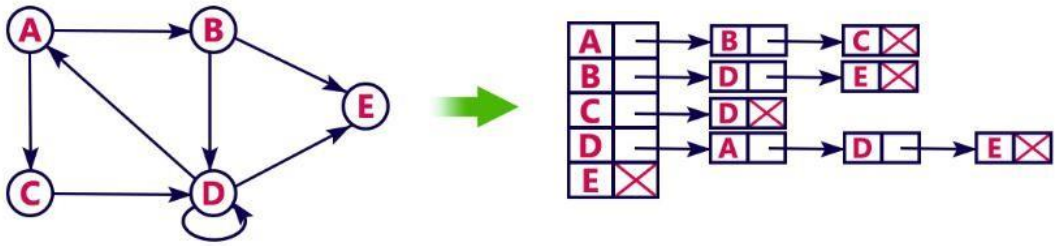



| Nodes | Branches | | | | | |
|---|---|---|---|---|---|---|
| | e1 | e2 | e3 | e4 | e5 | e6 |
| v1 | 1 | 1 | 0 | 0 | 0 | 0 |
| v2 | 0 | 0 | 1 | 1 | 0 | 1 |
| v3 | 0 | 0 | 0 | 0 | 1 | 1 |
| v4 | 1 | 0 | 1 | 0 | 0 | 0 |
| v5 | 0 | 1 | 0 | 1 | 1 | 0 |

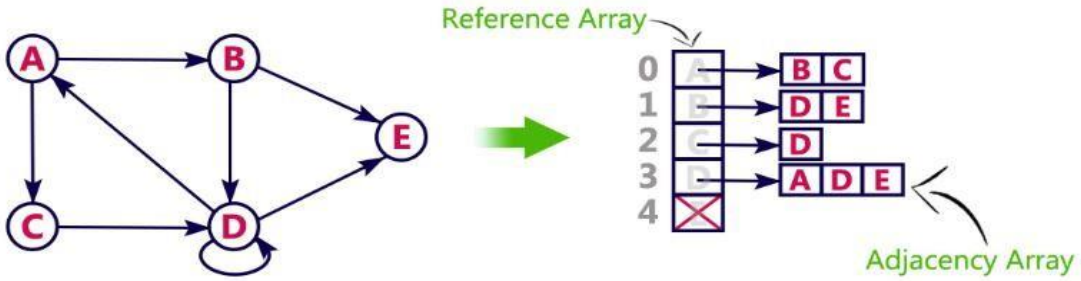
### 2.2.3 Adjacency List

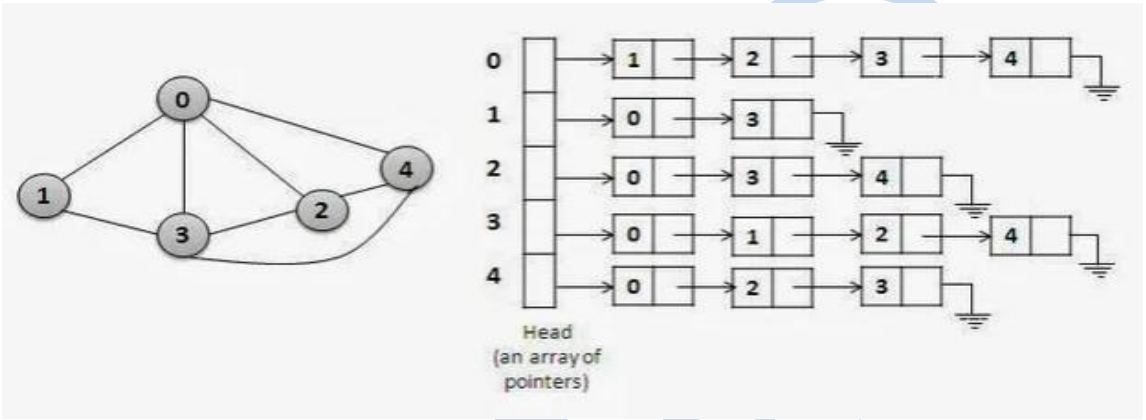
A graph can also be represented using alinked list. For each vertex, a list of adjacent vertices is maintained using alinked list. It creates a separate linked list for each vertex $Vi$ for the graph $G = (V, E)$.

Adjacency list of a graph with n nodes canbe represented by an array of pointers. Each pointer points to a linked list ofthe corresponding vertex. Below figure shows the adjacency list representationof a graph.




This representation can also be implemented using an array as follows..




*Merits*

- Easy to find successors of a node
- Easy to find all neighbouring nodes
- Space efficient as it only stores connected nodes

*Demerits*

- It is time consuming to check if an edge is part of a graph

### 2.2.4 Comparative analysis of Adjacency Matrix and Adjacency List

| OPERATIONS | ADJACENCY MATRIX | ADJACENCY LIST |
|---|---|---|
| **Storage Space** | Here, we use $VxV$ matrix, so space required in worst case is $O(\|V\|^2)$ | Here, every vertex stores its neighbours. In the worst case, if a graph is connected $O(V)$ is required for a vertex and $O(E)$ is required for storing neighbours corresponding to every vertex. Thus, overall space complexity is $O(\|V\| + \|E\|)$. |
| **Adding a vertex** | To add vertex to $VxV$matrix , the storage must be increases to $(\|V\| + 1)^2$. To achieve this we need to copy the whole matrix. Therefore the complexity is $O(\|V\|^2)$ | There are two pointers in adjacency list. 1st pointer points to the front node and 2nd pointer points to the rear node. Thus insertion of a vertex can be done directly in $O(1)$time. |

| OPERATIONS | ADJACENCY MATRIX | ADJACENCY LIST |
|---|---|---|
| Adding an edge | To add an edge say from i to j, $matrix[i][j] = 1$ requires $O(1)$ time. | Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in $O(1)$time. |
| Removing a vertex | In order to remove a vertex from V*V matrix the storage must be decreased to $\|V\|^2$ from$(\|V\|+1)^2$. To achieve this we need to copy the whole matrix. Therefore the complexity is $O(\|V\|^2)$ | In order to remove a vertex, we need to search for the vertex which will require **O(\|V\|)** time, after this we need to traverse the edges which require O(\|E\|) time. Hence, total time complexity is **O(\|V\|+\|E\|).** |
| Removing an edge | To remove an edge from i to j, matrix[i][j] = 0 requires **O(1)** time. | To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges. Thus, the time complexity is **O(\|E\|)**. |
| Querying | In order to find for an existing edge the content of matrix needs to be checked. Given two vertices say i and j matrix[i][j] can be checked in **O(1)** time. | In an adjacency list every vertex is associated with a list of adjacent vertices. For a given graph, in order to check for an edge we need to check for vertices adjacent to given vertex. A vertex can have at most O(\|V\|) neighbours and in worst can we would have to check for every adjacent vertex. Therefore, time complexity is **O(\|V\|)** . |

### 2.2.5 Time and space complexity of Graph representations

| Representation of Graph | Space Complexity | Time Complexity | | | | |
|---|---|---|---|---|---|---|
| | | Add Vertex | Add Edge | Remove Vertex | Remove Edge | Query |
| Adjacency Matrix | O(\|V\|²) | O(\|V\|²) | O(\|1) | O(\|V\|²) | O(\|1) | O(\|1) |
| Incidence Matrix | O(\|V\| + \|E\|) | O(\|V\| + \|E\|) | O(\|V\| + \|E\|) | O(\|V\| + \|E\|) | O(\|V\| + \|E\|) | O(\|E\|) |
| Adjacency List | O(\|V\| + \|E\|) | O(\|1) | O(\|1) | O(\|V\| + \|E\|) | O(\|E\|) | O(\|V\|) |

### 2.2.6 Visualize representations of Graphs generated in NetworkX using Matplotlib

```
import numpy as np
adjacency_matrix = np.array([[0,  1,  1,  0],
                             [1,  0,  0,  1],
                             [1,  0,  0,  0],
                             [1,  1,  0,  0]])
print(adjacency_matrix)
```

Output

```
[[0 1 1 0]
 [1 0 0 1]
 [1 0 0 0]
 [1 1 0 0]]
```

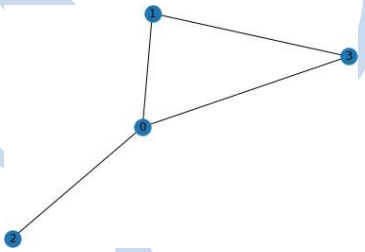
```
import numpy as np
adjacency_matrix = np.array([[0,  1,  1,  0],
                             [1,  0,  0,  1],
                             [1,  0,  0,  0],
                             [1,  1,  0,  0]])
print(adjacency_matrix)

G = nx.Graph(adjacency_matrix,nodetype=int)

nx.draw(G, with_labels = 'true')
```

Output

```
[[0 1 1 0]
 [1 0 0 1]
 [1 0 0 0]
 [1 1 0 0]]
```



**Undirected Graph**

First we create an undirected graph. When displaying the edges, we see that all edges are present independent of if they were defined in both rows or just in one.

```
G = nx.Graph(adjacency_matrix,
             nodetype=int)
G.edges()
```

Output

```
EdgeView([(0, 1), (0, 2), (0, 3), (1, 3)])
```

**Directed Graph**

In contrast when we use the same adjacency matrix to create a directed graph, only the edge from node 3 to node 0 is added, no edge is added in the opposite direction from node 0 to 3

```
G = nx.DiGraph(adjacency_matrix,
               nodetype=int)
G.edges()
```

```
OutEdgeView([(0, 1), (0, 2), (1, 0), (1, 3), (2, 0), (3, 0), (3, 1)])
```

```
# Printing Adjacency List

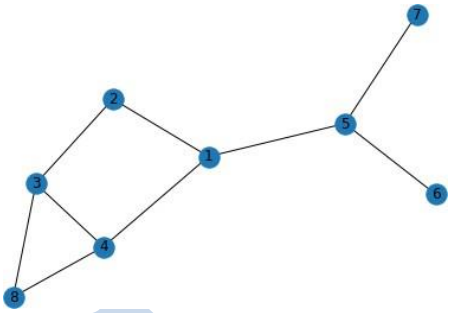
g = nx.Graph()
g.add_edges_from([(1,2),
                  (2, 3),
                  (3,4),
                  (1,4),
                  (1,5),
                  (5,6),
                  (5,7),
                  (4,8),
                  (3,8)])
```

Output

```
1 245
2 3
3 48
4 8
5 67
6
7
8
```

```
# drawing a graph
nx.draw(g, with_labels = True)
```

```
for line in nx.generate_adjlist(g):
  print(line)
```




```
#Printing weighted graph
import networkx as nx
import pylab

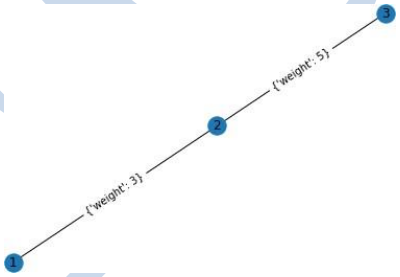

G = nx.Graph()
G.add_edge(1, 2, weight=3)
G.add_edge(2, 3, weight=5)

pos=nx.spring_layout(G)

nx.draw(G,pos, with_labels= 'true')
# use default edge labels
nx.draw_networkx_edge_labels(G,pos)
```

Output

```
{(1, 2): Text(-0.4691328677128329, -0.4157679345378789, "{'weight': 3}"), (2, 3): Text(0.49999999999999994, 0.443123860338469, "{'weight': 5}")}
```




```
#Printing adjacency list and calculating degree

import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()

G.add_edges_from([(1,2), (1,3),
                  (3,4), (3,5)])

print("Nodes")
print(G.nodes)
print("Edges")
print(G.edges)
print("Adjacency List")
print(G.adj)
print("Degree")
print(G.degree)

print()
nx.draw(G, with_labels = True)
print("Adjacency List for node 3")
```

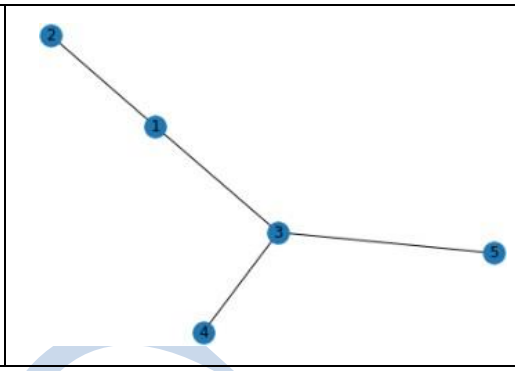
Output:

```
Nodes
[1, 2, 3, 4, 5]
Edges
[(1, 2), (1, 3), (3, 4), (3, 5)]
Adjacency List
{1: {2: {}, 3: {}},
 2: {1: {}},
 3: {1: {},  4: {},  5: {}},
 4: {3: {}},
 5: {3: {}}}
Degree
[(1, 2), (2, 1), (3, 3), (4, 1), (5, 1)]

Adjacency List for node 3
{1: {}, 4: {}, 5: {}}
Degree for node 3
3
```

APEC

```
print(G.adj[3])
print("Degree for node 3")
print(G.degree[3])
```




**#Printing successors, predecessors and calculating degree**

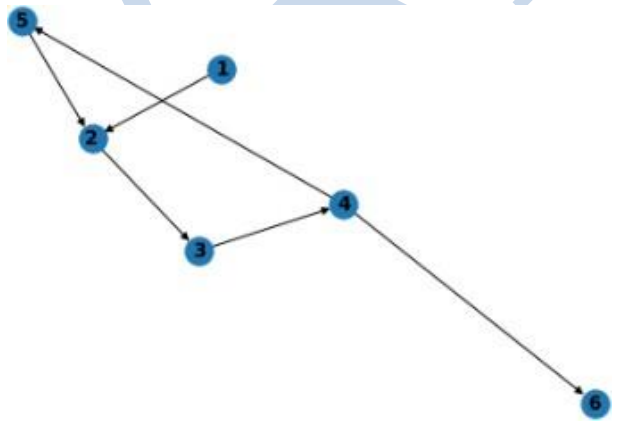
```
import networkx as nx
DG = nx.DiGraph()
DG.add_edges_from([(1,2), (2,3), (3,4), (4,5), (5,2), (4, 6)])

# Print edges going out from node 4
print("Out edges of node 4 are:",DG.out_edges(4))
# Print in degree of node 2
print("In Degree of node 2 is:",DG.in_degree(2))
# Print successors of node 4
print("Successors of node 4 are:",list(DG.successors(4)))
# Print predecessors of node 2
print("Predecessors of node 2 are:",list(DG.predecessors(2)))

nx.draw(DG, with_labels= True, font_weight='bold')
```

**Output:**

```
Out edges of node 4 are: [(4, 5), (4, 6)]
In Degree of node 2 is: 2
Successors of node 4 are: [5, 6]
Predecessors of node 2 are: [1, 5]
```



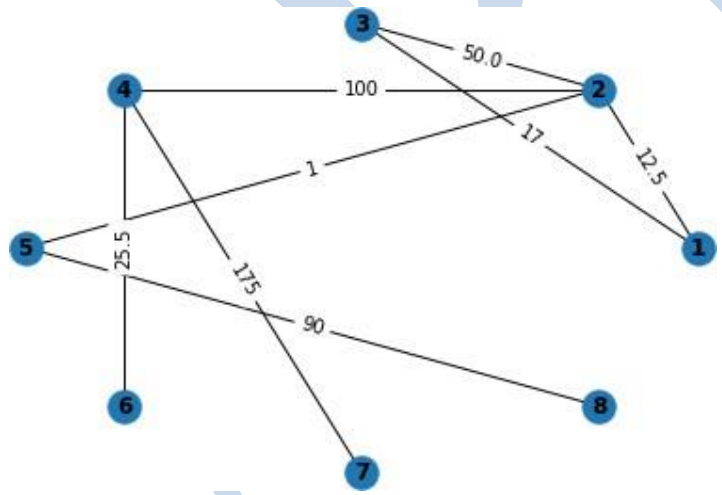
```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()

# Using add_edge
G.add_edge(1, 2, weight = 12.5)
G.add_edge(3, 2, weight = 50.0)
G.add_edge(1, 3, weight = 17)
G.add_edge(4, 2, weight = 100)
G.add_edge(2, 5, weight = 1)
G.add_edge(4, 6, weight = 25.5)
G.add_edge(7, 4, weight = 175)
G.add_edge(5, 8, weight = 90)


Pos = nx.circular_layout(G)
nx.draw(G, pos, with_labels = True, font_weight = 'bold')
edge_weight = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_weight)
plt.show()
```

**Output:**



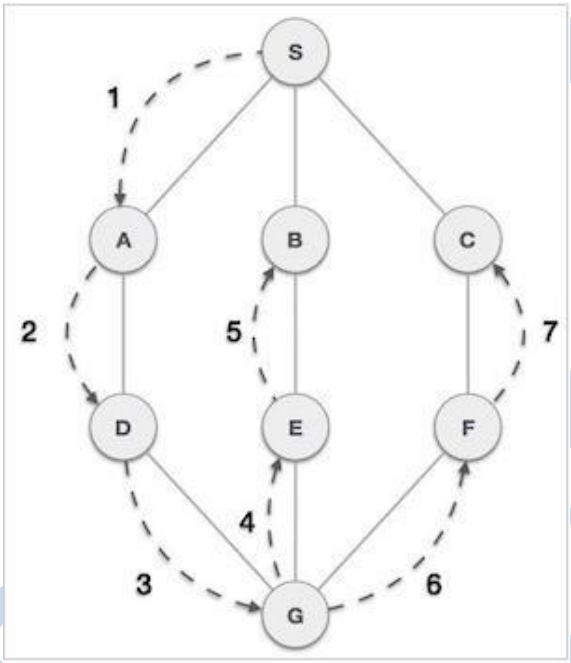

### 2.3 Graph Traversals

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

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

### 2.3.1 Depth-first search

DFS is an algorithm for traversing a finite graph. Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only difference is, graphs may contain cycles (a node may be visited twice). DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth. A stack (often the program's call stack via recursion) is generally used when implementing the algorithm.




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.A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

#### *2.3.1.1 Working of DFS algorithm*

*Step 1:* Start by putting any one of the graph's vertices on top of a stack.

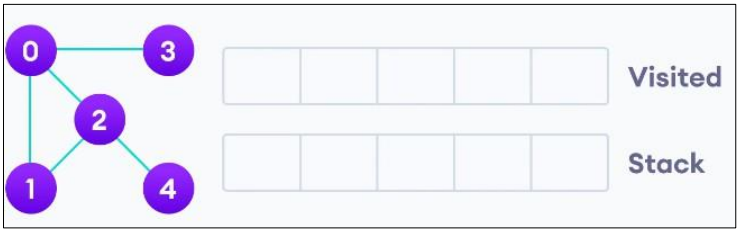
*Step 2:*Take the top item of the stack and add it to the visited list.

*Step 3:*Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

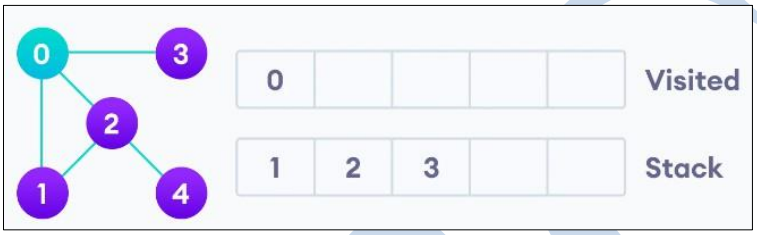
*Step 4:*Keep repeating steps 2 and 3 until the stack is empty.

#### *2.3.1.2 Example 1 – Undirected Graph:*

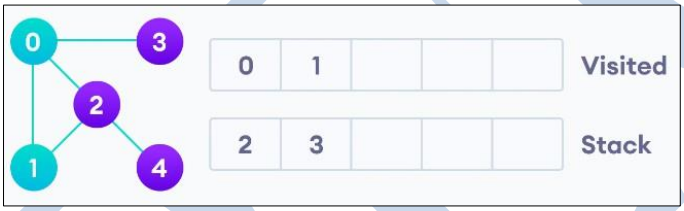
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



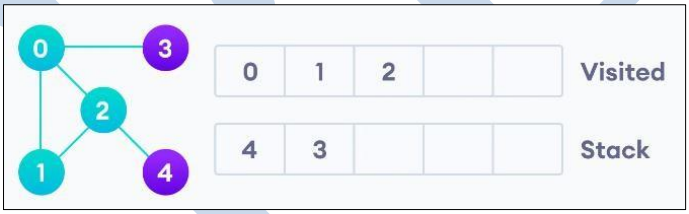

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



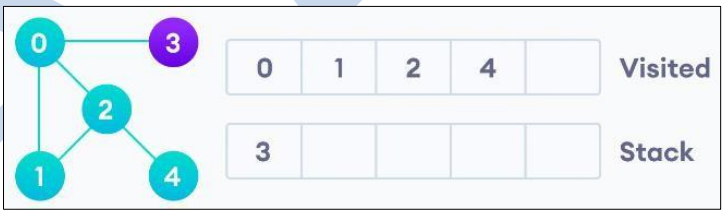

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

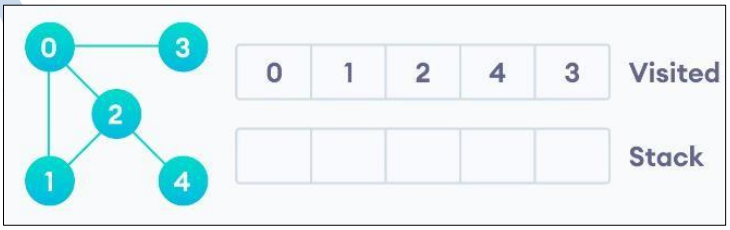



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

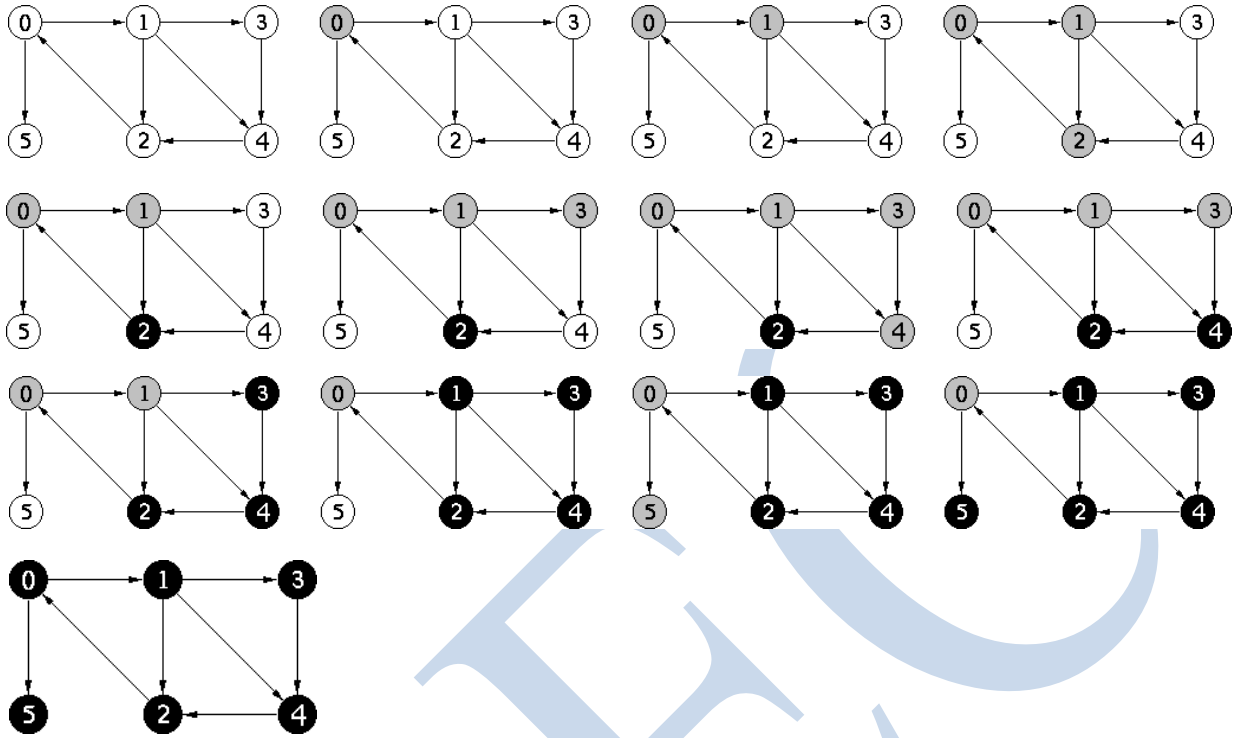



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



*2.3.1.2 Example 2 - Directed Graph:*



*2.3.1.3 Python Program for Depth-first search*

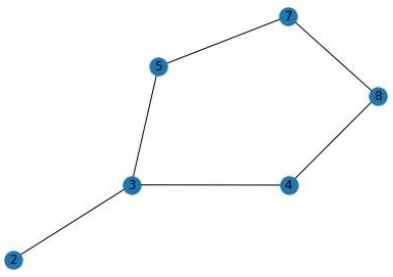
```
# Using adjacency list
g = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
G = nx.Graph(g)
nx.draw(G, with_labels = True)
visited = set()
# Set to keep track of visited nodes of graph
.

def dfs(visited, g, node):  dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in g[node]:
            dfs(visited, g, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, g, '5')
```

**Output:**

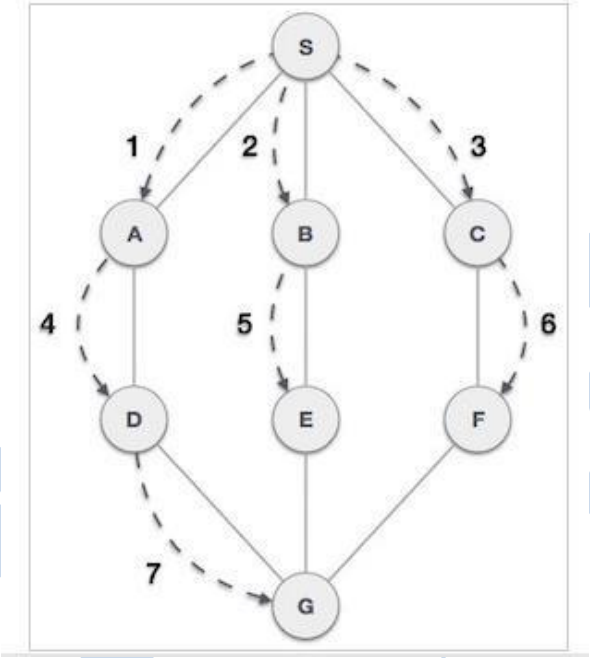
```
Following is the Depth-
First Search
5
3
2
4
8
7
```



Explanation for DFS algorithm:
It first checks if the current node is unvisited - if yes, it is appended in the visited set. Then for each neighbour of the current node, the DFS function is invoked again.
The base case is invoked when all the nodes are visited. The function then returns.

### 2.3.2 BFS algorithm

BFS - A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the neighbour vertices before visiting the child vertices. BFS uses a queue data structure for traversal.This algorithm is often used to find the shortest path from one vertex to another.




BFS traversal of a graph produces a spanning tree as final result. For simplicity, it is assumed that all vertices are reachable from the starting vertex. A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

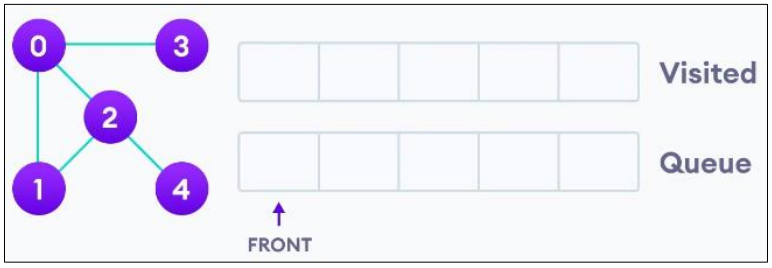
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

#### *2.3.2.1 Working of BFS algorithm*

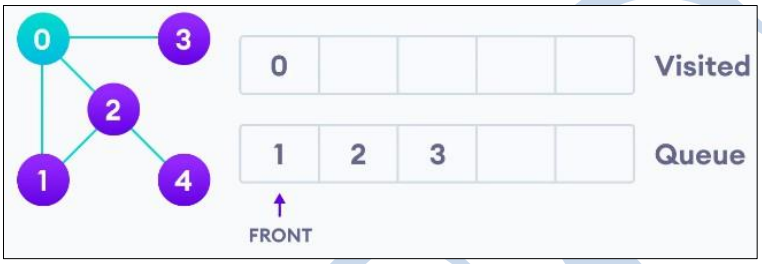
*Step 1:*Start by putting any one of the graph's vertices at the back of a queue.
*Step 2:*Take the front item of the queue and add it to the visited list.
*Step 3:*Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
*Step 4:*Keep repeating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

#### *2.3.2.2 Example 1 – Undirected Graph:*

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



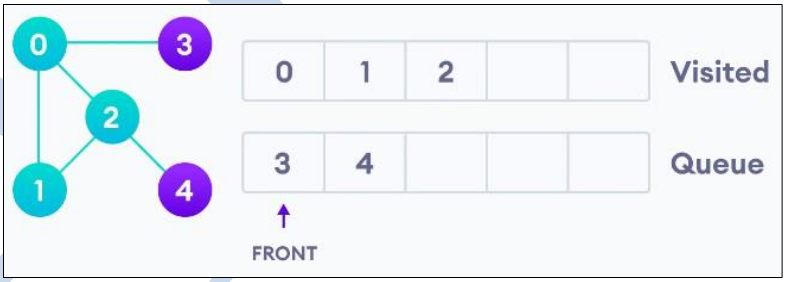

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

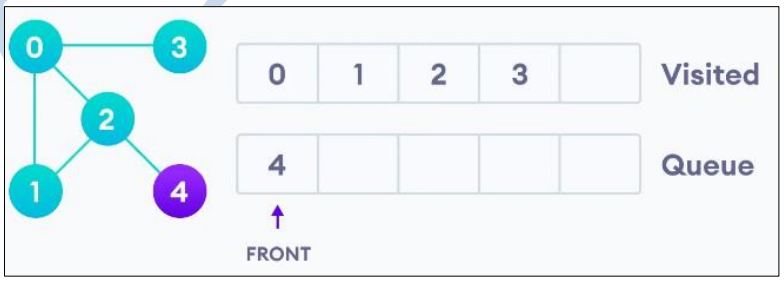



Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

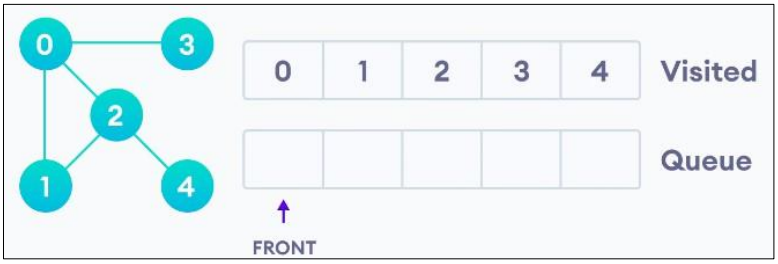



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



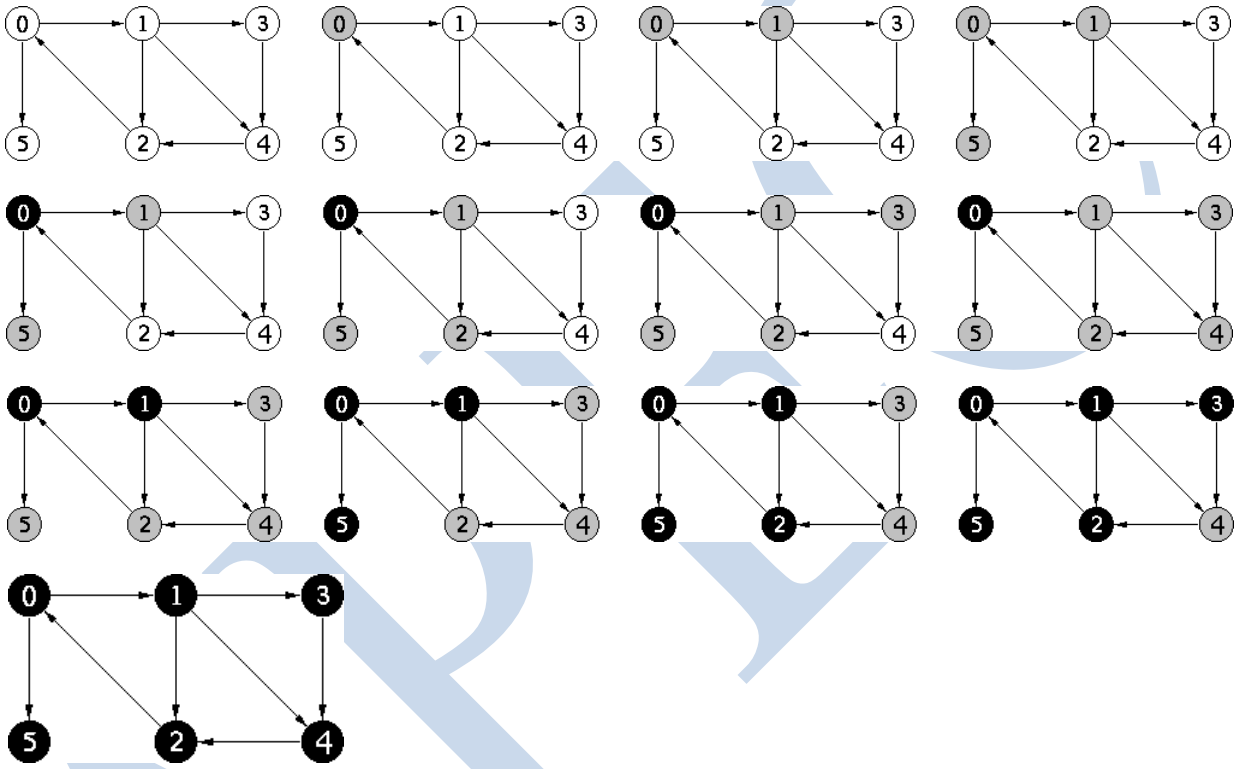




Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.




Since the queue is empty, we have completed the Breadth First Traversal of the graph.

*2.3.2.3 Example 2 – Directed Graph:*



*2.3.2.4 Python Program for Breadth-first search*

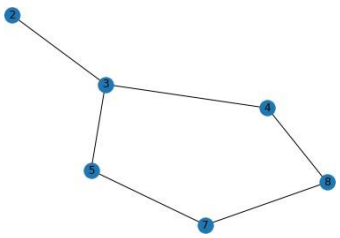
```
import networkx as nx
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
G = nx.Graph(graph)
nx.draw(G, with_labels = True)
visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
```

**Output**

Following is the Breadth-First Search

5 3 7 2 4 8



```
  visited.append(node)
  queue.append(node)

# Creating loop to visit each node
  while queue:
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')     # function calling
```

### 2.3.4 Differences between BFS and DFS

| Parameters | BFS | DFS |
|---|---|---|
| Stands for | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| Data Structure | BFS uses Queue data structure for finding the shortest path. | DFS uses Stack data structure. |
| Definition | BFS is a traversal approach in which we visit all nodes on the same level before moving on to the next level. | DFS is also a traversal approach in which the traversal begins at the root node and proceeds in depth nodes. |
| Approach used | FIFO (First In First Out). | LIFO (Last In First Out). |
| Time Complexity | Time complexity<br>O(V + E) when Adjacency List is used<br>O($V^2$) when Adjacency Matrix is used, where<br>V → vertices E<br>→ edges. | Time complexity<br>O(V + E) when Adjacency List is used<br>O($V^2$) when Adjacency Matrix is used, where<br>V → vertices<br>E → edges. |
| Space Complexity | Space complexity is O(V) V → vertices | Space complexity is O(l),<br>l→is the maximum number of nodes in a single level. |
| Visiting of Siblings/ Children | Here, siblings are visited before the children. | Here, children are visited before the siblings. |
| Backtracking | In BFS there is no concept of backtracking. | DFS algorithm is a recursive algorithm that uses the idea of backtracking |
| Applications | BFS is used in applications like | DFS is used in various applications such as |

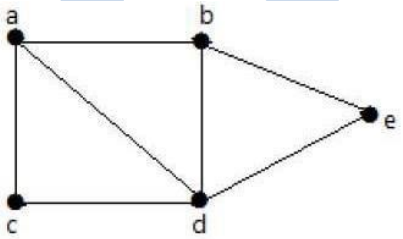
| Parameters | BFS | DFS |
|---|---|---|
| | bipartite graphs, finding shortest paths, used for finding neighbouring locations in GPS systems | acyclic graphs and topological order, used to find the solution to a puzzle — e.g., finding a path out of a maze, with the path stored in a stack. |
| Memory | BFS requires more memory. | DFS requires less memory. |
| Optimality | More optimal for finding the shortest path. | Not optimal for finding the shortest path. |
| Space complexity | In BFS, the space complexity is more critical as compared to time complexity. | DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node. |
| Speed | BFS is slow as compared to DFS. | DFS is fast as compared to BFS. |

## 2.4 Connectivity

A graph is said to be connected if there is a path between every pair of vertex. From every vertex to any other vertex, there should be some path to traverse. That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.

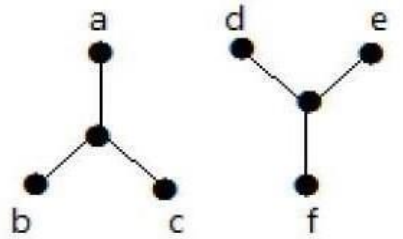
*Example 1*

In the following graph, it is possible to travel from one vertex to any other vertex. For example, one can traverse from vertex 'a' to vertex 'e' using the path 'a-b-e'.




*Example 2*

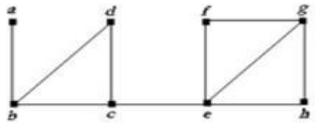
In the following example, traversing from vertex 'a' to vertex 'f' is not possible because there is no path between them directly or indirectly. Hence it is a disconnected graph.



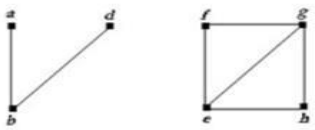

### *2.4.1 Cut Vertex*

A single vertex whose removal disconnects a graph is called a cut-vertex. Let G be a connected graph. A vertex v of G is called a cut vertex of G, if G-v (Remove v from G) results a disconnected graph. When we remove a vertex from a graph then graph will break into two or more graphs. This vertex is called a cut vertex.

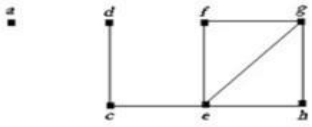
Original graph:



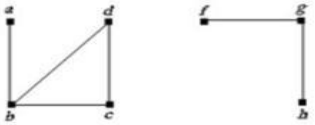
Vertex c is a cut vertex:



Vertex b is a cut vertex:



Vertex e is a cut vertex:

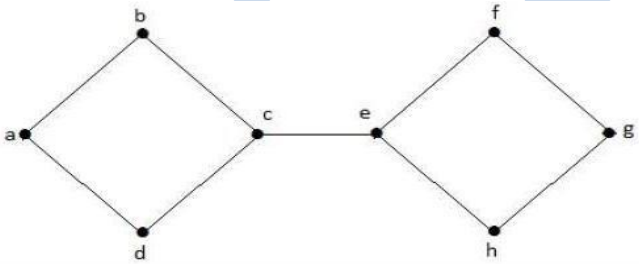


### *2.4.2 Cut Edge (Bridge)*

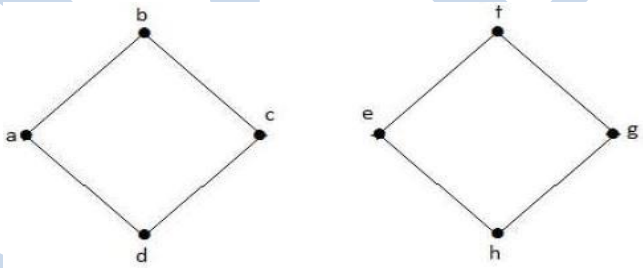
Let 'G' be a connected graph. An edge 'e' ∈ G is called a cut edge if 'G-e' results in a disconnected graph. If removing an edge in a graph results in to two or more graphs, then that edge is called a Cut Edge.

*Example*

In the following graph, the cut edge is [(c, e)].




By removing the edge (c, e) from the graph, it becomes a disconnected graph.



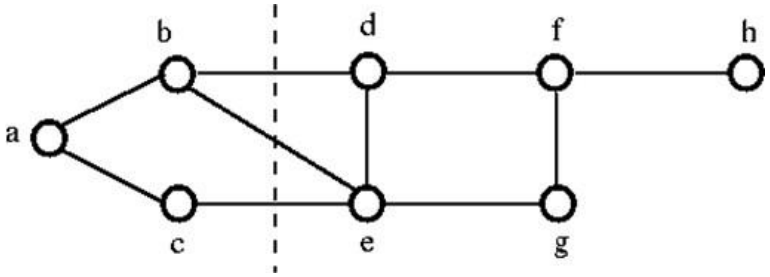

In the above graph, removing the edge (c, e) breaks the graph into two which is nothing but a disconnected graph. Hence, the edge (c, e) is a cut edge of the graph.

### *2.4.3 Cut Set*

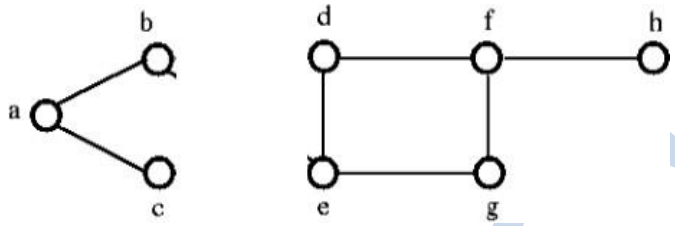
In a connected graph G, a cut set is a set S of edges with the following properties:

- The removal of all the edges in S disconnects G.
- The removal of some of edges (but not all) in S does not disconnect G.

*Example 1*



To disconnect the above graph G, we have to remove the three edges. i.e. bd, be and ce. We cannot disconnect it by removing just two of three edges. Hence, {bd, be, ce} is a cut set. After removing the cut set from the above graph, it would look like as follows:



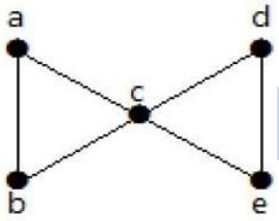

### *2.4.4 Edge Connectivity*

The **edge connectivity** of a connected graph G is the minimum number of edges whose removal makes G disconnected. It is denoted by **λ(G)**.

- When λ(G) ≥ k, then graph G is said to be **k-edge-connected.**

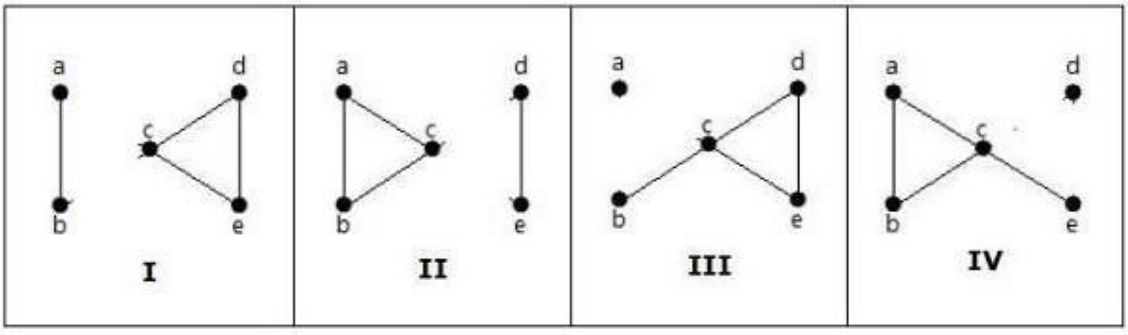
*Example*

Take a look at the following graph. By removing two minimum edges, the connected graph becomes disconnected. Hence, its edge connectivity (λ(G)) is 2.




From the above graph, by removing two minimum edges, the connected graph becomes disconnected graph. Hence, its edge connectivity is 2. Therefore the above graph is a **2-** edge-connected graph**.**

Here are the following four ways to disconnect the graph by removing two edges



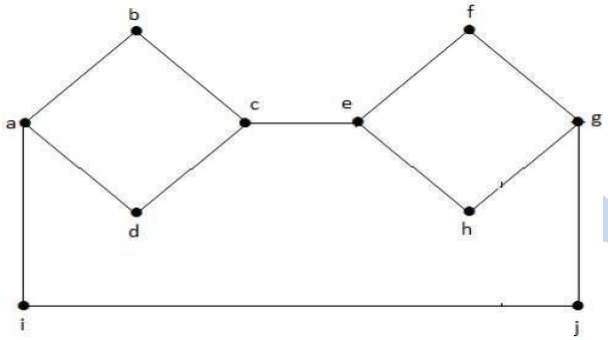

### 2.4.5 Vertex Connectivity

Let 'G' be a connected graph. The minimum number of vertices whose removal makes 'G' either disconnected or reduces 'G' in to a trivial graph is called its vertex connectivity.

- **Notation** – K(G)

*Example*

In the above graph, removing the vertices 'e' and 'i' makes the graph disconnected.



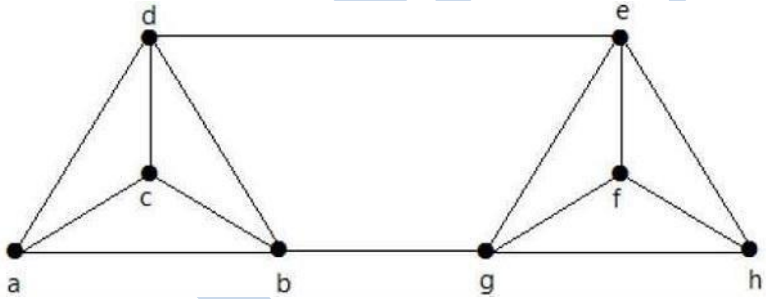

If G has a cut vertex, then K(G) = 1. Note:

- For any connected graph G, $\mathbf{K(G) \leq \lambda(G) \leq \delta(G)}$
  Vertex connectivity (K(G)), edge connectivity (λ(G)), minimum number of degrees G(δ(G)).

***Examples of Connectivity***

*Example 1*

Calculate λ(G) and K(G) for the following graph –




*Solution*

From the graph,

- δ(G) = 3
- K(G) ≤ λ(G) ≤ δ(G) = 3 (1)
- K(G) ≥ 2 (2)

Deleting the edges {d, e} and {b, h}, we can disconnect G. Therefore,

- λ(G) = 2
- 2 ≤ λ(G) ≤ δ(G) = 2 (3)
- From (2) and (3), vertex connectivity K(G) = 2

*Example 2*

If a complete graph has a total of 20 vertices, then find the number of edges it may contain.

***Solution***

The formula for the total number of edges in a k graph is given by;

Number of edges $= n(n-1)/2$
$= 20(20-1)/2$
$=10(19)$
$=190$

Hence, it contains 190 edges.

*Example 3*

If a graph has 40 edges, then how many vertices does it have?

***Solution***

Number of edges $= n\,(n-1)/2$

$40 = n(n-1)/2$

$\Rightarrow$ $n(n-1) = 80$

$n^2 - n - 80 = 0$

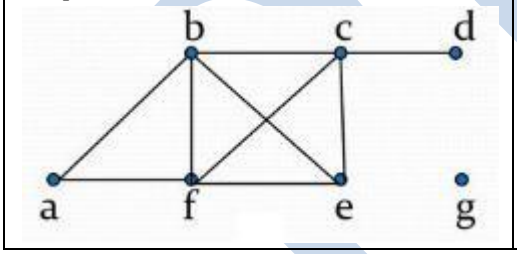
On solving the above quadratic equation, we get; $n \approx 9.45, -8.45$ Since, the number of vertices cannot be negative.

Therefore, number of vertices $\approx 9$

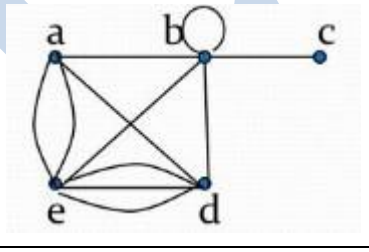
*Example 4*

Calculate the degrees for the following graphs.

Graph G




Graph H



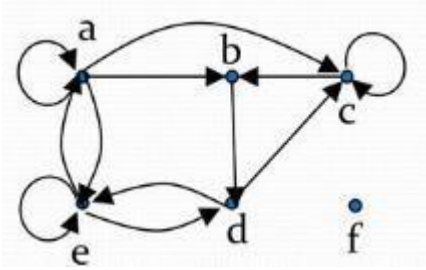

***Solution***

In Graph G, deg(a)=2, deg(b)=deg(c)=deg(f)=4, deg(d)=1, deg(e)=3, and deg(g)=0.There are two vertices d and e with odd number of degrees.

In Graph H, deg(a)=4, deg(b)=deg(e)=6, deg(c)=1, and deg(d)=5.There are two vertices c and d with odd number of degrees.

*Example 5*

Calculate the in-degrees and out-degrees for the following graph.




***Solution***

The in-degree in G are $\deg^-(a)=2$, $\deg^-(b)=2$, $\deg^-(c)=3$, $\deg^-(d)=2$, $\deg^-(e)=3$, $\deg^-(f)=0$

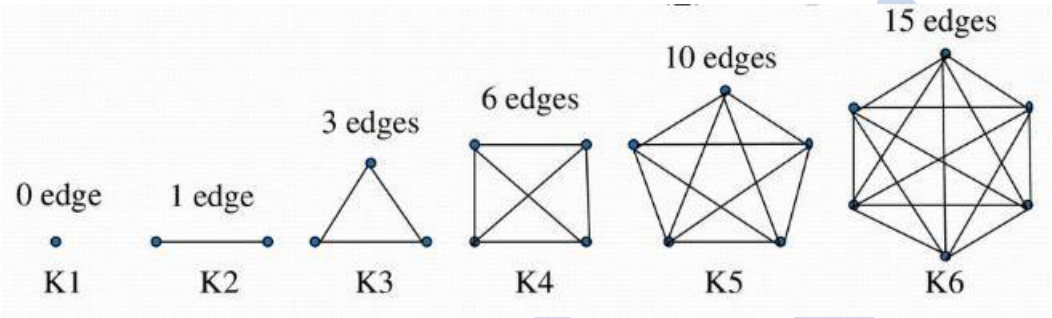
The out-degree in G are $deg^+(a)=4$, $deg^+(b)=1$, $deg^+(c)=2$, $deg^+(d)=2$, $deg^+(e)=3$, $deg^+(f)=0$

*Example 6*

How many edges does a complete graph have?

***Solution***

A complete graph on n vertices has $= n\,(n-1)/2$ edges




*Example 7*

Calculate edge connectivity for the following graph.

***Solution***

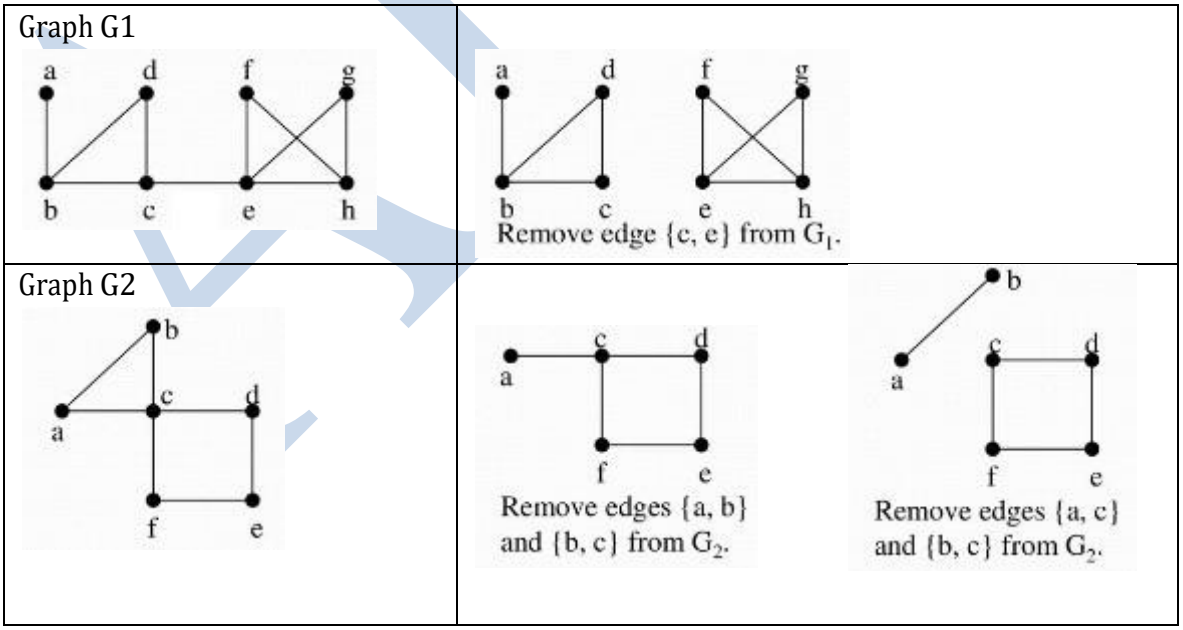
For graph G1, {{c, e}} is an edge cut. Hence, λ (G1)=1.

For graph G2, {{a, b}, {b, c}} and {{a, c}, {b, c}} are edge cuts. Hence, λ (G2)=2.

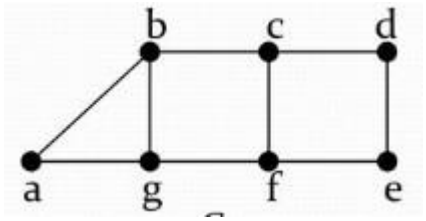
For graph G3, {{a, b},{a, g}} and {{b, c},{f, g}} are edge cuts. Hence, λ (G3)=2

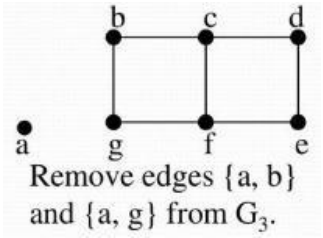
For graph G4, {{b, c}, {a, f}, {f, g}} is an edge cut. Hence, λ (G4)=3

For graph G5, {{a, b}, {a, g}, {a, h}} is an edge cut. Hence, λ (G5)=3

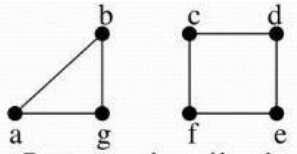


Graph G3



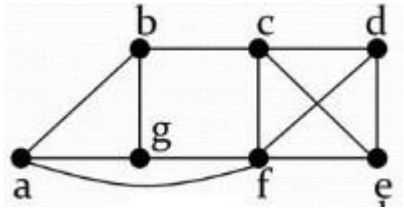



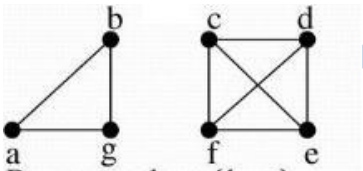

Remove edges {a, b} and {a, g} from $G_3$.




Remove edges {b, c} and {f, g} from $G_3$.

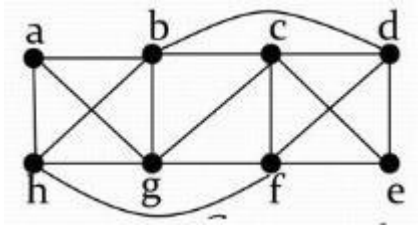
Graph G4

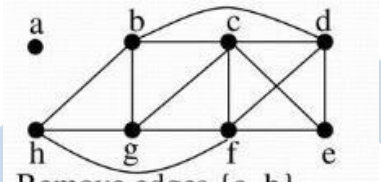



Remove edges {b, c}, {a, f}, and {f, g} from $G_4$.

Graph G5




Remove edges {a, b}, {a, g}, and {a, h} from $G_5$.

*Example 8*

Calculate vertex connectivity for the following graph.

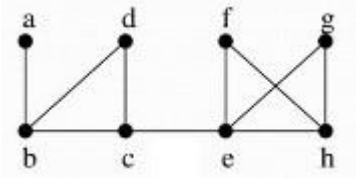
***Solution***

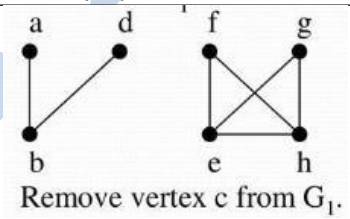
For graph G1, either {c} or {e} is a vertex cut. Hence, K(G1) = 1 For graph G2, {c} is a vertex cut. Hence, K(G2) = 1

For graph G3, either{b, g} or {c, f} is a vertex cut. Hence, K(G3) = 2 For graph G4, {c, f} is a vertex cut. Hence, K(G4) = 2

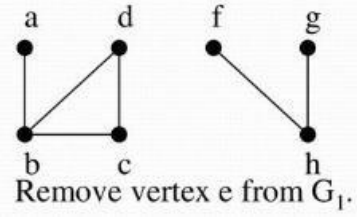
For graph G5, {b, c, f} is a vertex cut. Hence, K(G5) = 3

Graph G1

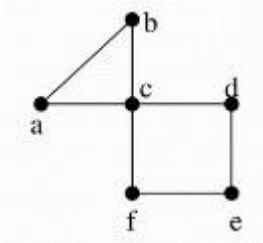
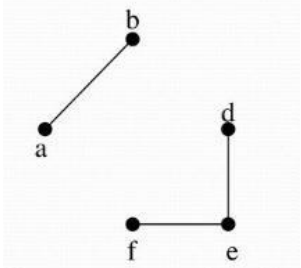
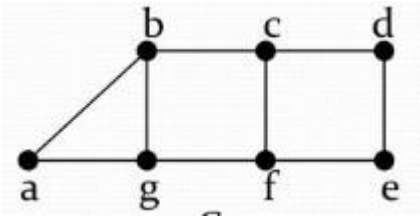
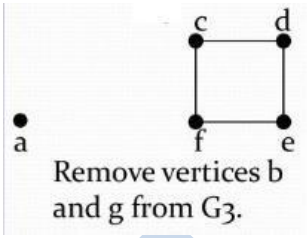
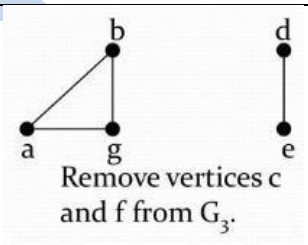
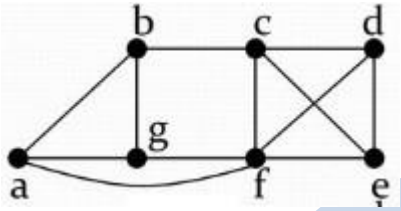
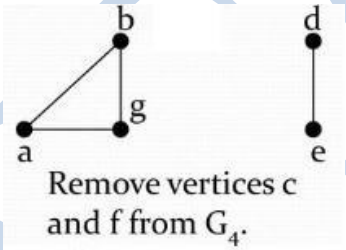
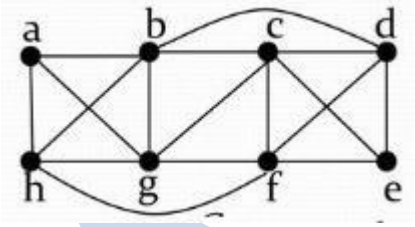
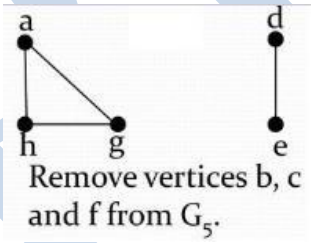



Remove vertex c from $G_1$.




Remove vertex e from $G_1$.

| | |
|---|---|
| Graph G2<br> | <br>Remove vertex c from $G_2$. |
| Graph G3<br> | <br>Remove vertices b and g from G3.<br><br>Remove vertices c and f from $G_3$. |
| Graph G4<br> | <br>Remove vertices c and f from $G_4$. |
| Graph G5<br> | <br>Remove vertices b, c and f from $G_5$. |

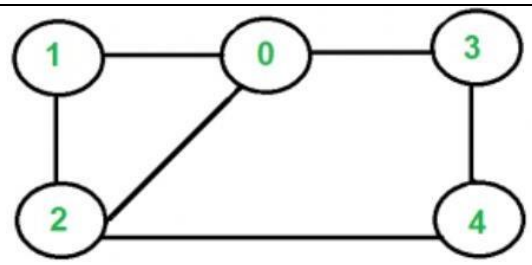
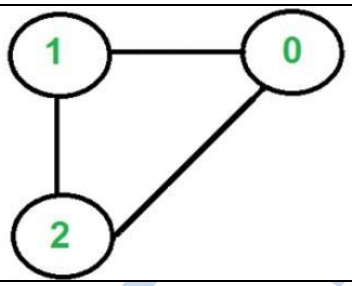
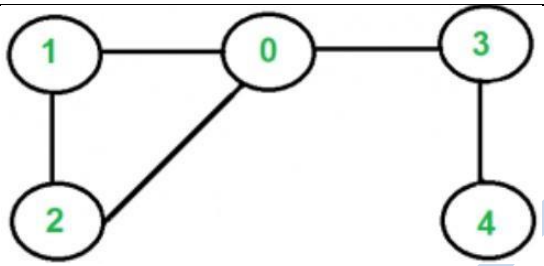
### 2.4.5 Biconnected graph

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices. We can say that a graph G is a bi-connected graph if it is connected, and there are no articulation points or cut vertex are present in the graph.

A graph is said to be Biconnected if:

- It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
- Even after removing any vertex the graph remains connected.

*Following are some examples*:

| Biconnected Graphs | |
|---|---|
|  |  |
| **Not Biconnected Graphs** | |
|  |  |

A connected graph is Biconnected if it is connected and doesn't have any Articulation Point. We mainly need to check two things in a graph.

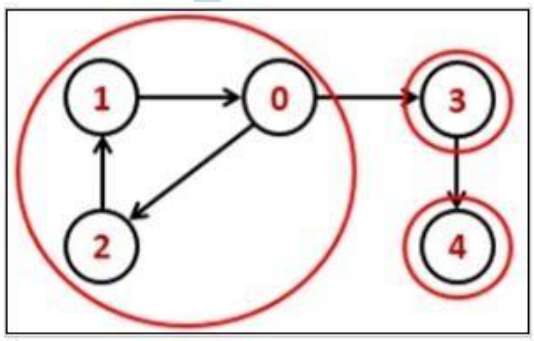
- The graph is connected.
- There is not articulation point in graph.

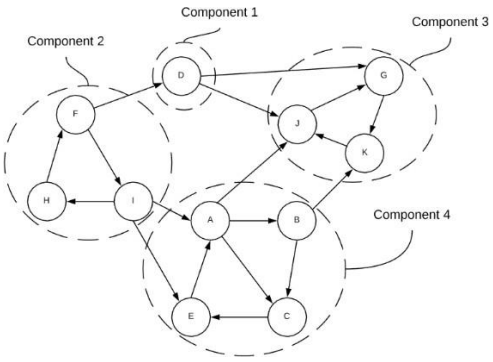
We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don't find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected.

### 2.4.6 Strongly connected

In a directed graph is said to be strongly connected, when there is a path between each pair of vertices in one component.

A strongly connected component is the portion of a directed graph in which there is a path from each vertex to another vertex. It is applicable only on a directed graph.




If each strongly connected component is treated as a single node, the graph becomes a directed acyclic graph. Just like nodes in a directed graph, a strongly connected

components with no incoming edges is known as a *source* and one without any outgoing edges is known as a *sink*.

Finding the *strongly connected components* of a directed graph is the most complex of the algorithms shown here. This algorithm (often referred to as the *Kosaraju* or *Kosaraju-Sharir* algorithm) performs DFS twice.

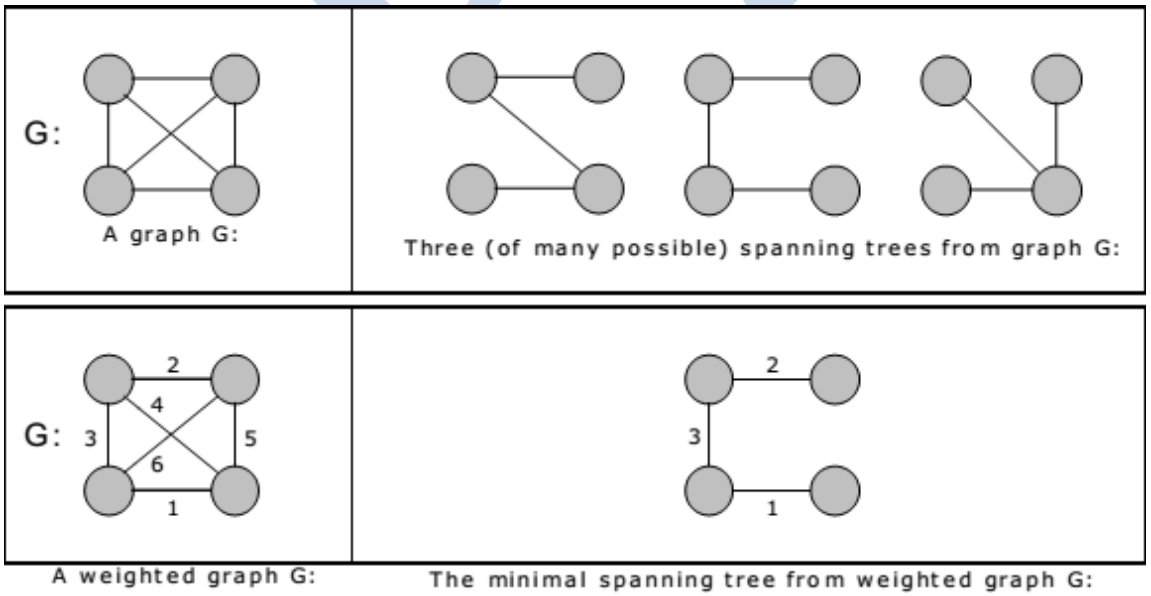
### 2.5 Define Spanning Tree

A graph can be represented as $G(V, E)$, where V is the number of vertices, and E is the number of edges. Thespanning tree is a subset to a connected graph G would be represented as G`(V`, E`).

- V` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph.
- E` = |V| - 1 means that number of edges in the spanning tree is the subset of the number of edges in the original graph ie. E` € E

#### 2.5.1 General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- Number of vertices in the spanning tree would be the same as the number of vertices in the original graph ie. V` = V
- Number of edges in the spanning tree would be equal to the number of edges minus 1 ie. E` = |V| - 1
- Spanning tree should not contain any cycle.
- Spanning tree should not be disconnected.
- Weight of the spanning tree is the sum of the weights of all edges in T.



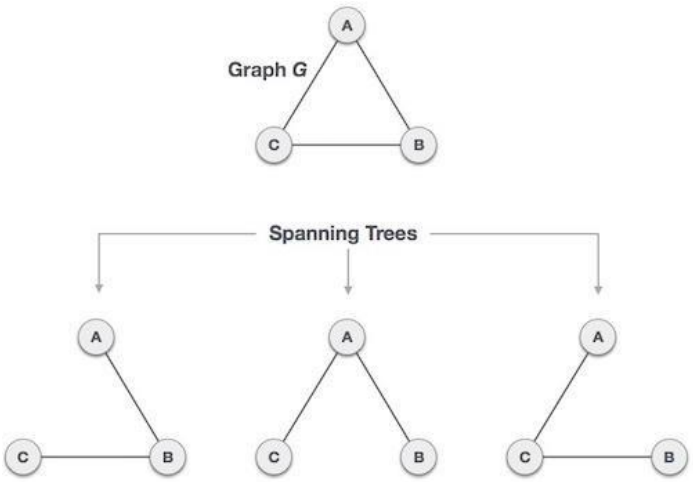



Figure . Minimum spanning Tree (MST)

We found three spanning trees for one complete graph. A complete undirected graph can have maximum $n^{n-2}$ number of spanning trees, where n is the number of vertices.

- When n = 3,number of spanning trees is $3^{3-2} = 3$.
- When n = 4,number of spanning trees is $4^{4-2} = 16$.



### 2.5.2 Minimum Spanning tree

Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

### 2.5.3 Application of Spanning Tree

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are:

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

### 2.5.4 Algorithms for finding the Minimum Spanning Tree

Famous two algorithms for finding minimum spanning trees are

- Kruskal's algorithm
- Prim's algorithm

### 2.5.4.1 Kruskal'sMinimum Spanning Tree 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. It follows the greedy approach to find an optimum solution at every stage instead of focusing on a global optimum.

#### *2.5.4.1.1 Working of Kruskal's algorithm*

Kruskal's algorithm, starts from edges with the lowest weight and keep adding the edges until the goal is reached.

*Steps*

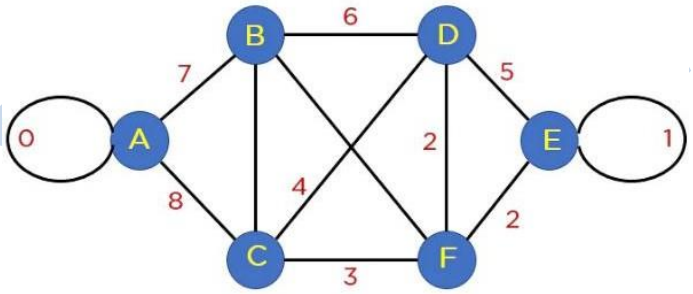
*Step 1*: First, sort all the edges from low weight to high.

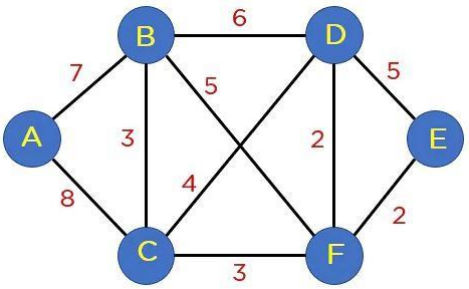
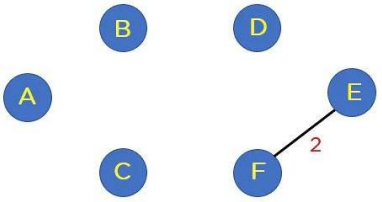
*Step 2*: Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.

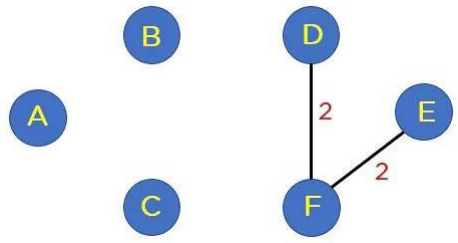
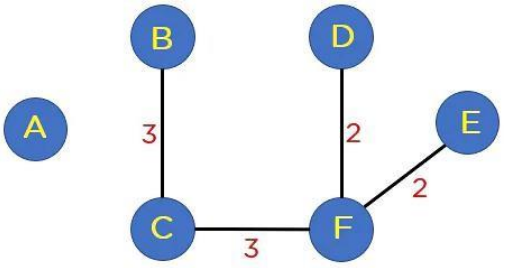
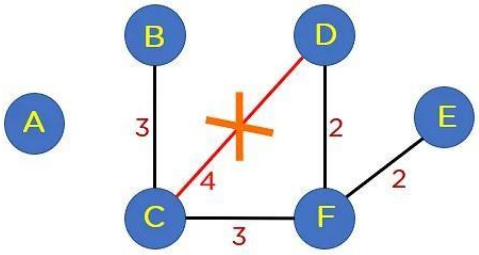
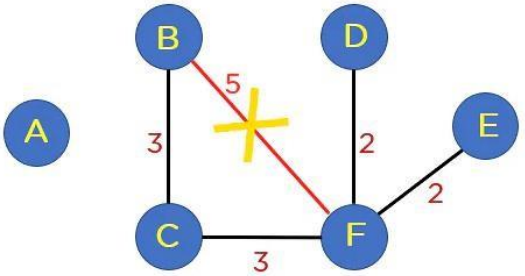
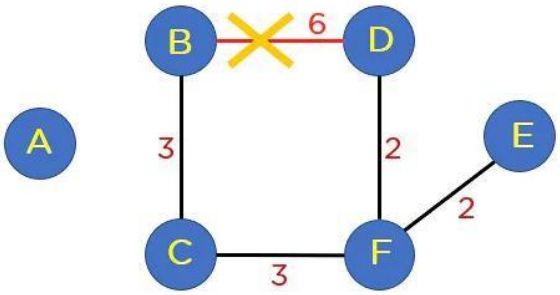
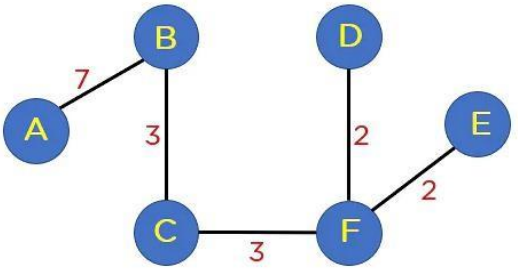
*Step 3*: Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

#### *2.5.4.1.2 Example*

Given Graph G(V,E):




| | |
|---|---|
| *Step 1:*<br>Removing parallel edges or loops from the graph.<br> | *Step 2:*<br>Include edges in the Minimum spanning Tree(MST) such that the included edge would not form a cycle in the tree structure. Pick the edge EF, as it has a minimum weight is 2.<br> |
| *Step 3:*<br>Add edge FD to the spanning tree. | *Step 4:*<br>Add edge BC and edge CF to the spanning tree as it does not generate any loop. |

<table>
<tr><td>
</td><td>
</td></tr>
<tr><td>Step 5:<br>Next add edge CD. This edge generates the loop in the tree structure. So, discard this edge.<br>
</td><td>Step 6:<br>Now add edge BF. This edge also creates the loop; hence discard it.<br>
</td></tr>
<tr><td>Step 7:<br>Next add edge BD. This edge also formulates a loop, so discard it.<br>
</td><td>Step 8:<br>Next add edge AB. By including this node, the MST includes all 5 edges, so we don't need to traverse further. The final MST is below:<br>
<br>Minimum Spanning Tree.</td></tr>
</table>

*Applications of Kruskal's algorithm*

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

*2.5.4.1.2Python program for Kruskal Algorithm using Networkx*

```
import matplotlib.pyplot as plt
# importing networkx
import networkx as nx
import pylab


# Create an empty Undirected Weighted Graph
G = nx.Graph()
```

```
# Add nodes
nodes_list = [1, 2, 3, 4, 5, 6, 7]
G.add_nodes_from(nodes_list)

# Add weighted edges
edges_list = [(1, 2, 13),(1, 4, 4),(2, 3, 2),(2, 4, 6), (2, 5, 4), (3, 5, 5),
              (3, 6, 6),(4, 5, 3),(4, 7, 4), (5, 6, 8), (5, 7, 7), (6, 7, 3)]
G.add_weighted_edges_from(edges_list)

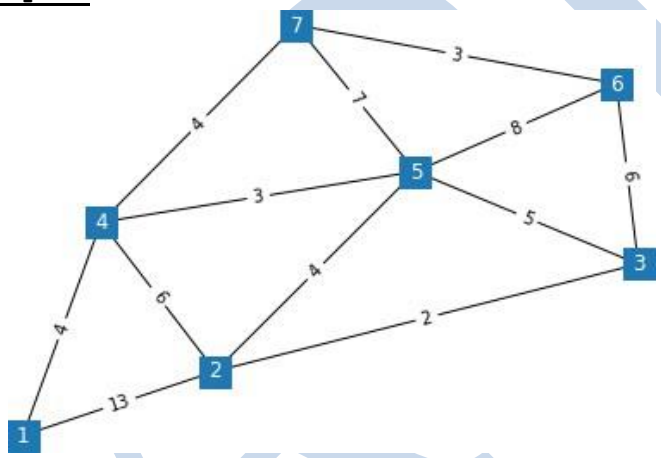
plt.figure()
pos = nx.spring_layout(G)
weight_labels = nx.get_edge_attributes(G,'weight')
nx.draw(G,pos,font_color = 'white', node_shape = 's', with_labels = True,)
nx.draw_networkx_edge_labels(G,pos,edge_labels=weight_labels)

# Calculate a minimum spanning tree of an undirected weighted graph with
# the kruskal algorithm
```

```
mst = nx.minimum_spanning_tree(G, algorithm='kruskal')
```

```
print(sorted(mst.edges(data=True)))
```

**Output:**




```
[(1, 4, {'weight': 4}), (2, 3, {'weight': 2}), (2, 5, {'weight': 4}), (4,
5, {'weight': 3}), (4, 7, {'weight': 4}), (6, 7, {'weight': 3})]
```

### 2.5.4.2 Prim's Minimum Spanning Tree Algorithm

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

#### *2.5.4.2.1 Working of prim's algorithm*

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached.

*Steps*

*Step 1*: Determine the arbitrary starting vertex.

*Step 2:* Keep repeating steps 3 and 4 until the fringe vertices (vertices not included in MST) remain.

*Step 3:* Select an edge connecting the tree vertex and fringe vertex having the minimum weight.

*Step 4:* Add the chosen edge to MST if it doesn't form any closed cycle.

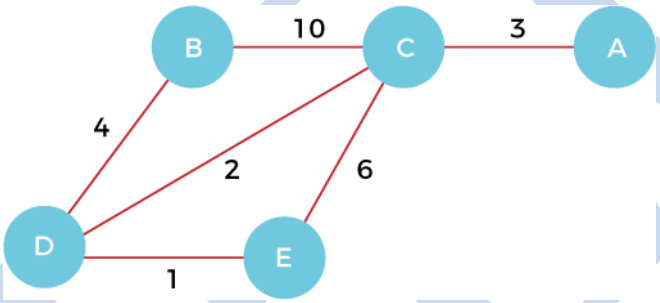
*Step 5:* Exit

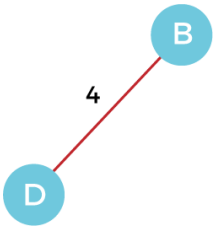
*Applications of prim's algorithm*

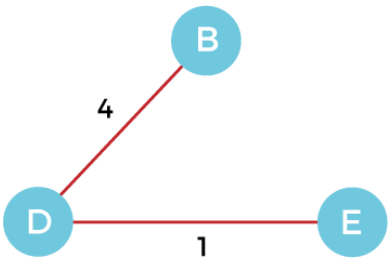
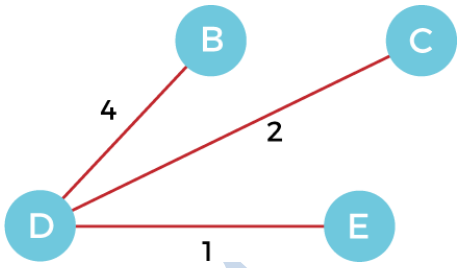
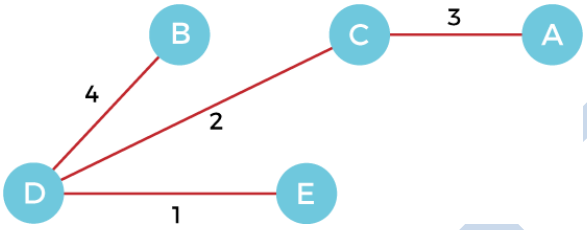
- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

*2.5.4.2.2 Example*

Given weighted graph is




| | |
|---|---|
| *Step 1*<br>First, we have to choose a vertex from the above graph. Let's choose B.<br><br> | *Step 2*<br>Add the shortest edge from vertex B. There are two edges from vertex B ie. BC with weight 10 and edge BD with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.<br><br> |
| *Step 3*<br>Again, choose the edge with the minimum weight. In this case, the edges DE and DC are such edges. DC has weight 2 and DE have weight 1.<br>So, select edge DE | *Step 4*<br>Now, select the edge CD, and add it to the MST. |

<table>
<tr>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>Step 5<br>Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.<br>
</td>
<td>Cost:<br>Cost of MST = 4 + 2 + 1 + 3 = 10 units.</td>
</tr>
</table>

*2.5.4.2.3 Python program for Prim's Algorithm using Networkx*

```
# Program 1
INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G =  [[0, 2, 7, 0, 0],
      [2, 0, 3, 8, 5],
      [7, 3, 0, 1, 0],
      [0, 8, 1, 0, 4],
      [0, 5, 0, 4, 0]]

selected_node = [0, 0, 0, 0, 0]
weight = 0
no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge      : Weight\n")
while (no_edge < N - 1):

    minimum = INF
    a = 0
    b = 0
```

```
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + " --> " + str(b) + "   : " + str(G[a][b]))
    weight = weight + G[a][b]
    selected_node[b] = True
    no_edge += 1

print("Total Weight : " , weight)
```

**Output:**

```
Edge       : Weight

0 --> 1    : 2
1 --> 2    : 3
2 --> 3    : 1
3 --> 4    : 4
Total Weight :  10
```

```
#Program 2
import matplotlib.pyplot as plt
import networkx as nx
import pylab



# Create an empty Undirected Weighted Graph
G = nx.Graph()

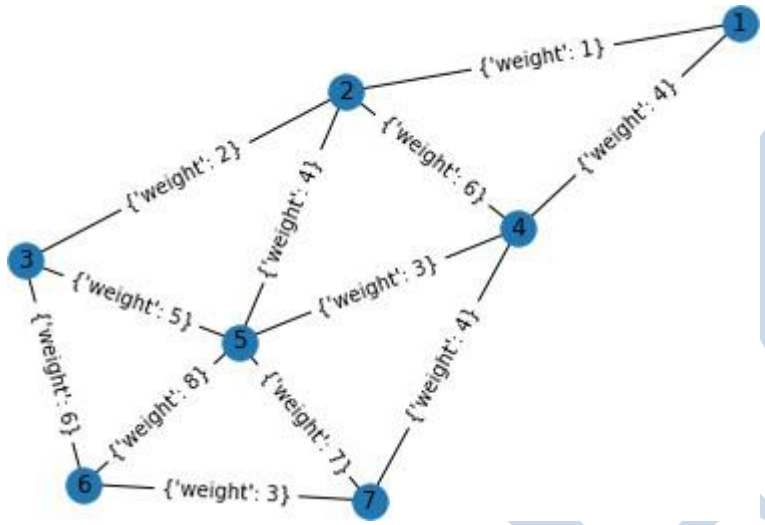
# Add nodes
nodes_list = [1, 2, 3, 4, 5, 6, 7]
G.add_nodes_from(nodes_list)

# Add weighted edges
edges_list = [(1, 2, 1), (1, 4, 4), (2, 3, 2), (2, 4, 6), (2, 5, 4), (3, 5, 5),
              (3, 6, 6), (4, 5, 3), (4, 7, 4), (5, 6, 8), (5, 7, 7), (6, 7, 3)]
G.add_weighted_edges_from(edges_list)

pos=nx.spring_layout(G)
pylab.figure(1)
nx.draw(G,pos, with_labels= 'true')
# use default edge labels
nx.draw_networkx_edge_labels(G,pos)
```

```
# Calculate a minimum spanning tree of an undirected weighted graph with
# the Prim algorithm
mst = nx.minimum_spanning_tree(G, algorithm='prim')
print(sorted(mst.edges(data=True)))
```

**Output:**




```
[(1, 2, {'weight': 1}), (1, 4, {'weight': 4}), (2, 3, {'weight': 2}), (4, 5,
{'weight': 3}), (4, 7, {'weight': 4}), (6, 7, {'weight': 3})]
```

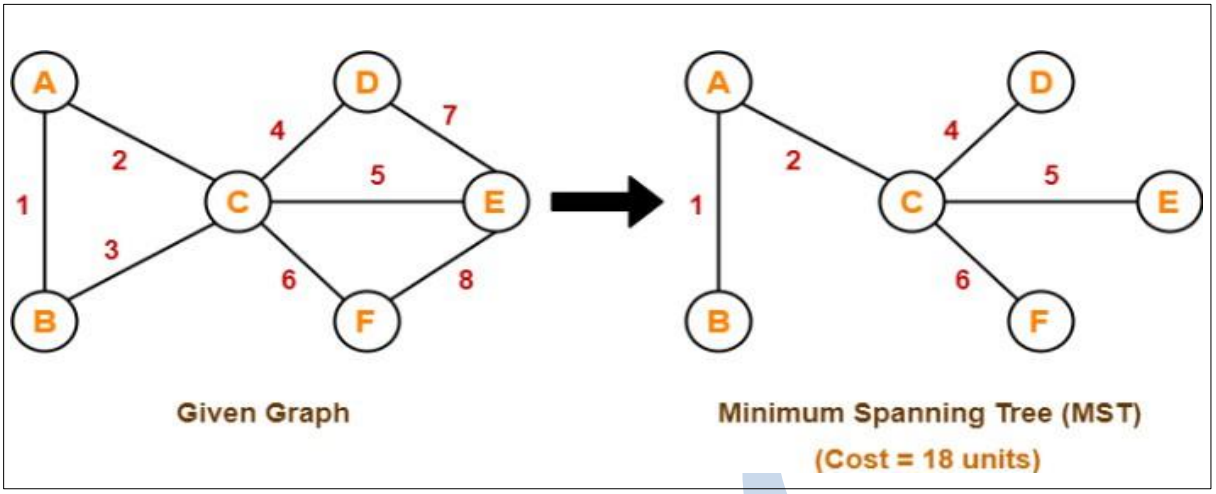
### 2.5.5 Difference between Prims and Kruskal Algorithm

| Prim's algorithm | Kruskal's algorithm |
|---|---|
| This algorithm begins to construct the minimum spanning tree from any vertex in the graph. | This algorithm begins to construct the minimum spanning tree which is having the lowest weight in the graph. |
| To obtain the minimum distance, it traverses one node more than one time. | It crosses one node only one time. |
| Time complexity is $O(V^2)$. | Time complexity is O(log V). |
| In Prim's algorithm, all the graph elements must be connected. | Kruskal's algorithm may have disconnected graphs. |
| More suitable for Dense graphs | More suitable for Sparse graphs |
| Prefers more for list data structure. | Prefers more for heap data structure. |

### 2.5.6 Important Notes

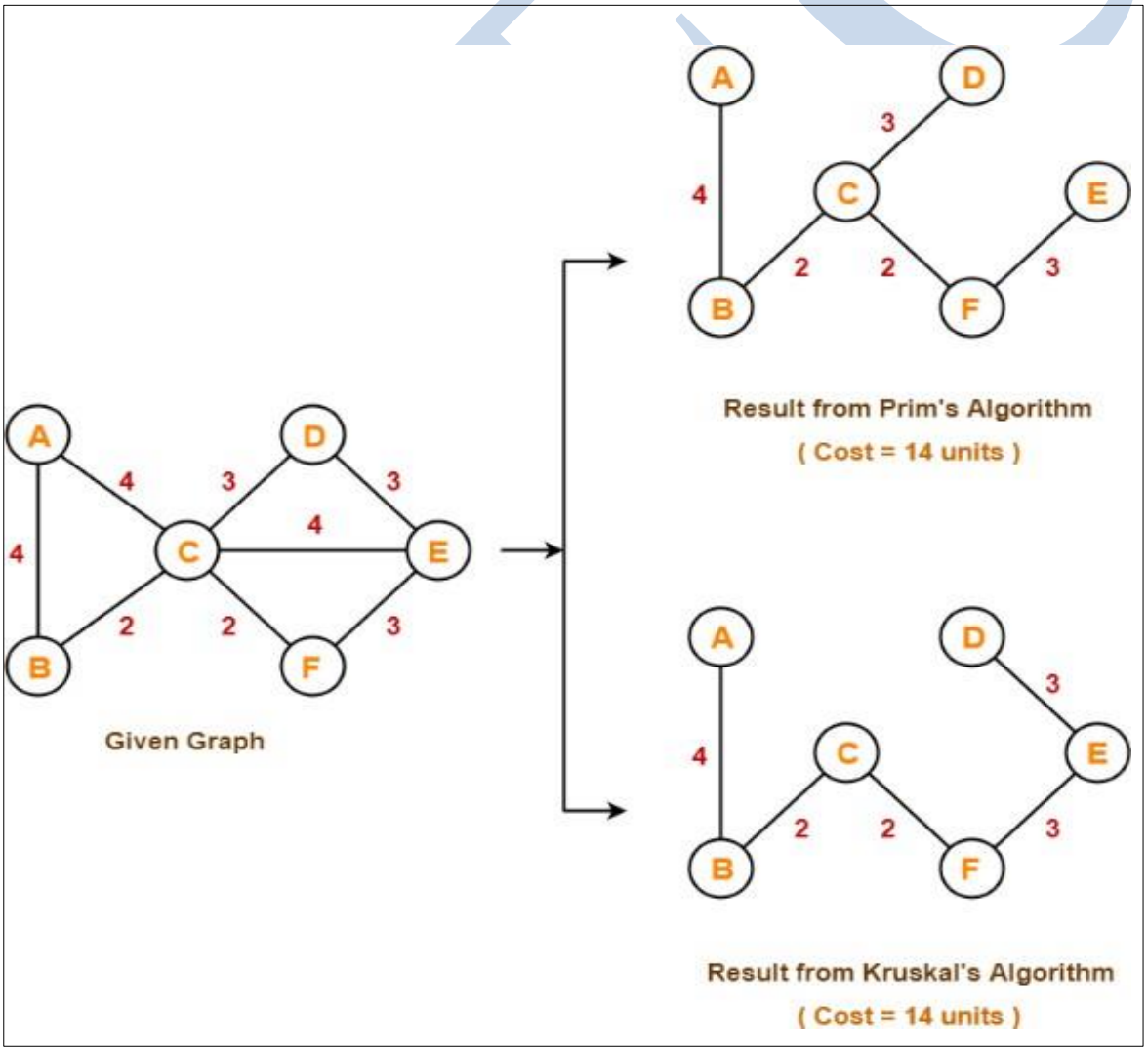
*Note 1:*

If all the edge weights are distinct, then both the algorithms are guaranteed to find the same MST.




*Note 2:*

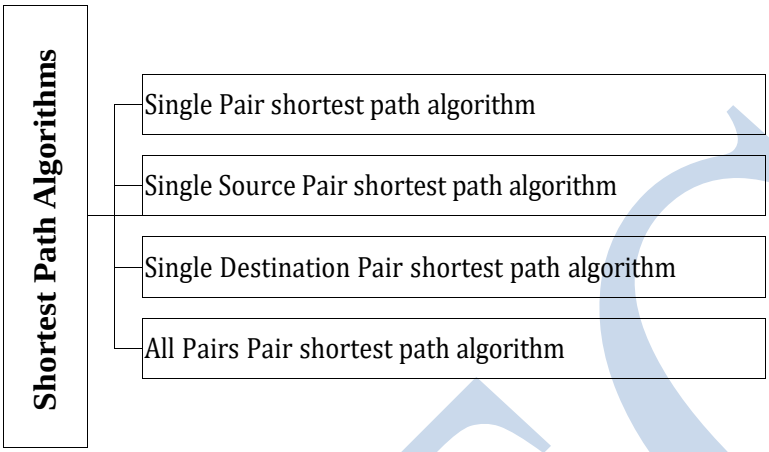
If all the edge weights are not distinct, then both the algorithms may not always produce the same MST.However, cost of both the MSTs would always be same in both the cases.



## 2.6 Shortest path Algorithms

Shortest path algorithms are a family of algorithms used for solving the shortest path problem.

### *2.6.1 Types of Shortest path algorithms*




Single-Pair Shortest Path Problem

- Here the shortest path between a given pair of vertices is computed.
- *Famous Algorithm:* A* Search Algorithm.

Single-Source Shortest Path Problem

- Here the shortest path from a given source vertex to all other remaining vertices is computed.
- *Famous Algorithms:* Dijkstra's Algorithm and Bellman Ford Algorithm.

Single-Destination Shortest Path Problem

- Here the shortest path from all the vertices to a single destination vertex is computed.
- By reversing the direction of each edge in the graph, this problem reduces to single- source shortest path problem.
- *Famous Algorithm:* Dijkstra's Algorithm is adapted for solving single-destination shortest path problem.

All Pairs Shortest Path Problem

- Here the shortest path between every pair of vertices is computed.
- *Famous Algorithms:* Floyd-Warshall Algorithm and Johnson's Algorithm.

Applications of Shortest path algorithms

- Google Maps
- Road Networks
- Logistics Research

### 2.6.2 Single Source Shortest Paths

Given a connected weighted directed graph $G(V, E)$, associated with each edge $\langle u, v\rangle \in E$, there is a weight w(u,v). The single source shortest paths (SSSP) problem is to find a shortest path from a given source r to every other vertex $v \in V - \{r\}$.The weight (length) of a path $p = \langle V_0, V_1, \ldots, V_k\rangle$ is the sum of the weights of its constituent edges:

$$W(Path\ P) = \sum_{i=1}^{k} W(V_{i-1}, V_i)$$

The weight of a shortest path from u to v is defined by

$$\delta(u, v) = min\{w(p): p\ is\ a\ path\ from\ u\ to\ v\}.$$

#### 2.6.2.1 Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

##### *2.6.2.1.1 Key points of Dijkstra's Algorithm*

- Dijkstra Algorithm is a very famous greedy algorithm.
- It is used for solving the single source shortest path problem.
- It computes the shortest path from one particular source node to all other remaining nodes of the graph.
- Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.
- Dijkstra algorithm works for directed as well as undirected connected graphs.

##### *2.6.2.1.2 Pseudocode*

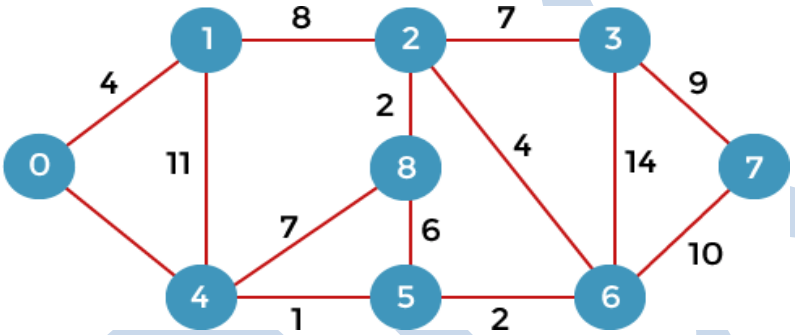
Dijkstra's algorithm in its original form, takes as input a graph with non-negative weights, a source node (starting point), and a target node (final destination), and returns the shortest path and the cost of that path. The algorithm uses two labels for each node to define the cost from the source node to the specific node and the previous node respectively. The pseudocode of the algorithm is the following:

```
1   function Dijkstra(Graph, source, target):
2
3       for each vertex v in Graph:
4            define the visited state FALSE
5            define the cost from the source node to infinity
6            define the previous node to None
7            if v == source:
8               define the cost from the source node to zero
9
10      while target is not visited:
11           selected_node ← vertex in Graph with min distance from
12                           the source node
13
```

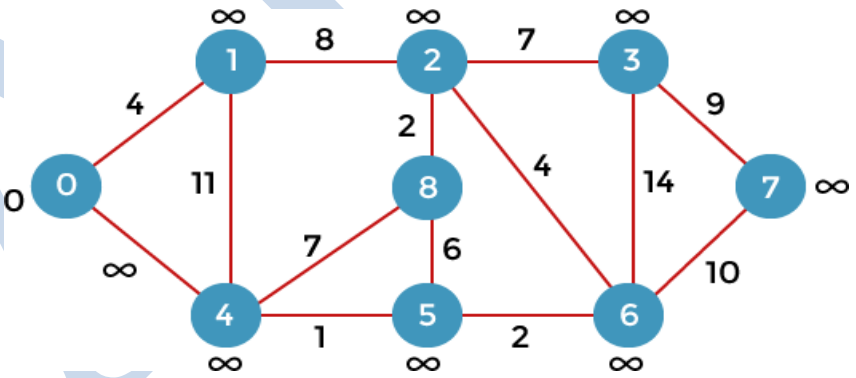
```
14          selected_node is characterized as visited
15
16          for each neighbor v of selected_node not visited:
17              alt ← selected_node distance from source node +
18                    length(selected_node, v)
19              if alt <neighbor distance from source node:
20                  neighbor distance from source node ← alt
21                  neighbor previous node ← selected_node
22      calculate the path
23      return path, target node distance from start
```

*2.6.2.1.3 Examples for Dijkstra's algorithm*

*Example 1 – Undirected Graph*




First, we have to consider any vertex as a source vertex. Suppose we consider vertex 0 as a source vertex.Here we assume that 0 as a source vertex, and distance to all the other vertices is infinity. Initially, we do not know the distances. First, we will find out the vertices which are directly connected to the vertex 0. From the above graph we can observe that two vertices are directly connected to vertex 0.




Let's assume that the vertex 0 is represented by 'x' and the vertex 1 is represented by 'y'. The distance between the vertices can be calculated by using the below algorithm:

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (0 + 4) <∞

= 4 <∞

Since 4 < ∞ so we will update d(y) from ∞ to 4.

Now we consider vertex 0 same as 'x' and vertex 4 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (0 + 8) <∞

= 8 <∞

Therefore, the value of d(y) is 8. We replace the infinity value of vertices 1 and 4 with the values 4 and 8 respectively. Now, we have found the shortest path from the vertex 0 to 1 and 0 to 4. Therefore, vertex 0 is selected. Now, we will compare all the vertices except the vertex 0. Since vertex 1 has the lowest value, i.e., 4; therefore, vertex 1 is selected.

Since vertex 1 is selected, so we consider the path from 1 to 2, and 1 to 4. We will not consider the path from 1 to 0 as the vertex 0 is already selected.

First, we calculate the distance between the vertex 1 and 2. Consider the vertex 1 as 'x', and the vertex 2 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (4 + 8) < ∞

= 12 < ∞

Now, we calculate the distance between the vertex 1 and vertex 4. Consider the vertex 1 as 'x' and the vertex 4 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (4 + 11) < 8

= 15 < 8

Since 15 is not less than 8, we will not update the value d(4) from 8 to 12.

Till now, two nodes have been selected, i.e., 0 and 1. Now we have to compare the nodes except the node 0 and 1. The node 4 has the minimum distance, i.e., 8. Therefore, vertex 4 is selected.

Since vertex 4 is selected, so we will consider all the direct paths from the vertex 4. The direct paths from vertex 4 are 4 to 0, 4 to 1, 4 to 8, and 4 to 5. Since the vertices 0 and 1 have already been selected so we will not consider the vertices 0 and 1. We will consider only two vertices, i.e., 8 and 5.

First, we consider the vertex 8. First, we calculate the distance between the vertex 4 and 8. Consider the vertex 4 as 'x', and the vertex 8 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (8 + 7) < ∞

= 15 < ∞

Since 15 is less than the infinity so we update d(8) from infinity to 15.

Now, we consider the vertex 5. First, we calculate the distance between the vertex 4 and 5. Consider the vertex 4 as 'x', and the vertex 5 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```
= (8 + 1) < ∞

= 9 < ∞

Since 5 is less than the infinity, we update d(5) from infinity to 9.

Till now, three nodes have been selected, i.e., 0, 1, and 4. Now we have to compare the nodes except the nodes 0, 1 and 4. The node 5 has the minimum value, i.e., 9. Therefore, vertex 5 is selected.

Since the vertex 5 is selected, so we will consider all the direct paths from vertex 5. The direct paths from vertex 5 are 5 to 8, and 5 to 6.

First, we consider the vertex 8. First, we calculate the distance between the vertex 5 and 8. Consider the vertex 5 as 'x', and the vertex 8 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```
= (9 + 15) < 15

= 24 < 15

Since 24 is not less than 15 so we will not update the value d(8) from 15 to 24.

Now, we consider the vertex 6. First, we calculate the distance between the vertex 5 and 6. Consider the vertex 5 as 'x', and the vertex 6 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```
= (9 + 2) < ∞<>

= 11 < ∞

Since 11 is less than infinity, we update d(6) from infinity to 11.

Till now, nodes 0, 1, 4 and 5 have been selected. We will compare the nodes except the selected nodes. The node 6 has the lowest value as compared to other nodes. Therefore, vertex 6 is selected. Since vertex 6 is selected, we consider all the direct paths from vertex 6. The direct paths from vertex 6 are 6 to 2, 6 to 3, and 6 to 7.

First, we consider the vertex 2. Consider the vertex 6 as 'x', and the vertex 2 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```
= (11 + 4) < 12

= 15 < 12

Since 15 is not less than 12, we will not update d(2) from 12 to 15

Now we consider the vertex 3. Consider the vertex 6 as 'x', and the vertex 3 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```
= (11 + 14) < ∞

= 25 < ∞

Since 25 is less than ∞, so we will update d(3) from ∞ to 25.

Now we consider the vertex 7. Consider the vertex 6 as 'x', and the vertex 7 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (11 + 10) < ∞

= 22 < ∞

Since 22 is less than ∞ so, we will update d(7) from ∞ to 22.

Till now, nodes 0, 1, 4, 5, and 6 have been selected. Now we have to compare all the unvisited nodes, i.e., 2, 3, 7, and 8. Since node 2 has the minimum value, i.e., 12 among all the other unvisited nodes. Therefore, node 2 is selected. Since node 2 is selected, so we consider all the direct paths from node 2. The direct paths from node 2 are 2 to 8, 2 to 6, and 2 to 3.

First, we consider the vertex 8. Consider the vertex 2 as 'x' and 8 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (12 + 2) < 15

= 14 < 15

Since 14 is less than 15, we will update d(8) from 15 to 14.

Now, we consider the vertex 6. Consider the vertex 2 as 'x' and 6 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (12 + 4) < 11

= 16 < 11

Since 16 is not less than 11 so we will not update d(6) from 11 to 16. Now, we consider the vertex 3. Consider the vertex 2 as 'x' and 3 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (12 + 7) < 25

= 19 < 25

Since 19 is less than 25, we will update d(3) from 25 to 19.

Till now, nodes 0, 1, 2, 4, 5, and 6 have been selected. We compare all the unvisited nodes, i.e., 3, 7, and 8. Among nodes 3, 7, and 8, node 8 has the minimum value. The nodes which are directly connected to node 8 are 2, 4, and 5. Since all the directly connected nodes are selected so we will not consider any node for the updation.

The unvisited nodes are 3 and 7. Among the nodes 3 and 7, node 3 has the minimum value, i.e., 19. Therefore, the node 3 is selected. The nodes which are directly connected to the node 3 are 2, 6, and 7. Since the nodes 2 and 6 have been selected so we will consider these two nodes. Now, we consider the vertex 7. Consider the vertex 3 as 'x' and 7 as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

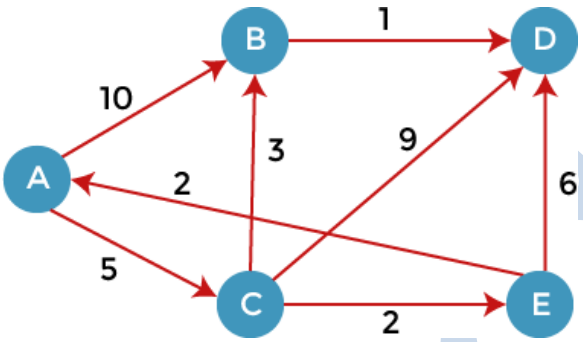
= (19 + 9) < 21

= 28 < 21

Since 28 is not less than 21, so we will not update d(7) from 28 to 21.

*Example 2 – Directed Graph*

Let's consider the directed graph.




Here, we consider A as a source vertex. A vertex is a source vertex so entry is filled with 0 while other vertices filled with ∞. The distance from source vertex to source vertex is 0, and the distance from the source vertex to other vertices is ∞. We

will solve this problem using the below table:

| A | B | C | D | E |
|---|---|---|---|---|
| ∞ | ∞ | ∞ | ∞ | ∞ |

Since 0 is the minimum value in the above table, so we select vertex A and added in the second row shown as below:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |

As we can observe in the above graph that there are two vertices directly connected to the vertex A, i.e., B and C. The vertex A is not directly connected to the vertex E, i.e., the edge is from E to A. Here we can calculate the two distances, i.e., from A to B and A to C. The same formula will be used as in the previous problem.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 5 | ∞ | ∞ |

As we can observe in the third row that 5 is the lowest value so vertex C will be added in the third row. We have calculated the distance of vertices B and C from A. Now we will compare the vertices to find the vertex with the lowest value. Since the vertex C has the minimum value, i.e., 5 so vertex C will be selected. Since the vertex C is selected, so we consider all the direct paths

from the vertex C. The direct paths from the vertex C are C to B, C to D, and C to E. First, we consider the vertex B. We calculate the distance from C to B. Consider vertex C as 'x' and vertex B as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

$= (5 + 3) < \infty$

$= 8 < \infty$

Since 8 is less than the infinity so we update d(B) from ∞ to 8. Now the new row will be inserted in which value 8 will be added under the B column.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 5 | ∞ | ∞ |
| | | 8 | | | |

We consider the vertex D. We calculate the distance from C to D. Consider vertex C as 'x' and vertex D as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

$= (5 + 9) < \infty$

$= 14 < \infty$

Since 14 is less than the infinity so we update d(D) from ∞ to 14. The value 14 will be added under the D column.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C | | 10 | 5 | ∞ | ∞ |
| | | 8 | | 14 | |

We consider the vertex E. We calculate the distance from C to E. Consider vertex C as 'x' and vertex E as 'y'.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

$= (5 + 2) < \infty$

$= 7 < \infty$

Since 14 is less than the infinity so we update d(D) from ∞ to 14. The value 14 will be added under the D column.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C | | 10 | 5 | ∞ | ∞ |
| | | **8** | | 14 | 7 |

As we can observe in the above table that 7 is the minimum value among 8, 14, and 7. Therefore, the vertex E is added on the left as shown in the below table:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C | | 10 | 5 | ∞ | ∞ |
| E | | 8 | | 14 | 7 |

The vertex E is selected so we consider all the direct paths from the vertex E. The direct paths from the vertex E are E to A and E to D. Since the vertex A is selected, so we will not consider the path from E to A.

Consider the path from E to D.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
```

= (7 + 6) < 14

= 13 < 14

Since 13 is less than the infinity so we update d(D) from ∞ to 13. The value 13 will be added under the D column.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C | | 10 | 5 | ∞ | ∞ |
| E | | 8 | | 14 | 7 |
| B | | 8 | | 13 | |

The value 8 is minimum among 8 and 13. Therefore, vertex B is selected. The direct path from B is B to D.

```
If d(x) + c(x, y) < d(y) then
      d(y) = d(x) + c(x, y)
             = (8 + 1) < 13
             = 9 < 13
```

Since 9 is less than 13 so we update d(D) from 13 to 9. The value 9 will be added under the D column.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | ∞ | ∞ | ∞ | ∞ |
| C | | 10 | 5 | ∞ | ∞ |
| E | | 8 | | 14 | 7 |
| B | | 8 | | 13 | |
| D | | | | 9 | |

### 2.6.2.2 Bellman ford algorithm

Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph.

#### *2.6.2.2.1 Key points of Bellman ford Algorithm*

- Dijkstra'salgorithm is a Greedy algorithm. Bellman-Ford follows the dynamic Programming approach.
- Bellman Ford's Algorithm works for Graphs with negative weights and is simpler than Dijkstra.
- Bellman-Ford works better (better than Dijkstra's) for distributed systems.
- Bellman-Ford does not work with an undirected graph with negative edges as it will be declared as a negative cycle.

#### *2.6.2.2.2 Pseudocode*

Bellman ford algorithm follows the dynamic programming approach by overestimating the length of the path from the starting vertex to all other vertices. And then it starts relaxing the estimates by discovering the new paths which are shorter than the previous ones. This process is followed by all the vertices for N-1 times for finding the optimized result.The pseudocode of the algorithm is the following:

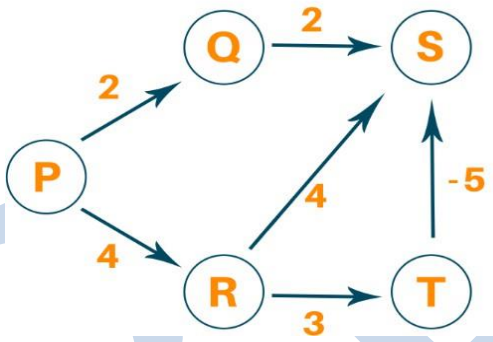
Input : A graph representing the network; and a source node, s Output :
Shortest path from s to all other nodes.

1. Initialize distances from s to all nodes as infinite (∞); distance to itself as 0; an array **dist[]** of size |**V**| (number of nodes) with all values as ∞ except **dist[s]**.
2. Calculate the shortest distances iteratively. Repeat |**V**|- **1** times for each node except s –
   Repeat for each edge connecting vertices u and v –
   If **dist[v] > (dist[u] + weight of edge u-v)**, Then Update
   **dist[v] = dist[u] + weight of edge u-v**
3. The array **dist[]** contains the shortest path from s to every other node.

*2.6.2.2.3 Example for Bellman Ford algorithm*

*Example 1:*

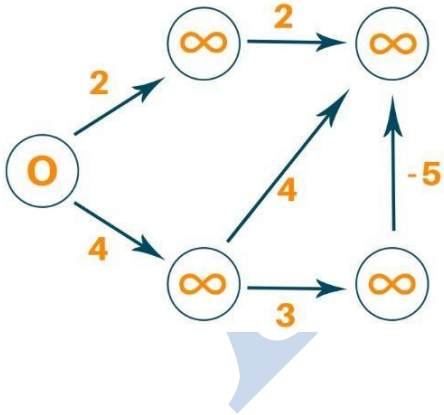
Consider the following weighted graph




*Solution*

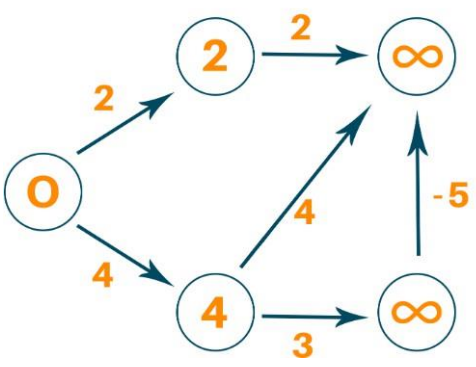
*Step 1:*

Select the source vertex with path value 0 and all other vertices as infinity.



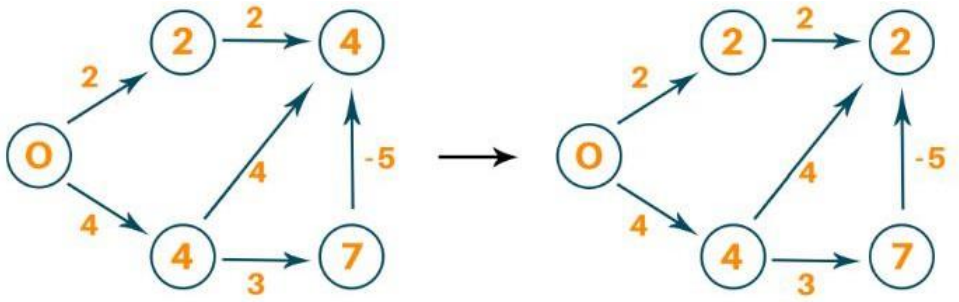

*Step 2:*

Visit the neighbouring edge from the source vertex and relax the path length of the neighbouring vertex if the new calculated path length is smaller than the previous path length




*Step 3:*

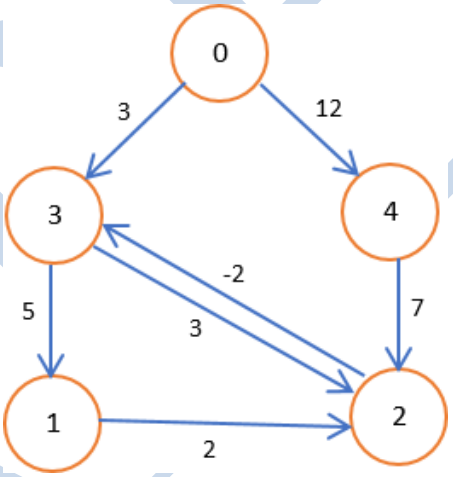
This process must be followed N-1 times where N is the total number of vertices. This is because in the worst-case scenario any vertex's path length can be changed to an even smaller path length for N times.



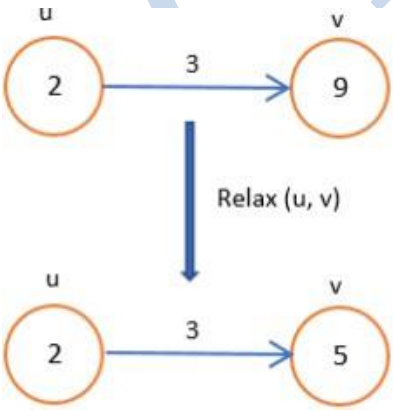

Therefore after N-1 iterations, we find our new path lengths and we can check if the graph has a negative cycle or not.

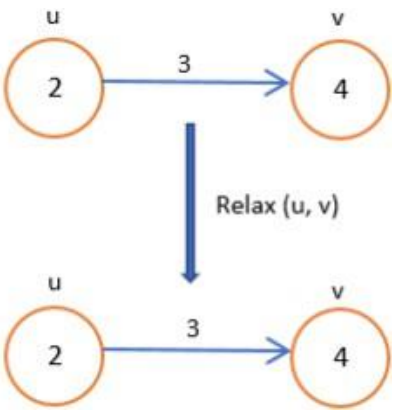
| | Q | R | S | T |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | 2 | 4 | ∞ | ∞ |
| 0 | 2 | 4 | 4 | 7 |
| 0 | 2 | 4 | 2 | 7 |
| 0 | 2 | 4 | 2 | 7 |

*Example 2:*

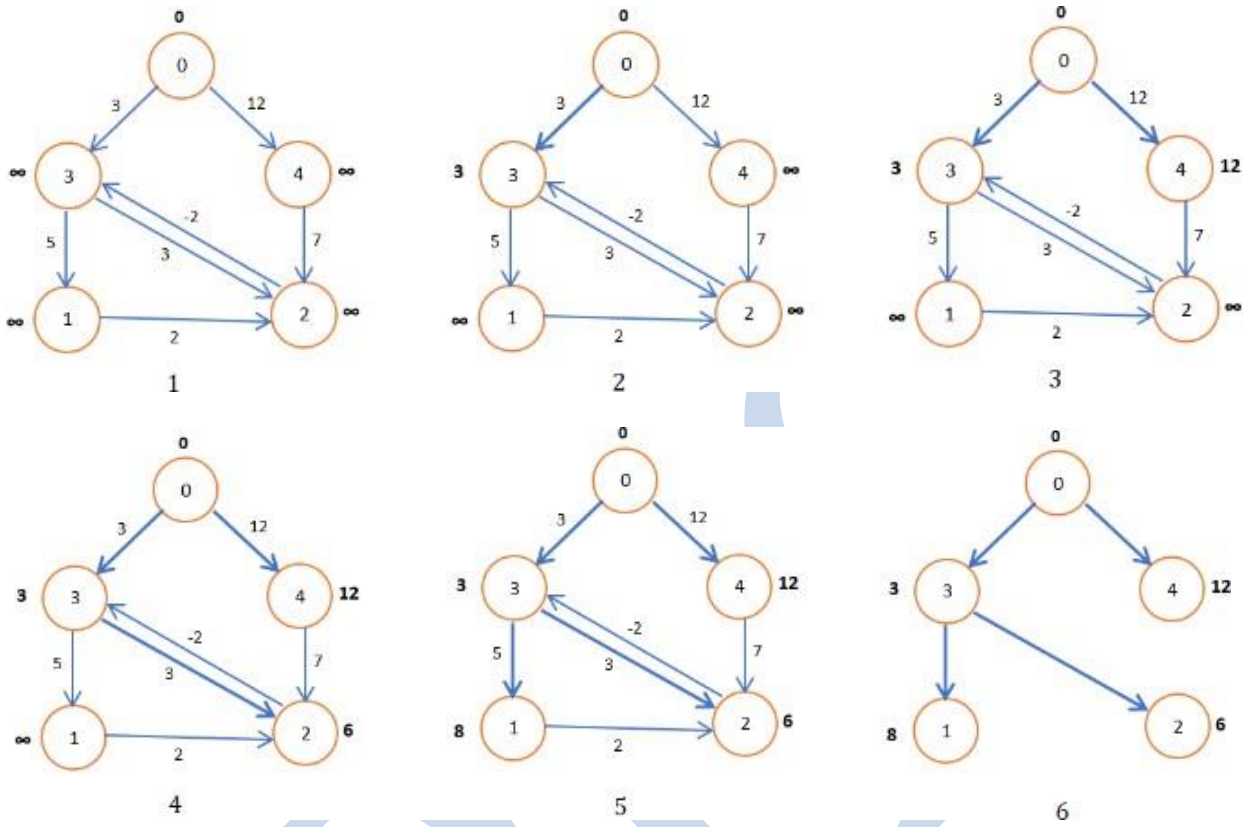



Relax compares a destination weight with addition of weight of source vertex and weight of weight and sets which is smaller, given in figure below:



*Solution:*




*2.6.2.2.4 Comparison between Dijkstra's Algorithm & Bellman – Ford algorithm*

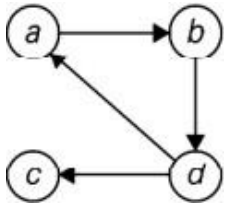
| Description | Dijkstra's Algorithm | Bellman – Ford algorithm |
|---|---|---|
| Positive Weights | Works correctly for directed and Undirected graphs | Works correctly for directed and Undirected graphs |
| Negative weights | Fails | Works correctly for directed graphs only. |

### 2.6.2.3 Floyd Warshall algorithm

*2.6.2.3.1 Transitive closure of a graph*

In any Directed Graph, let's consider a node i as a starting point and another node j as ending point. For all (i,j) pairs in a graph, transitive closure matrix is formed by the reachability factor, i.e if j is reachable from i (means there is a path from i to j) then we can put the matrix element as 1 or else if there is no path, then we can put it as 0.

*2.6.2.3.1 Transitive closure of a graph*




Digraph

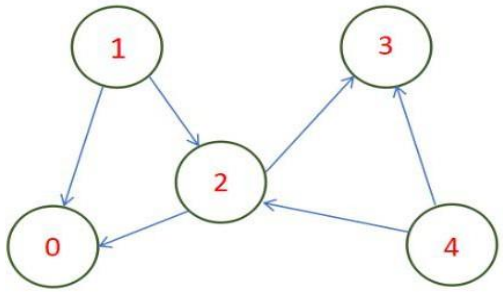
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Transitive Closure

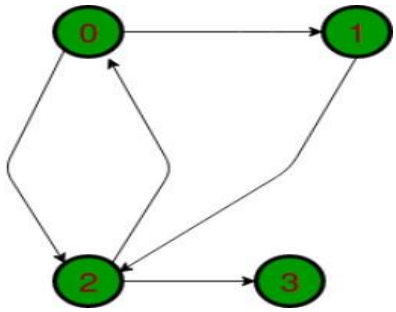
*Example 1*




Reachability matrix of the graph

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | 1 | 1 |

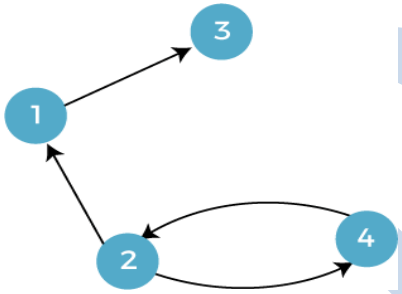
*Example 2*




Transitive closure of the graph

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 1 |

*Example 3*



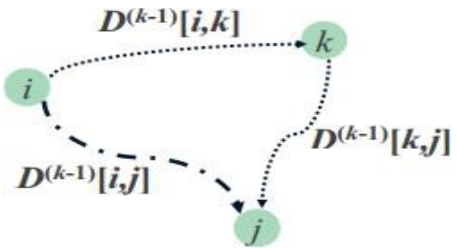

Transitive closure of the graph

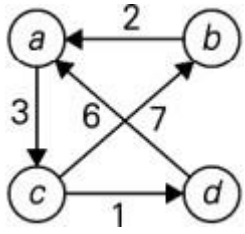
| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 1 |

*2.6.2.3.2 Floyd Warshall Algorithm : All pairs shortest paths*

On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among 1,…,k as intermediate

$$D^{(k)}[i,j] = min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



| | | |
|---|---|---|
|  Digraph | $W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$ Weight Matrix | $D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$ Distance Matrix |

*2.6.2.3.2 Key points of Floyd Warshall Algorithm*

- Floyd Warshall Algorithm is used to solve All Pairs Shortest Path Problem.
- It computes the shortest path between every pair of vertices of the given graph.
- It follows dynamic programming approach.
- Floyd Warshall Algorithm is best suited for dense graphs.
- Its complexity depends only on the number of vertices in the given graph.

*2.6.2.3.3 Pseudocode*

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

*Example 1:*

Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

*Solution:*

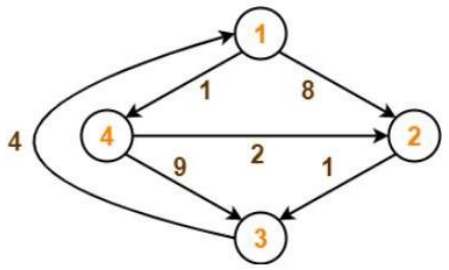
*Step-01:*

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

*Step-02:*

- Write the initial distance matrix. It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- Vertices having a direct edge between them, distance = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

*Given Digraph*




*Initial distance matrix*

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

*Step-03:*

Using Floyd Warshall Algorithm, write the following 4 matrices-

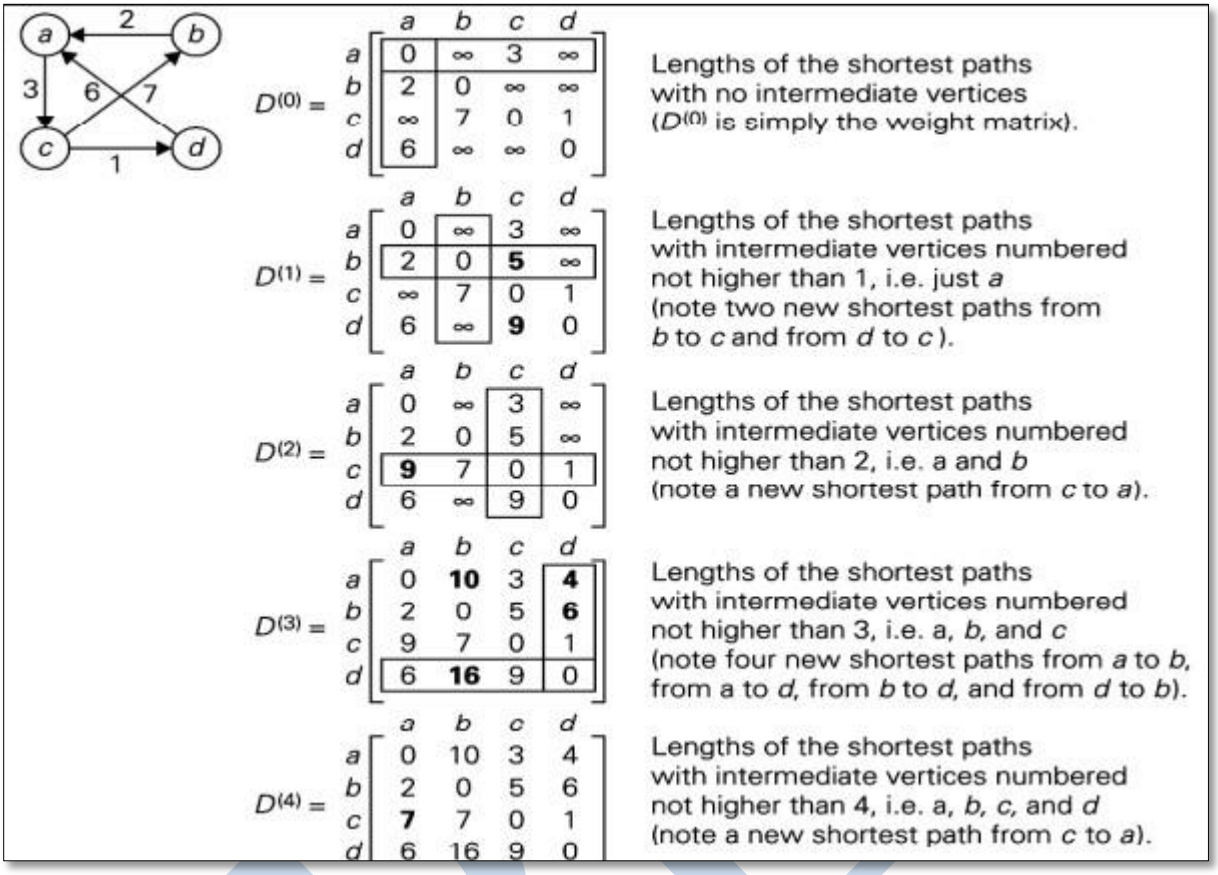
$$D_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D_3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

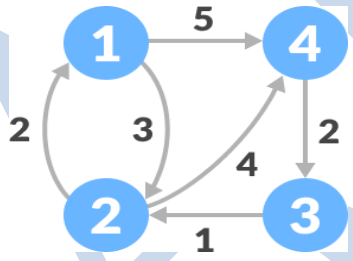
$$D_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

*Example 2:*




*Example 3:*

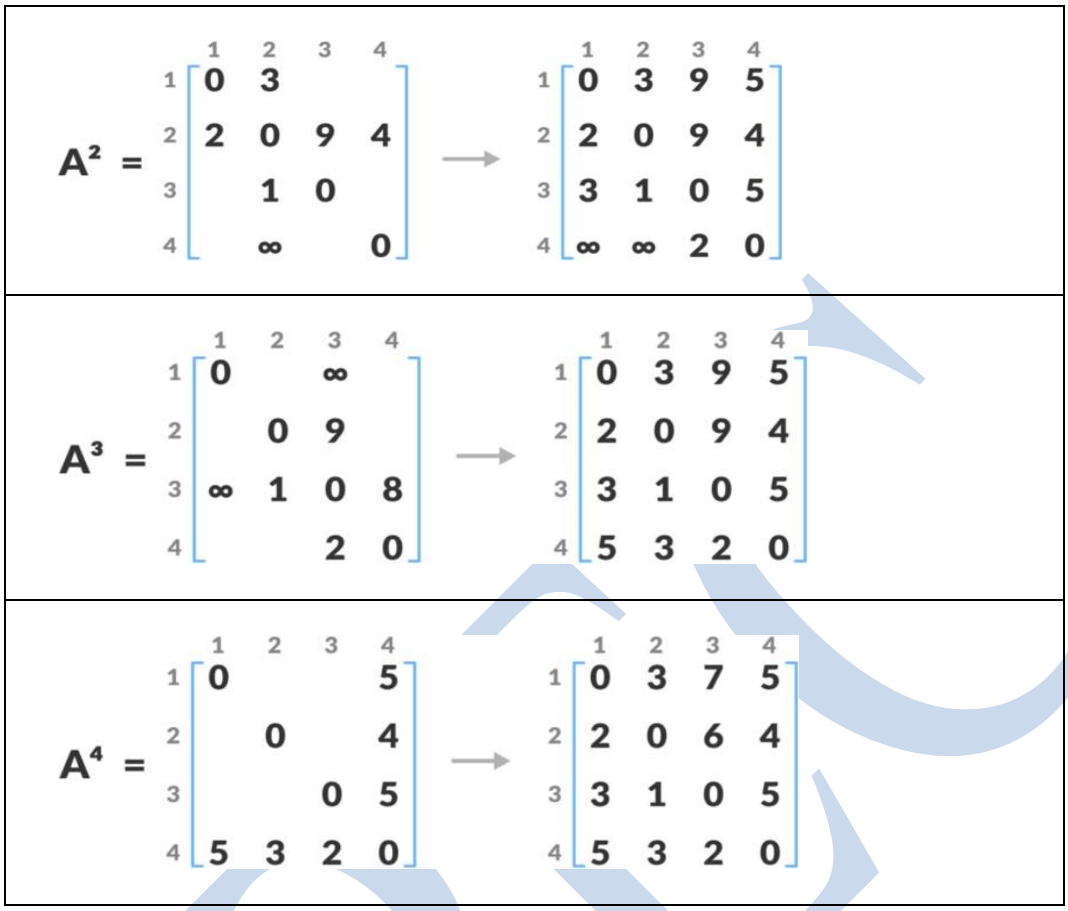
*Given Digraph*




*Initial distance matrix*

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

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




*2.6.2.4. Python program for shortest path algorithm*

```
import matplotlib.pyplot as plt
# importing networkx
import networkx as nx
import pylab
# Create an empty Undirected Weighted Graph
G = nx.Graph()


nodes_list = [1, 2, 3, 4, 5, 6, 7]
G.add_nodes_from(nodes_list)

# Add weighted edges
edges_list = [(1, 2, 13),(1, 4, 4),(2, 3, 2),(2, 4, 6), (2, 5, 4), (3, 5, 5),
              (3, 6, 6),(4, 5, 3),(4, 7, 4), (5, 6, 8), (5, 7, 7), (6, 7, 3)]
G.add_weighted_edges_from(edges_list)


plt.figure()
pos = nx.spring_layout(G)
weight_labels = nx.get_edge_attributes(G,'weight')
nx.draw(G,pos,font_color = 'white', node_shape = 's', with_labels = True,)
nx.draw_networkx_edge_labels(G,pos,edge_labels=weight_labels)
```

```
pos = nx.planar_layout(G)

#Give us the shortest paths from node 1 using the weights from the edges.
p1 = nx.shortest_path(G, source=1, weight="weight")

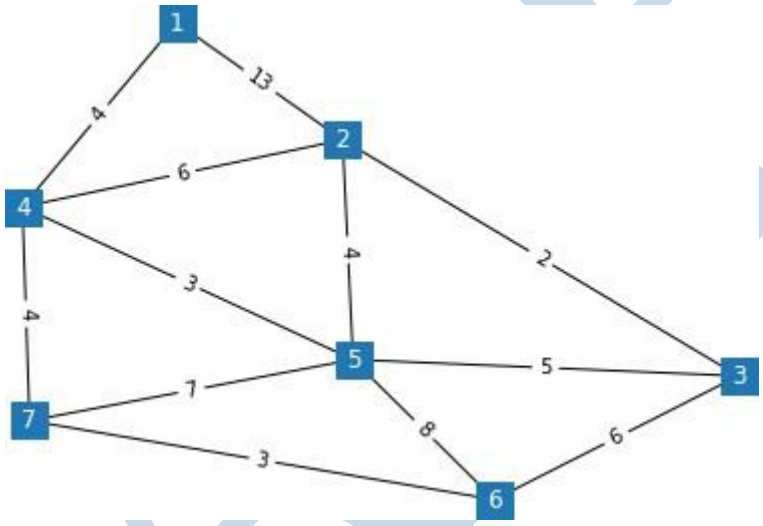
# This will give us the shortest path from node 1 to node 6.
p1to6 = nx.shortest_path(G, source=1, target=6, weight="weight")

# This will give us the length of the shortest path from node 1 to node 6.
length = nx.shortest_path_length(G, source=1, target=6, weight="weight")

print("All shortest paths from 1: ", p1)
print("Shortest path from 1 to 6: ", p1to6)
print("Length of the shortest path: ", length)
```

**Output:**

```
All shortest paths from 1:  {1: [1], 2: [1, 4, 2], 4: [1, 4], 5: [1, 4, 5], 7:
[1, 4, 7], 3: [1, 4, 5, 3], 6: [1, 4, 7, 6]}
Shortest path from 1 to 6:  [1, 4, 7, 6]
Length of the shortest path:  11
```




*2.6.2.5. Comparative analysis of shortest path Algorithms*

| Algorithm | Negative weight | Single source | All source | Time complexity | Space Complexity |
|---|---|---|---|---|---|
| Dijkstra | | √ | | O(\|E\| + V log\|V\|) | O(\|V\|²) |
| Bellman-Ford | √ | √ | | O(\|V\|.\|E\|) | O(\|V\|²) |
| Floyd Warshall | √ | | √ | O(\|V\|³) | O(\|V\|³) |

**Flow networks**

*Definition:*

A flow network is defined as a directed graph where each edge has:

→ Capacity (a non-negative real number)

→ Flow (a non-negative real number that is less than or equal to the capacity).

Additionally, there are two special nodes in the network:

→ Source node where the flow enters the network

→ Sink node where the flow leaves the network.

*Goal:*

→ To find the maximum amount of flow that can be sent from the source to the sink while satisfying the capacity constraints.

In the context of flow networks, augmenting means increasing the flow along a path in the network. An augmenting path is a simple path from the source node to the sink node, along which there is available capacity in all edges. An augmenting path can be used to increase the flow from the source to the sink, by sending additional flow along the path.

*Algorithms used for flow networks*

Several algorithms exist to solve the maximum flow problem in flow networks:

→ Ford-Fulkerson algorithm

→ Edmonds-Karp algorithm

→ Dinic's algorithm.

The Ford-Fulkerson algorithm, which is a general method for finding maximum flow, works by iteratively finding augmenting paths and augmenting the flow along them until no more augmenting paths exist.

*Applications:*

Flow networks have numerous applications:

→ Transportation networks

→ Communication networks

→ Electrical networks

→ Water distribution networks.

They can be used to optimize the flow of resources, minimize congestion, and improve efficiency in various systems.

*Representation of Flow network*

A flow network is a directed graph $G(V, E)$ where each edge $e \in E$ is associated with a $capacity\ c(e)$ and a $flow\ f(e)$, where $f(e)$ is the amount of flow that can be sent through the edge e. The flow can be thought of as a quantity of some resource (e.g., water, electricity, or data) that can be transported through the network.

In a flow network, there are two distinguished vertices: the source s and the sink t. The source is where the flow enters the network, and the sink is where the flow leaves the network. There are no incoming edges to the source and no outgoing edges from the sink.

To represent a flow network, we can use a graph G(V, E) with a capacity function c:E→R+. Additionally, we can add a flow function f:E→R to keep track of the flow through each edge. The value of the flow function f(e) represents the amount of flow currently passing through the edge e.

In summary, a flow network can be represented by a directed graph G(V, E) with a capacity function c:E→R+ and a flow function f:E→R. The source vertex s and the sink vertex t are identified, and each edge e in E is associated with a capacity c(e) and a flow f(e).

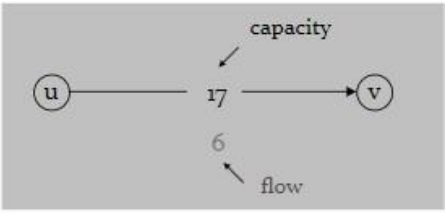
**Fulkerson algorithm**

The Ford-Fulkerson algorithm is an algorithm used to find the maximum flow in a flow network. It is named after L.R. Ford Jr. and D.R. Fulkerson who published the algorithm in 1956.

## Residual Graph

Original edge: $e = (u, v) \in E$.

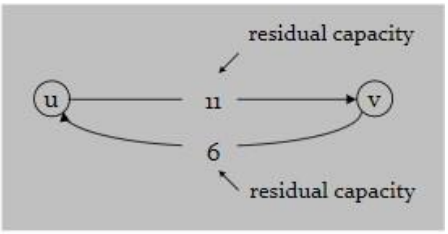
- Flow f(e), capacity c(e).




Residual edge.

- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$




Residual graph: $G_f = (V, E_f)$.

- Residual edges with positive residual capacity.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : c(e) > 0\}$.

# Augmenting Path Algorithm

```
Augment(f, c, P) {
   b ← bottleneck(P)
   foreach e ∈ P {
      if (e ∈ E) f(e) ← f(e) + b
      else       f(e^R) ← f(e) - b
   }
   return f
}
```

forward edge

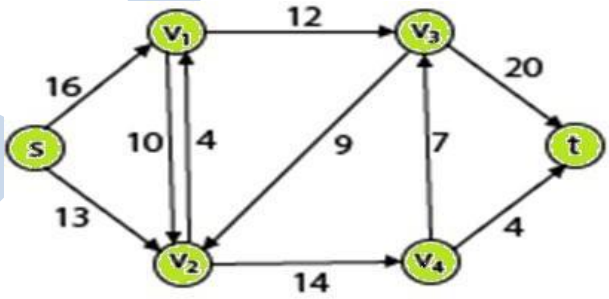
reverse edge

```
Ford-Fulkerson(G, s, t, c) {
   foreach e ∈ E  f(e) ← 0
   G_f ← residual graph

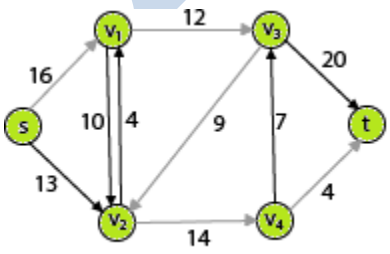
   while (there exists augmenting path P) {
      f ← Augment(f, c, P)
      update G_f
   }
   return f
}
```

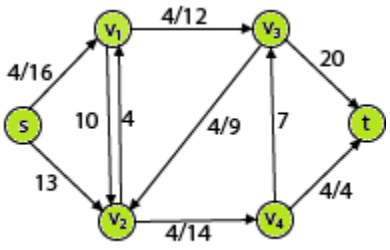
*Problems on Ford-Fulkerson algorithm*

*Problem 1:*

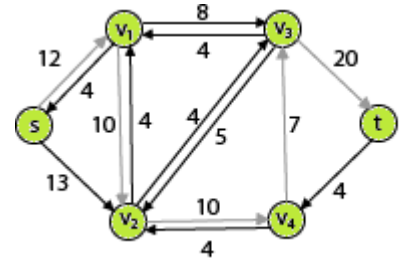


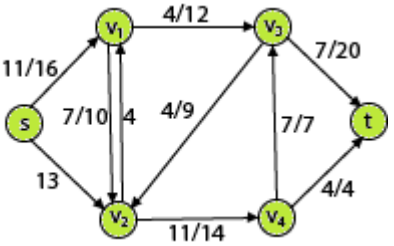

*Solution*:



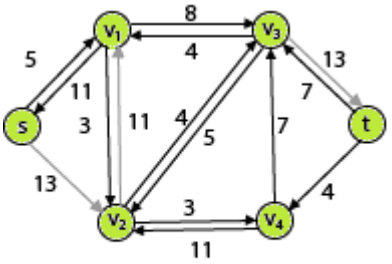



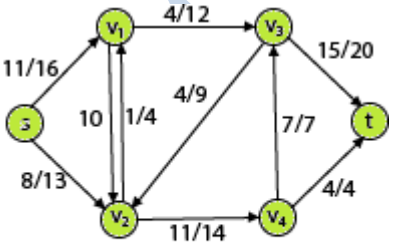

(a)



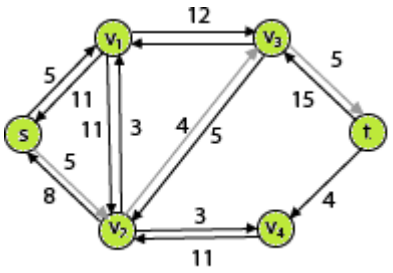

(b)

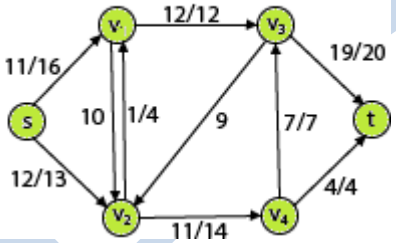


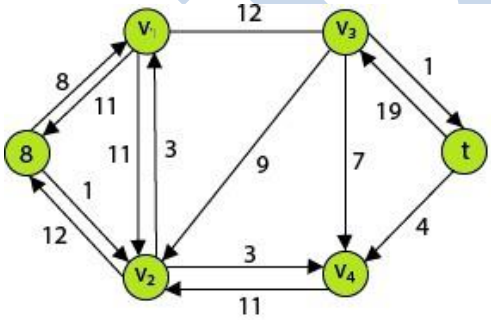

(c)

(In this , 8 is break into 7 and 1 and 7 is canclled by $v_1v_2$flow)



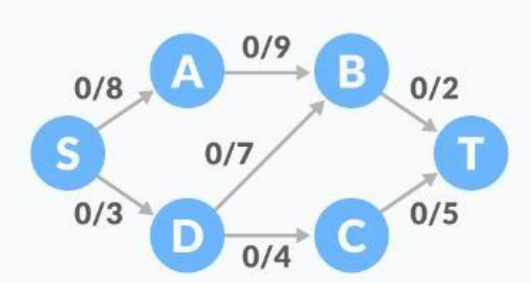

(d)



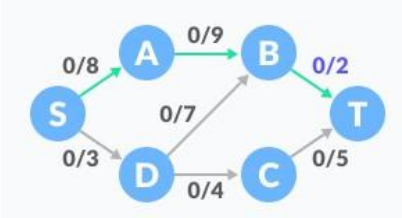
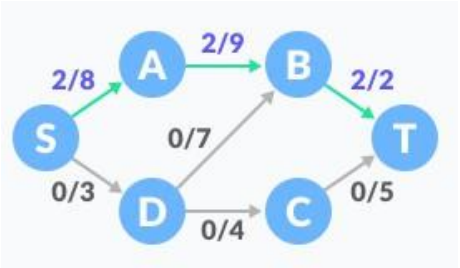
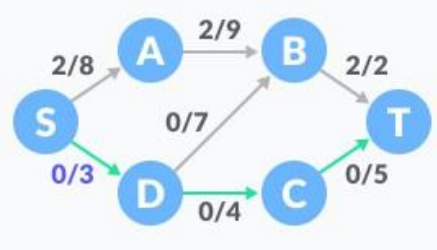
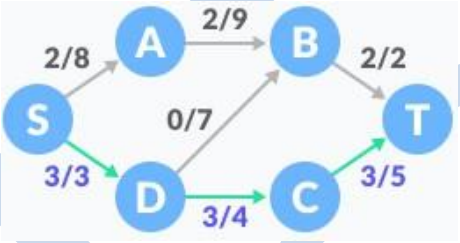
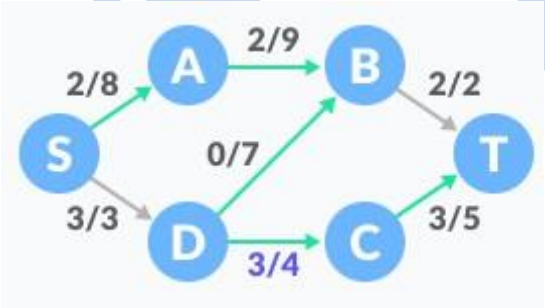
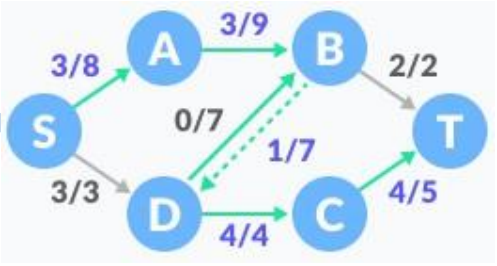

(e)

Now , it has no augmenting paths .So, the maximum flow shown in (d) is 23 is a maximum flow .

*Problem 2:*



*Solution*

<table>
<tr>
<td>Step 1<br>Select any arbitrary path from S to T. In this step, we have selected path S-A-B-T<br>
</td>
<td>Step 2<br>The minimum capacity among the three edges is 2 (B-T). Based on this, update the flow/capacity for each path.<br>
</td>
</tr>
<tr>
<td>Step 3<br>Select another path S-D-C-T. The minimum capacity among these edges is 3 (S-D)<br>
</td>
<td>Step 4<br>Update the capacities according to this.<br>
</td>
</tr>
<tr>
<td>Step 5<br>Now, let us consider the reverse-path B-D as well. Selecting path S-A-B-D-C-T. The minimum residual capacity among the edges is 1 (D-C).<br>
</td>
<td>Step 6<br>Updating the capacities.<br>
</td>
</tr>
<tr>
<td colspan="2">The capacity for forward and reverse paths are considered separately.<br>Adding all the flows = 2 + 3 + 1 = 6, which is the maximum possible flow on the flow network.</td>
</tr>
</table>

```
# Python program for Ford-Fulkerson algorithm
# Create a directed graph representing the flow network
G = nx.DiGraph()

# Add edges and capacities to the graph
G.add_edge('A', 'B', capacity=3)
G.add_edge('A', 'C', capacity=2)
G.add_edge('B', 'C', capacity=1)
G.add_edge('B', 'D', capacity=3)
G.add_edge('C', 'D', capacity=1)

# Find the maximum flow from 'A' to 'D' using the Ford-Fulkerson algorithm
source = 'A'
sink = 'D'
max_flow_value, flow_dict = nx.maximum_flow(G, source, sink)

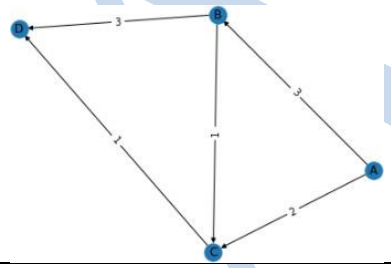
# Print the maximum flow value
print(f"Maximum flow from {source} to {sink} is {max_flow_value}")

# Draw the graph with capacities
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True)
labels = nx.get_edge_attributes(G, 'capacity')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

# Show the graph
plt.show()
```

***Execution:***

```
Maximum flow from A to D is 4
```




```
import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()
# Add nodes to the graph
G.add_nodes_from([1, 2, 3, 4, 5])
# Add edges to the graph with capacities
G.add_edge(1, 2, capacity=3)
G.add_edge(1, 3, capacity=2)
```

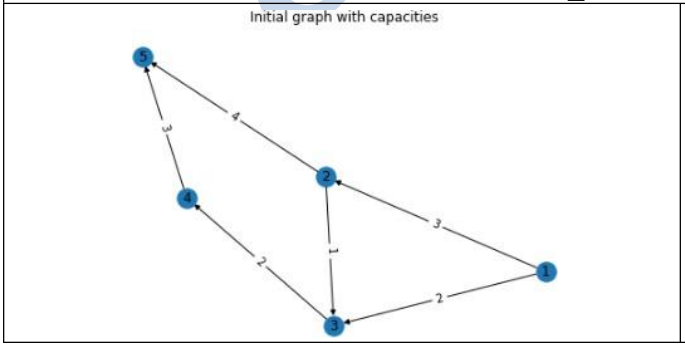
```
G.add_edge(2, 3, capacity=1)
G.add_edge(2, 5, capacity=4)
G.add_edge(3, 4, capacity=2)
G.add_edge(4, 5, capacity=3)

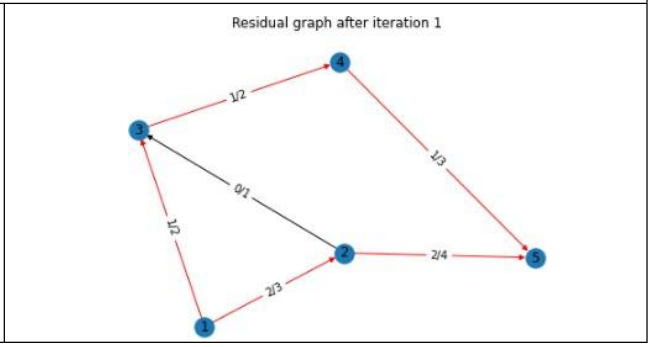
# Draw the initial graph with capacities
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True)
labels = nx.get_edge_attributes(G, 'capacity')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.title("Initial graph with capacities")
plt.show()

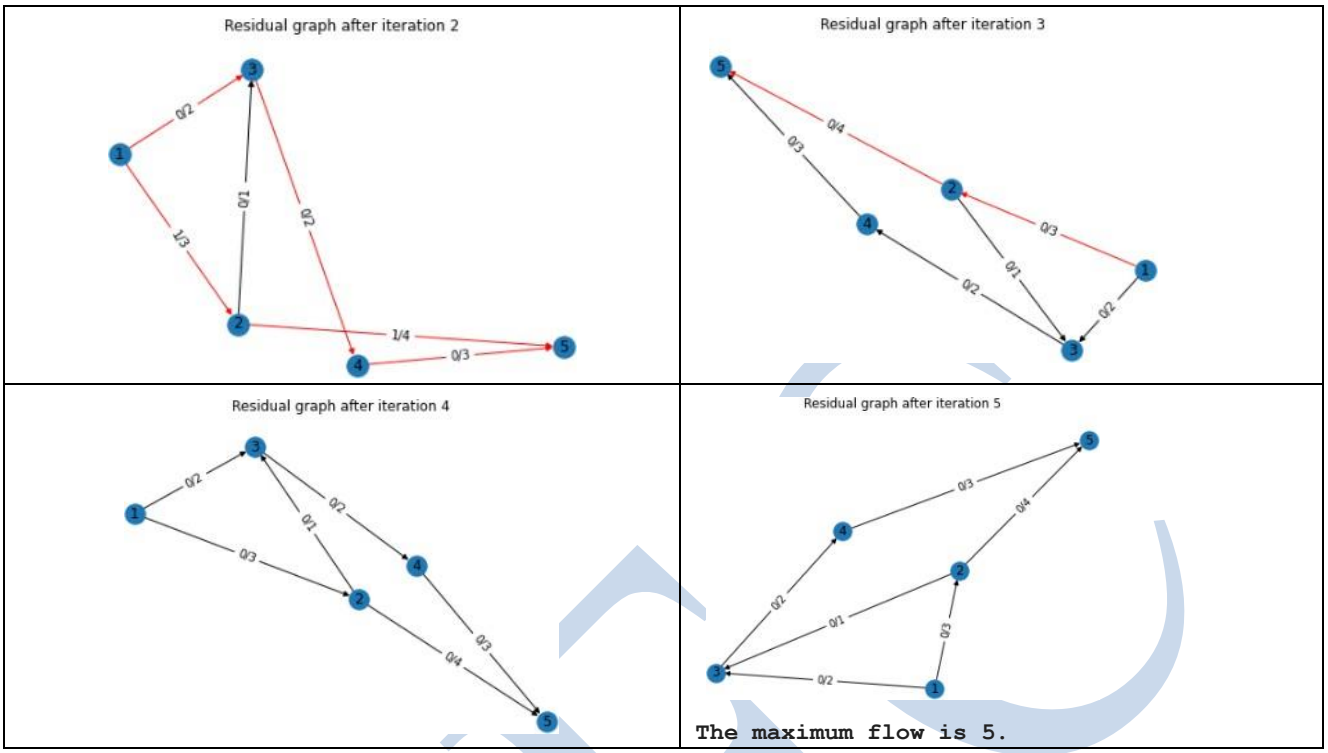
# Compute the maximum flow using the Ford-Fulkerson algorithm
flow_value, flow_dict = nx.maximum_flow(G, 1, 5)

# Print the intermediate steps of the algorithm and draw the corresponding g
raphs
for i in range(1, flow_value+1):
    H = G.copy()
    for u, nbrs in flow_dict.items():
        for v, flow in nbrs.items():
            if flow >= i:
                H[u][v]['color'] = 'r'
                H[u][v]['flow'] = flow - i
            else:
                H[u][v]['color'] = 'k'
                H[u][v]['flow'] = 0
    pos = nx.spring_layout(H)
    edge_colors = [H[u][v]['color'] for u, v in H.edges()]
    edge_labels = {(u, v): f"{H[u][v]['flow']}/{G[u][v]['capacity']}" for u,
 v in H.edges()}
    nx.draw(H, pos, with_labels=True, edge_color=edge_colors)
    nx.draw_networkx_edge_labels(H, pos, edge_labels=edge_labels)
    plt.title(f"Residual graph after iteration {i}")
    plt.show()

# Print the maximum flow value
print(f"The maximum flow is {flow_value}.")
```



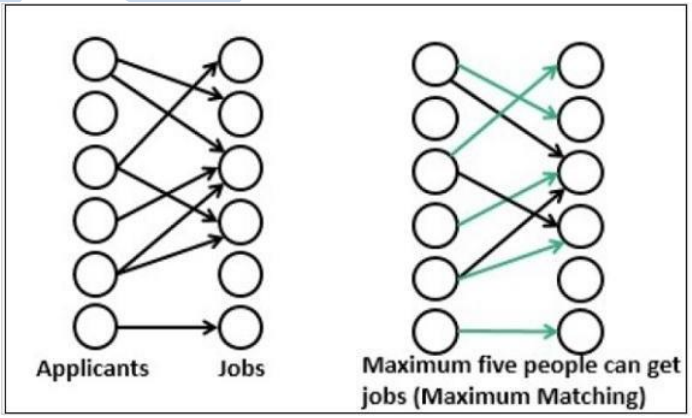



**Maximum Bipartite Matching**

The maximum bipartite matching problem is a problem in graph theory and combinatorial optimization. Given a bipartite graph, which is a graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set, the problem is to find the largest possible set of edges that do not share any vertices, also called a matching.

In other words, the goal is to find a subset of edges such that no two edges share an endpoint, and no other subset of edges has more edges than this subset. Each edge in the matching corresponds to a pair of vertices, one from each set, that are "matched" with each other.

*Example:*




There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be

appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs."

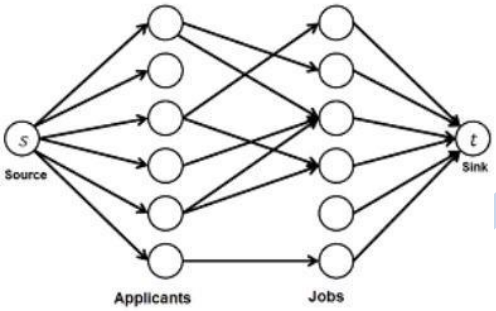
*Applications*

- Assigning tasks to workers
- Matching organ donors with recipients
- Matching job seekers with potential employers.

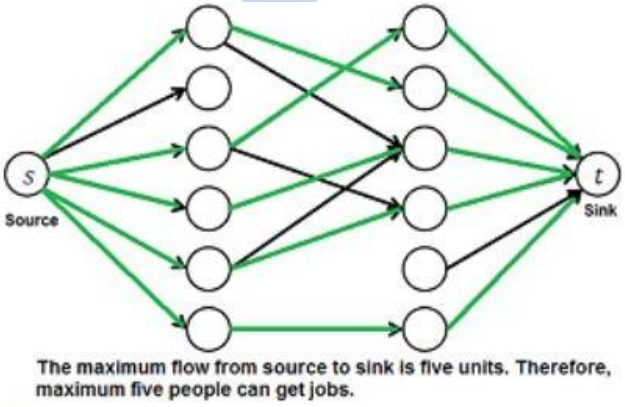
Finding the maximum bipartite matching can be done using various algorithms, including the Hopcroft-Karp algorithm, Edmonds' algorithm, or the Ford-Fulkerson algorithm.

Maximum Bipartite Matching (MBP) problem can be solved by converting it into a flow network

- Build a Flow Network: There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.




- Find the maximum flow: We use Ford-Fulkerson algorithm to find the maximum flow in the flow network built in above step. The maximum flow is actually the MBP we are looking for.




Algorithm

**bipartiteMatch(u, visited, assign)**

**Input:** Starting node, visited list to keep track, assign the list to assign node with another node.

**Output** – Returns true when a matching for vertex u is possible.

```
Begin
  for all vertex v, which are adjacent with u, do if
    v is not visited, then
      mark v as visited
      if v is not assigned, or bipartiteMatch(assign[v], visited, assign) is true, then assign[v]
        := u
        return true
  done
  return false
End
```

**maxMatch(graph)**

**Input** – The given graph.

**Output** – The maximum number of the match.

```
Begin
  initially no vertex is assigned
  count := 0
  for all applicant u in M, do
    make all node as unvisited
    if bipartiteMatch(u, visited, assign), then
      increase count by 1
  done
End
```

APEC