Aditya Chempakasseril  
Mayank Gupta

1. The dataset we are going to use is the California road network. We are going to use data from the University of Utah from the following link: “https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.” Points of interest on the network will represent nodes, and distances between nodes will represent edges, with weights being proportional to distances. The graph will be undirected. The dataset has 21048 nodes and 21693 edges.
2. Using our map, we sought to determine the most cost-effective deployment of water pipes to cover every node of the map using the least material possible. To do this, we read in the necessary data from the text file and produced an MST using Kruskal’s algorithm. We then used our graph-building algorithms, with a few modifications to deal with weighted edge data, from Part 1 to produce a graph of the MST in the form of a hashmap. With this, we could operate on the graph with any of the functions (e.g. pathfinder) that we previously built.

To implement Kruskal’s algorithm we used a disjoint set to keep track of node relationships and a priority queue to keep track of edge relationships. Because C++ doesn’t include a disjoint set data structure in the STL, we had to implement our own. In this implementation, we treated nodes as integers.

On an abstract level, the disjoint set is treated as a tree with a sentinel node as its root. Each subset is defined by its sentinel. If there are *n* nodes, an array of size *n*+1 (to handle cases where the initial node is defined as 1 instead of 0) is used to represent the nodes. Each index *i* corresponds to node *i*. The element at index *i* stores the index, *i’*, of the node, *n’*, that *n* points to. Following the tree structure from a given *n*, eventually a sentinel node will be reached. Every node that is connected, directly or indirectly, to the sentinel node is labeled by that sentinel node for further use in union and find functions.

When the find function is called on the disjoint set, path compression is implemented to shorten the size of the tree of the node being found for future find and union operations. When the union function is called on nodes *a* and *b*, first the find function is called on these nodes. If the two nodes have different sentinel nodes, then their subsets are combined. Here, union by size is used. To keep track of the sizes of subsets, we use a separate vector of. This vector stores the size of each subset at the index corresponding to the sentinel node of that subset. Using this vector, in the union function the sentinel of the smaller subset can be attached to the sentinel of the larger subset, producing a single tree of a form that is both short and ideal for path compression (“short and stubby”).

Using a disjoint set to keep track of node connections, we could then apply Kruskal’s algorithm to create an MST of the map data. Edges (which had a class of their own) were ordered using the STL priority queue. They were then systematically inserted into an MST such that no edge connecting nodes already in the MST was inserted.

Finally, after the MST was built, we outputted the results in a text file containing rows of data in the form “node1 node2 edge\_weight.” This text file was then fed into the Graph::loadFromFile method. At this point, the total length (sum of all weights) of the edges in the MST was easily calculated. If desired, Graph::pathfinder could be used to find the path between two nodes.

1. The method used to solve our problem is described above. The input for our program is a list of edges of the form “edge\_id node1 node2 edge\_weight.” The program reads the input file line-by-line and pushes each edge into a priority queue. After using Kruskal’s algorithm and producing an MST (as described above), the output of the program is the MST represented as a list of edges of the form “node1 node2 edge\_weight.” Below is a test case that we used (the same case that was used in lecture):

0 1 2 4

1 1 3 8

2 2 3 9

3 2 4 8

4 3 4 2

5 2 5 10

6 3 6 1

7 4 5 7

8 4 6 9

9 5 6 5

10 5 7 6

11 6 7 2

The output was:

3 6 1

3 4 2

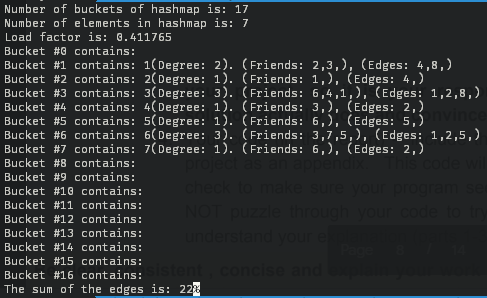
6 7 2

1 2 4

5 6 5

1 3 8

This is the minimum spanning tree of the original graph. After feeding this data into Graph::loadFromFile, Graph::printTree can be used to visualize the MST. The output is shown on the next page.



**Figure 1: Graph representation of MST\_1**

As can be seen, the graph is represented as a hashmap. The sum of the edges is easily calculated to be 22.

Included in this zip file is a larger test case with 50 edges. The original dataset with over 21000 edges is included as well. Below is the input and output of the 50 edge test case.

Input:

0 1 2 4

1 1 3 8

2 2 3 9

3 2 4 8

4 3 4 2

5 2 5 10

6 3 6 1

7 4 5 7

8 4 6 9

9 5 6 5

10 5 7 6

11 6 7 2

12 7 51 1

13 51 52 4

14 51 53 8

15 52 53 9

16 52 54 8

17 53 54 2

18 52 55 10

19 53 56 1

20 54 55 7

21 54 56 9

22 55 56 5

23 55 57 6

24 56 57 2

25 57 101 1

26 101 102 4

27 101 103 8

28 102 103 9

29 102 104 8

30 103 104 2

31 102 105 10

32 103 106 1

33 104 105 7

34 104 106 9

35 105 106 5

36 105 107 6

37 106 107 2

38 107 151 1

39 151 152 4

40 151 153 8

41 152 153 9

42 152 154 8

43 153 154 2

44 152 155 10

45 153 156 1

46 154 155 7

47 154 156 9

48 155 156 5

49 156 157 2

Output:

3 6 1

7 51 1

53 56 1

57 101 1

103 106 1

107 151 1

153 156 1

3 4 2

6 7 2

53 54 2

56 57 2

103 104 2

106 107 2

153 154 2

156 157 2

1 2 4

51 52 4

101 102 4

151 152 4

5 6 5

55 56 5

105 106 5

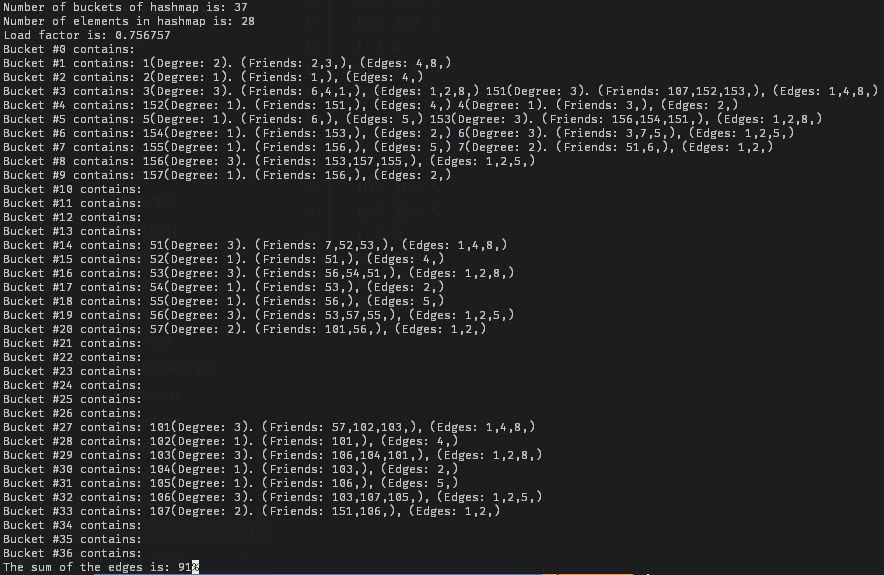
155 156 5

1 3 8

51 53 8

101 103 8

151 153 8



**Figure 2: Graph representation of 50-edge test case.**

The graph is verified to be an MST since the number of edges is equal to one less than the number of nodes (the original graph was *connected*). Kruskal’s algorithm works with disconnected graphs, but in the case of the problem we are solving, disconnected graphs make no physical sense. The reason we don’t include a test file of a smaller version of the roads dataset is that almost all 50-edge subsets of this set are themselves MSTs.

The solution to our original problem, that regarding the minimum length of piping to cover all nodes in the California road network, is 307.632. The units of the solution are somewhat difficult to interpret. The nodes of the graph represent some function of latitude and longitude, and edges represent the L2 norm between these nodes. There is no further explanation of what physical units this norm corresponds to in the source material. With some further research, a conversion can likely be found to shift from unitless L2 norms to physical distances.

**Appendix**

/\*Edge hpp\*/

#ifndef EDGE\_HPP

#define EDGE\_HPP

#include <vector>

using namespace std;

/\* Edge class containg the following fields

int id: edge id

float weight: edge weight

int node1: node at one side of the edge

int node2: node at other side of the edge

\*/

class Edge{

protected:

int id;

float weight;

int node1;

int node2;

public:

Edge(int id, int node1, int node2, float weight);

~Edge();

//Getters and setters

vector<int> getNode\_pair();

int getid();

float getWeight();

//Overloading less than operator for use in priority queue

bool operator<(const Edge& other);

};

#endif

/\*Edge cpp\*/

#include "Edge.hpp"

Edge::Edge(int id, int node1, int node2, float weight) : id(id), weight(weight), node1(node1), node2(node2){}

Edge::~Edge(){}

vector<int> Edge::getNode\_pair(){

return vector<int>{node1, node2};

}

int Edge::getid(){

return id;

}

float Edge::getWeight(){

return weight;

}

bool Edge::operator<(const Edge& other) {

if(this->weight != other.weight) return this->weight>other.weight;

return this->id>other.id;

}

/\*Disjoint set hpp\*/

#ifndef DISJOINTSET\_HPP

#define DISJOINTSET\_HPP

#include <vector>

using namespace std;

/\*\*

\* Disjoint set class containing the following fields:

\* vector<int> items: vector of nodes, where each node is represented by an index

\* vector<int> sizes: size of subset of sentinel nodes. ONLY indices corresponding

\* to sentinel nodes are guaranteed to contain correct subset sizes.

\*/

class DisjointSet{

protected:

vector<int> items, sizes;

public:

DisjointSet(int numItems);

~DisjointSet();

//Finds item in the disjoint set. Utilizes path compression.

//Returns index of sentinel node for subset that item is in.

int ds\_find(int item);

//Combines to subsets. Utilizes union by size.

//Returns true if union was successful.

bool ds\_union(int item1, int item2);

};

#endif

/\*Disjoint set cpp\*/

#include "DisjointSet.hpp"

#include <iostream>

#include <fstream>

#include <sstream>

using namespace std;

/\*\*Items and sizes are initialized to number of nodes plus one.

\* Each index is initially set to -1 (sentinel). This disjoint set is NOT applicable

\* for general use. It is assumed that all of indices of the vector

\* correspond to and actual node in the dataset. Even if the node isn't

\* in the dataset, this won't matter with respect to building the mst since

\* union will never be called on nodes that don't exist in the dataset.

\* For general use, this class can be somewhat easily modified.

\*/

DisjointSet::DisjointSet(int numItems){

items = vector<int>(numItems+1, -1);

sizes = vector<int>(numItems+1, 1);

}

DisjointSet::~DisjointSet(){}

int DisjointSet::ds\_find(int item){

if(item>items.size()) return -2; //item doesn't exist in set

if(items[item]==-1) return item; //item is a sentinel node

vector <int> v;

int temp = item;

//Keeps track of which nodes are children of a specific sentinel

while(items[temp] != -1){

v.push\_back(temp);

temp = items[temp];

}

//Implements path compression using the data from the previous loop

for(unsigned i=0; i<v.size()-1; i++){

items[v[i]] = temp;

}

return temp;

}

bool DisjointSet::ds\_union(int item1, int item2){

int sentinel1 = DisjointSet::ds\_find(item1);

int sentinel2 = DisjointSet::ds\_find(item2);

//Returns false if either of the items are not in the disjoint set or if they share the

//same sentinel node

if(sentinel1==sentinel2 || sentinel1==-2 || sentinel2==-2) return false;

int size1 = sizes[sentinel1]; //Size of item1's subset

int size2 = sizes[sentinel2]; //Size of item2's ubset

//Union by size

if(size1<size2){

items[sentinel1] = sentinel2;

sizes[sentinel1] = size2+size1;

}

else{

items[sentinel2] = sentinel1;

sizes[sentinel2] = size1+size2;

}

return true;

}

/\*MST hpp\*/

#ifndef MST\_HPP

#define MST\_HPP

#include "Edge.hpp"

#include "DisjointSet.hpp"

#include <iostream>

#include <queue>

using namespace std;

/\*

Class used to define operator for use in priority queue edge comparisons

\*/

class EdgePtrComp {

public:

bool operator()(Edge\*& lhs, Edge\*& rhs) const {

return \*lhs < \*rhs;

}

};

/\*

MST class with two fields.

int numnodes: number of nodes in MST

priority\_queue<Edge\*, vector<Edge\*>, EdgePtrComp>\* pq: priority queue of edges in MST

\*/

class MST{

protected:

int numnodes;

priority\_queue<Edge\*, vector<Edge\*>, EdgePtrComp>\* pq;

public:

MST(void);

~MST(void);

//loads data from file into priority queue

bool loadFromFile(const char\* in\_filename);

//adds edge to priority queue

void addEdge(vector<string> edgeinfo);

//builds mst using disjoint set of nodes and pq

void mstbuild(ofstream& out\_file);

//getter and setter to modify numnodes field

void setNumnodes(int n);

int getNumnodes();

//prints final mst to out\_filename

bool printToFile(const char\* out\_filename);

};

#endif

/\*MST cpp\*/

#include <vector>

#include <string>

#include <iostream>

#include <fstream>

#include <sstream>

#include "MST.hpp"

using namespace std;

MST::MST():numnodes(0){

pq = new priority\_queue<Edge\*, vector<Edge\*>, EdgePtrComp>;

}

MST::~MST(){

while(!pq->empty()){

delete pq->top();

pq->pop();

}

delete pq;

}

void MST::setNumnodes(int n){

numnodes = n;

}

int MST::getNumnodes(){

return numnodes;

}

bool MST::loadFromFile(const char\* in\_filename){

ifstream infile(in\_filename);

while (infile) {

string s;

if (!getline(infile, s)) break;

istringstream ss(s);

vector<string> record;

while (ss) {

string s;

if (!getline(ss, s, ' ')) break;

record.push\_back(s);

}

if (record.size() != 4) {

continue;

}

MST::addEdge(record); // Add id pairs into hashmap

}

if (!infile.eof()) {

cerr << "Failed to read " << in\_filename << "!\n";

return false;

}

infile.close();

return true;

}

/\*\*

\* Converts edge data from string to varoius integers representing

\* edge id, node1, node2, and weight. Pushes new edge to pq using

\* this data.

\*/

void MST::addEdge(vector<string> edgeinfo){

int id = stoi(edgeinfo[0]);

int node1 = stoi(edgeinfo[1]);

int node2 = stoi(edgeinfo[2]);

float weight = stof(edgeinfo[3]);

if(getNumnodes() < node1) setNumnodes(node1);

if(getNumnodes() < node2) setNumnodes(node2);

pq->push(new Edge(id, node1, node2, weight));

}

/\*\*

\* Build mst using Kruskal's algorithm. A disjoint set is

\* used to keep track of node relatoinships.

\*/

void MST::mstbuild(ofstream& out\_file){

DisjointSet ds(getNumnodes());

Edge\* curr = pq->top();

int counter = 0;

while(!pq->empty()){

counter++;

pq->pop();

int node1 = curr->getNode\_pair()[0];

int node2 = curr->getNode\_pair()[1];

bool flag = ds.ds\_union(node1, node2);

if(flag){

out\_file<<node1<<" "<<node2<<" "<<curr->getWeight()<<endl;

}

delete curr;

curr = pq->top();

}

}

/\*Tester file\*/

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

#include <vector>

#include "MST.hpp"

#include "Node.hpp"

#include "Edge.hpp"

#include "Graph.hpp"

#include "DisjointSet.hpp"

using namespace std;

void usage(char\* program\_name) {

cerr << program\_name << " called with incorrect arguments." << endl;

cerr << "Usage: " << program\_name

<< " roadmap\_file mst\_output\_file"

<< endl;

exit(-1);

}

int main(int argc, char\* argv[]) {

if (argc != 3) {

usage(argv[0]);

}

char\* roadmap\_filename = argv[1];

char\* mst\_output\_filename = argv[2];

MST\* a = new MST();

ofstream outfile(mst\_output\_filename);

a->loadFromFile(roadmap\_filename);

a->mstbuild(outfile);

outfile.close();

delete a;

Graph\* b = new Graph();

b->loadFromFile(mst\_output\_filename);

b->printGraph();

cout<<"The sum of the edges is: "<<b->getSum\_edges();

}