**Pseudocode for Kruskal: -**

// Cost – 2 dimensional matrix, cost[u][v] is cost of edge (u,v);

// n no. of vertices.

//‘t’ set of edges in minimum spanning tree – it us a 2D array. t[1…..n][1….2]

// initially each vertex is in different set.

1. Construct a minimum heap out of edge costs using heapify.
2. For( i = 1 to n) do parent[i] = -1;

// each vertex is in different set.

1. I=0; mincost=0;
2. While(i

)>

1. {
2. Delete minimum cost edge (u,v) from heap
3. Reheapify using adjust.
4. J=find(u); k=find(v);
5. If (j!=k) then\
6. {
7. I++; t[i][1]=u,t[i][2]=v;
8. Union(j,k); mincost = mincost+cost[u][v];
9. }
10. }

Algorithm SimpleUnion (I, j) // i & j are roots of sets.

{

p[i] = j;

}

Algorithm SimpleFind (i ) // return root of ‘i’

{

while (p[i] >=0) do i=p[i];

return I;

}

If we want to perform n-1 unions then can be processed in Linear time O(n)

The time required to process a find for an element at level ‘i’ of a tree is O(i), so total time needed to process n finds is**O(n^2)**

One can improve the performance of our union & find by avoiding creation of degenerating trees.

We apply weighting rule for Union, which says that if the number of nodes in the tree with root i is less than the number of nodes in the tree with root j, then make j as the parent of i, otherwise make i, as the parent of j. Thus we avoid degenerated/skewed trees.

Similarly we use CollapseFind , to improve the performance of find. In CollapsingFind, we are required to go up to reach the root node and then reset the links. Every first find has to go through intermediate nodes to go up to root node, but every next find on similar node will require only one link to go up, thus reducing the cost cost of find. Collapsing rule says that, if j is a node on the path from i to its root node and p[i] is not root[i], then set p[j] to root[i].

Weighted union(I,j)

{

Temp=p[i]+p[j];

If(p[i]>p[j]) // i has fewer nodes

{

p[i]=j, p[j]=temp;

}

Else

{

p[j]=I; // j has fewer nodes

p[i]=temp;

}

}

Collapse Find(i)

{

r=I;

While(p[r]>=0)

r=p[r];

while(i=r)

{

s=p[i];

p[i]=r;

i=s;

}

}

**Analysis of Kruskal algorithm :**

Time required to construct the initial heap is log E.

Removal of edge from heap requires again log E and the edge is removed in the while loop which runs for E times, hence the time complexity of algorithm is O(E log E). Here since we have used sets to represent trees and we have used efficient find and union algorithms which take almost O(1) time.