**Detail design – PS2 (Telecom Network - MST)**

**group-258**

1. Purpose and Scope

This document describes the Low-level design for the Telecom Network Minimum Spanning Tree system. It aims to provide the detailed technical descriptions. The scope of the solution is bound by the requirements given in the assignment i.e. Assignment 2\_PS02\_Telecom Network.pdf

1. Solution Overview

The Telecom Network system is maintaining the records of the Vertices/Nodes, Edges and the cost for each edge, that is being used to build a graph for the telecom offices to be connected. The purpose of this system is to find the minimum cost spanning Tree (MST) to connect all the Nodes of the network i.e. to find the lowest possible implementation cost.

1. General Assumptions
2. The program only accepts the inputs in the specified format as given in the requirement.
3. Exception handling is done for the data structure used in only inserting and deleting a new record.
4. Input and output filenames are hardcoded as per the requirement given. No other input files other than mentioned in the requirement will be considered.
5. The input file is mandatory for the program to execute.
6. Data Structure Model – Graph (Disjoint-set) - ADT

The abstract data type graph is being used to implement the data structure/storage. It contains the set of vertices and edges with its cost. The insert and update basic operations are implemented explicitly instead of using any library functions in Python.

The graph is allocating the data dynamically and performance wise its better than linked list as it’s not linear. Unlike linear data structure which have only one logical way to traverse them, It can be traversed in different ways. The algorithm is written in such a way that traversing in ascending order of the cost is simpler. This is required to implement the Kruskal’s MST (Minimum Spanning Tree) algorithm so that minimum costing can be done by finding the right set of edges.

It works on the principal of disjoin-set data structure that stores a collection of disjoin (non-overlapping) sets (a partition of set into disjoint subsets). It provides operation of adding new sets, merging sets (by union) and finding a representative member of a set. It provides a mechanism where edges are being selected with a greedy approach of minimum cost to connect ad long as the circle is not formed.Example (Graph – Spanning Tree):

F

A

D

G

E

C

B

1. Graph – Functions:
   1. **vertex\_from\_name** – Return the object of a Vertex by Name

In the list of vertices, return the vertex object by comparing the name.

**Worst case time complexity = O(V)**

**Best case time Complexity = O(1)**

* 1. **add\_edge**- add the edges to list of edges and vertices to the set of vertices

With a condition check whether the start and end vertex is already added or not, it keeps on appending the list of vertices and edge.

**Worst case time complexity = O(V)**

**Best case time Complexity = O(1)**

* 1. **union** - Given a list of disjoint sets of vertices, merges v1 root set with v2 root set and returns merged sets

For a given set of vertices, check if the start and end vertex given belong to the same set (having the same parent node) or not. If belong to the same set, merge them to the exiting set and return false not to append for MST. If not, keep them appended.

**Worst case time complexity = O(logV)**

**Average case time Complexity = O(logV)**

* 1. **MST** – Kruskal’s minimum spanning tree algorithm. A very handy algorithm for a sparse graph. Starts with sorting all the edges as per the weight of a graph. The solution evolves with the First edge is being selected and added for a MST. Keeps on adding the minimum cost edges and checking if a cycle is being created. If cycle is created, skip that edge and go on till the number of edges in the MST is equal to (number (V) – 1). That will be the terminating condition. The data structure required is a simple graph on which an MST in being build.

In this API:

The edge is sort based on weight – **O (E log E)**

Place each vertex into its own set – **O (V)**

In worst case we iterate through all the edges (E) with a union run for each edge

**O (E log V)**

**Worst case time complexity = O (E log E) + O(V) + O (E log V) = E log V** (E >=V-1 for connected graphs)

1. Alternate way of modelling the problem

For a graph with **V** vertices **E** edges, Kruskal's algorithm runs in **O(E log V)** time and Prim's algorithm can run in **O(E + V log V)** amortized time. if you use a [Fibonacci Heap](http://en.wikipedia.org/wiki/Fibonacci_heap).

Prim's algorithm is significantly faster in the limit when you've got a really dense graph with many more edges than vertices. **Kruskal performs better in a typical situations (sparse graphs) because it uses simpler data structures. Thus, we have used Kruskal’s greedy algorithm because of its simplistic approach.**