

# DSA Graph theory

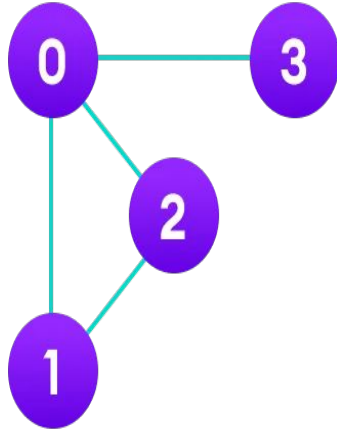
## Chapter 8

# Graph

A graph data structure is a collection of nodes that have data and are connected to other nodes.

A graph is a data structure  $(V, E)$  that consists of

- A collection of vertices  $V$
- A collection of edges  $E$ , represented as ordered pairs of vertices  $(u,v)$



In this graph,

$$V = \{0, 1, 2, 3\}$$

$$E = \{(0,1), (0,2), (0,3), (1,2)\}$$

$$G = \{V, E\}$$

A **directed graph**  $G$  is defined as an ordered pair  $(V, E)$  where,  $V$  is a set of vertices and the ordered pairs in  $E$  are called edges on  $V$ . A directed graph can be represented geometrically as a set of marked points (called vertices)  $V$  with a set of arrows (called edges)  $E$  between pairs of points (or vertex or nodes) so that there is at most one arrow from one vertex to another vertex.

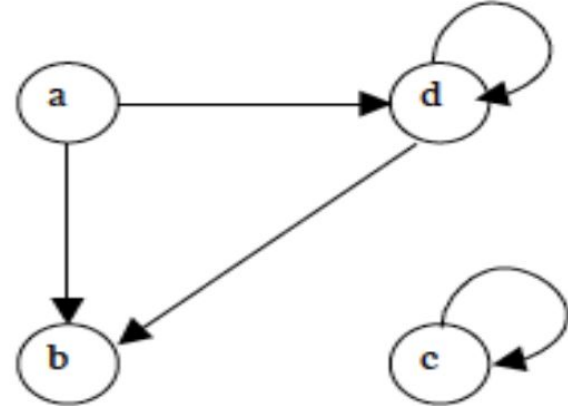


Fig2: Directed Graph

An ***undirected graph***  $G$  is defined abstractly as an ordered pair  $(V, E)$ , where  $V$  is a set of vertices and the  $E$  is a set at edges. An undirected graph can be represented geometrically as a set of marked points (called vertices)  $V$  with a set at lines (called edges)  $E$  between the points.

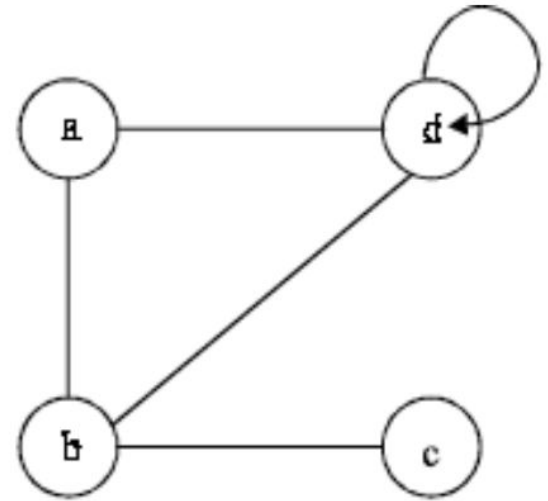
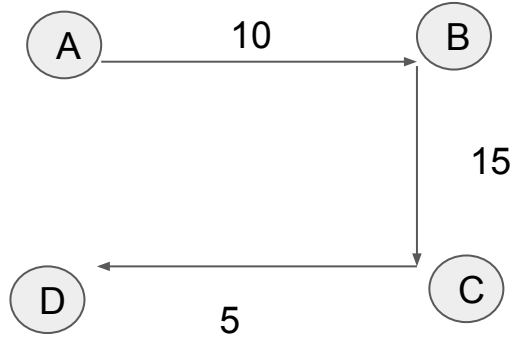


Fig3: Undirected Graph

A graph  $G$  is said to be ***weighted graph*** if every edge and/or vertices in the graph is assigned with some weight or value.



An undirected graph is said to be **connected graph** if there exist a path from any vertex to any other vertex. Otherwise it is said to be disconnected.

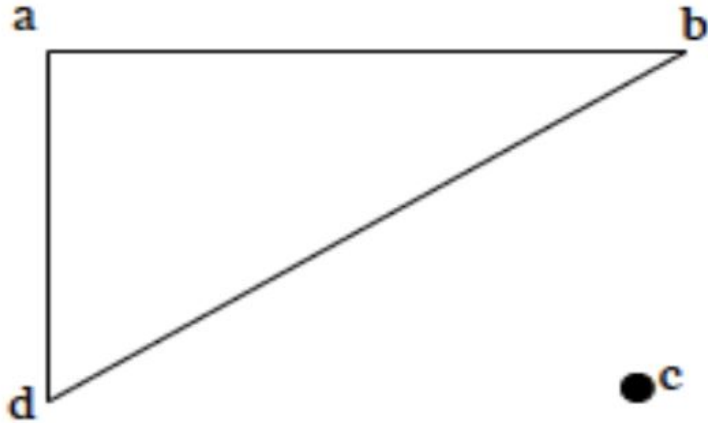


Fig6 (a): Disconnected Graph

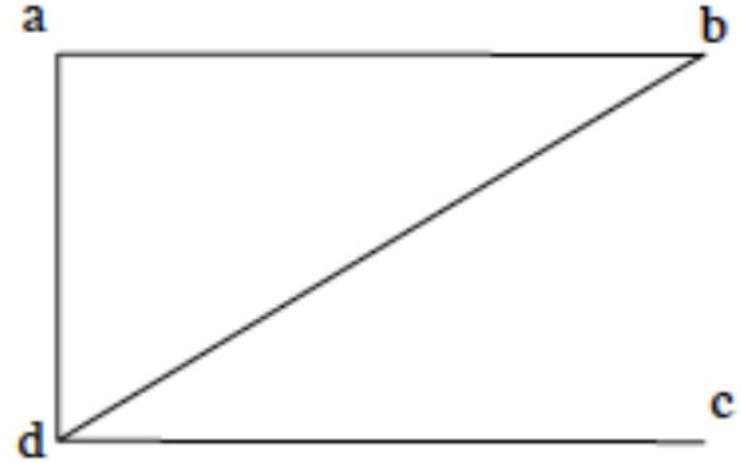


Fig6 (b): Connected Graph

A graph  $G$  is said to be complete (*strongly connected*) if there is a path from every vertex to every other vertex. Let  $a$  and  $b$  be two vertices in the directed graph, then it is a complete graph if there is a path from  $a$  to  $b$  as well as a path from  $b$  to  $a$ . A complete graph with  $n$  vertices will have  $n(n-1)/2$  edges

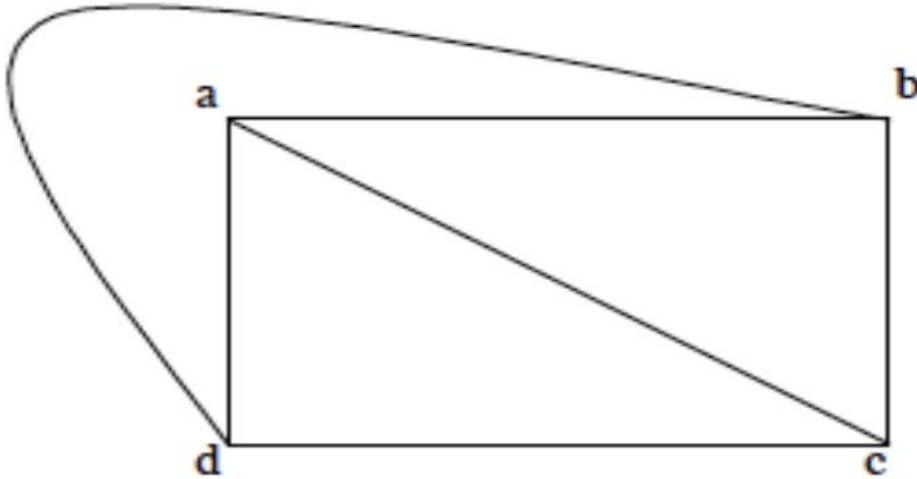


Fig7 (a): Complete undirected Graph

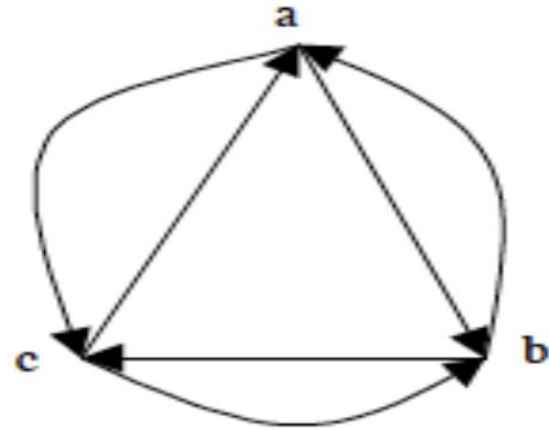


Fig7 (b): Complete Directed Graph

In a directed graph, a path is a sequence of edges ( $e_1, e_2, e_3, \dots, e_n$ ) such that the edges are connected with each other (i.e., terminal vertex  $e_n$  coincides with the initial vertex  $e_1$ ). A path is said to be *elementary* if it does not meet the same vertex twice. A path is said to be simple if it does not meet the same edges twice.

A circuit is a path ( $e_1, e_2, \dots, e_n$ ) in which terminal vertex of  $e_n$  coincides with initial vertex of  $e_1$ . A circuit is said to be simple if it does not include (or visit) the same edge twice. A circuit is said to be elementary if it does not visit the same vertex twice.



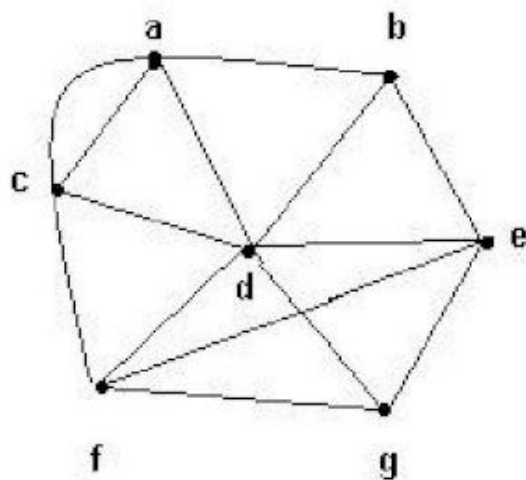
# Degree of a vertex

---

- The *degree* of a vertex  $v$ , denoted by  $\delta(v)$ , is the number of edges incident on  $v$

- Example:

- $\delta(a) = 4, \delta(b) = 3,$
- $\delta(c) = 4, \delta(d) = 6,$
- $\delta(e) = 4, \delta(f) = 4,$
- $\delta(g) = 3.$



# Sum of the degrees of a graph

---

Theorem: If  $G$  is a graph with  $m$  edges and  $n$  vertices  $v_1, v_2, \dots, v_n$ , then

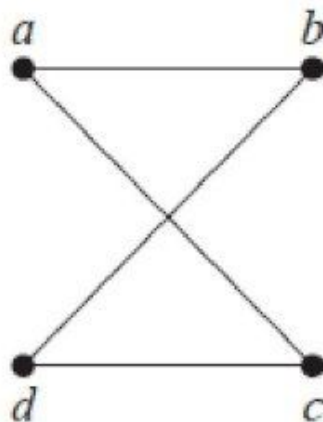
$$\sum_{i=1}^n \delta(v_i) = 2m$$

In particular, the sum of the degrees of all the vertices of a graph is even.

## Try yourself

---

- Draw a graph with the adjacency matrix



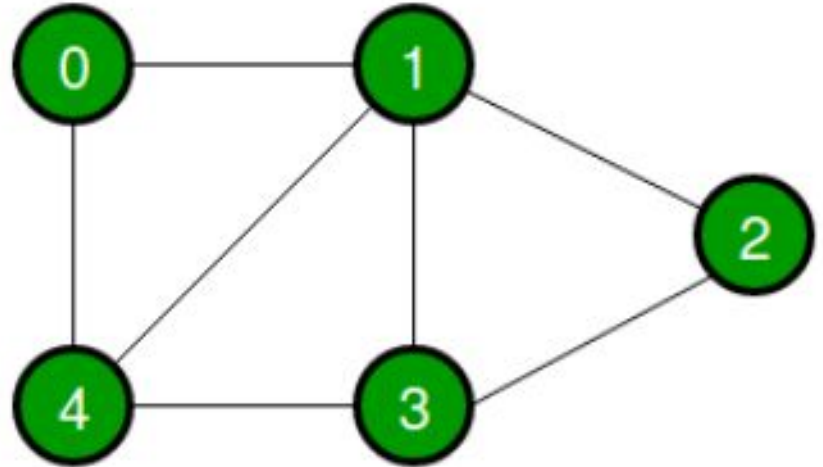
# Representation of Graphs

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix

2. Adjacency List

Let us consider an undirected graph



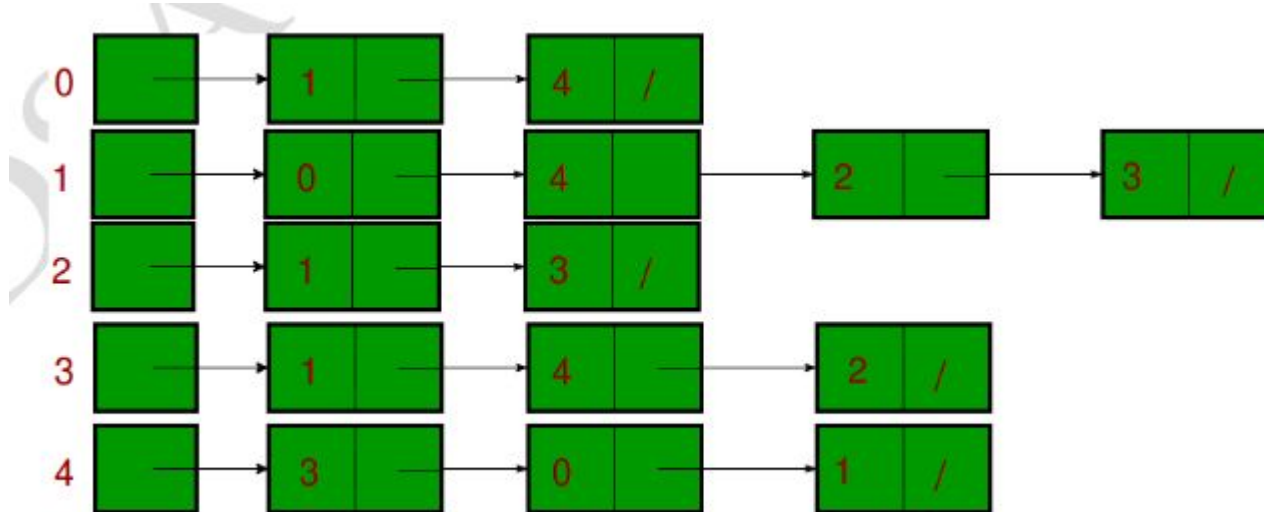
# Adjacency Matrix

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

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

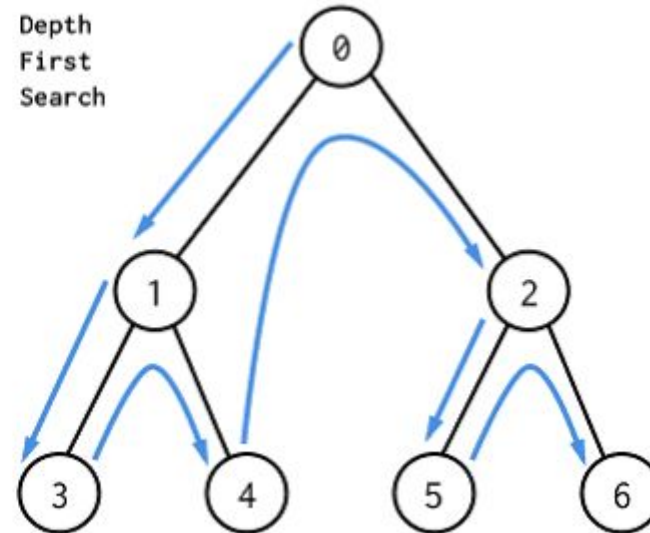
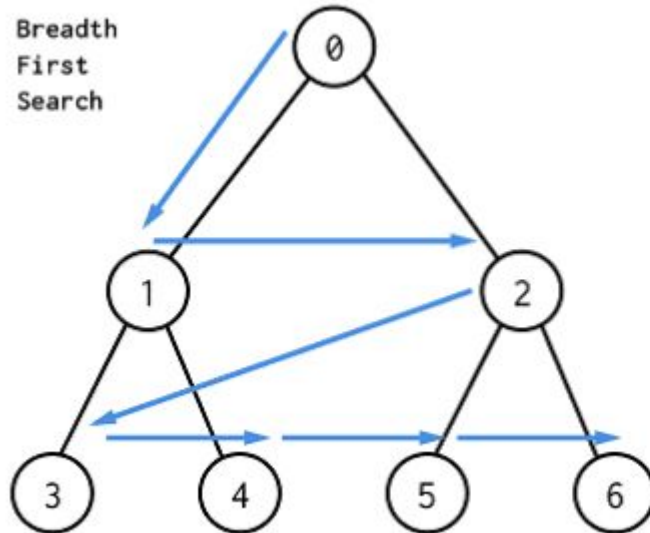
# Adjacency List

- An array of linked lists is used.
- The size of the array is equal to the number of vertices. Let the array be an array[].
- An entry array[i] represents the list of vertices adjacent to the ith vertex.
- This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.



# Graph Traversal

- Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph.
- Such traversals are classified by the order in which the vertices are visited.
- Tree traversal is a special case of graph traversal.

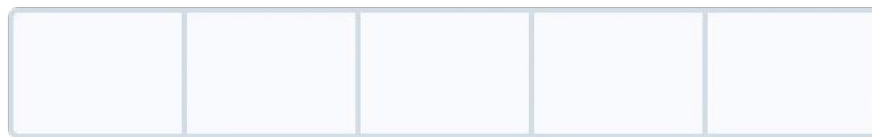
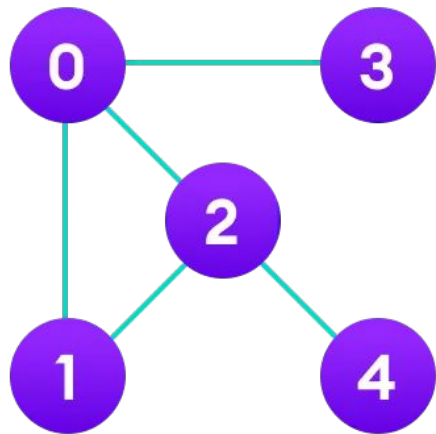


# BFS

The algorithm works as follows:

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



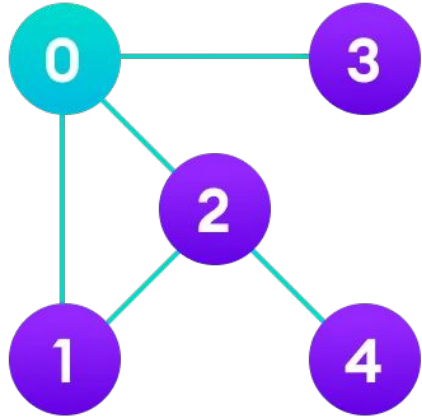


**Visited**



**Queue**

↑  
**FRONT**

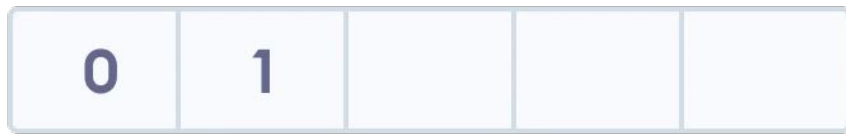
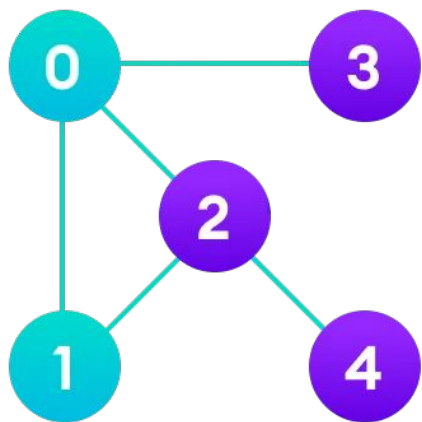


Visited



Queue

↑  
FRONT



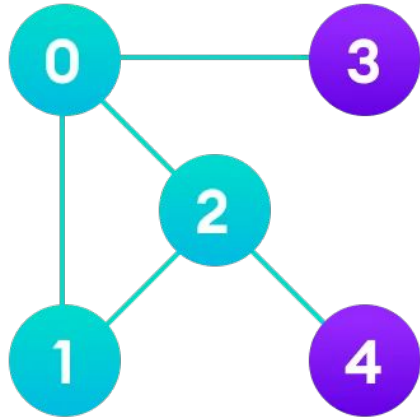
**Visited**



**Queue**



**FRONT**



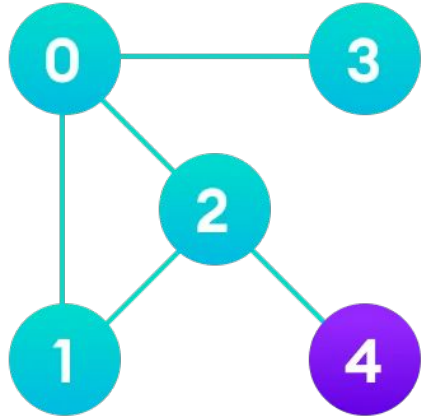
0	1	2		
---	---	---	--	--

**Visited**

3	4			
---	---	--	--	--

**Queue**

↑  
**FRONT**



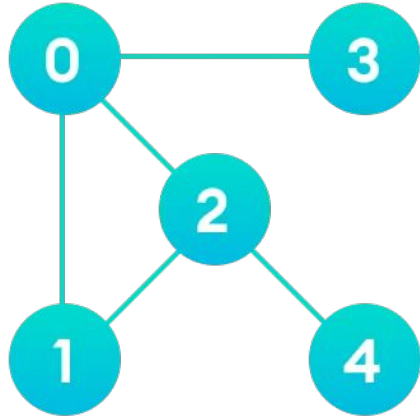
0	1	2	3	
---	---	---	---	--

Visited

4				
---	--	--	--	--

Queue

↑  
FRONT



Visited



Queue



FRONT

# BFS Algorithm

1. Input the vertices of the graph and its edges  $G = (V, E)$
2. Input the source vertex and assign it to the variable  $S$ .
3. Add or push the source vertex to the queue.
4. Repeat the steps 5 and 6 until the queue is empty (i.e.,  $\text{front} > \text{rear}$ )
5. Pop the front element of the queue and display it as visited.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the queue and displayed (i.e., not visited).
7. Exit

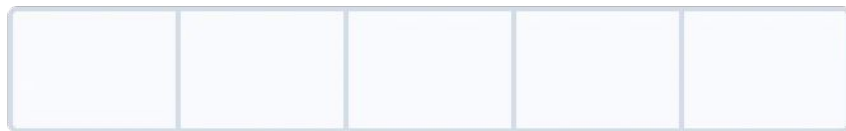
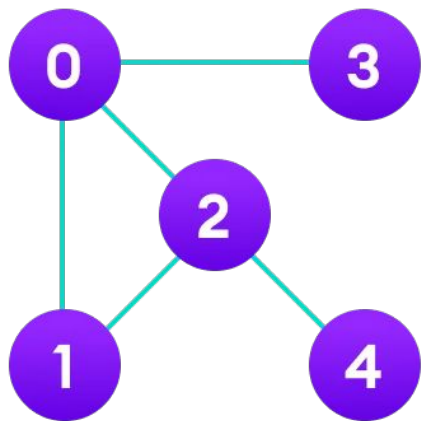
# DFS

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

The DFS algorithm works as follows:

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

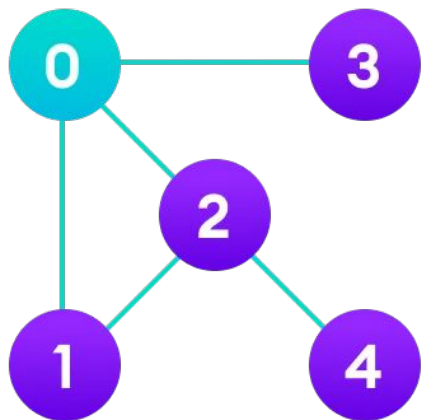




**Visited**



**Stack**

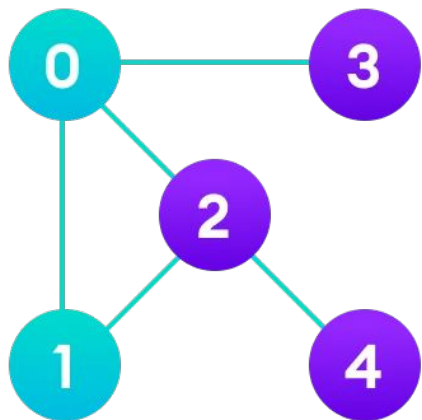


0				
---	--	--	--	--

**Visited**

1	2	3		
---	---	---	--	--

**Stack**

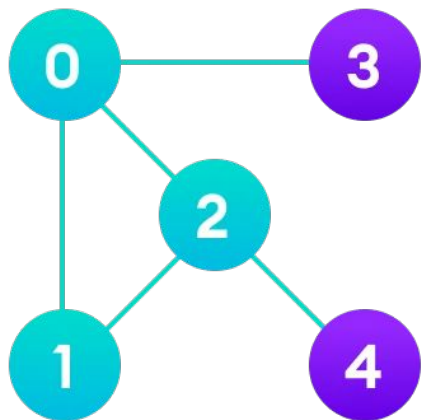


0	1			
---	---	--	--	--

**Visited**

2	3			
---	---	--	--	--

**Stack**

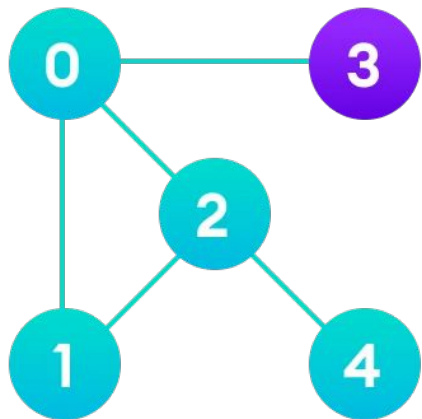


0	1	2		
---	---	---	--	--

**Visited**

4	3			
---	---	--	--	--

**Stack**

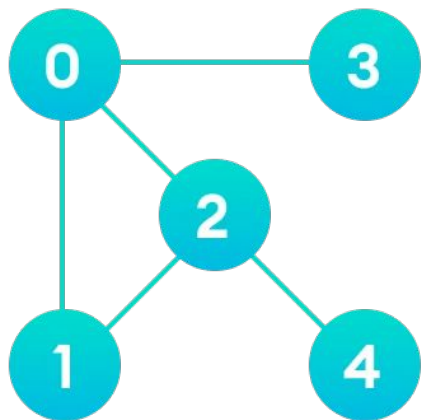


0	1	2	4	
---	---	---	---	--

**Visited**

3				
---	--	--	--	--

**Stack**



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

**Visited**

--	--	--	--	--

**Stack**

# DFS Algorithm

1. Input the vertices and edges of the graph  $G = (V, E)$ .
2. Input the source vertex and assign it to the variable  $S$ .
3. Push the source vertex to the stack.
4. Repeat the steps 5 and 6 until the stack is empty and all nodes visited.
5. Pop the top element of the stack and display it.
6. Push the vertices, which are neighbors to just popped element, if it is not in the stack and displayed (ie; not visited).
7. Exit.

# Transitive Closure and Warshall Algorithm

Transitive closure: The adjacency matrix,  $A$ , of a graph,  $G$ , is the matrix with elements  $a_{ij}$  such that  $a_{ij} = 1$  implies there is an edge from  $i$  to  $j$ . If there is no edge then  $a_{ij} = 0$ .

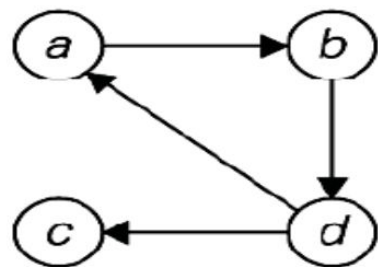
Let  $A$  be the adjacency matrix of  $R$  and  $T$  be the adjacency matrix of transitive closure of  $R$ .  $T$  is called the reachability matrix of digraph (directed graph)  $D$  due to the property that  $T_{i,j} = 1$  if and only if  $v_j$  can be reached from  $v_i$  in  $D$  by a sequence of arcs(edges). In simple word the definition is as:

A path exists between two vertices  $i, j$ , if

- there is an edge from  $i$  to  $j$ ; or
- there is a path from  $i$  to  $j$  going through vertex 1; or
- there is a path from  $i$  to  $j$  going through vertex 1 and/or 2; or
- there is a path from  $i$  to  $j$  going through vertex 1, 2, and/or 3; or
- there is a path from  $i$  to  $j$  going through any of the other vertices



The following figure shows that the digraph(directed graph, its adjacency matrix and its transitive closure).



(a)

Digraph

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

(b)

Adjacency matrix

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

(c)

Transitive closure

## Warshall's algorithm for finding transitive closure from digraph

In warshall's algorithm we construct a sequence of Boolean matrices  $A = W[0], W[1], W[2], \dots, W[n] = T$ , where  $A$  is adjacency matrix and  $T$  is its transitive closure. This can be done from digraph  $D$  as follows.

$[W[1]]_{i,j} = 1$  if and only if there is a path from  $v_i$  to  $v_j$  with elements of a subset of  $\{v_1\}$  as interior vertices.

$[W[2]]_{i,j} = 1$  if and only if there is a path from  $v_i$  to  $v_j$  with elements of a subset of  $\{v_1, v_2\}$  as interior vertices.

continuing this process, we generalized to

$[W[k]]_{i,j} = 1$  if and only if there is a path from  $v_i$  to  $v_j$  with elements of a subset of  $\{v_1, v_2, \dots, v_k\}$  as interior vertices.

Transitive closure: Transitive closure of a binary relation  $R$  on a set  $A$  is the smallest transitive relation on a set  $A$  that contains  $R$ .

$$R_t^+ = R \cup \{(a, c) \mid (a, b) \in R \wedge (b, c) \in R\}$$

**Problem:** Let  $R$  be the relation on a set  $\{1, 2, 3\}$  containing the ordered pairs  $(1, 1)$ ,  $(2, 3)$ , and  $(3, 1)$ . Find the transitive closure of  $R$ .

**Solution:**  $R = \{(1, 1), (2, 3), (3, 1)\}$   
 $A = \{1, 2, 3\}$

$$R_t^+ = R \cup \{(a, c) \mid (a, b) \in R \wedge (b, c) \in R\}$$

$$R_t^+ = \{(1, 1), (2, 3), (3, 1), (2, 1)\}$$

# Warshall Algorithm (Finding the transitive closure)

## Finding the transitive closure using Warshall's Algorithm

Sometimes it is difficult to find all the ordered pairs in transitive closure of a relation. Warshall's Algorithm is considered an efficient method in finding the transitive closure of a relation.

**Example:** By using Warshall's algorithm, find the transitive closure of the relation  $R = \{(2, 1), (2, 3), (3, 1), (3, 4), (4, 1), (4, 3)\}$  on set  $A = \{1, 2, 3, 4\}$ .

**Solution:** First, we will represent the relation  $R$  in matrix form.

$$R = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}_{4 \times 4}$$

Understand that there are 4 elements in set  $A$ . Therefore, 4 steps are required in order to find the transitive closure of relation  $R$  [According to Warshall's Algorithm].

>> In Step 1, we will consider 1st column and 1st row of the above matrix i.e.,  $C_1$  and  $R_1$ .

Write all positions where 1 is present in column 1.

$$C_1 = \{2, 3, 4\}$$

Also, write all position where 1 is present in row 1

$$R_1 = \emptyset$$

Now, take the cross product of  $C_1$  and  $R_1$ .

$$C_1 \times R_1 = \emptyset.$$

Therefore, no new additions.

$$R = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}_{4 \times 4}$$

>> In Step 2, we will consider 2nd column and 2nd row of the above matrix.

$$C_2$$

$$R_2$$

$$C_2 \times R_2 = \emptyset$$

$$\emptyset$$

$$\{1, 3\}$$

Therefore, no new additions.

>> In Step 3, we will consider 3rd column and 3rd row.

$$C_3$$

$$R_3$$

$$C_3 \times R_3 = \{(2, 1), (2, 4), (4, 1), (4, 4)\}$$

$$\{2, 4\}$$

$$\{1, 4\}$$

$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{array}{c} \\ \\ \\ \end{array} \begin{array}{c} \text{New Matrix} \\ \\ \\ \end{array}$$

$4 \times 4$

>> In Step 4, we will consider 4th column and 4th row of the above matrix.

$$\begin{array}{cc} C_4 & R_4 \\ \{2, 3, 4\} & \{1, 3, 4\} \end{array}$$

$$C_4 \times R_4 = \{(2, 1), (2, 3), (2, 4), (3, 1), (3, 3), (3, 4), (4, 1), (4, 3), (4, 4)\}$$

$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{array}{c} \\ \\ \\ \end{array} \begin{array}{c} \text{New Matrix} \\ \\ \\ \end{array}$$

$4 \times 4$

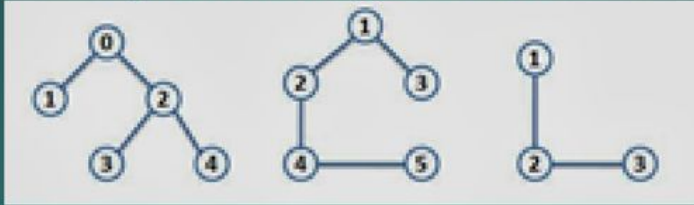
$$R_t^+$$

$$R_t^+ = \{(2, 1), (2, 3), (2, 4), (3, 1), (3, 3), (3, 4), (4, 1), (4, 3), (4, 4)\}$$

# Spanning Tree

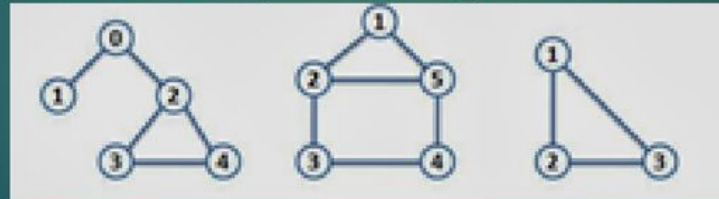
– A graph that connects all vertices, WITHOUT creating/containing a cycle.

## Spanning trees



These graphs do NOT contain cycles

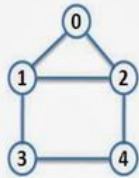
## NOT Spanning trees



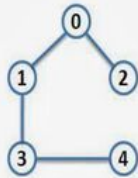
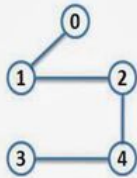
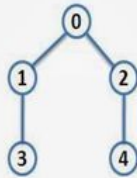
These graphs contain cycles



# Creating a spanning tree from a connected graph



Connected Graph (G)



Spanning Trees of Graph (G)

- ▶ Three possible spanning trees have been created from Graph (G).
- ▶ Notice how all of the vertices are connected in each spanning tree, but none of these graphs contain any of the original cycles.



# Minimum Spanning tree

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together.

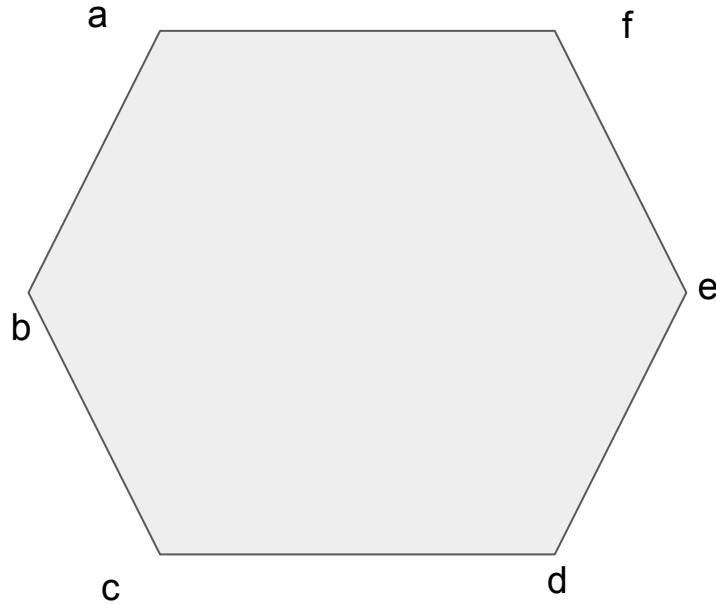
A single graph can have many different spanning trees.

A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree.

The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

A minimum spanning tree has  $(V - 1)$  edges where  $V$  is the number of vertices in the given graph.

Given graph, how many spanning tree can be formed?



## Minimizing costs



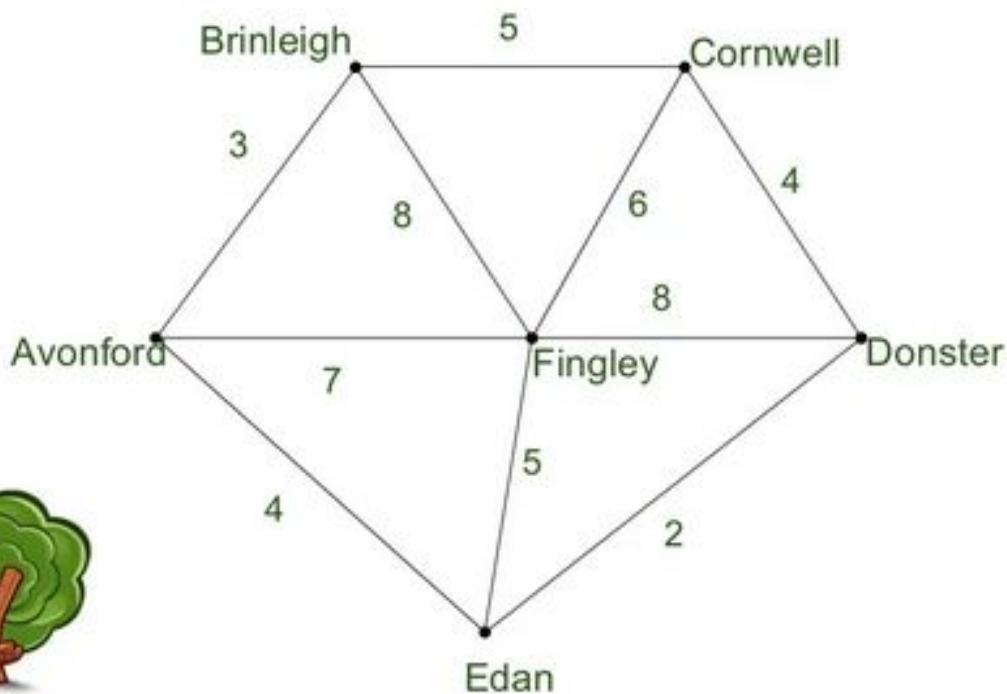
Suppose you want to supply a set of houses (say, in a new subdivision) with:

- electric power
- water
- sewage lines
- telephone lines

- ✓ To keep costs down, you could connect these houses with a spanning tree (of, for example, power lines)
- ✓ However, the houses are not all equal distances apart
- ✓ To reduce costs even further, you could connect the houses with a *minimum-cost* spanning tree

## Example

A cable company want to connect five villages to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



# MINIMUM SPANNING TREE



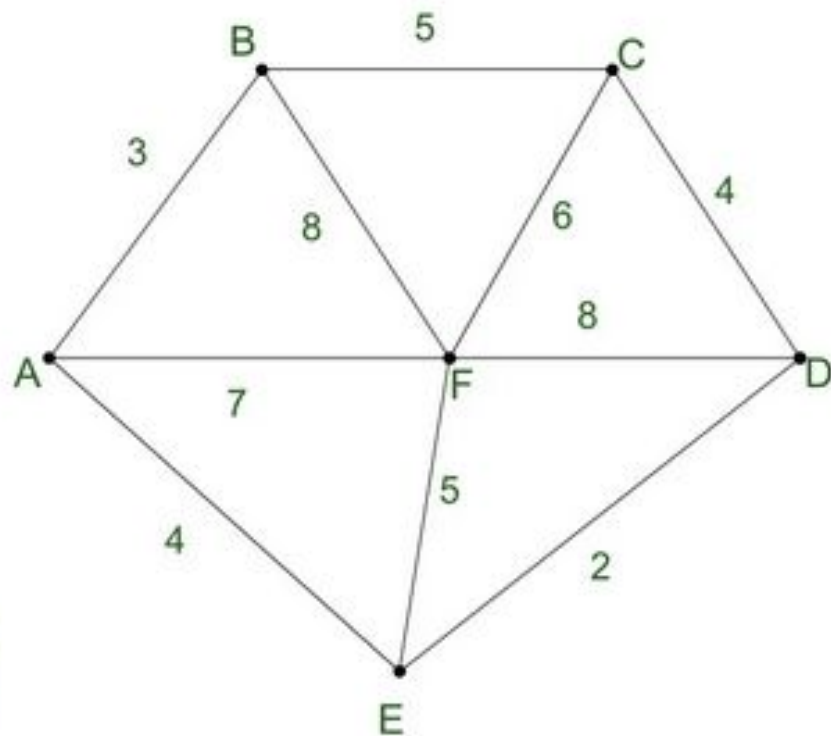
Let  $G = (N, A)$  be a connected, undirected graph where  $N$  is the set of nodes and  $A$  is the set of edges. Each edge has a given nonnegative length. The problem is to find a subset  $T$  of the edges of  $G$  such that all the nodes remain connected when only the edges in  $T$  are used, and the sum of the lengths of the edges in  $T$  is as small as possible possible. Since  $G$  is connected, at least one solution must exist.

# Finding Spanning Trees



- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms
- **Kruskal's algorithm:**  
Created in 1957 by Joseph Kruskal
- **Prim's algorithm**  
Created by Robert C. Prim

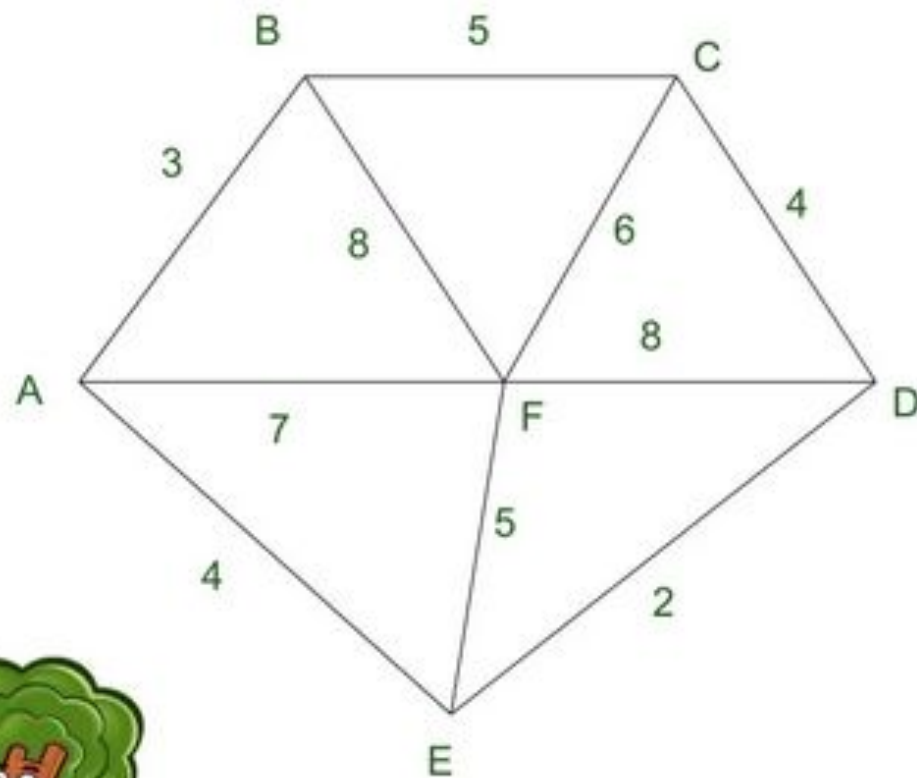
We model the situation as a network, then the problem is to find the minimum connector for the network





# Kruskal's Algorithm

List the edges in  
order of size:

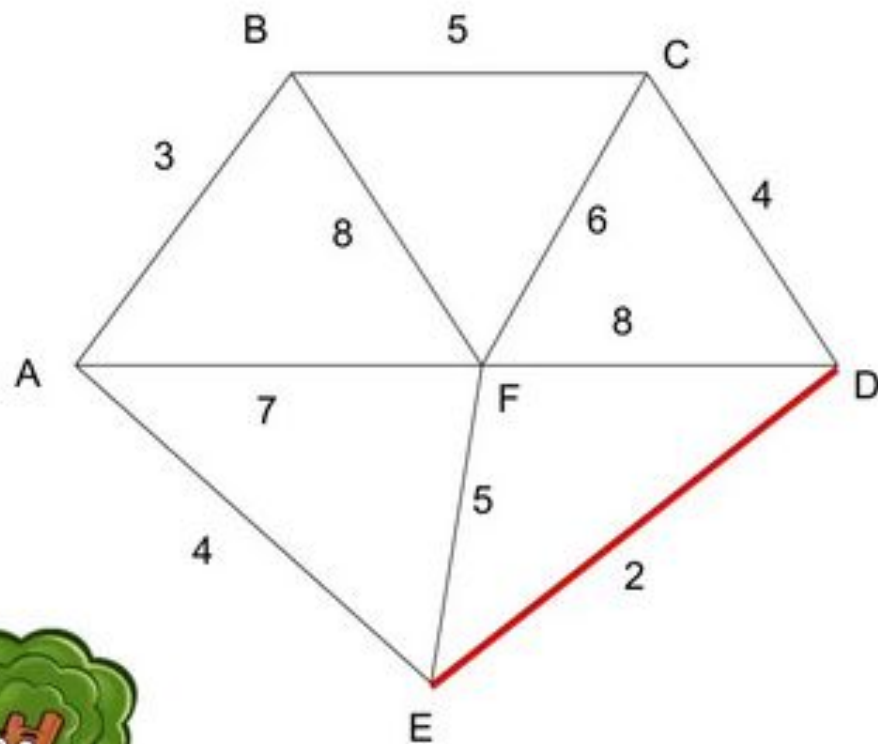


ED 2  
AB 3  
AE 4  
CD 4  
BC 5  
EF 5  
CF 6  
AF 7  
BF 8  
CF 8





# Kruskal's Algorithm

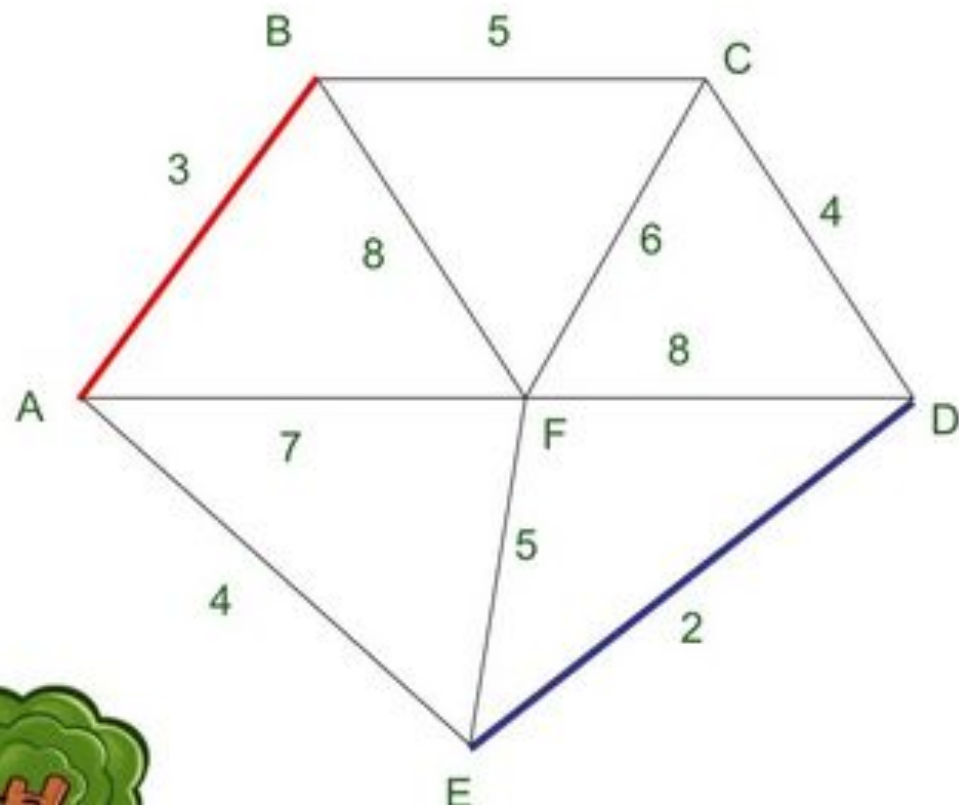


Select the shortest  
edge in the network

ED 2



# Kruskal's Algorithm



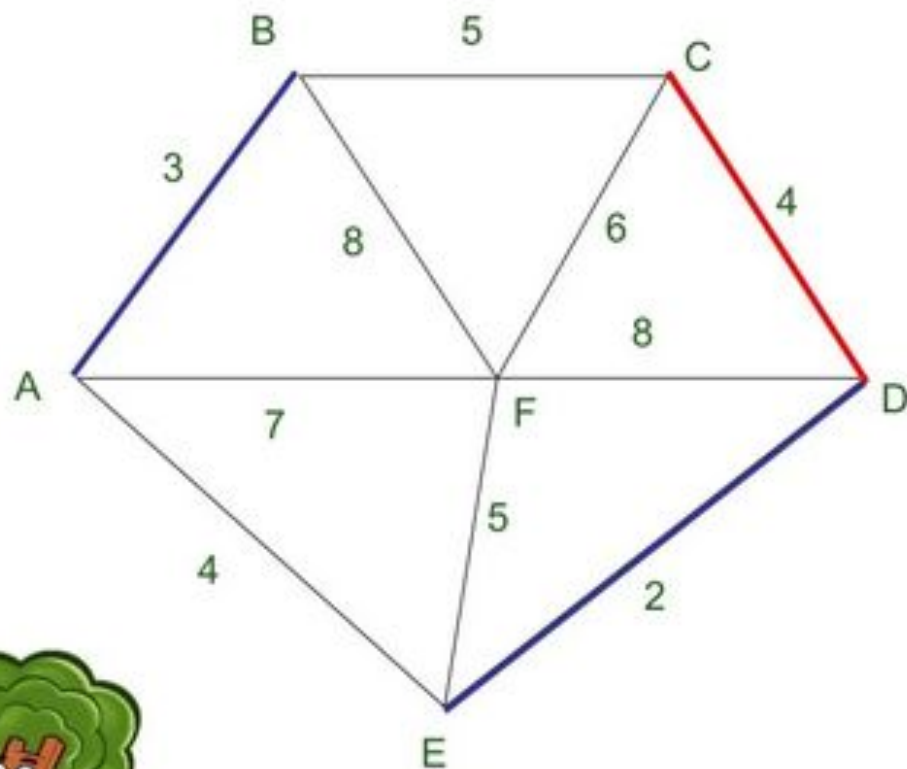
Select the next  
shortest  
edge which does not  
create a cycle

ED	2
AB	3

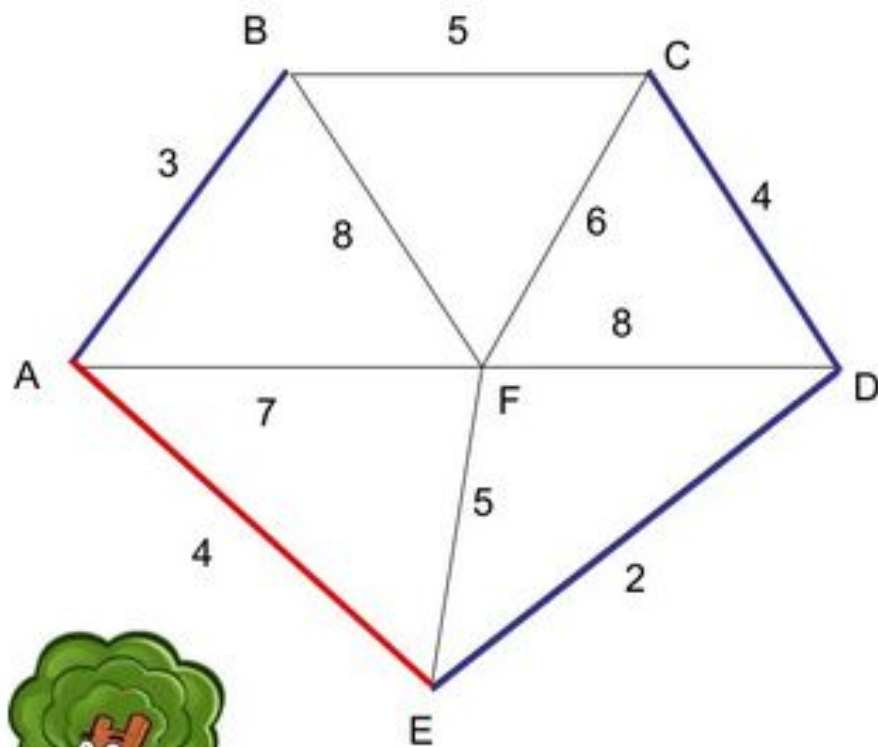
## Kruskal's Algorithm

Select the next  
shortest  
edge which does not  
create a cycle

ED 2  
AB 3  
CD 4 (or AE 4)



# Kruskal's Algorithm



Select the next  
shortest edge which  
does not create a cycle

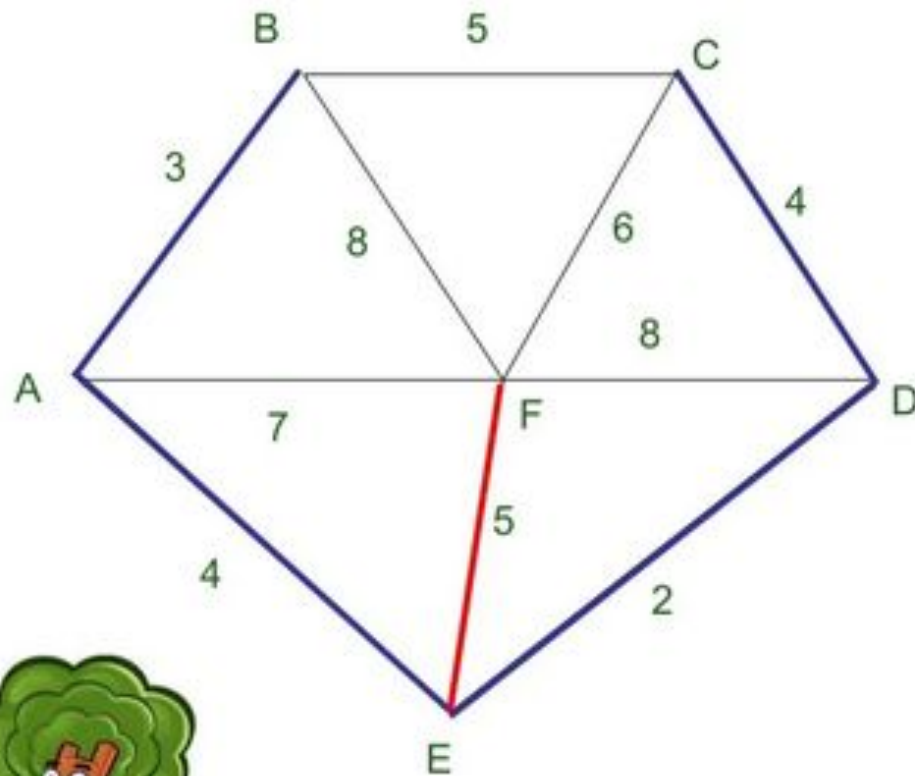
E

A

C



# Kruskal's Algorithm

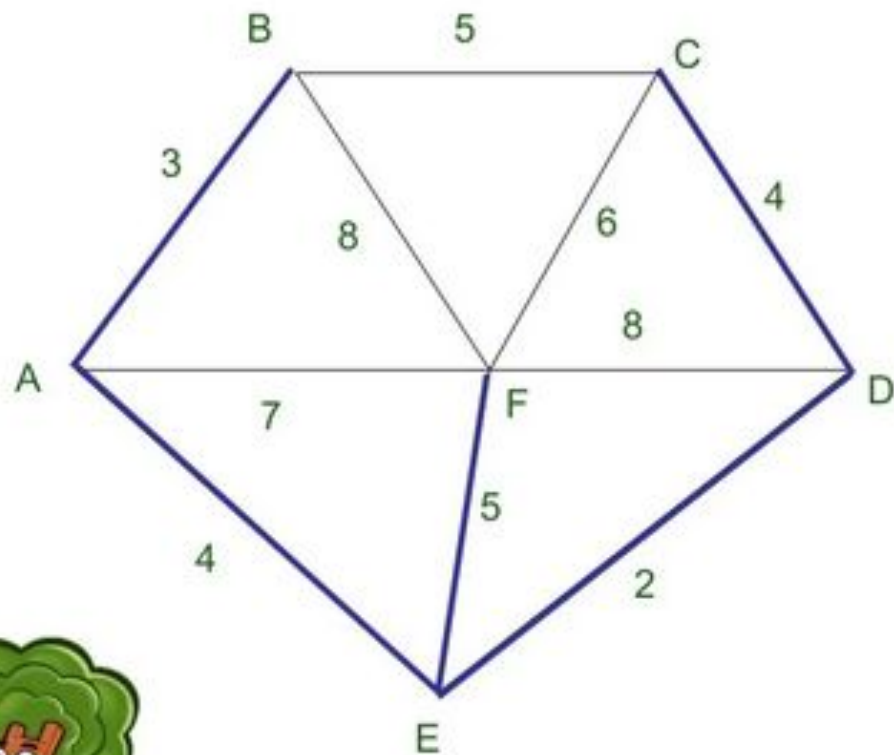


Select the next shortest edge which does not create a cycle

ED	2
AB	3
CD	4
AE	4
BC	5 - forms a cycle
EF	5



# Kruskal's Algorithm



All vertices have been connected.


The solution is

ED	2
AB	3
CD	4
AE	4
EF	5


Total weight of tree:  
18



# Algorithm



```
function Kruskal ( $G=(N,A)$ : graph ; length :  $A \rightarrow \mathbb{R}^+$ ):set of edges
{initialisation}
sort A by increasing length
 $N \leftarrow$  the number of nodes in N
 $T \leftarrow \emptyset$  {will contain the edges of the minimum spanning tree}
initialise n sets, each containing the different element of N
{greedy loop}
repeat
     $e \leftarrow \{u, v\} \leftarrow$  shortest edge not yet considered
     $u_{comp} \leftarrow \text{find}(u)$ 
     $v_{comp} \leftarrow \text{find}(v)$ 
    if  $u_{comp} \neq v_{comp}$  then
        merge( $u_{comp}$  ,  $v_{comp}$ )
         $T \leftarrow T \cup \{e\}$ 
until T contains n-1 edges
return T
```



# Round Robin Algorithm

## **Round Robin Algorithm**

Round Robin algorithm, which provides even better performance when the number of edges is low. The algorithm is similar to Kruskal's except that there is a priority queue of arcs associated with each partial tree, rather than one global priority queue of all unexamined arcs.



## Shortest Path

A path from a source vertex  $a$  to  $b$  is said to be shortest path if there is no other path from  $a$  to  $b$  with lower weights. There are many instances, to find the shortest path for traveling from one place to another. That is to find which route can reach as quick as possible or a route for which the traveling cost is minimum. Dijkstra's Algorithm is used to find shortest path.

## Greedy Algorithm

An optimization problem is one in which we want to find, not just a solution, but the *best* solution. A greedy algorithm sometimes works well for optimization problems. A greedy algorithm works in phases. At each phase: we take the best we can get right now, without regard for future consequences and we hope that by choosing a *local* optimum at each step, we will end up at a *global* optimum.

# Dijkstra's Algorithm

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

Let  $G$  be a directed graph with  $n$  vertices  $V_1, V_2, V_3, \dots, V_n$ . Suppose  $G = (V, E, W_e)$  is a weighted graph. i.e., each edge  $e$  in  $G$  is assigned a non-negative number, we call the weight or length of the edge  $e$ . Consider a starting vertex. Dijkstra's algorithm will find the weight or length to each vertex from the source vertex.

Set  $V = \{V_1, V_2, V_3, \dots, V_n\}$  contains the vertices and the edges  $E = \{e_1, e_2, \dots, e_m\}$  of the graph  $G$ .  $W(e)$  is the weight of an edge  $e$ , which contains the vertices  $V_1$  and  $V_2$ .  $Q$  is a set of vertices, which are not visited.  $m$  is the vertex in  $Q$  for which weight  $W(m)$  is minimum, i.e., minimum cost edge.  $S$  is a source vertex.

1. Input the source vertices and assigns it to  $S$

- (a) Set  $W(s) = 0$  and

- (b) Set  $W(v) = \_\_\_\_$  for all vertices  $V$  is not equal to  $S$

2. Set  $Q = V$  which is a set of vertices in the graph

3. Suppose  $m$  be a vertices in  $Q$  for which  $W(m)$  is minimum

4. Make the vertices  $m$  as visited and delete it from the set  $Q$

5. Find the vertices  $l$  which are incident with  $m$  and member of  $Q$  (That is the vertices which are not visited)

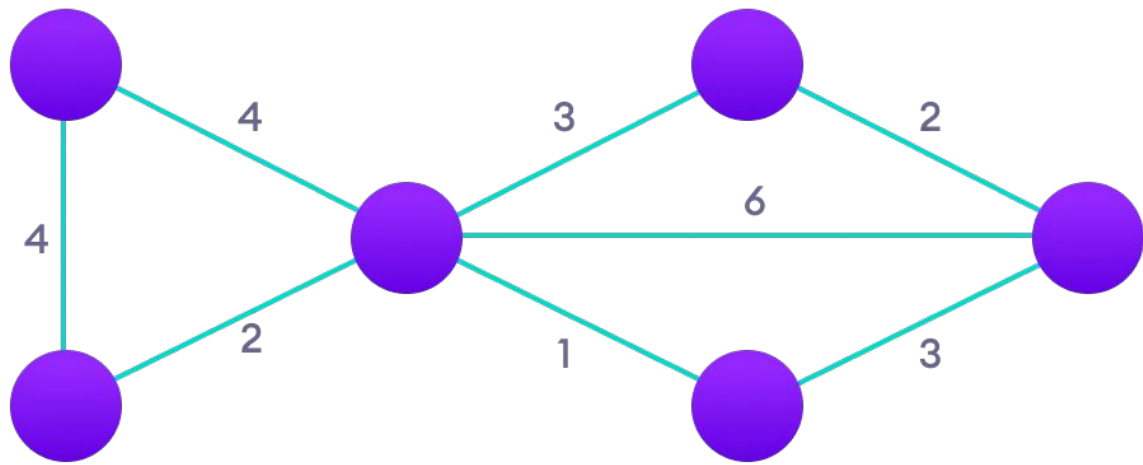
6. Update the weight of vertices  $I = \{i_1, i_2, \dots, i_k\}$  by

(a)  $W(i_1) = \min [W(i_1), W(m) + W(m, i_1)]$

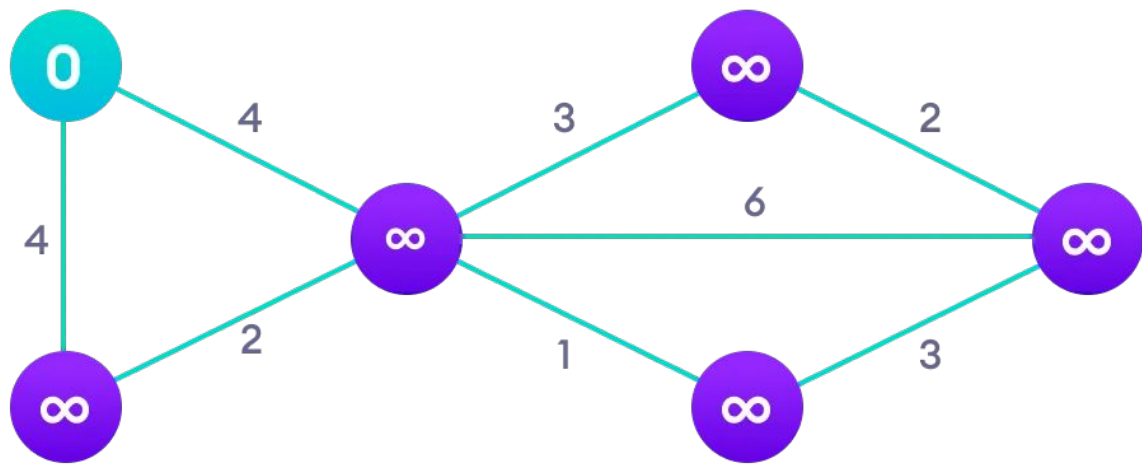
7. If any changes is made in  $W(v)$ , store the vertices to corresponding vertices  $i$ , using the array, for tracing the shortest path.

8. Repeat the process from step 3 to 7 until the set  $Q$  is empty

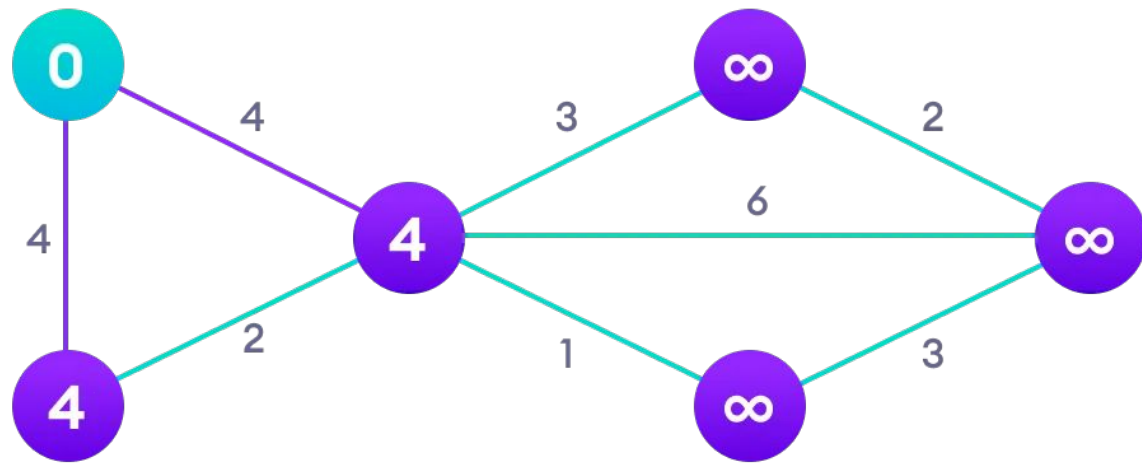
9. Exit



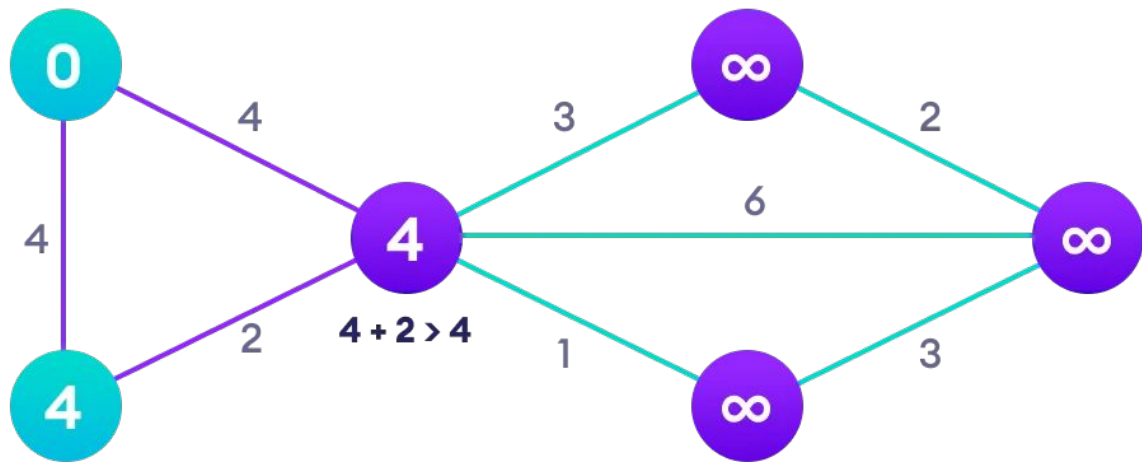
Step: 1



Step: 2

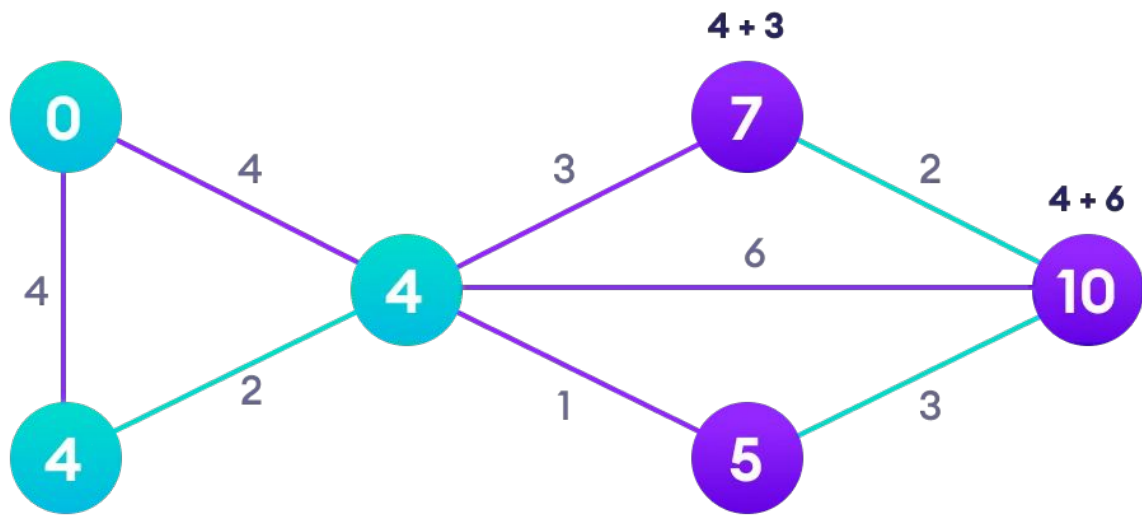


Step: 3

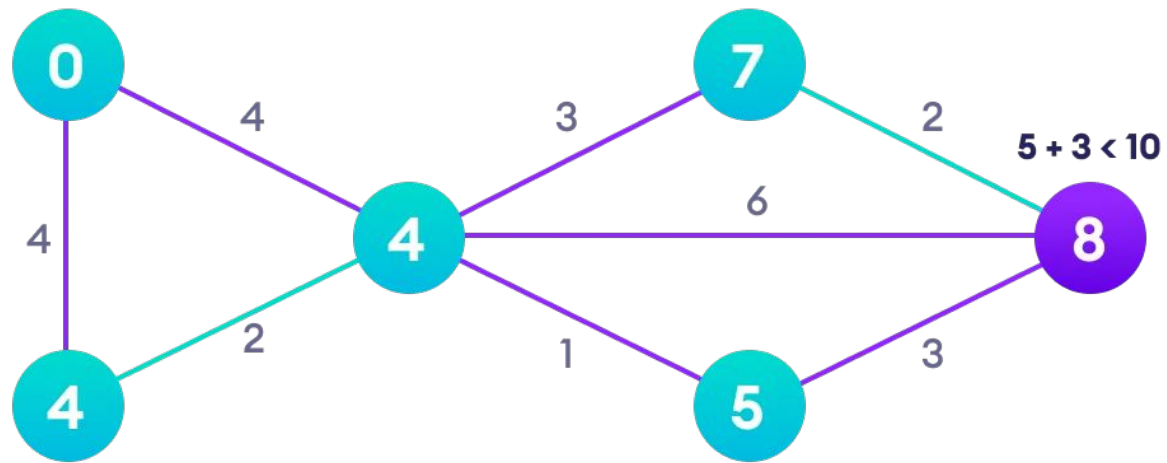


Step: 4

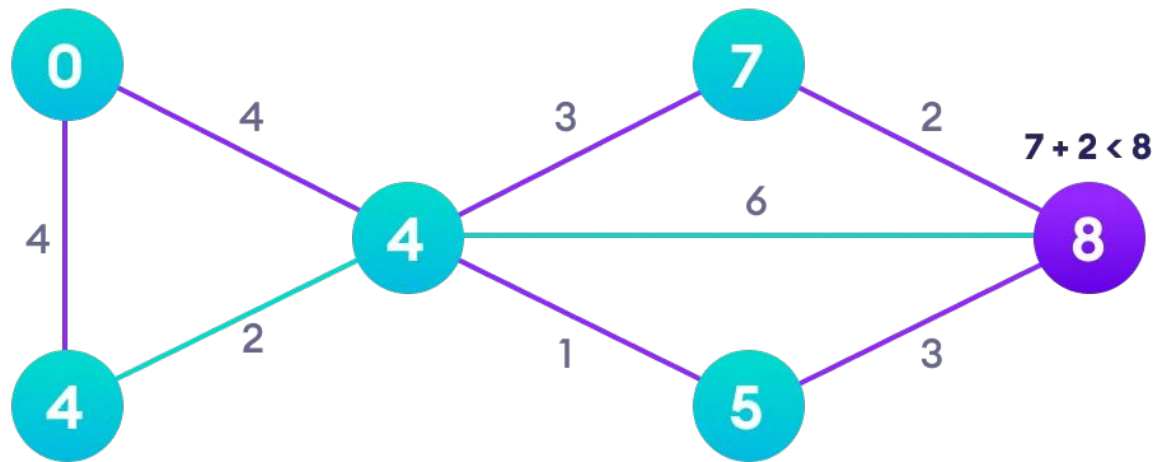




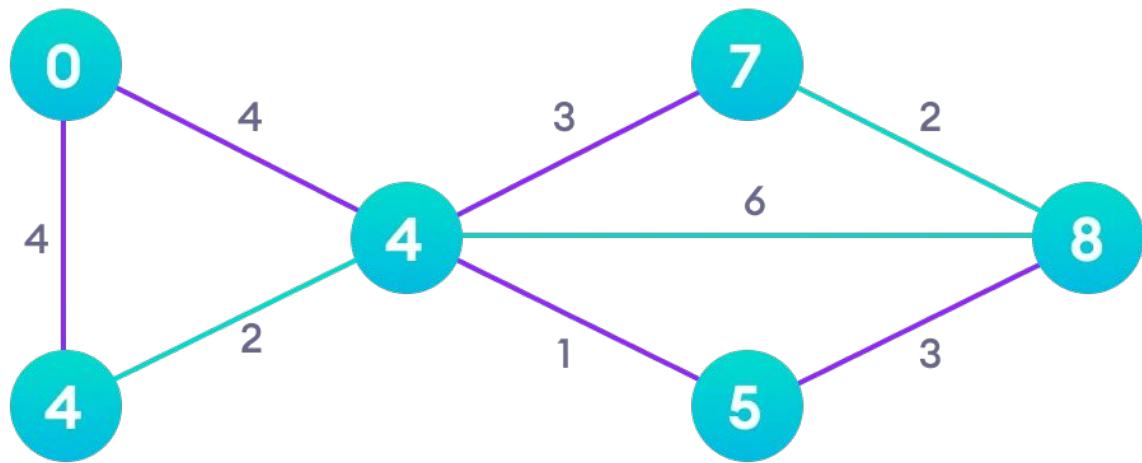
Step: 5



Step: 6



Step: 7

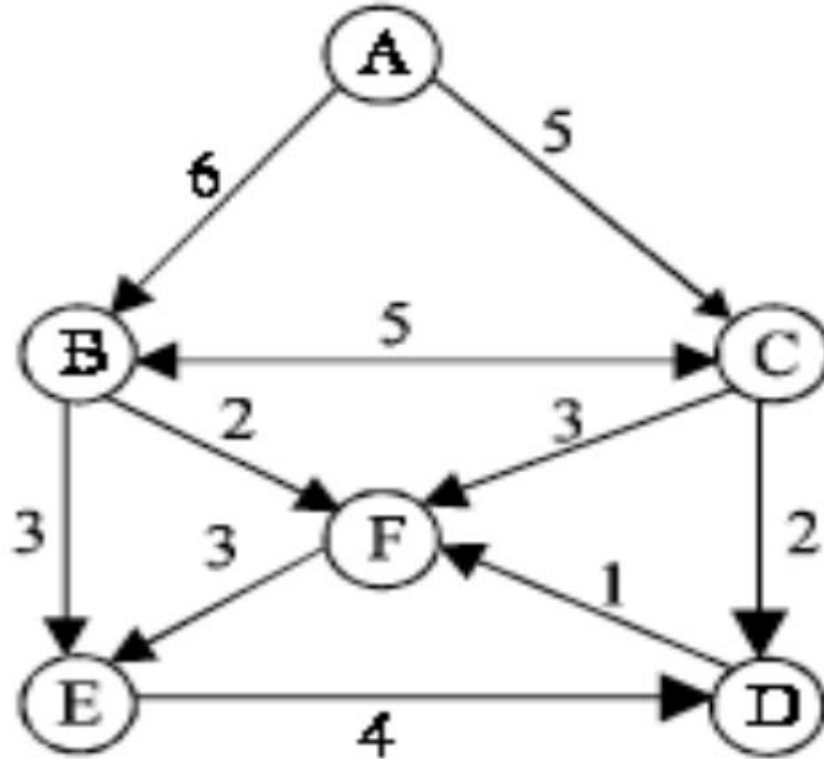


Step: 8

# Dijkstra's Pseudo Code

```
function dijkstra(G, S)
  for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
    If V != S, add V to Priority Queue Q
  distance[S] <- 0
  while Q IS NOT EMPTY
    U <- Extract MIN from Q
    for each unvisited neighbour V of U
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U
  return distance[], previous[]
```

Calculate the shortest path using Dijkstra Algorithm



Source vertices is = A  $W(A) = 0$

$V = \{A, B, C, D, E, F\} = Q$

V	A	B	C	D	E	F
W (V)	0					
Q	A	B	C	D	E	F

A	
B	
C	
D	
E	
F	

ITERATION 1

$m = A$

$W(A, A) = 0$  (Distance from A to A ) Now the  $Q = \{ B, C, D, E, F\}$ . Two edges are incident with m i.e.,  $I = \{ B, C\}$

$$\begin{aligned}
 W(B) &= \min [W(B), W(A) + W(A, B)] \\
 &= \min ( \_, 0 + 6) \\
 &= 6
 \end{aligned}$$

$$\begin{aligned}
 W(C) &= \min [W(C), W(A) + W(A, C)] \\
 &= \min ( -, 0 + 5) = 5
 \end{aligned}$$

V	A	B	C	D	E	F
W (V)	0	6	5			
Q		B	C	D	E	F

A	
B	A
C	A
D	
E	
F	

ITERATION 2:

$m = C$  (Because  $W(v)$  in minimum vertex and is also a member of  $Q$  )

Now the  $Q$  become  $(B, D, E, F)$

Two edge are incident with  $C = \{D, F\} = I$

$$W(D) = \min ( W(D), [ W(C) + W(C, D)] )$$

$$= \min ( \_, [5+2] ) = 7$$

$$W(F) = \min ( W(F), [ W(C) + W (C, F)] )$$

$$= \min ( \_, [5+3] ) = 8$$



<b>V</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
<b>W(V)</b>	<b>0</b>	<b>6</b>	<b>5</b>	<b>7</b>		<b>8</b>
<b>Q</b>		<b>B</b>		<b>D</b>	<b>E</b>	

### ITERATION 3:

$m = B$  (Because  $w(V)$  in minimum in vertices  $B$  and is also a member of  $Q$ )

Now the  $Q$  become  $(D, E, F)$

Three edge are incident with  $B = \{C, E, F\}$

Since  $C$  is not a member of  $Q$  so  $I = \{E, F\}$

$$W(E) = \min(\_, 6 + 3) = 9$$

$$W(F) = \min(8, 6 + 2) = 8$$

V	A	B	C	D	E	F
W(V)	0	6	5	7	9	8
Q				D	E	

ITERATION 4:

$m = D$

$Q = \{E, F\}$

Incident vertices of  $D = \{F\} = I$

$W(F) = \min (W(F), [W(D) + W(D,F)])$

$W(F) = \min (8, 7 + 1) = 8$

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8
Q					E	F

ITERATION 5:

$s = F$

$Q = \{ F \}$

Incident vertices of  $F = \{ E \}$

$W(E) = \min (W(F) , [W(E) + W(F,E)])$

$W(E) = \min (9 , 9 + 3) = 9$

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8
Q					E	

now E is the only chain, hence we stop the iteration and the final table

V	A	B	C	D	E	F
W (V)	0	6	5	7	9	8

If the source vertex is A and the destination vertex is D then the weight is 7 and the shortest path can be traced from table at the right side as follows. Start finding the shortest path from destination vertex to its shortest vertex. For example we want to find the shortest path from A to D. Then find the shortest vertex from D, which is C. Check the shortest vertex, is equal to source vertex. Otherwise assign the shortest vertex as new destination vertex to find its shortest vertex as new destination vertex to find its shortest vertex. This process continued until we reach to source vertex from destination vertex.

D→C

C→A

A, C, D is the shortest path