Project Report

Name of the project

Designing minimum-cost transportation network in rural area

Group Member

Qi Zheng
Xun Cao

Introduction

I. Background

Our project is to help a non-profit institution named ***Habitat for Humility*** to design paved roads among scattered villages in a large rural area. The goal of our project is to connect the villages by a simple transportation network. However, since the budget for this project is very tight, we want to minimize the total cost of paving the roads, while still ensure that local residents from any village can travel to any other village via paved roads.

The pavement work will be done on the existing natural dirt roads. As a result of differences in topography and distances, the estimated costs of paving different dirt roads would vary a lot. In our case, the construction cost for each dirt road is estimated by the joint effort of local governments, engineering consulting firms, and NGOs. This information has been provided to us.

There are two parts included in our tasks:

1. We now have nine RMAT graphs[1] at hand. There are synthetic graphs with power-law degree distributions and small-world characteristics. Each of the graphs represents a rural area with scattered villages. The village numbers range from several tens to several thousands. These graphs were obtained from field survey. Since there could be multiple roads between different villages, these graphs could have multiple arcs between same pair of vertices, namely, they are multigraphs. We want to find the minimum cost transportation network (road network to be paved) for each set.

2. Survey in the rural area could be harsh, and some dirt roads are difficult to locate. So the local government conducted additional survey in some area, and found new roads that had not been included in the original data set. They want the software to have the feature to re-compute the optimal network when new roads (edges in the graph) are added onto the

graph. So our codes need to take that into consideration.

II. Interpretation

As shown in Figure 1, each node in the graph represents a village, and each edge in the graph corresponds to an existing dirt road. The estimated cost of each potential road is also provided as the weight of each edge. The graph is a connected graph. Our software will output a road network which would provide minimal total cost.

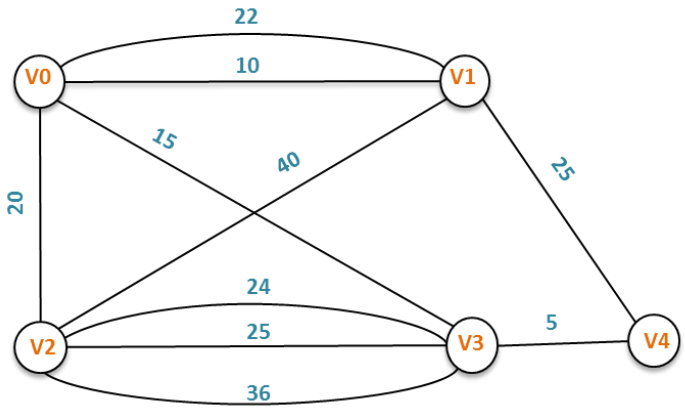

Figure 1 Graph Interpretation

Solution

I. Algorithm Design:

For the first part, the optimal road network with minimum cost can be computed by computing minimum spanning tree on the graph with weighted edges illustrated above. We use Prim's algorithm to find the MST. Prim's algorithm is a type of Greedy Algorithm. To implement this algorithm, we would start from a random root node s, and greedily "grow" a tree T from s outward. At each step, we will add the cheapest-cost edge e to T which has exactly one end point in T. When all nodes have been included in T, the algorithm is done, and the solution is found. An illustration for the process is provides in Figure 2.

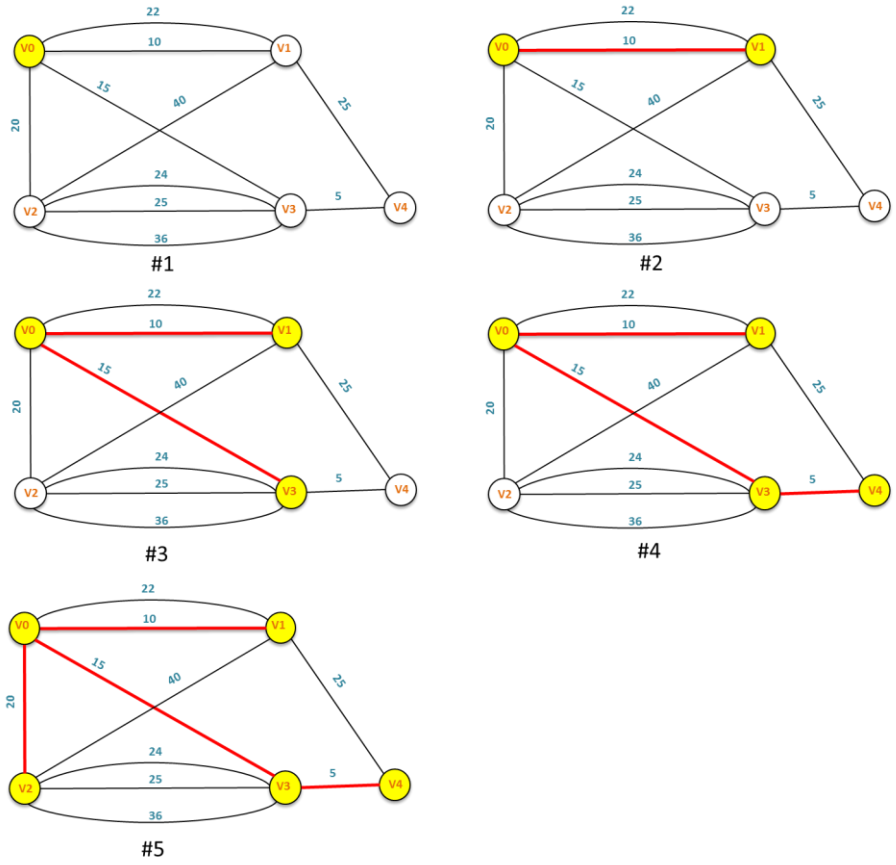

Figure 2. An illustration of the Prim's Algorithm

For the second part, to be able to update the transportation network (MST) with additional edges, we can, of course, simply add the new edge to the original graph, and re-run the simulation in the first part. However, it could be time consuming to rerun the model, especially given very large graphs. So we want to design a more efficient algorithm to recompute the MST with any new edge to be added. We have already computed the MST, now given an new edge (u, v, w) (u and v represent two end nodes, and w represents the weight of the edge between them, we will use the same notations in the following texts), we can perform a depth first search (DFS), to find the path on MST from u to v. We pick the max weight edge out of all the edges in the path, then, compare it with the new edge we want to add, keep the smaller weight one in the MST, and discard the larger weight edge. This process is illustrated in an example in Figure 3. It works because of the cycle property of MST. From Kleinberg and Tardos[2], "Assume that all edge costs are distinct. Let C be any cycle in G, and let edge e = (v, w) be the most expensive edge belonging to C. Then e does not belong to any minimum spanning tree of G".

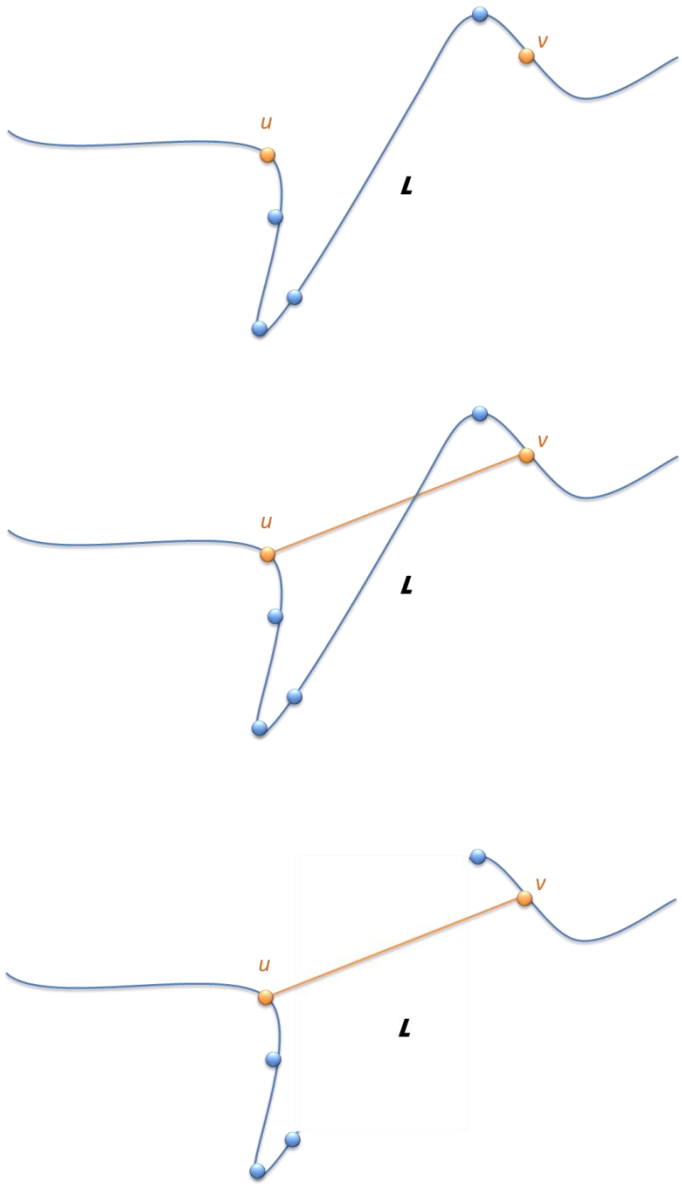

Figure 3. An illustration of how Part 2 works (the new edge (u, v, w) to be added is the orange edge in the graph)

II. Pseudo Code:

Part One: Compute the MST

```
parseEdge(G(V,E))
{
    // Initialize graph G as linked list of adjacent edges to each node
    // It is better to store the edges since we want to use edges for this problem
    list** G = (list**)malloc(V * sizeof(list));
    for (i=0; i<V; i++)
        G[i] = create_list(i);
    //Read in the raw graph data from the input file, and parse them to G
    for each E(u, v, w) read in from input file
```

```
        {
                add_edge_to_list(G[u], u, v, w);
                add_edge_to_list(G[v], v, u, w);
        }
}

Initialization()
{
        // create an arraylist to store the selected edges for MST
        edge** MST = malloc((V-1)*sizeof(edge));
        // construct the heap prior queue: heapq
        edge** heapq = malloc(2*E*sizeof(edge));
        // construct an array to store the visited node numbers
        int *S = (int*)calloc(V, sizeof(int));
}

primMst(G)
{
    current = 0;          // choose v0 as the source node
    T[current] = 1;          // mark the source node as visited
    // find the (V-1) edges to compose the MST with V vertices
    for (i=0; i<V-1; i++)
    {
        // add all edges with exactly one vertex in MST into the candidate pool
        for each edge e linked to G[current]
            push e into heap_pq;
        while (1)
        {
            curr ← pop out an edge e from heap_pq; //pop out the min w edge
            if ((e.u in MST && e.v not in MST)||(e.u not in MST && e.v in MST ))
            {
                MST[i] ←e;                    // add the edge to MST
                current ← the node outside of MST;      // move on
                                                        // to the next vertex
                break;
            }
        }
    }
}
```

Part Two: reCompute the MST

```
DFSfindMaxEdge(MSTG, u, v){
    label u as visited
    for all edges from u to w in MST.adjacentEdges(v){
            if (w = v)
                    return the edge
```

```
            else if vertex w is not labeled as visited
                recursively call DFSfindMaxEdge(MST, w, v)
            }
            keep track of the max edge at each level of recursion when an edge
            returns
}
 if new_edge.weight < DFSfindMaxEdge.weight{
      replace DFSfindMaxEdge with new_edge
 }
```

Input and Output

This software set includes two parts:
1. Codes to compute MST for given graph
2. Codes to compute the new MST when a new edge needs to be added

There are two folders:
1. "src" folder contains all source files (codes)
2. "data" folder contains all input graphs (.gr files), and the output (computed) new graphs (.txt files) will also be placed in the "data" folder.

Input and Output

1. In the first part, "compute_MST.c" will take a "originalName.gr" graph file as input file, and output the MST graph named as "originalName_MST.txt" in the same "data" folder. You will have to type in the name of your input graph (without file extension) in the command line as an argument to the software.
   The input rmat (.gr) file is formatted as:
   Line 1: V E
   (V is the total vertex number, and E is the total edge number)
   Line 2 to E: u, v, w
   (u, and v are two nodes, and w is the weight of the edge in between)
   The format of the output file is similar to the input .gr file, and will be further discussed in the Results section.

2. In the second part, "reCompute.c" will take a txt file named "new_edge.txt", which contains one new edge to be added to the graph. You can change the edge to any one edge, but please don't change the file name. Then you will choose which MST graph generated from Part 1 you want to add the new edge to from the output files ("originalName_MST.txt" from Part 1). You will provide the input MST graph name (without file extension) to the software via command line argument. The software will output the updated MST graph named as "originalName_MST_new_edge.txt", and place them in the same "data" folder.

   The format of the output file will be a linked list representation, and will be

further discussed in the Results section.

Please feel free to modify data in the .gr file or “new_edge.txt” file to test the software.

Execution

The command lines for running the software for both parts will take an argument of the input file name (without file extension). So use the following command line to execute the program (in Windows):

For Part 1: Use the command lines similar as:
gcc compute_MST.c -o compute_MST
.\compute_MST input_graph_name

For Part 2: Use command lines similar as
gcc reCompute.c -o reCompute
.\reCompute MSTgraph_name

For example, we want to load rmat1.gr as the input file for Part 1, use the following command lines:
gcc compute_MST.c -o compute_MST
.\compute_MST rmat1

Then we want to add the additional edge (from ”new_edge.txt”, but you don’t need to specify the edge file.), use the following command lines:
gcc reCompute.c -o reCompute
.\reCompute rmat1_MST

Results and Time Complexity

Results

The result graph files will be automatically placed in the “data” folder.

Part 1. The “rmat(i)_MST.txt” file is the result file for the first part, they each contains an MST computed for each input graph. The first line contains the minimum cost of the network. The second line contains two values: total number of vertices, and total number of edges in the graph. The 3 to n lines each contains an edge in the MST in the format of u, v, w, where u and v are two nodes and w is the weight of edge between them. There are no repeated edges in the output file(namely, no (u, v, w) and (v, u, w) for the same edge will appear in this output file). Note that when you run the software, it will print out the MST graph in the linked list format on screen. You can easily retrieve the MST graph from the output file.

Part 2. The “rmat(i)_MST_new_edge.txt” files are result files for part two, with an additional edge provided in the “new_edge.txt” file. They are formatted as linked list format. The first line provides the updated cost. The 2 to n lines, each starts with an integer which is the index of the node in MST, followed by a set of edges connected to it in the form of (u, v, w). You can also simply construct the updated MST graph from such output files. Note that when you run the software with an added edge, the software will print on the screen whether an edge in the original MST graph will be replaced by the new edge, and if so, which old edge (u, v, w) will be replaced by the new edge (u’, v’, w’).

Time Complexity

In the first part, we pushed all E edges into the heap, and pop them out according to their weights in ascending order, and decide if the popped edge should be added to the MST graph (since the original graph was a multigraph with multiple arcs between same pair of edges, hence the two nodes of the popped edge could have already been in the MST). Hence the big-O time complexity for this part is O(Elog(E)) since E>V. In the second part, we perform DFS on the MST (V vertices, V-1 edges) to find the max edge along the path from u to v, hence, the big-O time complexity for this part is O(V).

Discussion: MST Applications

I. Network design.

Telephone, electrical, hydraulic, TV cable, computer, road

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn’t a tree you can always remove some edges and save money.

II. Approximation algorithms for NP-hard problems.

Traveling salesperson problem, Steiner tree[3][4][5]

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, it’s a special kind of tree. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it’s a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

III. Indirect applications.

- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- model locality of particle interactions in turbulent fluid flows
- autoconfig protocol for Ethernet bridging to avoid cycles in a network

IV. Cluster Problem[6]

Let U be the set of n objects labelled $P_1, P_2, \ldots, P_n$. For every pair P_i, P_j, we have a distance d(P_i, P_j). We require d>0. Given a positive integer k, a k-clustering of U is a partition of U into k non-empty subsets or "clusters" C_1, C_2,…, C_K. The spacing of a clustering is the smallest distance between objects in two different subsets.

We can use Kruskal' s algorithm to solve clustering as a mst problem. First, let C be a set of n clusters, with each object in U in its own clusters. We can process pair of objects in increasing order of distance. Add $C_p \cup C_q$ to C, and delete C_p, C_q from C. Stop when there are k clusters in C.

Reference

[1] Chakrabarti, Deepayan, Yiping Zhan, and Christos Faloutsos. *"R-MAT: A Recursive Model for Graph Mining."* SDM. Vol. 4. 2004.

[2] Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. Boston:Pearson/Addison-Wesley, 2006. Print.

[3] Nilsson, Christian. *Heuristics for the traveling salesman problem*. Tech. Report, Linköping University, Sweden, 2003.

[4] Reinelt, Gerhard. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.

[5] Reinelt, Gerhard. *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.

[6] Robert Sedgewick, Algorithms in Java, 3rd Edition