Kruskal's Algorithm

Finding the shortest path to lay cables across a city or group of cities.

PRAVEEN
RA2011003010886
RAGHAVA VARMA
RA2011003010887
SAI MANIKANTA PITCHAIAH
RA2011003010893

Most of the cable network companies use the Disjoint Set Union data structure in Kruskal’s algorithm to find the shortest path to lay cables across a city or group of cities.

What are Disjoint Sets?

A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets.

With the help of disjoints sets, you can keep a track of the existence of elements in a particular group.

Let’s say there are 6 elements A, B, C, D, E, and F. B, C, and D are connected and E and F are paired together.

This gives us 3 subsets that have elements (A), (B, C, D), and (E, F).

Disjoint sets help us quickly determine which elements are connected and close and to unite two components into a single entity.

A disjoint set data structure consists of two important functions:

Find() – It helps to determine which subset a particular element belongs to.

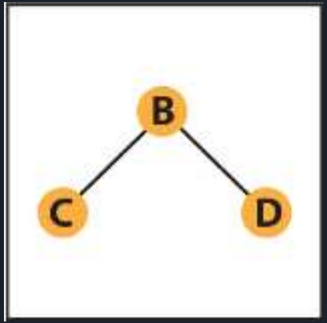
It also helps determine if the element is in more than one subset.

Union() – It helps to check whether a graph is cyclic or not. And helps connect or join two subsets.

Implementation of Disjoint Set

For the previous example, we assume that for the set (B, C, D), B is a parent node.

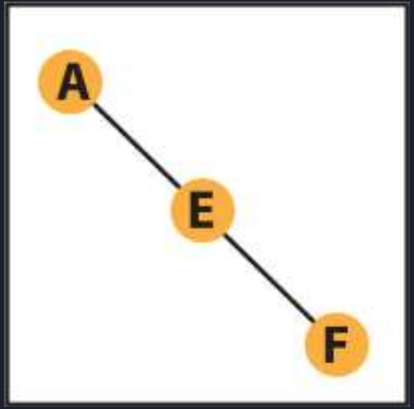
For the disjoint set, we keep a single representative for each node.



If we search for an element in a particular node, it leads us to the parent of that particular node.

Therefore, when you search for D, the answer would be B.

Similarly, we can connect the subset (A) to (E, F) which would result in node A as the parent node.



Now we have two subsets, but both B and A don’t have any parent node.

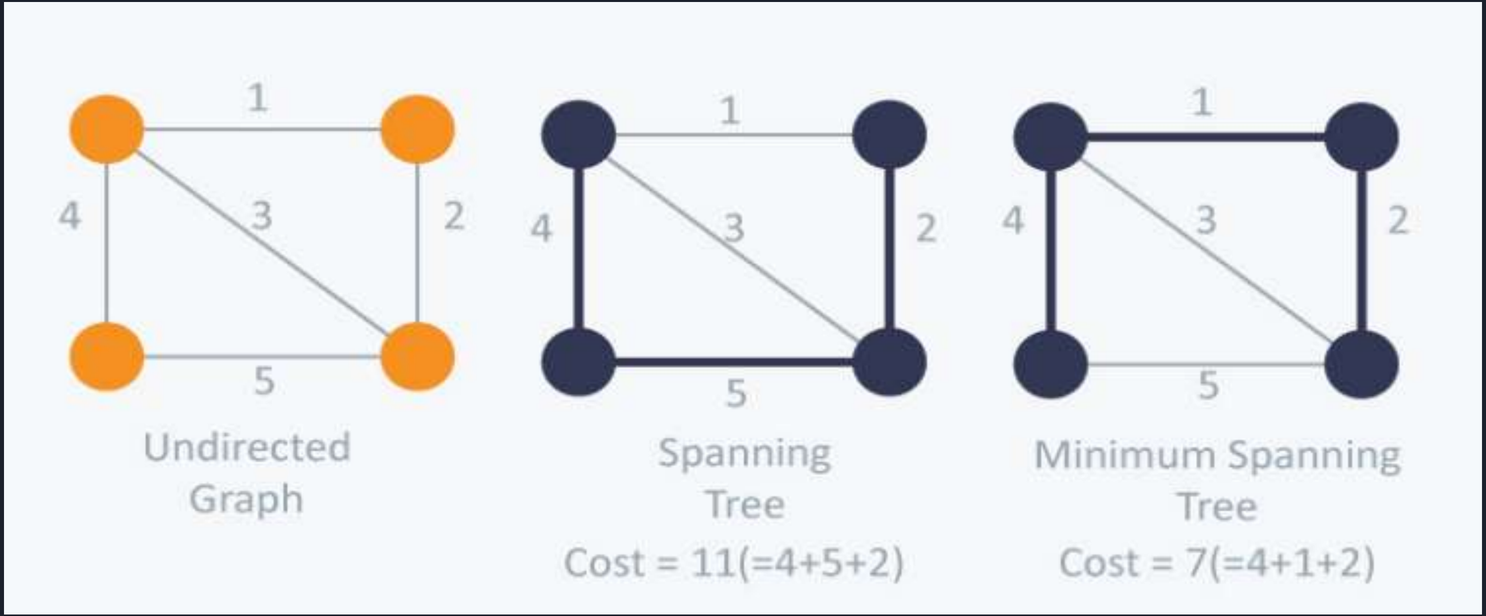
Each tree is an independent disjoint set, that is if two or more elements are in the same tree, they are part of the same disjoint set, else they are independent.

So if for a particular tree B is a representative, then Parent[i]=B.

If B is not a representative, we can move up the tree to find the parent or representative for the tree.

What is Kruskal’s algorithm?

Spanning tree is the sum of weights of all the edges in a tree.
A minimum spanning tree (MST) is one which costs the least among all spanning trees.
Here is an example of a minimum spanning tree.



Kruskal’s Algorithm and Prim’s minimum spanning tree algorithm are two popular algorithms to find the minimum spanning trees.

Kruskal’s algorithm uses the greedy approach for finding a minimum spanning tree.

Kruskal’s algorithm treats every node as an independent tree and connects one with another only if it has the lowest cost compared to all other options available.

Step to Kruskal’s algorithm:

- Sort the graph edges with respect to their weights.
- Start adding edges to the minimum spanning tree from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which don’t form a cycle—edges which connect only disconnected components.

Or as a simpler explanation,

Step 1 – Remove all loops and parallel edges

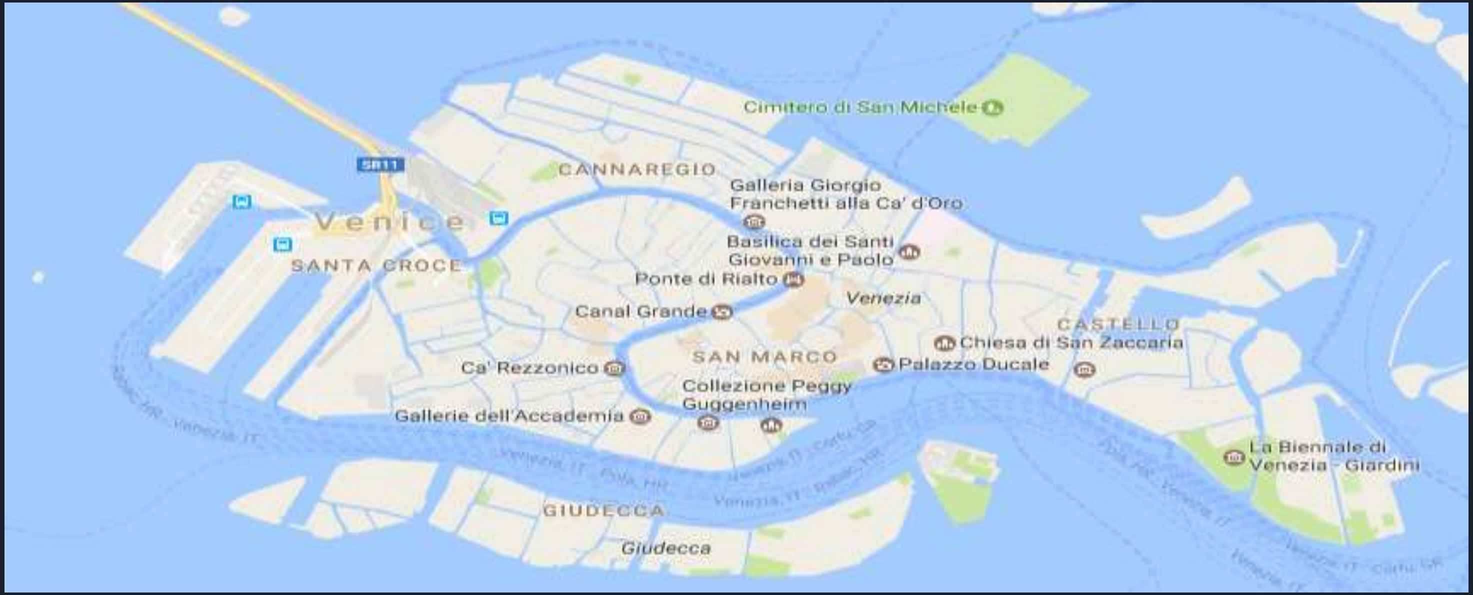
Step 2 – Arrange all the edges in ascending order of cost

Step 3 – Add edges with least weight

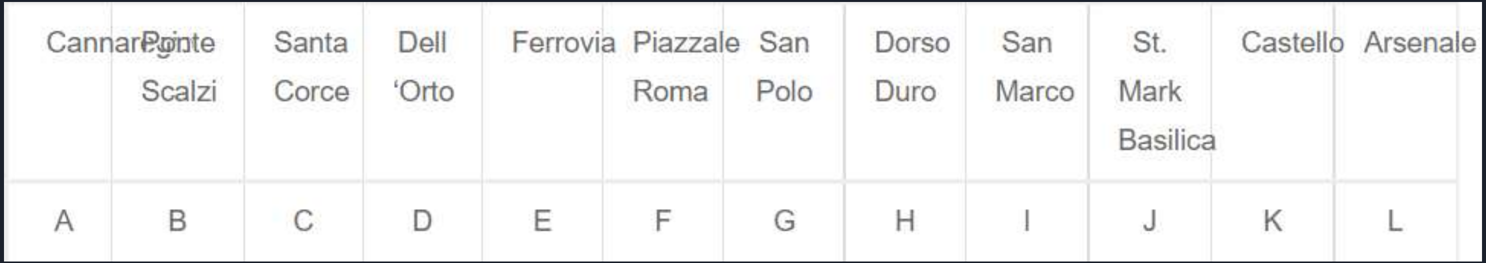
But how do you check whether two vertices are connected or not? That’s where the real-life example of Disjoint Sets come into use.

Kruskal’s algorithm example in detail

I am sure very few of you would be working for a cable network company, so let’s make the Kruskal’s minimum spanning tree algorithm problem more relatable.
On your trip to Venice, you plan to visit all the important world heritage sites but are short on time. To make your itinerary work, you decide to use Kruskal’s algorithm using disjoint sets.
Here is a map of Venice.



Let’s simplify the map by converting it into a graph as below and naming important locations on the map with letters and distance in meters (x 100).

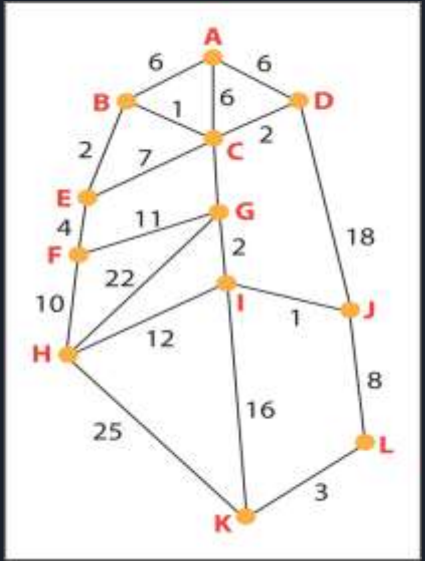


Cannaregio	Ponte Scalzi	Santa Corce	Dell ‘Orto	Ferrovia	Piazzale Roma	San Polo	Dorso Duro	San Marco	St. Mark Basilica	Castello	Arsenale
A	B	C	D	E	F	G	H	I	J	K	L

Step 1- Remove all loops and parallel edges

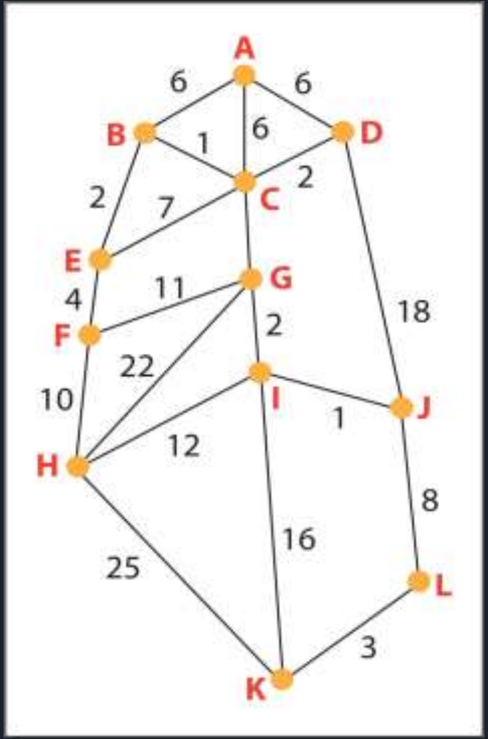
So for the given map, we have a parallel edge running between Madonna dell’Orto (D) to St. Mark Basilica (J), which is of length 2.4kms(2400mts).

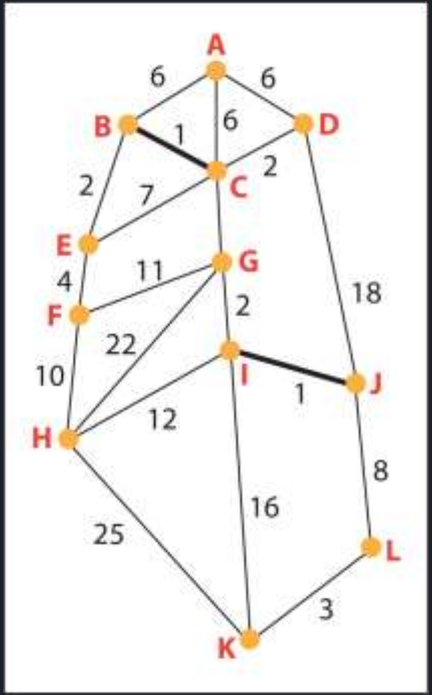
We will remove the parallel road and keep the 1.8km (1800m) length for representation.



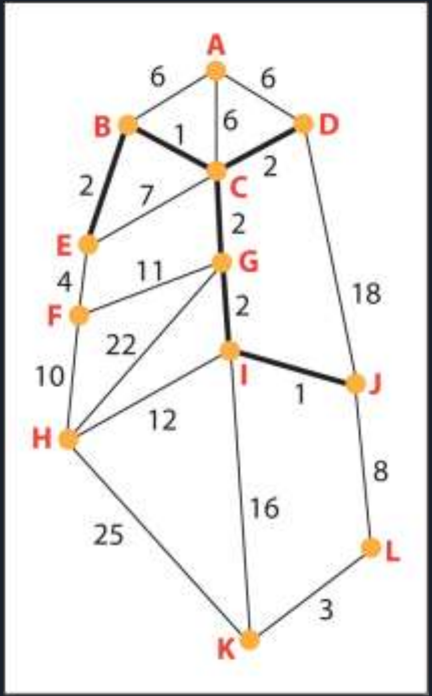
Step 2 – Arrange all the edges on the graph in ascending order. Kruskal's algorithm considers each group as a tree and applies disjoint sets to check how many of the vertices are part of other trees.

Step 3 – Add edges with least weight; we begin with the edges with least weight/cost. Hence, B, C is connected first considering their edge cost only 1.

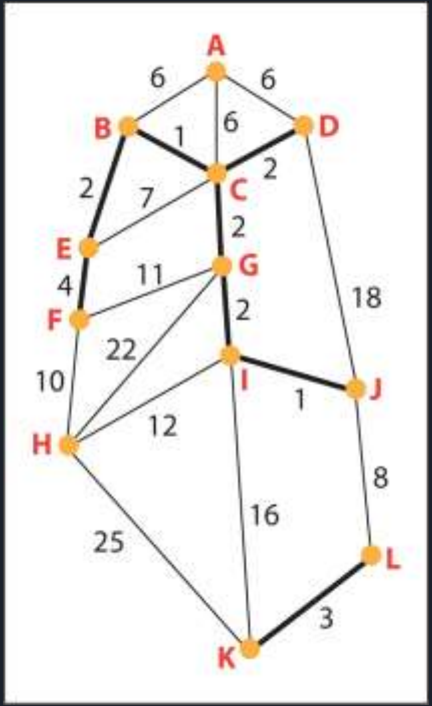




I, J has cost 1; it is the edge connected next.

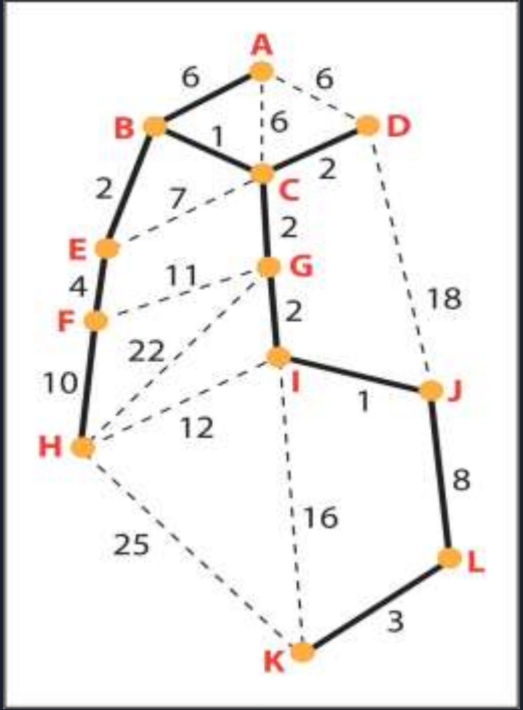


Then, we connect edges with weight = 2.



Similarly, we connect node K, L which has an edge with weight = 3.

As given in the table above, all the edges are connected in ascending order, ensuring no loop or cycle is formed between 2 vertices.
This gives us the following graph, which is the minimum spanning tree for the given problem.



Here Kruskal’s algorithm using C++

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>
using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];
void initialize()
{
for(int i = 0;i < MAX;++i)
    id[i] = i;
}
int root(int x)
{
   while(id[x] != x)

{
        id[x] = id[id[x]];
        x = id[x];
   }
   return x;
}
void union1(int x, int y)
{
   int p = root(x);
   int q = root(y);
   id[p] = id[q];
}
long long kruskal(pair<long long,
pair<int, int> > p[])
{
   int x, y;
   long long cost, minimumCost = 0;
   for(int i = 0;i < edges;++i)
   {
```

```
// Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
    }
    return minimumCost;
}
int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
   for(int i = 0;i < edges;++i)
    {
```

```
cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
   }
```

THANK YOU
THE END