**MST**

Spanning tree : a connected subgraph S of graph G(V,E) is said to be spanning tree iff

1. S should contain all vertices of G.
2. S should contain |v| - 1 edges.
3. While connecting edges it should not form cycle.

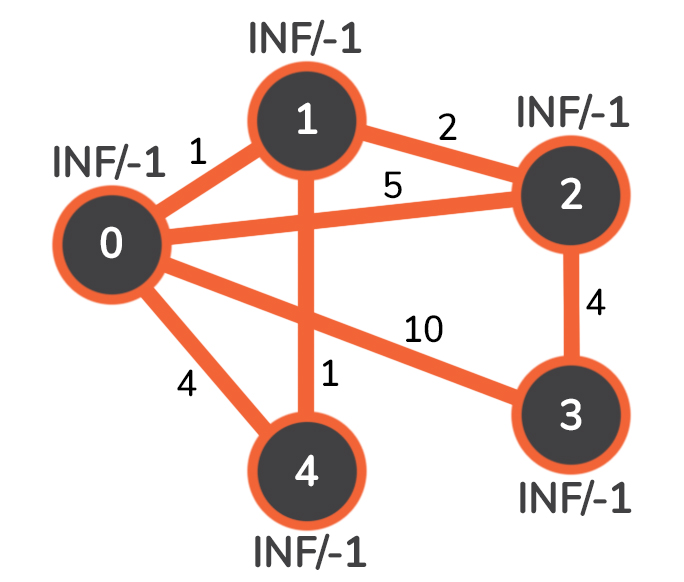
**Prim’s Minimum Spanning Tree (MST)**

 Assuming we have a weighted graph G with a set of vertices (nodes) V and a set of edges E:

* We choose one of the nodes s as the starting node, and set the distance from s to s as 0.
* We'll assign a number from node s to every other node, marking it as infinity at the beginning. This number will change and update as we progress along the algorithm.
* Every node s will also have a number representing the "parent" node, from which we connect it in the MST. This number is initialized as -1, and every other node *except* the starting node will have a number different from -1 associated with it by the end of Prim's algorithm.
* For every node s we'll find the minimum edge connecting a node that is ***not*** already included in the MST. Since Prim's is a greedy algorithm, once we enter the node we're sure that we've chosen the shortest path connecting it to it's parent. We repeat this step until all of the nodes are added to the MST.
* Finally, we loop through the our MST array and add up the edges, getting the value of the MST.

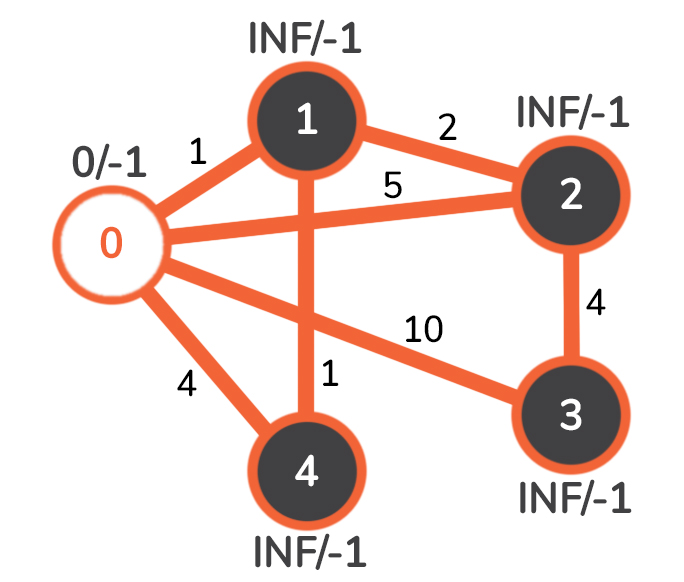
**Visualizing Prim's Algorithm**

Let's quickly visualize a simple example - and *manually* use Prim's Algorithm to find a Minimum Spanning Tree on the following graph:



We'll have 5 nodes, numbered 0 through 4, and on each of the edges the number represents the weight of that edge. Let's describe the INF/-1 pair: -1 at the beginning represents the parent from which there is an edge connecting to the current node that is of weight INF. Of course, as the algorithm progresses, these values will also get updated.

Let's say that 0 will be our starting node. We mentioned earlier that when we choose our starting node, we need to set the distance from itself as 0. Since 0 is the node with the minimal edge to itself, we can safely assume that 0 belongs in the MST and we'll add it. After that little change the graph looks as follows:



White nodes represent the ones we added to the MST.

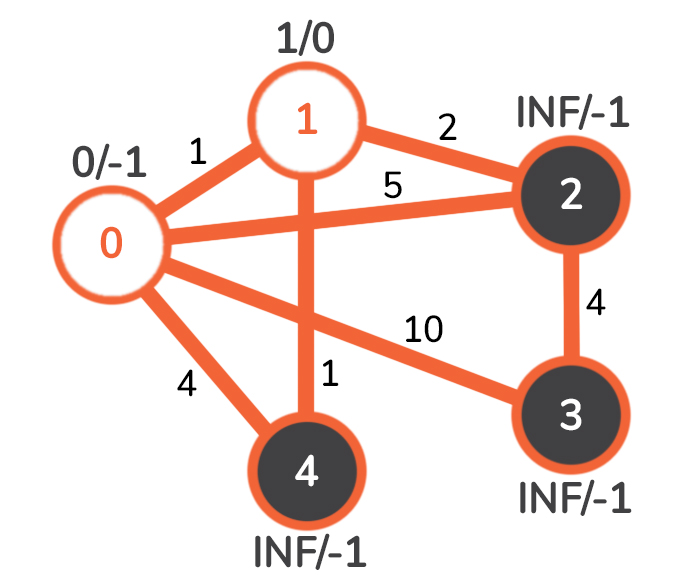
The next step is the one that makes Prim's algorithm what it is. We loop through all of the neighbours of the node 0, checking for a few things along the way:

1. If the edge exists at all
2. If the neighbour node is already added to the MST
3. If the cost of the edge leading to the neighbour is lower than the current smallest-cost edge leading to that neighbour

The first neighbour of 0 is 1. The edge connecting them has a weight of 1. The edge exists, and the current node 1 is not in the MST, so the only thing left is to check if the edge from 0 to 1 is the smallest weighted edge leading to node 1. Obviously, 1 is less than INF, so we update the distance/parent pair of node 1 to 1/0.

We follow exactly the same steps for every other neighbour of node 0, after which we choose the node with the minimal edge weight to be added to the MST, and mark it blue. That node here is 1.

Now we have the following graph:



The node we're considering now is 1. As we've done with node 0, we check all of the neighbors of node 1.

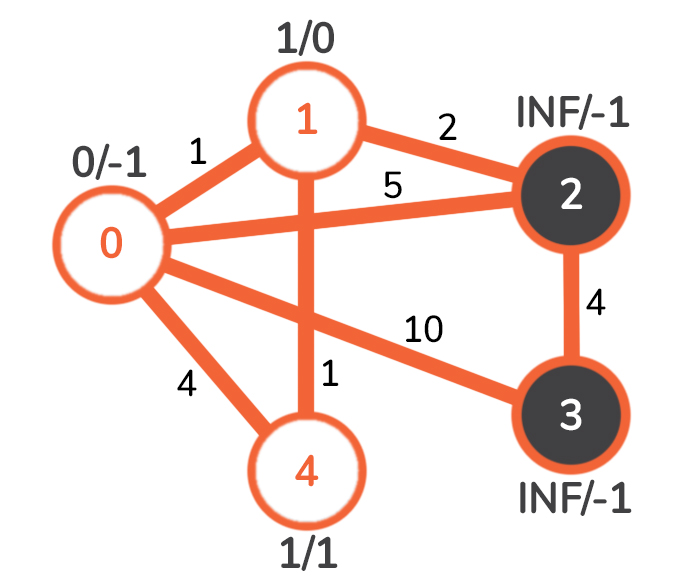
Node 0 is already added to the MST, so we skip that one.

Node 2 is the next neighbor, and the weight of the edge leading to it from node 1 is 2. This edge has a smaller weight than the one that previously led to that node, which had weight of 5 and came from node 0.

The same is with the other neighbor node 4: the weight of the edge leading to it from node 1 is 1, and previously the smallest weighted edge leading to node 4 from node 0 was 4.

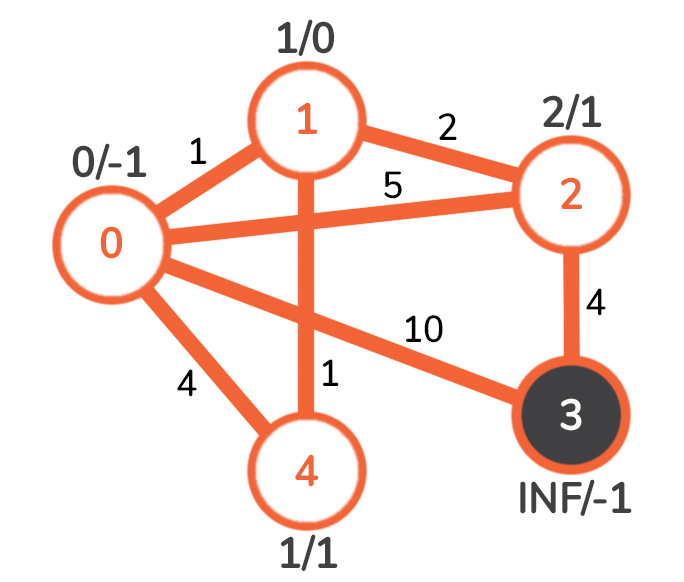
We choose the next node that isn't added to the MST and has the smallest weighted edge from node 1. That node here is node 4.

After the update we have the following graph:

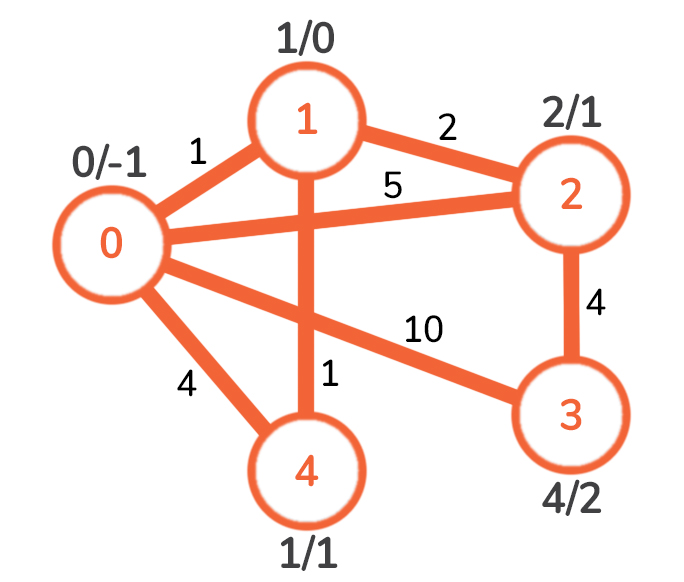


As we consider node 4, we see that we can't update any of the current edges. Namely, both neighbors of node 4 already belong to the MST, so there is nothing to update there, and we just move along in the algorithm without doing anything in this step.

We continue looking for a node that is connected to a node belonging to the MST and has the smallest weighted edge possible. That node is currently 2, and it connects to node 1 via the edge that has the weight of 2. The graph looks as follows:



Both nodes 0 and 1 already belong in the MST, so the only possible node we can go to is 3. The weight of the edge leading to node 3 from node 2 is 4, which is obviously less than the previous 10 leading from node 0. We update that, getting the following graph:



With this, we've visited and added all of the existing nodes to the MST, and because Prim's is a greedy algorithm, this means we've found our MST.

Let's recollect; the edges that were added to the array that keeps track of our MST are the following:

* Edge 0-1 of weight 1
* Edge 1-2 of weight 2
* Edge 1-4 of weight 1
* Edge 2-3 of weight 4

The time complexity of Prim's algorithm is O((|E| + |V|)log|V|), where |E| is the number of edges in the graph, and |V| is the number of vertices(nodes) in the graph.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/mst/PrimsAlgorithmMST.java>

**Prim’s MST for Adjacency List Representation**

The time complexity for the matrix representation is O(V^2).

In this post, O(ELogV) algorithm for adjacency list representation is discussed.

**Algorithm:**

1. Include the source vertex into mst and set weight value to 0 so that it can be extracted first from the priority queue.
2. Add all the edges into priority queue
3. While priority queue is not empty
4. Extract the node from priority queue with min weight
5. Include that node into mst
6. For all the adjacent node of the extracted node do the following operation
7. Check if it is present in the mst or not
8. If it is not present then

**if** (currentAdjEdge.weight < edge[currentAdjEdge.destination].weight) {

pq.remove(edge[currentAdjEdge.destination]);

edge[currentAdjEdge.destination].weight = currentAdjEdge.weight;

pq.add(edge[currentAdjEdge.destination]);

parent[currentAdjEdge.destination] = minWeightEdge.destination;

}

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/mst/PrimsAlgorithAdjacencyList.java>

**Kruskal’s Minimum Spanning Tree Algorithm**

Algorithm

1) create a priority queue and add all the edges into it

2) while queue is not empty

3) do the poll() it will return vertex with min weight

4) check if adding the extracted vertex will form circle or not

5) if it doesn’t form cycle then add this edge into mst

6) do the union operation

Whole for loop looks like bellow

**while** (!pq.isEmpty()) {

Edge edge = pq.poll();

// check if adding this edge creates a cycle

**int** x = find(parent, edge.source);

**int** y = find(parent, edge.destination);

// here parents are diff it means both belongs to diff sets

**if** (x != y) {

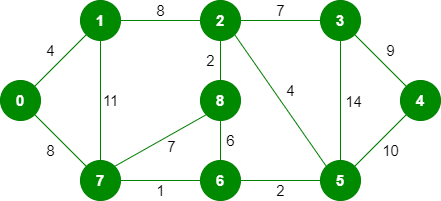
mst.add(edge);

union(parent, x, y);

}

}

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 edge.

After sorting based on weight:

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

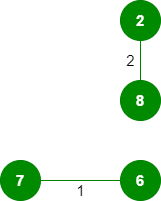
11 1 7

14 3 5

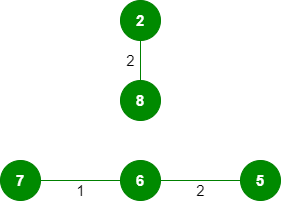
Now pick all edges one by one from the sorted list of edges   
**1.** *Pick edge 7-6:* No cycle is formed, include it. 

Kruskal’s Minimum Spanning Tree Algorithm

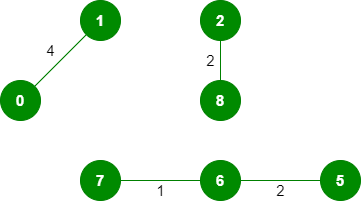
**2.** *Pick edge 8-2:* No cycle is formed, include it. 



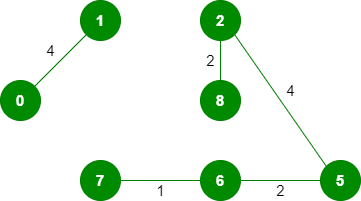
**3.** *Pick edge 6-5:* No cycle is formed, include it. 



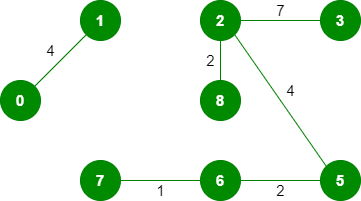
**4.** *Pick edge 0-1:* No cycle is formed, include it. 



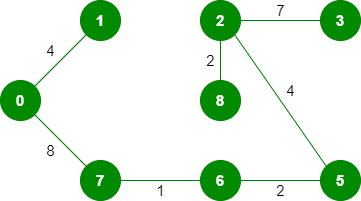
**5.** *Pick edge 2-5:* No cycle is formed, include it. 



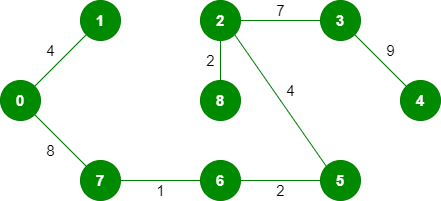
**6.***Pick edge 8-6:*Since including this edge results in the cycle, discard it.  
**7.** *Pick edge 2-3:* No cycle is formed, include it. 



**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.

**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(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/mst/KrushkalMst.java>