Tower Network Optimization with Interference and Cost Constraints

Introduction

As technology advances and revolutionises how we work, it also becomes extremely important to ensure that the services are interconnected and do not lose connection. Tower networks are an essential pillar in guaranteeing connectivity and communication between communities and societies in different geographical landscapes. They are critical infrastructure consisting of a network of cell towers, base stations, and associated infrastructure related to transmit signals to and from mobile devices.

Despite that, the efficient operations of the network are challenging due to several factors:

- *Interference*: Interference occurs when signals between neighbouring towers overlap each other, this can cause severe disruption in communications leading to poor signal quality or even network connection loss. Managing network interference is very important to ensure performance of the network and the service quality.
- *Cost*: There is significant cost associated with deploying, operating and maintaining towers. These costs include tower construction, equipment installation, spectrum licensing fees, energy consumption and ongoing tower maintenance.

This problem has a direct bearing on the quality of wireless services that consumers and businesses can access. A tower network that is not well optimised can cause calls to be cut off, data speeds to be slow, and connectivity to be unreliable. In addition, network operation that is not efficient can result in avoidable costs, which can have a negative impact on the profitability of network operators and, in some cases, increase the costs for end-users.

The assignment uses Minimal Spanning Tree (MST) algorithms to optimise the tower networks. It uses the algorithms to reduce towers that have high interference while also making sure that there are connections between other adjacent towers so that the network does not have any connectionless towers. It also checks the cost of the towers as a second measure, if the interference values turn out the same; to further optimise the network to make sure we are not having any redundant extra towers. The assignment is also using a modified version of the graph which made it undirected, this was done to replicate it to a real world scenario as one tower can reach another one and vice versa moreover, the focus of the assignment is on connection of the towers as well which makes it necessary to make the graph undirected. The two algorithms that the assignment uses are Kruskal and Prim’s algorithm to optimise the tower network. For this assignment the towers represent the vertice and the edges represent the connection between each tower. To run the two different algorithms, comment out one of the function calls in the main, and comment the other one. Besides to use sparse graph data modify the “input.txt” by removing its content with the values stored in the “input_sparse.txt” file.

Methodology

The assignment first used the graph data structure from the tutorial work as a base to build up a basic graph. It was later modified and adapted heavily to suit the problem statement. The graph from the tutorials was a directed graph and the edges coming from one of the vertices had a destination to go to, this was done using the “to_vertex” variable specified in the “Edge” struct. To transform it to an undirected graph another variable called “src” (source) was added to the struct, along with additional variables that hold the cost and weight (interference) of each tower to align with the problem. Moreover the Node structure has also been changed, in the undirected version we have a dest variable which is storing the destination vertex or node to which there is an edge from the current node along with the cost and interference. Besides when we are adding an edge using the “add_edge” function we are adding edges in both the directions, both source and destination. These changes make the graph undirected and have the edges pointing from both source and destination and contrarily. The assignment also uses Kruskal and Prim’s algorithms to create an optimised network. At first a generic version of both the algorithms were created by following the pseudocode provided in the lecture content and by referencing other pseudocode found online. The pictures of the pseudocode for the generic version of the algorithm are provided below.

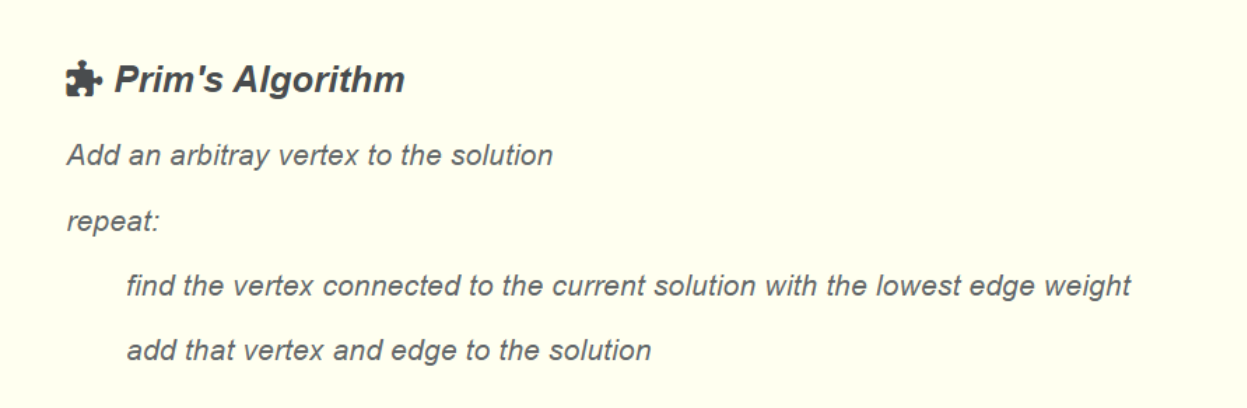


Figure 1: Picture of the Prim’s Algorithm pseudocode provided in the lecture content of KIT205 (https://mylo.utas.edu.au/d2l/le/content/598992/viewContent/5103660/View)

Pseudocode for Prim's algorithm

```
Prim(G, w, s)
//Input: undirected connected weighted graph G = (V,E) in adj list representation,
         source vertex s in V
//Output: p[1..|V|], representing the set of edges composing an MST of G
01 for each v in V
02    color(v) <- WHITE
03    key(v) <- infinity
04    p(v) <- NIL
05 Q <- empty list   // Q keyed by key[v]
06 color(s) <- GRAY
07 Insert(Q, s)
08 key(s) <- 0
09 while Q != empty
10    u <- Extract-Min(Q)
11    for v in Adj[u]
12       if color(v) = WHITE
13          then color(v) <- GRAY
14               Insert(Q,v)
15               key(v) <- w(u,v)
16               p(v) <- u
17       elseif color(v) = GRAY
18          then if key(v) > w(u,v)
19                  then key(v) <- w(u,v)
20                       p(v) <- u
21    color(v) <- BLACK
22 return(p)
```

Figure 2: A detailed and expanded pseudocode of Prim's algorithm.
(prim.pdf (uchicago.edu))

Kruskal's Algorithm

Add each vertex to it's own set

Sort all edges by weight

repeat:

remove the next edge from the sorted list

if the 'from' vertex is in a different set to the 'to' vertex (no cycle)

add the edge to the solution and merge the two sets

Figure 3: Picture of the Kruskal's Algorithm pseudocode provided in the lecture content of KIT205

(*https://mylo.utas.edu.au/d2l/le/content/598992/viewContent/5103660/View*)

```
KRUSKAL(V, E, w)

A ← { }           ▷ Set A will ultimately contains the edges of the MST
for each vertex v in V
    do MAKE-SET(v)
sort E into non decreasing order by weight w
for each (u, v) taken from the sorted list
    do if FIND-SET(u) = FIND-SET(v)
        then A ← A U {(u, v)}
            UNION(u, v)
return A
```

Figure 4: An expanded pseudocode of the Kruskal algorithm
(*https://www.mycplus.com/source-code/c-source-code/kruskals-algorithm-implementation-c-programming/*)

The Kruskal algorithm has been modified to fit the problem statement. Usually the Kruskal algorithm has one edge value but since we are working with two constraints, in the modified version of the algorithm the edge has two values. Furthermore in the beginning of the algorithm we are sorting the interference values in ascending order, this is done to ensure the algorithm prioritises edges with lowest interference since part of the problem is to decrease interference. Moreover there is an interference and cost constraint identified as “max_interference” and “max_cost”, the constraint is there to make sure that there are no edges that have values above the maximum limit in the network. The assignment is also using a modified version of Prim’s algorithm. Similar to the modification we did on Kruskal, Prims is also using an edge that has two values: the cost and interference. Additionally it is also making sure that edges with values lower or the same as the maximum limits are being added to the MST for ensuring minimal network interference and cost. This is done by having an array of edges called “result” which keeps a track on the minimum weighted edges by having the maximum values of interference and cost. Another difference between the standard prim is edge selection. The algorithm is selecting an edge during each iteration that minimises both the interference and cost while still maintaining connectivity at the same time. The algorithm terminates after constructing an MST with the maximum possible coverage.

```
procedure kruskal_MST(graph, max_interference, max_cost):
    Initialize variables
    Initialize subsets for each vertex

    Sort edges by weight (interference)

    Initialize empty result

    for each edge in sorted_edges:
        if edge.weight <= max_interference:
            if edge does not form a cycle in result:
                Add edge to result

    Print total interference before MST
    Print minimum spanning tree edges and costs

    Free memory
end procedure
```

Figure 5: Pseudocode for the modified Kruskal algorithm.

```
procedure prim_MST(graph, maxInterference, cost_limit):
    Initialize variables and memory for result and subsets

    for each vertex in result:
        Initialize it with infinity values

    while e < V - 1:
        Initialize min_weight and min_cost with infinity

        for each edge (u, v) in the graph:
            if edge meets constraints and is smaller than min_weight:
                Update min_weight, min_cost, and result[e] with the edge

        if an edge is found:
            Perform union on subsets
            Increment e
        else:
            Break the loop

    Print minimum spanning tree edges and costs

    Free memory
end procedure
```

Figure 6: Pseudocode for the modified Prim's algorithm.

These two algorithms both produce a MST, but the way they do it is different, Prim's technique uses a greedy algorithm to find the smallest MST in the graph, while Kruskal locates the lowest one. Prim's algorithm does this by having two sets of vertices, one that is already included in the MST and one that is not; it then selects the edge with the least weight between all the edges that are connected in the two sets. Since Prim's algorithm travels one node more than one time it has a time complexity of $O(V^2)$ whereas Kruskal only visits one node one time and has a time complexity of O(E log V). Furthermore in theory these two algorithms have different speeds on different types of graphs. Prim's algorithm is faster on dense graphs while Kruskal is faster on sparse graphs. So testing was done on the speed of the algorithms on two types of graphs, to validate whether this is the case.

Below is the visual output made by running python scripts on a small sample from the test dataset.

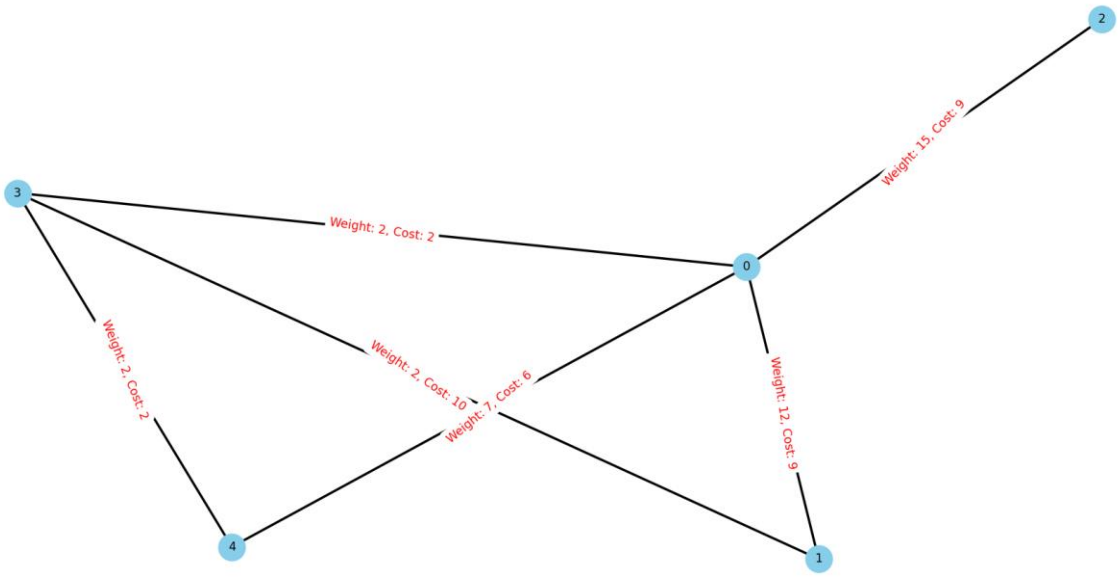


Figure 7: A visual representation of the graph from the sample test dataset.

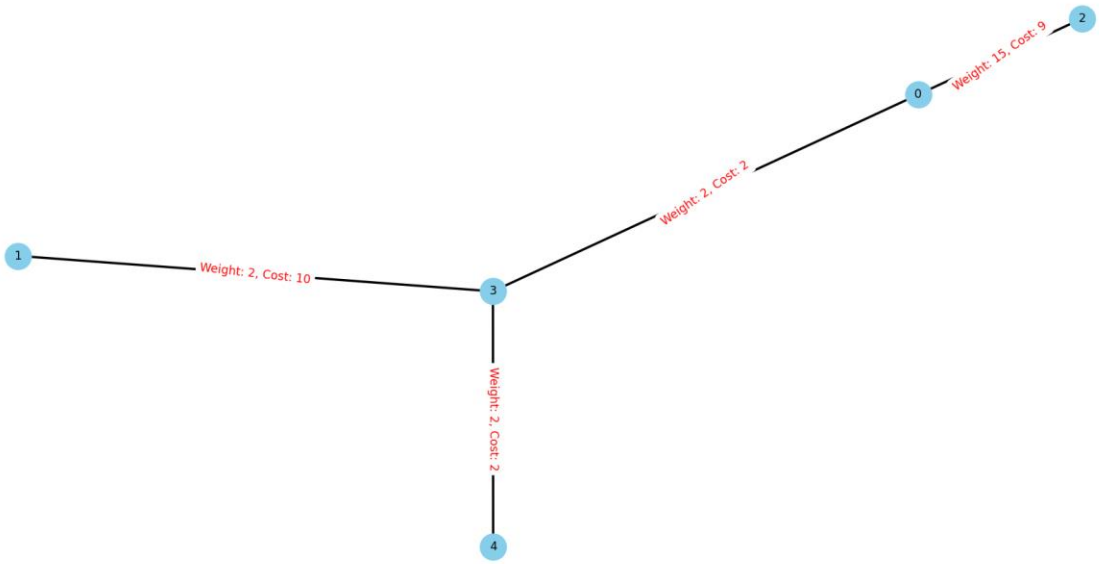


Figure 8: The visual representation of the graph after running Kruskal's algorithm.

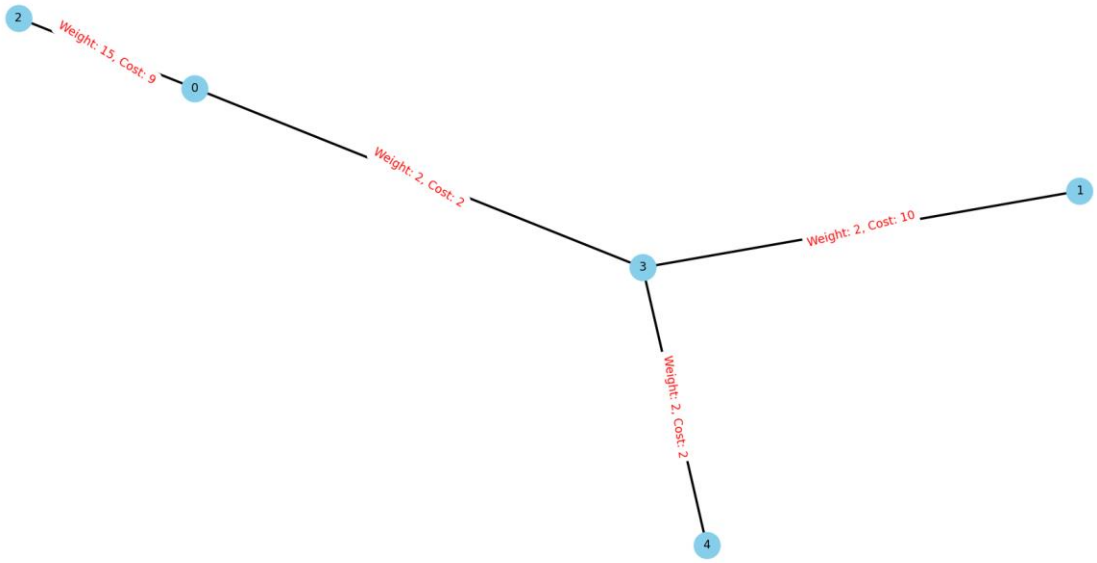


Figure 9: The visual representation of the graph after running Prim's algorithm.

Results and discussion

To test the program, two different datasets were generated. One dataset contains data for a dense graph and the other contains data for a sparse graph. This was done to test whether in practice the graph algorithms behave as they should in theory. The dataset for testing the program is being generated by running a python script that is generating data in a text file called "input.txt" and the data is formatted in a way for the program to read it directly and perform the necessary operations. Firstly we test the two different algorithms on a sparse graph and the results of the testing reflect what was discussed in the theory that Kruskal is indeed faster than Prim when used on a sparse graph. The results of the testing are shown below.

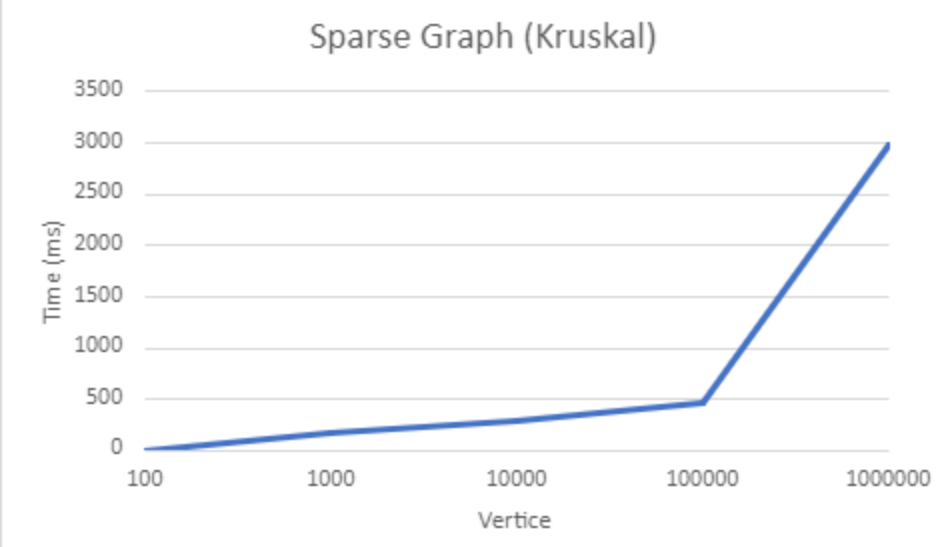
Sparse Graph (Kruskal)

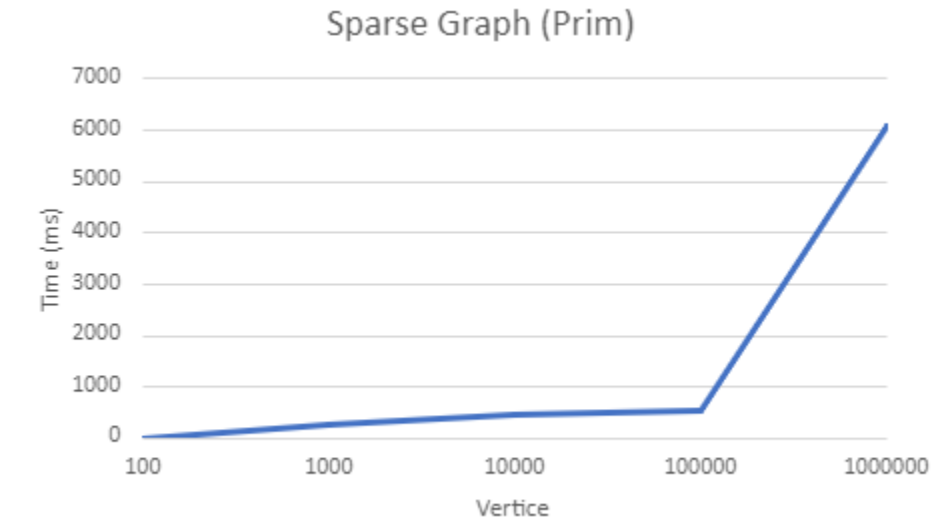
Vertice	Edge	Time(ms)
100	10	3
1000	1000	173
10000	1000	298
100000	1000	468
1000000	10000	2977

Sparse Graph (Prims)

Vertice	Edge	Time(ms)
100	10	7
1000	1000	268
10000	1000	467
100000	1000	524
1000000	10000	6074

Visual representation of the above testing:





In the visualisation we can see that the graph for Prim is steeper as the number of vertices increase compared to the one for Kruskal, which signifies that more time is required to process the vertices and generate a MST for the graph input.

Next we test the algorithms on a dataset that consists of dense graphs, this is done to prove that Prim's algorithm is faster on a denser graph. The result of the testing is shown below:

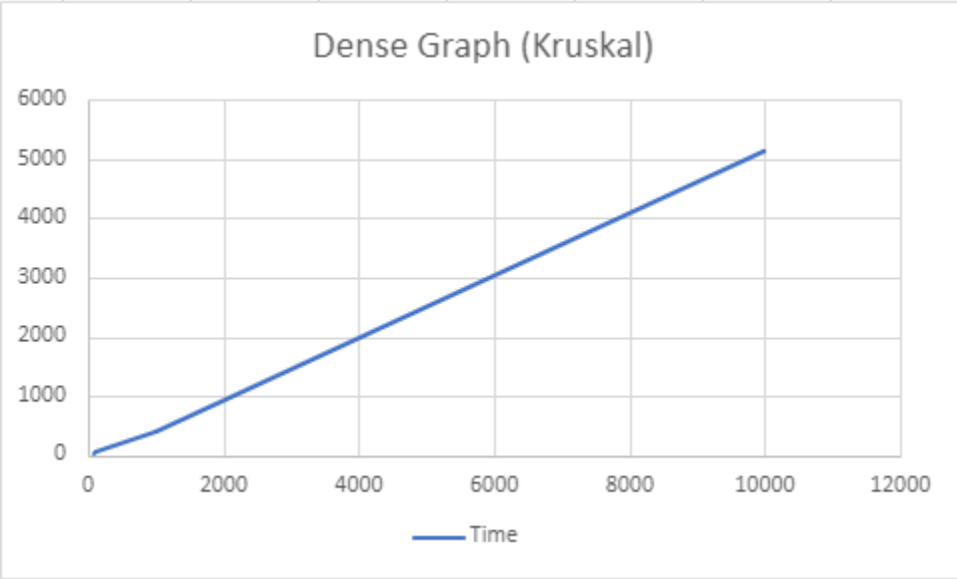
Dense Graph (Kruskal)

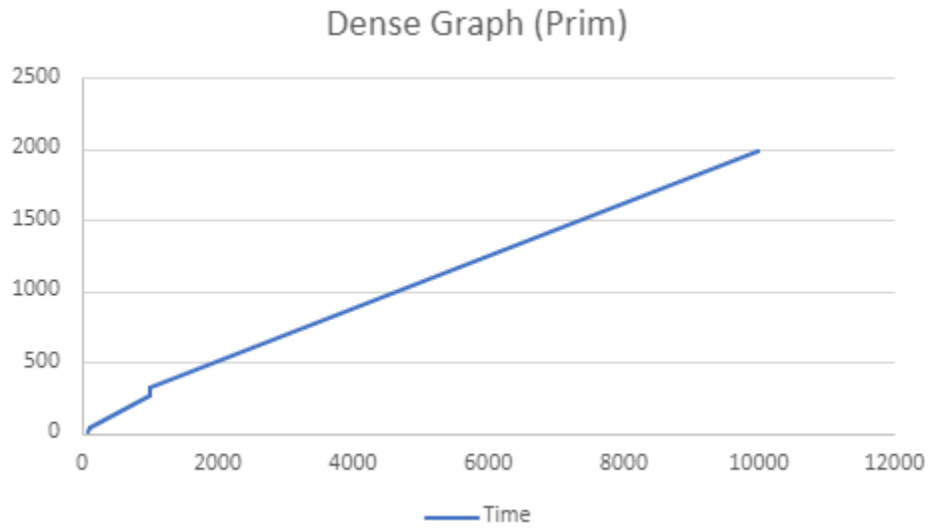
Vertice	Edge	Time(ms)
15	75	31
20	100	41
50	1000	384
500	1000	405
50	10000	5150

Dense Graph (Prim)

Vertice	Edge	Time(ms)
15	75	10
20	100	37
50	1000	263
500	1000	325
50	10000	1986

By seeing the data we can identify that there is a stark difference between the two algorithms that is especially visible when the number of data in the dataset increases, and we can see both the algorithms in action. Below is a visual representation of the edges against time:





For Kruskal we can see that the gradient of the curve is far steeper compared to Prim's and this proves and signifies that Kruskal algorithm is slower when used on a dense graph. Furthermore the cost and interference constraints also play a vital factor in the run time of the algorithms. This is because the algorithms are trying to find the best minimised tree, i.e optimised connections and also makes sure that the connections are within the specified cost and interference constraints, this takes in extra processing time for both the algorithms in the two different types of graph dataset (Sparse and Dense).

Conclusion

In conclusion, the assignment goes through two different approaches to solve the tower network optimisation problem. It also tests through two different scenarios where there are more towers and less connection and vice versa. Moreover it does some testing and proves the theoretical concept as to which algorithm to use in the two different cases. The code can further be refined to process real world data instead of using a data generator to be used as a datasource. Besides for future implementation Boruvka's algorithm can be used, which provides a balanced runtime between a dense and sparse graph. Since optimising tower networks is very vital in our everyday life, it will be quite fascinating to see how these algorithms can be used to solve the problems facing tower installations.

Github link:
Eusha425/Assignment-2 (github.com)

Reference

Saqib, M. (2023) *Kruskal's algorithm implementation in C - MYCPLUS - C and C++ programming resources*, *MYCPLUS*. Available at: *https://www.mycplus.com/source-code/c-source-code/kruskals-algorithm-implementation-c-programming/* (Accessed: 20 October 2023).

Brady, G. (2009) *Pseudocode for Prim's algorithm - Department of Computer Science*, *prim.pdf*. Available at: *https://people.cs.uchicago.edu/~brady/CSPP55001/notes/prim.pdf* (Accessed: 20 October 2023).

MyLo content: *https://mylo.utas.edu.au/d2l/le/content/598992/viewContent/5103660/View*

ChatGPT was used as assistance for solving the problem and finishing the assignment, specifically modifying the base algorithms into specialised versions for this particular problem also was used to write the modified version pseudocode.