**Kruskal’s algorithm to compute MST**

1) sort all edges in ascending order of cost(weight)

2) add the next edge of the sorted list into a set T unless doing so would create a cycle in T

3) T will be the MST once T contains all vertices or (|v| - 1) edges

a)

We first sort all the edges by weight in ascending order, with E being the set of edges, wgt being the weight of each edge.

E = { (0 – 1), (2 – 5), (0 – 2), (4 – 6), (2 – 3), (6 – 7), (3 – 5), (5 – 6), (4 – 5), (0 – 3), (5 – 7), (3 – 4),

wgt: 4 5 6 7 8 9 10 11 14 16 18 21

(2 – 7), (1 – 7) }

23 24

**# of vertices: 8** (we stop when MST contains all 0 – 7 vertices, or 7 edges)

**# of edges: 14** (total number of edges in the input Graph)

Now that the edges have been sorted in ascending order, our set MST is currently empty:

MST = { }

We will add any edge starting from the smallest weight to MST if the edge does not create a cycle in the current MST.

Step 1:

MST = { }

e(0 – 1) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1) }

Step 2:

MST = { e(0 – 1) }

e(2 – 5) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1), e(2 – 5) }

Step 3:

MST = { e(0 – 1), e(2 – 5) }

e(0 – 2) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1), e(2 – 5), e(0 – 2) }

Step 4:

MST = { e(0 – 1), e(2 – 5), e(0 – 2) }

e(4 – 6) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6) }

Step 5:

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6) }

e(2 – 3) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3) }

Step 6:

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3) }

e(6 – 7) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3), e(6 – 7) }

Step 7:

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3), e(6 – 7) }

e(3 – 5) <- current edge, creates a cycle in MST (2 – 3 – 5), skip

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3), e(6 – 7) }

Step 8:

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3), e(6 – 7) }

e(5 – 6) <- current edge, does not create a cycle, add into MST

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3), e(6 – 7), e(5 – 6) }

The MST is finished since there are now (V – 1) number of edges.

So the MST result is:

MST = { e(0 – 1), e(2 – 5), e(0 – 2), e(4 – 6), e(2 – 3), e(6 – 7), e(5 – 6) }

b)

**Kruskal’s MST Correctness Proof**

Proposition: Kruskal’s algorithm computes the MST

Proof:

Case 1: suppose that adding e to MST create a cycle

- e is the max-cost edge in cycle C

- e is not in the MST due to the cycle property

Cycle Property: let C be any cycle in a graph, and let f be the max-cost edge belonging to C,

then the MST does not contain f

Since the list of edges is sorted, the edge f will always be the max-cost edge belonging to a cycle in the MST. Same thing can be said with the following case with it instead being a min-cost edge.

Case 2: suppose adding e = (v,w) to T does not create a cycle in T

- let S be the vertices in v’s connected component

- w is not in S

- e is the min-cost edge with exactly one end point in S

- e is in MST due to the cut property

Cut Property: let S be any subset of vertices and e be the min-cost edge with exactly one end point in S, then the MST contains e

c)

The most efficient data structure that is applicable in the implementation of Krustal’s algorithm is Union-Find (UF). With the following pseudocode for his algorithm, we can analyze the time complexity using the UF class.

The Union-Find(UF) data structure

- maintain a set for each connected component

- if v and w are in the same component already, then adding e(v, w) create a cycle

- if not creating a cycle, to add e(v, w) to T, you merge the sets containing v and w

public class Kruskal

{

private Set<Edge> mst = new HashSet<Edge>();

public Krustal(weightedGraph G)

{

Edge [] edges = G.edges();

Arrays.sort(edges, Edge.BY\_WEIGHT);

UnionFind uf = new UnionFind(G.v());

for(Edges e: edges())

{

if(!uf.find(e.either(), e.other(e.either())))

{

mst.add(e);

uf.unite(e.either(), e.other(e.ether()));

}

}

}

public Iterable<Edge> mst()

{

return mst;

}

}

With V being vertices and E edges:

1) Sorting the array of edges by their weight takes O(E logE) with quick sort or merge sort

- this happens only once, so we add this to the GRF

2) The for loop will iterate at worst case scenario, the number of edges E in the graph.

GRF = E + (E logE)

3) For each edge we will have to check the addition of the correct edges into the components

This will take O(log V) since we only need 1 component with all vertices 1 time through adding and merging. Now our GRF is;

GRF = (E logV) + (E logE)

E is bigger in worst case scenarios so we can drop first term, however, it all depends on if the edges have been sorted or not. Thus we have two scenarios:

With sorted edges: time complexity is O(E logV)

Without sorted edges: time complexity is O(E logE)