Data Structure and Algorithm chapter 09

Presented by: Er. Aruna Chhatkuli

Nepal College of Information Technology, Balkumari, Lalitpur

Graph

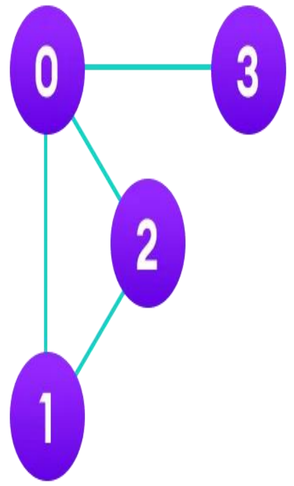
A graph data structure is a collection of nodes that have data and are connected to other nodes.

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}

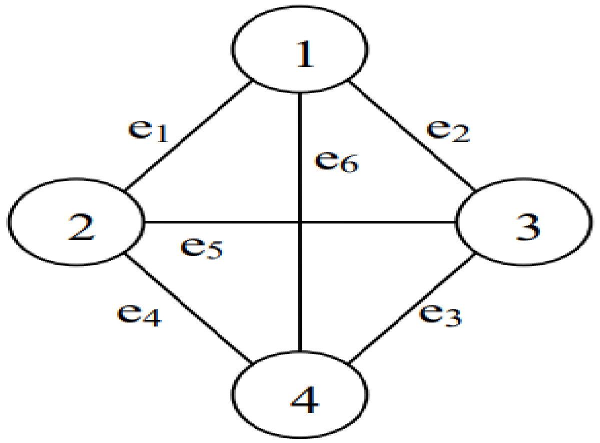


Graph

- Definition: A graph G = (V, E) consists of a finite set of non-empty set of vertices V and set of edges E.
- V = {v1, v2, …, vn}
- E = {e1, e2, …en}
- Each edge e is a pair (u, v) where u, v ϵ V. The edge is also called arc.
- Example of graphs given below:

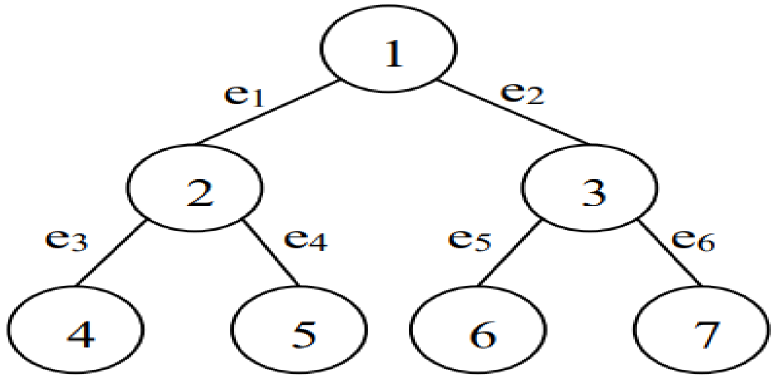
Graph

The vertices are represented by points or circles and the edges are line segments connecting the vertices. If the graph is directed, then the line segments have arrow heads indicating the direction.



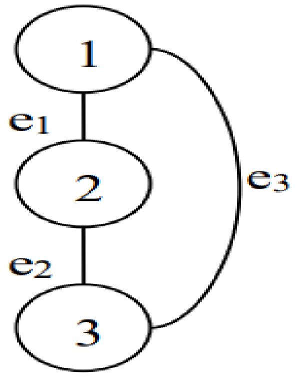
G_1

$V(G_1) = \{1, 2, 3, 4\}$
$E(G_1) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 = (1, 2)$
$e_2 = (1, 3)$
$e_3 = (3, 4)$
$e_4 = (2, 4)$
$e_5 = (2, 3)$
$e_6 = (1, 4)$



G_2

$V(G2) = \{1,2,3,4,5,6,7\}$
$E(G2) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$
$e_1 = (1, 2)$
$e_2 = (1, 3)$
$e_3 = (2, 4)$
$e_4 = (2, 5)$
$e_5 = (3, 6)$
$e_6 = (3, 7)$



G_3

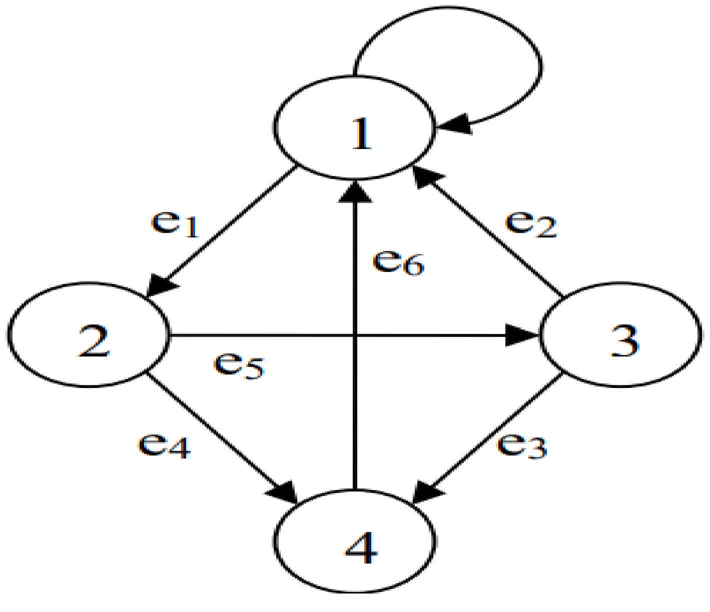
$V(G_3) = \{1, 2, 3\}$
$E(G_3) = \{e_1, e_2, e_3\}$
$e_1 = (1, 2)$
$e_2 = (2, 3)$
$e_3 = (1, 3)$

Undirected Graph:

- If the directions are not marked for any edge, the graph is called undirected graph. The graphs G1, G2, G3 are undirected graphs.
- In an undirected graph, we say that an edge e = (u, v) is incident on u and v (u and v are connected).
- Undirected graph don’t have self loops.
- Incidence is a symmetric relation i.e. if e = (u, v) Then u is a neighbor of v and vice versa.
- The degree of a vertex, d(v), is the total number of edges incident on it.
- The sum of the degrees of all the vertices in a graph equals twice the number of edges **if d(v) = 0, v is isolated. If d(v) = 1, v is pendent**.

Directed Graphs:

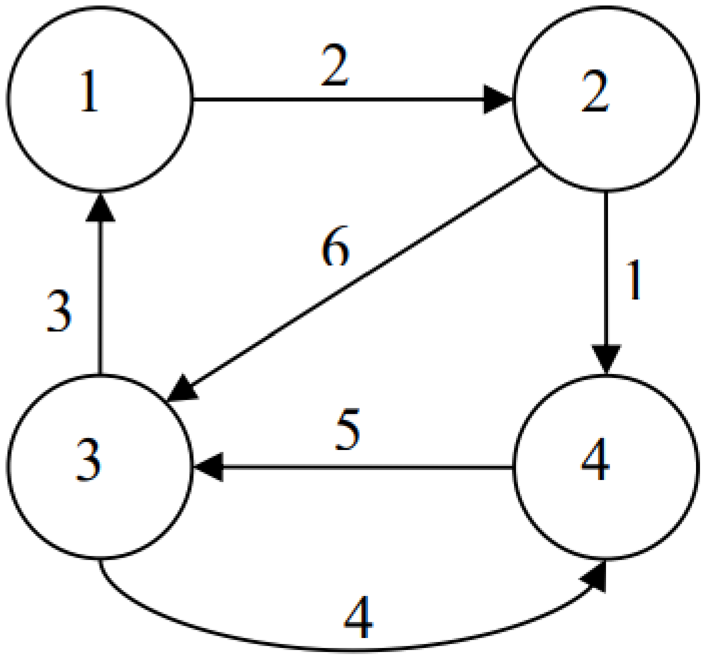
- If every edge (i, j), in E(G) of a graph G is marked by a direction from i to j, then the graph is called directed graph. It is often called diagraph.
- In edge e = (i, j), we say, e leaves i and enters j.
- In diagraphs, self loops are allowed.
- The indegree of a vertex v is the number of edge entering v.
- The outdegree of a vertex v is the number of edges leaving v.
- The sum of the indegrees of all the vertices in a graph equals to the sum of outdegree of all the vertices.



G_4

Weighted Graphs:

- A graph is said to be weighted graph if every edge in the graph is assigned some weight or value.
- The weight is a positive value that may represent the cost of moving along the edge, distance between the vertices etc.
- The two vertices with no edge (path) between them can be thought of having an edge (path) with weight infinite.



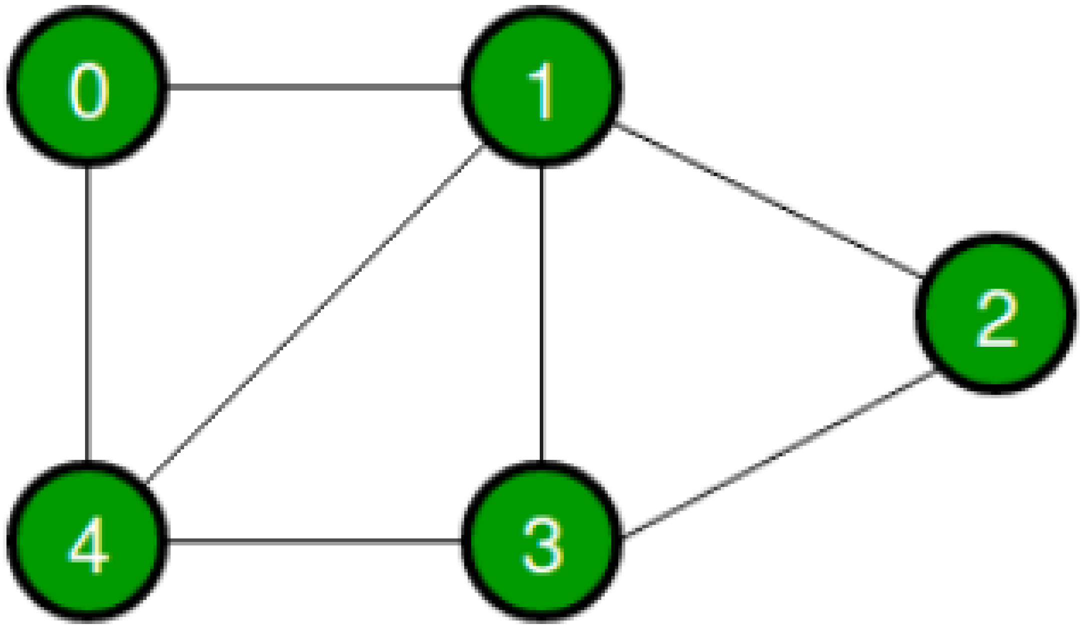
Representation of Graph

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

1. Adjacency Matrix

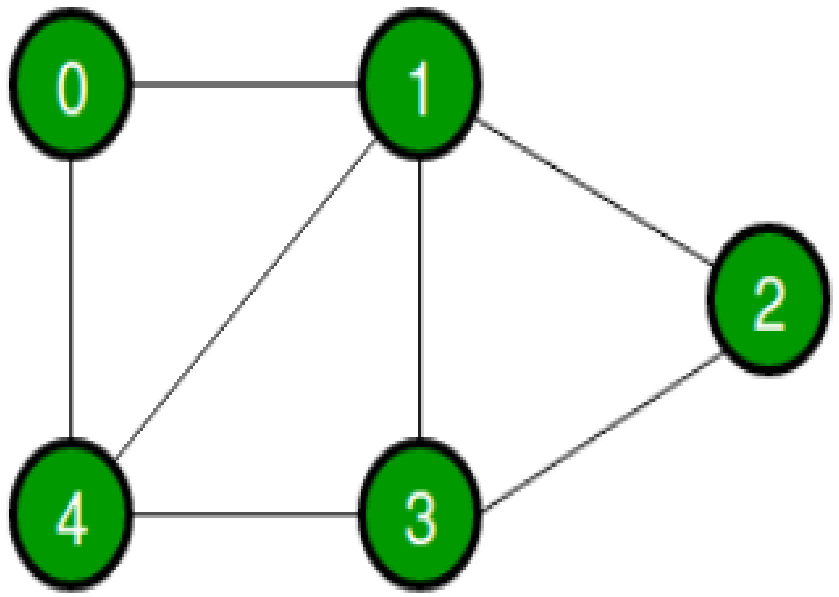
2. Adjacency List

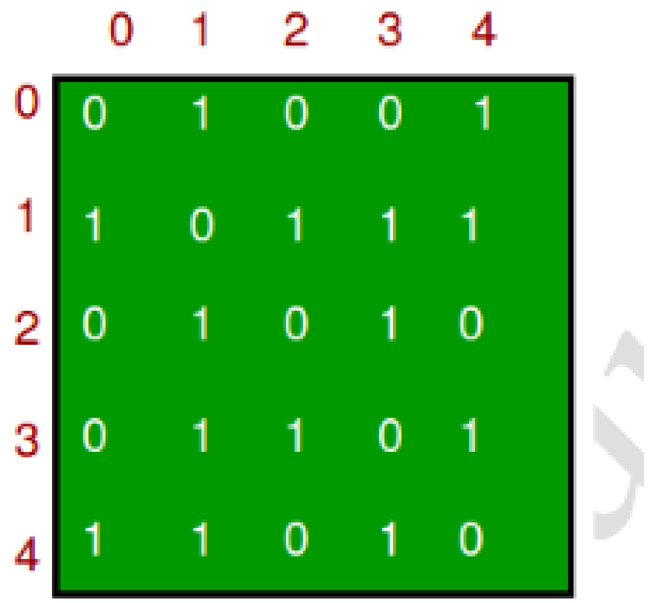
Let us consider an undirected graph



Representation of Graph: Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency Matrix 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.

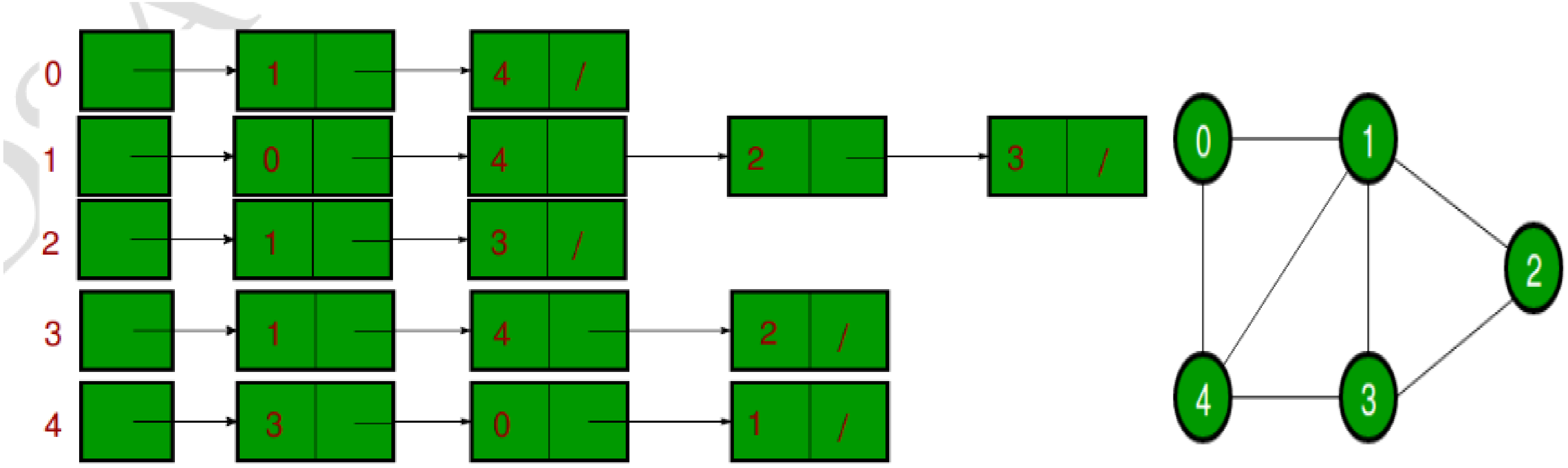




$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 0 & 1 & 0 \end{array}$$

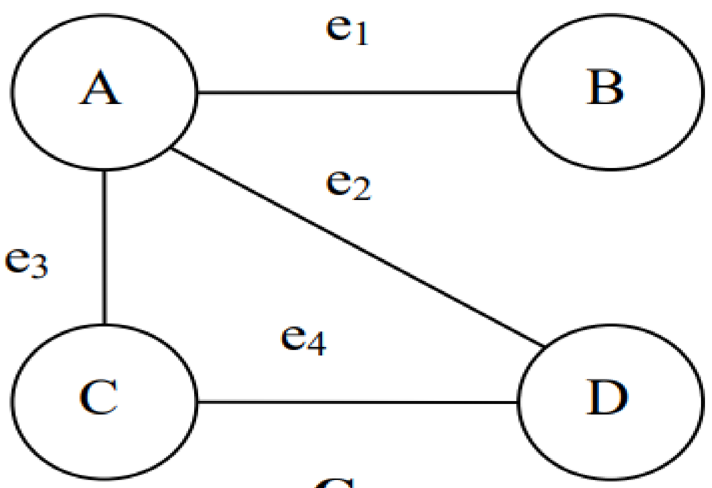
Representation of Graph: Adjacency List

- 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.



Path:

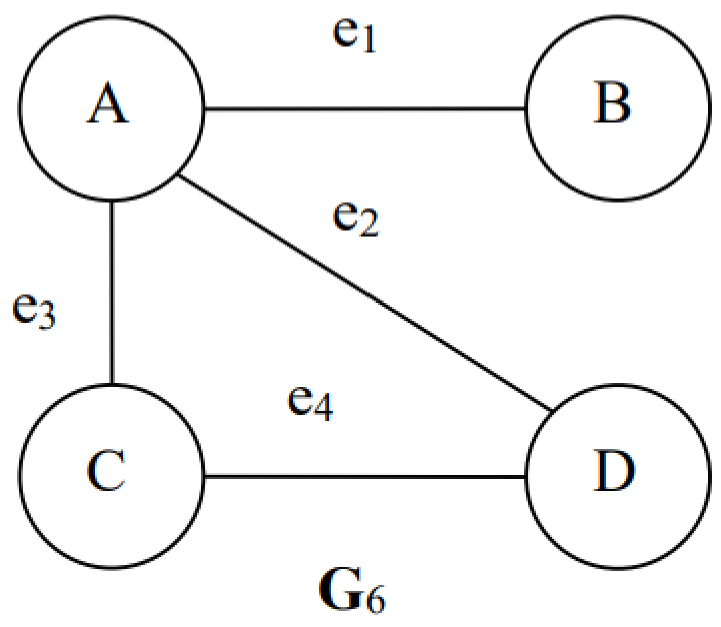
- A path is a sequence or distinct vertices, each adjacent to the next.
- For e.g. the sequence of vertices e1, e3, e4 (i.e (B,A), (A,C), (C,D)) of the above graph form a path from B to D.
- The length of the path is the number of edges in the path. So, the path from B to D has length equal to three.
- In a weighted graph, the cost of a path is the sum of the costs of its edges. Loops have path length of 1.



G_6

Cycle:

- A cycle is a path containing at least three vertices such that the last vertex on the path is adjacent to the first.
- In the above graph G6, A, C, D, A is a cycle.

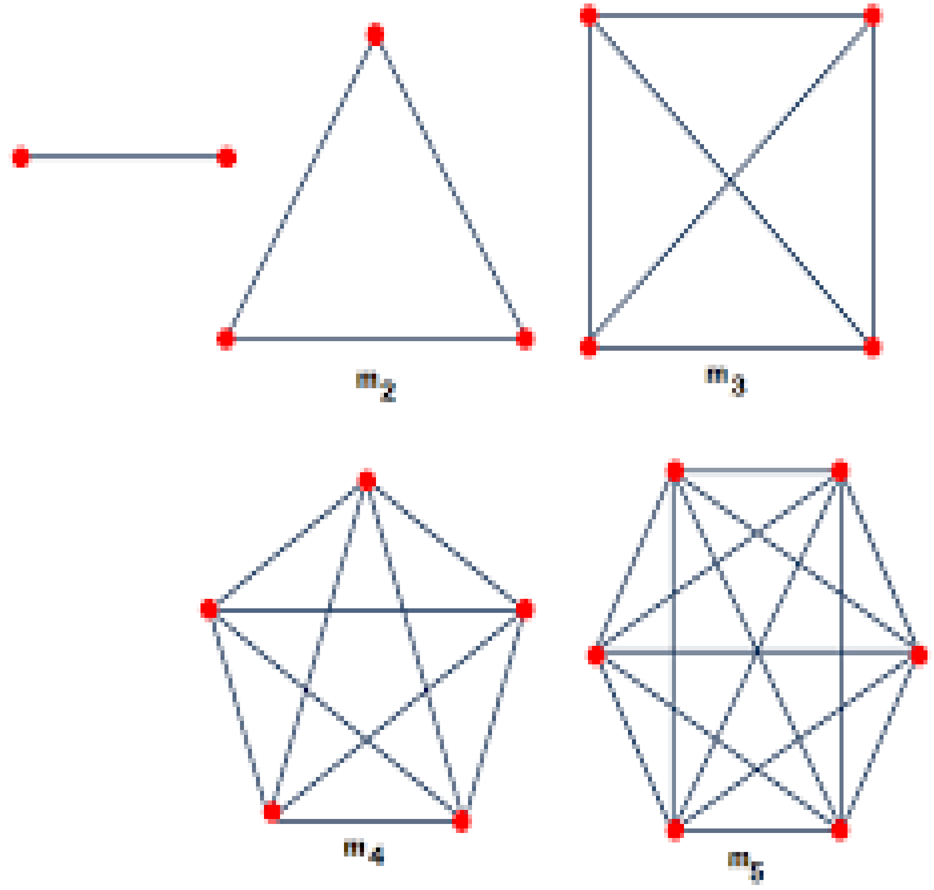


Complete graph

- A complete graph is a graph in which there is an edge between every pair of vertices.

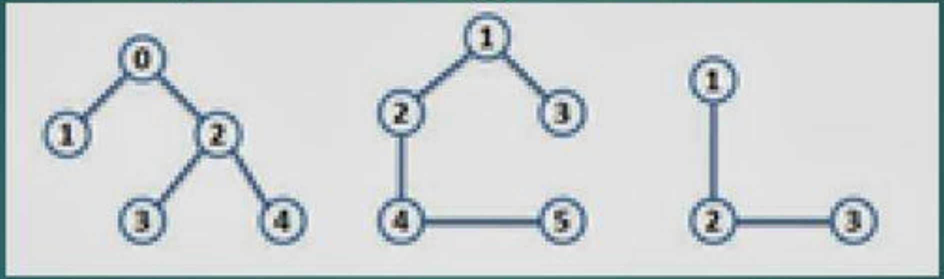
Trees and Graph

- A tree is a connected graph with no cycle.
- A tree has |E| = |V| - 1 edges. Since, it is connected, there is a path between any two vertices.



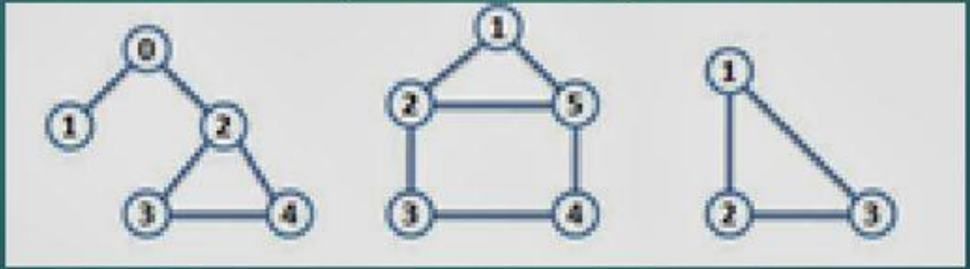
Spanning Tree – A graph that connects all vertices, WITHOUT creating/containing a cycle.

Spanning trees



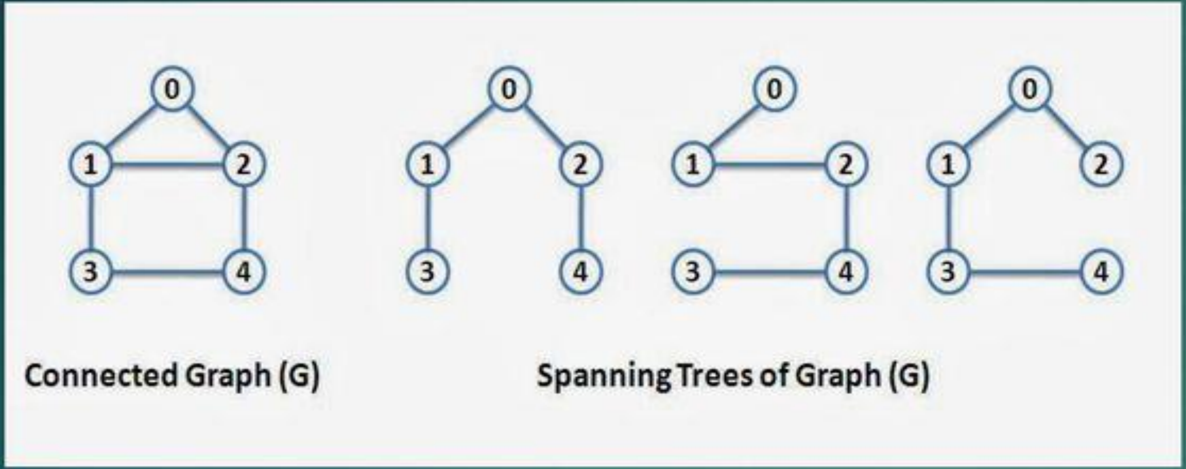
These graphs do NOT contain cycles

NOT Spanning trees



These graphs contain cycles

Creating a spanning tree from a connected graph

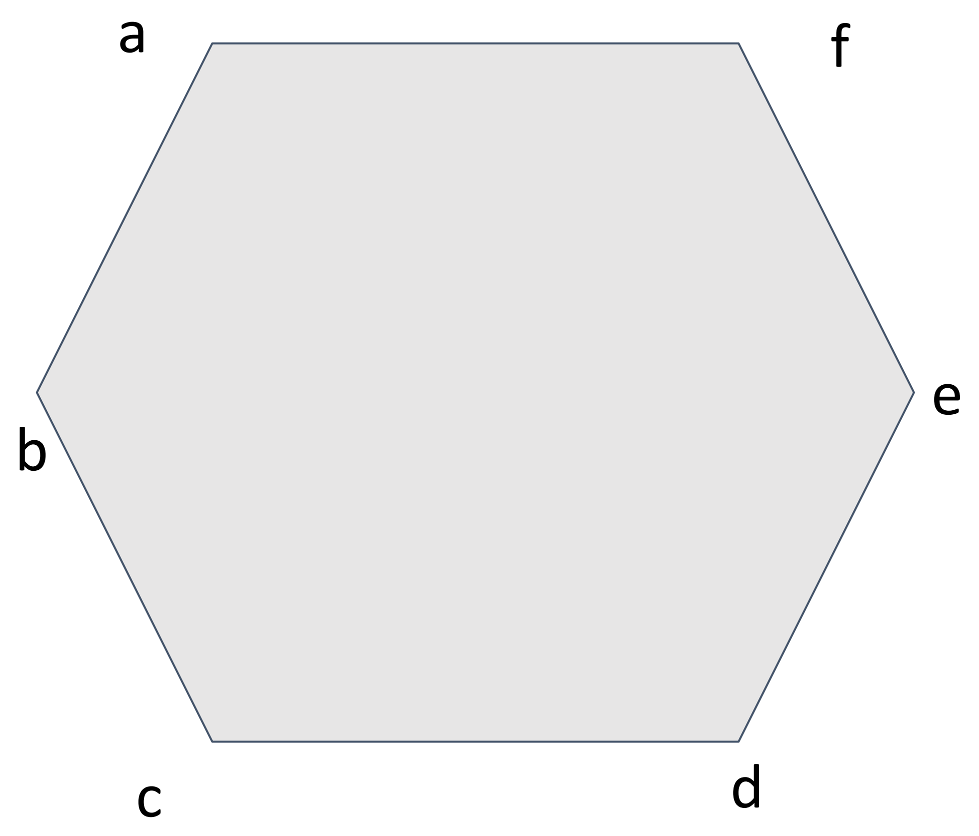


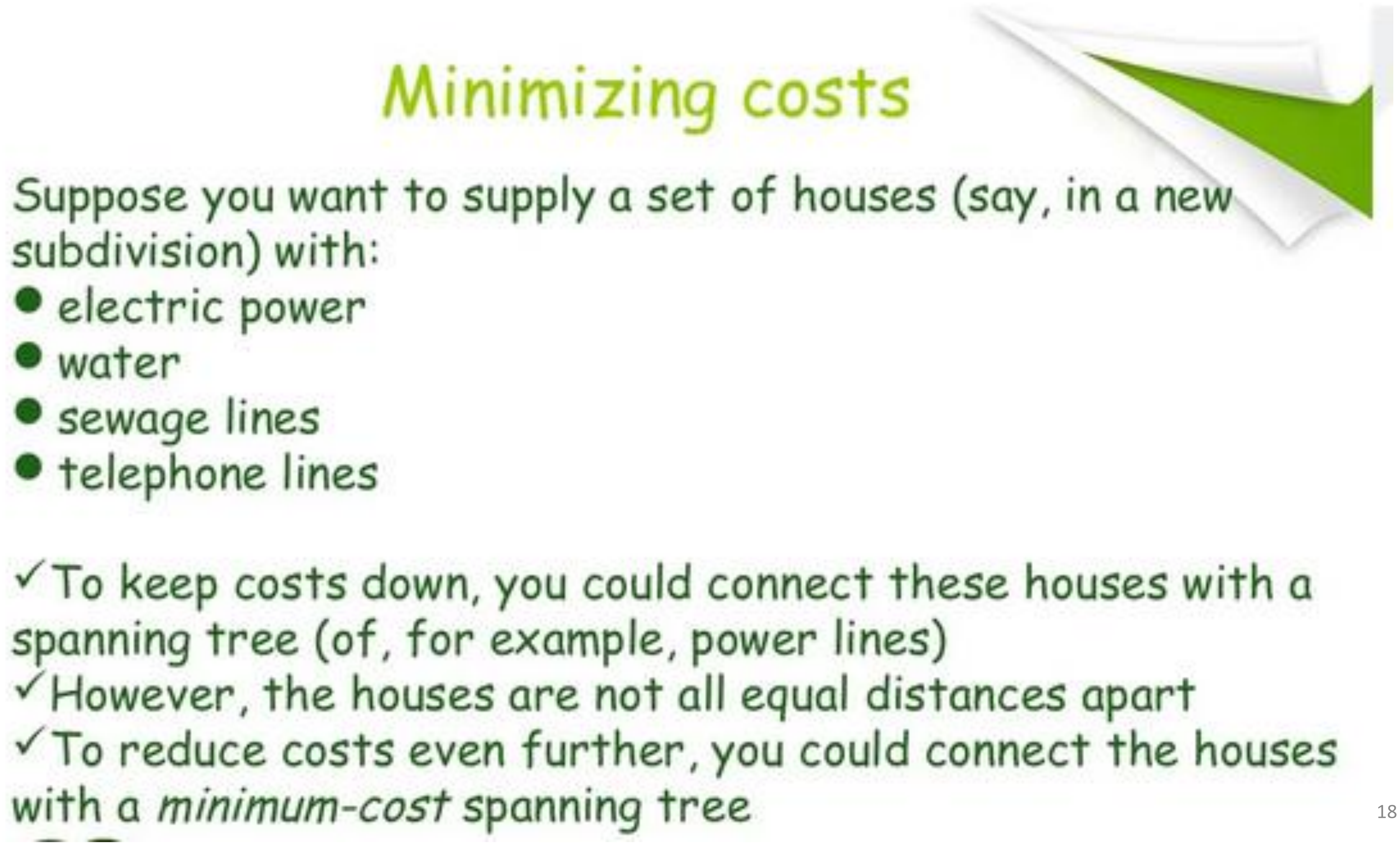
- Three possible spanning trees have been created from Graph (G).
- Notice how all of the vertices are connected in each spanning tree, but none of these graphs contain any of the original cycles.

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

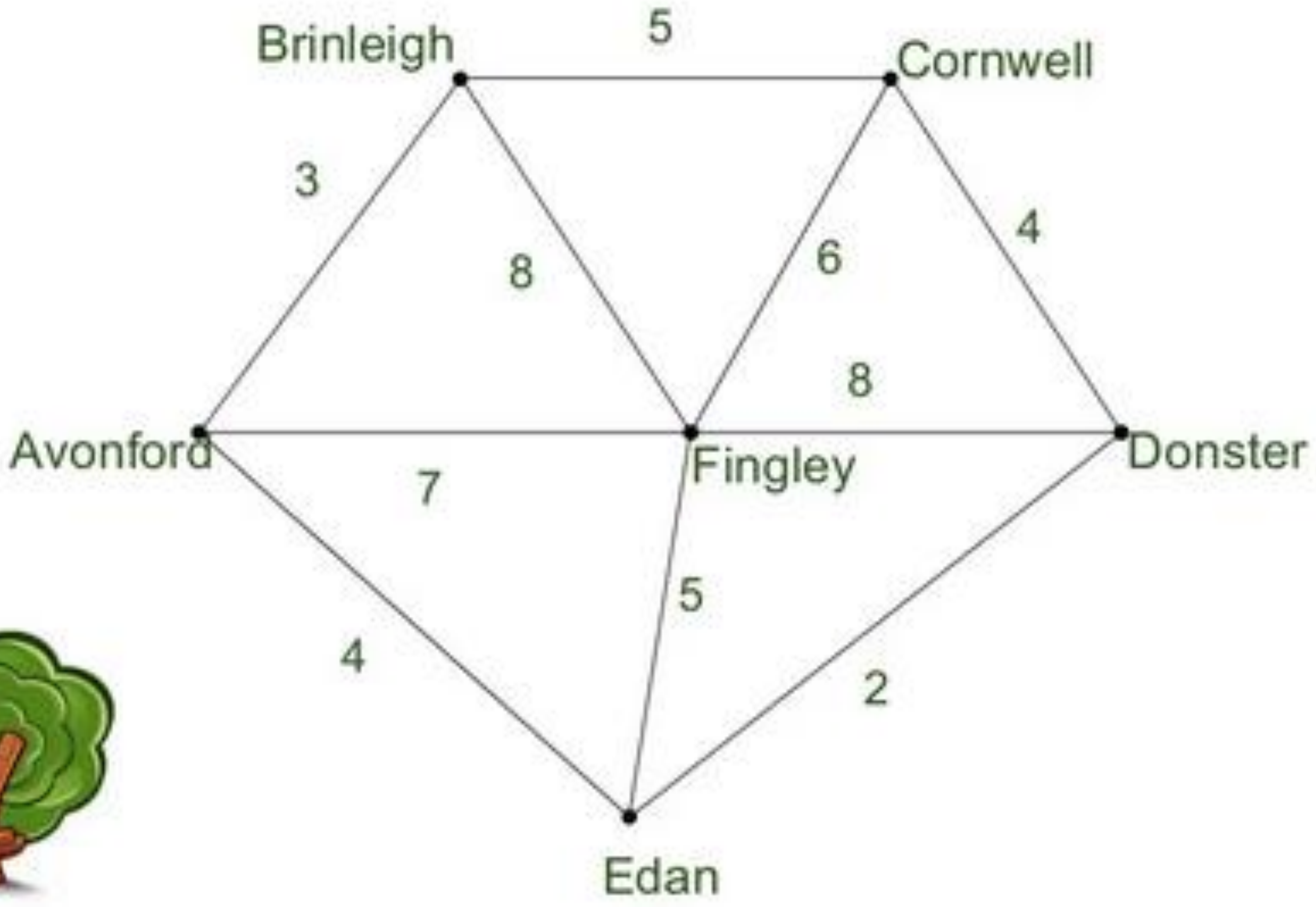
Suppose you want to supply a set of houses (say, in a new subdivision) with:

- 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

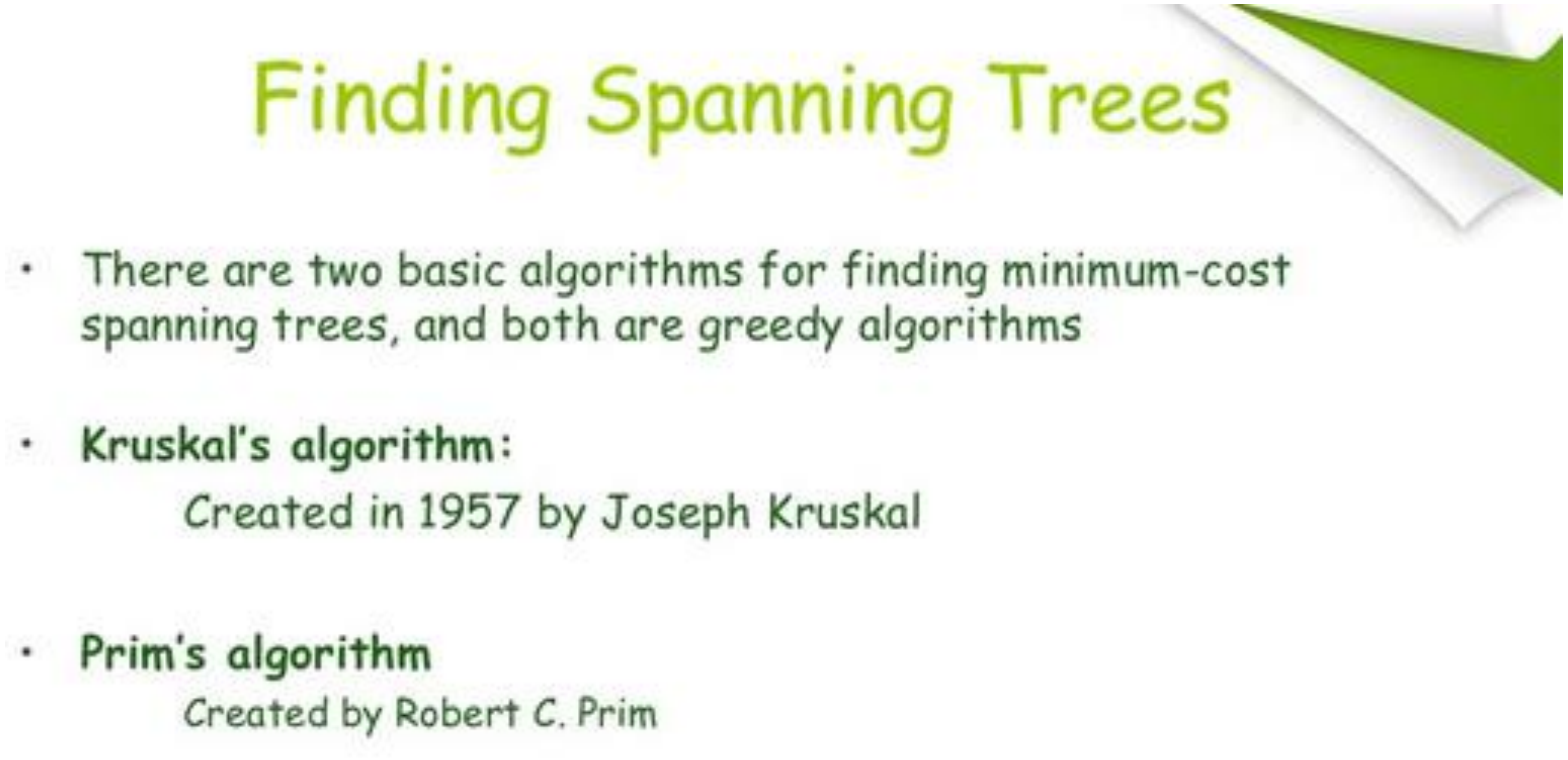
Example

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.



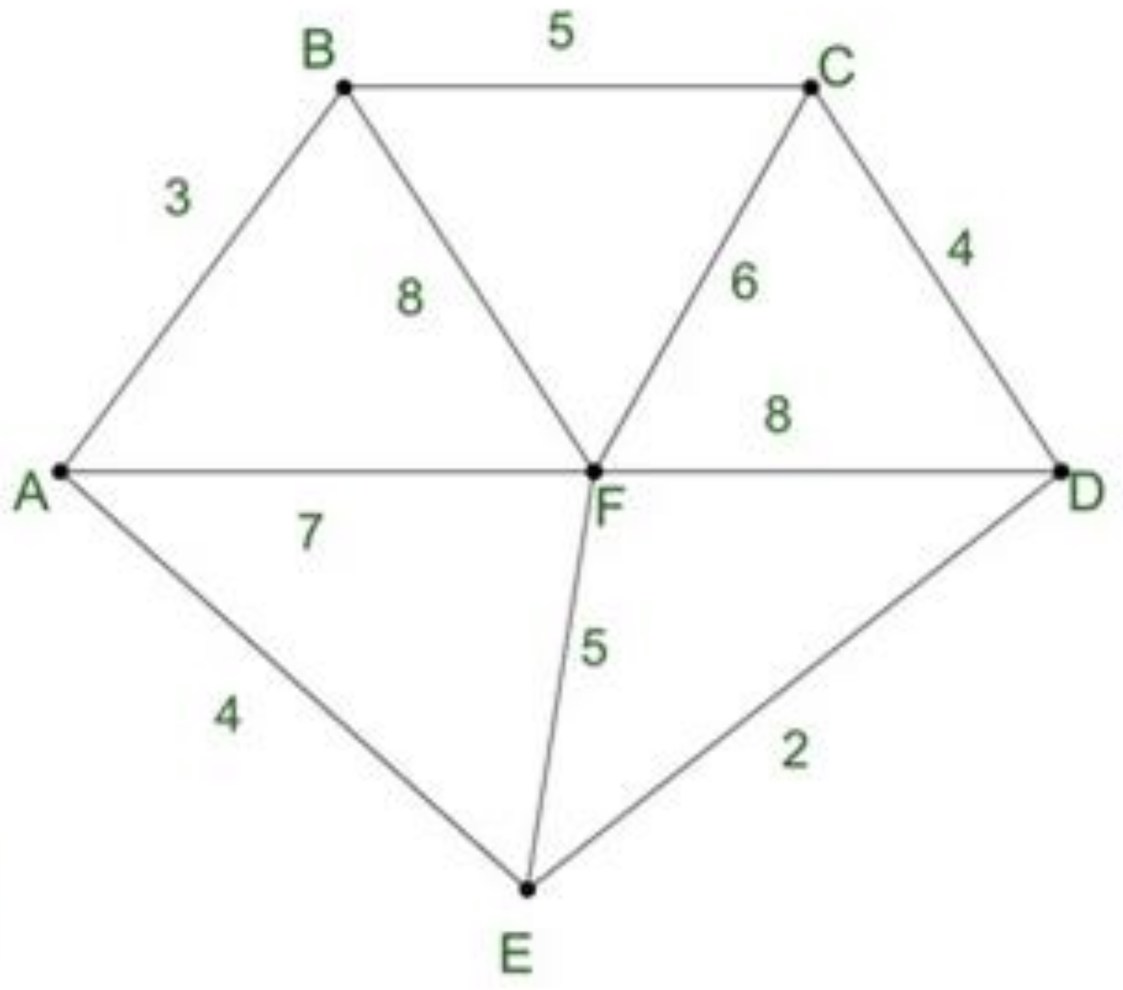
Finding Spanning Trees

- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms
- **Kruskal's algorithm:**

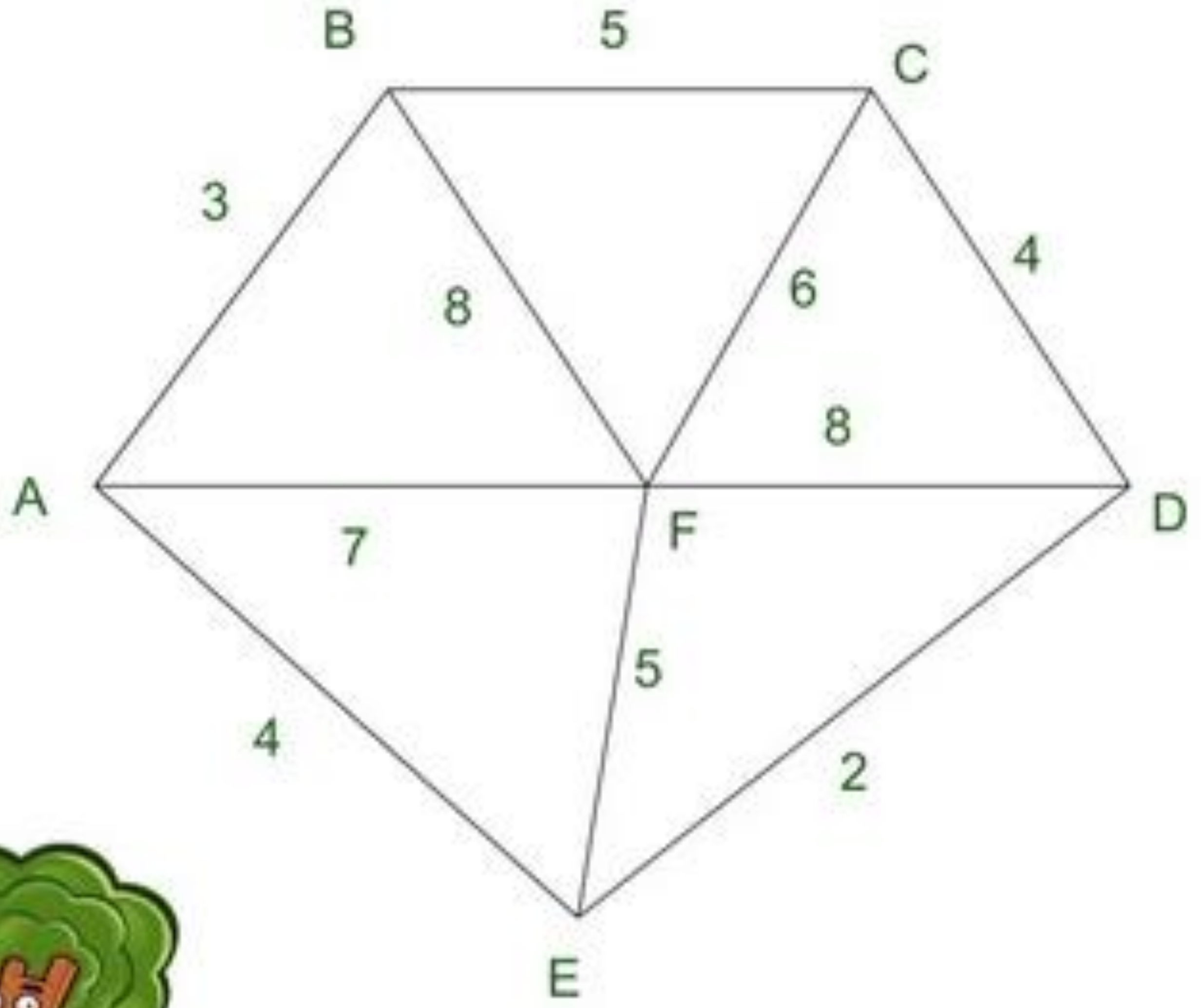
  Created in 1957 by Joseph Kruskal
- **Prim's algorithm**

  Created by Robert C. Prim

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



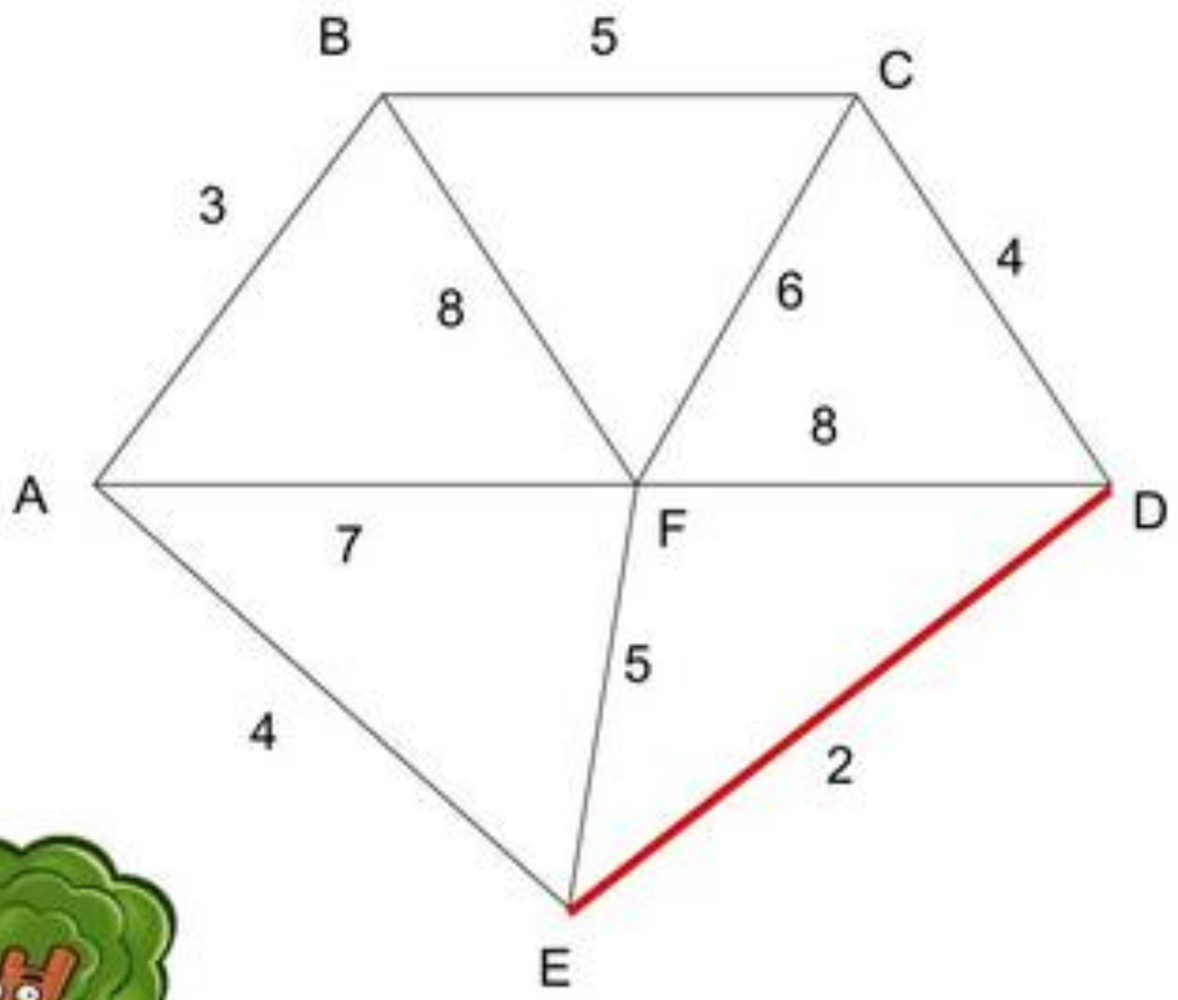
Kruskal's Algorithm



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

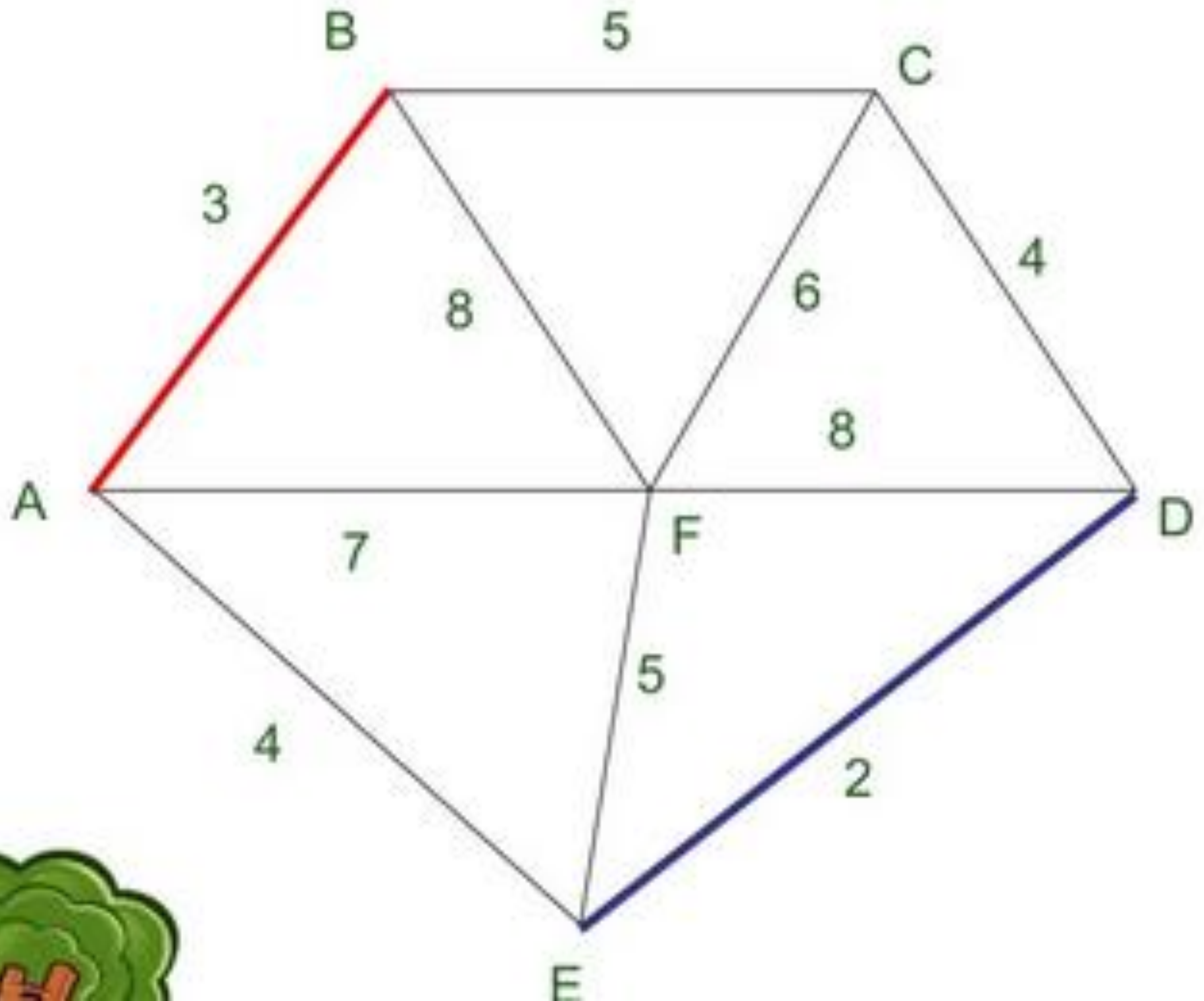
Kruskal's Algorithm



Select the shortest edge in the network

ED 2

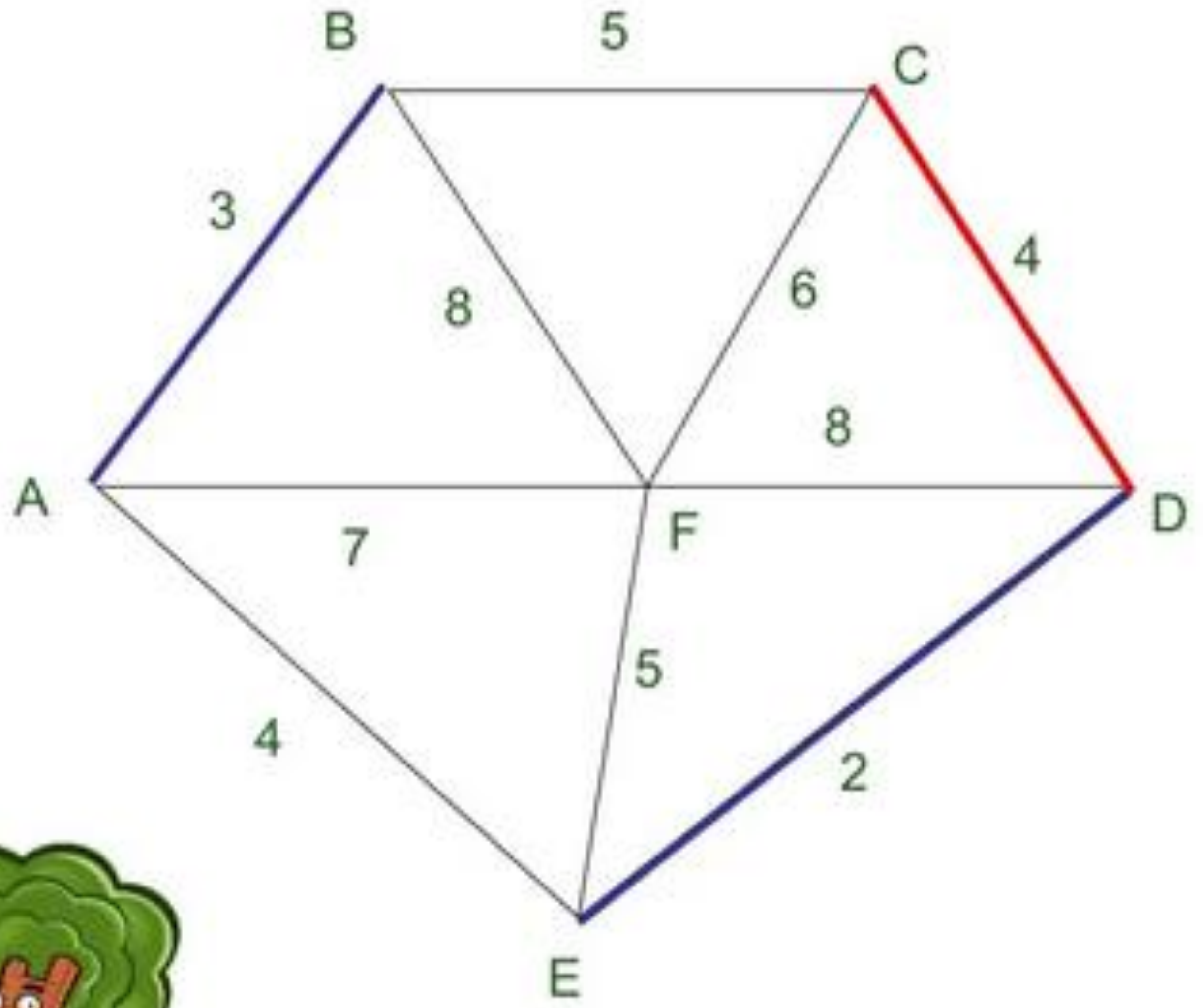
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED	2
AB	3

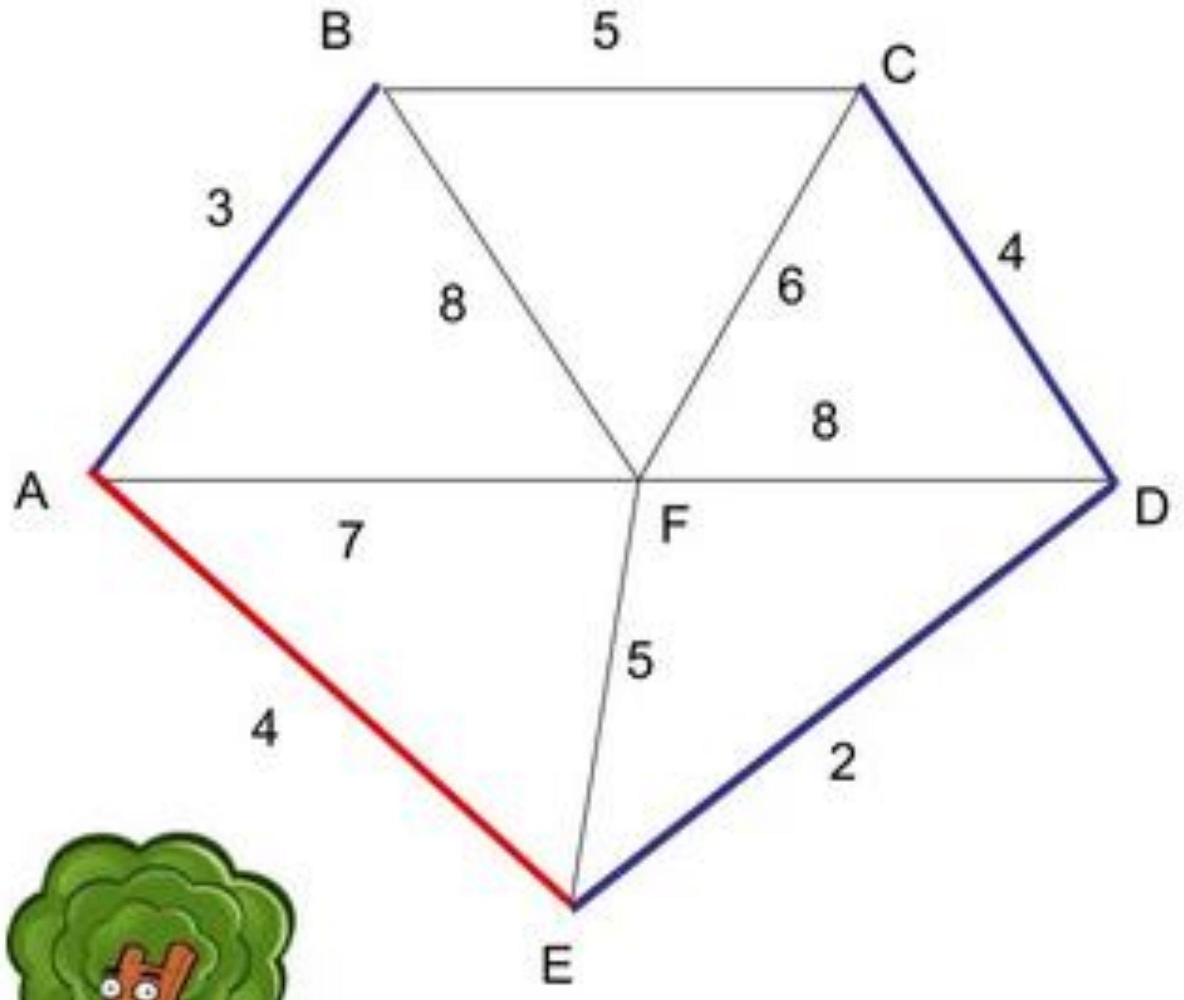
Kruskal's Algorithm



Select the next
shortest
edge which does not
create a cycle

ED 2
AB 3
CD 4 (or AE 4)

Kruskal's Algorithm



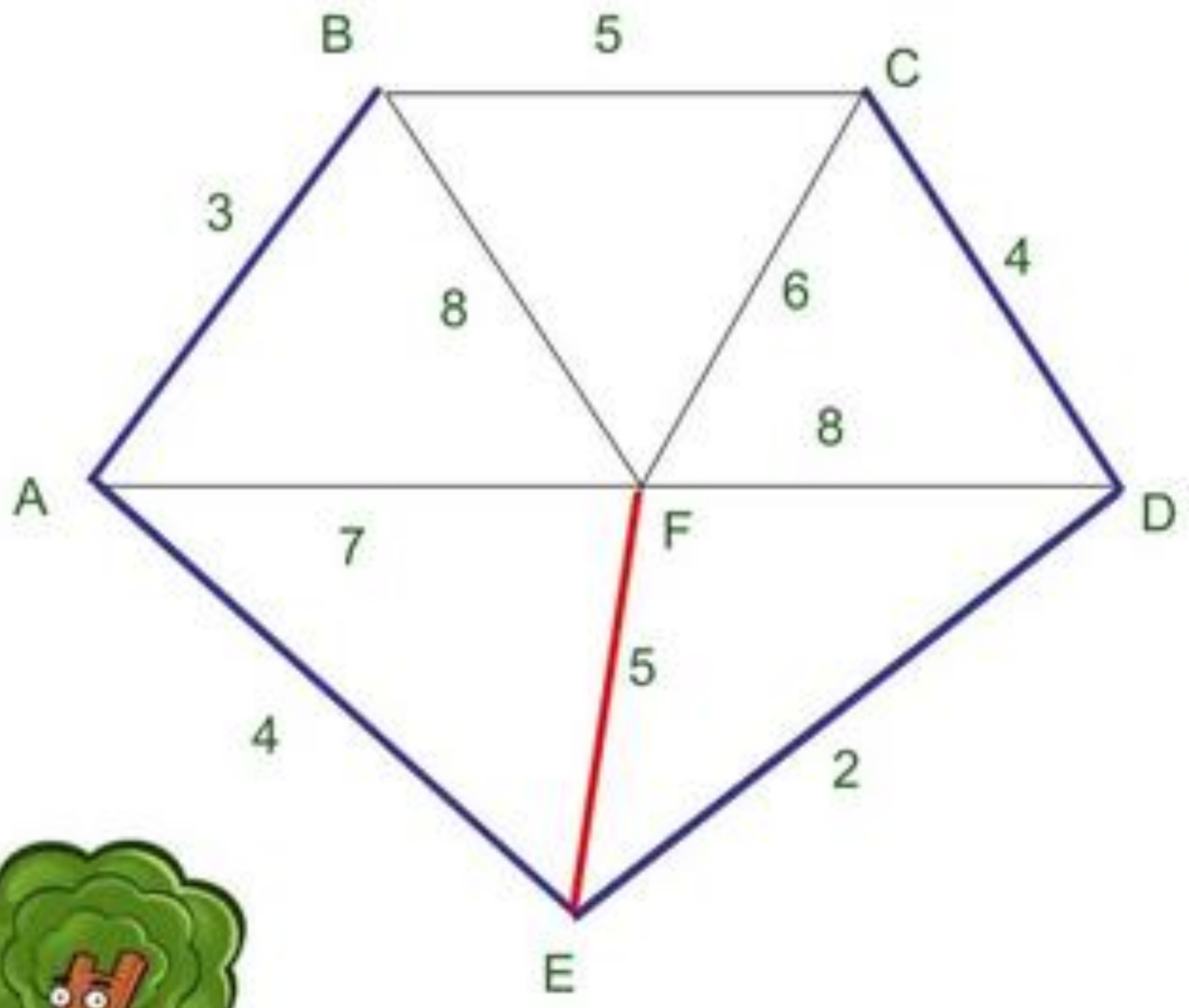
Select the next shortest edge which does not create a cycle

E

A

C

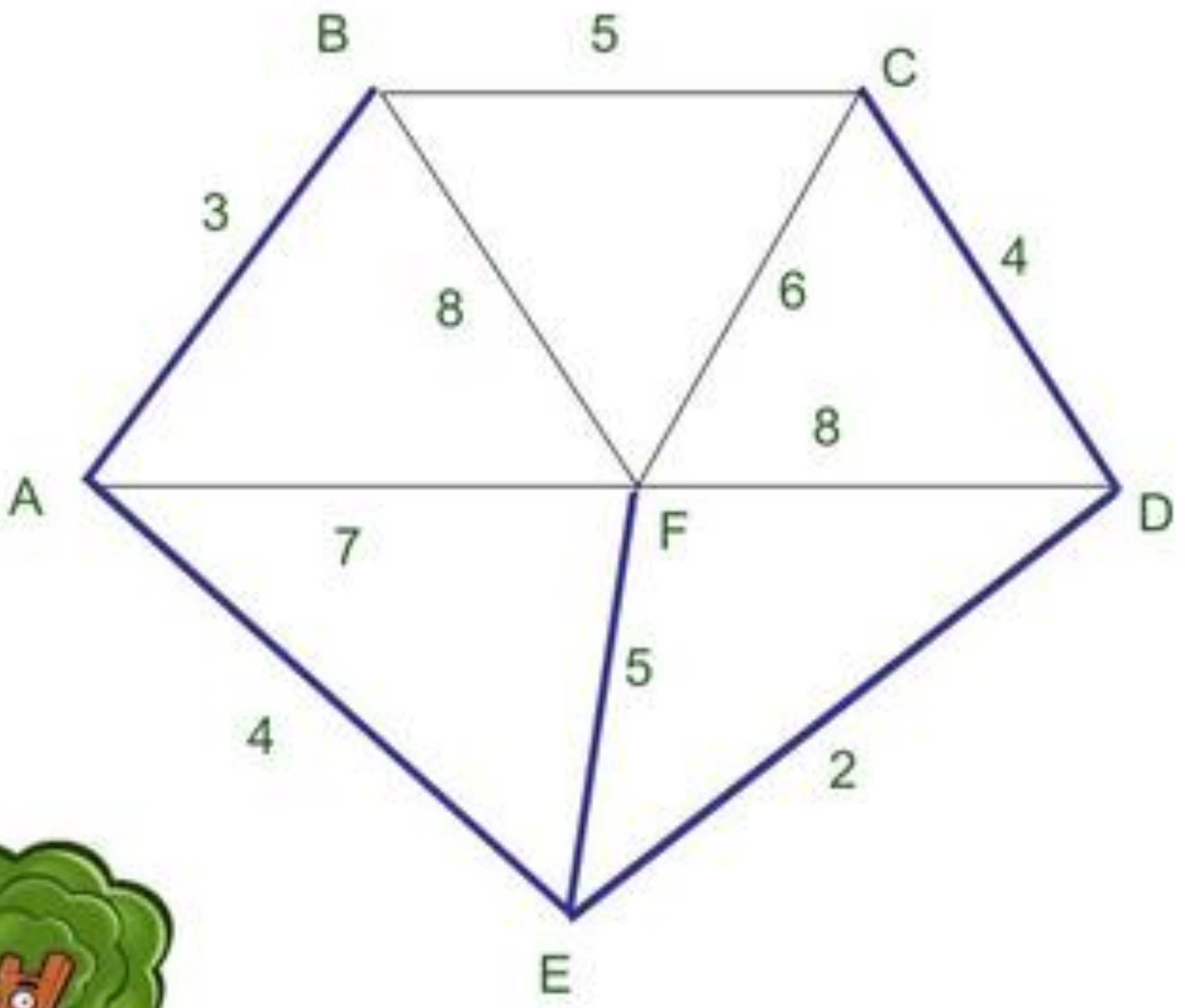
Kruskal's Algorithm



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

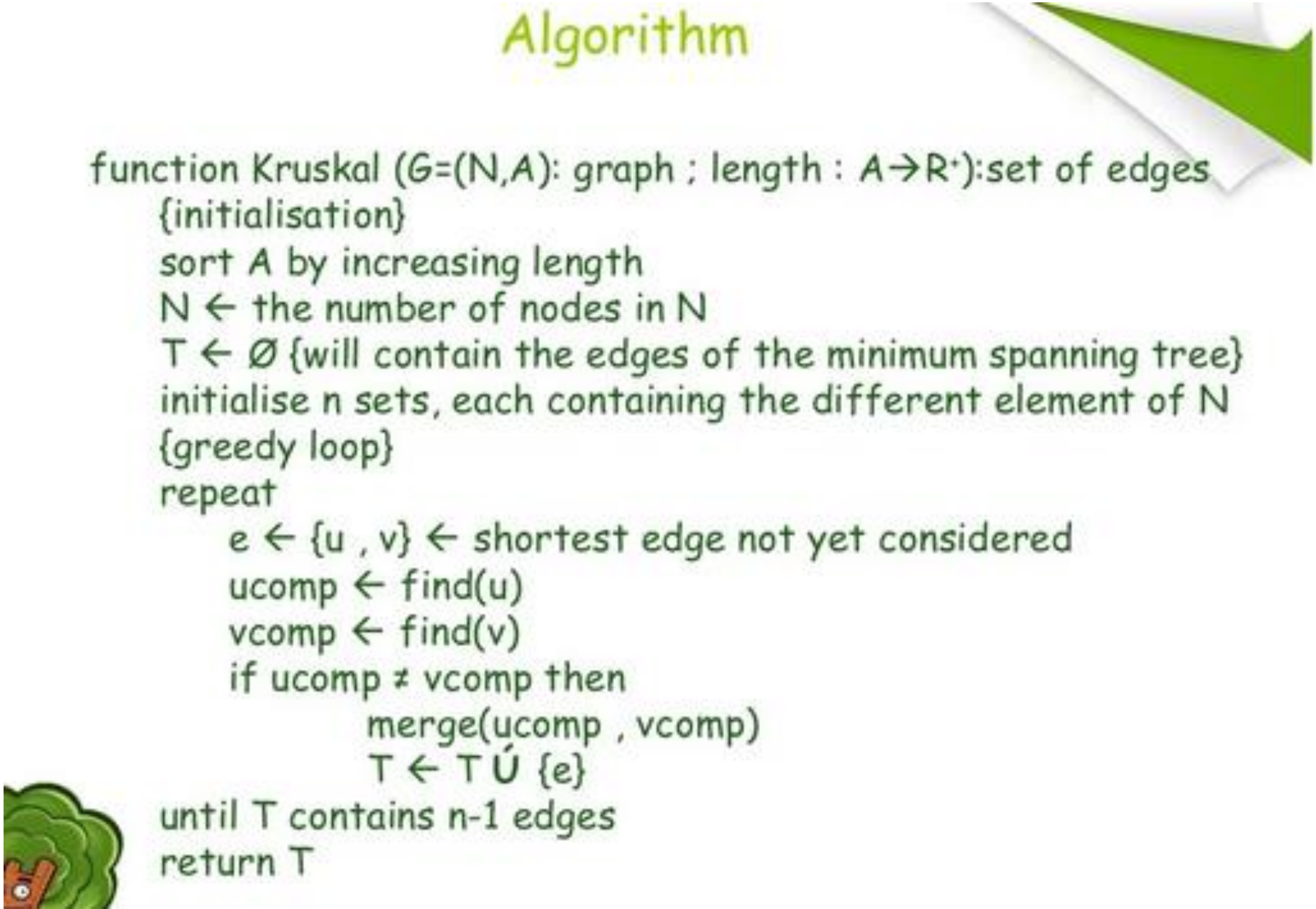


All vertices have been connected.

The solution is

ED	2
AB	3
CD	4
AE	4
EF	5

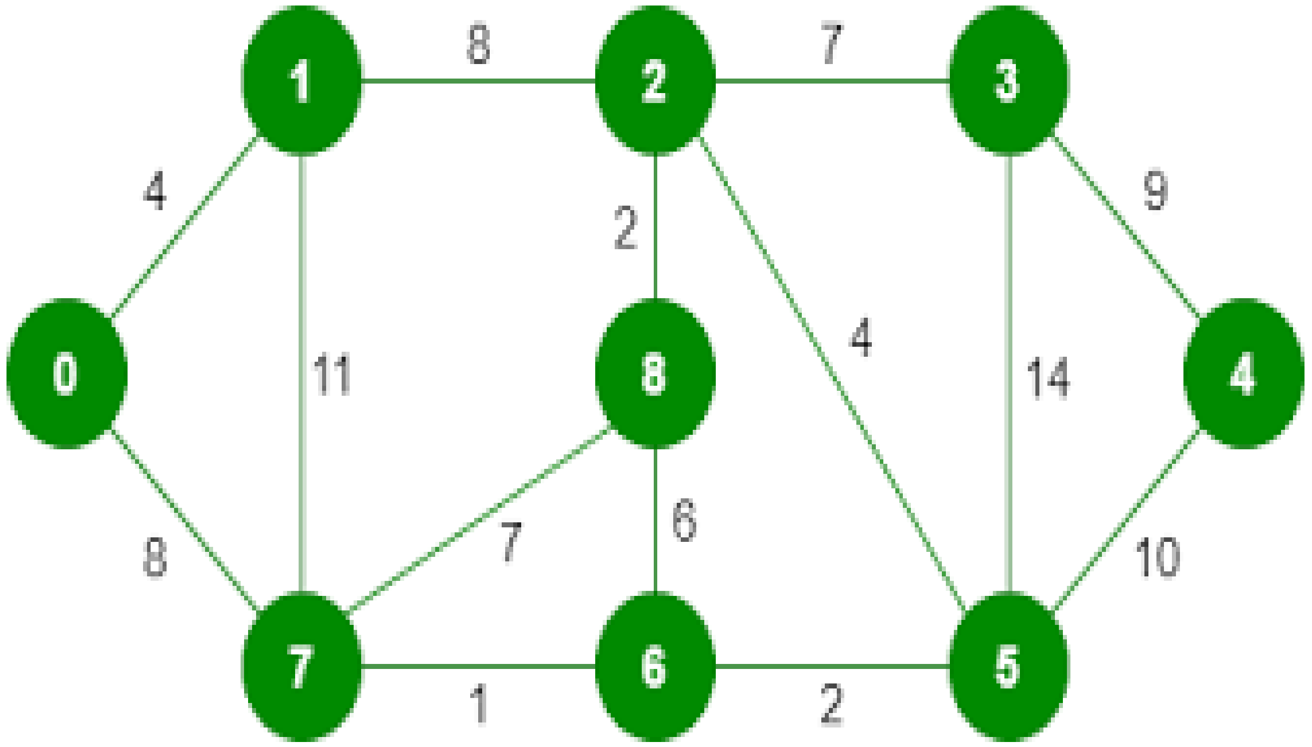
Total weight of tree: 18



Algorithm

```
function Kruskal (G=(N,A): graph ; length : A→R⁺):set of edges
    {initialisation}
    sort A by increasing length
    N ← the number of nodes in N
    T ← Ø {will contain the edges of the minimum spanning tree}
    initialise n sets, each containing the different element of N
    {greedy loop}
    repeat
        e ← {u , v} ← shortest edge not yet considered
        ucomp ← find(u)
        vcomp ← find(v)
        if ucomp ≠ vcomp then
                merge(ucomp , vcomp)
                T ← T Ú {e}
    until T contains n-1 edges
    return T
```

Find the minimum spanning tree using Kruskal algorithm



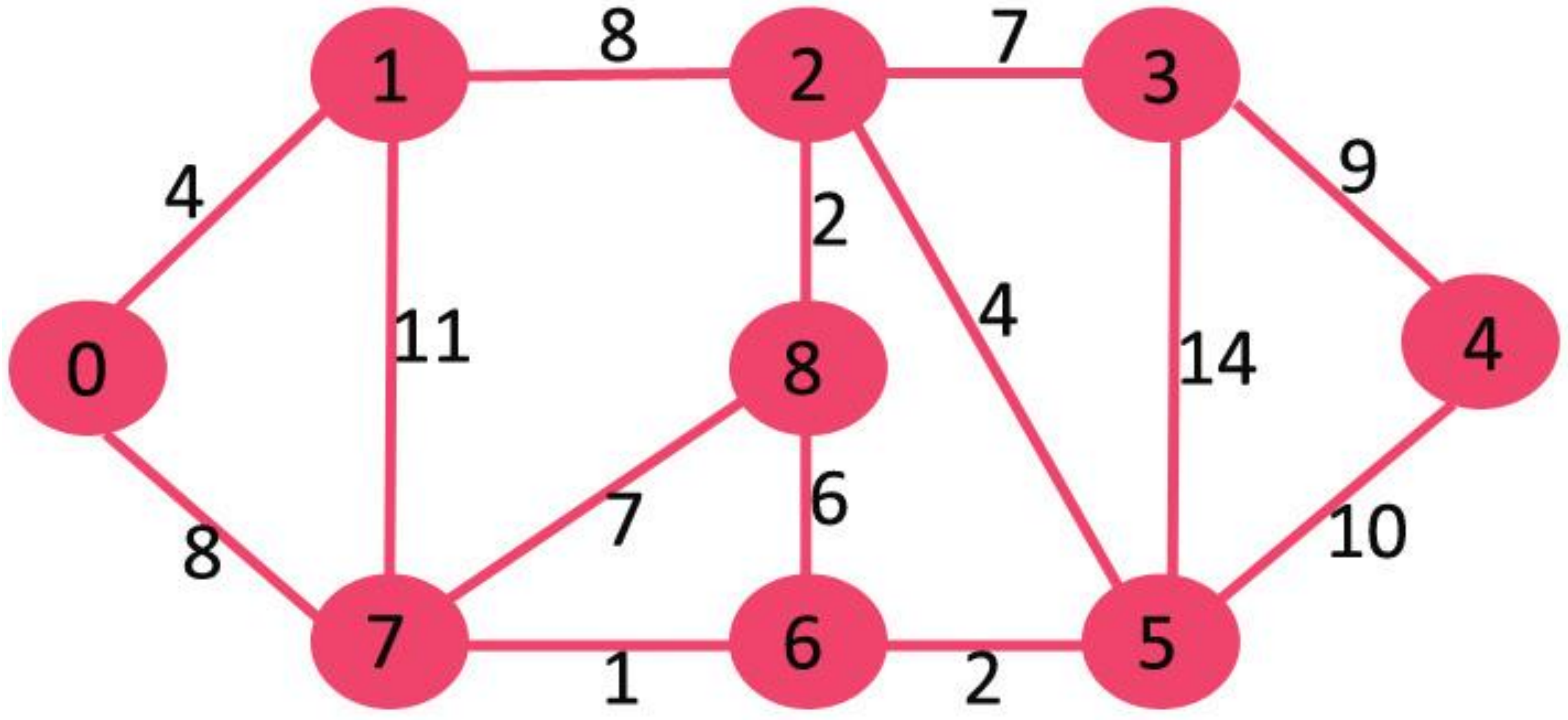
Minimum Spanning Tree: Prims Algorithm

- We have discussed Kruskal’s algorithm for Minimum Spanning Tree.
- Like Kruskal’s algorithm, Prim’s algorithm is also a Greedy algorithm.
- Prim’s algorithm always starts with a single node and it moves through several adjacent nodes, in order to explore all of the connected edges along the way.

Minimum Spanning Tree: Prims Algorithm

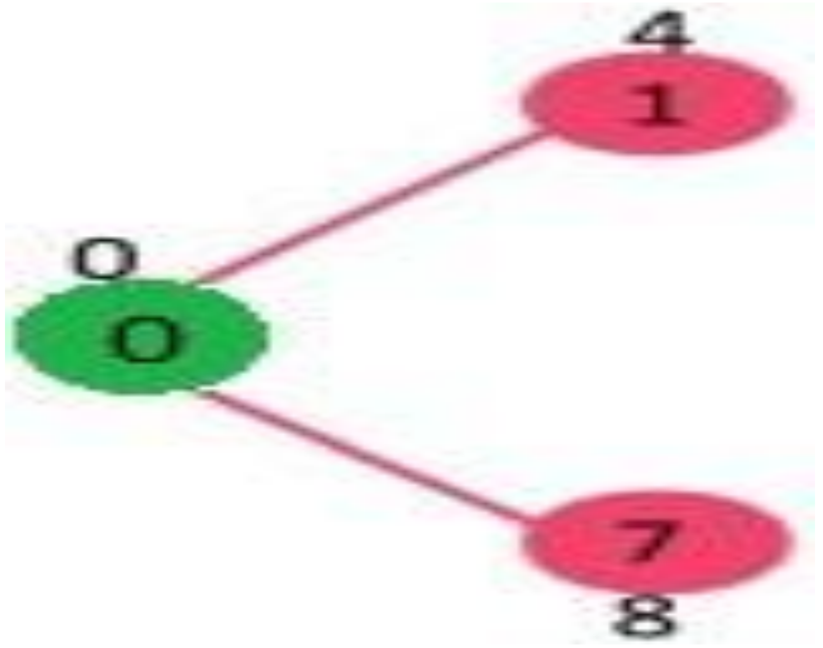
- It starts with an empty spanning tree.
- The idea is to maintain two sets of vertices.
- The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Minimum Spanning Tree: Prims Algorithm



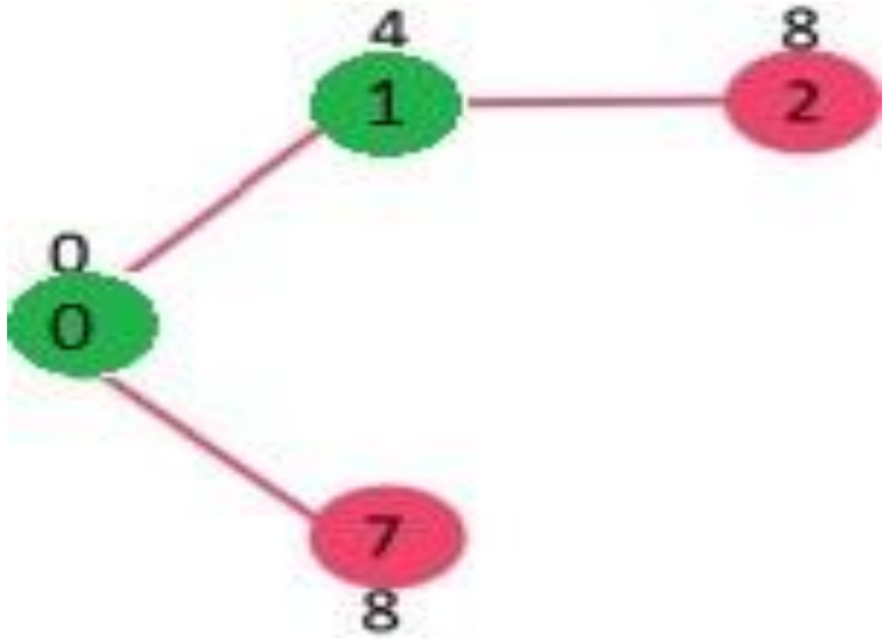
Minimum Spanning Tree: Prims Algorithm

- ***Step 1:*** *The set mstSet is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.*
- *Now pick the vertex with the minimum key value.*
- *The vertex 0 is picked, include it in mstSet. So mstSet becomes {0}.*
- *After including it to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7.*
- *The key values of 1 and 7 are updated as 4 and 8.*
- *Following subgraph shows vertices and their key values, only the vertices with finite key values are shown.*
- *The vertices included in MST are shown in green color.*



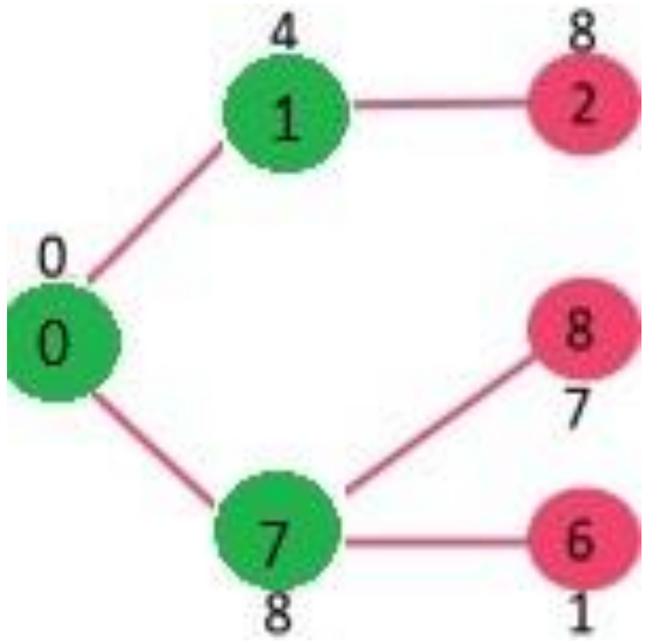
Prims Algorithm

- ***Step 2:*** *Pick the vertex with minimum key value and which is not already included in the MST (not in mstSET).*
- *The vertex 1 is picked and added to mstSet.*
- *So mstSet now becomes {0, 1}.*
- *Update the key values of adjacent vertices of 1.*
- *The key value of vertex 2 becomes 8.*



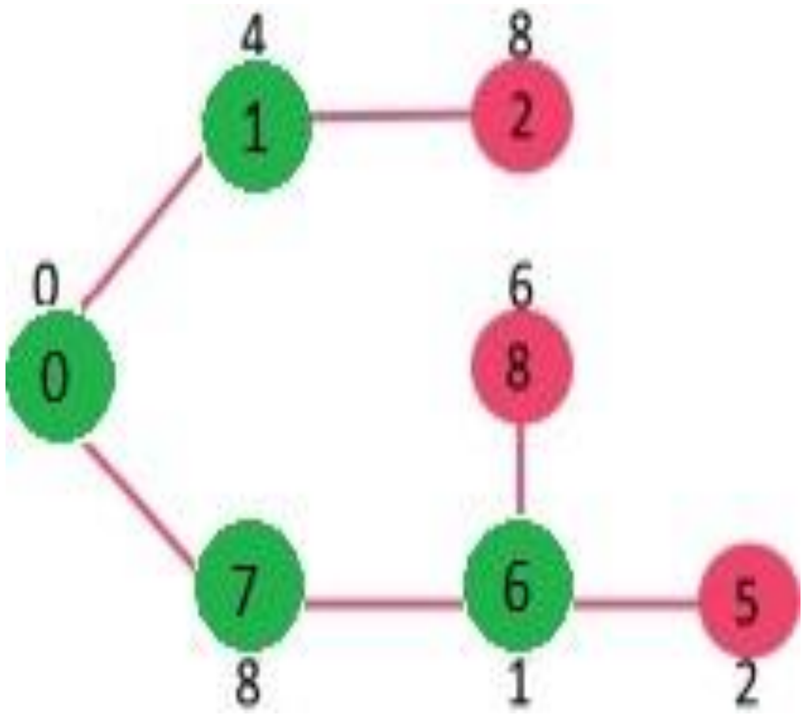
Prims Algorithm

- ***Step 3:*** *Pick the vertex with minimum key value and which is not already included in the MST (not in mstSET).*
- *We can either pick vertex 7 or vertex 2, let vertex 7 is picked.*
- *So mstSet now becomes {0, 1, 7}.*
- *Update the key values of adjacent vertices of 7.*
- *The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).*



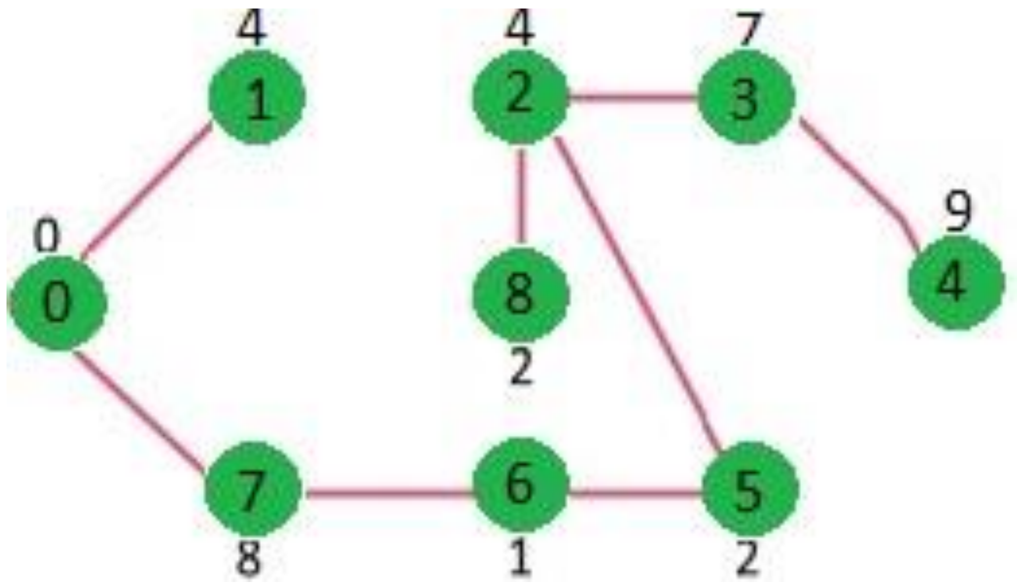
Prims Algorithm

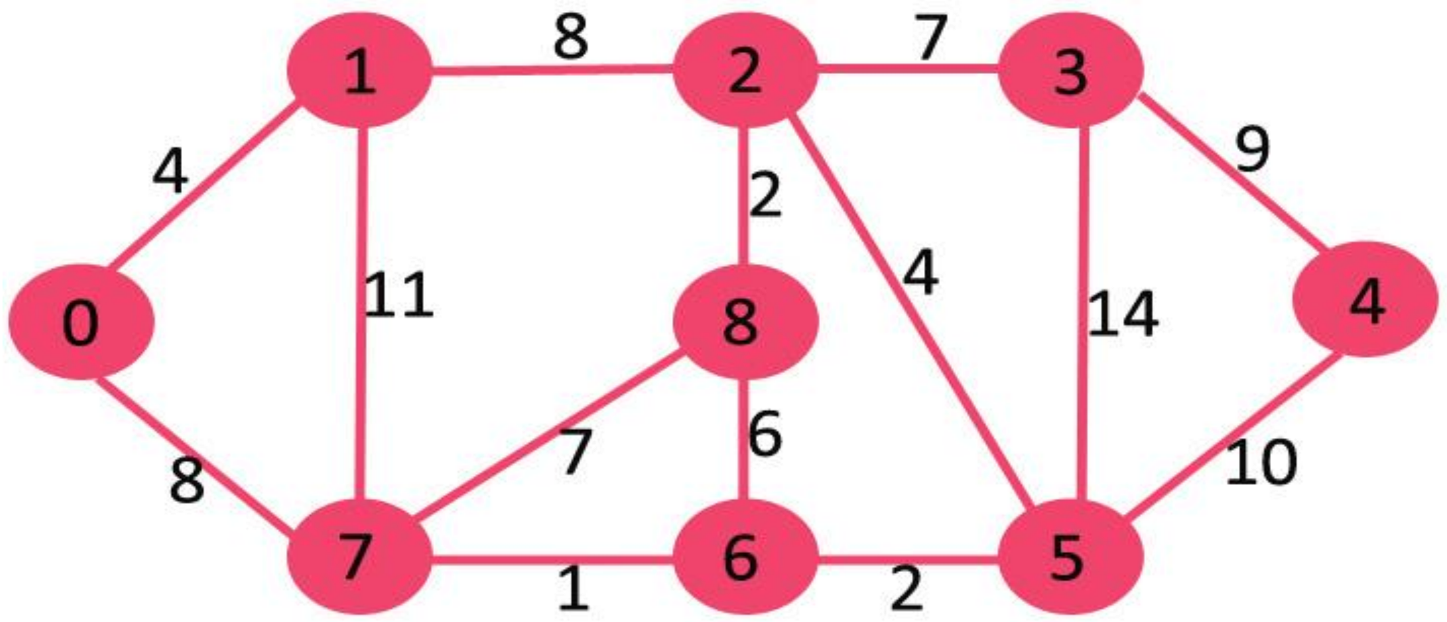
- ***Step 4:*** *Pick the vertex with minimum key value and not already included in MST (not in mstSET).*
- *Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}.*
- *Update the key values of adjacent vertices of 6.*
- *The key value of vertex 5 and 8 are updated.*



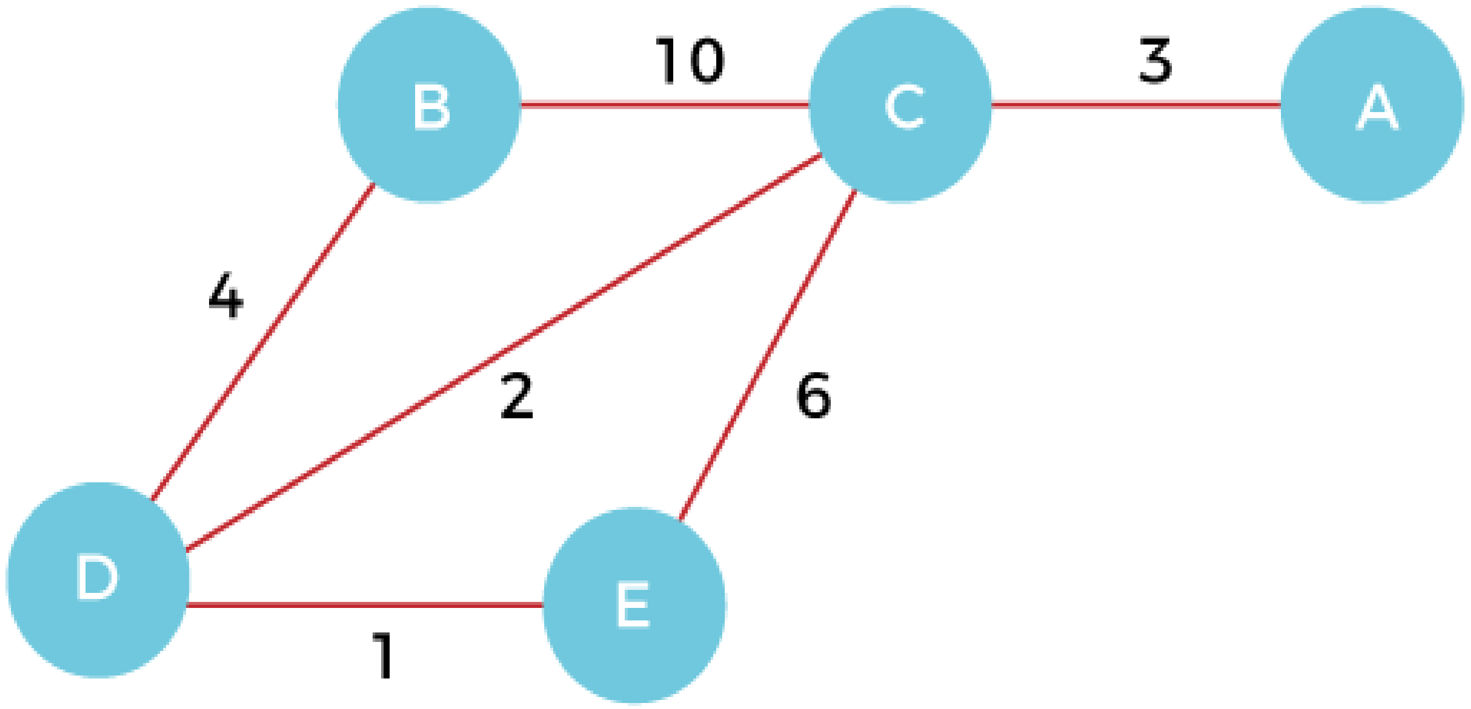
Prims Algorithm

- ***Step 5:*** *Repeat the above steps until mstSet includes all vertices of given graph.*
- *Finally, we get the following graph.*





Find minimum spanning tree

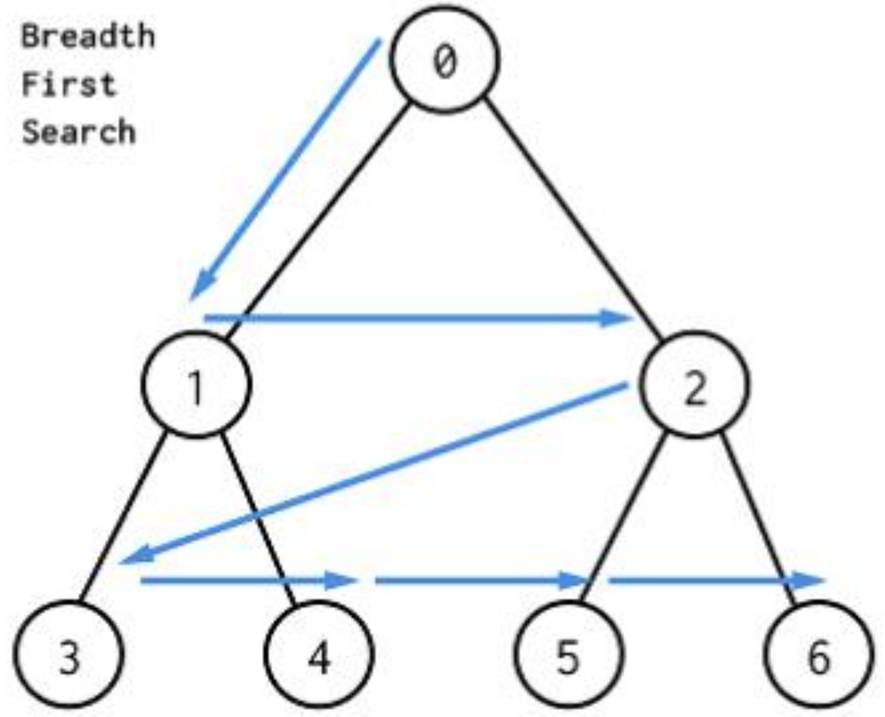


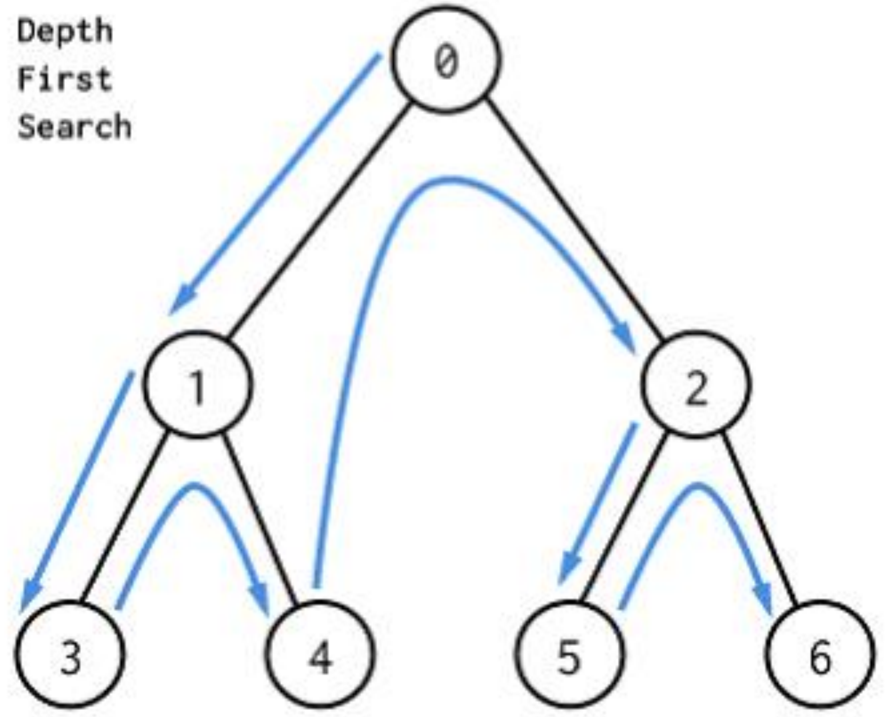
Graph as an ADT

- A graph is an abstract data type that is meant to implement the **undirected graph and directed graph** concepts from the field of graph theory within mathematics.

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 Algorithm

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., front > 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

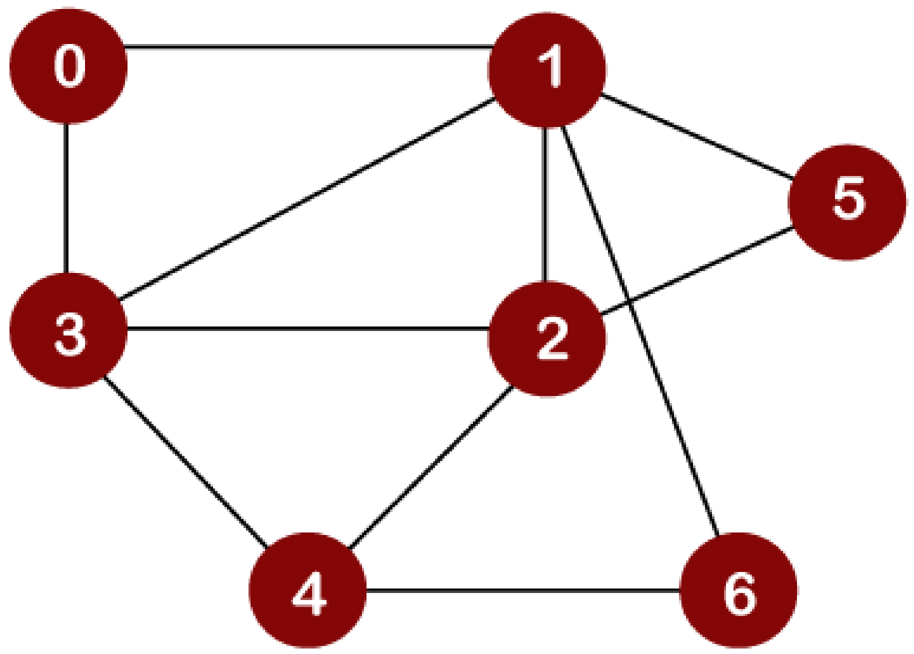
Breadth First Search Traversal

- BFS stands for ***Breadth First Search***.
- It is also known as **level order traversal**.
- The Queue data structure is used for the Breadth First Search traversal.
- When we use the BFS algorithm for the traversal in a graph, we can consider any node as a root node.

Let's consider the below graph for the breadth first search traversal.

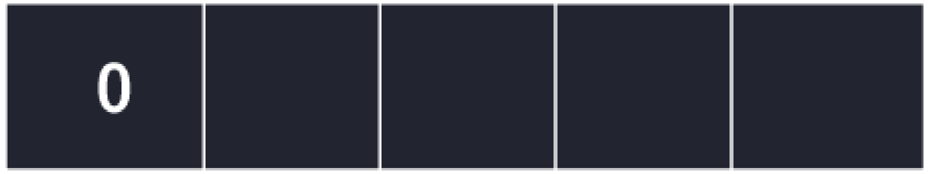
i. **Visiting vertex**

ii. **Exploring vertex**

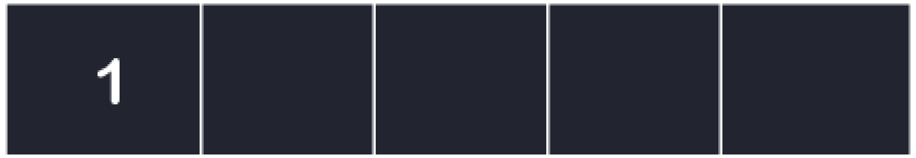


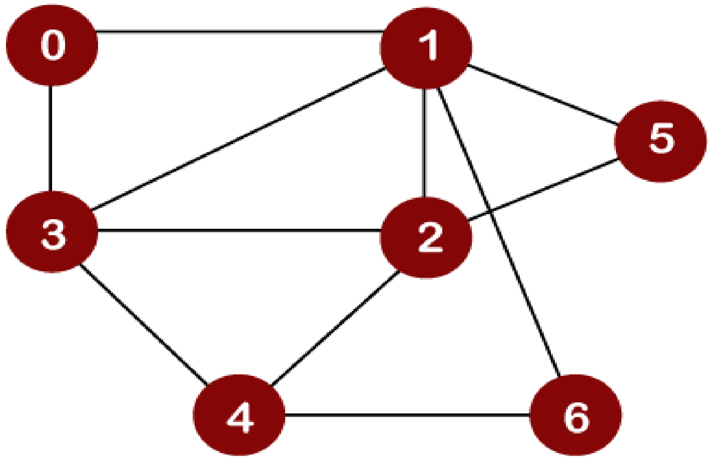
Breadth First Search Traversal

- Suppose we consider node 0 as a root node. Therefore, the traversing would be started from node 0.



- Once node 0 is removed from the Queue, it gets printed and marked as a ***visited node.***
- Once node 0 gets removed from the Queue, then the adjacent nodes of node 0 would be inserted in a Queue as shown below:

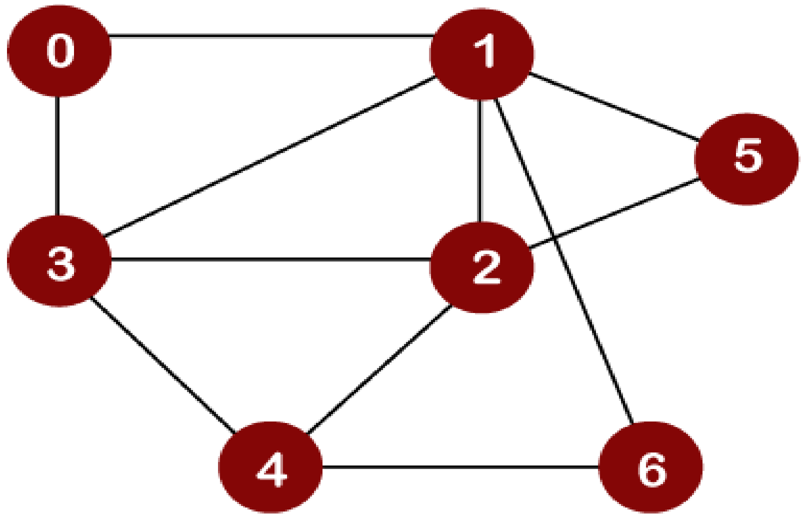


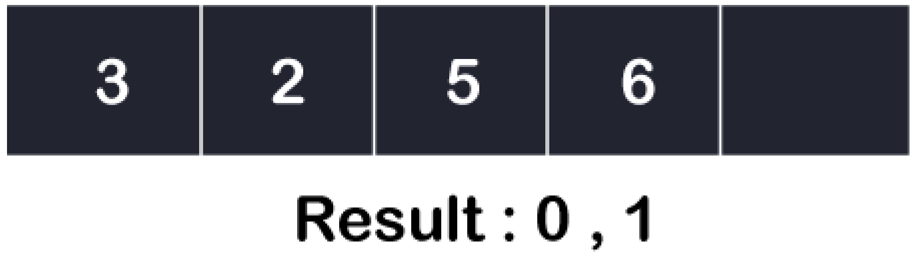


Result : 0

Breadth First Search Traversal

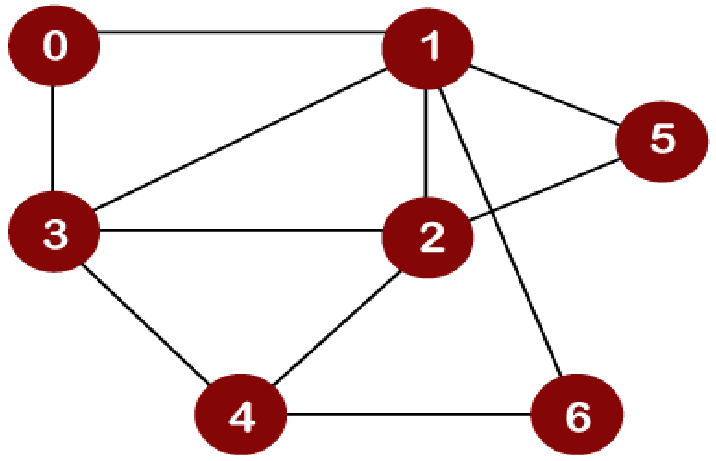
- Once node 1 gets removed from the Queue, then all the adjacent nodes of a node 1 will be added in a Queue.
- The adjacent nodes of node 1 are 0, 3, 2, 6, and 5.
- But we have to insert only unvisited nodes in a Queue.
- Since nodes 3, 2, 6, and 5 are unvisited; therefore, these nodes will be added in a Queue as shown below:

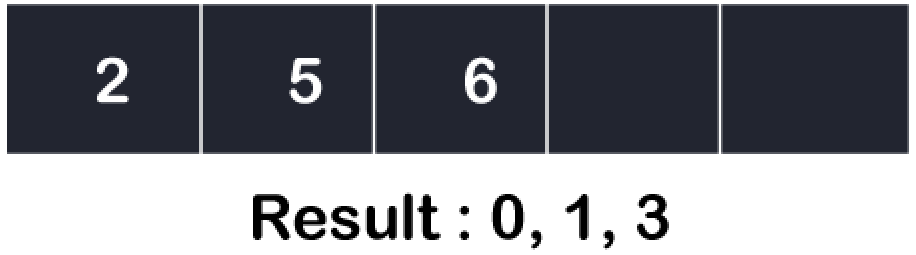




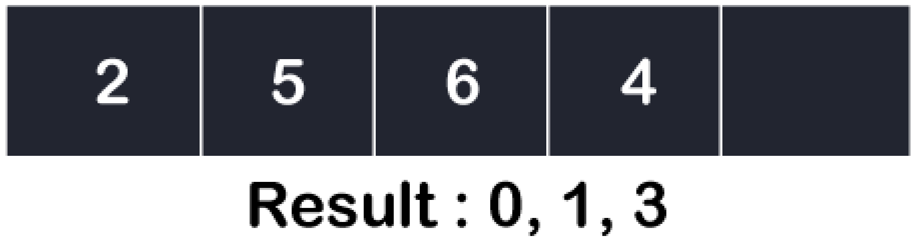
Breadth First Search Traversal

- The next node is 3 in a Queue. So, node 3 will be removed from the Queue, it gets printed and marked as visited as shown below:



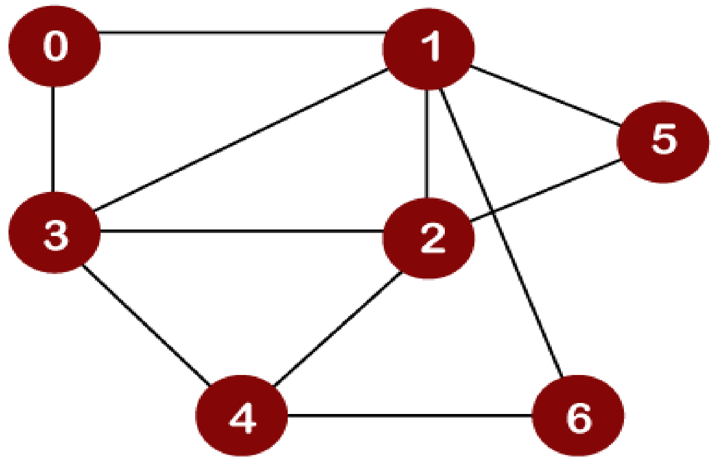


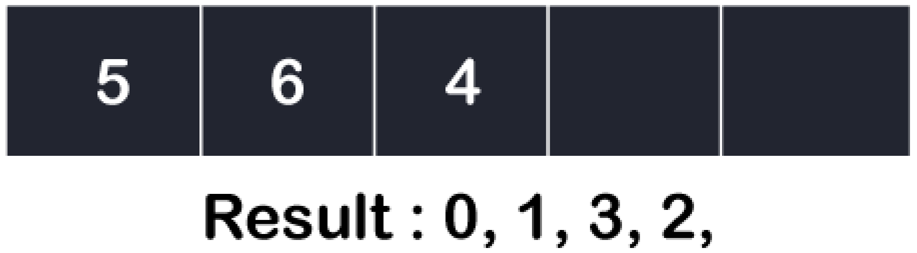
- Once node 3 gets removed from the Queue, then all the adjacent nodes of node 3 except the visited nodes will be added in a Queue.
- The adjacent nodes of node 3 are 0, 1, 2, and 4.
- Since nodes 0, 1 are already visited, and node 2 is present in a Queue; therefore, we need to insert only node 4 in a Queue.



Breadth First Search Traversal

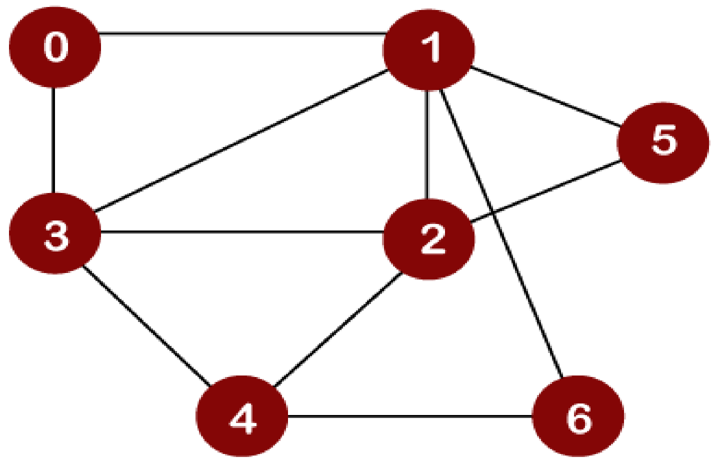
- Now, the next node in the Queue is 2. So, 2 would be deleted from the Queue. It gets printed and marked as visited as shown below:



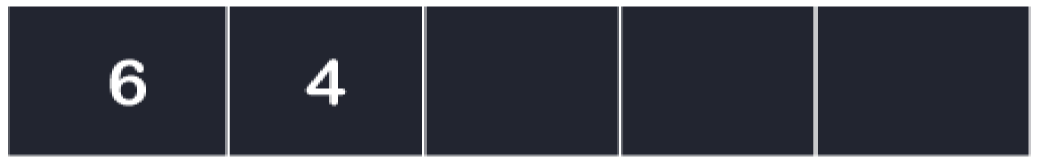


- Once node 2 gets removed from the Queue, then all the adjacent nodes of node 2 except the visited nodes will be added in a Queue.
- The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.

Breadth First Search Traversal



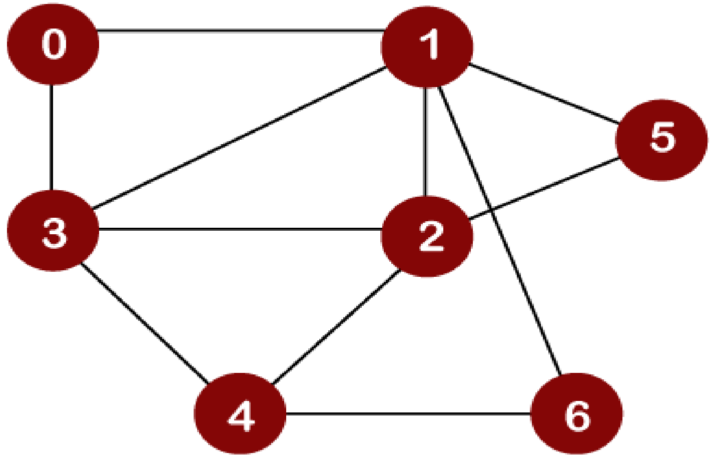
- The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.
- The next element is 5. So, 5 would be deleted from the Queue. It gets printed and marked as visited as shown below:

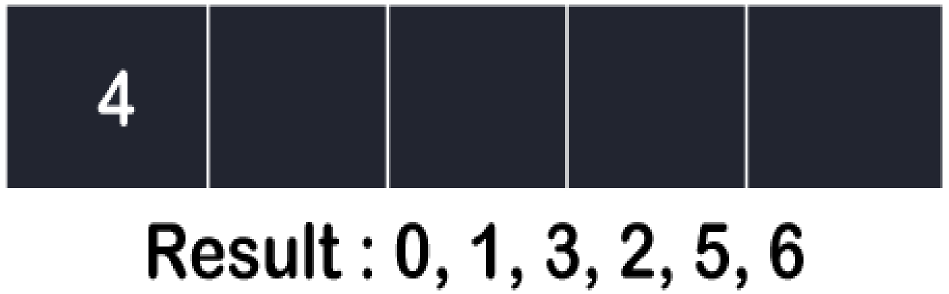


Result : 0, 1, 3, 2, 5

Breadth First Search Traversal

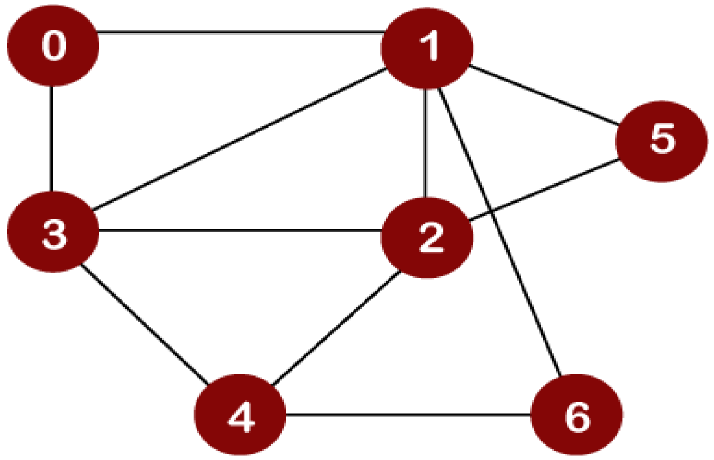
- Once node 5 gets removed from the Queue, then all the adjacent nodes of node 5 except the visited nodes will be added in the Queue.
- The adjacent nodes of node 5 are 1 and 2.
- Since both the nodes have already been visited; therefore, there is no vertex to be inserted in a Queue.
- The next node is 6. So, 6 would be deleted from the Queue. It gets printed and marked as visited as shown below:

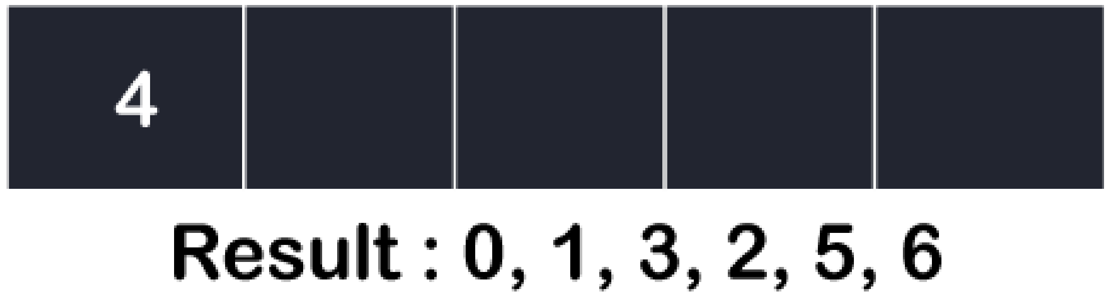




Breadth First Search Traversal

- Once the node 6 gets removed from the Queue, then all the adjacent nodes of node 6 except the visited nodes will be added in the Queue.
- The adjacent nodes of node 6 are 1 and 4.
- Since the node 1 has already been visited and node 4 is already added in the Queue; therefore, there is not vertex to be inserted in the Queue.
- The next element in the Queue is 4. So, 4 would be deleted from the Queue. It gets printed and marked as visited.





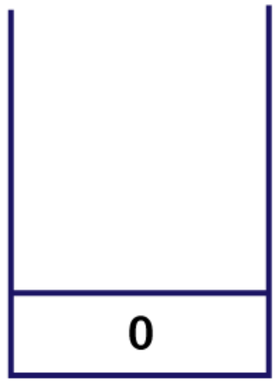
DFS Algorithm

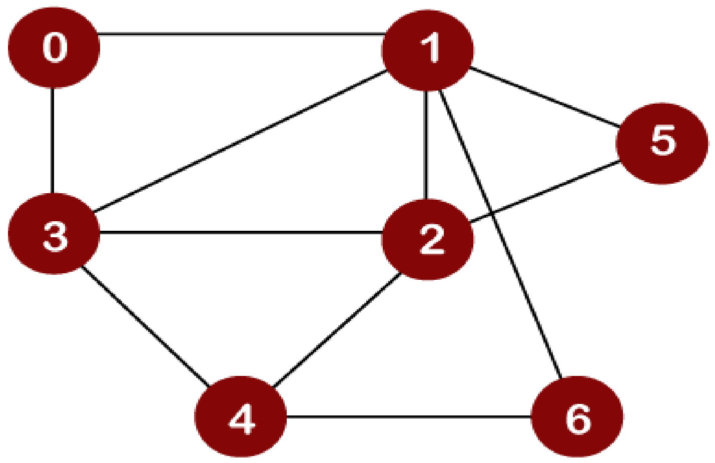
Algorithm

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 note visited.
5. Pop the top element of the stack and display it.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the stack and displayed (ie; not visited).
7. Exit.

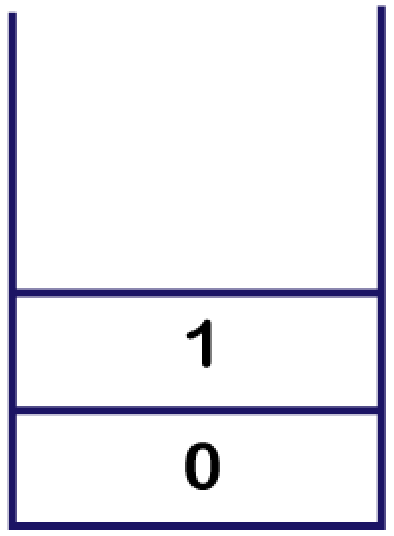
Depth First Search Traversal

- Consider node 0 as a root node.
- First, we insert the element 0 in the stack as shown below:



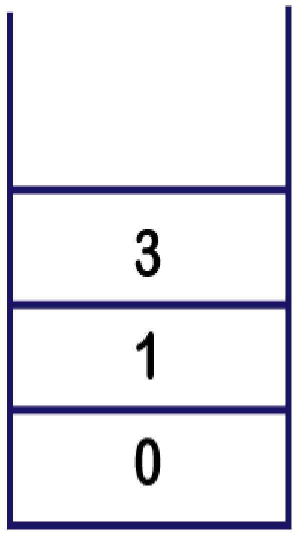


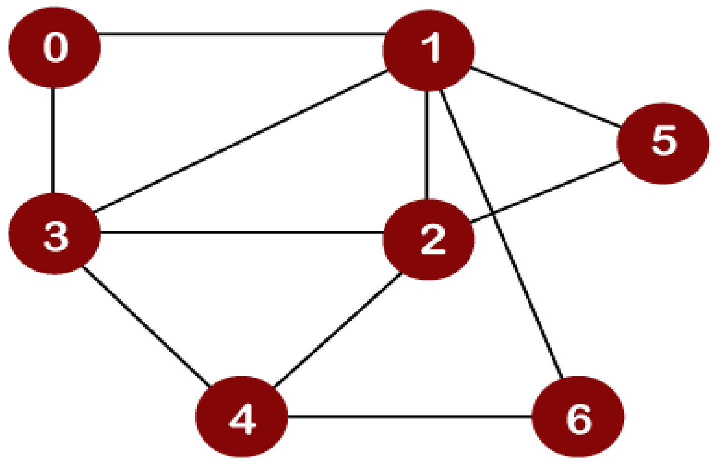
- The node 0 has two adjacent nodes, i.e., 1 and 3. Now we can take only one adjacent node, either 1 or 3, for traversing. Suppose we consider node 1; therefore, 1 is inserted in a stack and gets printed as shown below:



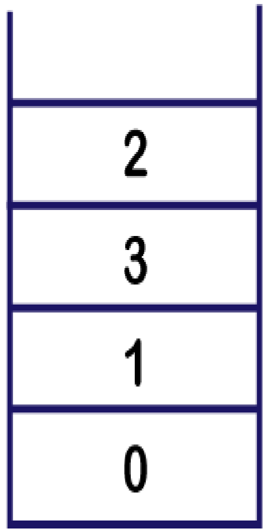
Depth First Search Traversal

- Now we will look at the adjacent vertices of node 1. The unvisited adjacent vertices of node 1 are 3, 2, 5 and 6. We can consider any of these four vertices. Suppose we take node 3 and insert it in the stack as shown below:



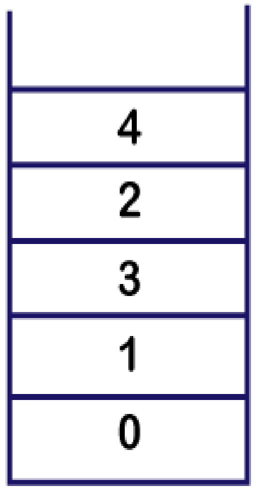


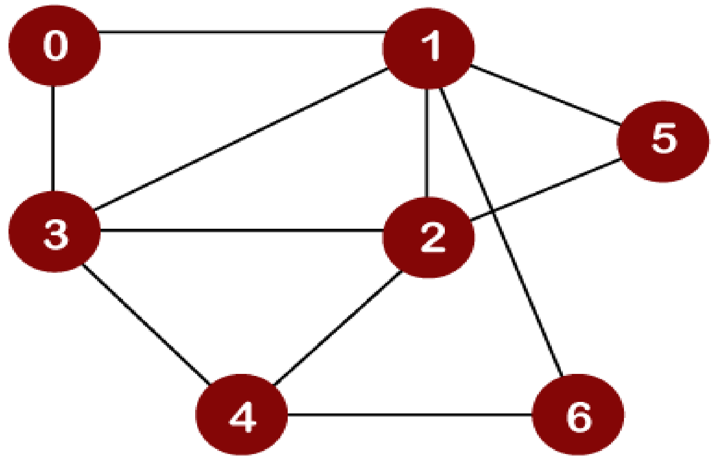
- Consider the unvisited adjacent vertices of node 3. The unvisited adjacent vertices of node 3 are 2 and 4. We can take either of the vertices, i.e., 2 or 4. Suppose we take vertex 2 and insert it in the stack as shown below:



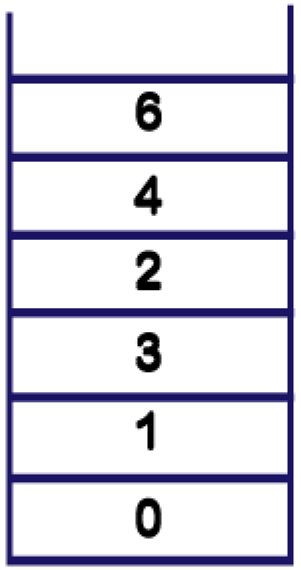
Depth First Search Traversal

- The unvisited adjacent vertices of node 2 are 5 and 4. We can choose either of the vertices, i.e., 5 or 4. Suppose we take vertex 4 and insert in the stack as shown below:



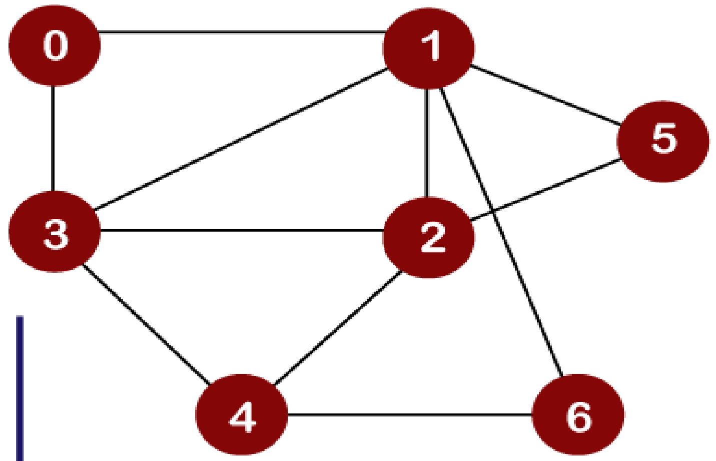


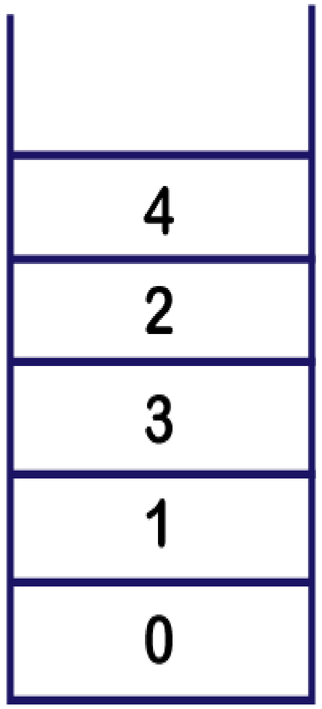
- Now we will consider the unvisited adjacent vertices of node 4. The unvisited adjacent vertex of node 4 is node 6. Therefore, element 6 is inserted into the stack as shown below:

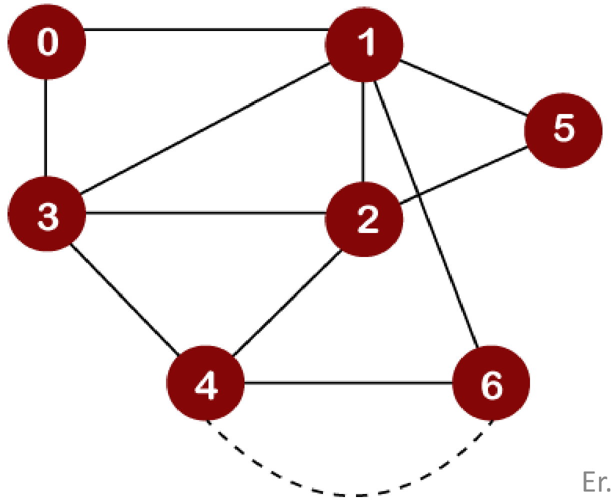


Depth First Search Traversal

- After inserting element 6 in the stack, we will look at the unvisited adjacent vertices of node 6.
- As there is no unvisited adjacent vertices of node 6, so we cannot move beyond node 6.
  In this case, we will perform **backtracking**.
- The topmost element, i.e., 6 would be popped out from the stack as shown below:

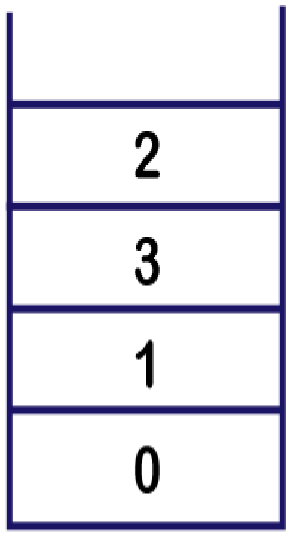


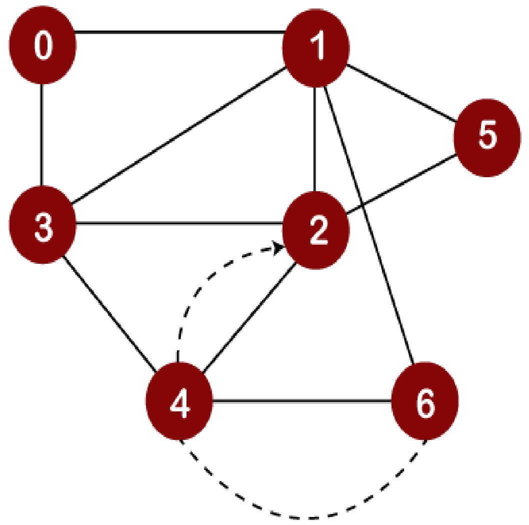




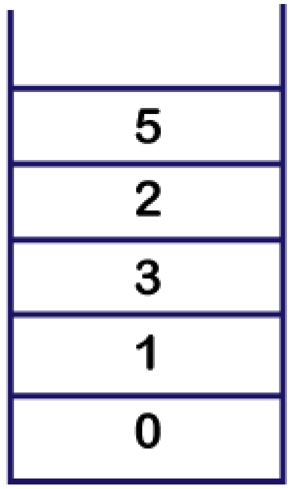
Depth First Search Traversal

- The topmost element in the stack is 4.
- Since there are no unvisited adjacent vertices left of node 4; therefore, node 4 is popped out from the stack as shown below:



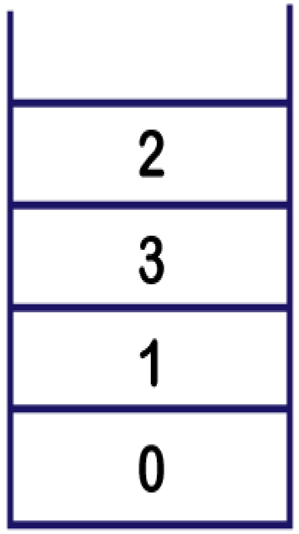


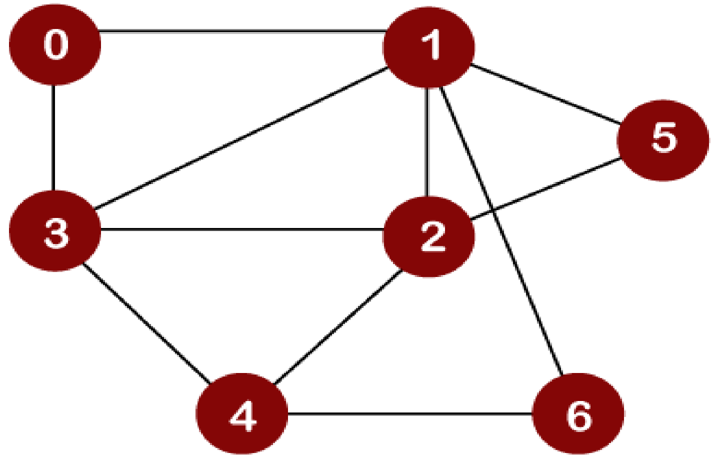
- The next topmost element in the stack is 2. Now, we will look at the unvisited adjacent vertices of node 2.
- Since only one unvisited node, i.e., 5 is left, so node 5 would be pushed into the stack above 2 and gets printed as shown below:



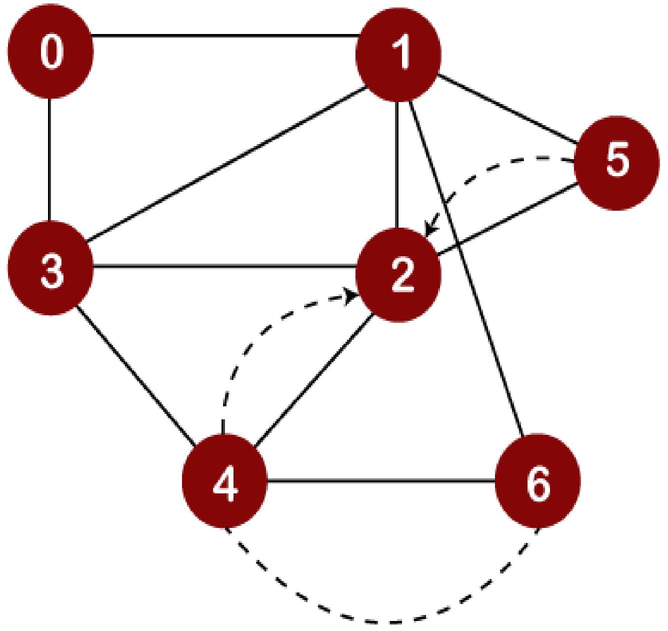
Depth First Search Traversal

- Now we will check the adjacent vertices of node 5, which are still unvisited.
- Since there is no vertex left to be visited, so we pop the element 5 from the stack as shown below:



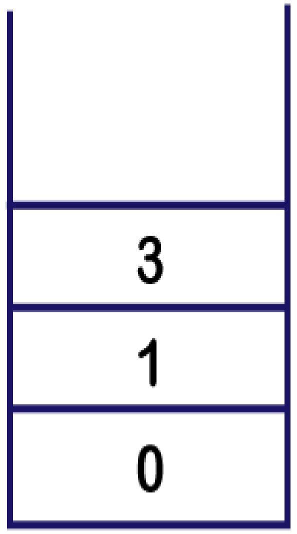


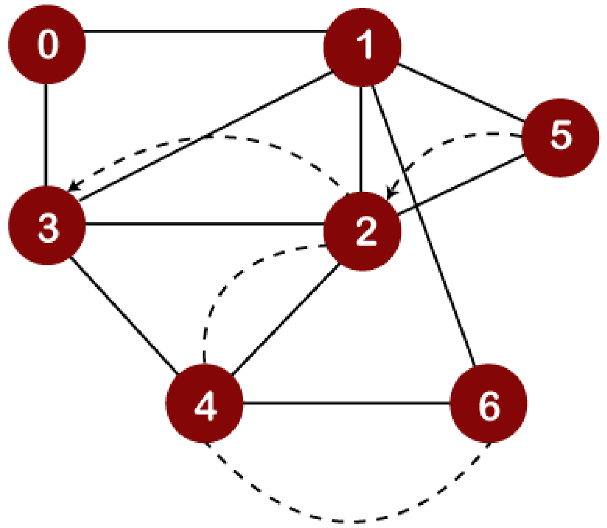
- We cannot move further 5, so we need to perform backtracking.
- In backtracking, the topmost element would be popped out from the stack.
- The topmost element is 5 that would be popped out from the stack, and we move back to node 2 as shown below:



Depth First Search Traversal

- Now we will check the unvisited adjacent vertices of node 2.
- As there is no adjacent vertex left to be visited, so we perform backtracking.
- In backtracking, the topmost element, i.e., 2 would be popped out from the stack, and we move back to the node 3 as shown below:

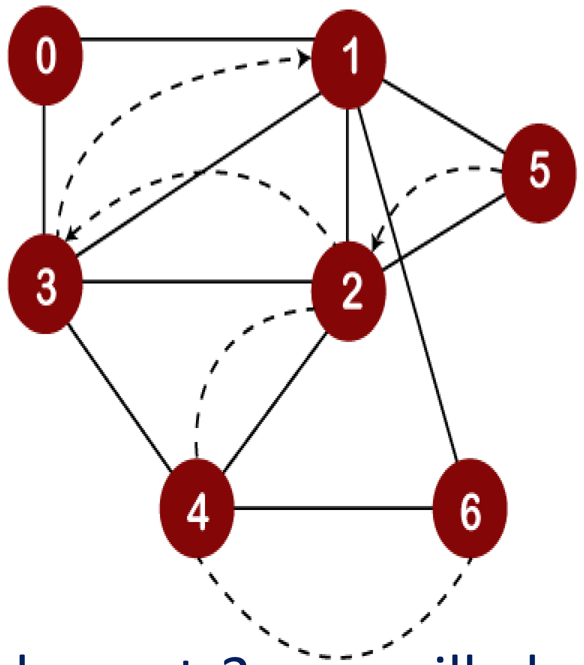


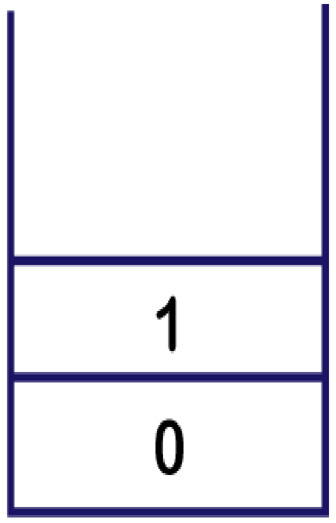


- Now we will check the unvisited adjacent vertices of node 3.
- As there is no adjacent vertex left to be visited, so we perform backtracking.
- In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:

Depth First Search Traversal

- Now we will check the unvisited adjacent vertices of node 3.
- As there is no adjacent vertex left to be visited, so we perform backtracking.
- In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:

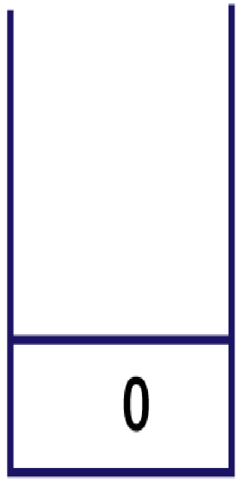


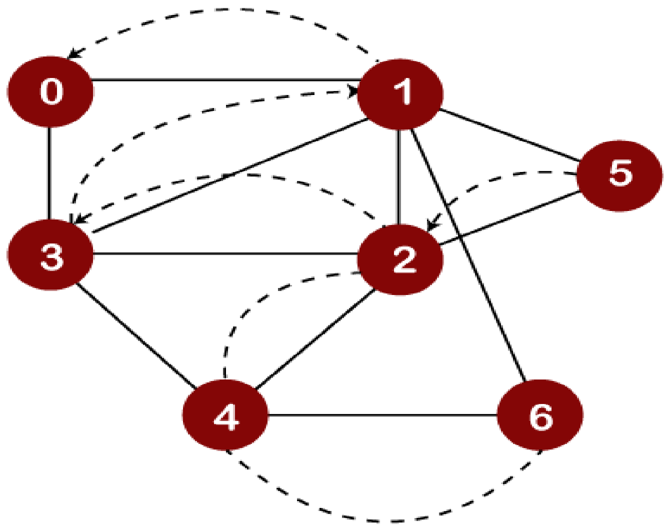


- After popping out element 3, we will check the unvisited adjacent vertices of node 1.
- Since there is no vertex left to be visited; therefore, the backtracking will be performed.
- In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:

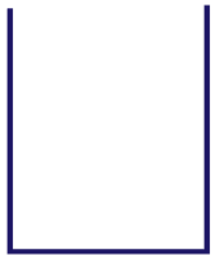
Depth First Search Traversal

- After popping out element 3, we will check the unvisited adjacent vertices of node 1.
- Since there is no vertex left to be visited; therefore, the backtracking will be performed.
- In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:





- We will check the adjacent vertices of node 0, which are still unvisited.
- As there is no adjacent vertex left to be visited, so we perform backtracking.
- In this, only one element, i.e., 0 left in the stack, would be popped out from the stack as shown below:



Empty

- As we can observe in the above figure that the stack is empty.
- So, we have to stop the DFS traversal here, and the elements which are printed is the result of the DFS traversal.

Shortest Path Algorithm

- There are many problems that can be modeled using graph with weight assigned to their edges.
- Eg, we may set up the basic graph model by representing cities by vertices and flights by edges.
- Problems involving distances can be modeled by assigning distances between cities to the edges. Problems involving flight time can be modeled by assigning flight times to edges.
- Problem involving fares can be modeled by assigning fares to the edges.
- Thus, we can model airplane or other mass transit routes by graphs and use shortest path algorithm to compute the best route between two points.

Shortest Path Algorithm

- ➢Similarly, if the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs, delay costs, then we can use the shortest-path algorithm to find the cheapest way to send electronic news, data from one computer to a set of other computers.
- ➢Basically, there are three types of shortest path problems:

Single path: Given two vertices, s and d, find the shortest path from s to d and its length (weights).

Single source: Given a vertex, s find the shortest path to all other vertices. (BFS)

All pairs: Find the shortest path from all pair of vertices.

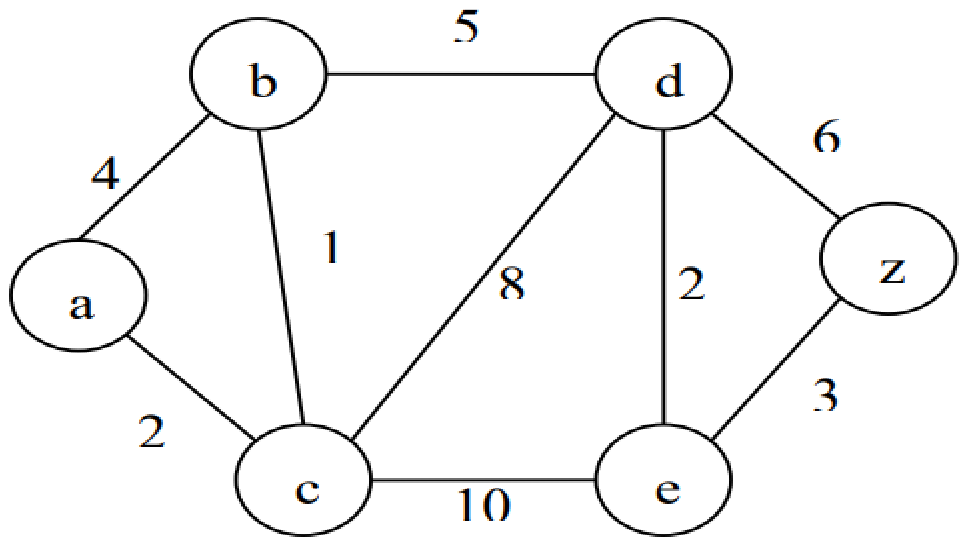
Dijkstra shortest path algorithm

Dijkstra's Algorithm to find the shortest path

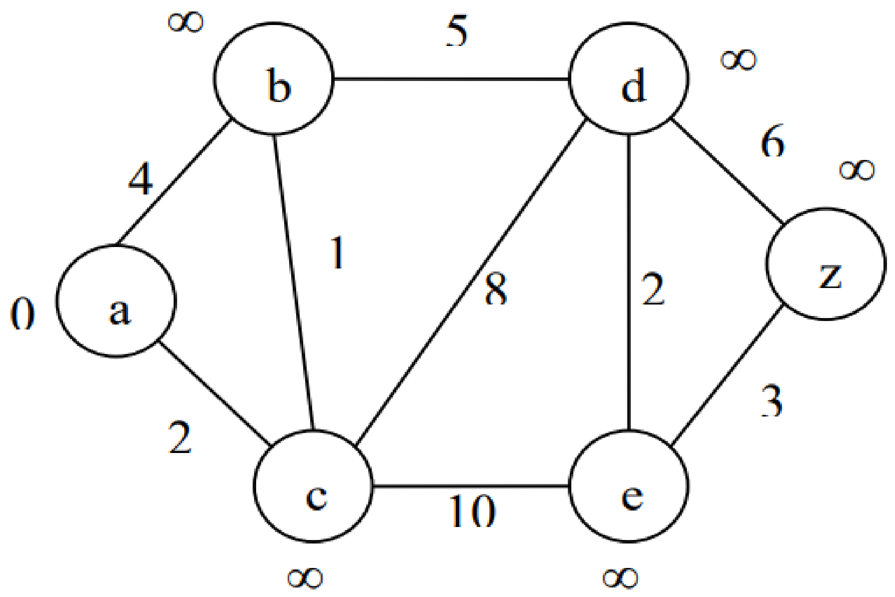
1. Mark all the vertices as unknown.
2. For each vertex v keep a distance d(v) from source vertex s to v initially set to infinity except for s which is set to ds = 0.
3. Repeat these steps until all vertices are known
   I. Select a vertex v, which has the smallest d(v) among all the unknown vertices
   II. Mark v as known
   III. For each vertex w adjacent to v
      i. if w is unknown and dv + cost(v, w) < d(w)
      ii. update d(w) to d(v) + cost(v, w)

Dijkstra shortest path algorithm

Dijkstra's Algorithm to find the shortest path

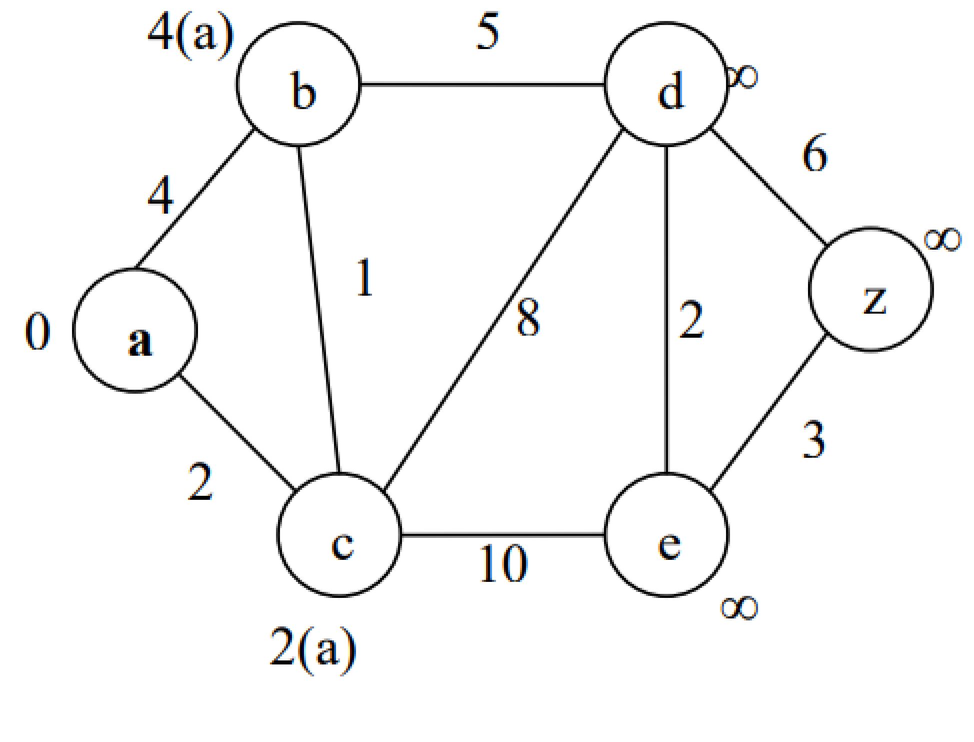


Given graph with weights

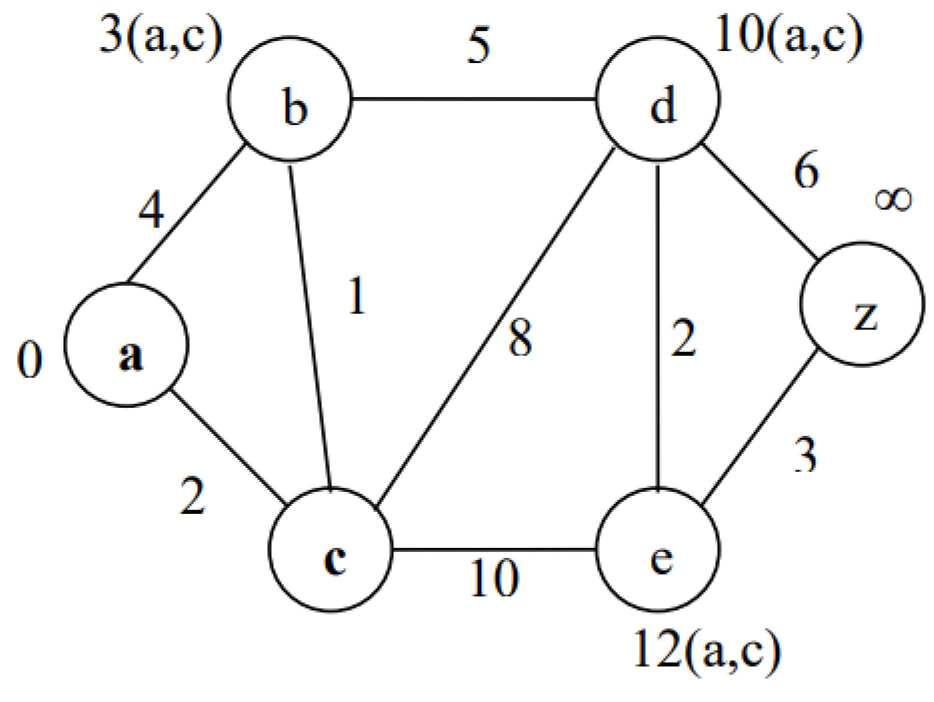


(a)

Dijkstra shortest path algorithm

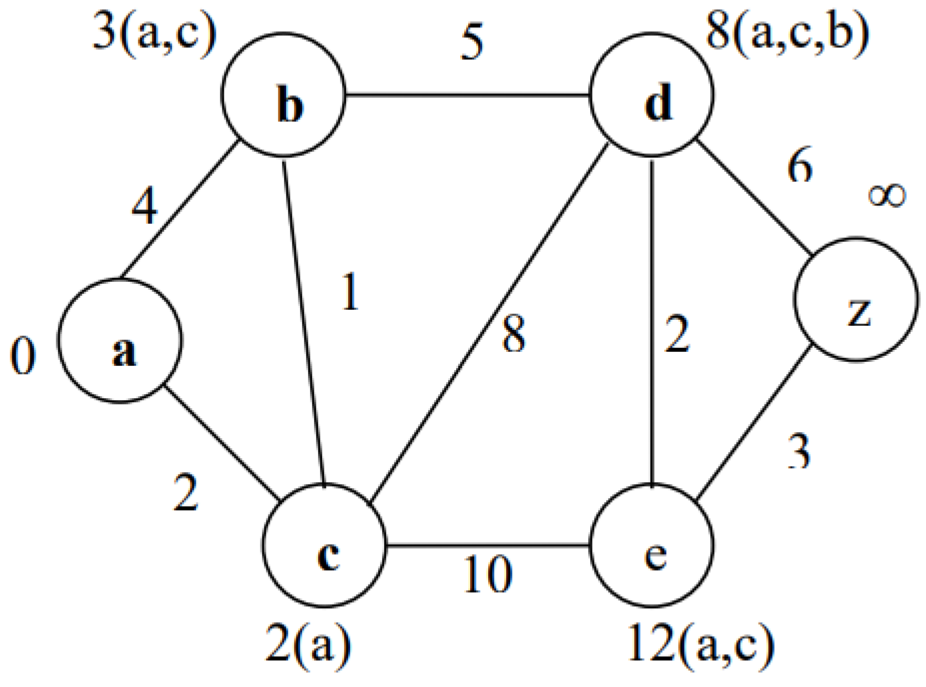


(b)

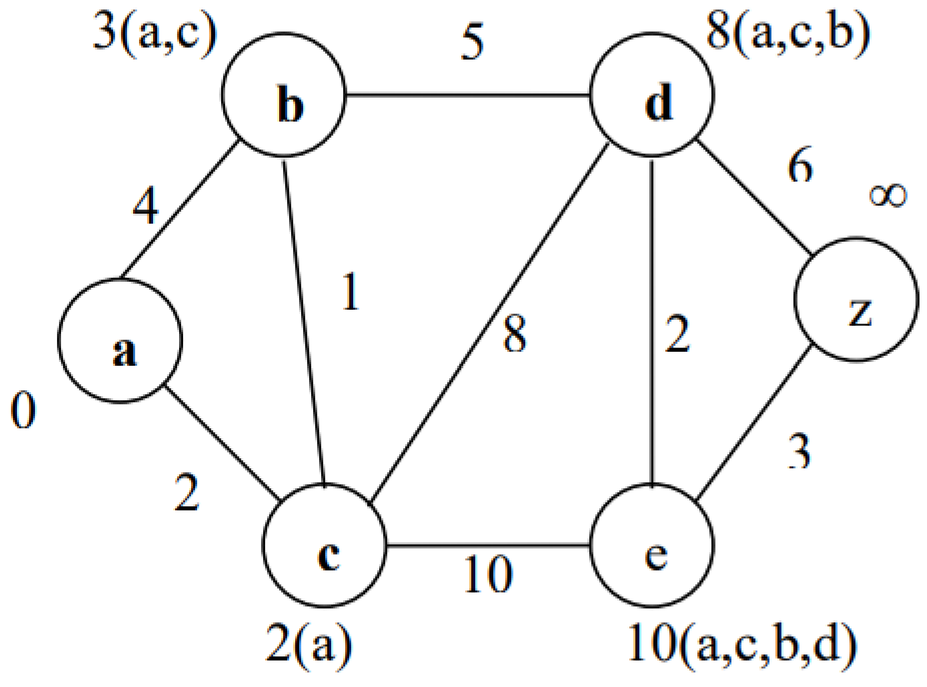


(c)

Dijkstra shortest path algorithm

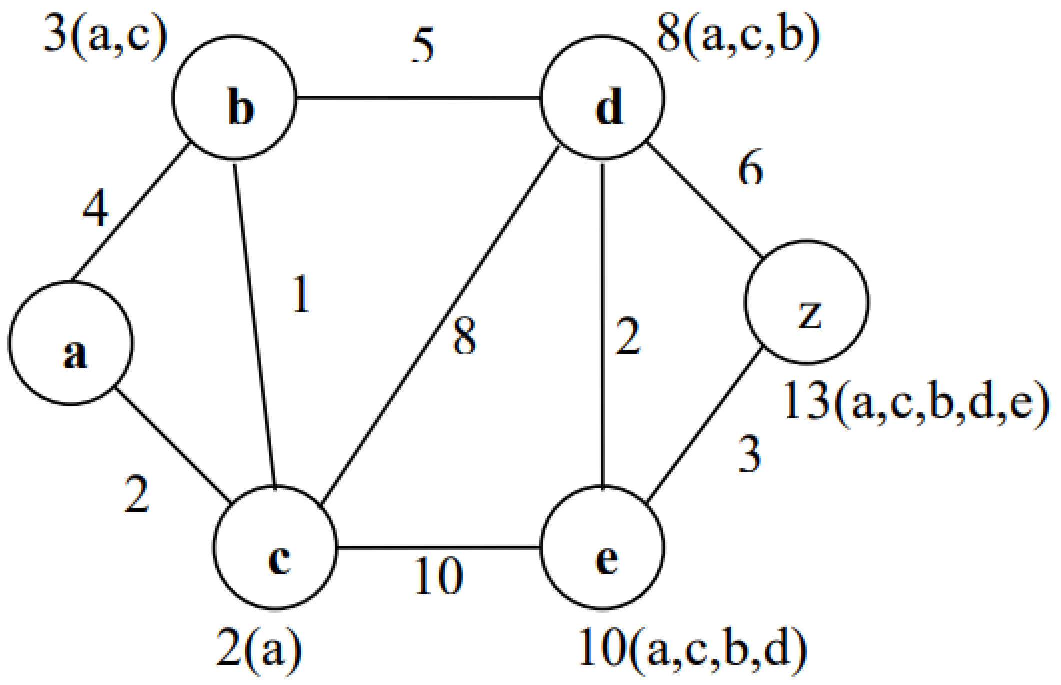


(d)

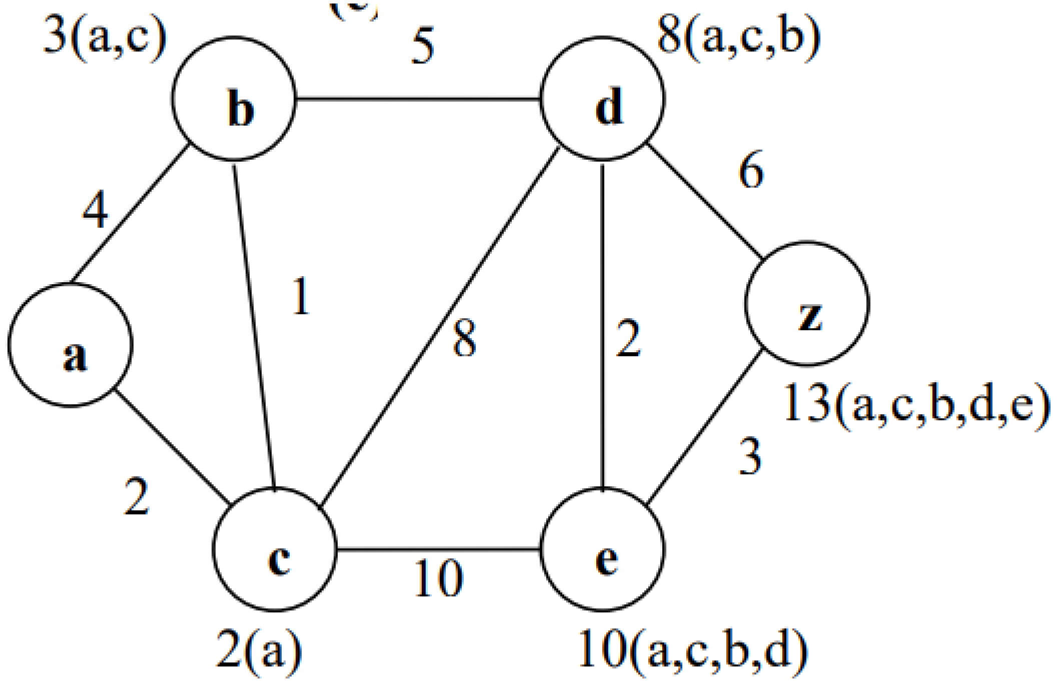


(e)

Dijkstra shortest path algorithm



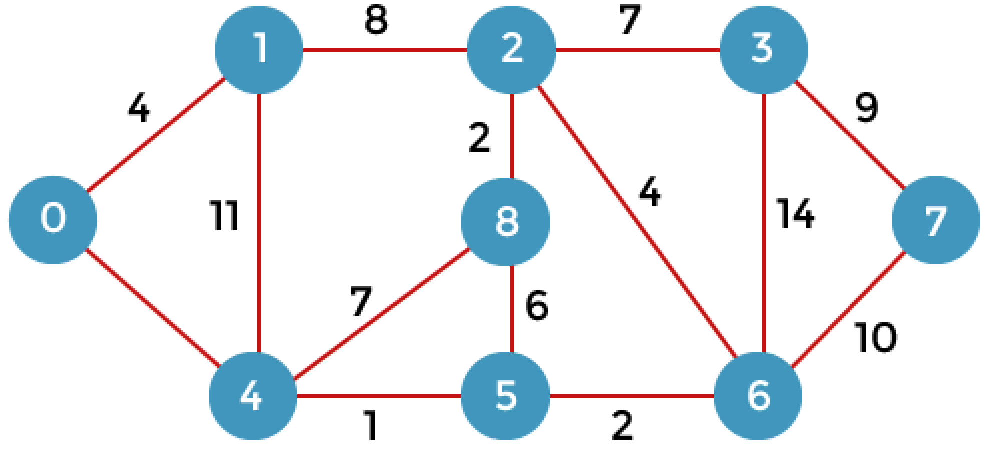
(f)



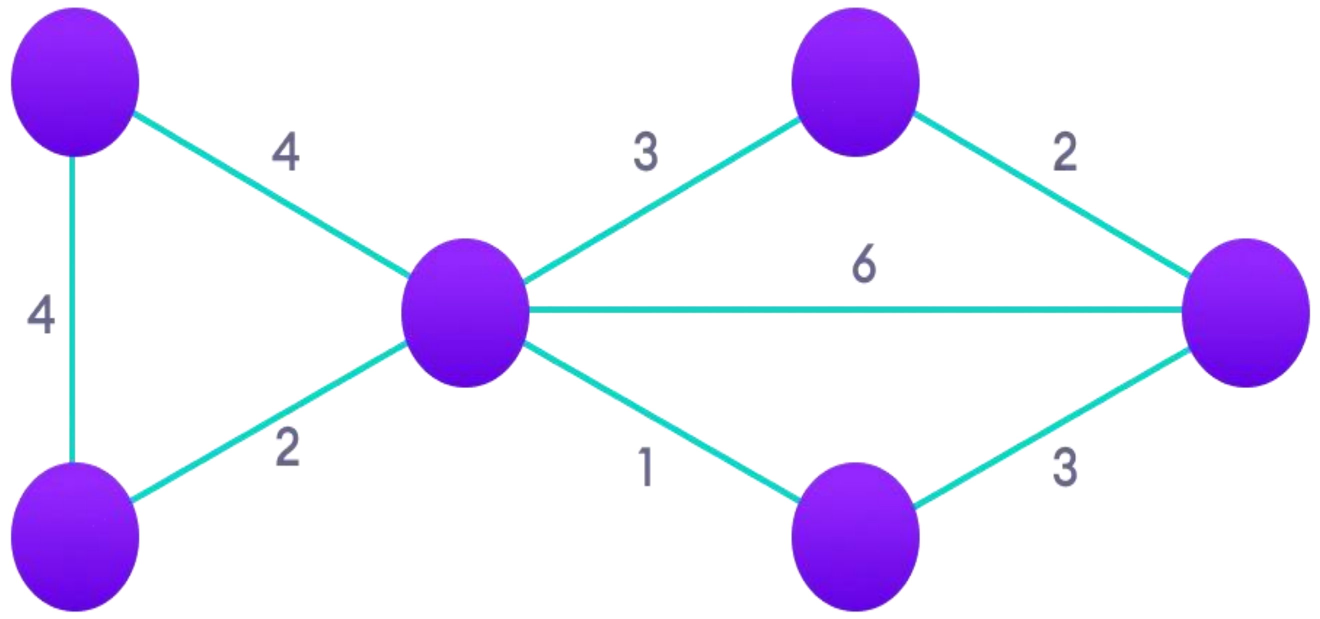
(g)

Fig: Using Dijkstra's Algorithm to find shortest path from a to z

Dijkstra shortest path algorithm



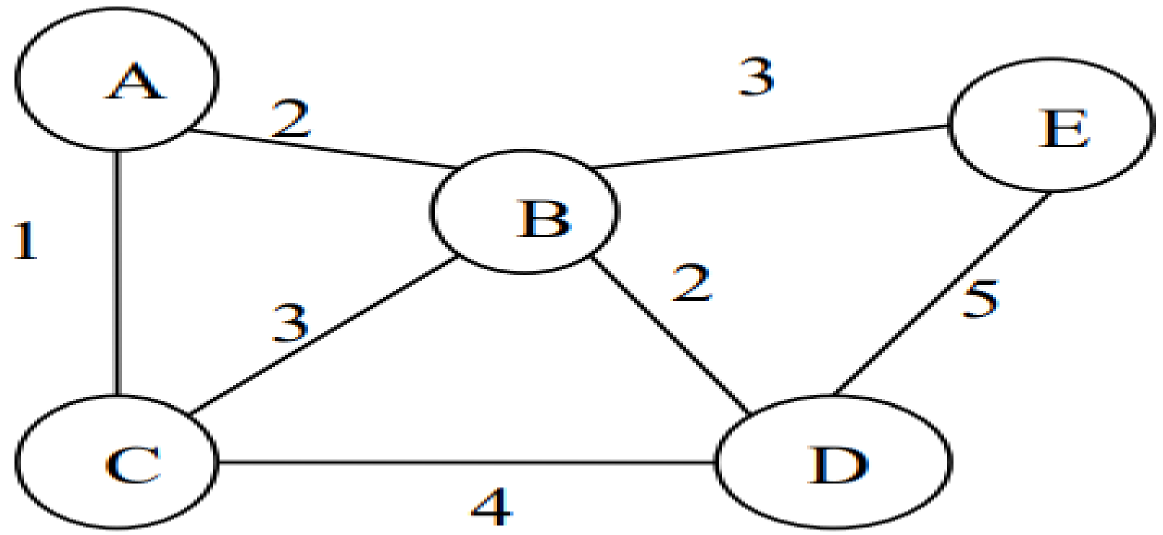
Find shortest path using Dijkstra



Step: 1

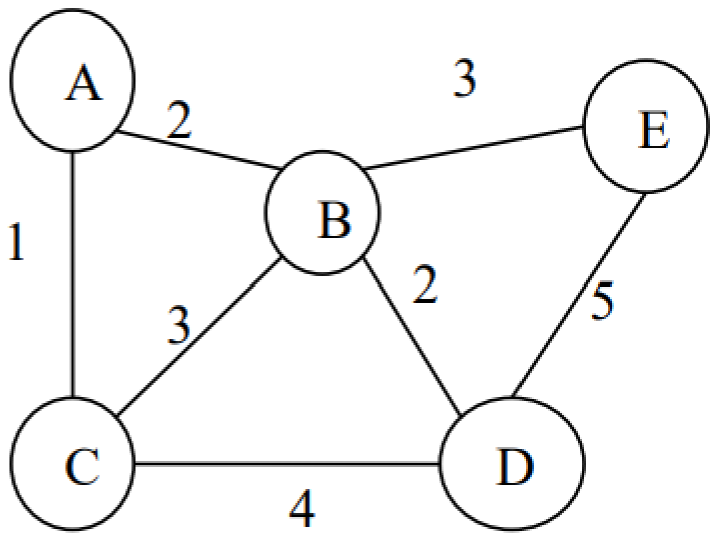
Round Robin Algorithm

- This method provides better performance when the number of edges is low.
- Initially each node is considered to be a partial tree. Each partial tree is maintained in a queue Q.
- Priority queue is associated with each partial tree, which contains all the arcs ordered by their weights.



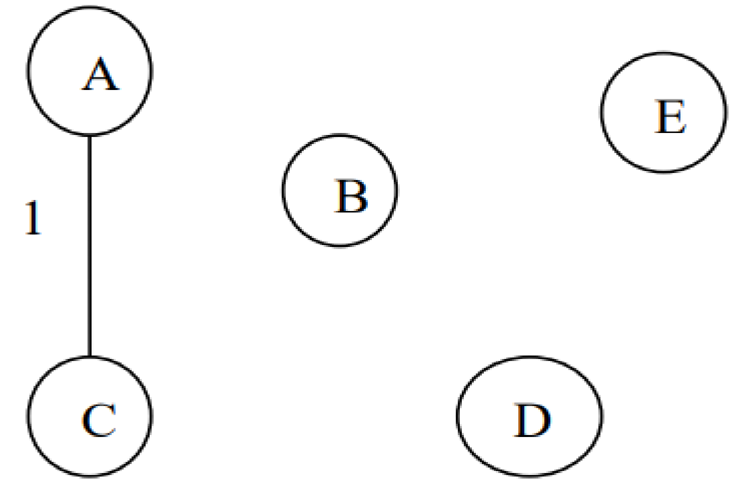
Round Robin Algorithm

- The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight arc a in T1; deleting from Q, the tree T2, at the other end of arc a; combining T1 and T2 into a single new tree T3 and at the same time combining priority queues of T1 and T2 and adding T3 at the rear of priority queue.
- This continues until Q contains a single tree, the minimum spanning tree.

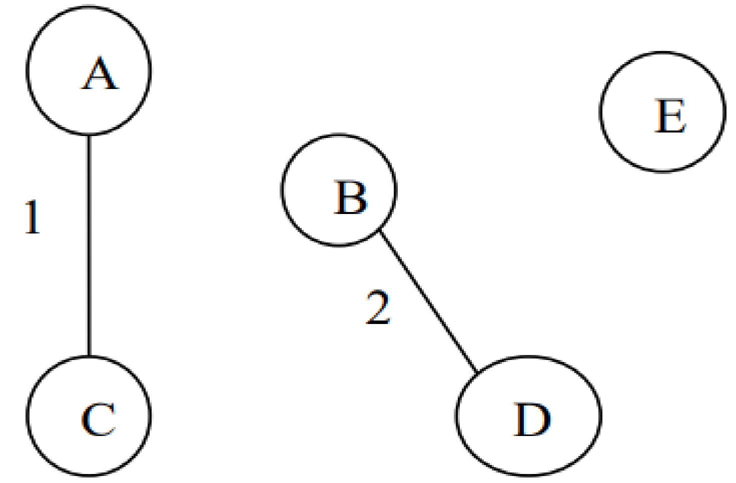


Q	Priority queue
{A}	1, 2
{B}	2, 2, 3, 3
{C}	1, 3, 4
{D}	2, 4, 5
{E}	3, 5

Round Robin Algorithm

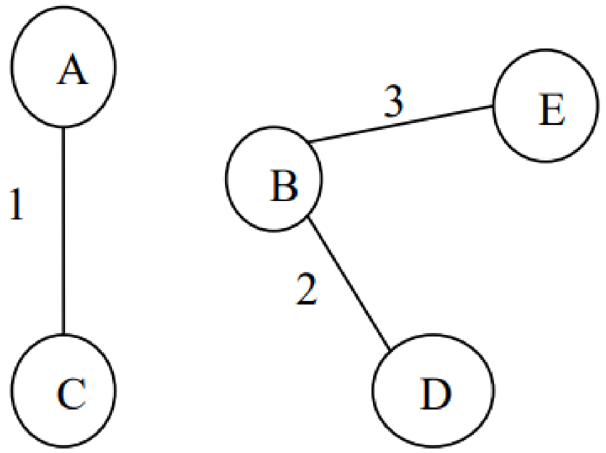


Q	Priority queue
{B}	2, 2, 3, 3
{D}	2, 4, 5
{E}	3, 5
{A, C}	2, 3, 4

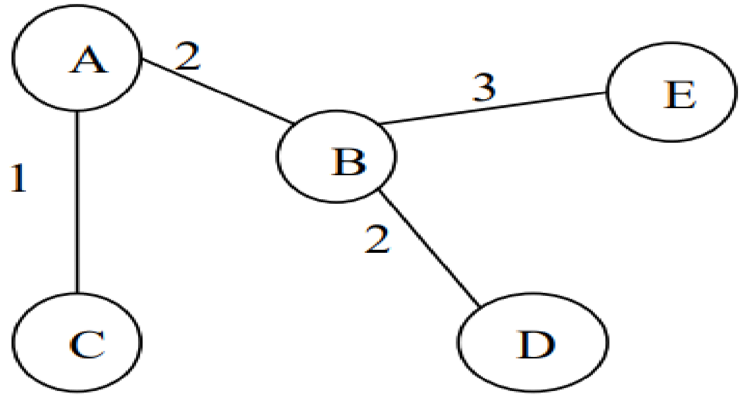


Q	Priority queue
{E}	3, 5
{A, C}	2, 3, 4
{B, D}	2, 3, 3, 4, 5

Round Robin Algorithm



Q	Priority queue
{A, C}	2, 3, 4
{E, B, D}	2, 3, 4, 5, 5



Q	Priority queue
{A, C, E, B, D}	3, 3, 4, 4, 5, 5

Only 1 partial tree is left in the queue, which is the required minimum spanning tree.

Round Robin Algorithm

$$\begin{tikzcd}
 & 1 \arrow[rr, "2"] \arrow[dr, "1"] & & 2 \arrow[dl, "3"'] \arrow[dr, "10"] & \\
3 \arrow[ur, "4"] \arrow[dr, "5"'] & & 4 \arrow[ll, "2"'] \arrow[rr, "2"] \arrow[dl, "8"] \arrow[dr, "4"'] & & 5 \arrow[dl, "6"] \\
 & 6 & & 7 \arrow[ll, "1"] &
\end{tikzcd}$$

Thank you!!!!!