Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm)

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 and undirected graph is a spanning tree with 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 Minimum Spanning Tree?

See this for applications of MST.

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

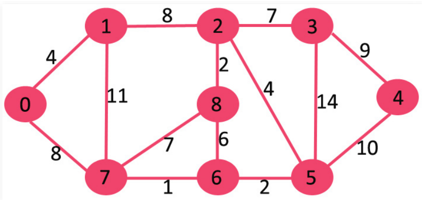
```
1. Sort all the edges in non-decreasing order of their weight.

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.
```

The step#2 uses Union-Find algorithm to detect cycle. So we recommend to read following post as a prerequisite.

Union-Find Algorithm | Set 1 (Detect Cycle in a Graph)

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.



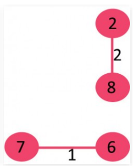
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

```
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
8         0      7
8         1      2
9         3      4
10        5      4
11        1      7
14        3      5
```

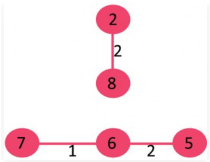
Now pick all edges one by one from sorted list of edges

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

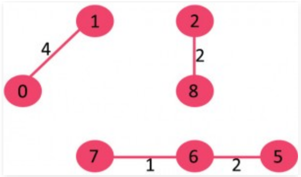
2. *Pick edge 8-2:* No cycle is formed, include it.



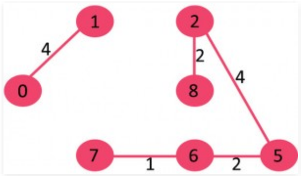
3. *Pick edge 6-5:* No cycle is formed, include it.



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

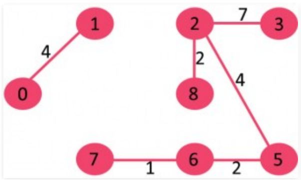


5. *Pick edge 2-5:* No cycle is formed, include it.



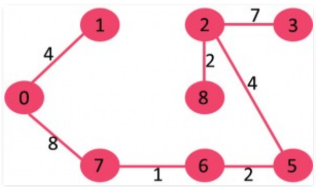
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



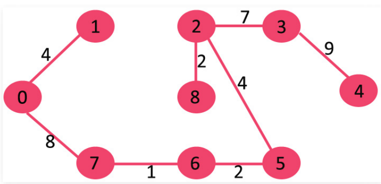
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in 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++ Java Python

```
# Python program for Kruskal's algorithm t
# of a given connected, undirected and wei

from collections import defaultdict

#Class to represent a graph
class Graph:

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


    # 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 ar
    # (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 se
    # (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 r
        # (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
        # its rank by one
        else :
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST u
    def KruskalMST(self):

        result =[] #This will store the re

        i = 0 # An index variable, used fo
        e = 0 # An index variable, used fo

        #Step 1:  Sort all the edges in no
        # weight.  If we are not allowed t
        # can create a copy of graph
        self.graph =  sorted(self.graph,ke
        #print self.graph

        parent = [] ; rank = []

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

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

            # Step 2: Pick the smallest ed
            # 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'
            # in result and increment the
            if x != y:
                e = e + 1
                result.append([u,v,w])
                self.union(parent, rank, x
            # Else discard the edge

        # print the contents of result[] t
        print "Following are the edges in
        for u,v,weight  in result:
            #print str(u) + " -- " + str(v
            print ("%d -- %d == %d" % (u,v


g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()

#This code is contributed by Neelam Yadav
```

Run on IDE

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

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

References:

http://www.ics.uci.edu/~eppstein/161/960206.html