Tuijin Jishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 44 No. 5 (2023)

Minimum Spanning Tree (MST): A Comprehensive Survey & Analysis

[1*] Dr. Leena Jain, [2]Saket Kumar, [3]Charanjit Singh, [4]Aanchal Madaan, [5]Amit Puri
[1,2,4,5] Department of Computer Applications, Global Group of Institutes, Amritsar, Punjab, India
[3]Department of Applied Sciences & Humanities, Global Group of Institutes, Amritsar, Punjab, India
[1*]Leenajain79@gmail.com, [2]Saketsah007@gmail.com, [3]csmathematics@gmail.com,
[4]Aanchalmadaan20@yahoo.com, [5]puri0881.ap@gmail.com

Abstract

The Minimum Spanning Tree (MST) problem is a fundamental optimization problem in computer science and graph theory, with applications in various domains such as network design, clustering, and resource allocation. In this paper, we explore and compare different algorithms for solving the MST problem. We discuss the key characteristics, complexities, and implementations of popular algorithms including Kruskal's Algorithm, Prim's Algorithm, Borůvka's Algorithm, and others. Overall, this paper serves as a comprehensive survey and analysis of different algorithms for solving the Minimum Spanning Tree problem, providing valuable insights for researchers, practitioners, and students in the field of algorithm design and graph theory.

Keywords: Minimum Spanning Tree, Greedy Algorithm, Kruskal's Algorithm, Prim's Algorithm, Borůvka's Algorithm

Introduction:

The spanning tree of a connected graph is the connected subgraph with the least edges, which must contain all vertexes of the connected graph [1]. In all spanning trees of a connected graph, the minimum spanning tree (MST) is that in which the sum of the weights of all edges is the smallest [2, 3]. The Minimum Spanning Tree (MST) problem is a fundamental optimization problem in computer science and graph theory, with applications in various domains such as VLSI, network design[4,], medical images, water supply networks and transportation networks [5]etc.

Here are several algorithms to solve the Minimum Spanning Tree (MST) problem, each with its own advantages and disadvantages. Here are some of the most commonly used algorithms to find the MST of a graph:

Kruskal's Algorithm:

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected, undirected graph[6,7]. It works by adding edges to the MST in increasing order of edge weights, as long as adding the edge does not create a cycle. The algorithm uses a disjoint-set data structure to efficiently check for cycles.

Here's the pseudocode for Kruskal's Algorithm to find the Minimum Spanning Tree (MST) of a graph:

```
Kruskal(Graph G):
    Initialize an empty list of edges for the MST
        // Step 1: Sort the edges by weight
    Sort all the edges of G in non-decreasing order of their weights
    // Initialize a disjoint-set data structure
    Initialize a disjoint-set DS with a set for each vertex in G
    // Step 2: Process each edge in sorted order
    for each edge (u, v) in the sorted list of edges:
        if the sets containing u and v are different:
```

Add edge (u, v) to the MST
Merge the sets containing u and v in the disjoint-set data structure
return the list of edges in the MST

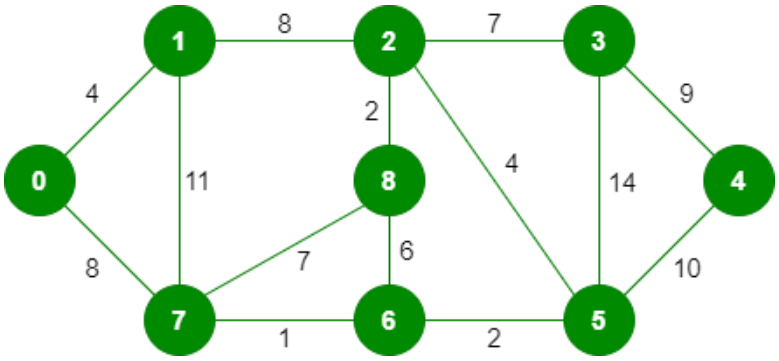
Explanation:
The algorithm starts by sorting all the edges of the graph in non-decreasing order of their weights. It then initializes a disjoint-set data structure with a set for each vertex in the graph. Next, it iterates through the sorted list of edges. For each edge, if the sets containing its endpoints are different (i.e., adding the edge does not create a cycle), the edge is added to the MST, and the sets are merged in the disjoint-set data structure. Finally, the list of edges in the MST is returned.

Time Complexity: O(E log E) or O(E log V)
Sorting the edges by weight: O(E log E) using efficient sorting algorithms like Merge Sort or Heap Sort. Union-Find operations for cycle detection: O(log V) for each edge, where V is the number of vertices. Overall, the time complexity is dominated by the sorting step, making it O(E log E) or O(E log V), depending on the implementation of sorting.

Space Complexity: O(V + E) for storing the graph and additional data structures.

- **Advantages**:
  - Suitable for sparse graphs with a large number of edges.
  - Works well on graphs with distributed edges and a small number of vertices.
  - Easy to implement.
- **Disadvantages**:
  - Slower on dense graphs due to sorting overhead.
  - Requires extra space for sorting and storing the edges.

Illustration:
Below is the illustration of the above approach:
Input Graph:



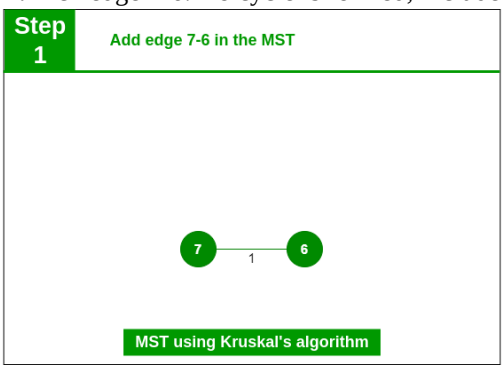
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.
After sorting:

Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
Source	7	8	6	0	2	8	2	7	0	1	3	5	1	3

Destination	6	2	5	1	5	6	3	8	7	2	4	4	7	5

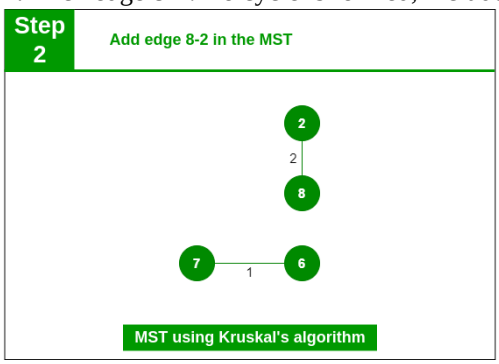
Now pick all edges one by one from the sorted list of edges

Step 1: Pick edge 7-6. No cycle is formed, include it.



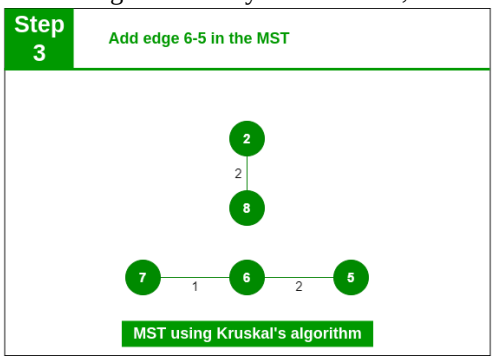
Add edge 7-6 in the MST

Step 2: Pick edge 8-2. No cycle is formed, include it.



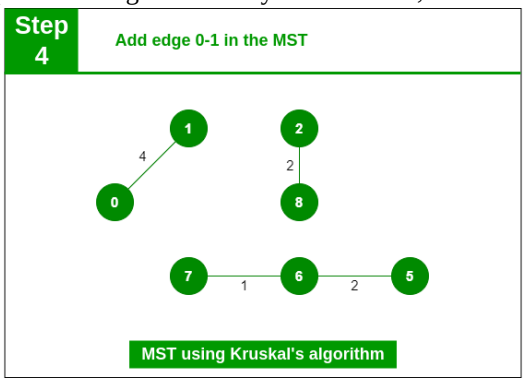
Add edge 8-2 in the MST

Step 3: Pick edge 6-5. No cycle is formed, include it.



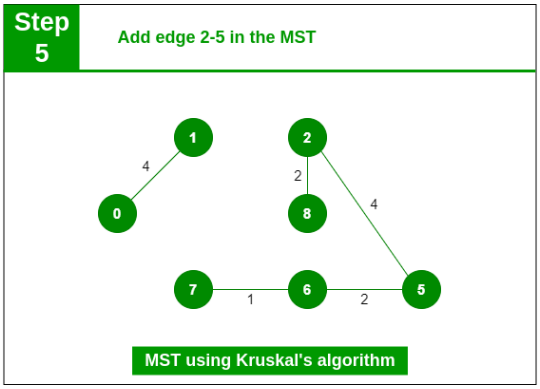
Add edge 6-5 in the MST

Step 4: Pick edge 0-1. No cycle is formed, include it.



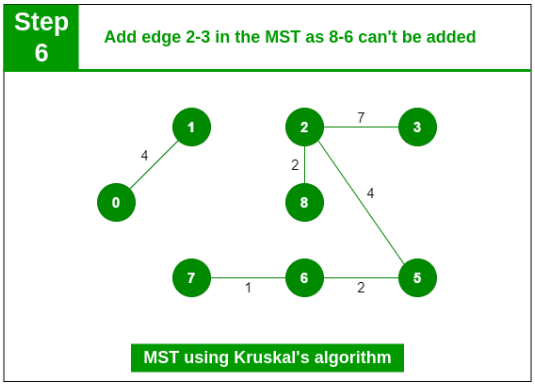
Add edge 0-1 in the MST

Step 5: Pick edge 2-5. No cycle is formed, include it.



Add edge 2-5 in the MST

Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.

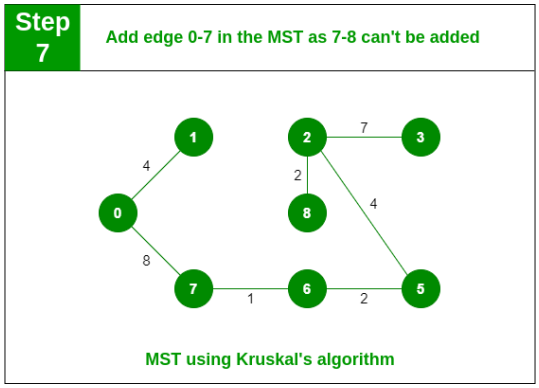


Add edge 2-3 in the MST

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is

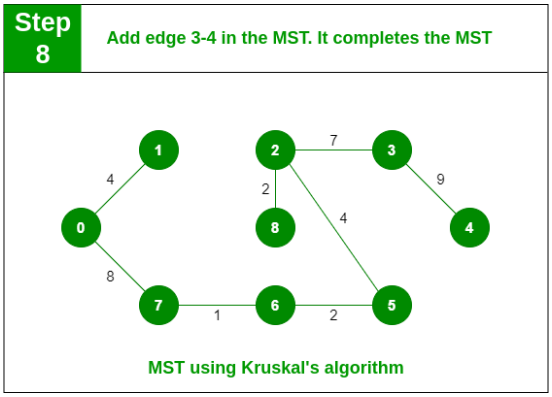
Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is

formed, include it.



Add edge 0-7 in MST

formed, include it.



Add edge 3-4 in the MST

Note: Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here

Prim's Algorithm:

Prim's algorithm is another greedy algorithm that finds a minimum spanning tree for a connected, undirected graph[9]. It starts with an arbitrary vertex and repeatedly grows the MST by adding the shortest edge that connects a vertex in the MST to a vertex outside the MST. Prim's algorithm can be implemented using a priority queue or a min-heap to efficiently select the next edge to add to the MST.

Here's the pseudocode for Prim's Algorithm to find the Minimum Spanning Tree (MST) of a graph:

```
Prim(Graph G):
    Initialize an empty list of edges for the MST
    Initialize a priority queue (min-heap) to store vertices with their respective key values
    // Step 1: Choose an arbitrary starting vertex as the root of the MST
    Choose an arbitrary vertex r from G and set its key value to 0
    Insert vertex r into the priority queue
    // Step 2: Grow the MST by iteratively selecting the next closest vertex
    while the priority queue is not empty:
        Extract the vertex u with the minimum key value from the priority queue
        // Add the edge (u, parent[u]) to the MST
        if parent[u] is not null:
            Add edge (u, parent[u]) to the MST
        // Update the key values of adjacent vertices and insert them into the priority queue
        for each vertex v adjacent to u:
            if v is in the priority queue and the weight of edge (u, v) is less than v's key value:
                Set v's parent to u
                Update v's key value to the weight of edge (u, v)
                Decrease v's priority in the priority queue to v's updated key value
        return the list of edges in the MST
```

Explanation:

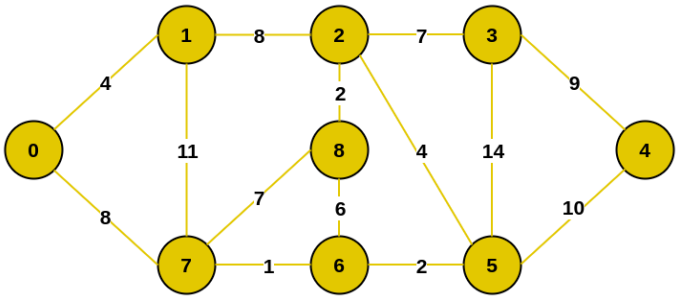
The algorithm starts by choosing an arbitrary starting vertex as the root of the MST and setting its key value to 0. It then inserts the root vertex into a priority queue. It repeatedly selects the vertex with the minimum key value from the priority queue and adds the corresponding edge to the MST. It updates the key values of adjacent vertices and

inserts them into the priority queue if necessary. The process continues until the priority queue is empty. Finally, the list of edges in the MST is returned.

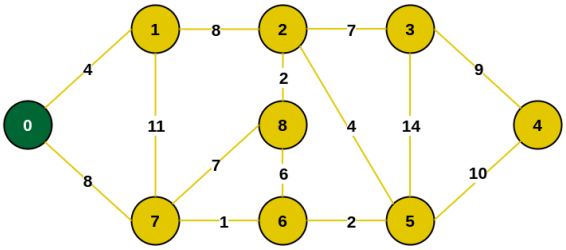
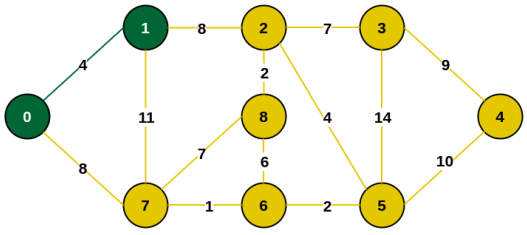
Time Complexity: O(E + V log V) or O(E log V)
Building the priority queue or min-heap: O(V) initialization and O(log V) for each insertion/update. Extracting the minimum element and updating the priority queue: O(E) in total. If using an adjacency matrix: O(V^2) for building the priority queue and O(V^2) for updating it, resulting in a total of O(V^2). Overall, the time complexity is O(E + V log V) or O(E log V), depending on the implementation.

- **Space Complexity**: O(V) for storing the vertices and additional data structures.
- **Advantages**:
  - Efficient for dense graphs.
  - Suitable for graphs with a small number of edges but a large number of vertices.
  - Easy to implement.
- **Disadvantages**:
  - Less efficient for sparse graphs with a large number of edges.
  - May require additional space for priority queue or heap.

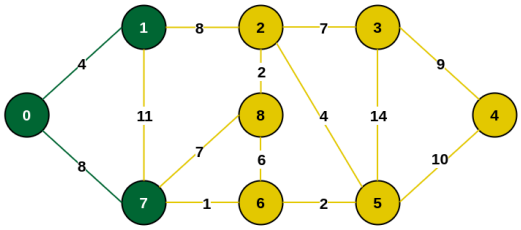
Illustration of Prim's Algorithm:
Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST).



Example of a Graph

<table>
<tr>
<td><u>Step 1:</u> Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex 0 as the starting vertex.
 
Select an arbitrary starting vertex. Here we have selected 0
0 is selected as starting vertex</td>
<td><u>Step 2:</u> All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.
 
Minimum weighted edge from MST to other vertices is 0-1 with weight 4
1 is added to the MST</td>
</tr>
<tr>
<td><u>Step 3:</u> The edges connecting the incomplete MST to</td>
<td><u>Step 4:</u> The edges that connect the incomplete MST</td>
</tr>
</table>

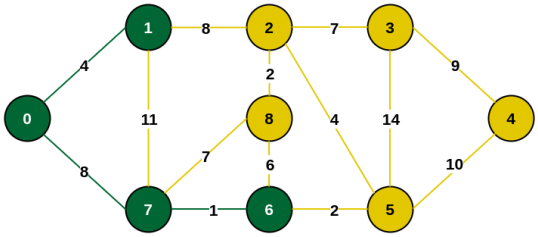
other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



Minimum weighted edge from MST to other vertices is 0-7 with weight 8

7 is added in the MST

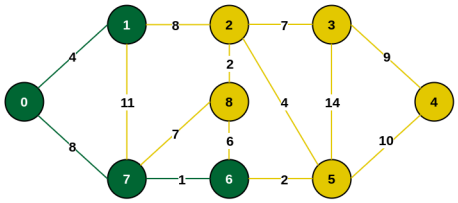
with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).



Minimum weighted edge from MST to other vertices is 7-6 with weight 1

6 is added in the MST

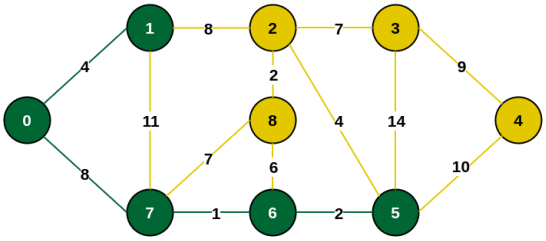
Step 4: The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).



Minimum weighted edge from MST to other vertices is 7-6 with weight 1

6 is added in the MST

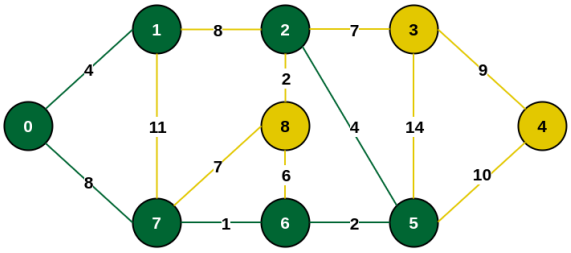
Step 5: The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.



Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Include vertex 5 in the MST

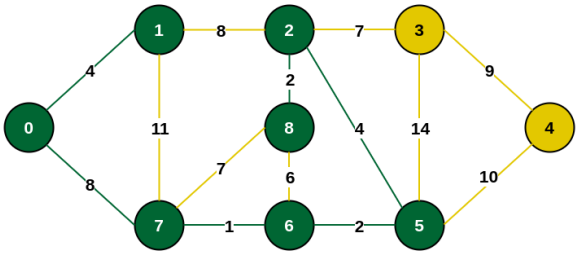
Step 6: Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.



Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Include vertex 2 in the MST

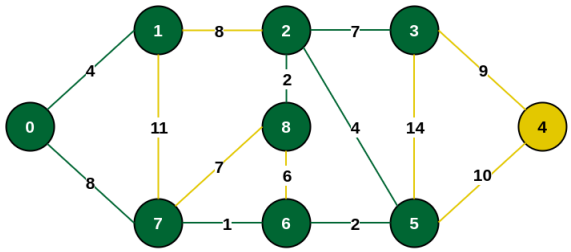
Step 7: The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.



Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Add vertex 8 in the MST

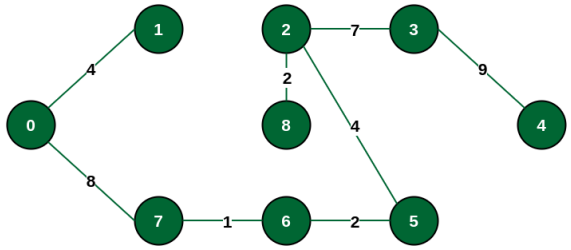
Step 8: See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.



Minimum weighted edge from MST to other vertices is 2-3 with weight 7

Include vertex 3 in MST

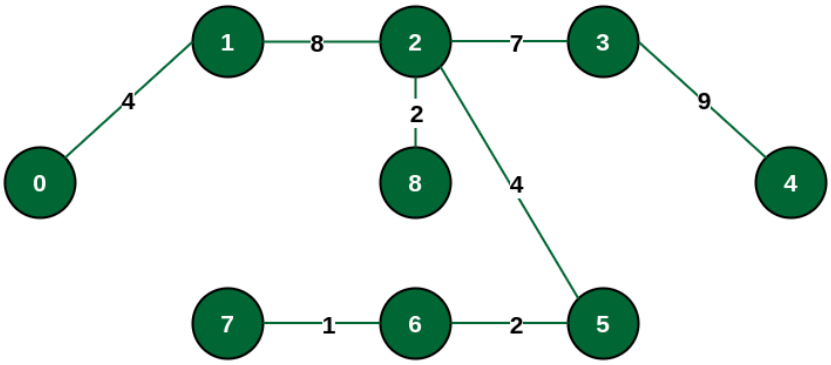
The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = **37**.



The final structure of MST

The structure of the MST formed using the above method

Note: If we had selected the edge {1, 2} in the third step then the MST would look like the following.



Alternative MST structure

Structure of the alternate MST if we had selected edge {1, 2} in the MST

Borůvka's Algorithm:

Borůvka's algorithm is a parallel algorithm[10,11,12] that finds a minimum spanning tree for a connected, undirected graph. It works by repeatedly selecting the cheapest edge incident on each vertex and adding it to the MST. Borůvka's algorithm can be parallelized to run efficiently on parallel computing architectures.

Here's the pseudocode for Borůvka's Algorithm to find the Minimum Spanning Tree (MST) of a grap

```
Boruvka(Graph G):
  Initialize an empty list of edges for the MST
  Initialize a disjoint-set data structure (DS) to store the connected components of the graph
    // Step 1: Initialize each vertex as a separate component
  for each vertex v in G:
    MakeSet(v) // Create a set containing only v in the disjoint-set data structure
    // Step 2: Repeat until there is only one connected component
  while the number of connected components in DS is greater than 1:
    Initialize a list of cheapest edges for each component
      // Find the cheapest edge for each component
    for each edge (u, v) in G:
      if Find(u) is not equal to Find(v) and weight of (u, v) is less than the weight of the cheapest edge for Find(u):
        Update the cheapest edge for Find(u) to (u, v)
      if Find(u) is not equal to Find(v) and weight of (u, v) is less than the weight of the cheapest edge for Find(v):
        Update the cheapest edge for Find(v) to (u, v)
      // Add the cheapest edge for each component to the MST
    for each component in DS:
      if the component has a cheapest edge:
        Add the cheapest edge to the MST
        Union(Find(u), Find(v)) // Merge the sets containing u and v in the disjoint-set data structure
    return the list of edges in the MST
```

Time Complexity: O(E log V)

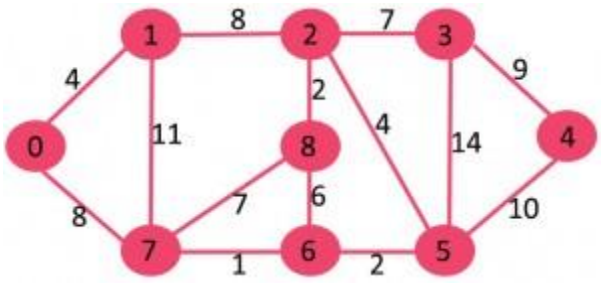
Selecting the cheapest edge incident on each vertex: O(E) in total.

Each iteration reduces the number of components by at least half.

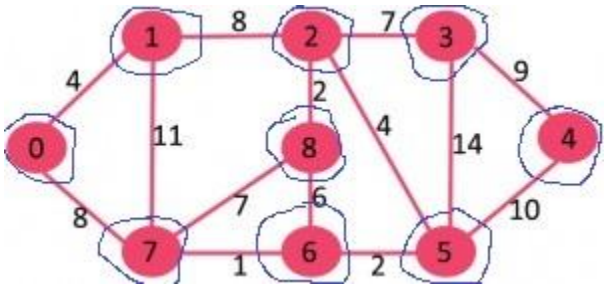
Overall, the time complexity is O(E log V).

- **Space Complexity**: O(V + E) for storing the graph and additional data structures.
- **Advantages**:
  - Parallelizable, making it suitable for distributed environments.
  - Works well for both sparse and dense graphs.
  - Provides a faster alternative for sparse graphs compared to Kruskal's and Prim's algorithms.
- **Disadvantages**:
  - More complex to implement compared to Kruskal's and Prim's algorithms.
  - May require additional space for storing components.

Let us understand the algorithm in the below example.



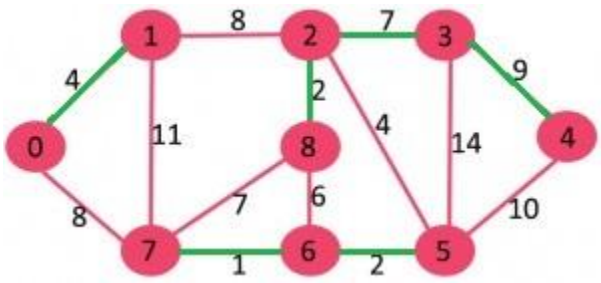
Initially, MST is empty. Every vertex is single component as highlighted in blue color in the below diagram.



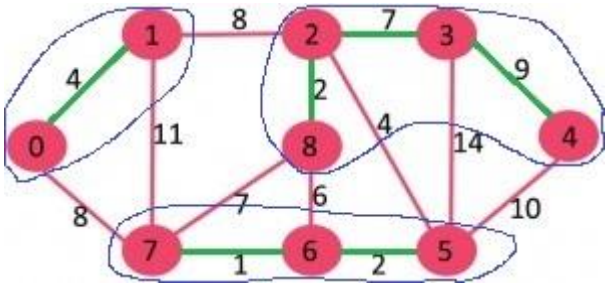
For every component, find the cheapest edge that connects it to some other component.

Component	**Cheapest Edge that connects it to some other component**
{0}	0-1
{1}	0-1
{2}	2-8
{3}	2-3
{4}	3-4
{5}	5-6
{6}	6-7
{7}	6-7
{8}	2-8

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.



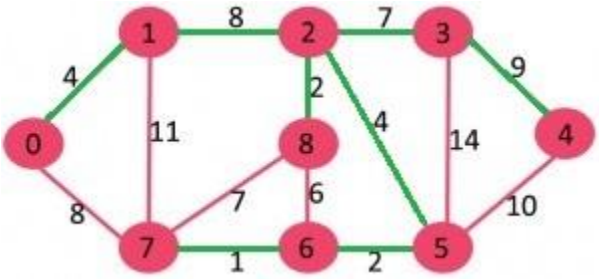
After above step, components are {{0,1}, {2,3,4,8}, {5,6,7}}. The components are encircled with blue color.



We again repeat the step, i.e., for every component, find the cheapest edge that connects it to some other component.

Component	**Cheapest Edge that connects it to some other component**
{0,1}	1-2 (or 0-7)
{2,3,4,8}	2-5
{5,6,7}	2-5

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5}



At this stage, there is only one component {0, 1, 2, 3, 4, 5, 6, 7, 8} which has all edges. Since there is only one component left, we stop and return MST.

Reverse-Delete Algorithm:

The reverse-delete algorithm is a simple algorithm that finds a minimum spanning tree for a connected, undirected graph. It starts with all edges in the graph and repeatedly removes the most expensive edge that does not disconnect the graph until only the edges of the MST remain. The Reverse-Delete Algorithm is a simple approach for finding the Minimum Spanning Tree (MST) of a graph by iteratively deleting edges from the graph while ensuring that the remaining edges still form a spanning tree.

Here's the pseudocode for the Reverse-Delete Algorithm:

```
ReverseDelete(Graph G):
    Initialize an empty list of edges for the MST
    Sort all the edges of G in non-increasing order of their weights
        // Step 1: Construct an initial spanning tree using all the edges
    Initialize a disjoint-set data structure (DS) to keep track of the connected components
    for each edge (u, v) in the sorted list of edges:
        if Find(u) is not equal to Find(v): // Check if adding the edge creates a cycle
            Add edge (u, v) to the MST
```

```
        Union(Find(u), Find(v)) // Merge the sets containing u and v in the disjoint-set data structure
    // Step 2: Iteratively delete edges from the MST while ensuring it remains connected
  for each edge (u, v) in the MST in non-decreasing order of their weights:
    if removing edge (u, v) disconnects the MST:
      Remove edge (u, v) from the MST
    else:
      Add edge (u, v) back to the MST
    return the list of edges in the MST
```

Time Complexity: O(E^2)

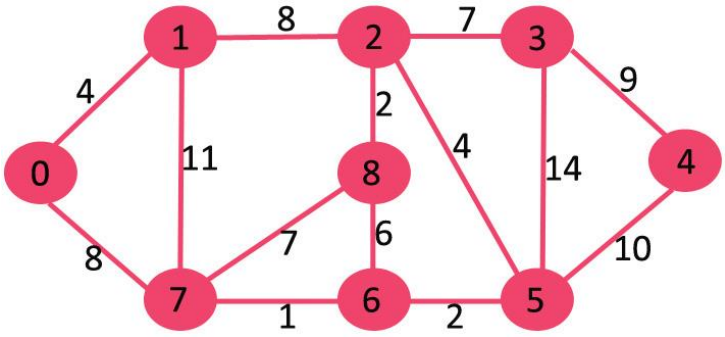
Removing the most expensive edge that does not disconnect the graph: O(E) for each edge deletion.

The algorithm repeats E times.

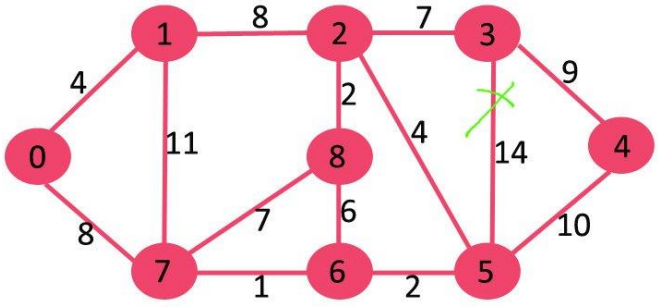
Overall, the time complexity is O(E^2).

- **Space Complexity**: O(V + E) for storing the graph and additional data structures.
- **Advantages**:
  - Suitable for graphs where the cost of cycle detection is low.
  - Can be faster than Kruskal's and Prim's algorithms in specific cases.
- **Disadvantages**:
  - Not as efficient as other algorithms in general.
  - May require additional space for storing the graph and intermediate data structures.

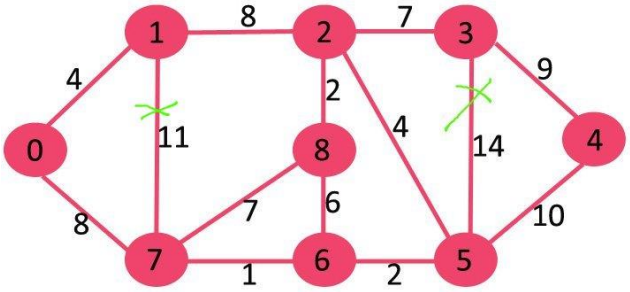
Let us understand with the following example:



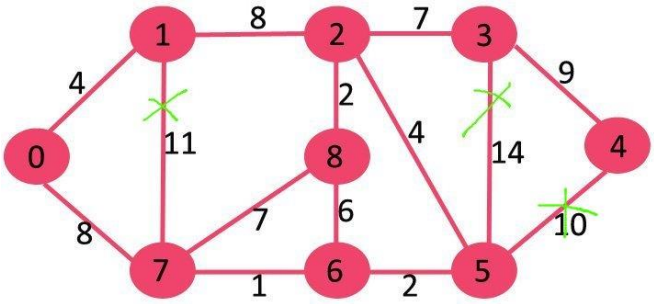
If we delete highest weight edge of weight 14, graph doesn't become disconnected, so we remove it.



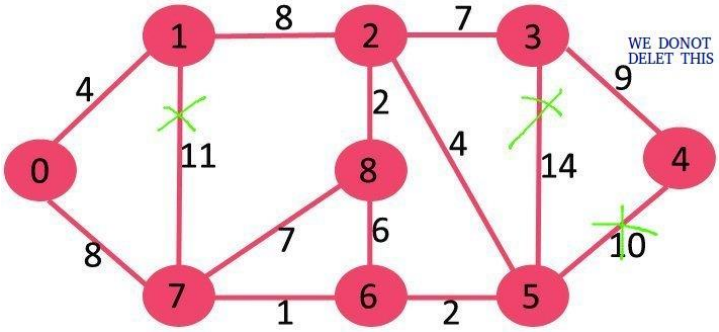
Next we delete 11 as deleting it doesn't disconnect the graph.



Next we delete 10 as deleting it doesn't disconnect the graph.



Next is 9. We cannot delete 9 as deleting it causes disconnection.



We continue this way and following edges remain in final MST.
Edges in MST
(3, 4)
(0, 7)
(2, 3)
(2, 5)
(0, 1)
(5, 6)
(2, 8)
(6, 7)

Note : In case of same weight edges, we can pick any edge of the same weight edges.

Borůvka–Chandrasekaran Algorithm:
The Borůvka–Chandrasekaran algorithm is a parallel algorithm that combines Borůvka's algorithm with Chandrasekaran's method for efficiently merging components in a parallel setting.
It is particularly suited for distributed-memory parallel computing environments.
The algorithm has a time complexity of O(E log V) and can achieve good scalability on large graphs.
These are some of the most commonly used algorithms to solve the Minimum Spanning Tree problem. The choice of algorithm depends on factors such as the characteristics of the input graph, the available computational resources, and the desired trade-offs between simplicity, efficiency, and parallelizability.
The Borůvka–Chandrasekaran Algorithm is a parallel algorithm for finding the Minimum Spanning Tree (MST) of a graph. Here's the pseudocode for the Borůvka–Chandrasekaran Algorithm:

```
BoruvkaChandrasekaran(Graph G):
    Initialize an empty list of edges for the MST
    Initialize a disjoint-set data structure (DS) to keep track of the connected components

    // Step 1: Initialize each vertex as a separate component
    for each vertex v in G:
        MakeSet(v) // Create a set containing only v in the disjoint-set data structure

    // Step 2: Repeat until there is only one connected component
    while the number of connected components in DS is greater than 1:
        Initialize an empty list of edges to store the cheapest edge for each component
        Initialize a list of sets to store the new components after merging

        // Step 2a: Find the cheapest edge for each component
        for each edge (u, v) in G:
            if Find(u) is not equal to Find(v) and weight of (u, v) is less than the weight of the cheapest edge for Find(u):
                Update the cheapest edge for Find(u) to (u, v)
            if Find(u) is not equal to Find(v) and weight of (u, v) is less than the weight of the cheapest edge for Find(v):
                Update the cheapest edge for Find(v) to (u, v)

        // Step 2b: Merge the components using the cheapest edges
        for each component in DS:
            if the component has a cheapest edge:
                Add the cheapest edge to the MST
                Union(Find(u), Find(v)) // Merge the sets containing u and v in the disjoint-set data structure
                Add the merged component to the list of new components

        // Step 2c: Update the disjoint-set data structure with the new components
        DS = NewComponents

    return the list of edges in the MST
```

Time Complexity: O(E log V)
Similar to Borůvka's algorithm, with additional steps for merging components efficiently in parallel.
Overall, the time complexity is O(E log V).

Conclusion:

Overall, Kruskal's and Prim's algorithms are the most commonly used for finding the MST due to their efficient time complexity. Borůvka's algorithm can be efficient in certain cases, especially for parallel implementations. The Reverse-Delete algorithm is simple but less efficient compared to the others, while the Borůvka–Chandrasekaran algorithm is designed for parallel environments. The choice of algorithm depends on factors such as the characteristics of the input graph, available computational resources, and desired performance trade-offs. In summary, the choice of algorithm depends on various factors such as the characteristics of the graph (sparse or dense), available computational resources, ease of implementation, and specific requirements of the application. Kruskal's and Prim's algorithms are commonly used due to their simplicity and efficiency in different scenarios, while Borůvka's algorithm offers parallelization advantages and Reverse-Delete algorithm may be suitable for specific graph structures with low cycle detection cost.

References

1. R. L. Graham and P. Hell, “On the history of the minimum spanning tree problem,” IEEE Annals of the History of Computing, vol. 7, no. 1, pp. 43–57, 1985.
2. A. Singh, “An artificial bee colony algorithm for the leaf-constrained minimum spanning tree problem,” Applied Soft Computing, vol. 9, no. 2, pp. 625–631, 2009.
3. L. Gouveia, M. Leitner, and I. Ljubić, “A polyhedral study of the diameter constrained minimum spanning tree problem,” Discrete Applied Mathematics, vol. 285, pp. 364–379, 2020.
4. N. Akpan and I. Iwok, “A minimum spanning tree approach of solving a transportation problem,” International Journal of Mathematics and Statistics Invention, vol. 5, no. 3, pp. 09–18, 2017.
5. R. L. Graham and P. Hell, “On the history of the minimum spanning tree problem,” Annals of the History of Computing, vol. 7, no. 1, pp. 43–57, 1985.
6. J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” Proceedings of the American Mathematical Society, vol. 7, no. No. 1, Feb., 1956.
7. P. C. Pop, “The generalized minimum spanning tree problem: an overview of formulations, solution procedures and latest advances,” European Journal of Operational Research, vol. 283, pp. 1–15, 2020.
8. R. C. Prim, “Shortest connection networks and some generalizations,” Bell System Technical Journal, vol. 36, no. 6, pp. 1389–1401, 1957.
9. Borůvka, Otakar (1926). "O jistém problému minimálním" [About a certain minimal problem]. Práce Mor. Přírodověd. Spol. V Brně III (in Czech and German). **3**: 37–58.
10. Borůvka, Otakar (1926). "Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí (Contribution to the solution of a problem of economical construction of electrical networks)". Elektronický Obzor (in Czech). **15**: 153–154.
11. Nešetřil, Jaroslav; Milková, Eva; Nešetřilová, Helena (2001). "Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history". Discrete Mathematics. **233** (1–3): 3–36. doi:10.1016/S0012-365X(00)00224-7. hdl:10338.dmlcz/500413. MR 1825599.