Kruskal’s Minimum Spanning Tree Algorithm | Greedy Algo-2

Difficulty Level : Hard • Last Updated : 19 Apr, 2021

What is 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.
How many edges does a minimum spanning tree has?
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.
What are the applications of the *Minimum Spanning Tree?*
See this for applications of MST.

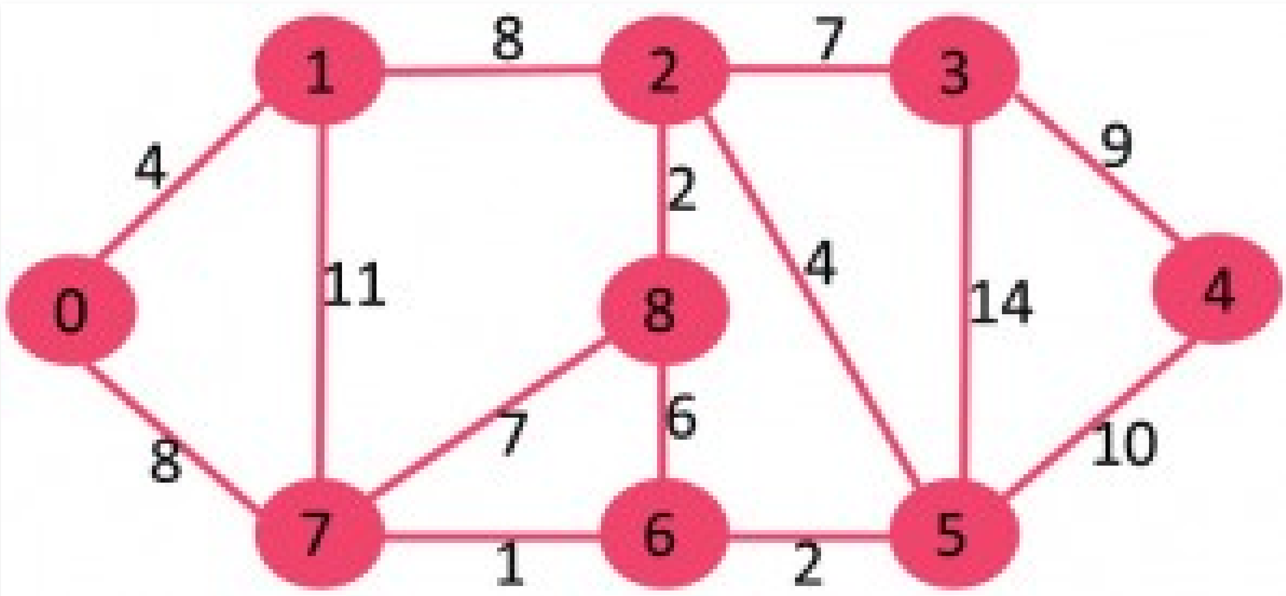
Below are the steps for finding MST using Kruskal’s algorithm

1. *Sort all the edges in non-decreasing order of their weight.*
2. *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
3. *Repeat step#2 until there are (V-1) edges in the spanning tree.*

Step #2 uses the Union-Find algorithm to detect cycles. So we recommend reading the

[Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)]

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



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

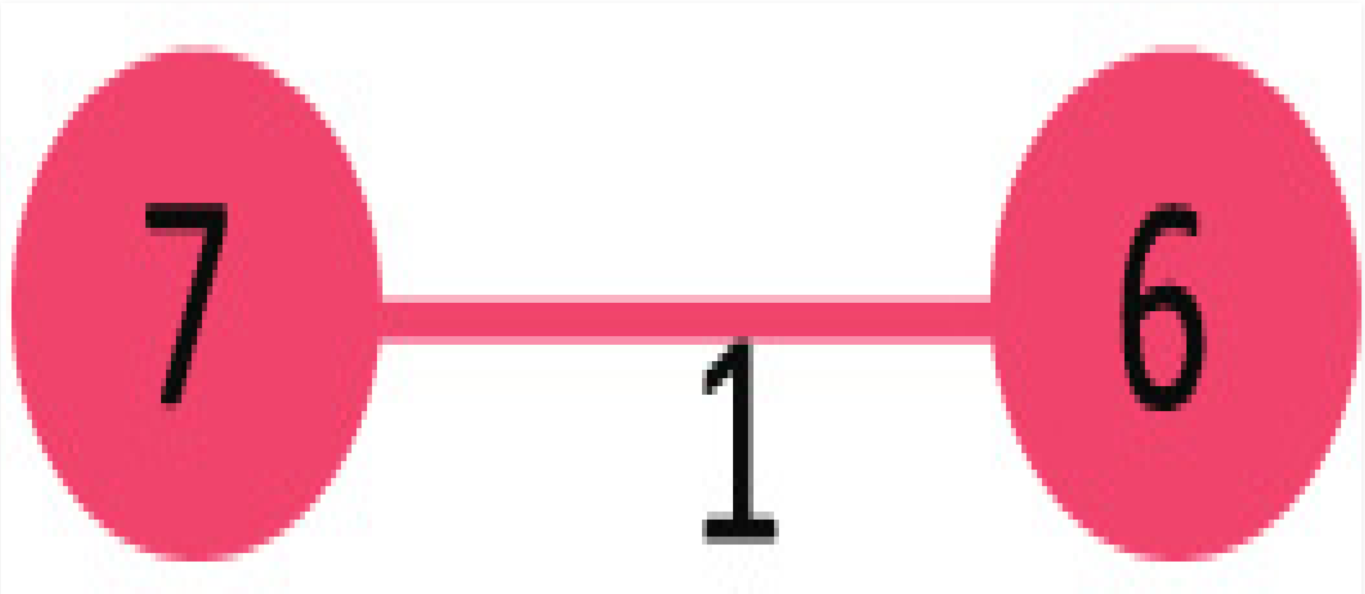
```
After sorting:

Weight   Src    Dest
1         7      6
2         8      2
2         6      5
4         0      1
4         2      5
6         8      6
7         2      3
7         7      8
```

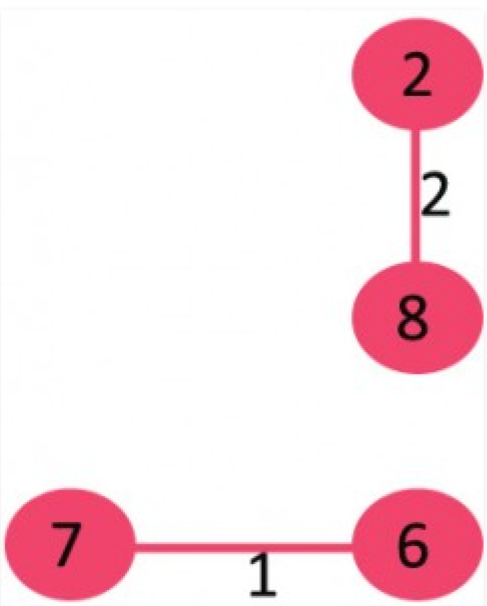
10	5	4
11	1	7
14	3	5

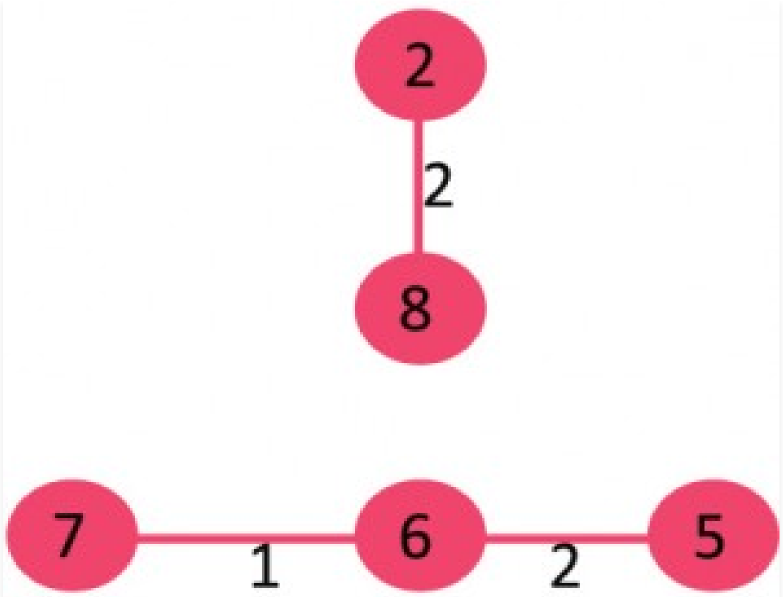
Now pick all edges one by one from the sorted list of edges

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

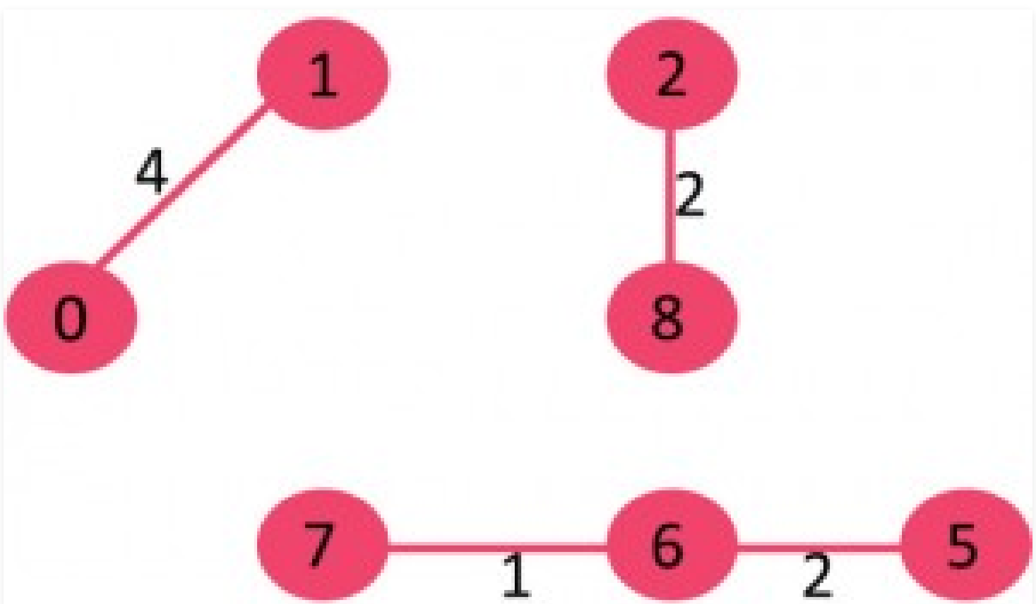


2. *Pick edge 8-2:* No cycle is formed, include it.

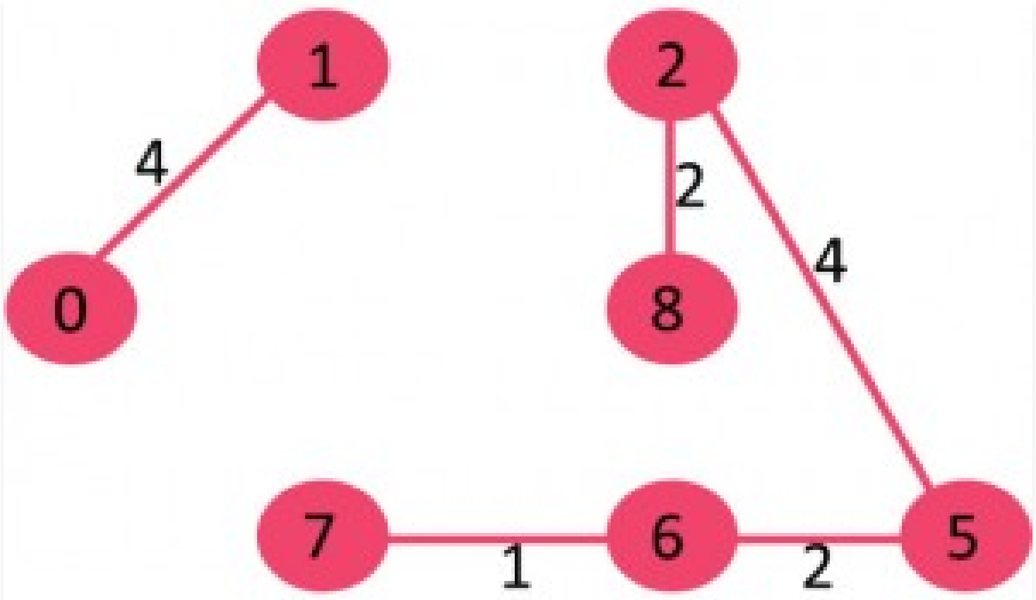




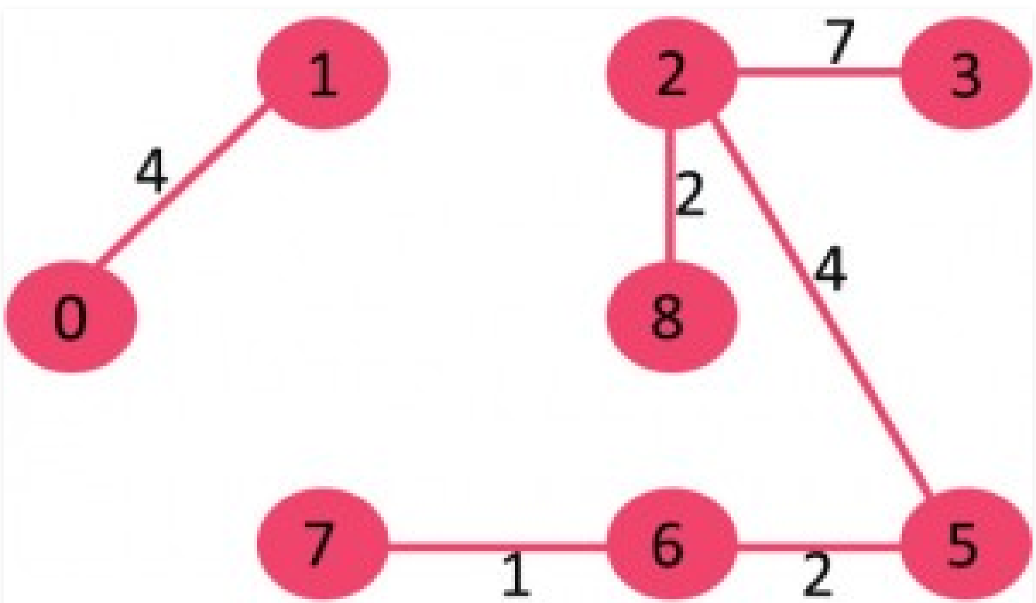
4. *Pick edge 0-1:* No cycle is formed, include it.



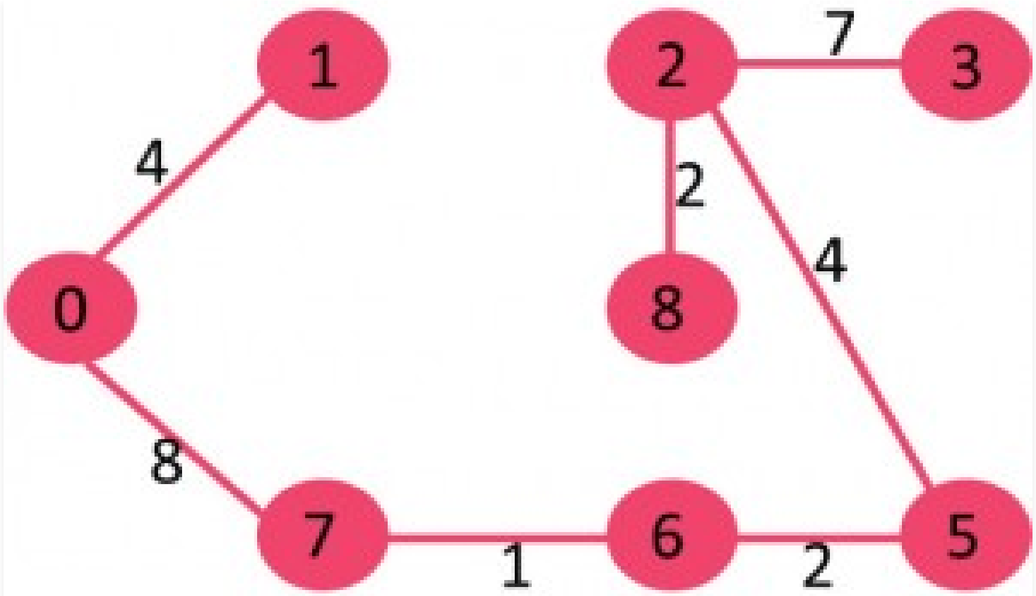
5. *Pick edge 2-5:* No cycle is formed, include it.



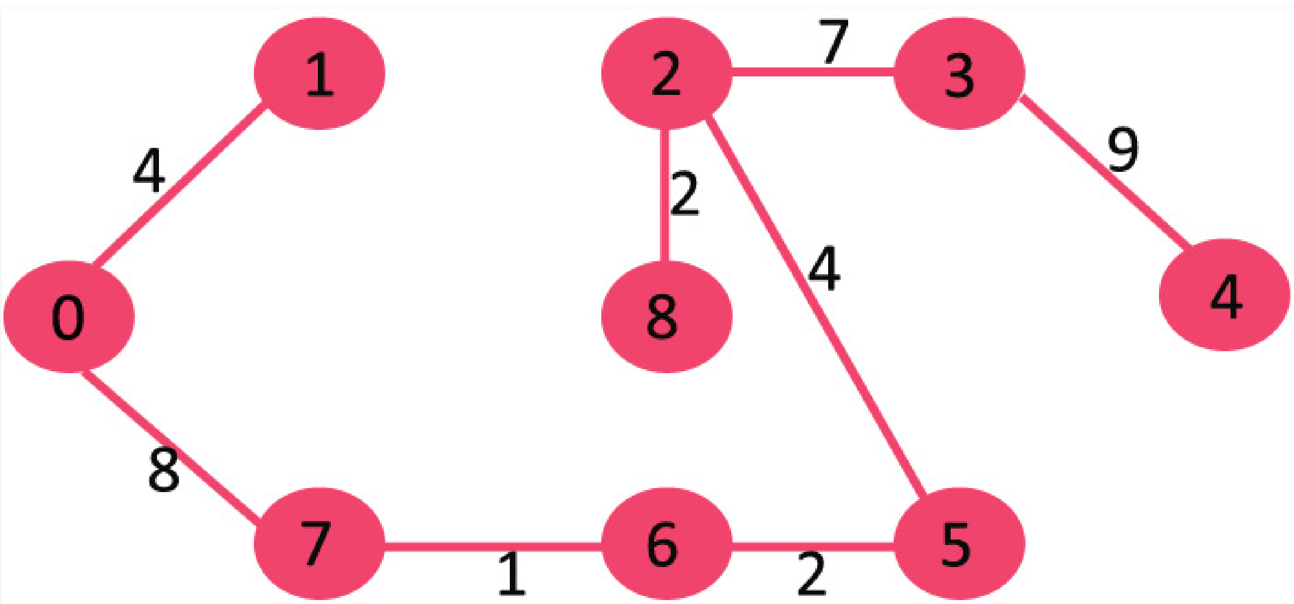
6. *Pick edge 8-6:* Since including this edge results in the cycle, discard it.
7. *Pick edge 2-3:* No cycle is formed, include it.



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



10. *Pick edge 1-2:* Since including this edge results in the cycle, discard it.
11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

C

```c
// C program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*)(malloc(sizeof(struct Graph)));
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*)malloc(sizeof( struct Edge));

    return graph;
}

// A structure to represent a subset for union-find
struct subset {
    int parent;
    int rank;
};

// A utility function to find set of an element i
```

```
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent
            = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's
// algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge
```

```
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
        myComp);

    // Allocate memory for creating V ssubsets
    struct subset* subsets
        = (struct subset*)malloc(V * sizeof(struct subset));

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E) {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the
    // built MST
    printf(
        "Following are the edges in the constructed MST\n");
    int minimumCost = 0;
    for (i = 0; i < e; ++i)
    {
        printf("%d -- %d == %d\n", result[i].src,
            result[i].dest, result[i].weight);
        minimumCost += result[i].weight;
    }
```

```
// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
            10
        0--------1
        | \      |
    6|  5\  |15
        |      \ |
        2--------3
            4      */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

Java

```java
//connected, undirected and  weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph {
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

        // Comparator function used for
        // sorting edgesbased on their weight
        public int compareTo(Edge compareEdge)
        {
            return this.weight - compareEdge.weight;
        }
    };

    // A class to represent a subset for
    // union-find
    class subset
    {
        int parent, rank;
    };

    int V, E; // V-> no. of vertices & E->no.of edges
    Edge edge[]; // collection of all edges

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i = 0; i < e; ++i)
            edge[i] = new Edge();
    }

    // A utility function to find set of an
    // element i (uses path compression technique)
    int find(subset subsets[], int i)
    {
        // find root and make root as parent of i
        // (path compression)
        if (subsets[i].parent != i)
            subsets[i].parent
                = find(subsets, subsets[i].parent);
```

```
    // A function that does union of two sets
    // of x and y (uses union by rank)
    void Union(subset subsets[], int x, int y)
    {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        // Attach smaller rank tree under root
        // of high rank tree (Union by Rank)
        if (subsets[xroot].rank
            < subsets[yroot].rank)
            subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank
                 > subsets[yroot].rank)
            subsets[yroot].parent = xroot;

        // If ranks are same, then make one as
        // root and increment its rank by one
        else {
            subsets[yroot].parent = xroot;
            subsets[xroot].rank++;
        }
    }

    // The main function to construct MST using Kruskal's
    // algorithm
    void KruskalMST()
    {
        // Tnis will store the resultant MST
        Edge result[] = new Edge[V];

        // An index variable, used for result[]
        int e = 0;

        // An index variable, used for sorted edges
        int i = 0;
        for (i = 0; i < V; ++i)
            result[i] = new Edge();

        // Step 1:  Sort all the edges in non-decreasing
        // order of their weight.  If we are not allowed to
        // change the given graph, we can create a copy of
        // array of edges
        Arrays.sort(edge);

        // Allocate memory for creating V ssubsets
```

```java
        // Create V subsets with single elements
        for (int v = 0; v < V; ++v)
        {
            subsets[v].parent = v;
            subsets[v].rank = 0;
        }

        i = 0; // Index used to pick next edge

        // Number of edges to be taken is equal to V-1
        while (e < V - 1)
        {
            // Step 2: Pick the smallest edge. And increment
            // the index for next iteration
            Edge next_edge = edge[i++];

            int x = find(subsets, next_edge.src);
            int y = find(subsets, next_edge.dest);

            // If including this edge does't cause cycle,
            // include it in result and increment the index
            // of result for next edge
            if (x != y) {
                result[e++] = next_edge;
                Union(subsets, x, y);
            }
            // Else discard the next_edge
        }

        // print the contents of result[] to display
        // the built MST
        System.out.println("Following are the edges in "
                           + "the constructed MST");
        int minimumCost = 0;
        for (i = 0; i < e; ++i)
        {
            System.out.println(result[i].src + " -- "
                               + result[i].dest
                               + " == " + result[i].weight);
            minimumCost += result[i].weight;
        }
        System.out.println("Minimum Cost Spanning Tree "
                           + minimumCost);
    }

    // Driver Code
    public static void main(String[] args)
    {
```

```
            0--------1
            |  \     |
           6|   5\   |15
            |      \ |
            2--------3
                4       */
        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        Graph graph = new Graph(V, E);

        // add edge 0-1
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = 10;

        // add edge 0-2
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 6;

        // add edge 0-3
        graph.edge[2].src = 0;
        graph.edge[2].dest = 3;
        graph.edge[2].weight = 5;

        // add edge 1-3
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 15;

        // add edge 2-3
        graph.edge[4].src = 2;
        graph.edge[4].dest = 3;
        graph.edge[4].weight = 4;

        // Function call
        graph.KruskalMST();
    }
}
// This code is contributed by Aakash Hasija
```

Python

```
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
```

```python
# Class to represent a graph


class Graph:

    def __init__(self, vertices):
        self.V = vertices  # No. of vertices
        self.graph = []  # default dictionary
        # to store graph

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's
        # algorithm
    def KruskalMST(self):

        result = []  # This will store the resultant MST

        # An index variable, used for sorted edges
```

```
        # Step 1:  Sort all the edges in
        # non-decreasing order of their
        # weight.  If we are not allowed to change the
        # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge does't
            #  cause cycle, include it in result
            #  and increment the indexof result
            # for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        minimumCost = 0
        print ("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree" , minimumCost)

# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
```

```
# Function call
g.KruskalMST()

# This code is contributed by Neelam Yadav
```

C#

```
// C# Code for above approach
using System;

class Graph {

    // A class to represent a graph edge
    class Edge : IComparable<Edge> {
        public int src, dest, weight;

        // Comparator function used for sorting edges
        // based on their weight
        public int CompareTo(Edge compareEdge)
        {
            return this.weight
                   - compareEdge.weight;
        }
    }

    // A class to represent
    // a subset for union-find
    public class subset
    {
        public int parent, rank;
    };

    int V, E; // V-> no. of vertices & E->no.of edges
    Edge[] edge; // collection of all edges

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i = 0; i < e; ++i)
            edge[i] = new Edge();
    }
```

```
        // find root and make root as
        // parent of i (path compression)
        if (subsets[i].parent != i)
            subsets[i].parent
                = find(subsets, subsets[i].parent);

        return subsets[i].parent;
    }

    // A function that does union of
    // two sets of x and y (uses union by rank)
    void Union(subset[] subsets, int x, int y)
    {
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        // Attach smaller rank tree under root of
        // high rank tree (Union by Rank)
        if (subsets[xroot].rank < subsets[yroot].rank)
            subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
            subsets[yroot].parent = xroot;

        // If ranks are same, then make one as root
        // and increment its rank by one
        else {
            subsets[yroot].parent = xroot;
            subsets[xroot].rank++;
        }
    }

    // The main function to construct MST
    // using Kruskal's algorithm
    void KruskalMST()
    {
        // This will store the
        // resultant MST
        Edge[] result = new Edge[V];
        int e = 0; // An index variable, used for result[]
        int i
            = 0; // An index variable, used for sorted edges
        for (i = 0; i < V; ++i)
            result[i] = new Edge();

        // Step 1: Sort all the edges in non-decreasing
        // order of their weight. If we are not allowed
        // to change the given graph, we can create
```

```
subset[] subsets = new subset[V];
for (i = 0; i < V; ++i)
    subsets[i] = new subset();

// Create V subsets with single elements
for (int v = 0; v < V; ++v) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment
    // the index for next iteration
    Edge next_edge = new Edge();
    next_edge = edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge does't cause cycle,
    // include it in result and increment the index
    // of result for next edge
    if (x != y) {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display
// the built MST
Console.WriteLine("Following are the edges in "
                  + "the constructed MST");

int minimumCost = 0
for (i = 0; i < e; ++i)
{
    Console.WriteLine(result[i].src + " -- "
                      + result[i].dest
                      + " == " + result[i].weight);
  minimumCost += result[i].weight;
}
```

```
    // Driver Code
    public static void Main(String[] args)
    {

        /* Let us create following weighted graph
                10
            0--------1
            | \ |
        6| 5\ |15
            | \ |
            2--------3
                4 */
        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        Graph graph = new Graph(V, E);

        // add edge 0-1
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = 10;

        // add edge 0-2
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 6;

        // add edge 0-3
        graph.edge[2].src = 0;
        graph.edge[2].dest = 3;
        graph.edge[2].weight = 5;

        // add edge 1-3
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 15;

        // add edge 2-3
        graph.edge[4].src = 2;
        graph.edge[4].dest = 3;
        graph.edge[4].weight = 4;

        // Function call
        graph.KruskalMST();
    }
}
```

C++

```
// C++ program for Kruskal's algorithm
//  to find Minimum Spanning Tree of a
//  given connected, undirected and weighted
//  graph
#include <bits/stdc++.h>
using namespace std;

// a structure to represent a
// weighted edge in graph
class Edge {
public:
    int src, dest, weight;
};

// a structure to represent a connected,
// undirected and weighted graph
class Graph {
public:

    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[E];

    return graph;
}

// A structure to represent a subset for union-find
class subset {
public:
```

```
// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent
            = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's
// algorithm
```

```
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
          myComp);

    // Allocate memory for creating V ssubsets
    subset* subsets = new subset[(V * sizeof(subset))];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E)
    {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the
    // built MST
    cout << "Following are the edges in the constructed "
            "MST\n";
    int minimumCost = 0;
    for (i = 0; i < e; ++i)
    {
```

```
    // return;
    cout << "Minimum Cost Spanning Tree: " << minimumCost
         << endl;
}

// Driver code
int main()
{
    /* Let us create following weighted graph
            10
        0--------1
        | \ |
    6| 5\ |15
        | \ |
        2--------3
            4 */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;


    // Function call
```

```
// This code is contributed by rathbhupendra
```

Output

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Time Complexity: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most $O(V^2)$, so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

References:
http://www.ics.uci.edu/~eppstein/161/960206.html
http://en.wikipedia.org/wiki/Minimum_spanning_tree
This article is compiled by Aashish Barnwal and reviewed by the GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Next

Prim's Minimum Spanning Tree (MST) | Greedy Algo-5

RECOMMENDED ARTICLES

Page : 1 2 3

01 Prim's Minimum Spanning Tree (MST) | Greedy Algo-5
18, Nov 12

02 Boruvka's algorithm for Minimum Spanning Tree
29, Mar 11

03 Reverse Delete Algorithm for Minimum Spanning Tree
18, Feb 17

04 Spanning Tree With Maximum Degree (Using Kruskal's Algorithm)
22, Mar 19

05 Maximum Spanning Tree using Prim's Algorithm
18, Jan 21

06 Greedy Algorithm to find Minimum number of Coins
19, Aug 15

07 Minimum number of subsequences required to convert one string to another using Greedy Algorithm
27, Feb 20

08 Applications of Minimum Spanning Tree Problem
04, Mar 11

Article Contributed By :

GeeksforGeeks

Current difficulty : Hard

Easy Normal Medium Hard Expert

Improved By : MoustafaElsayed, rathbhupendra, reddybharathab426, mkshah141, dhyeydhyey7654, hitesh_tripathi, neerajx86, siddhantgore

Article Tags : Kruskal, Kruskal'sAlgorithm, MST, Graph, Greedy

Practice Tags : Greedy, Graph

Improve Article Report Issue

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

GeeksforGeeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org