CS2500 Project 4
Phase 1

Evan Wilcox

Due April 23, 2019

1 Motivation

Imagine you are the owner of a internet service provider. Your job is to provide your customers with internet by connecting their houses back to your data center. To be the most cost effective, you would want to run as little cable as possible while still connecting all locations. If you assume that all cables will be laid in straight lines from point to point, this begins to look like an undirected graph. Using the distances between points, a minimum spanning tree can be used to connect all points together while using the least amount of cable.

This report will investigate two minimum spanning tree algorithms: Prims's and Kruskal's. The run time complexity of both algorithm's will be measured, analysed and compared for the given input of city's longitudes and latitudes.

2 Background

A minimum spanning tree is a subset of a connected weighted graph that connects all the vertices in the graph together without any cycles and with the lowest possible total edge weight. Prim's algorithm builds a minimum spanning tree by first starting at an arbitrary vertex then adding the vertex with the cheapest connection from the MST to another vertex until all vertexes from the graph are in the MST. Kruskal's algorithm works slightly different. It first starts by sorting all the edges in the graph in increasing order. It then adds the edge with the lowest weight until all vertexes from the graph are in the MST.

There are two popular ways of storing a graph: an adjacency matrix and an adjacency list. An adjacency matrix uses a 2d array to store the weights of the edges of the graph. The matrix would be initialized to something to indicate that there is not an edge at that specific index like -1. An adjacency list is a collection of list that represent a graph. Each list describe each node that has a connection to that node. Adjacency matrixes are best for dense graphs because they take up a consistent amount of space, $|V|^2$, and will store less non-edges. While adjacency lists are better for sparse graphs because they only store edges where they exist. Adjacency lists were used to store the graphs in this report because there were few edges compared to the amount of vertices leading to a sparse graph.

3 Procedures

1. Develop a precondition, postcondition, and loop invariants for Prim's algorithm and Kruskal's algorithm.
2. Show that the previous preconditions, postconditions, and loop invariants are correct.
3. Express Prim's and Kruskal's using pseudocode.
4. Implement Prim's and Kruskal's in c++.
5. Implement preconditions, postconditions, and invariants using c++ assert statements to validate correctness.
6. Measure run times of the two algorithms to experimentally determine their run time complexity and compare to their expected run time complexity.
7. List problems encountered during development.
8. Produce a conclusion addressing the efficacy of the methods used.

4 Pseudocode

Prim's Algorithm Pseudocode
priority $Q \leftarrow V$
initialize all $v \in V$ with dist$(v) \leftarrow \infty$
dist$[s] \leftarrow 0$
while Q is not empty
    $u \leftarrow$ *extract-min* $\leftarrow Q$
    for each neighbor v of u
        if length$(u, v) <$ dist$[v]$
            dist$[v] \leftarrow$ length(u, v)

Kruskal's Algorithm Pseudocode
T $\leftarrow$ Empty Forest of Trees
Sort edges, E, by weight (nondecreasing) into $e_1, e_2, ..., e_n$.
for each edge $e_i, i = 1, ..., n$
    Add e_i to the T if it is safe

5 Problems Encountered

The given input file was difficult to work with so a python script was used to extract the necessary data and put it in a .csv file which was easier to work with in c++. The method I used for finding the distance between two longitude and latitudes is not entirely accurate, I treated them as x and y coordinates and used the pythagorean theorem but after comparing the final MST to an actual map, it seems that this method achieved a close enough result. The distance I used for the largest distance between two cities for them to have a direct connection was 50 miles because it was the smallest distance where every city would have at least one connection.

6 Performance Results

The average time complexity of kruskal's algorithm is $|e|\lg|e|$, where e is the number of edges in the graph, because kruskal's first sorts the edges based on their weight. While prim's algorithm runs in $|e|\lg|v|$. Starting prim's at different vertices created the same MST each time. Both algorithms created the same MST.

7 Conclusion

After implementing both algorithms in c++ and analysing them we can conclude that because of their time complexities, kruskal's algorithm is better when the graph is sparse and prim's algorithm is better when the graph is dense.

Appendix A - Source Code

```
/*
* Author: Evan Wilcox
* File: main.cpp Date: 4/23/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: File used to create a minimum spanning tree of cities in Missouri
*        from a file of cites names, latitudes, and longitudes.
*/

#include <fstream>
#include "Graph.h"

using namespace std;


int main()
{
    ifstream fin;
    fin.open("data.csv");

    //ofstream fout;
    //fout.open("out.txt");

    vector<Node> V;

    string name;
    float lat;
    float lon;

    // Input cites names and coordinates and put them in a vector V of Nodes.
    string line;
    while(getline(fin, line))
    {
        name = line.substr(0, line.find(','));
        lat = stof(line.substr(line.find(',')+1, 10));
        lon = stof(line.substr(line.find_last_of(',')+1, 10));

        V.emplace_back(name, lat, lon);
    };

    Graph G(&V);

    float dis;

    for(int x = 0; x < V.size(); x++)
    {
        for(int y = 0; y < V.size(); y++)
        {
```

```
            dis = sqrt(pow((V[x].m_lat-V[y].m_lat), 2) + pow((V[x].m_long-V[y].m_long), 2)

            if(dis < 50)
            {
                G.addEdge(x, y, dis);
            }
        }
    }

    G.print();
    cout << "=============Prim's algorithm: =================" << endl;
    G.PrimMST(0).print();
    cout << "=============Kruskal's algorithm: =================" << endl;
    G.KruskalMST().print();

    fin.close();

    return 0;
}


/*
* Author: Evan Wilcox
* File: Graph.h Date: 4/23/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: Header file for a Graph class
*/

#ifndef GRAPH_H
#define GRAPH_H

#include <list>
#include <queue>
#include <bits/stdc++.h>
#include "Node.h"
#include "DisjointSets.h"
#define INF 0x3f3f3f3f
using namespace std;

/*
* Class: Graph
* Brief: A class that implements a graph and multiple minimum spamming tree algorithms.
*/
class Graph
{
    private:
        vector<Node> *V;                                // pointer to a vector of verticie
        list<pair<int, float>> *adj;                    // adjency list
        vector< pair<float, pair<int, int>>> edges;     // vector of edges
```

```
public:
    /*
    * Function  : Graph
    * Brief     : Constructor
    * Pre       : none
    * Post      : A Graph object is created, adj = a new adjency list
    * Param V   : a pointer to a vector of Nodes
    * Return    : none
    */
    Graph(vector<Node> *V);

    /*
    * Function  : ~Graph
    * Brief     : Deconstructor
    * Pre       : none
    * Post      : A Graph object is deleted
    * Return    : none
    */
    ~Graph();

    /*
    * Function  : addEdge
    * Brief     : adds an edge to the edges vector and adds a connection in adj
    * Pre       : none
    * Post      : A edge is add to edges and a connection is add to adj
    * Param u   : index of first node of edge
    * Param v   : index of second node of edge
    * Param w   : weight of edge
    * Return    : none
    */
    void addEdge(int u, int v, float w);

    /*
    * Function  : PrimMST
    * Brief     : Creates a minimum spanning tree using Prim's algorithm
    * Pre       : V is a vector of Nodes, adj is an adjency list
    * Post      : a MST is printed
    * Param src : source node to start the MST at
    * Return    : MST in a Graph class
    */
    Graph PrimMST(int src);

    /*
    * Function  : KruskalMST
    * Brief     : Creates a minimum spanning tree using Kruskal's algorithm
    * Pre       : V is a vector of Nodes, edges is a vector of edges
    * Post      : a MST is printed
    * Return    : MST in a Graph class
    */
```

```
        Graph KruskalMST();

        /*
        * Function  : print
        * Brief     : prints adj
        * Pre       : V is a vector of Nodes, adj is an adjencency list
        * Post      : an adjencency list is printed to the screen
        * Return    : none
        */
        void print();
};

#endif


/*
* Author: Evan Wilcox
* File: Graph.cpp Date: 4/23/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: Implementation file for a Graph class
*/

#include "Graph.h"

Graph::Graph(vector<Node> *V)
{
    this->V = V;
    adj = new list<pair<int, float>> [V->size()];
}


Graph::~Graph()
{
    delete adj;
}


void Graph::addEdge(int u, int v, float w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));

    edges.push_back({w, {u, v}});
}


Graph Graph::PrimMST(int src)
{
    // Priority queue to store vertices
```

```
priority_queue<pair<int, float>, vector<pair<int, float>> , greater<pair<int, float>>>

// Vector to store distances
vector<float> dis(V->size(), INF);

// To store parent array which stores MST
vector<int> parent(V->size(), -1);

// Keeps track if an edge is in MST
vector<bool> inMST(V->size(), false);

// Insert src in priority queue and set it to 0.
Q.push(make_pair(0, src));
dis[src] = 0;

while (!Q.empty())
{
    int u = Q.top().second;
    Q.pop();

    inMST[u] = true;  // Include vertex in MST

    // i is used to get all adjacent vertices of a vertex
    list< pair<int, float> >::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        // Get vertex label and weight of current adjacent of u.
        int v = (*i).first;
        float weight = (*i).second;

        //  If v is not in MST and weight of (u,v) is smaller than current dis of v
        if (inMST[v] == false && dis[v] > weight)
        {
            // Updating dis of v
            dis[v] = weight;
            Q.push(make_pair(dis[v], v));
            parent[v] = u;
        }
    }
}

Graph MST(V);

for(int i = 1; i < V->size(); i++)
{
    MST.addEdge(i, parent[i], dis[i]);
    cout << dis[i] << endl;
}

return MST;
```

```
}


Graph Graph::KruskalMST()
{
    Graph MST(V);

    // Sort edges in increasing order based on distance
    sort(edges.begin(), edges.end());

    // Create disjoint sets
    DisjointSets ds(V->size());

    // Iterate through all sorted edges
    vector< pair<float, pair<int, int>> >::iterator it;
    for (it = edges.begin(); it != edges.end(); it++)
    {
        int u = it->second.first;
        int v = it->second.second;

        int set_u = ds.find(u);
        int set_v = ds.find(v);

        // Check if the selected edge is creating a cycle or not
        if (set_u != set_v)
        {
            MST.addEdge(u, v, it->first);

            // Merge two sets
            ds.merge(set_u, set_v);
        }
    }

    return MST;
}


void Graph::print()
{
    ofstream fout;
    fout.open("out.txt");

    fout << "2" << endl;

    for(int x = 0; x < V->size(); x++)
    {
        fout << (*V)[x].m_name << ": ";
        list< pair<int, float> >::iterator y;
        for (y = adj[x].begin(); y != adj[x].end(); ++y)
        {
```

```
            fout << (*V)[y->first].m_name << ", ";
        }
        fout << endl << endl;
    }

    fout.close();
}



/*
* Author: Evan Wilcox
* File: DisjointSets.h Date: 4/23/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: Header file for a Disjoint Set class
*/

#ifndef DISJOINT_H
#define DISJOINT_H

/*
* Class: DisjointSets
* Brief: A class that implements disjoint sets as a data type
*/
struct DisjointSets
{
    int *parent;    // array that stores the parent of each node
    int *rnk;       // array that stores the rank of each node
    int n;          // number of nodes in set

    /*
    * Function  : DisjointSets
    * Brief     : Constructor
    * Pre       : none
    * Post      : A DisjointSets object is created
    * Param n   : the number of nodes in the set
    * Return    : none
    */
    DisjointSets(int n)
    {
        // Allocate memory
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];

        // Initially, all vertices are in different sets and have rank 0.
        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;
```

```
            parent[i] = i;
        }
    }

    /*
    * Function  : find
    * Brief     : Find the parent of a node
    * Pre       : none
    * Post      : none
    * Param u   : node whose parent will be found
    * Return    : parent of node u
    */
    int find(int u)
    {
        /* Make the parent of the nodes in the path
            from u--> parent[u] point to parent[u] */
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    /*
    * Function  : merge
    * Brief     : unions two sub trees
    * Pre       : none
    * Post      : two sub trees are merged
    * Param x   : root of tree 1 to merge
    * Param y   : root of tree 2 to merge
    * Return    : none
    */
    void merge(int x, int y)
    {
        x = find(x);
        y = find(y);

        /* Make tree with smaller height
            a subtree of the other tree  */
        if (rnk[x] > rnk[y])
            parent[y] = x;
        else // If rnk[x] <= rnk[y]
            parent[x] = y;

        if (rnk[x] == rnk[y])
            rnk[y]++;
    }
};

#endif
```

```
/*
* Author: Evan Wilcox
* File: Node.h Date: 4/23/19
* Class: CS 2500 section A
* Instructor : Bruce McMillan
* Brief: Header file for a Node class
*/

#ifndef NODE_H
#define NODE_H

#include <string>
#include <iostream>
using std::string;
using std::ostream;

/*
* Class: Node
* Brief: A class that pairs a name, latitude and longitude in to a node.
*/
struct Node
{
    string m_name;  // name of the node
    float m_lat;    // latitude of the node
    float m_long;   // longitude of the node

    /*
    * Function  : Node
    * Brief     : Constructor using a initialization list
    * Pre       : None
    * Post      : A Node object is created
    * Param n   : name to be given to m_name
    * Param lat : latitude to be given to m_lat
    * Param lon : longitude to be given to m_lon
    * Return    : None
    */
    Node(string n, float lat, float lon): m_name(n), m_lat(lat), m_long(lon){}
};

#endif
```

Appendix B - Full adjacency list

The full adjacency list for the cities in Missouri can be found here:
https://tinyurl.com/y63jcede