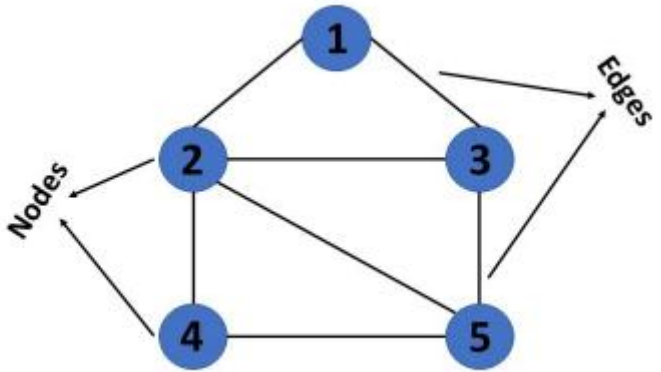
Algorithms and Data Structures Using Java

Soumya

Graph

• A graph is a non-linear kind of data structure made up of nodes or vertices and edges.

• The edges connect any two nodes in the graph, and the nodes are also known as vertices.



Graph Theory

• Graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects.

• A graph in this context is made up of vertices, which are also called nodes or points, which are connected by edges.

• Edges can be directed or undirected, and they can be weighted or unweighted.

Graph Theory

• Graph theory is a very powerful tool that can be used to model a wide variety of real-world problems,including:

– Social networks: Graph theory can be used to model social networks, such as the relationships between people on Facebook or Twitter.

– Transportation networks: Graph theory can be used to model transportation networks, such as the roads and highways in a city.

Graph Theory

- **Electrical circuits:**

  – Graph theory can be used to model electrical circuits, where the vertices represent components and the edges represent wires.

- **Computer networks:**

  – Graph theory can be used to model computer networks, where the vertices represent computers and the edges represent network connections.

Graph Theory

Here are some of the basic concepts in graph theory:

– Vertex: A vertex is a point in a graph. It is also called a node or a point.

– Edge: An edge is a line that connects two vertices in a graph. Edges can be directed or undirected, and they can be weighted or unweighted.

– Directed edge: A directed edge connects two vertices in a specific direction.

Graph Theory

- Undirected edge:
  - An undirected edge connects two vertices in either direction.
- Weighted edge:
  - A weighted edge has a weight associated with it. The weight can represent the distance between the two vertices, the cost of traveling between the two vertices, or some other measure.
- Unweighted edge:
  - An unweighted edge does not have a weight associated with it.

Graph Theory

- Path
  - A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

- Closed Path
  - A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.

Graph Theory

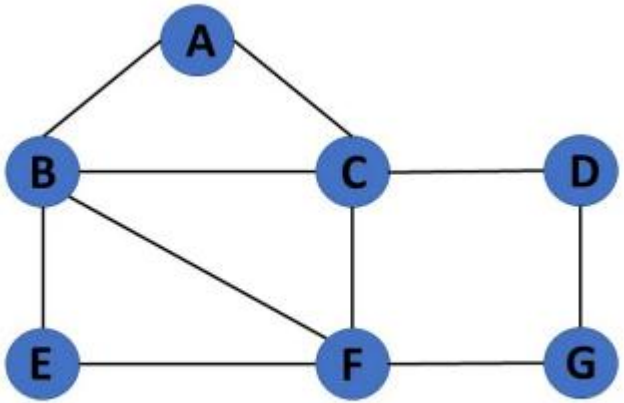
- Simple Path
  - If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.
- Cycle
  - A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Graph Theory

- Loop
  - An edge that is associated with the similar end points can be called as Loop.
- Adjacent Nodes
  - If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- Degree of the Node
  - A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

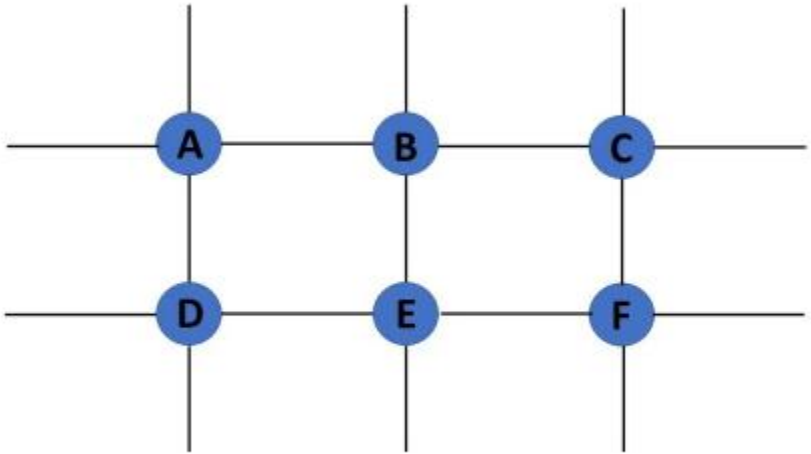
Type: Finite Graph

- The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number.



Type: Infinite Graph

- The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.

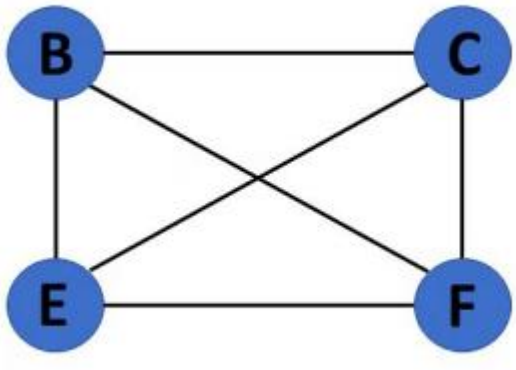


Type: Trivial Graph

- A graph G= (V, E) is trivial if it contains only a single vertex and no edges.

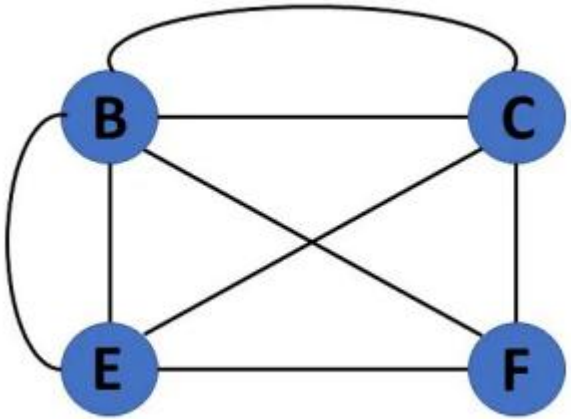
Type: Simple Graph

- If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



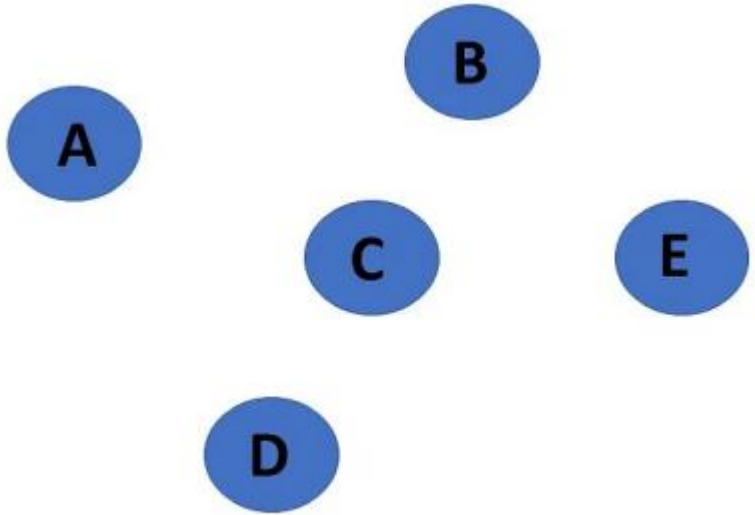
Type: Multi Graph

- If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



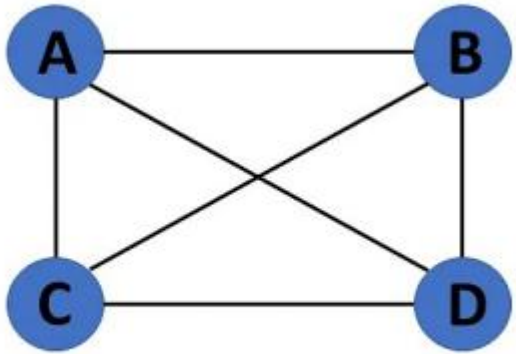
Type: Null Graph

- It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



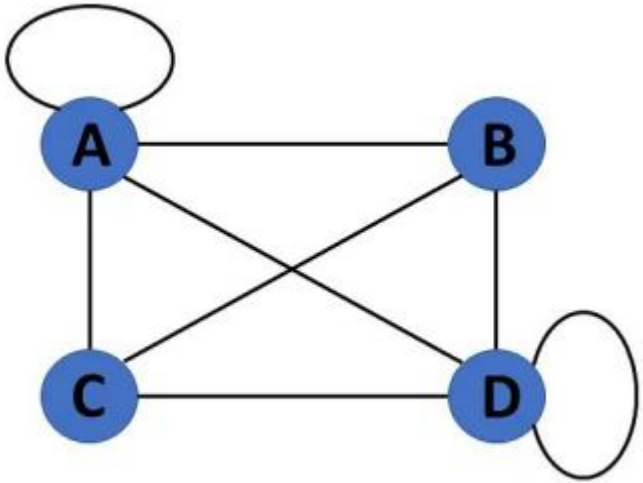
Type: Complete Graph

- If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



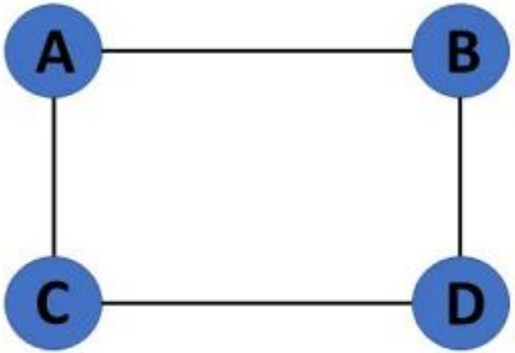
Type: Pseudo Graph

- If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.



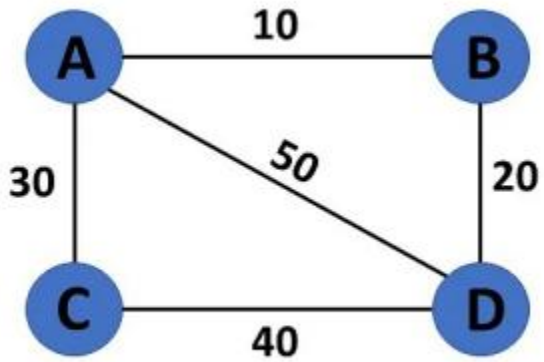
Type: Regular Graph

- If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.



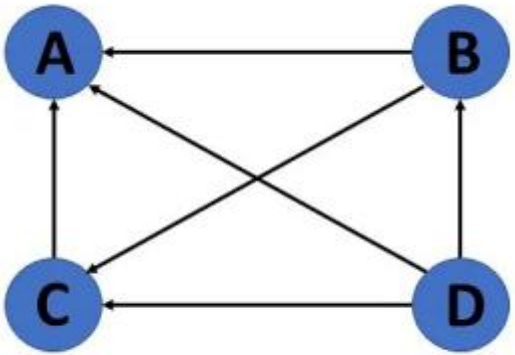
Type: Weighted Graph

- A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



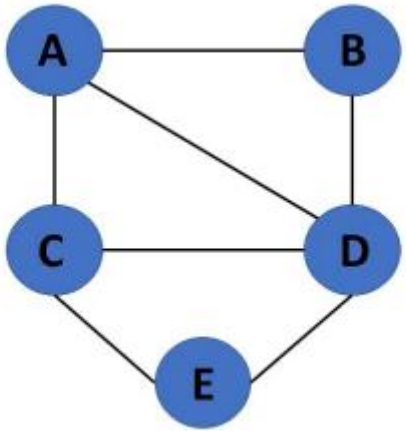
Type: Directed Graph

- A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.



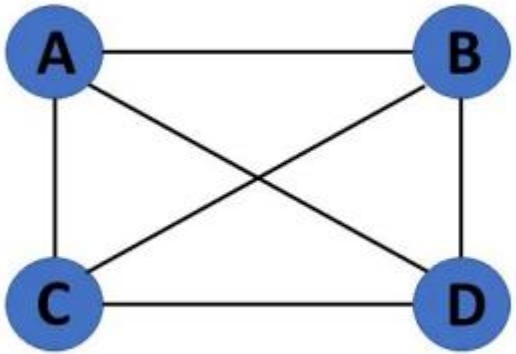
Type: Undirected Graph

- An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



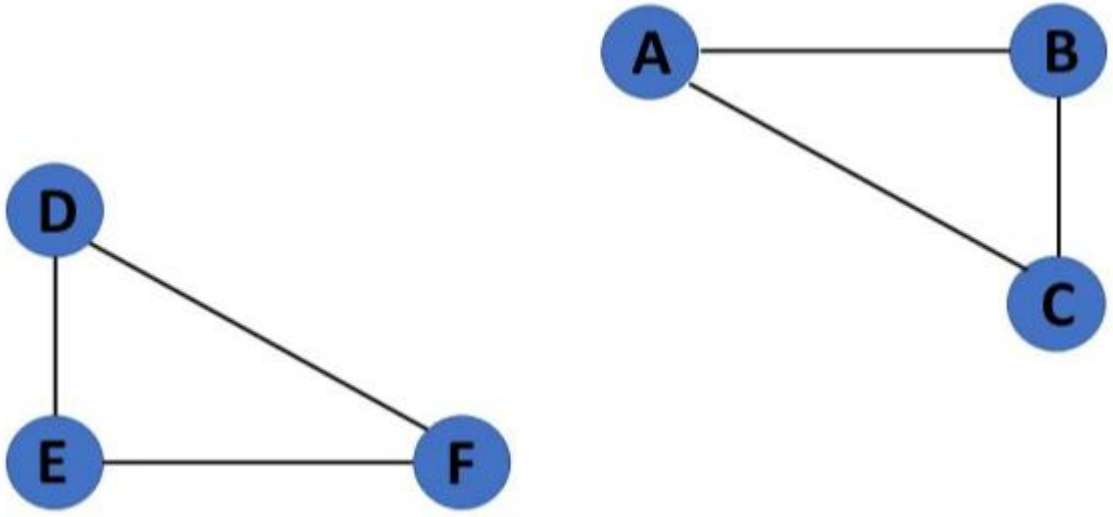
Type: Connected Graph

- If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



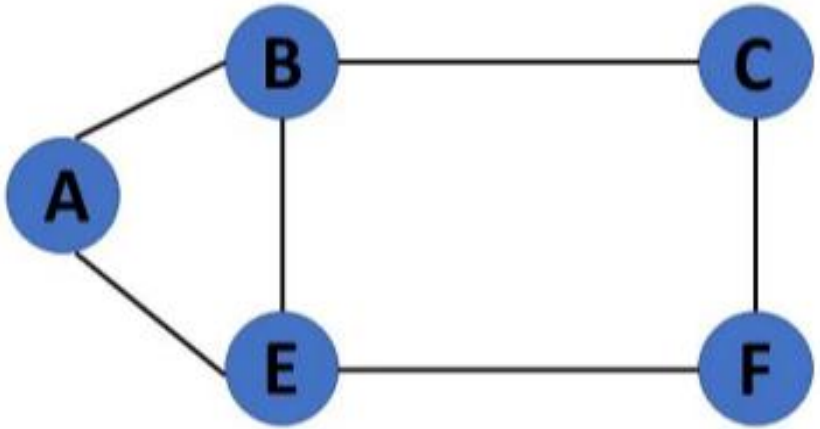
Type: Disconnected Graph

- When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



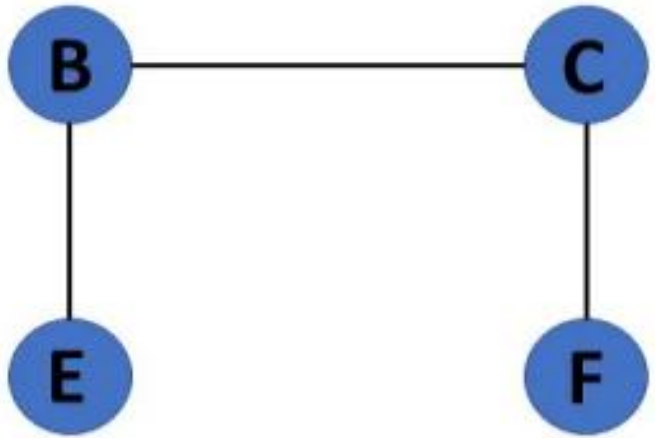
Type: Cyclic Graph

• If a graph contains at least one graph cycle, it is considered to be cyclic.



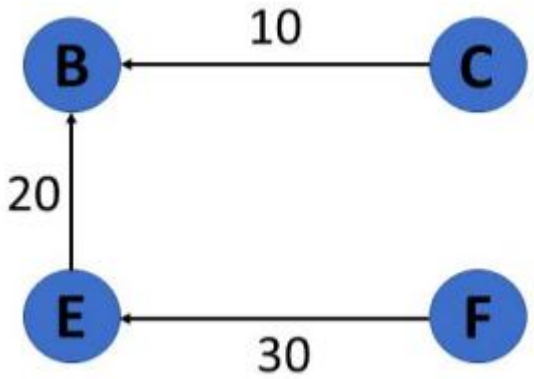
Type: Acyclic Graph

- When there are no cycles in a graph, it is called an acyclic graph.



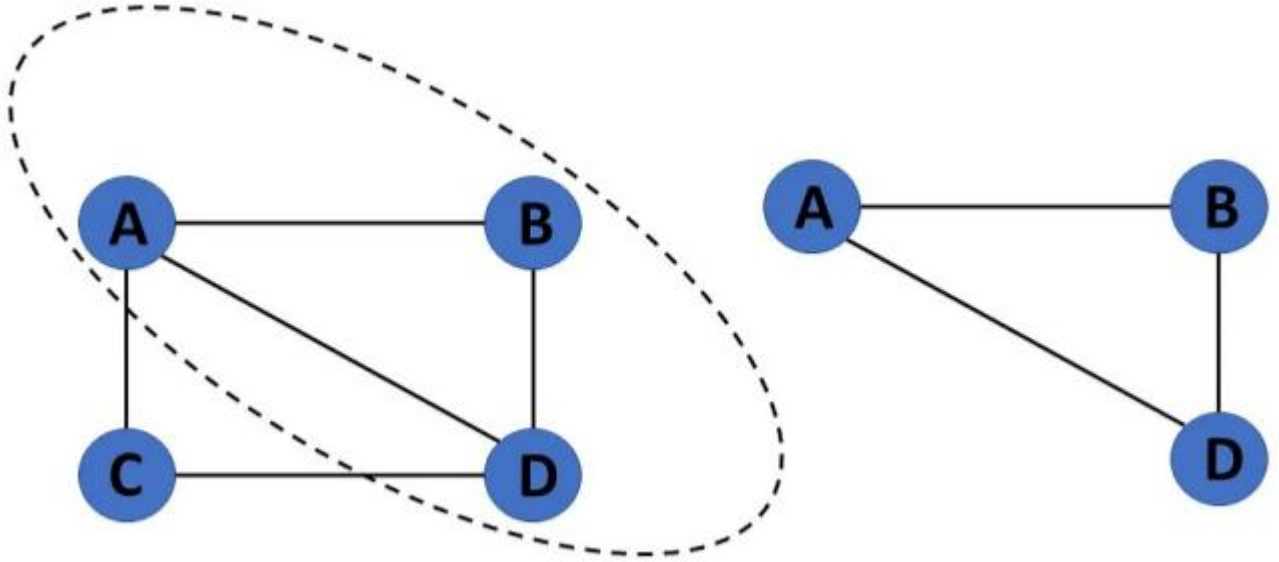
Type: Acyclic Graph

- It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle.
- It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



Type: Subgraph

- The vertices and edges of a graph that are subsets of another graph are known as a subgraph.

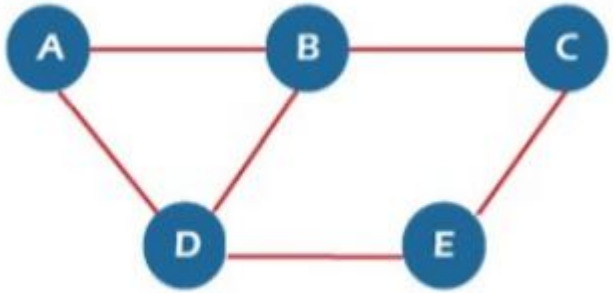


Graph Representation

- By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.
- A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:
  - Sequential representation (or, Adjacency matrix representation)
  - Linked list representation (or, Adjacency list representation)

Graph Representation

- In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph.
- We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.
- If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.



Undirected Graph

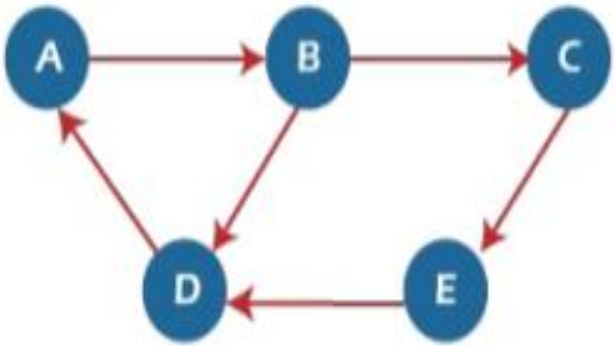
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Graph Representation

- Adjacency matrix for a directed graph
- In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.
- Consider the below-directed graph and try to construct the adjacency matrix of it.

Graph Representation



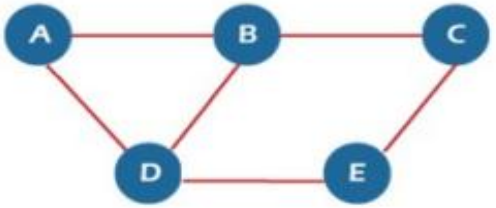
Directed Graph

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

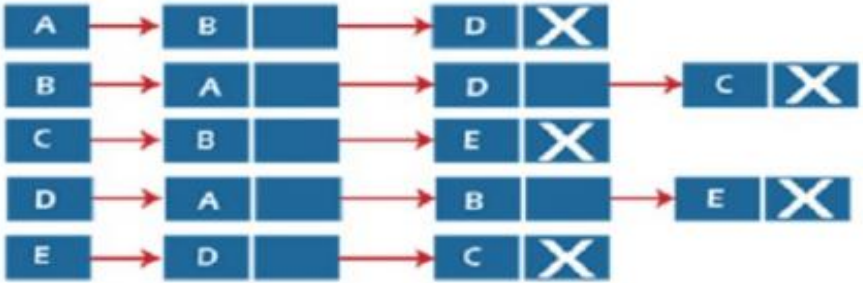
Adjacency Matrix

Graph Representation

- An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.



Undirected Graph



Adjacency List

Graph Traversal Algorithms

- The process of visiting or updating each vertex in a graph is known as graph traversal.
- The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a Subset of tree traversal.
- There are two techniques to implement a graph traversal algorithm:
  - Breadth-first search
  - Depth-first search

Breadth-First Search or BFS

- BFS is a search technique for finding a node in a graph data structure that meets a set of criteria.
  - It begins at the root of the graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.
  - To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally you require a queue.

Breadth-First Search or BFS

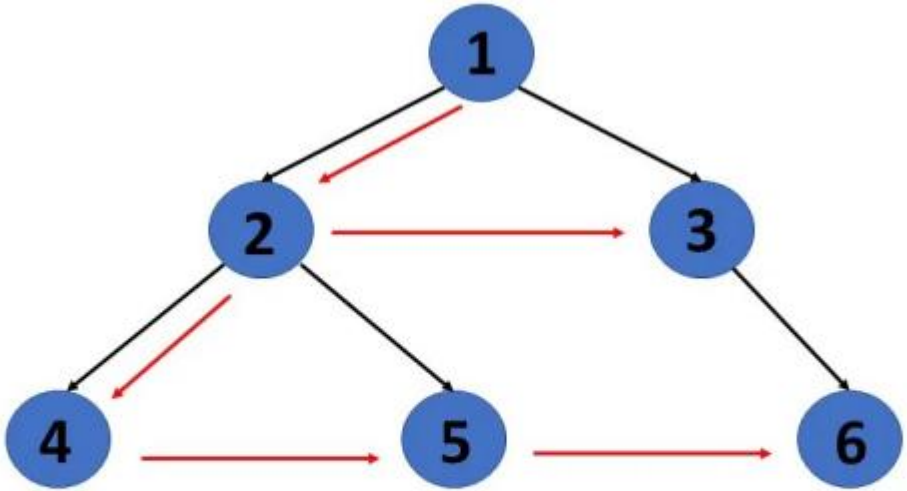
- Step 1: Consider the graph you want to navigate.
- Step 2: Select any vertex in your graph, say v1, from which you want to traverse the graph.
- Step 3: Examine any two data structures for traversing the graph.
  - Visited array (size of the graph)
  - Queue data structure
- Step 4: Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

Breadth-First Search or BFS

- Step 5: Now, using the FIFO concept, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.
- Step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.

BFS():

- Create isVisited array of size vertexCount.
- Set all elements in isVisited to FALSE.
- Add startVertex to the queue. // We use 0 as startVertex
- while (queue is not empty) do
    - Remove vertex, vi, from queue.
    - if (vi is not visited) then
        - Mark vi as visited and process it.
        - For every adjacent vertex, vj, to vi that is not visited
    - Add vj to queue
- Stop



BFS	1	2	3	4	5	6

Depth First Search

• DFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

– The depth-first search (DFS) algorithm traverses or explores data structures such as trees and graphs.

The DFS algorithm begins at the root node and examines each branch as far as feasible before backtracking.

– To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally a stack, is required.

Depth First Search-iterative approach

- Step 1: Consider the graph you want to navigate.
- Step 2: Select any vertex in our graph, say v1, from which you want to begin traversing the graph.
- Step 3: Examine any two data structures for traversing the graph.
  - Visited array (size of the graph)
  - Stack data structure

Depth First Search

- Step 4: Insert v1 into the array's first block and push all the adjacent nodes or vertices of vertex v1 into the stack.
- Step 5: Now, pop the topmost element and put it into the visited array, pushing all of the popped element's nearby nodes into it.
- Step 6: If the topmost element of the stack is already present in the array, discard it instead of inserting it into the visited array.
- Step 7: Repeat step 6 until the stack data structure isn't empty.

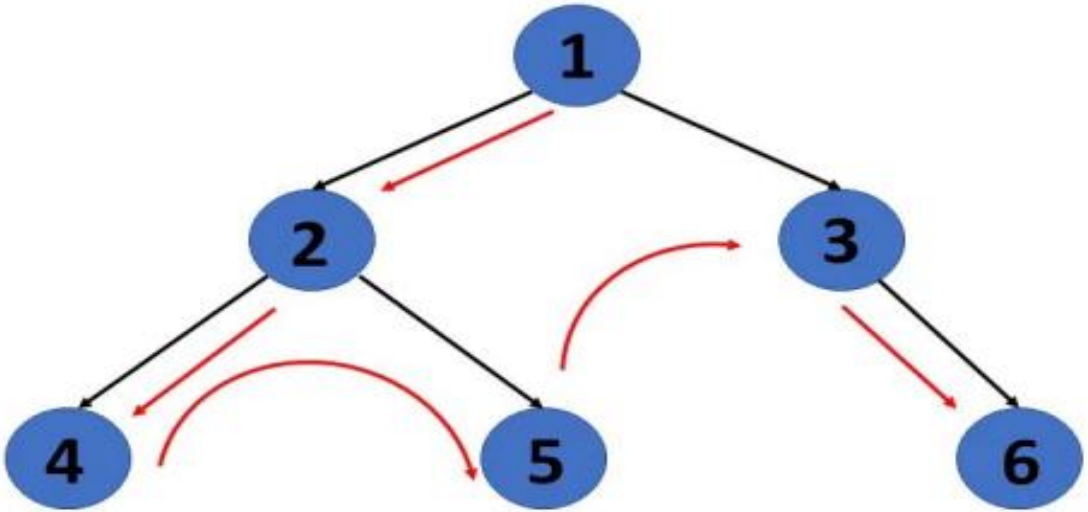
Depth First Search : Recursive approach

- Create isVisited array of size vertexCount.
- Set all elements in isVisited to FALSE.
- DFSHelper(1, isVisited)
- Stop

DFSHelper(startVertex, isVisited)

- if (startVertex is visited) then
- Stop
- Mark startVertex as visited
- Process startVertex
- For every adjacent vertex, vj, to startVertex that is not visited
- DFSHelper(vj, isVisited)
- Stop

Depth First Search



DFS	1	2	4	5	3	6

Shortest Path

- Shortest path algorithms are a family of algorithms designed to solve the shortest path problem. The shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.
- Two main types of shortest path algorithms:
  - Single-source shortest path algorithms: These algorithms find the shortest path from a single source vertex to all other vertices in the graph.
  - All-pairs shortest path algorithms: These algorithms find the shortest path between all pairs of vertices in the graph.

Shortest Path

- **Dijkstra's algorithm**: Dijkstra's algorithm is a single-source shortest path algorithm that works by greedily adding edges to the shortest path tree, starting at the source vertex.
- **Bellman-Ford algorithm**: Bellman-Ford algorithm is another single-source shortest path algorithm that can also handle negative edge weights.
- **Floyd-Warshall algorithm**: Floyd-Warshall algorithm is an all-pairs shortest path algorithm that works by iteratively relaxing edges in the graph.

Shortest Path

- Shortest path algorithms have many applications in real-world problems, such as:

– Routing: Shortest path algorithms can be used to find the shortest route between two destinations on a roadmap.

– Network optimization: Shortest path algorithms can be used to optimize the flow of data through a network.

– Supply chain management: Shortest path algorithms can be used to find the shortest path to deliver goods from a warehouse to a customer.

Dijkstra's algorithm

• Initialize a set of visited vertices to be empty.

• Initialize the distance to each vertex from the source vertex to be infinity.

• Mark the source vertex as visited and set its distance to 0.

• While there are still unvisited vertices:

– Find the unvisited vertex with the shortest distance from the source vertex.

– Mark this vertex as visited.

– For each neighbor of this vertex:

• If the neighbor is unvisited, update its distance to the source vertex if the current path is shorter than the current distance.

• Once all vertices have been visited, the shortest path tree has been constructed.

Dijkstra's algorithm

• Initialize the set of visited vertices to be empty.

• Initialize the distance to each vertex from the source vertex to be infinity.

– Mark the source vertex as visited and set its distance to 0.

distance[0] = 0

visited = {0}

• While there are still unvisited vertices:

– Find the unvisited vertex with the shortest distance from the source vertex.

• current_vertex = min(set(range(len(distance))) - visited, key=lambda vertex:

distance[vertex]

• * Mark this vertex as visited.visited.add(current_vertex)

• * For each neighbor of this vertex: for neighbor in range(len(distance)):

– * If the neighbor is unvisited, update its distance to the source vertex if the current path is shorter than the current distance.

– if distance[current_vertex] + adjacency_matrix[current_vertex][neighbor] distance[neighbor]:

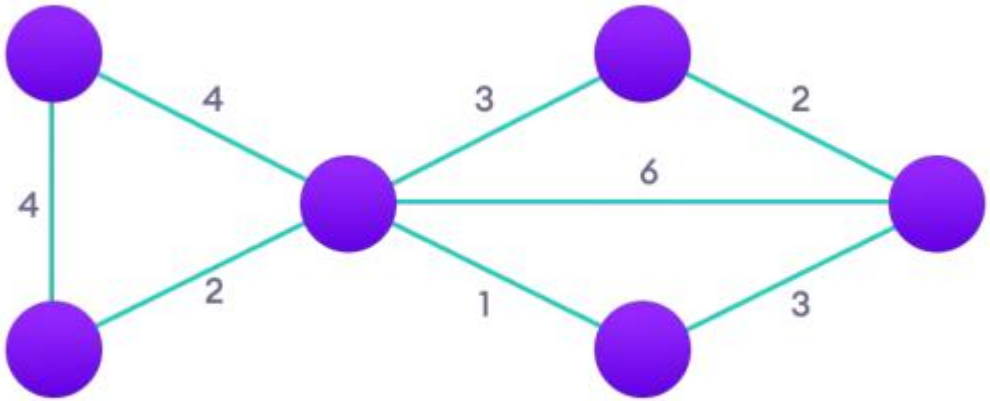
– distance[neighbor] = distance[current_vertex]+adjacency_matrix[current_vertex][neighbor]

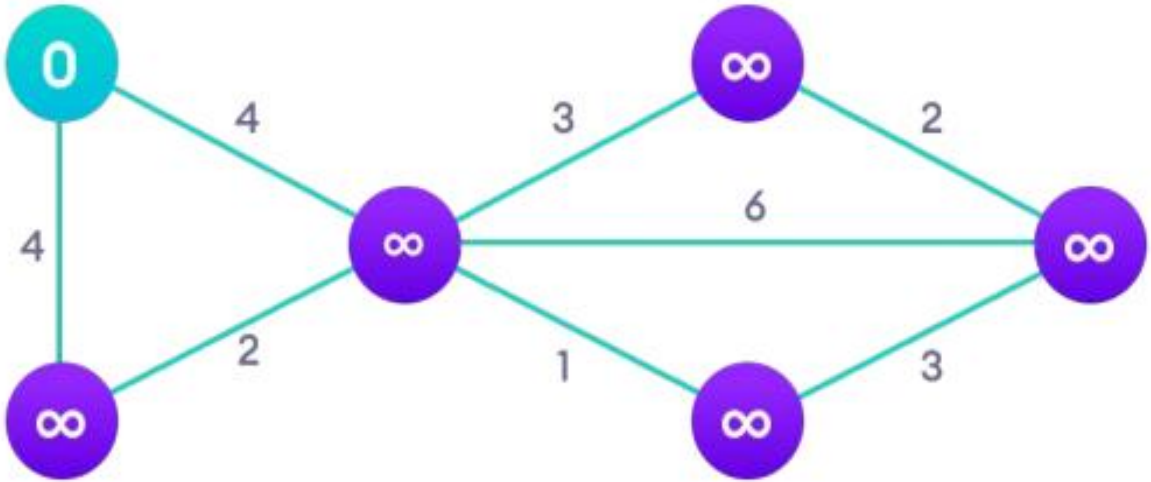
Dijkstra's algorithm:Overview

Dijkstra’s Shortest Path (startVertex, endVertex)
- Set currentDistance for all vertices as infinity.
- Set currentDistance for startVertex as 0.
- Create a list of all vertices in graph, vertexList.
- while vertexList is not empty do
- Get a vertex, u, from vertexList such that u has smallest distance
// Update the distance of each vertex v, adjacent to u.
- distanceToVviaU = currentDistance[u] + weight of edge (u, v)
- if (currentDistance[v] > distanceToVvisU) then
- Set currentDistance[v] to distanceToVviaU
- Set predecessor of v to u.
- Stop

Dijkstra's algorithm

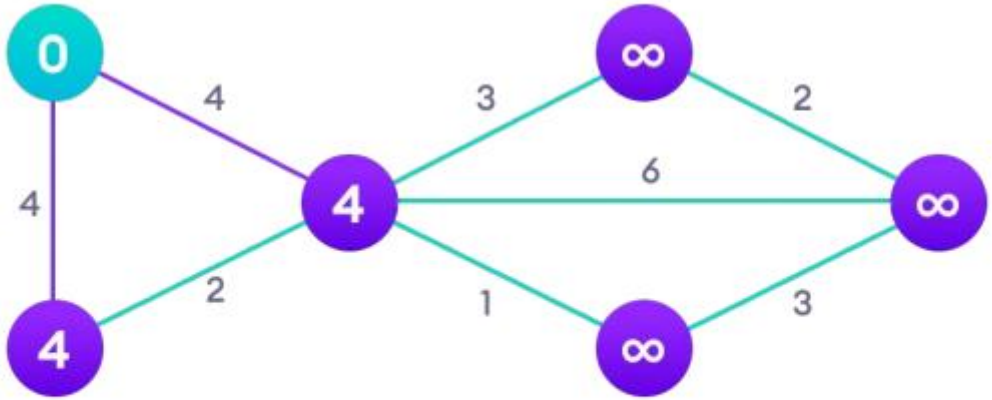
• Once all vertices have been visited, the shortest path tree has been constructed.





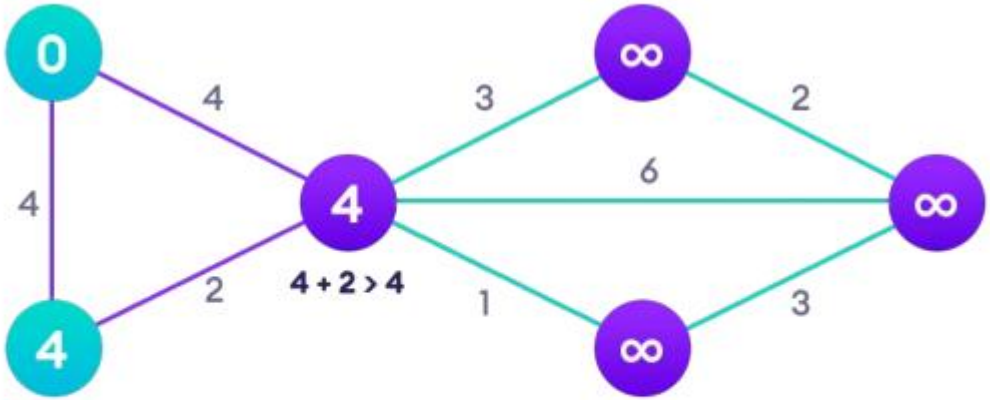
Step: 2

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



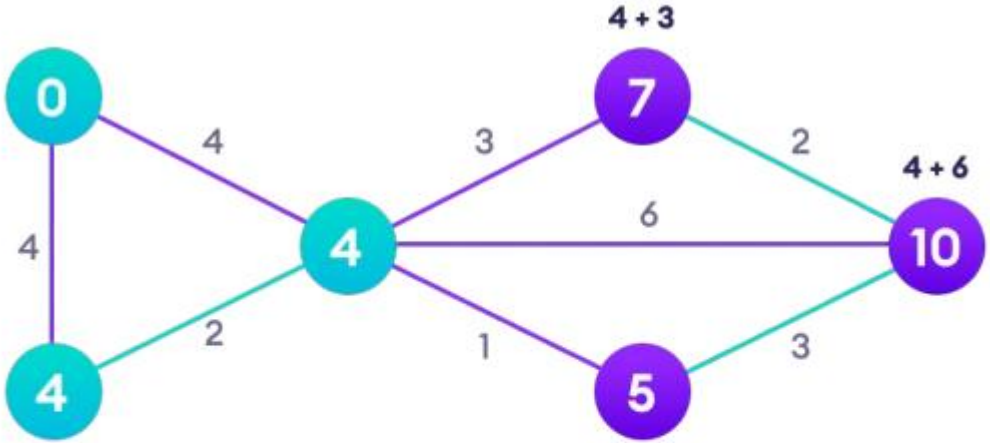
Step: 3

Go to each vertex and update its path length



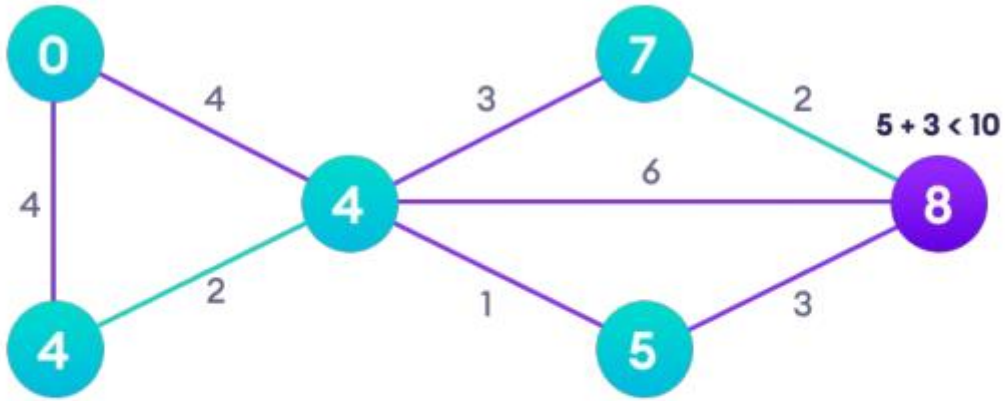
Step: 4

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



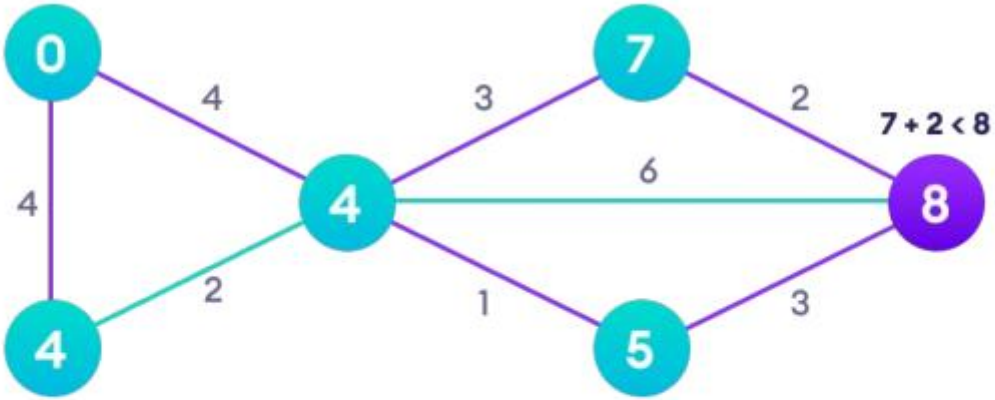
Step: 5

Avoid updating path lengths of already visited vertices.



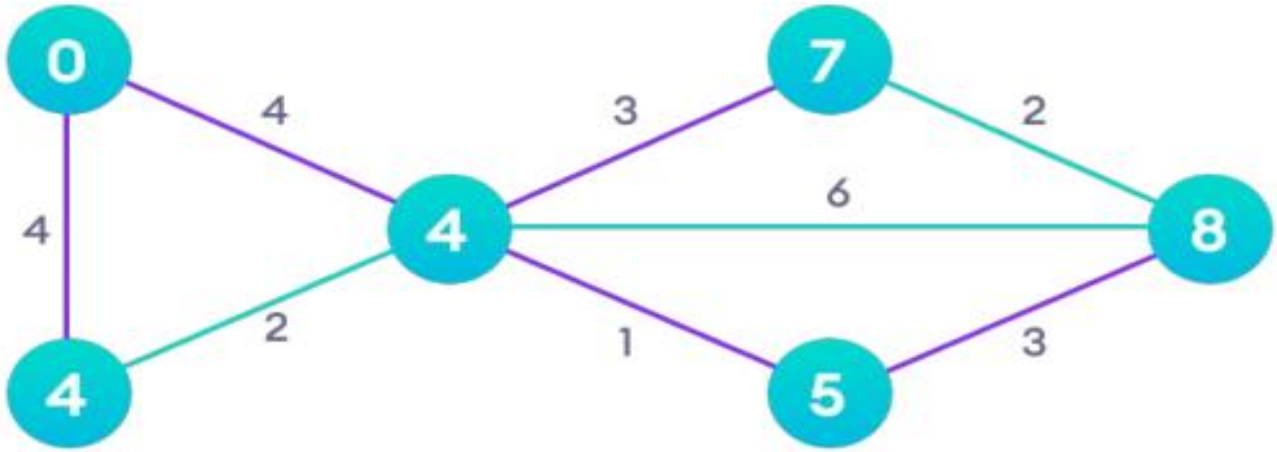
Step: 6

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



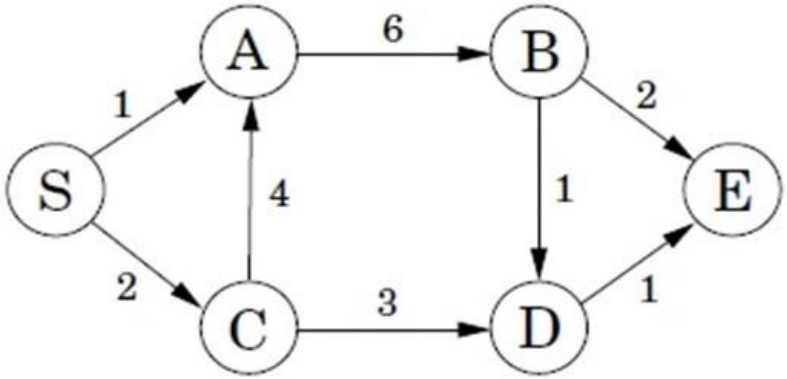
Step: 7

Notice how the rightmost vertex has its path length updated twice

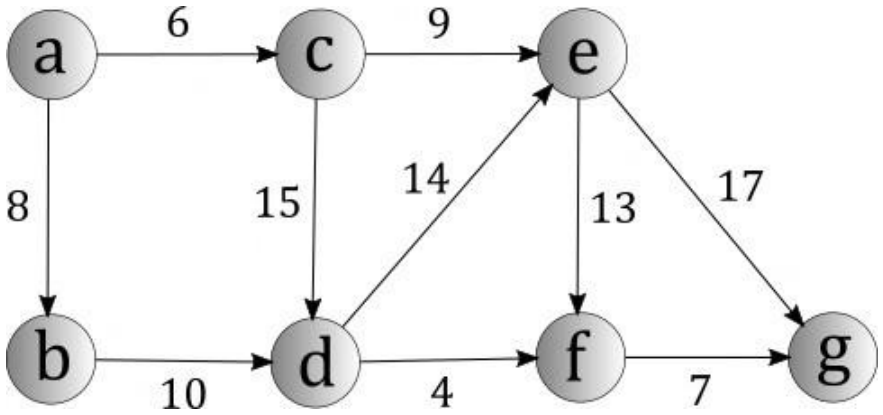


Step: 8

Repeat until all the vertices have been visited



	Init	S	A	C	D	E	B
S	0						
A	∞	1,S					
B	∞	∞	7,A	7,A	7,A	7,A	
C	∞	2,S	2,S				
D	∞	∞	∞	5,C			
E	∞	∞	∞	∞	6,D		



	Init	a	c	b	e	d	f	g
a	0							
b	∞	8,a	8,a					
c	∞	6,a						
d	∞	∞	21,c	18,b	18,b			
e	∞	∞	15,c	15,c				
f	∞	∞	∞	∞	28,e	22,d		
g	∞	∞	∞	∞	32,e	32,e	29,f	

Dijkstra's algorithm

- Time Complexity: O(E Log V)

– where, E is the number of edges and V is the number of vertices.

- Space Complexity: O(V)

Bellman-Ford Shortest Path Algorithm

Like Dijkstra, it is also a single-source shortest Path algorithm.It allows edges with negative weight in the graph.

Bellman-Ford Shortest Path Algorithm

-Set the current distance for all vertices as infinity.
-For startVertex, set the current distance to 0.
-Create a list of all edges in the graph.
-For |V| - 1 times do
-distanceToVviaU = currentDistance[u] + weight of edge (u, v)
-if (currentDistance[v] > distanceToVviaU) then
-Set currentDistance[v] to distanceToVviaU
-Set predecessor of v to u
-Done.

Floyd-Warshall algorithm

- Step 1: Initialize the distance matrix D such that D[i][j] is the shortest distance from vertex i to vertex j, or infinity if there is no path between them.
- for i in range(n):

for j in range(n):

if i == j:

D[i][j] = 0

else:

D[i][j] = infinity

Floyd-Warshall algorithm

- Step 2: For each vertex k, relax all edges connecting

vertex k to any other vertex.

```
for k in range(n):

for i in range(n):

for j in range(n):

D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

Floyd-Warshall algorithm

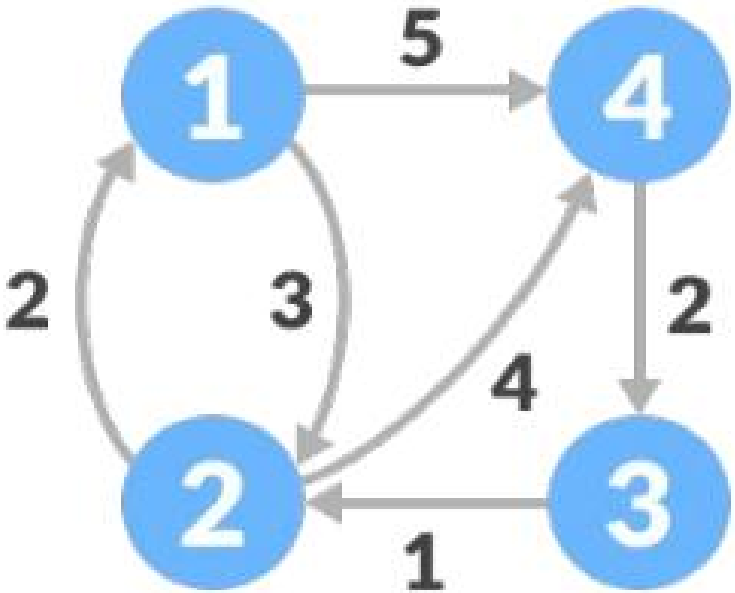
• Step 3: Repeat step 2 until the distance matrix no longer changes.

```
while True:
changed = False
for k in range(n):
for i in range(n):
for j in range(n):
if D[i][j] > D[i][k] + D[k][j]:
D[i][j] = D[i][k] + D[k][j]
changed = True
if not changed:
break
```

Floyd-Warshall - All-pairs shortest path

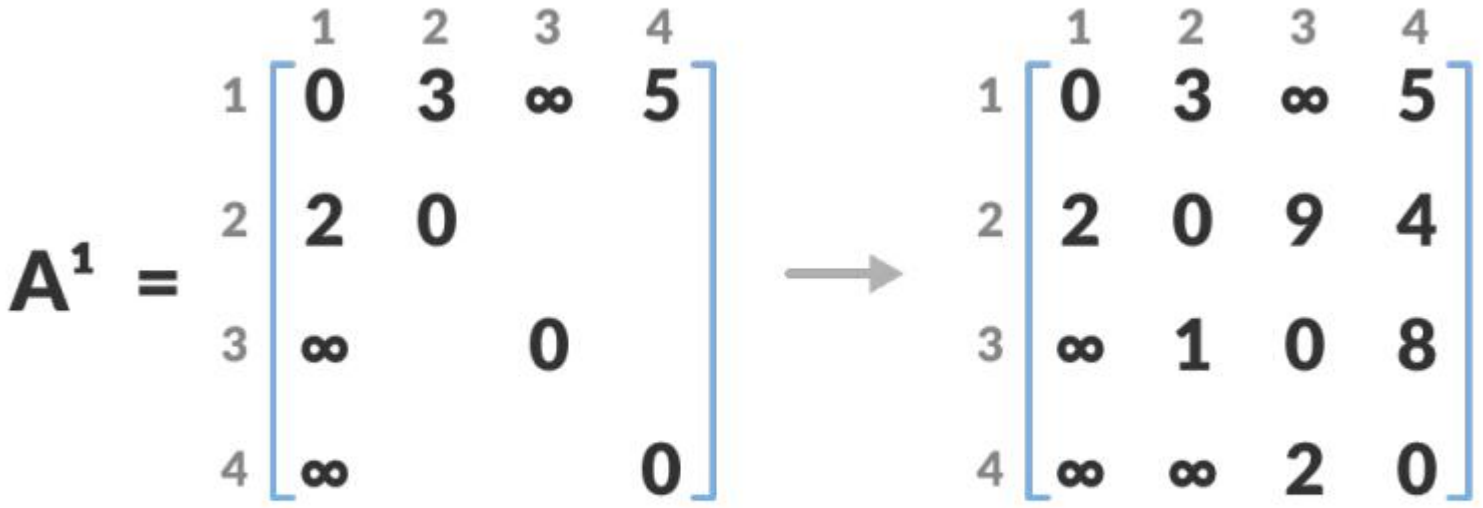
Find shortest paths in a weighted graph with positive and negative edge weights (but no negative weight cycles).
Break every possible path between two vertices (u, v) by inserting another vertex k.
Compare the sum of distance of two new paths formed (uk, kv) with the original path (uv) and record shorter path.

Floyd-Warshall - All-pairs shortest path
- Start with adjacency matrix, d,representing the graph and initialize path and set non-adjacent vertex distanceto infinity
- For each vertex k from 0 to |V| - 1
- For each vertex u from 0 to |V| - 1
- For each vertex v from 0 to |V| - 1
- if (d[u][v] > (d[u][k] + d[k][v])) then
- d[u][v]= d[u][k] + d[k][v]
- path[u][v]= path[k][v]
- Stop

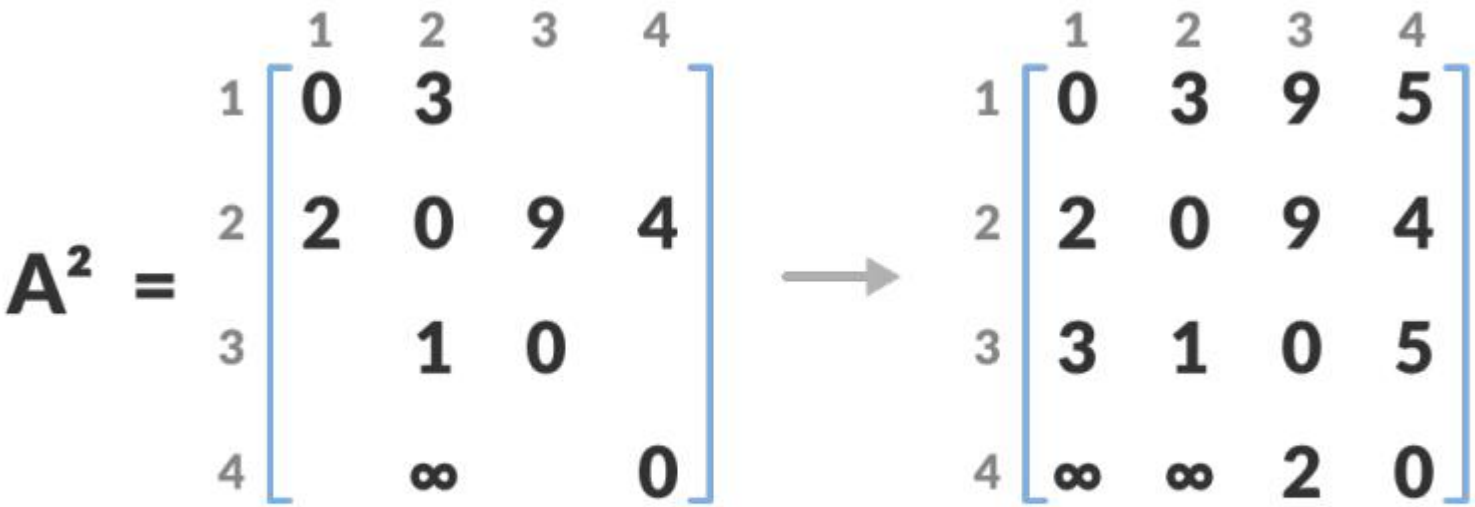

1
4
2
3
5
2
3
4
2
1

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

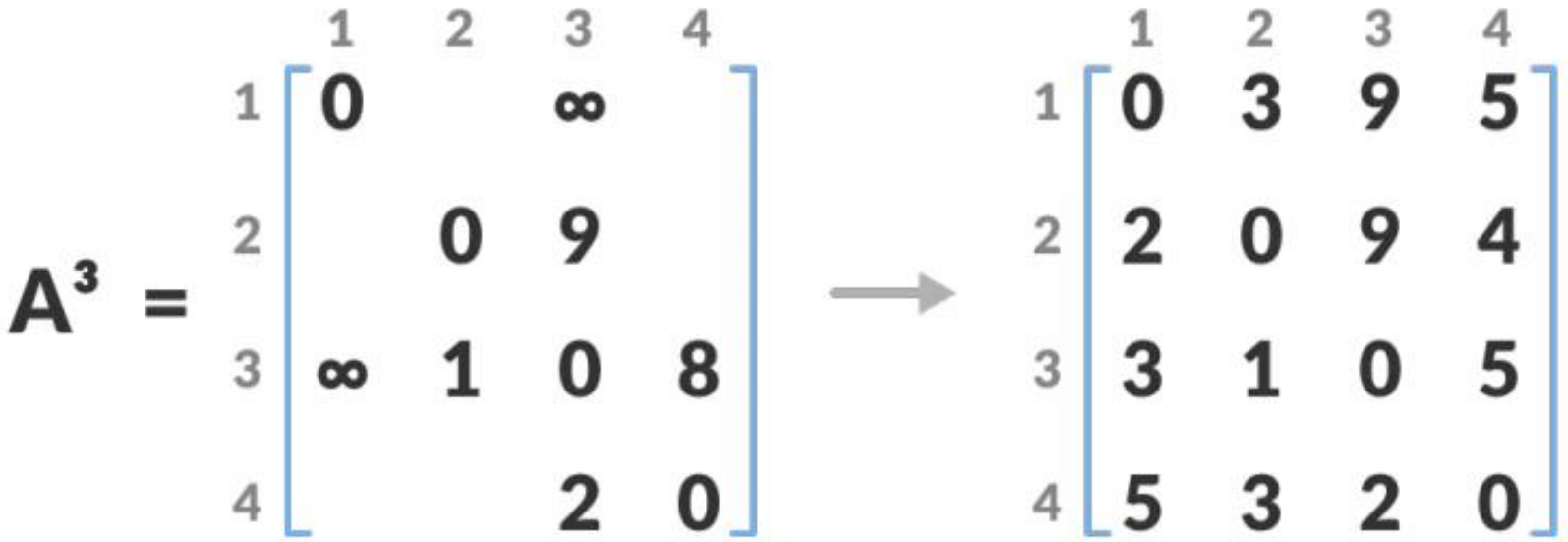
Fill each cell with the distance between ith and jth vertex



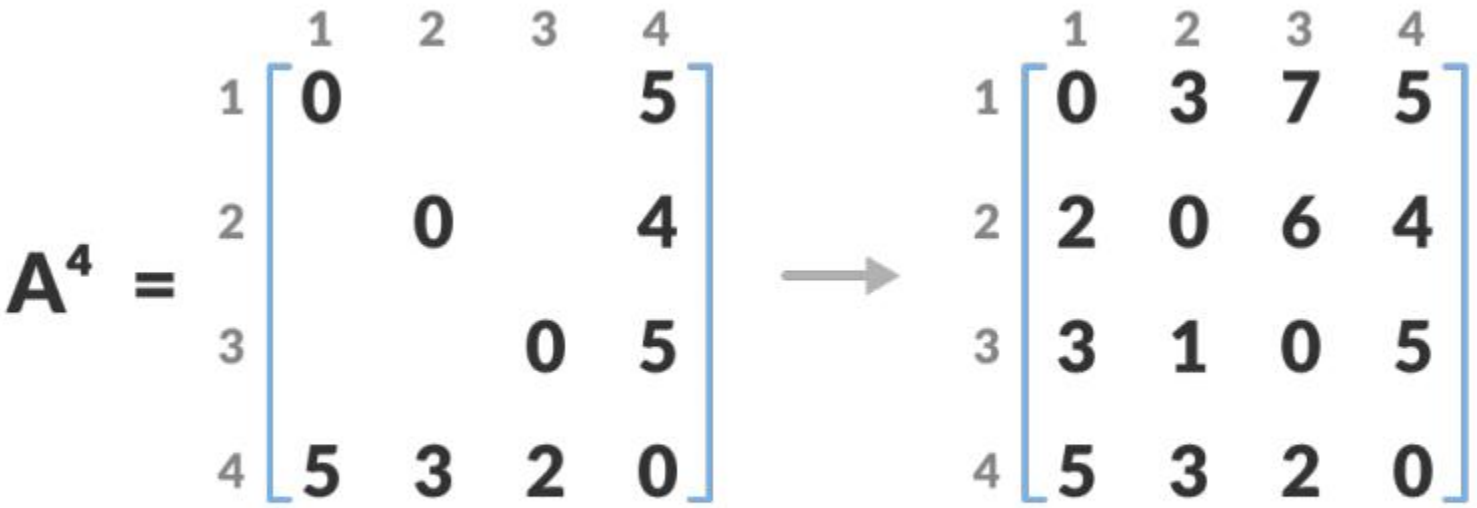
Calculate the distance from the source vertex to destination vertex through this vertex k



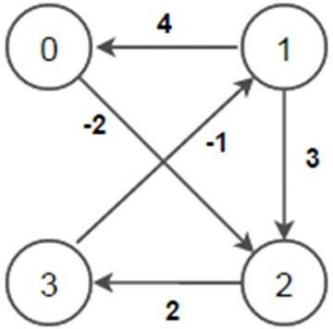
Calculate the distance from the source vertex to destination vertex through this vertex 2



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



Calculate the distance from the source vertex to destination vertex through this vertex 4



Init	0	1	2	3
0	0	∞	-2,0	∞
1	4,1	0	3,1	∞
2	∞	∞	0	2,2
3	∞	-1,3	∞	0

K = 0	0	1	2	3
0	0	∞	-2,0	∞
1	4,1	0	2,0	∞
2	∞	∞	0	2,2
3	∞	-1,3	∞	0

K = 1	0	1	2	3
0	0	∞	-2,0	∞
1	4,1	0	2,0	∞
2	∞	∞	0	2,2
3	3,1	-1,3	1,0	0

K = 2	0	1	2	3
0	0	∞	-2,0	0,2
1	4,1	0	2,0	4,2
2	∞	∞	0	2,2
3	3,1	-1,3	1,0	0

K = 3	0	1	2	3
0	0	-1,3	-2,0	0,2
1	4,1	0	2,0	4,2
2	5,1	1,3	0	2,2
3	3,1	-1,3	1,0	0

Floyd-Warshall algorithm

- Once the Floyd-Warshall algorithm has terminated, the distance matrix D will contain the shortest distances between all pairs of vertices in the graph.
- Time Complexity

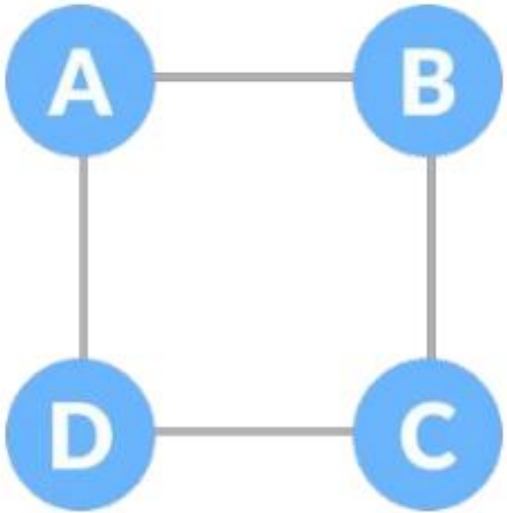
– There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is O(n3).

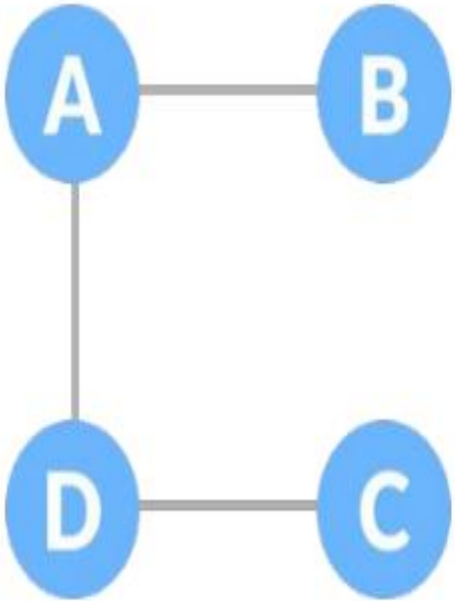
- Space Complexity

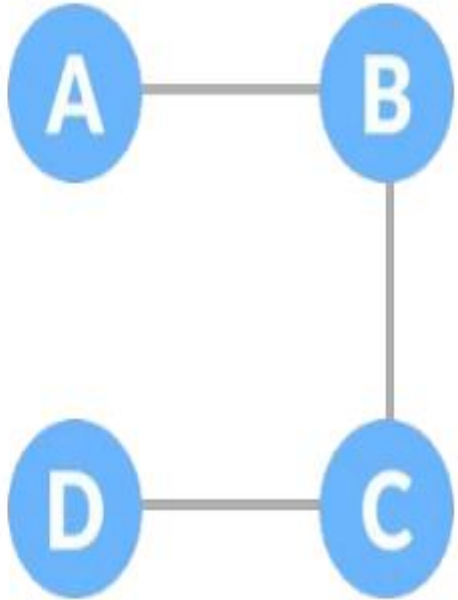
– The space complexity of the Floyd-Warshall algorithm is O(n2).

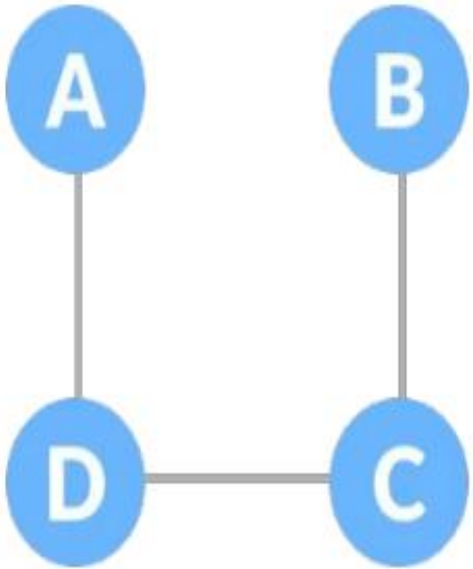
Spanning tree

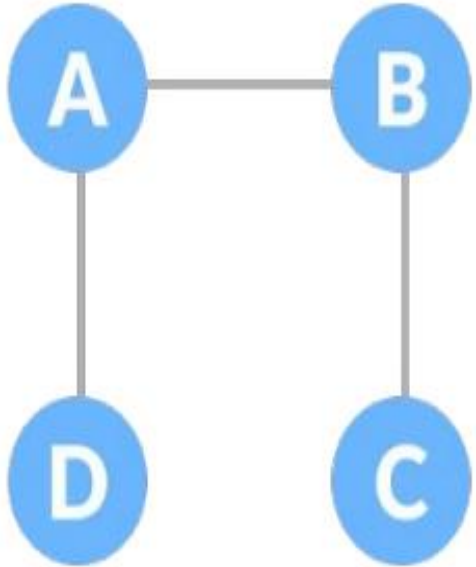
- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.
- If a vertex is missed, then it is not a spanning tree.
- The edges may or may not have weights assigned to them.
- The total number of spanning trees with n vertices that can be created from a complete graph is equal to n(n-2).

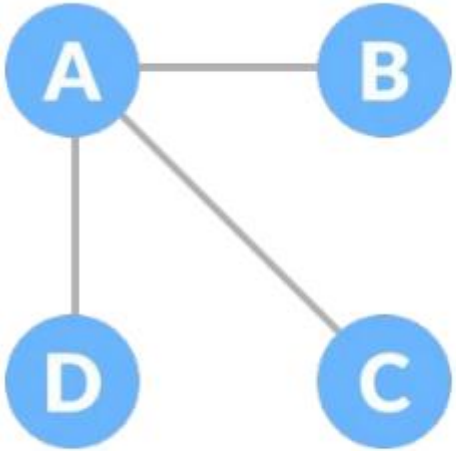

A
B
D
C

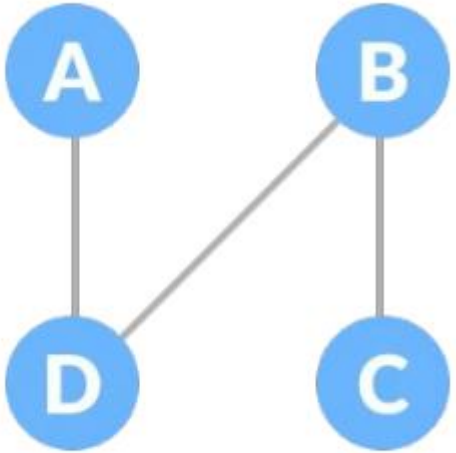

A
B
D
C


A
B
D
C


A
B
D
C

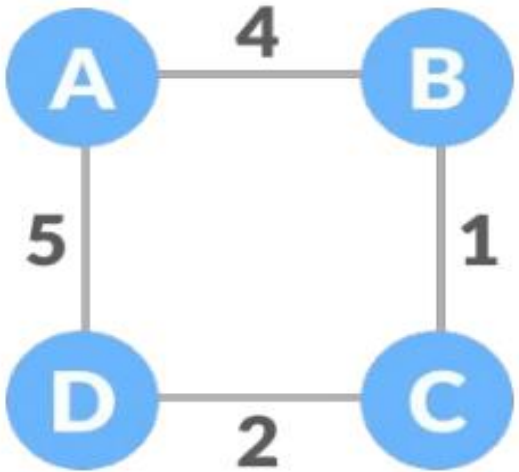

A
B
D
C

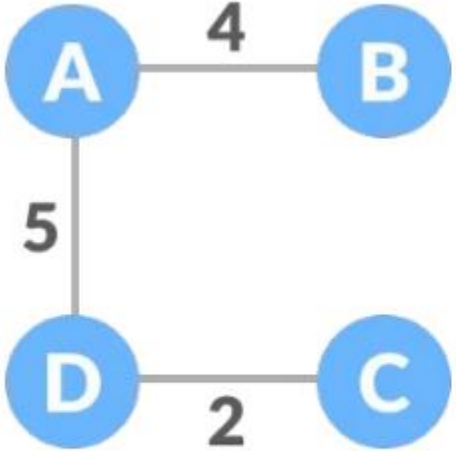

A
B
D
C


A
B
D
C

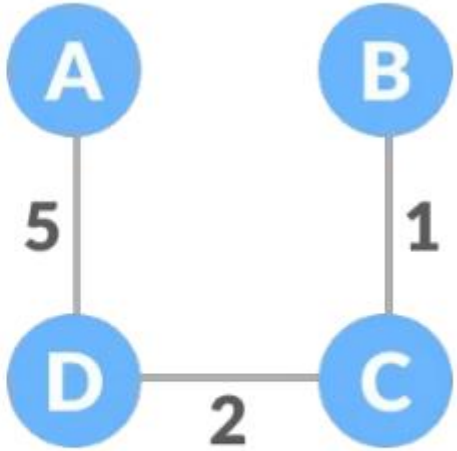
Minimum Spanning Tree

• A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

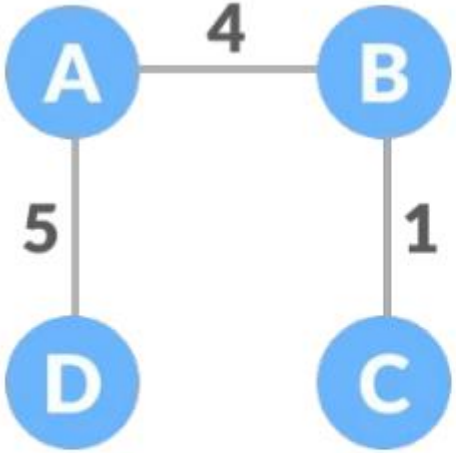




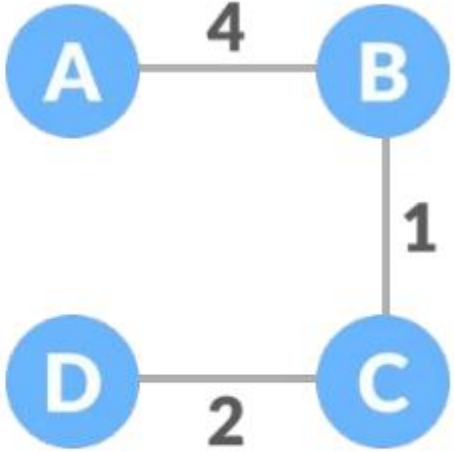
sum = 11



sum = 8



sum = 10



sum = 7

Applications

- Network routing: MSTs can be used to find efficient routes for sending data through a network. For example, a telecommunications company might use an MST to find the most cost-effective way to connect all of its cell towers.
- Circuit design: MSTs can be used to design circuits that minimize the amount of wire needed. For example, a computer chip designer might use an MST to design The circuit layout for a new chip.
- Image segmentation: MSTs can be used to segment images into different regions. For example, a medical imaging system might use an MST to segment an image of a brain into different tissue types.

Applications

• Google Maps: Google Maps uses MSTs to find the shortest routes between two points.

• Amazon: Amazon uses MSTs to design its delivery routes.

• Netflix: Netflix uses MSTs to recommend movies

and TV shows to users.

• Facebook: Facebook uses MSTs to identify groups

of friends and suggest new connections.

• Tesla: Tesla uses MSTs to design the charging

network for its electric vehicles.

Algorithms

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

– Prim's Algorithm

– Kruskal's Algorithm

Prim’s Algorithms

- Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

– form a tree that includes every vertex

– has the minimum sum of weights among all the trees that can be formed from the graph.

Prim’s Algorithms

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

– Initialize the minimum spanning tree with a vertex chosen at random.

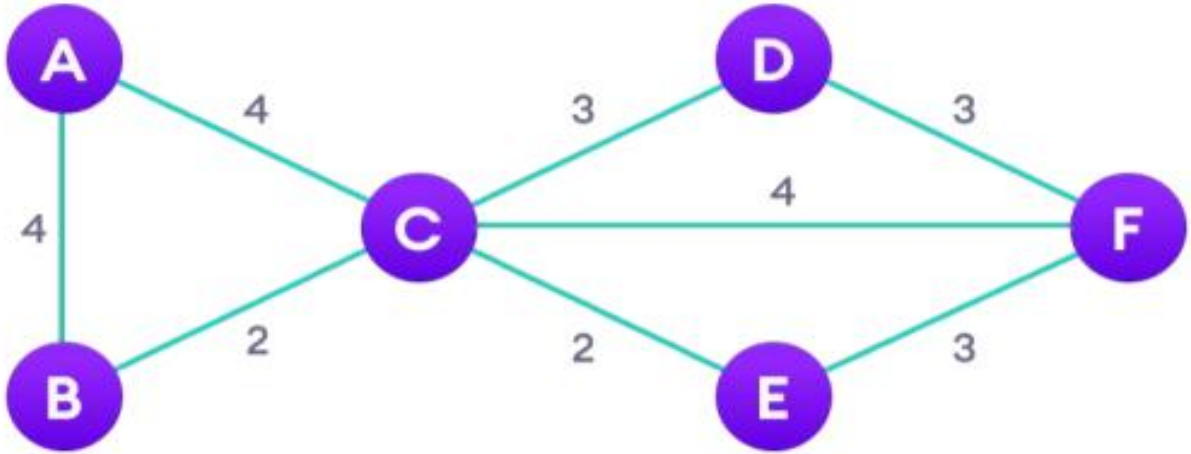
– Find all the edges that connect the tree to new vertices,

find the minimum and add it to the tree

– Keep repeating step 2 until we get a minimum spanning tree

Prim’s Algorithms

- Prim's Minimum Spanning Tree Algorithm
- -Initialise min spanning tree with any vertex from graph.
- -while min spanning tree do not have vertexCount number of vertex
- -Get all edges in the graph that connect the tree 2 newly added
- -Add the edge with min weight to the tree, if no cycle is formed.
- -Stop.

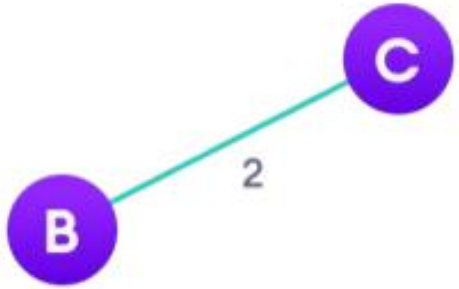


Step: 1

Start with a weighted graph

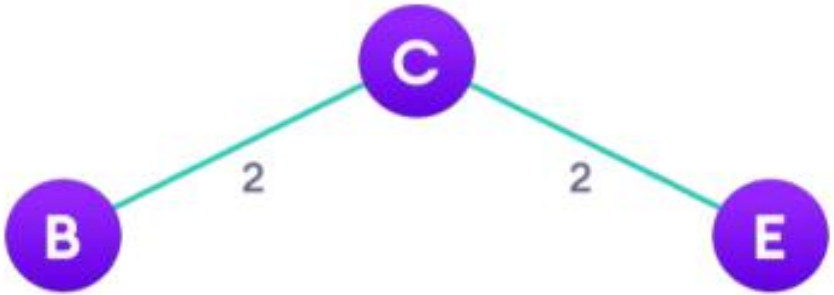
Step: 2

Choose a vertex



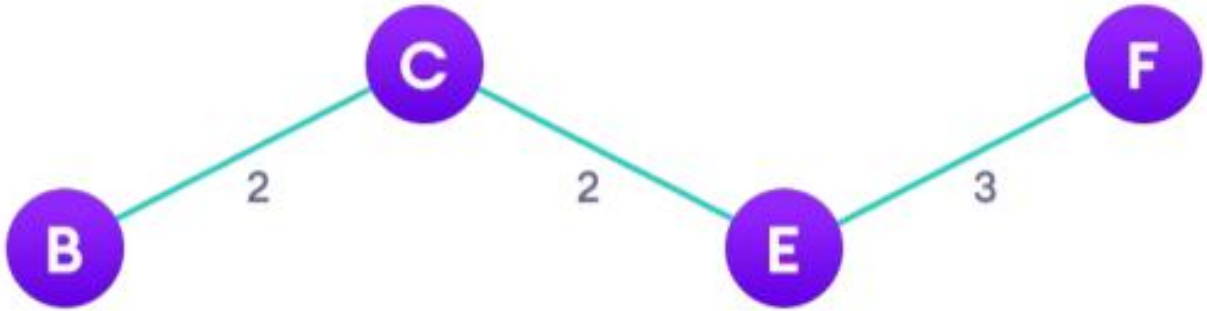
Step: 3

Choose the shortest edge from this vertex and add it



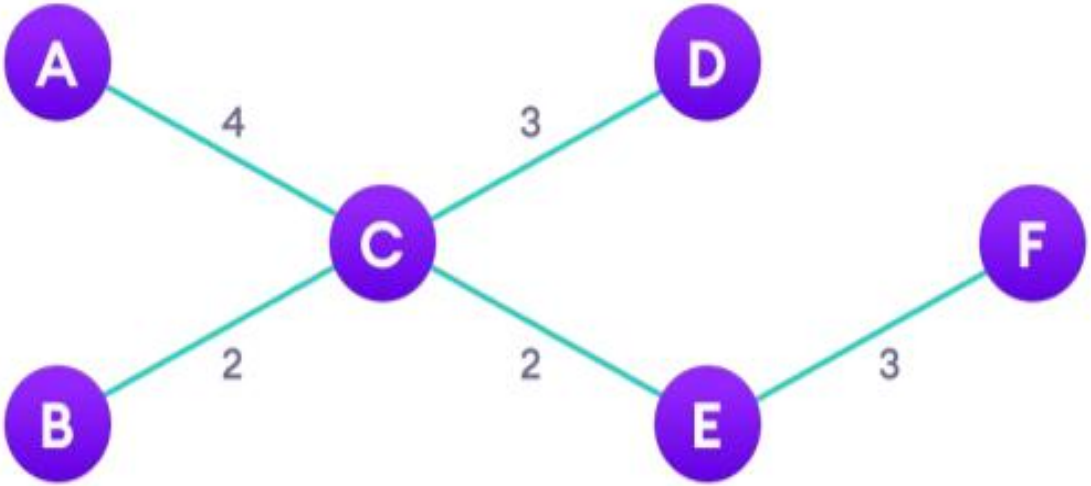
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Pseudocode

• The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't.One by one, we move vertices from set V-U to set U by connecting the least weight edge.

T = ; ∅

U = { 1 };

while (U ≠ V)

– let (u, v) be the lowest cost edge such that u U and v V ∈ ∈

- U;

– T = T {(u, v)} ∪

– U = U {v} ∪

Complexity

- The time complexity of Prim's algorithm is O(E logV).

Kruskal’s Algorithm

• It falls under a class of algorithms called greedy algorithms that find the local optimum

in the hopes of finding a global optimum.

• We start from the edges with the lowest weight and keep adding edges until we reach

our goal.

• The steps for implementing Kruskal's algorithm are as follows:

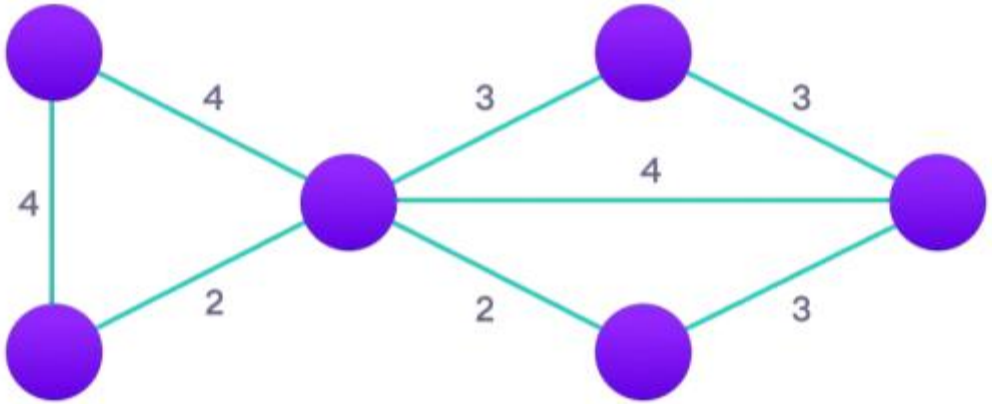
– Sort all the edges from low weight to high

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

– Keep adding edges until we reach all vertices.

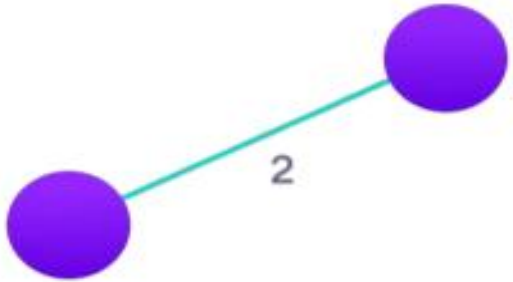
Krushkal’s Algorithm

- Kruskal's Minimum Spanning Tree Algorithm
- -Initialise min spanning tree to empty (only vertices).
- -Sort all edges in ascending order of their weight.
- -while (vertexCount - 1) edges are not added to tree do
- -Pick the edge with min weight.
- -Add an edge to the tree if it does not form a cycle.
- Stop.



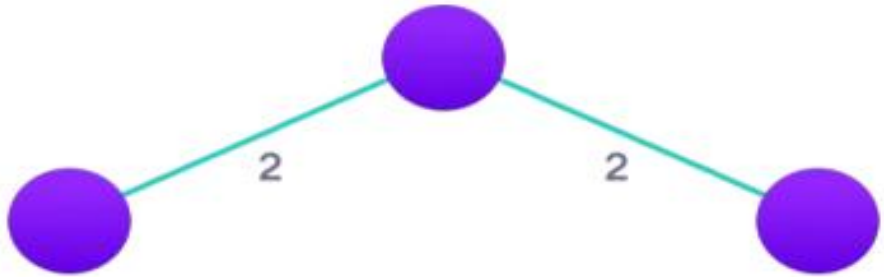
Step: 1

Start with a weighted graph



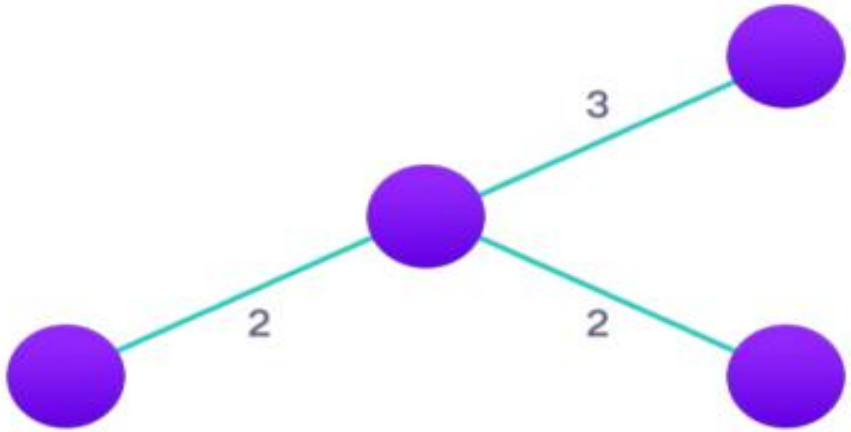
Step: 2

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



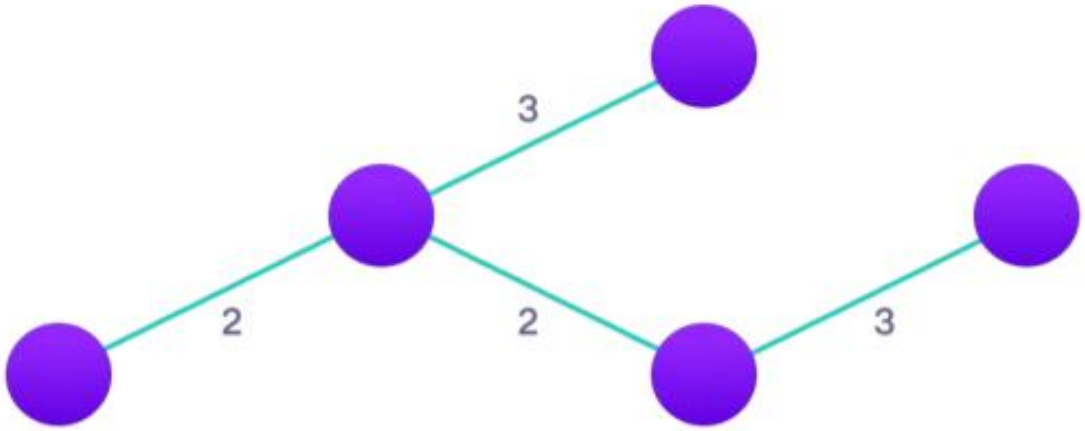
Step: 3

Choose the next shortest edge and add it



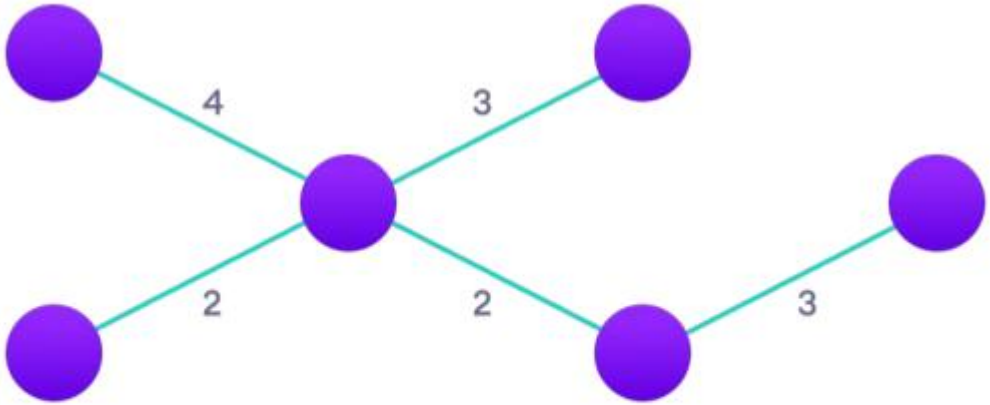
Step: 4

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



Step: 5

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



Step: 6

Repeat until you have a spanning tree