Minimum Spanning Tree using Heap

Maumita Chakraborty [1],Rahul Singh [2], Ruchi Mehta [3]

Abstract— Minimum spanning trees are one of the most important primitives used in graph algorithms. They find applications in numerous fields ranging from taxonomy to image processing to computer networks. In this paper, we present a different approach or algorithm to find the minimum spanning tree (MST) for large graphs based on boruvka's algorithm. We analyze the CPU timing of the algorithm and compare it with the existing algorithms of finding MST under certain assumptions. Finally, we compare the performance of the three algorithms on a set of graph instances.

Keywords: graphs, weighted graph, undirected graphs, minimum spanning tree, union-find algorithm, kruskal's algorithm, prim's algorithm, heap, stack,boruvka's algorithm.

1 Introduction

Graph is a set of edges and vertices represented as *G (V, E)* and tree is a connected graph having *n* vertices and *n-1* edges. Minimum spanning tree is a connected subset of graph having *n* vertices and *n-1* edges so basically it is a tree but the total weight of the minimum spanning tree is always less than or equal to weight of any possible subset of connected graph having *n* vertices and *n-1* edges which is a tree.

Here we are showing some application for minimum spanning trees. One example would be a telecommunications company or electrical companies which are trying to lay off cables in city or town. To connect the whole city or the part of the city, there must be a graph representing which points are connected by which one along the direction of some path. Some of those paths might be more expensive, because they are longer, or might be some difficulty in laying the cables; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight graph – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A *spanning tree* for that graph would be a subset of those paths that has no cycles but still connects to every point directly or via some other point; there might be several spanning trees possible. A *minimum spanning tree* would be one with the lowest total cost, thus would represent the least expensive path for laying the cable. Boruvka's algorithm[1] was also invented to construct an efficient electric network.
The proposed algorithm is the implementation of Boruvka's algorithm and hence could be said as the extension of the mentioned algorithm.

2. Related Work

2.1 PRIM'S ALGORITHM:

In computer science, **Prim's algorithm [**2**]** is basically a greedy algorithm that is used to find a minimum spanning tree for a weighted undirected graph. What we can say is that it finds that subset of edges forming a tree that includes all the vertices, such that the total weight of edges is kept minimum.
The algorithm starts by taking any arbitrary vertex as the starting vertex and then adding the edges in such a manner that they produce the minimum weight.

The algorithm may informally be described as performing the following steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

In more detail, it may be implemented following the pseudo code below [2].

```
1. Associate with each vertex $v$ of the graph a number $C[v]$ (the cheapest cost of a connection to $v$)
   and an edge $E[v]$ (the edge providing that cheapest connection). To initialize these values, set all
   values of $C[v]$ to $+\infty$ (or to any number larger than the maximum edge weight) and set each $E[v]$
   to a special flag value indicating that there is no edge connecting $v$ to earlier vertices.
2. Initialize an empty forest $F$ and a set $Q$ of vertices that have not yet been included in $F$
   (initially, all vertices).
3. Repeat the following steps until $Q$ is empty:
      a. Find and remove a vertex $v$ from $Q$ having the minimum possible value of $C[v]$
      b. Add $v$ to $F$ and, if $E[v]$ is not the special flag value, also add $E[v]$ to $F$
      c. Loop over the edges $vw$ connecting $v$ to other vertices $w$. For each such edge, if $w$ still
         belongs to $Q$ and $vw$ has smaller weight than $C[w]$, perform the following steps:
            i. Set $C[w]$ to the cost of edge $vw$
            ii. Set $E[w]$ to point to edge $vw$.
4. Return $F$
```

As described in the above pseudo code, the starting vertex for the algorithm will be chosen arbitrarily, because the first iteration of the main loop of the algorithm will have a set of vertices in Q that all have equal weights, and the algorithm will automatically start a new tree in F when it completes a spanning tree of each connected component of the input graph. The algorithm may be modified to start with any particular vertex *s* by setting *C*[*s*] to be a number smaller than the other values of *C* (for instance, zero), and it may be modified to only find a single spanning tree rather than an entire spanning forest (matching more closely the informal description) by stopping whenever it encounters another vertex flagged as having no associated edge.

Different variations of the algorithm differ from each other in how the set *Q* is implemented: as a simple linked list or array of vertices, or as a more complicated priority queue data structure. This choice leads to differences in the time complexity of the algorithm. In general, a priority queue will be quicker at finding the vertex *v* with minimum cost, but will entail more expensive updates when the value of *C*[*w*] changes.

TIME COMPLEXITY:

The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight, which can be done using a priority queue.

The following table shows the typical choices:

[2] Table 1: Time complexity using different data structure

Minimum edge weight data structure	**Time complexity**
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

A simple implementation of Prim's, using an adjacency matrix or an adjacency list graph representation and linearly searching an array of weights to find the minimum weight edge, to add requires $O(|V|^2)$ running time. However, this running time can be greatly improved further by using heaps to implement finding minimum weight edges in the algorithm's inner loop.

A first improved version uses a heap to store all edges of the input graph, ordered by their weight. This leads to an $O(|E| \log |E|)$ worst-case running time. But storing vertices instead of edges can improve it still further. The heap should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree (MST) (or infinity if no such edge exists). Every time a vertex *v* is chosen and added to the MST, a decrease-key operation is performed on all vertices *w* outside the partial MST such that *v* is connected to *w*, setting the key to the minimum of its previous value and the edge cost of (*v*,*w*).

Using a simple binary heap data structure, Prim's algorithm can now be shown to run in time $O(|E| \log |V|)$ where $|E|$ is the number of edges and $|V|$ is the number of vertices. Using a more sophisticated Fibonacci heap, this can be brought down to $O(|E| + |V| \log |V|)$, which is asymptotically faster when the graph is dense enough that $|E|$ is $\omega(|V|)$, and linear time when $|E|$ is at least $|V| \log |V|$. For graphs of even greater density (having at least $|V|^c$ edges for some $c > 1$), Prim's algorithm can be made to run in linear time even more simply, by using a *d*-ary heap in place of a Fibonacci heap.

2.2 KRUSKAL'S ALGORITHM

Kruskal's algorithm [3] is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step .This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

This algorithm first appeared in *Proceedings of the American Mathematical Society*, pp. 48–50 in 1956, and was written by Joseph Kruskal [4].

Algorithm

1. Create a forest *F* (a set of trees), where each vertex in the graph is a separate tree.
2. Create a set *S* containing all the edges in the graph
3. While *S* is nonempty and *F* is not yet spanning
4. Remove an edge with minimum weight from *S*
5. If the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

6. At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree

Pseudo code [3]

The following code is implemented with disjoint-set data structure:

```
KRUSKAL (G):
1. A=∅
2. for each v ∈ G.V:
3. MAKE-SET (v)
```

4. **For each** (u, v) in $G.E$ ordered by weight (u, v), increasing
5. **If** FIND-SET $(u) \neq$ FIND-SET (v):
6. $A = A \cup \{(u, v)\}$
7. UNION (u, v)
8. **Return** A

Complexity

Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, where E is the number of edges in the graph and V is the number of vertices, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2 \log V$ is $O(\log V)$.
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V \leq 2E$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure (Union & Find) to keep track of which vertices are in which components. We need to perform $O(V)$ operations, as in each iteration we connect a vertex to the spanning tree, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(V)$ operations in $O(V \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

3. PROPOSED ALGORITHM

We tried to find a algorithm to find minimal spanning tree. For this, we use the cut property and greedy property of minimum spanning tree. First of all we make a new set for each vertex of the graph $G(V,E)$, now as we know that for each node($p \in V$) we have a minimum weight edge which will lead us to new vertex ($q \in V$) ,if the node p and q doesn't belong to same set, union them. Now repeat these steps for all the vertices. In worst case it will lead us to $n/2$ disconnected tree. Now heapify the remaining edges to select minimum edge from the rest of the edges then check the corresponding vertex of the edge belongs to same set or not and if they belong to different set union them and repeat until we have only a single set of graph left.

DESCRIPTION WITH EXAMPLE

Let us consider a graph $G(V, E)$ having 8 vertices and 15 edges and each vertex have its own set, now from the list of vertices select any arbitrary vertex and search for the minimum edge from that vertex, lets say in this example we select vertex A and the minimum edge corresponding to the vertex A is (A,B)now we find that A and B are in different set so we union them and form a new set say set 1(White, background 1, darker 15%).

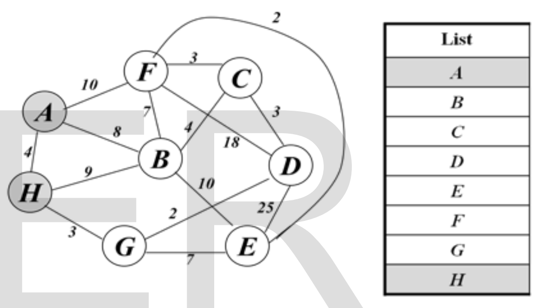


Fig .2. After combining set a and H

Now we select vertex B and the minimum edge from B is (B,C),we check for the set of B and C, as they belong to different set so we Union them and put them in a new set 2 (White, background 1, darker 25%).

.

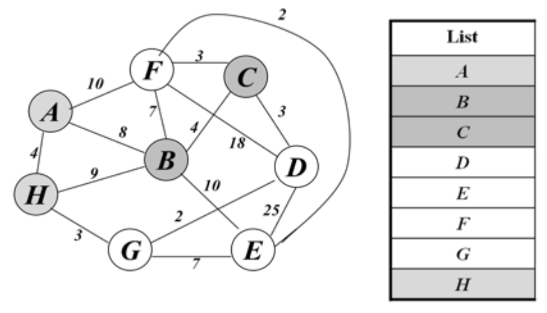
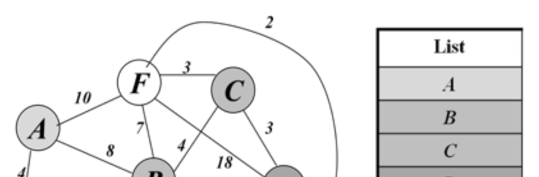


Fig.3. After combining set B and C

N

vertex D is (D,G), as D and G belong to different set so we Union them and put them in a new set say 3(White, background 1, darker 50%).



Now choose the vertex *E* and the minimum edge from the vertex *E* is*(E,F),* as *E* and *F* belong to different set so we Union them and put them in a new set say 4(White ,Background 1).

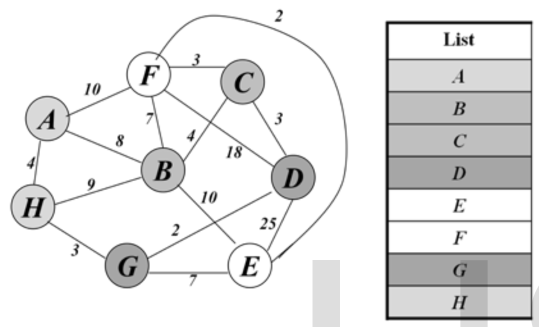


Fig.6. After combining set E and F

Now apply the heapify algorithm to find the least edge and check whether the nodes containing the edge belongs to same set or not, if they belong to same set then discard the edge and also remove from the list of edges otherwise union them and remove the edge from the set of remaining edges.

Table 2: List of Remaining edges

Edges	Weight
F-B	18
B-H	9
C-D	3
G-E	7
F-C	3
A-B	8
H-G	3
B-E	10
B-F	7
D-E	25
A-F	10

Table 3: After applying heapify on the table 1

Edges	Weight
F-C	3
B-H	9
C-D	3
G-E	7
F-B	18
A-B	8
H-G	3
B-E	10
B-F	7
D-E	25
A-F	10

Now select the least edge (*F,C*) which we got after applying heapify on the remaining edges ,as *F* and *C* belong to different set so union the set *F* and *C*. Figure below shows the union of

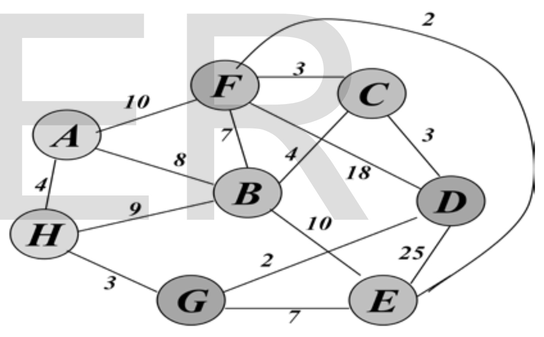


Fig.7.After combining the set 2 and 4

Again apply heapify algorithm to find the next least weighted edge and check whether the nodes containing the edge belongs to same set or not, if they belong to same set then discard the edge and also remove from the list of edges otherwise union them and remove the edge from the set of remaining edges.

Table 4: After removing edge (F,C) and applying heapify again.

Edges	Weight
H-G	3
B-H	9
C-D	3
G-E	7
A-B	8
F-B	18
B-E	10
B-F	7
D-E	25
A-F	10

Now select the least edge (*H*,*G*) which we got after applying heapify on the remaining edges ,as *H* and *G* belong to different set so union the set *H* and *G*. Figure below shows the union of *H* and *G* into the set 1.

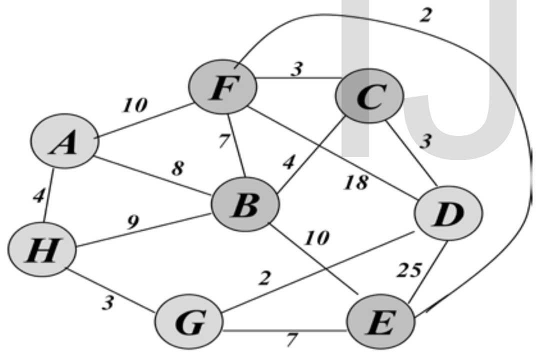


Fig.8. After combining set 1 and 3

Again apply heapify algorithm to find the next least weighted edge and check whether the nodes containing the edge belongs to same set or not, if they belong to same set then discard the edge and also remove from the list of edges otherwise union them and remove the edge from the set of remaining edges.

Table:5 After removing edge (H,G) and applying heapify again.

Edges	Weight
C-D	3
B-H	9
G-E	7
A-B	8
F-B	18
B-E	10
B-F	7
D-E	25
A-F	10

Now select the least edge (C,*D*) which we got after applying heapify on the remaining edges ,as *C* and *D* belong to different set so union the set *C* and *D*. Figure below shows the union of *C* and *D.*

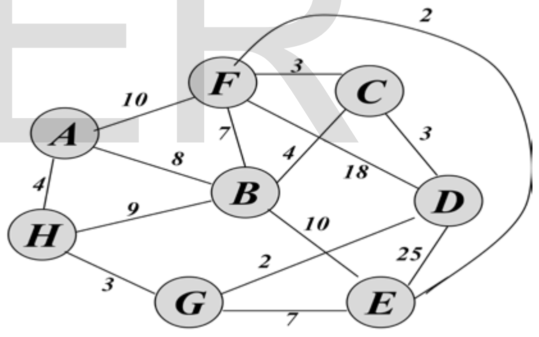


Fig.9.After combining set 1 and 3

Now we have left a single set of the entire vertex with some edges and this is the desire minimal spanning tree.

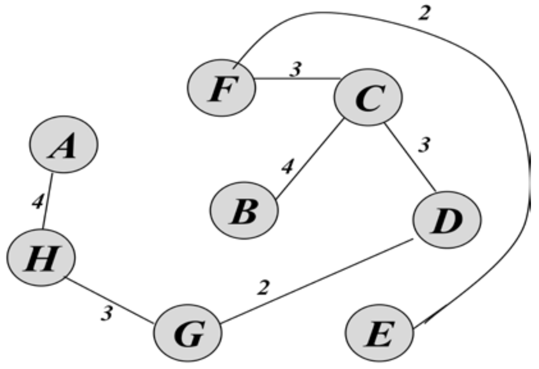


Fig.10. Final MST

Algorithm

Input: A connected graph *G(V,E)* where *V* is set of vertices and *E* is set of edges.

Output: Minimum Spanning Tree(MST) of *G*

```
1.  Create $|V|$ no of sets, $V_s$, for each vertex of $V$ and empty set MST for the edges of the output spanning tree
2.  for each vertex $u$ into set $V$:
3.    Select the least weighted edge $(u, v)$ from $V$
4.    If set of $u$ not equal to set of $v$:
5.      Union the set of $u$ and $v$ and put the edge $(u, v)$ into MST set
6.      $V_s = V_s - 1$
7.      Remove edge $(u, v)$ from set of edge E
8.    End if
9.End for
7.  While number of sets $V_s$ greater than one:
8.    Select least edge $(u, v)$ from set E
9.    If set of $u$ and $v$ are not equal:
10.     Union the set of $u$ and $v$ and put the edge $(u, v)$ into MST set
11.     Remove edge $(u, v)$ from set of edge E
12.     $V_s = V_s - 1$
13.   End if
14. End While
15. Set MST is desired minimal spanning tree.
```

4. Data structures

4.1 Heap

In our algorithm we used heap as a tool to find the next shortest edge from the set of remaining edges, as we know that we have two types of heap min heap and max heap and we have used min heap here . We used heap here because after repeating the step 2-9 in our algorithm it will lead us to *n*/2 disjoint graphs so in order to connect the *n*/2 graph into a single graph we need (*n*/2)*-1* edges and In best case it could be the first *(n/2)-1* least weighted edges so if we use min heap we have to apply heapify algorithm only *(n/2)*-1 time and it will help us to reduce the time complexity, but in worst case we have to move through all the remaining edges.

4.2 Union Find

We have union find algorithm to find the sets of vertices so that we can decide whether they belong to the same or different set.
In Union-Find:
Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
Union: Join two subsets into a single subset.

5. Experimental Results

Environment:

Processor Intel(R) Core(TM) i5-4200U CPU @ 1.60 GHz 2.30 GHz
Installed memory (RAM) 4.00 GB (3.89GB usable)
System type: 64-bit Operating System, x64-bit based processor
Windows 10
The instances represent the adjacency matrix of the graphs
In which edge weight 0 represent it is traversing to itself and 1000 represent there is no path between the vertices and the remaining values represent paths between the vertices

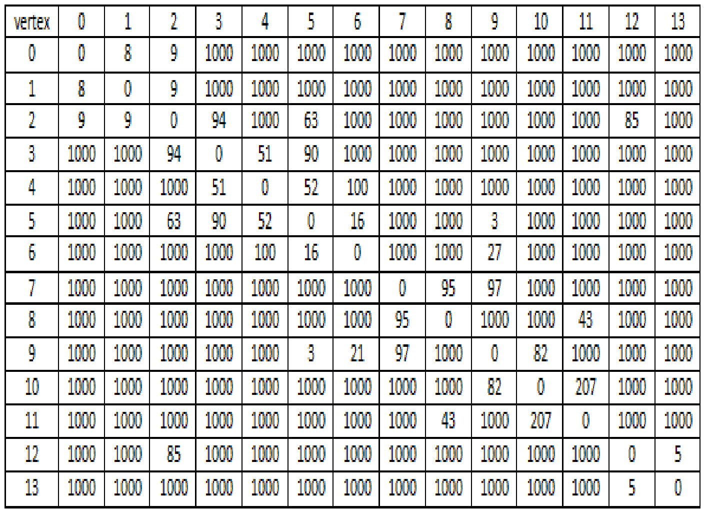
Table 6: Graph Instance (I1) with 9 vertices

vertex	0	1	2	3	4	5	6	7	8
0	0	5	1000	2	4	1000	1000	1000	1000
1	5	0	1000	8	1000	1000	1000	18	1000
2	1000	1000	0	3	1000	1000	1000	1000	7
3	2	8	3	0	1000	1000	6	1000	1000
4	4	1000	1000	1000	0	1000	10	1000	11
5	1000	1000	1000	1000	1000	0	3	15	1000
6	1000	1000	1000	6	10	3	0	1000	1000
7	1000	3	1000	1000	1000	15	1000	0	9
8	1000	1000	7	1000	11	1000	1000	9	0

Table 7: Graph Instance (I2) with 8 vertices

vertex	0	1	2	3	4	5	6	7
0	0	8	1000	1000	1000	1000	1000	4
1	8	0	4	1000	10	7	1000	9
2	1000	4	0	3	1000	3	1000	1000
3	1000	1000	3	0	25	18	2	1000
4	1000	10	1000	25	0	2	7	1000
5	10	7	3	18	2	0	1000	1000
6	1000	1000	1000	2	7	1000	0	3
7	4	9	1000	1000	1000	1000	3	0

Table : Graph Instance (I3) with 14 vertices



vertex	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	8	9	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
1	8	0	9	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
2	9	9	0	94	1000	63	1000	1000	1000	1000	1000	1000	85	1000
3	1000	1000	94	0	51	90	1000	1000	1000	1000	1000	1000	1000	1000
4	1000	1000	1000	51	0	52	100	1000	1000	1000	1000	1000	1000	1000
5	1000	1000	63	90	52	0	16	1000	1000	3	1000	1000	1000	1000
6	1000	1000	1000	1000	100	16	0	1000	1000	27	1000	1000	1000	1000
7	1000	1000	1000	1000	1000	1000	1000	0	95	97	1000	1000	1000	1000
8	1000	1000	1000	1000	1000	1000	1000	95	0	1000	1000	43	1000	1000
9	1000	1000	1000	1000	1000	3	21	97	1000	0	82	1000	1000	1000
10	1000	1000	1000	1000	1000	1000	1000	1000	1000	82	0	207	1000	1000
11	1000	1000	1000	1000	1000	1000	1000	1000	43	1000	207	0	1000	1000
12	1000	1000	85	1000	1000	1000	1000	1000	1000	1000	1000	1000	0	5
13	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	5	0

Time comparison for each instance

Instance	Prims (in ns)	Kruskal's (in ns)	MSTH (in ns)
I1	2089062	4033138	2793251
I2	1436336	3858923	2520475
I3	3432891	5536545	3567747

6. Conclusion

After studying the algorithm and finding the results from the algorithm we concluded that the algorithm proposed by us is faster than the kruskal's algorithm but slower than prim's algorithm under some constraints. There will be a great use of this algorithm in future as it can be used effectively to find the shortest path between any two places and also to find the most cost efficient path of all. Thus if studied further and done some research a lot of scope is there in this particular domain.

7. References

[1]https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

[2] https://en.wikipedia.org/wiki/Prim%27s_algorithm

[3] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[4] *Kruskal, J. B. ,"On the shortest spanning subtree of a graph and the traveling salesman problem". Proceedings of the American Mathematical Society. Vol.* ***7****,pp 48–50 ,1956.*

Maumita Chakraborty received her B.Tech. degree in Information Technology from University of Kalyani, India, and her M.E. degree in Software Engineering from Jadavpur University, India. She is currently pursuing her Ph.D. degree at the Department of Computer Science and Engineering, University of Calcutta, India. She is currently working as an Assistant Professor of the Department of Information Technology, Institute of Engineering and Management, Kolkata, India. Her major research interests include Graph theory and its applications, Data structure and algorithms, Networking, and Mobile computing

Rahul Singh is currently pursuing his B.Tech degree at the Department of Information Technology, Institute of Engineering &Management, Kolkata, India.

Ruchi Mehta is currently pursuing her B.Tech degree at the Department of Information Technology, Institute of Engineering &Management, Kolkata, India.