Syllabus: Module 6

Representation of graphs, Topological sorting, Shortest path algorithms- Dijkstra's algorithm, Floyd-Warshall algorithm, Minimum spanning trees - Reverse delete algorithm, Boruvka’s algorithm.

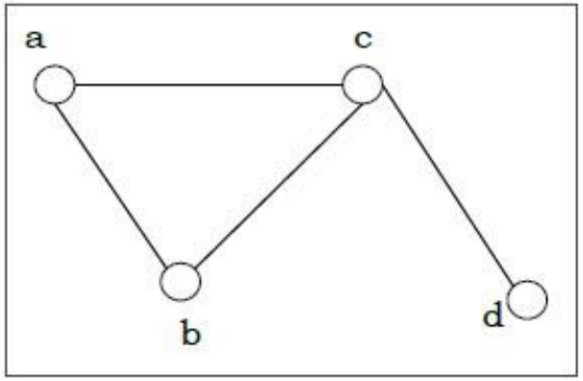
Graph

- Graph is a discrete structure.
- A graph is a set of points, called nodes or vertices, which are interconnected by a set of lines called edges.
- The study of graphs, or **graph theory** is an important part of a number of disciplines in the fields of mathematics, engineering and computer science.

Graph

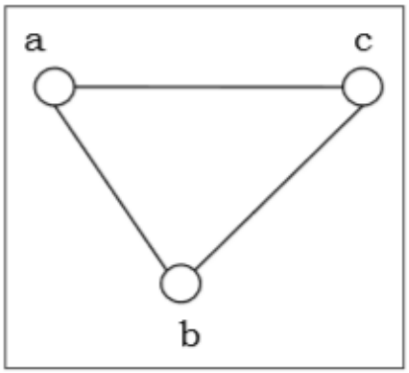
Definition − A graph (denoted as G=(V,E)) consists of a non-empty set of vertices or nodes V and a set of edges E.

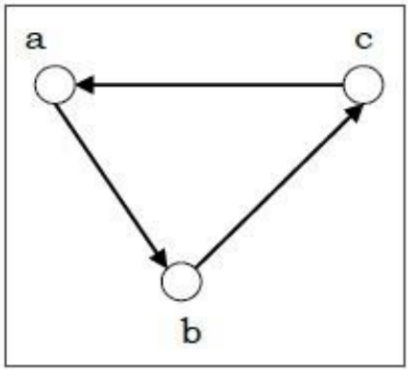
Example − Let us consider a Graph G=(V,E) where V={a,b,c,d}and E={{a,b},{a,c},{b,c},{c,d}}



Directed and Undirected Graph

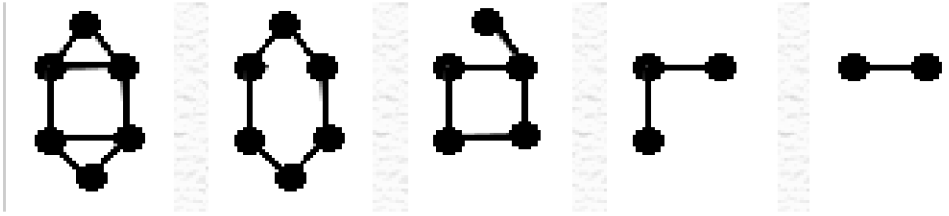
- A graph G=(V,E) is called a directed graph if the edge set is made of ordered vertex pair and a graph is called undirected if the edge set is made of unordered vertex pair.
- Edge has a direction in directed graph (digraph)





Subgraph

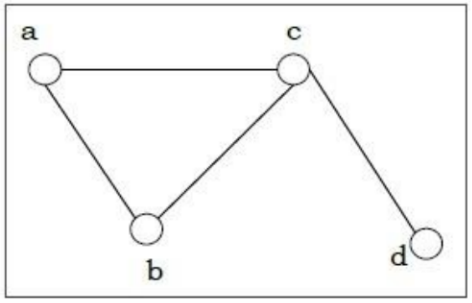
- A *subgraph S* of a graph *G* is a graph whose set of vertices and set of edges are all subsets of *G*. (Since every set is a subset of itself, every graph is a subgraph of itself.)
- All the edges and vertices of *G* might not be present in *S*; but if a vertex is present in *S*, it has a corresponding vertex in *G* and any edge that connects two vertices in *S* will also connect the corresponding vertices in *G*. All of these graphs are subgraphs of the first graph.

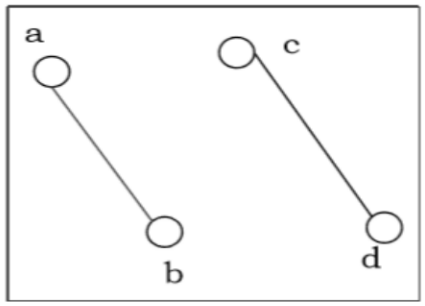


Graph models

Connected and Disconnected Graph

- A graph is connected if any two vertices of the graph are connected by a path;
- while a graph is disconnected if at least two vertices of the graph are not connected by a path. If a graph G is disconnected, then every maximal connected subgraph of G is called a connected component of the graph G.





Representation of graphs

Graph is represented using following ways:

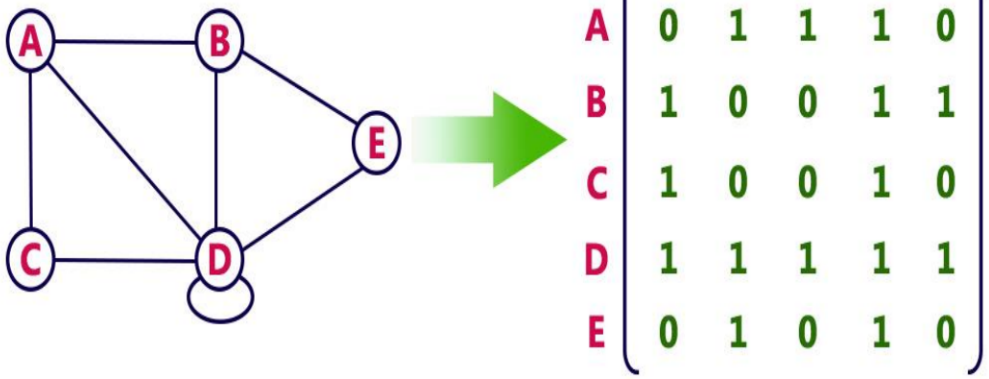
- Adjacency Matrix
- Incidence Matrix
- Adjacency List

Adjacency Matrix

- The graph is represented using a matrix of size total number of vertices by a total number of vertices.
- That means a graph with 4 vertices is represented using a matrix of size 4X4.
- In this matrix, both rows and columns represent vertices.
- This matrix is filled with either 1 or 0.
- Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

Adjacency Matrix

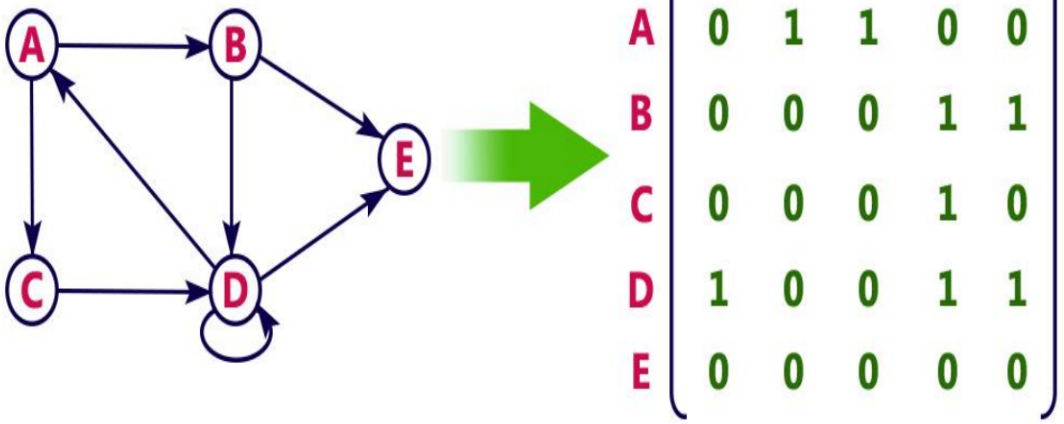
Undirected graph representation



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

Adjacency Matrix

Directed graph representation

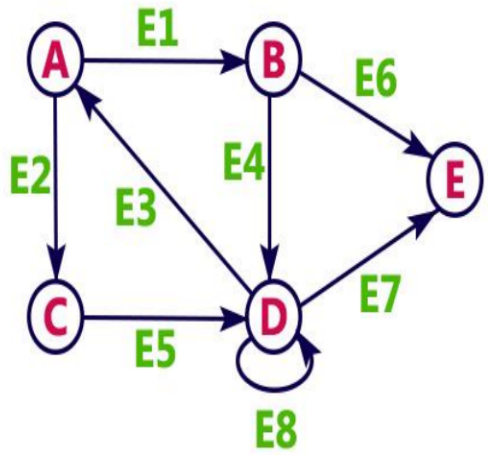


Incidence Matrix

- The graph is represented using a matrix of size total number of vertices by a total number of edges.
- That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6.
- In this matrix, rows represent vertices and columns represents edges.
- This matrix is filled with 0 or 1 or -1.
- Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

Incidence Matrix

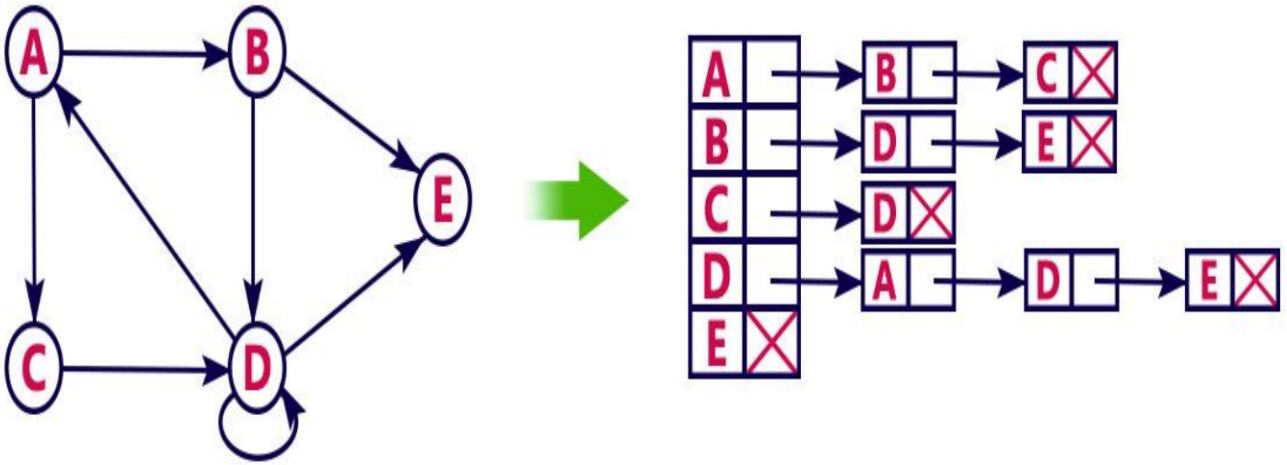
Directed graph representation



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

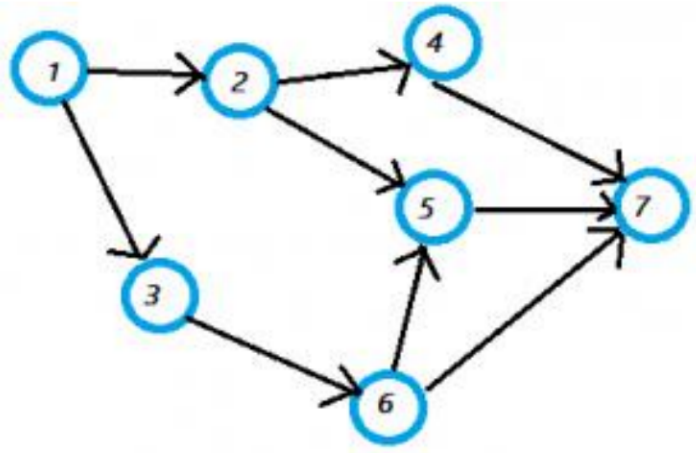
Adjacency List

- Every vertex of a graph contains list of its adjacent vertices.
- The following directed graph representation is implemented using linked list.



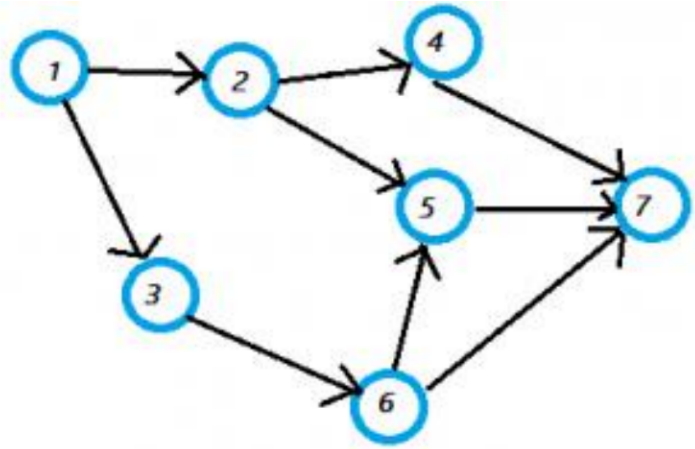
Directed Acyclic Graph (DAG)

- An acyclic graph is a graph without cycles (a cycle is a complete circuit).
- When following the graph from node to node, you will never visit the same node twice.



DAG

- A directed acyclic graph is an acyclic graph that has a direction as well as a lack of cycles.



Topological Ordering in DAG

The parts of the above graph are:

- **Vertices set** = {1,2,3,4,5,6,7}.
- **Edge set** = {(1,2), (1,3), (2,4), (2,5), (3,6), (4,7), (5,7), (6,7)}.
- A directed acyclic graph has a **topological ordering**.
- This means that the nodes are ordered so that the starting node has a lower value than the ending node.
- A DAG has a unique topological ordering if it has a directed path containing all the nodes; in this case the ordering is the same as the order in which the nodes appear in the path.

Topological Sort

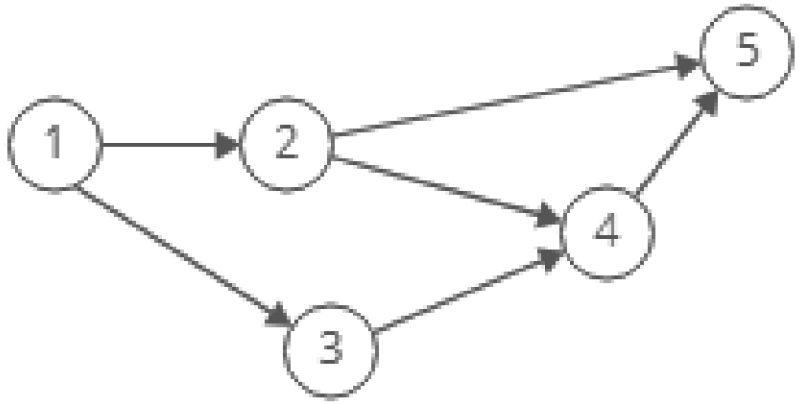
- Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.
- **Note:** Topological Sorting for a graph is not possible if the graph is not a DAG.
- Every DAG will have atleast one topological ordering, and there can be any number of topological ordering

Topological Sort

- The topological sort algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to.

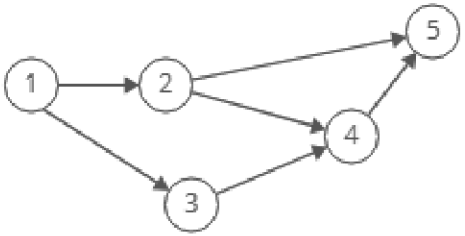
There can be more than one topological sorting for a graph.

Topological Sort



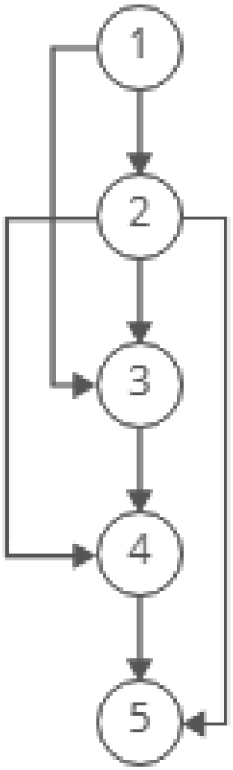
Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

Topological Sort



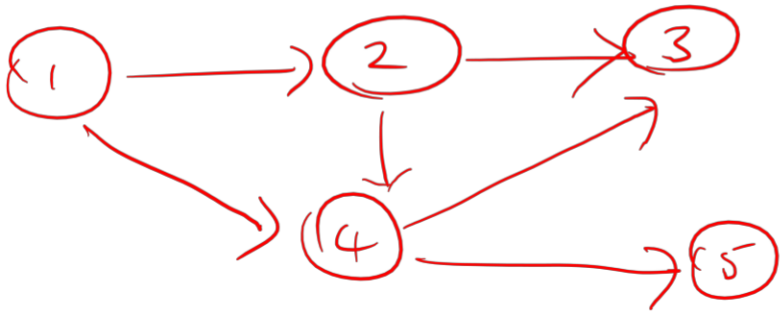
So [1, 2, 3, 4, 5] would be a topological ordering of the graph.

[1, 3, 2, 4, 5] works too.

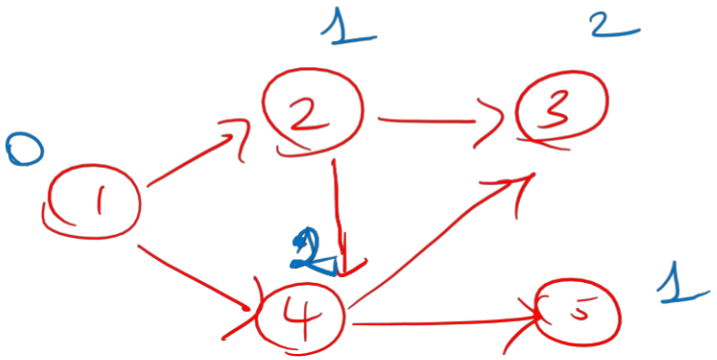


Topological Sort

Example :-



Write the IN DEGREE for every node



Topological Sort

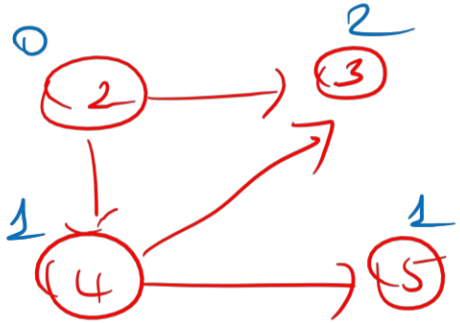
Take the node with the least indegree.

It is ①

Topological sort :

1

Remove ① from the graph along with its Edges & update the indegree.



Topological Sort

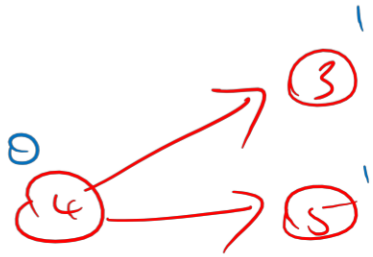
Take the node with least indegree.

Here it is (2)

Topological sort

1, 2

Remove (2) from the graph & update indegree



Topological Sort

Topological sort

1, 2, 4

So, topological sort is :

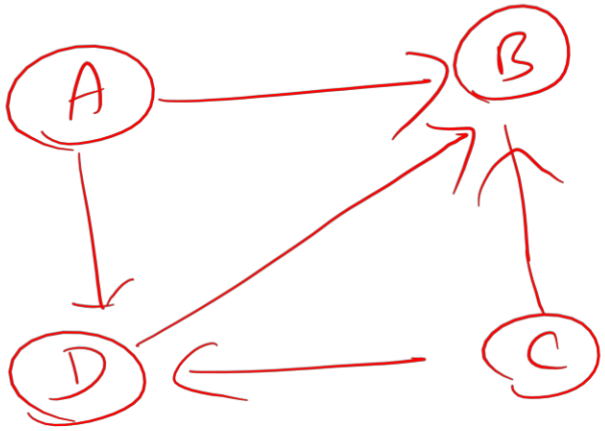
1, 2, 4, 3, 5

(or)

1, 2, 4, 5, 3

Topological Sort

Exercise :



Application

- Task scheduling
- Prerequisite problems

Shortest path algorithms

Shortest path algorithms

Shortest path problem:-

- This problem of a graph is about finding a path between two vertices in such a way that this path will satisfy some criteria of optimization.

E.g:

- For non-weighted graph, the number of edges will be minimum
- For a weighted graph, the sum of weights on all its edges in the path will be minimum.

Shortest path algorithms

Shortest path problem:-

- Dijkstra's algorithm
- Floyd –Warshall's algorithm

Dijkstra's Algorithm: (Single source shortest path problem)

- Here, there is a distinct vertex, called the source vertex and it requires to find the shortest path from this source vertex to all other vertices.
- It is a ***single- source shortest path problem:***
  - for a given vertex called the source in a weighted connected graph, find shortest path to all its other vertices.
  - The best-known algorithm for the single – source shortest path problem called Dijkstra's algorithm.

Dijkstra's Algorithm: (Single source shortest path problem)

- Dijkstra's algorithm finds shortest path to a graph's vertices in order of their distance from a given source.
- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest and so on.
- Dijkstra's algorithm compares path length and therefore must add edge weights.

Dijkstra's algorithm

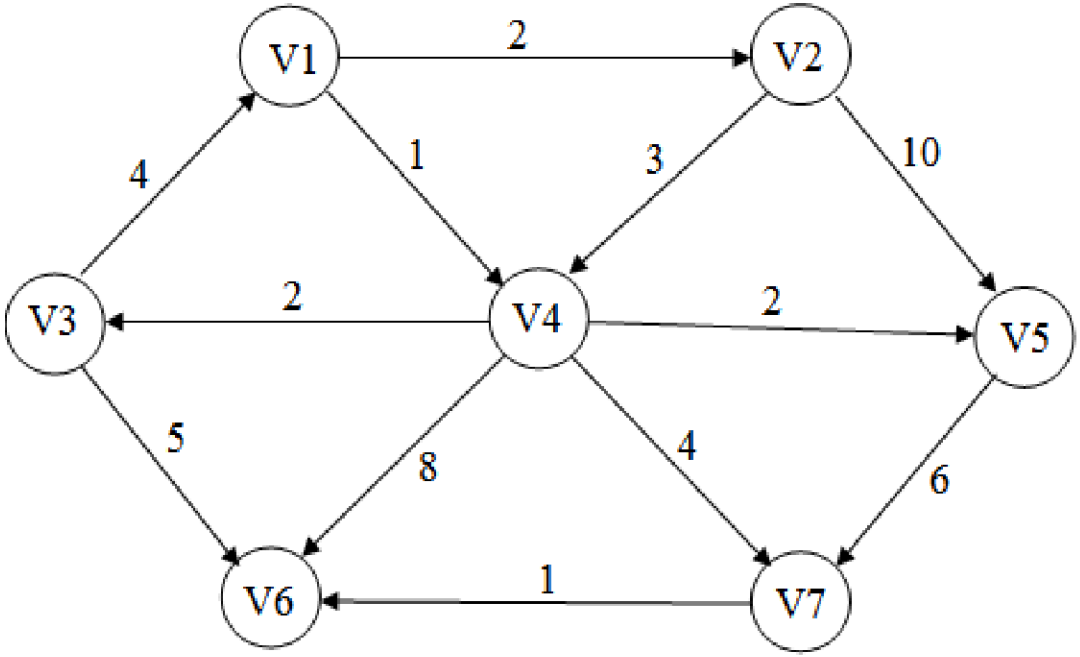
```
Function Dijkstra(Graph, source):
    dist[source] = 0                    // Distance from source to source is set to 0
    for each vertex v in Graph:         // Initializations
        if v = = source
            dist[v] = infinity          // Unknown distance function from source to each node set to infinity
        add v to Q                      // All nodes initially in Q

    while Q is not empty:
        v = vertex in Q with min dist[v]  // In the first run-through, this vertex is the source node
        remove v from Q

        for each neighbor u of v:       // where neighbor u has not yet been removed from Q.
            alt = dist[v] + length(v, u)
            if alt < dist[u]            // A shorter path to u has been found
                dist[u] = alt           // Update distance of u
    return dist[]
end function
```

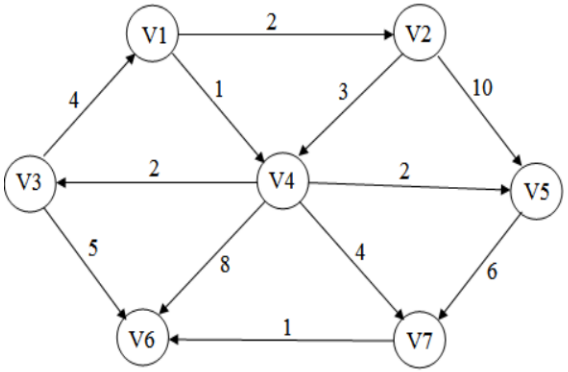
Example

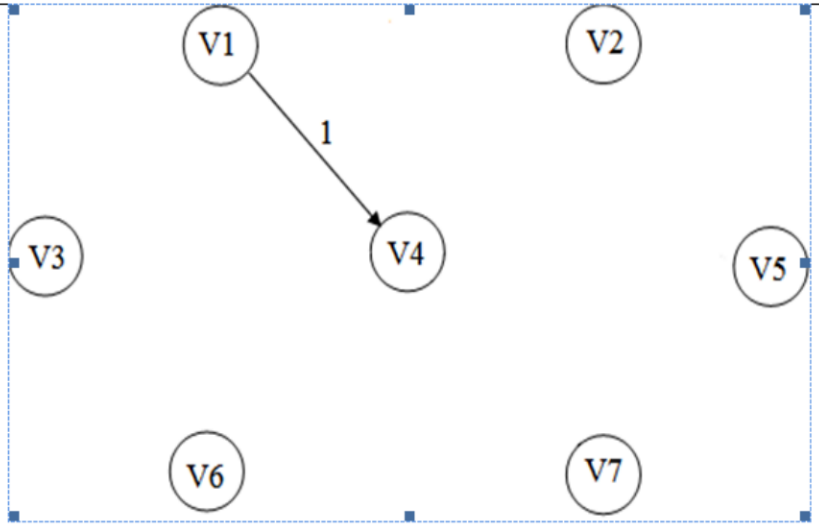
Consider the graph:



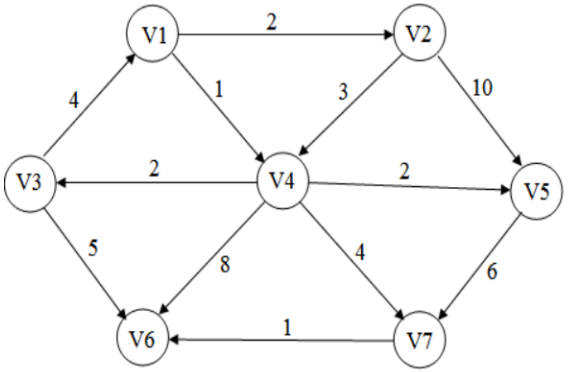
Start/source vertex is V1

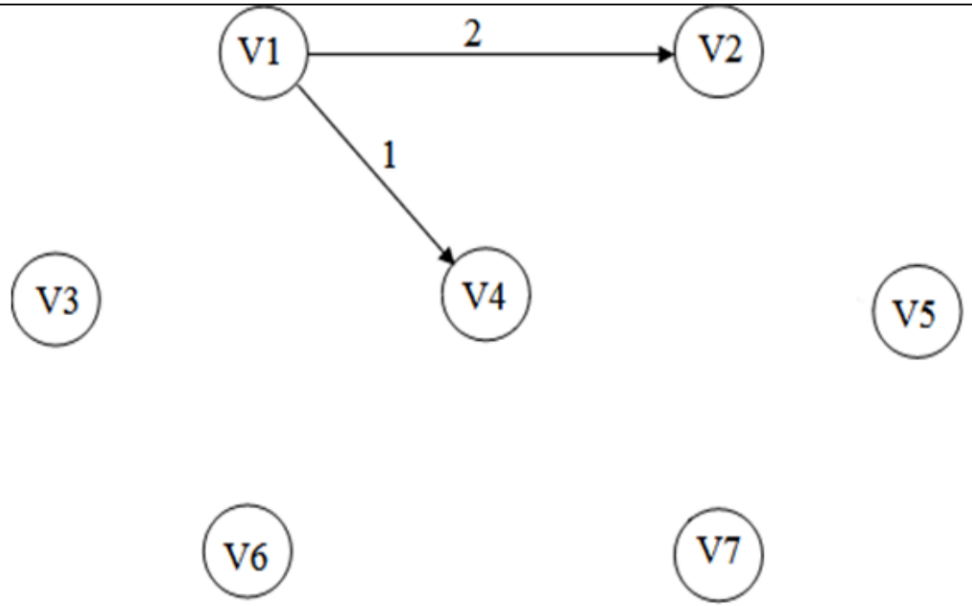
Example



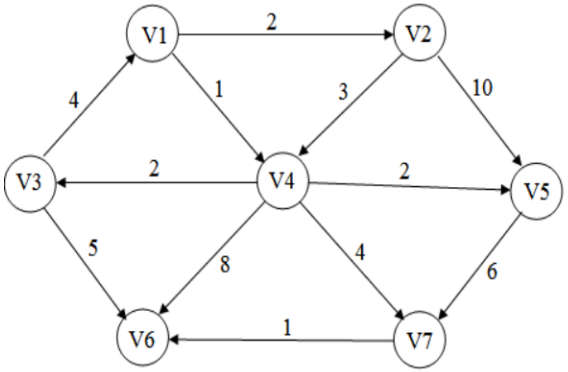
Tree vertices	Remaining vertices	Illustration
V1(- ,-)	V2 (V1, 2), V3 (- , ∞), **V4 (V1, 1),** V5(- , ∞), V6(- , ∞) , V7(- , ∞).	

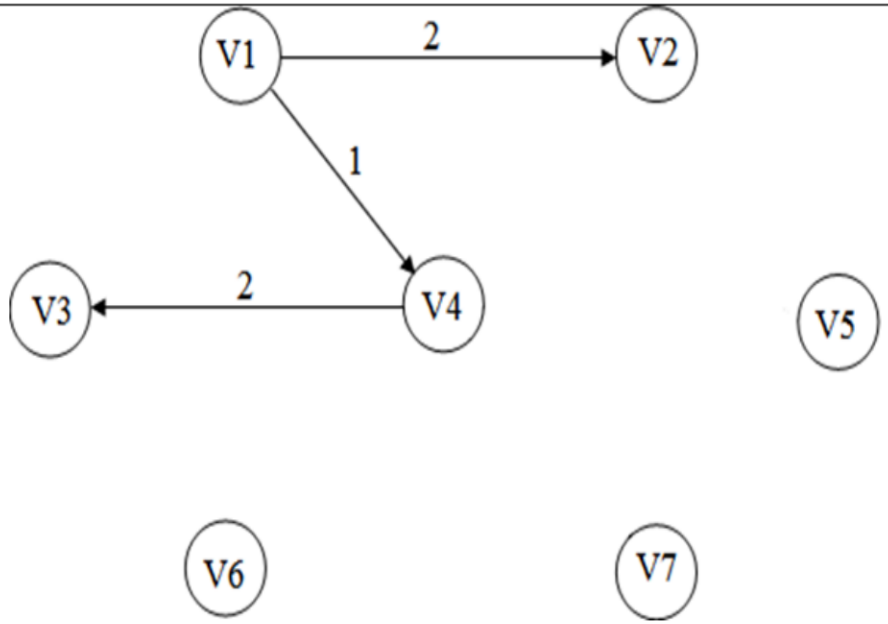
Example



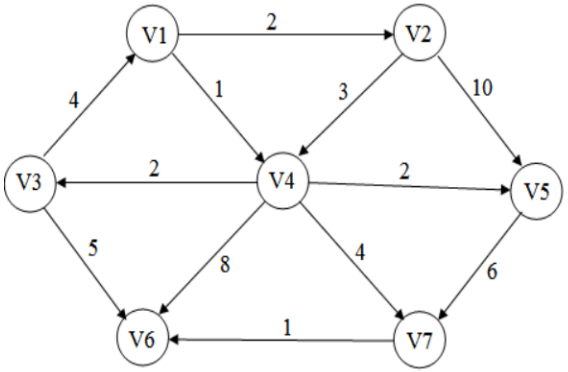
V4(V1 , 1)	V2(V1 , 2), V3 (V4 ,2+1), V5(V4 , 2+1) , V6(V4 ,8+1) V7(V4 , 1+4)	

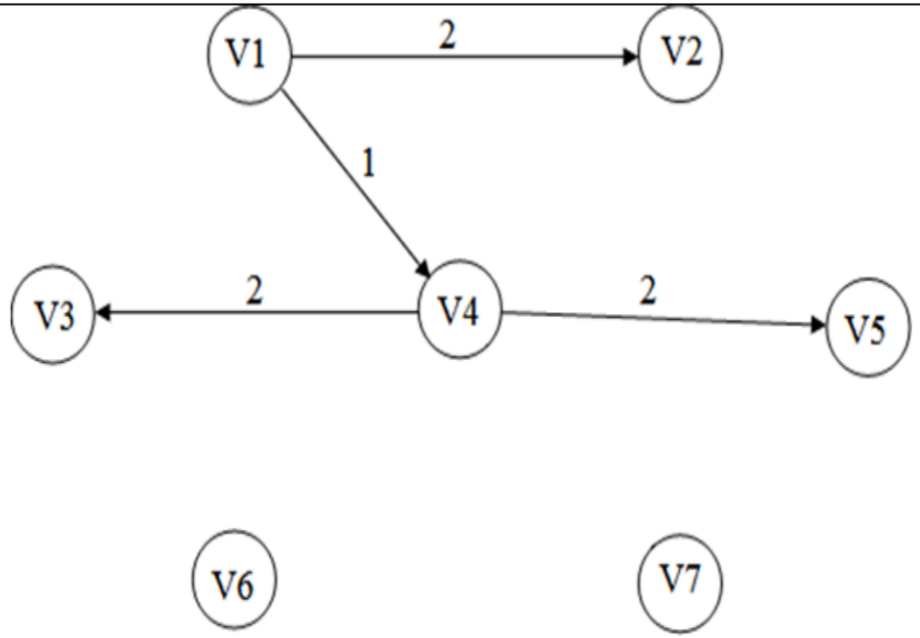
Example



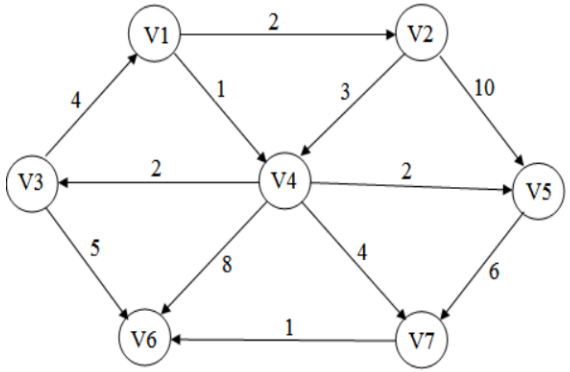
V2(V1 ,2)	V3 (V4 ,2+1), V5(V4 , 2+1), V6(V4 ,8+1) V7(V4 , 1+4)	

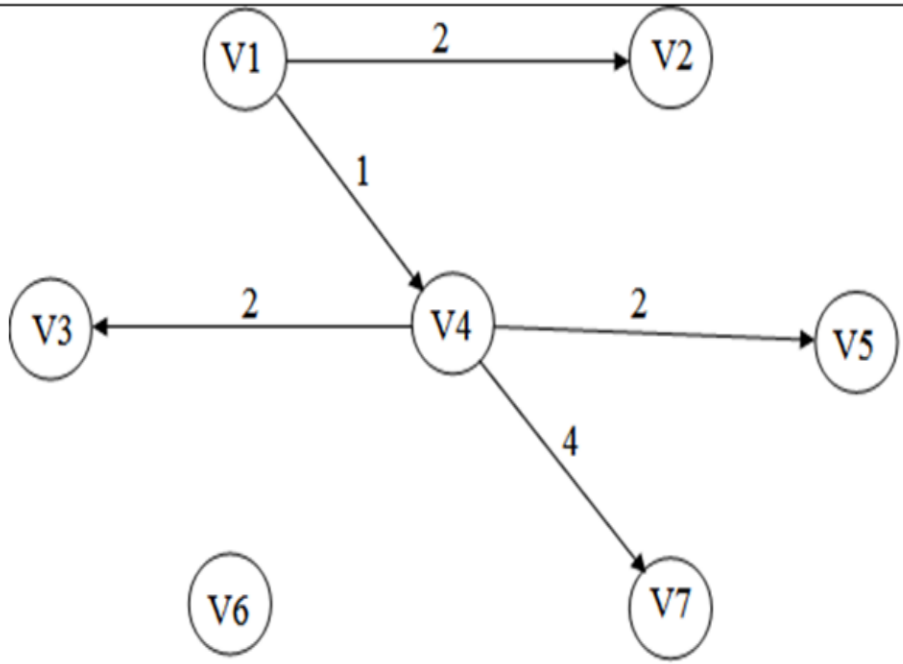
Example



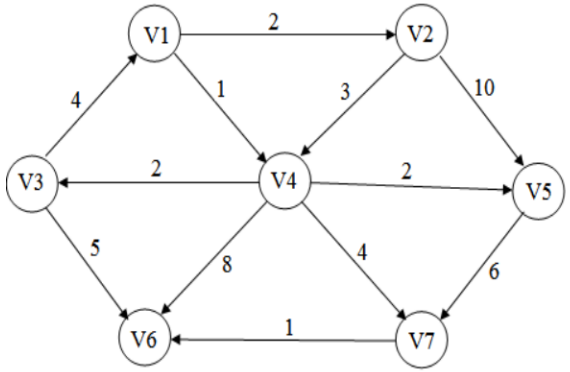
V3(V4 , 3)	V5(V4, 2+1) V6(V3, 1+2+5) V7(V4, 1+4)	

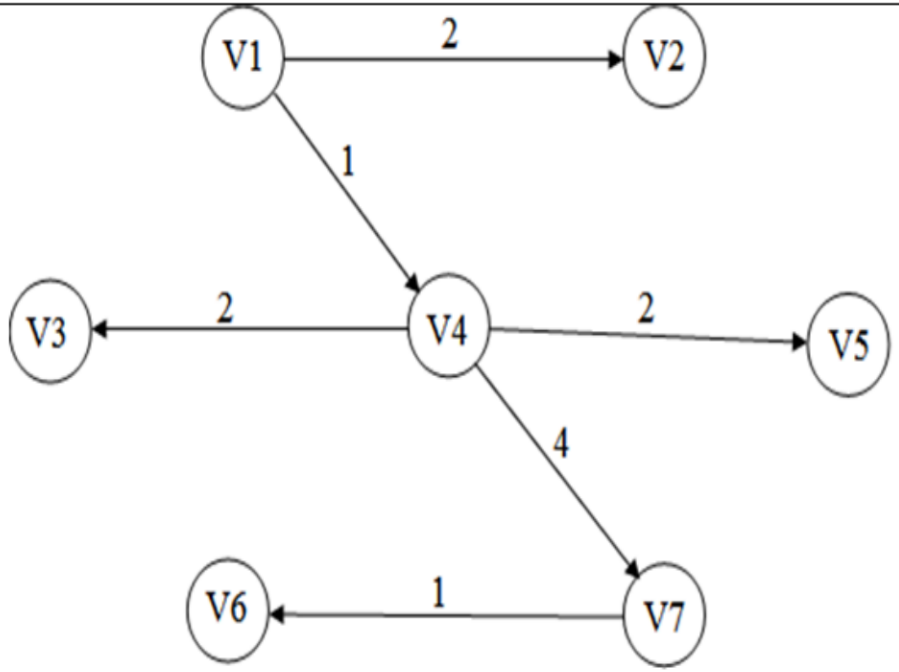
Example



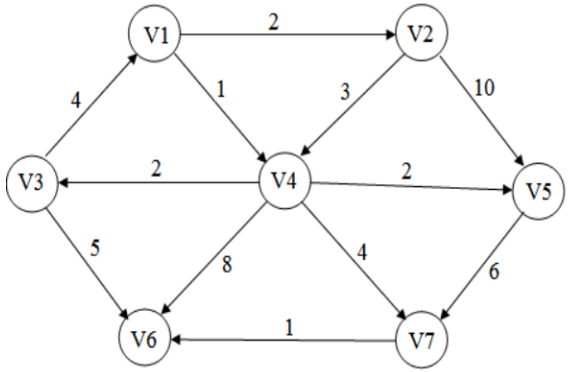
V5(V4 , 3)	V6(V3, 1+2+5), V7(V4, 1+4)	

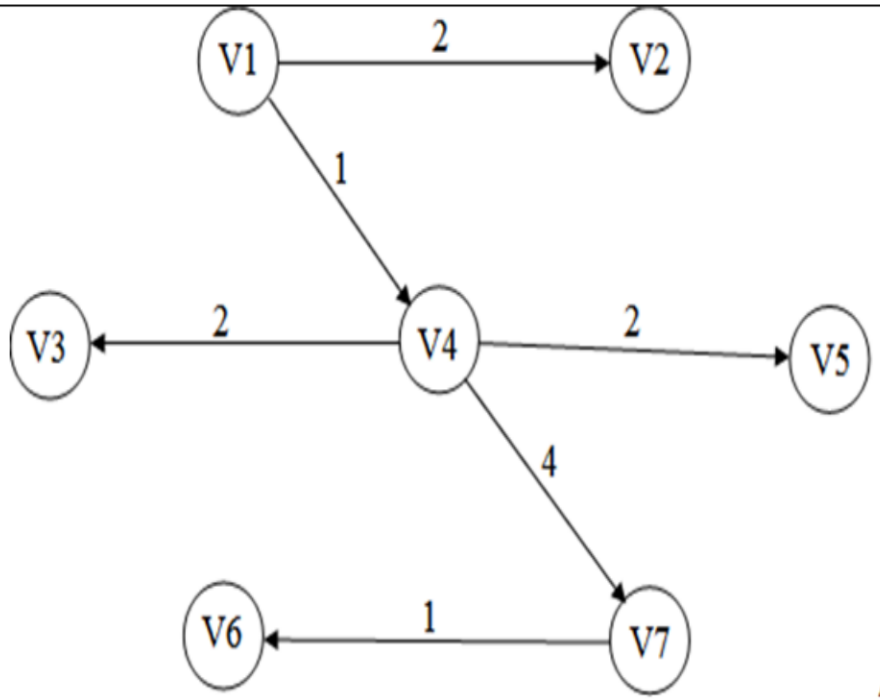
Example



V7(V4,5)	V6(V7, 1+4+1)	

Example



V6(V7,6)		

//…

Time Complexity of Dijkstra's Algorithm

Case-01:

This case is valid when-

- The given graph G is represented as an adjacency matrix.
- Priority queue Q is represented as an unordered list.

Here,

- A[i,j] stores the information about edge (i,j).
- Time taken for selecting i with the smallest dist is O(V).
- For each neighbor of i, time taken for updating dist[j] is O(1) and there will be maximum V neighbors.
- Time taken for each iteration of the loop is O(V) and one vertex is deleted from Q.
- Thus, total time complexity becomes $O(V^2)$.

Time Complexity of Dijkstra's Algorithm

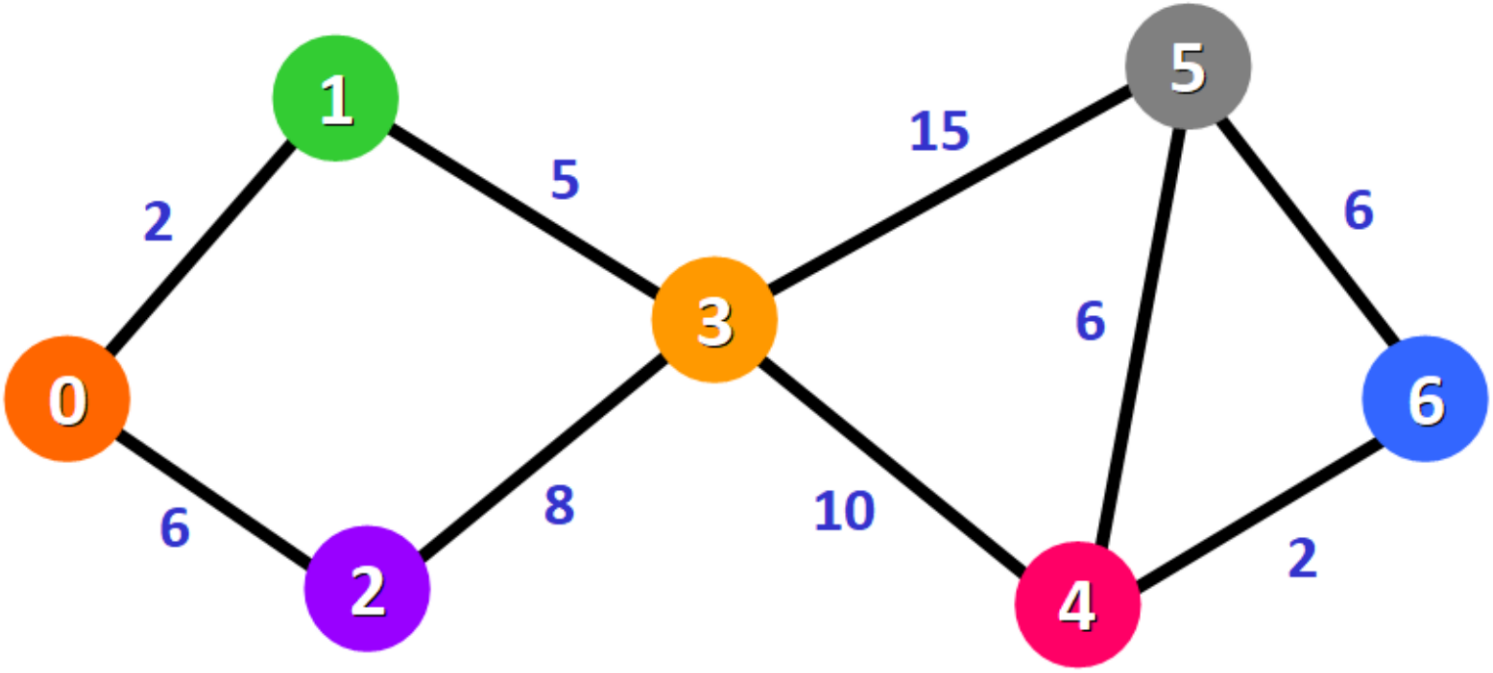
Case-02:

- This case is valid when-
  - The given graph G is represented as an adjacency list.
  - Priority queue Q is represented as a binary heap.

Here,

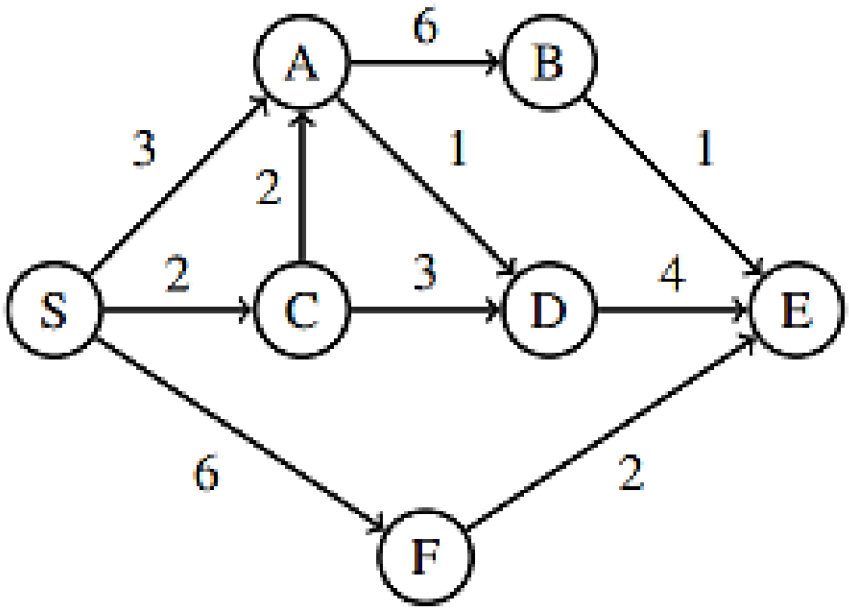
- With adjacency list representation, all vertices of the graph can be traversed using BFS in O(V+E) time.
- In min heap, operations like extract-min and decrease-key value takes O(logV) time.
- So, overall time complexity becomes O(E+V) x O(logV) which is O((E + V) x logV) = O(ElogV)

Exercise-Dijkstra's Algorithm on undirected graph



Start/source vertex is 1

Exercise-Dijkstra's Algorithm-directed graph



Start/source vertex is A

Floyd-Warshall Algorithm-Shorest path algorithm

- Floyd-Warshall Algorithm is an algorithm for solving All Pairs Shortest path problem which gives the shortest path between every pair of vertices of the given graph.
- The main advantage of Floyd-Warshall Algorithm is that it is extremely simple and easy to implement.

Floyd-Warshall Algorithm

Pseudocode:

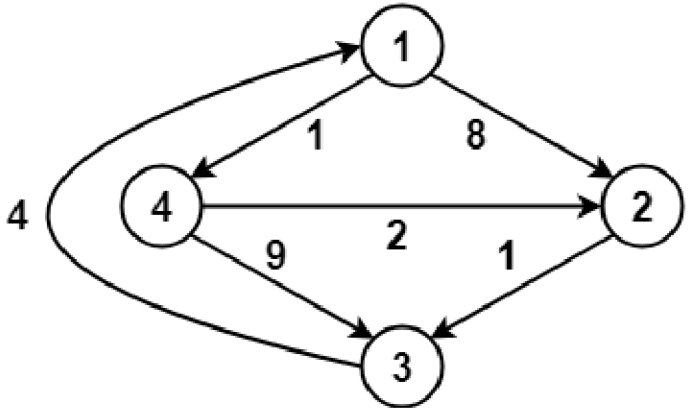
```
Create a |V| x |V| matrix  M       // It represents the distance between every pair of
                                   vertices as given
for each cell (i,j) in M do
      if i = = j
            M[ i ][ j ] = 0              // For all diagonal elements, value = 0
      if (i , j) is an edge in E
            M[ i ][ j ] = weight(i,j)    // If there exists a direct edge between the vertices,
                                                 value = weight of edge
      else
            M[ i ][ j ] = infinity       // If there is no direct edge between the vertices,
                                                 value = ∞
for k from 1 to |V|
      for i from 1 to |V|
            for j from 1 to |V|
                  if M[ i ][ j ] > M[ i ][ k ] + M[ k ][ j ]
                        M[ i ][ j ] = M[ i ][ k ] + M[ k ][ j ]
```

Time Complexity

- Floyd Warshall Algorithm consists of three loops over all nodes.
- Hence, the asymptotic complexity of Floyd-Warshall algorithm is $O(n^3)$, where n is the number of nodes in the given graph.

Problem

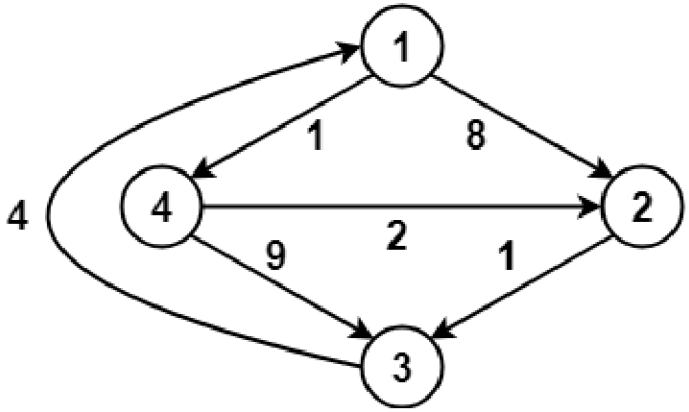
- Consider the following directed weighted graph
- Using Floyd-Warshall Algorithm, find the shortest path distance between every pair of vertices.



Solution

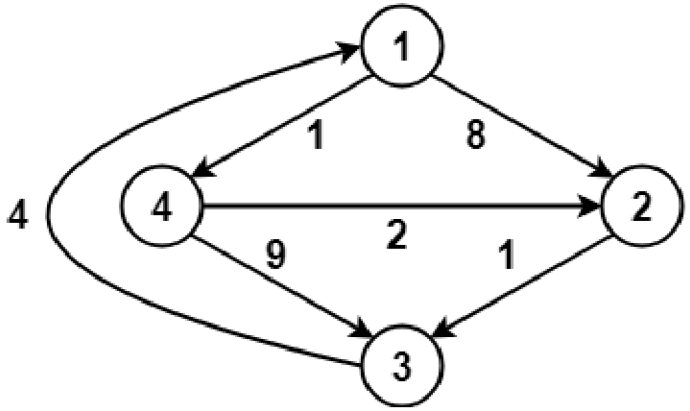
Step-01:

- Remove all the self loops and parallel edges (keeping the edge with lowest weight) from the graph if any.
- In our case, we don't have any self edge and parallel edge.



Step-02:

- Now, write the initial distance matrix representing the distance between every pair of vertices as mentioned in the given graph in the form of weights.
- For diagonal elements (representing self-loops), value = 0
- For vertices having a direct edge between them, value = weight of that edge
- For vertices having no direct edges between them, value = ∞



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

Step-03:

- From step-03, we will start our actual solution.

NOTE

- Since, we have total 4 vertices in our given graph, we will have total 4 matrices of order 4 x 4 in our solution. (excluding initial distance matrix)

The four matrices are-

- In D_1, write the first row and first column of the D_0 matrix.
- Next, find the row and column which are having ∞, and fill the values as it is in D_0.
- Then, sum up the corresponding row and column value and enter it in place of ∞
- In the remaining cells, If higher value is present, it can be replaced with a lower value, else leave as it is
- Diagonals always have 0, don't update it

$D_0 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	∞	0	∞
4	∞	2	9	0

$D_1 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

- In D_2, copy the second row and second column of D_1 matrix.
- Next, find the row and column which are having ∞, and fill the values as it is in D_1.
- Then, sum up the corresponding row an column value and enter it in ∞ places. In the remaining cells, If higher value is present, it can be replaced with a lower value

$D_1 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

$D_2 =$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

- In D3, copy the third row and third column of D_2 matrix.
- Next, find the row and column which are having ∞, and fill the values as it is in D_2.
- But, here no ∞, hence fill the diagonals with 0.
- If we add 12+9, we get 21 to put in (1,2) position, but this 21 is greater than 8 in that position in the previous matrix D_2, hence it is not updated and the old entry is maintained.
- similarly do for other entries.

Similarly do for other matrices.

$D_2 =$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

$D_3 =$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

$D_1 =$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

$D_2 =$

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

$D_3 =$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

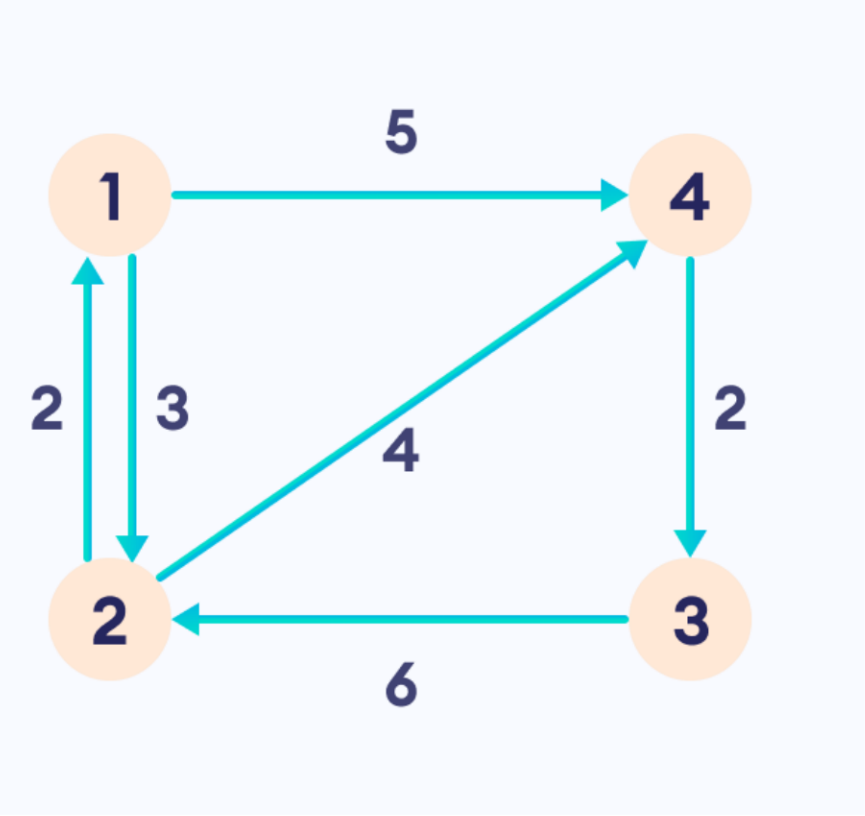
$D_4 =$

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0

- The last matrix D_4 represents the shortest path distance between every pair of vertices.
- Diagonals always contain 0, so don't update.

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

Exercise- Floyd-Warshall Algorithm



Minimum Spanning Tree

Spanning Tree

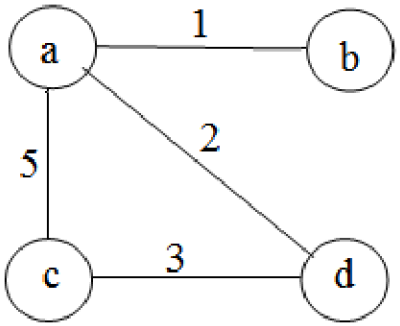
- A ***Spanning Tree*** of a connected graph is its connected acyclic sub graph that contains all the vertices of the graph

Spanning Tree

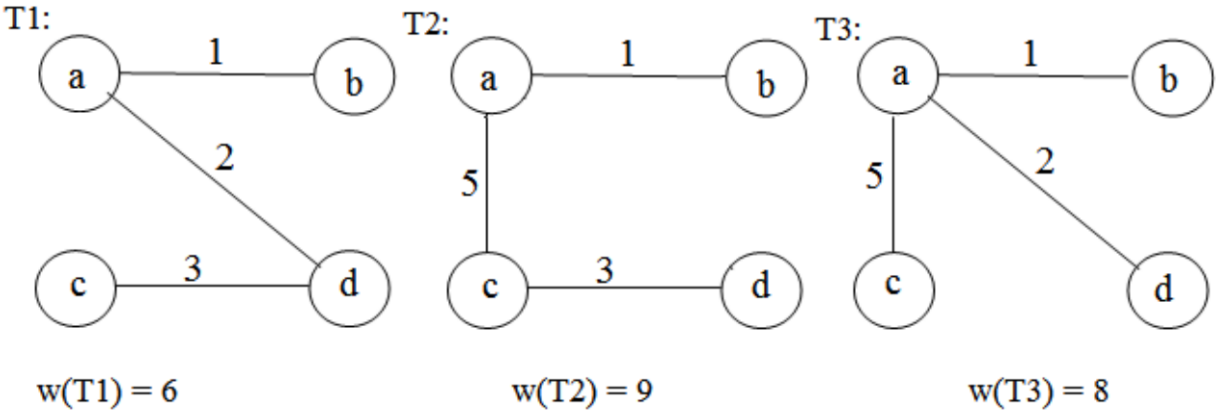
- A ***Minimum Spanning Tree*** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.
- The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Example

Consider a graph:



The spanning Tree is:



Here T1 is the minimum spanning tree.

Drawback of Exhaustive Search Approach

- The number of spanning trees grows exponentially with the graph size.
- Generating all spanning trees for a given graph is not easy.
- To overcome this drawback, we make use of some efficient algorithms such as
  - Prim's algorithm
  - Kruskal's algorithm

Reverse delete algorithm

- Reverse Delete Algorithm is for Minimum Spanning Tree

- Reverse Delete algorithm is closely related to Kruskal's algorithm.

Reverse delete algorithm

- Kruskal's algorithm :
  - Sort edges by **increasing order** of their weights. After sorting, one by one pick edges in increasing order. We include current picked edge if including this in spanning tree does not form any cycle. We repeat it until there are V-1 edges in spanning tree, where V = number of vertices.
- Reverse Delete algorithm:
  - Sort all edges in **decreasing** order of their weights. After sorting, we one by one pick edges in decreasing order. We **include current picked edge if excluding current edge causes disconnection in current graph**. The main idea is delete an edge if it's deletion does not lead to disconnection of graph.

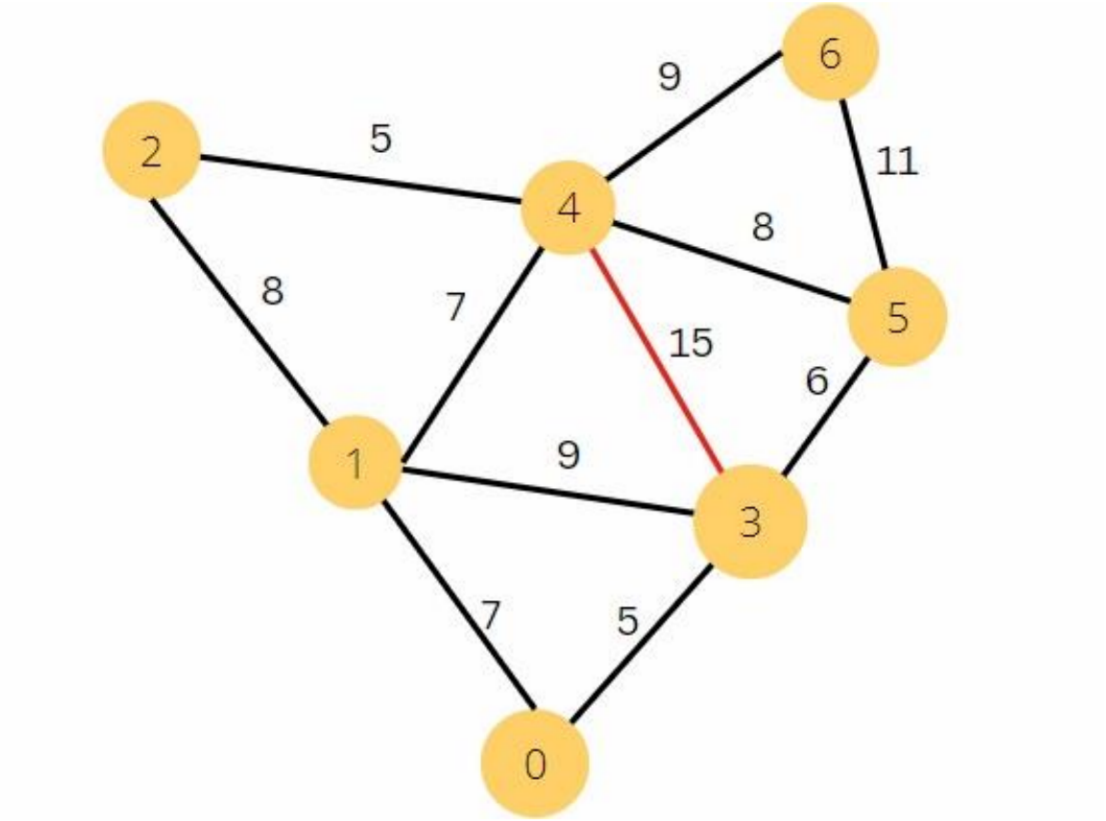
Reverse delete algorithm

Algorithm:

- By weight, order the edges
- Set up MST with all edges.
- The highest weight edge should be removed.
- Restore the edge if removing it causes the graph to become disconnected.
- Otherwise, keep going

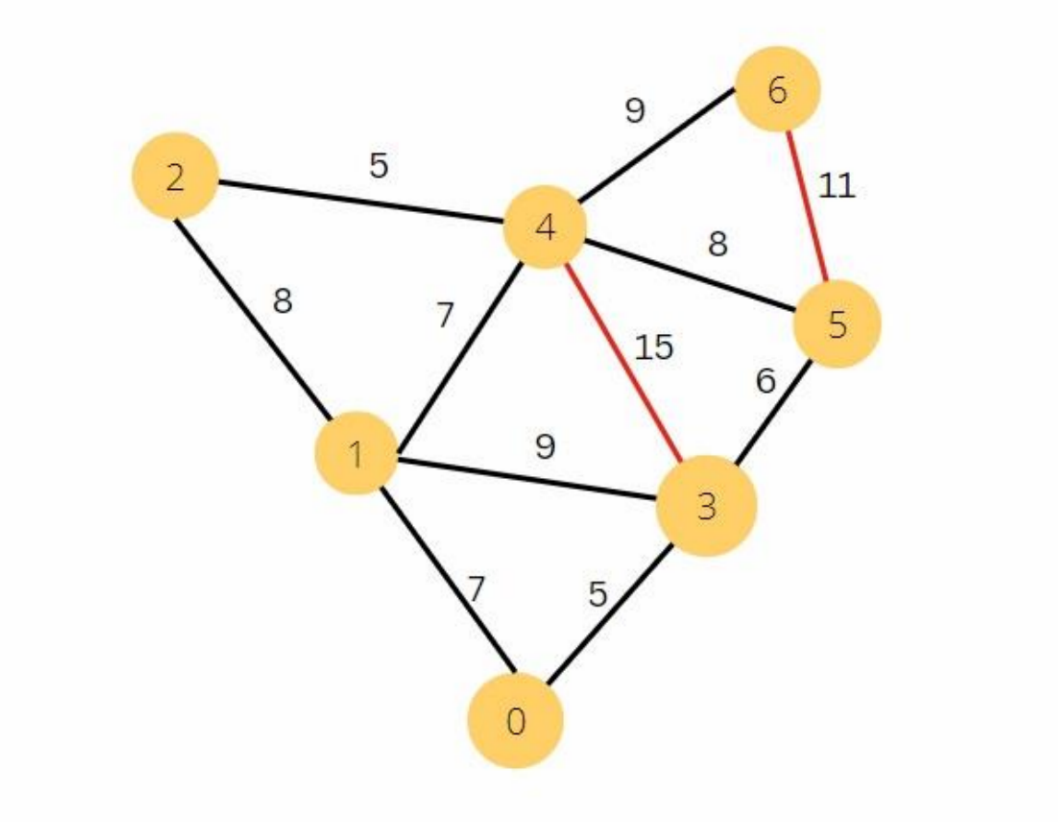
Reverse delete algorithm

•We begin with Edges 3 to 4, which have the highest weight.
•Since removing edges 3 and 4 does not cause the graph to become disconnected, the edge may be deleted.



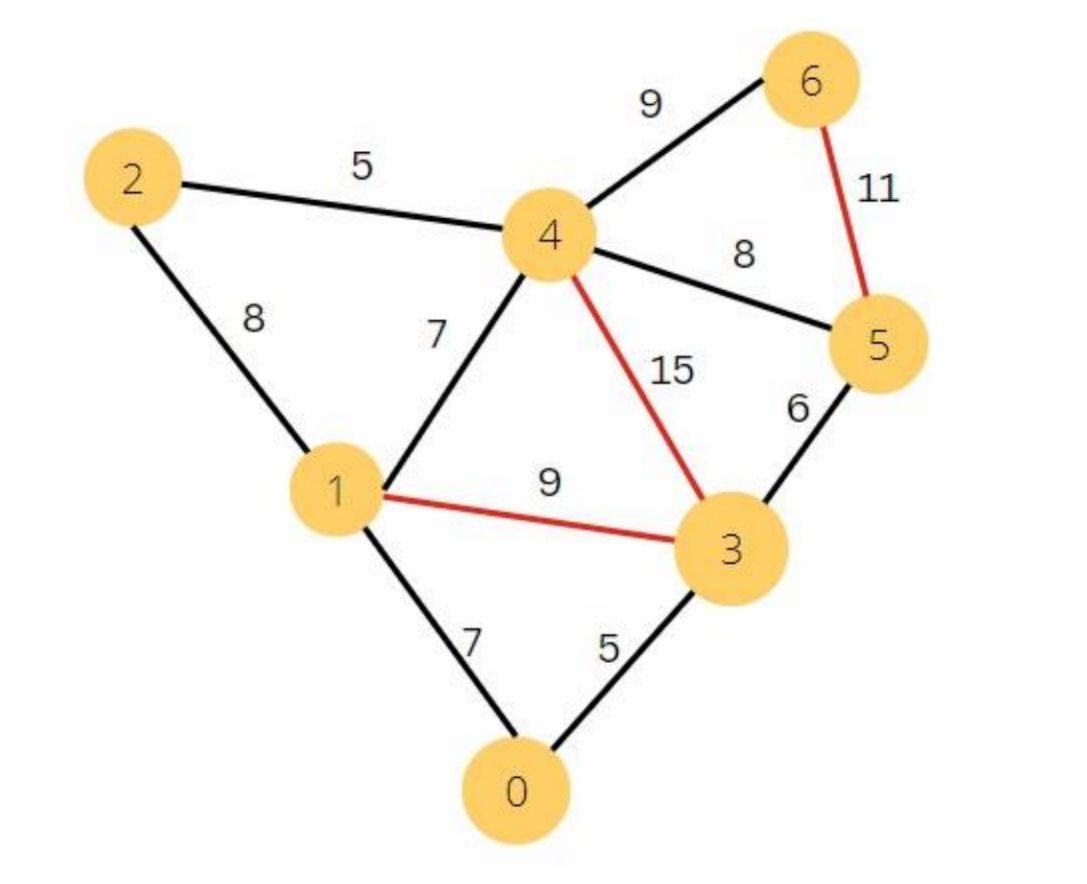
Reverse delete algorithm

•Pick the edge 5-6 with weight 11 next. Since removing edges 5 and 6 does not cause the graph to become disconnected, the edge may be deleted.



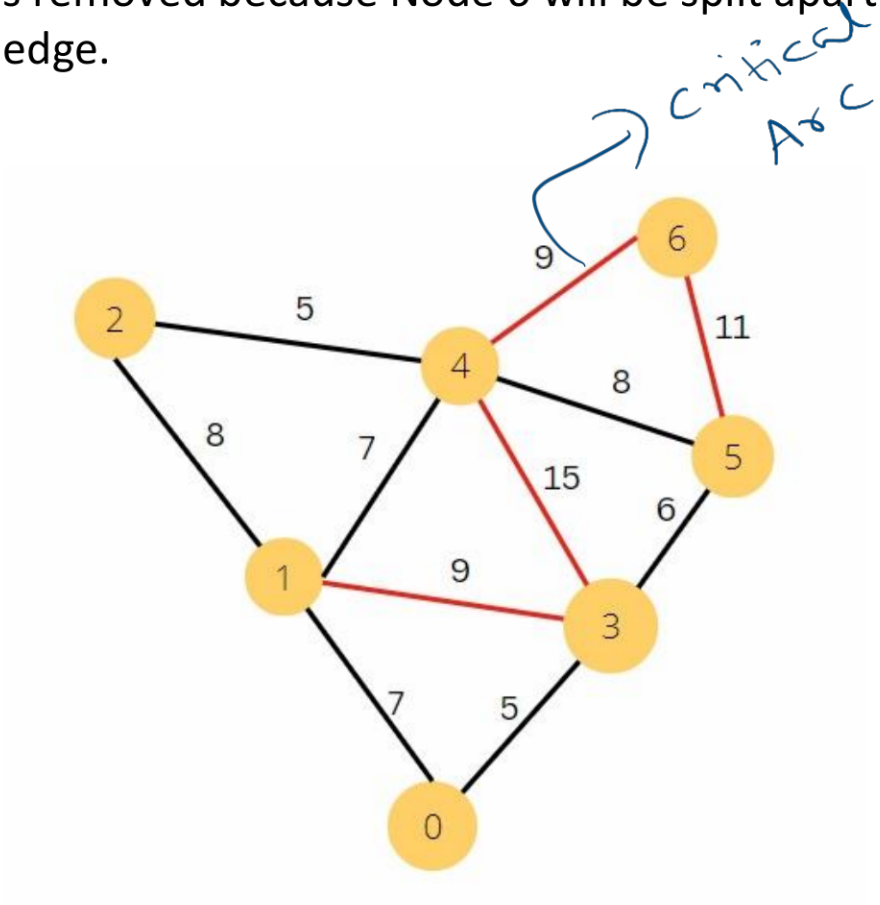
Reverse delete algorithm

•Pick the 1-3 edge with weight 9. Since removing edges 1-3 does not cause the graph to become disconnected, the edge may be deleted.



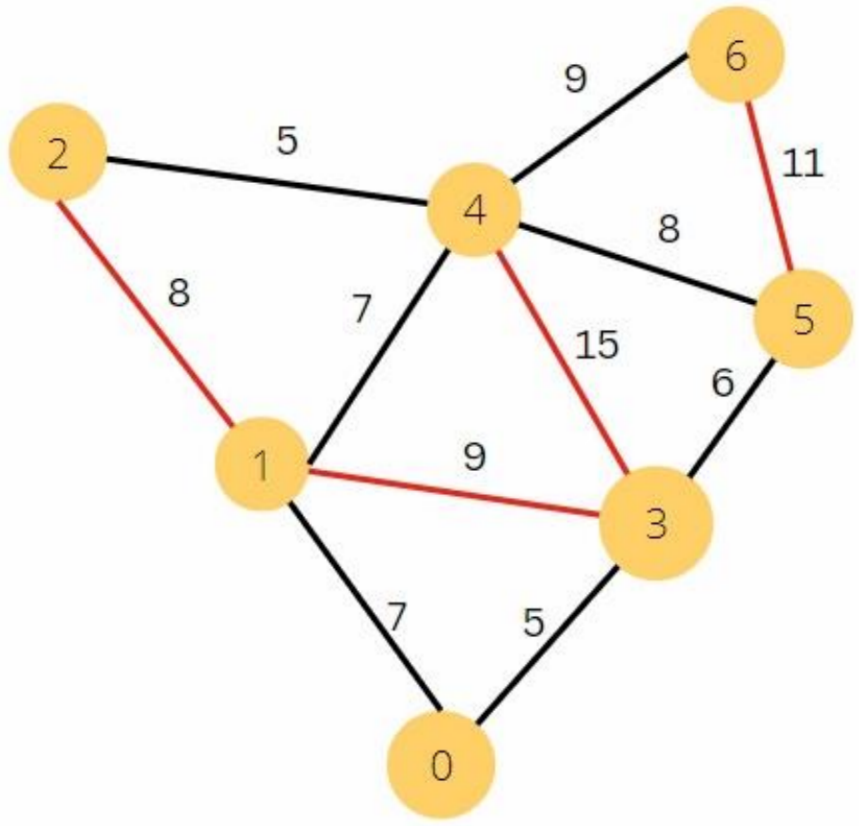
Reverse delete algorithm

•Choose the edge 4-6 with weight nine next. The graph will become disconnected if this edge is removed because Node 6 will be split apart. We, therefore, do not remove the edge.

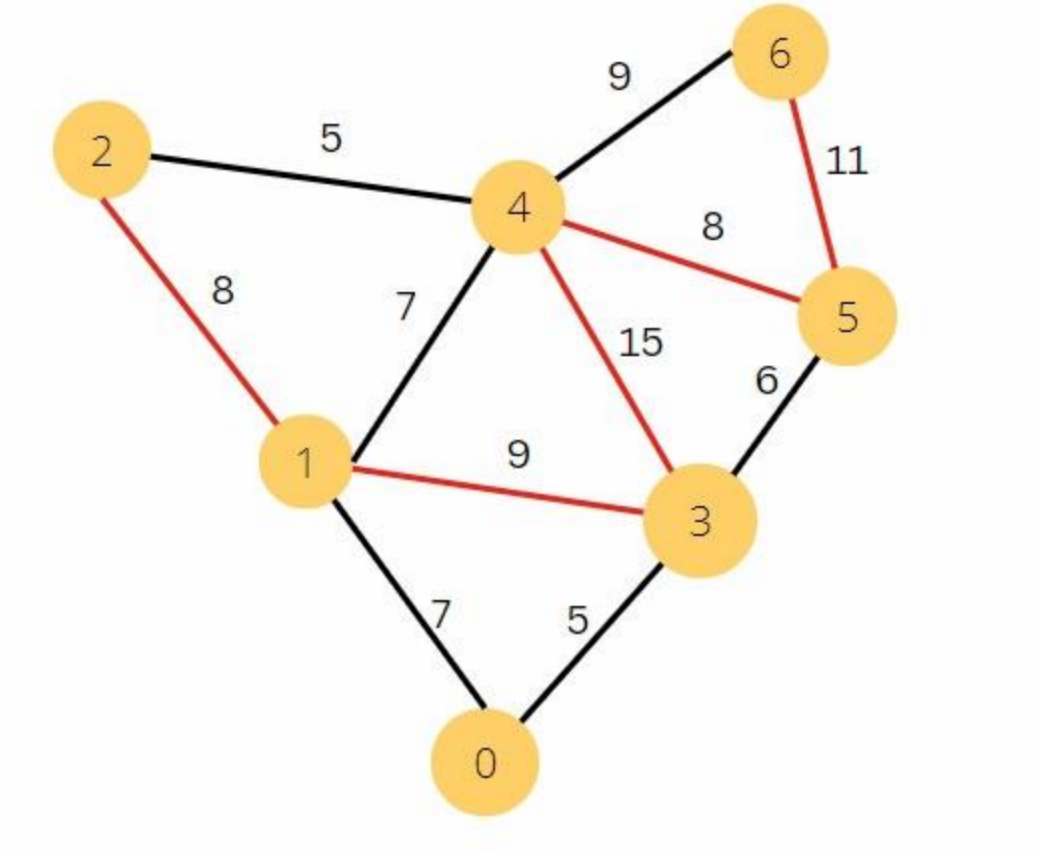


Reverse delete algorithm

•Choose edge 2 with weight 8 next. Since removing edges 1-2 does not cause the graph to become disconnected, the edge may be deleted.

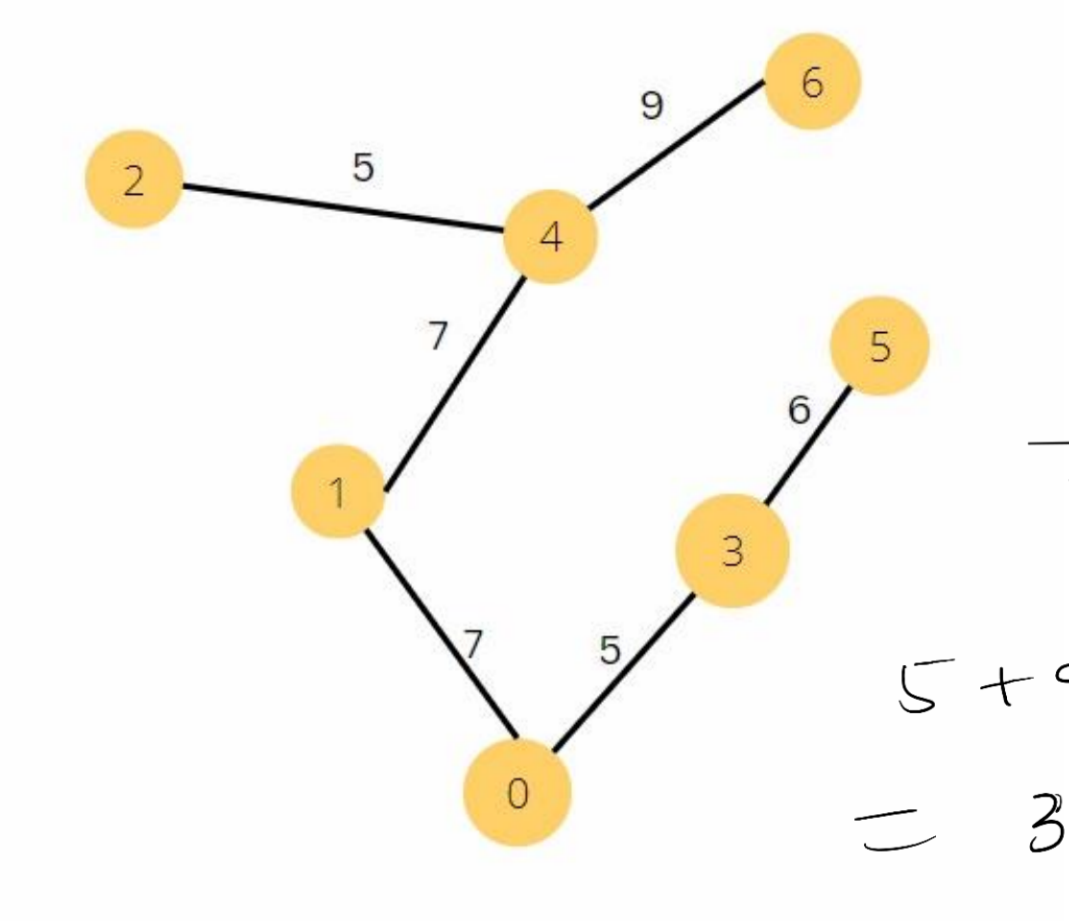


Reverse delete algorithm



Reverse delete algorithm

•This graph's minimum spanning tree is:



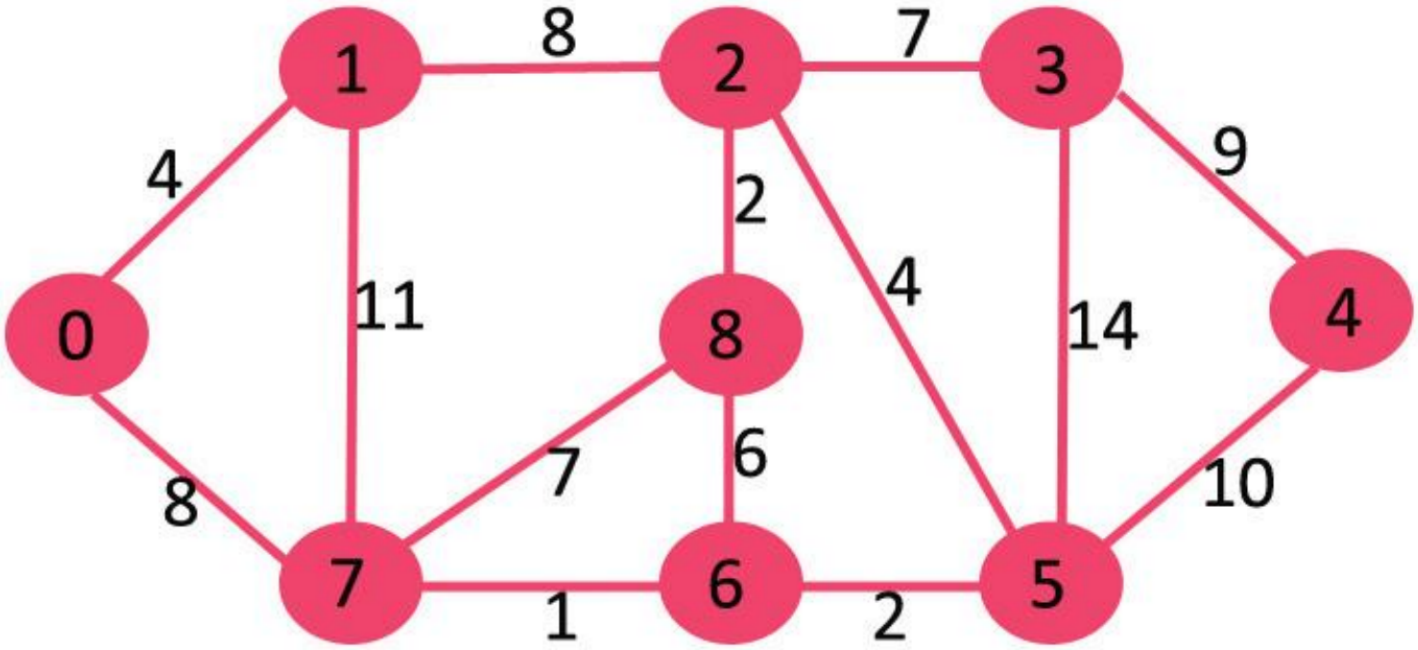
The weight of MST is

5 + 9 + 7 + 7 + 5 + 6

= 39

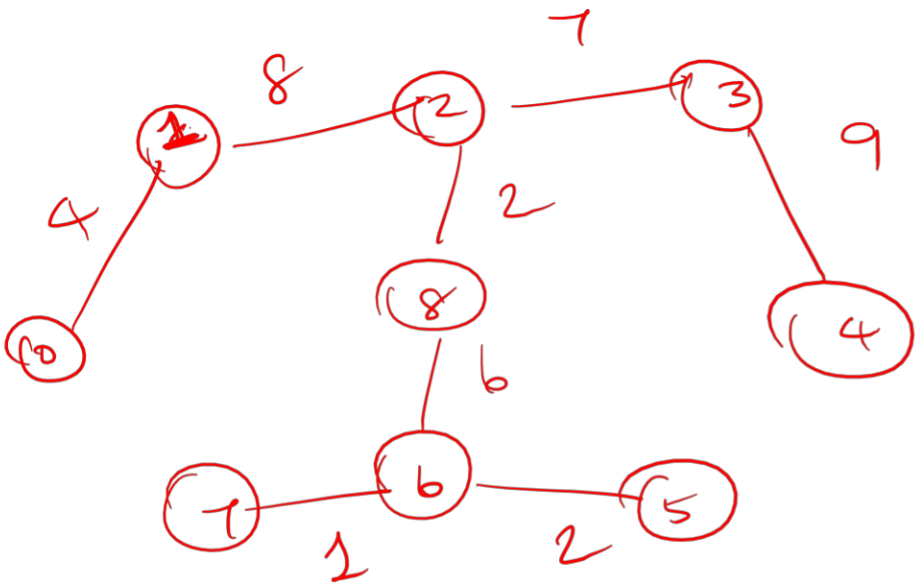
Reverse delete algorithm

- Example 2



Note : In case of same weight edges, we can pick any edge of the same weight edges as long as it does not disconnect the graph

Reverse delete algorithm



The MST weight is

$4+8+7+9+2+6+1+2$

$= 37$

Boruvka's algorithm

- It is used to identify the minimum spanning trees.
- It is one of the oldest minimum spanning tree algorithms.
- On record, Boruvka came up in 1926.
- It was published as a method of constructing an efficient electricity network.

Boruvka’s algorithm

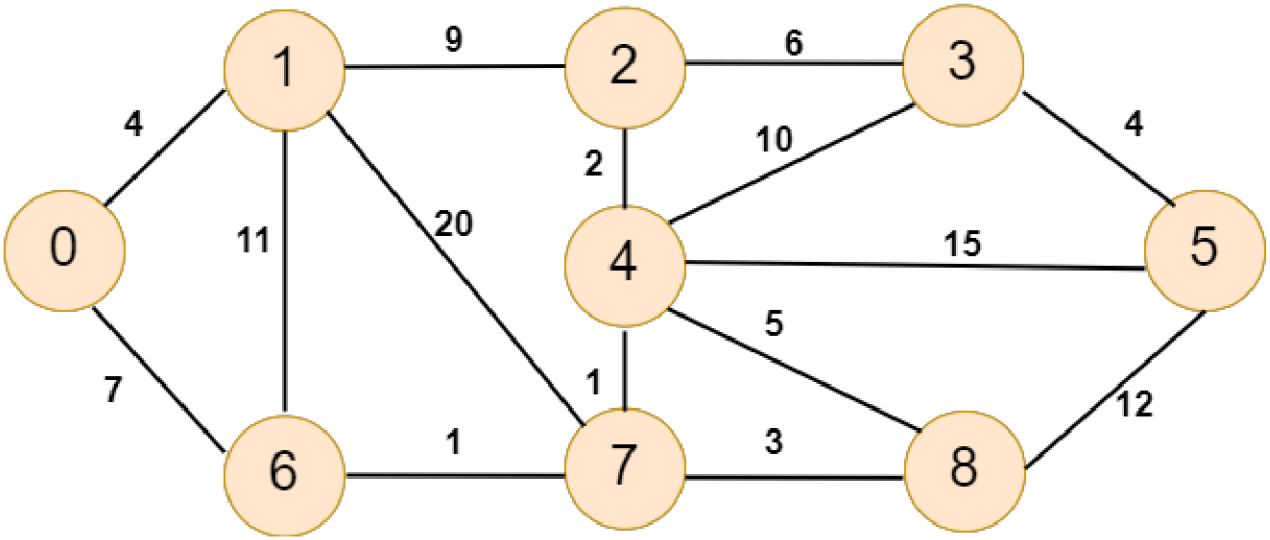
- It is a greedy algorithm that constructs a globally "optimal" solution using smaller, locally optimal solutions for smaller sub problems.

Boruvka's algorithm

Algorithm:

1. Take a connected, weighted, and undirected graph as input.
2. Initialize all the nodes as the individual components.
3. Initialize the empty graph minimum spanning tree (MST). It will contain the solution.
4. Perform the following operation, if there is more than one component.
    1. Find the minimum-weighted edge that connects this vertex to any other vertex.
    2. Add the least weighted edge to the MST if not exists already.
5. If there is one vertex remaining, return the minimum spanning tree.

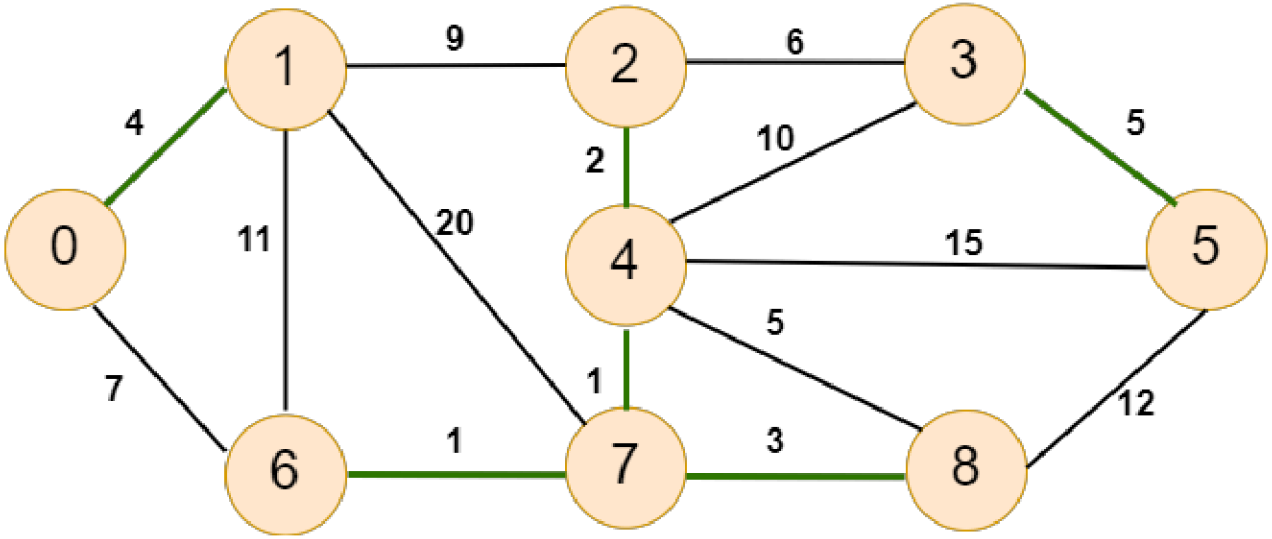
Boruvka's algorithm



Boruvka's algorithm

Vertices	Smallest weight edge that connects it to other vertex	Weight of the edge
{0}	0 - 1	4
{1}	0 - 1	4
{2}	2 - 4	2
{3}	3 - 5	5
{4}	4 - 7	1
{5}	3 - 5	10
{6}	6 - 7	1
{7}	4 - 7	1
{8}	7 - 8	3

Boruvka's algorithm



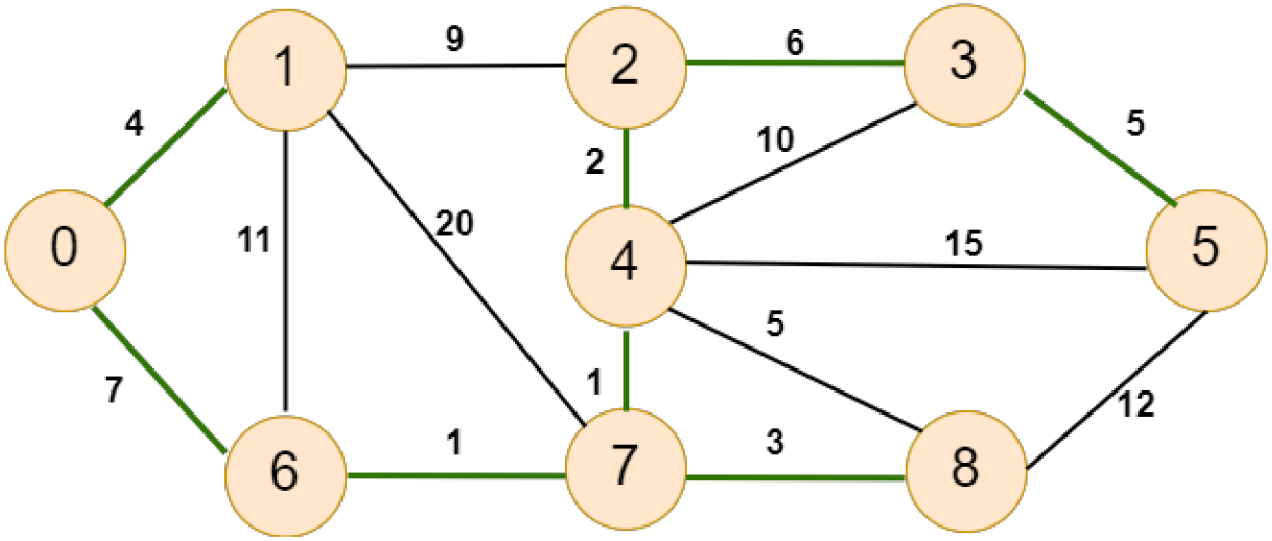
The highlighted line shows the nearest vertices that are bound together. As we can see, now we have the components: **{0, 1}, {2, 4, 6, 7, 8}** and **{3, 5}.** Again, we apply the algorithm to try to find the minimum-weight edges.

Boruvka's algorithm

Vertices	Smallest weight edge that connects it to other vertex	Weight of the edge
{0, 1}	0 - 6	7
{2, 4, 6, 7, 8}	2 - 3	6
{3, 5}	2 - 3	6

Boruvka's algorithm

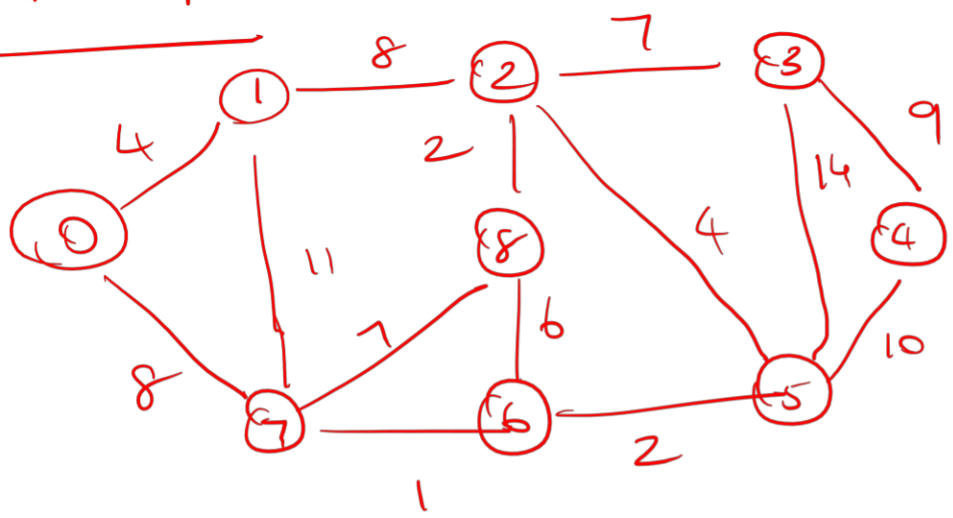
Now our graph will look like as below -



As we can observe, there is only one vertex in the graph, representing the minimum spanning tree. The weight of this tree is 29, which we get after summing all of the edges.

Boruvka's algorithm

Example



Component

{0}

{1}

{2}

{3}

{4}

{5}

{6}

{7}

{8}

Cheapest Edge

0-1

0-1

2-8

2-3

3-4

5-4

6-7

6-7

2-8

Boruvka's algorithm

Selection of cheapest Edge

0 - 1 (4) ✓
0 - 7 (8)

1 - 2 (8)
1 - 0 (4) ✓
1 - 7 (11)

2 - 8 (2) ✓
2 - 3 (7)
2 - 5 (4)
2 - 1 (8)

3 - 2 (7) ✓
3 - 4 (9)
3 - 5 (14)

4 - 3 (9) ✓
4 - 5 (10)

5 - 4 (10)
5 - 3 (14)
5 - 2 (4)
5 - 6 (2) ✓

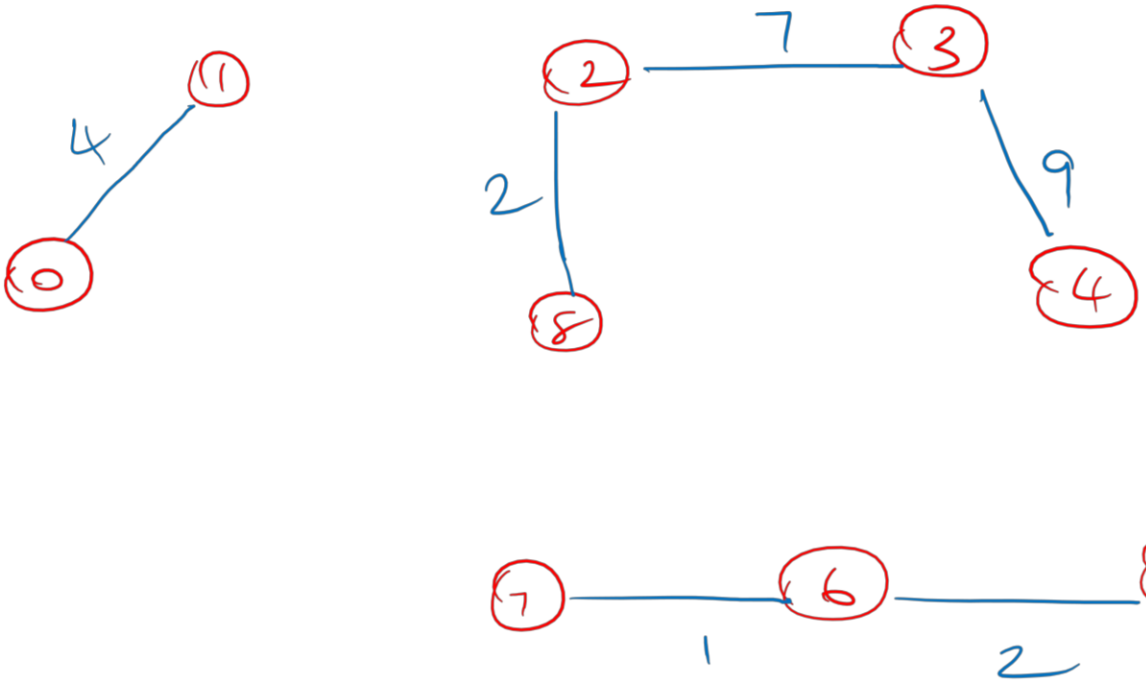
6 - 7 (1) ✓
6 - 8 (6)
6 - 5 (2)

7 - 6 (1) ✓
7 - 8 (7)
7 - 0 (8)
7 - 1 (11)

8 - 2 (2) ✓
8 - 6 (6)
8 - 7 (7)

Boruvka's algorithm

Highlight the cheapest edge



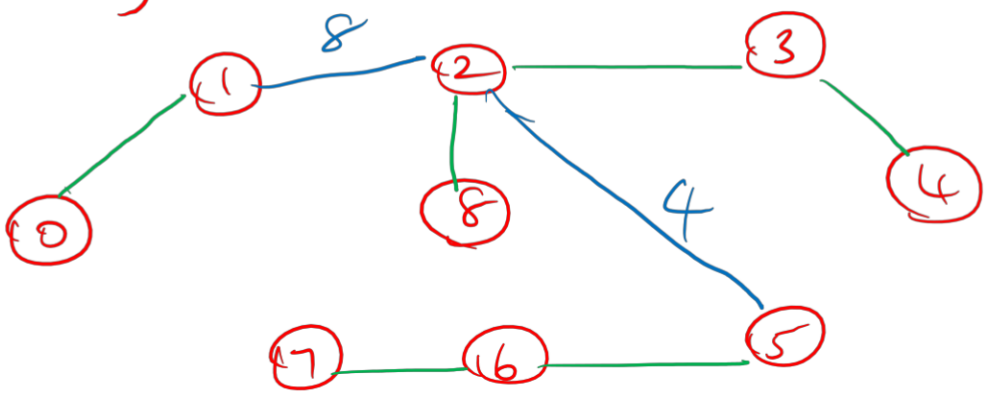
MST = { 0-1, 2-8, 2-3, 3-4, 5-6, 6-7 }

It has 3 disconnected components

Boruvka's algorithm

To connect these disconnected components, find the cheapest edge.

Component	cheapest Edge
{0, 1}	1 – 2
{2, 3, 4, 8}	2 – 5
{5, 6, 7}	2 – 5



Boruvka's algorithm

$\therefore$ MST = { 0-1, 2-8, 2-5, 3-4, 5-6, 6-7, 1-2, 2-5 }

MST weight = 4 + 2 + 7 + 9 + 2 + 1

8 + 4

= 37