CSED 233: Assignment #5

Due on Friday, December 12, 2014

BoHyung Han 12:30pm

Newell Wunrow

Problem and Solution

I will give a brief explanation of the problem and my solution.

Firstly, the assignment was to find the shortest path between two subway staitons in the Seoul subway system, and then to output the minimum distance,the transfer stations and the distances between them. A transfer stations is defined to be a station where you switch from one line to the other.

In short, I solved this problem by first organizing the data given to create a graph where the vertices represented subway stations and the edges represented the railways between each station. One key data manipulation I did was that I added an edge with weight 0 at each transfer station. For example, if you wanted to transfer from line 3 to line 2 at you would travel along the edge from station 0330 to station 0224. I then used Dijkstra's Algorithm to find the minimum distance path between the given source and destination stations.

Explanation of Implementaiton

Here is a brief explanation of my implementations.

SeoulSubway Class

This is the main class which reads the arguments given in the command prompt. First, we parse the data given the text files by utilizing the method already provided in the ReadSubwayData class. Next for each row in the text files we create an object of type edge and vertice. As stated on the previous page, I then inserted edges at transfer stations with length 0. I then created a graph that represents the Seoul subway station system using these two lists of edges and vertices. Next for each edge in the graph, we assign a start and finish vertex in the list vertices. We then call the dijkstra function to find the minimum path between our source and destination stations. Finally, we output our results in the format as stated in the assignment pdf.

Edge Class

The edge class I created has 8 variables.

line : represents the line the railway is on

station name1 : the name of the start station

station code1 : the code of the start station

station name2 : the name of the finish station

station code2 : the code of the finish station

weight : the distance between the two stations

start : the vertex associated with the start station

finish : the vertex associated with the finish station

Within the class there are basic constructor, mutator, and accessor methods.

Vertice Class

The vertice class has 7 variables.

line : the line the station is on

name : the name of the station

code : the code of the station

lat : the longitude of the station, was not used in the program at all

lon : the latitude of the station, was not used in the program at all

distance : the distance from the source station, used in Dijkstra's Algorithm

previous : the station visited previous in a path, used in Dijkstra's Algorithm

Within the class there are constructor, mutator, and accessor methods.

Graph Class

The Graph class has two data fields.

Vertices : list of stations in the subway graph

Edges : holds the railways between stations in the subway graph

The functions in this class include contrsuctor methods and the following:

findVertices(String c) :
this searches the list of vertices in the graph for a vertec with a specific code c

findVertices(String c, List V) :
this searches a specific list V for a vertex with a specific code c

isEmpty( ) :
returns true if there is at least one vertex in the graph, otherwise fale

getEdgeDistance(Vertice a, Vertice b) :
returns the distance between two adjacent vertices

incidentVertice(Vertice v) :
returns a list of all the incident stations of a station, if it is a transfer station this includes all the incident stations on other subway lines, it was used in Dijkstra's Algorithm

removeMin(List V) :
removes the minimum distance vertex from the graph and returns it, it was used in Dijkstra's Algorithm

replaceKey(Vertice v, double d, List V) :
sets the distance for a specific node in a list of nodes,, it was used in Dijkstra's Algorithm

transferfy(List V) :
returns the list of transfer stations along a path, it also calculates the distance between each transfer station and assigns it to the distance variable of the starting station, it was used in Dijkstra's Algorithm

displayVertice :
displays the line, code and distance information for each vertex in a list, this was used A LOT for debugging purposes

Dijkstra's Algorithm

Here is a brief description of Dijkstra's algorithm.

1. Set the distance at the starting vertex to 0 and each vertex to inifinity.
2. Remove the vertex with the smallest distance.
3. For all the incident vertices v to the removed node, if $distance_{edge} + distance_{removed} < distance_v$, then update the distance.
4. In order to keep track of the path, if the distance of a vertex is updated set the previous vertex to the removed minimum distance vertex.

After completing this, you have a minimum spanning tree of the graph. (aka the shortest path from the starting vertex to all other vertices denoted by $path$). Next I defined a new path $finishPath$ with minimum distance from the starting to ending vertices by backtracking along $path$.
Finally, in compliance with the assignment. I created a list of the transfer vertices visited along $finishPath$. To do this I simply backtracked along $finishPath$ until I found a vertex where its previous vertex had a different line. To get the distance between the transfer vertices I simply use the Dijkstra distance (denoted by $distance_i$ at each vertice with the following formula:

$$transferDistance_{a,b} = distance_b - distance_a$$

We now have our desired list of transfer stations and distances between them for the output.
A copy of my code is in the appendix for reference.

Appendix

Listing **??**

Listing 1: SeoulSubway

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class SeoulSubway extends ReadSubwayData{

     public static void main(String[] args) throws IOException{
          if (args.length == 4){

               //Check to see if source and destination stations are the same
               if(args[0].equals(args[1])){
                    System.out.println("Source and desintation are the same, you don’t need to g
                    return;
               }

               //read vertices and edges from file
               List<String[]> vertices_string = parseVertices("subway_vertice.txt");
               List<String[]> edges_string = parseEdges("subway_edges.txt");

               //initialize Vertice and Edge lists to store data from file
               List<Graph.Vertice> vertice = new ArrayList<Graph.Vertice>(vertices_string.size()
               List<Graph.Edge> edge = new ArrayList<Graph.Edge>(edges_string.size());

               //for each row in the subway_vertice.txt file a new vertex is created and added t
               for(String[] v : vertices_string) {
                    Graph.Vertice a = new Graph.Vertice(Integer.parseInt(v[0]),v[1],v[2],Double.
                    vertice.add(a);
               }

               //for each row in the subway_edges.txt file a new edge is created and added to th
               for(String[] e : edges_string){
                    Graph.Edge b = new Graph.Edge(Integer.parseInt(e[0]), e[1], e[2], e[3], e[4]
                    edge.add(b);
               }


               //creates an edge between transfer stations
               for(Graph.Vertice i : vertice){
                    for(Graph.Vertice j : vertice)
                         if(i.name.equals(j.name) && !i.code.equals(j.code)){
                              Graph.Edge b = new Graph.Edge(10, i.getName(), i.getCode(), j.getN
                              edge.add(b);
                         }
               }

               //initialize a graph using the Vertice and Edge lists just created
               //this graph represents the subway network
               Graph subway = new Graph(vertice, edge);

               //for each edge in the graph we assign a start and finish vertex to make it direc
               for(Graph.Edge e : subway.Edges){
                    e.setStart(subway.findVertice(e.station_code1));
                    e.setFinish(subway.findVertice(e.station_code2));
               }

               //we find the source and destination vertices given as the first two arguments
               Graph.Vertice source = subway.findVertice(args[0]);
               Graph.Vertice destination = subway.findVertice(args[1]);



               //use a variation of Dijkstra's algorithm to return a list of transfer stations a
               //path from the destination station to the source station
               //a transfer station is defined to be a station where you switch from one line to
               List<Graph.Vertice> transfer_stations = subway.Dijkstra(subway, source, destinati

               //calculate the total distance from the source vertex to destination vertex by su
               double total_distance = 0;
               for(Graph.Vertice v : transfer_stations){
                    total_distance += v.getTransferDistance();
               }

               //output the result of the path in the format:
                    //STATION_CODE1, STATION_CODE2, ALL_DISTANCE, TRANSFER_CODE1, TRANSFER_CODE2
               String msg = String.format("%s, %s, %f", source.getCode(), destination.getCode(),
               System.out.print(msg);
               for(Graph.Vertice v : reversed(transfer_stations)){
                    if(v != destination)
                         System.out.print(", " + v.getCode());
               }
               for(Graph.Vertice v : reversed(transfer_stations)){
                    String msg2 = String.format(", %f", v.getTransferDistance());
                    System.out.print(msg2);
               }

          }

          else{
            // Print error message if wrong given arguments
            System.out.println("Wrong argument is given");
          }
     }

     //reverses the order of a list of vertices by utilizing the ListIterator Class
     //this was needed since my dijkstra's algorithm returns a path from destination to source;
     public static List<Graph.Vertice> reversed(List<Graph.Vertice> original){
          List<Graph.Vertice> reverse = new ArrayList<Graph.Vertice>();
          ListIterator<Graph.Vertice> iter = original.listIterator(original.size());
          while (iter.hasPrevious()){
               Graph.Vertice vertex = iter.previous();
```

```
               reverse.add(vertex);
          }
          return reverse;
     }
}
```

Listing ??

Listing 2: Graph

```
import java.util.List;
import java.util.ArrayList;


public class Graph {
     public static class Edge {
          public int line;                                        //stores which line the subway
          public String station_name1;                       //the name of the start station
          public String station_code1;                       //the code of the start station
          public String station_name2;                       //the name of the finish station
          public String station_code2;                       //the code of the finish station
          public double weight;                                   //the distance between the two
          public Vertice start;                                   //the vertex associated with th
          public Vertice finish;                                  //the vertex associated with th

          //Constructor method
          Edge(int a, String b, String c, String d, String e, double f){
               line = a;
               station_name1 = b;
               station_code1 = c;
               station_name2 = d;
               station_code2 = e;
               weight = f;
               start = null;
               finish = null;
          }

          //mutator for start
          public void setStart(Vertice s){
               start = s;
          }

          //mutator for finish
          public void setFinish(Vertice f){
               finish = f;
          }
     }

     public static class Vertice {
          public int line;                              //the line the station is on
          public String name;                           //the name of the station
          public String code;                           //the code of the station
          public double lat;                            //the longitude of the station, was not u
          public double lon;                            //the latitude of the station, was not us
```

```
        public double distance;                      //the distance from the source station, u
        public Vertice previous;                //the station visited previous in a path, used
        public double transfer_distance;             //the distance from one transfer station

        //Constructor method
        Vertice(int a, String b, String c, double d, double e){
             line = a;
             name = b;
             code = c;
             lat = d;
             lon = e;
        }

        //Constructor method
        Vertice(String a){
             code = a;
        }

        //Mutator for distance
        public void setDistance(double d){
             distance = d;
        }

        //Mutator for distance_transfer
        public void setTransferDistance(double d){
             transfer_distance = d;
        }

        //Accessor for distance
        public double getDistance(){
             return distance;
        }

        //Accessor for distance_transfer
        public double getTransferDistance(){
             return transfer_distance;
        }

        //Accessor for line
        public int getLine(){
             return line;
        }

        //Accessor for name
        public String getName(){
             return name;
        }

        //Accessor for code
        public String getCode(){
             return code;
        }
```

```
          //Mutator for previous
          public void setPrevious(Vertice p) {
               previous = p;
          }

     }

     public List<Vertice> Vertices;          //holds the stations in the subway graph
     public List<Edge> Edges;           //holds the railways between stations in the subway grap

     //Default constructor method
     Graph(){
          Vertices = new ArrayList<Vertice>(300);
          Edges = new ArrayList<Edge>(300);
     }

     //Constructor method
     Graph(List<Vertice> v, List<Edge> e){
          Vertices = v;
          Edges = e;
     }

     //searches all vertices for a specific subway code the and returns that vertex if found
     public Vertice findVertice(String c){
          for(Vertice v : Vertices){
               if(v.code.equals(c))
                    return v;
          }
          return null;
     }

     //returns true if there is at least one station in the graph
     public boolean isEmpty(){
          if (Vertices.size() == 0)
               return false;
          else
               return true;
     }

     //returns the distance between two adjacent vertices
     public double getEdgeDistance(Vertice a, Vertice b){
          for(Edge e : Edges){
               if((a.name.equals(e.station_name1) && b.name.equals(e.station_name2)) ||
                         (b.name.equals(e.station_name1) && a.name.equals(e.station_name2))){
                         return e.weight;
               }
          }
          return 0; //throw exception
     }

     //returns a list of all the incident stations of a station
     //if it is a transfer station this includes all the incident stations on other subway lines
     public List<Vertice> incidentVertice(Vertice v){
        List<Vertice> incident= new ArrayList<Vertice>();

        for (Edge e : Edges){
            if (v.code.equals(e.station_code1)){
                incident.add(e.finish);
            }
            if (v.code.equals(e.station_code2)){
                incident.add(e.start);
            }
        }
        return incident;
    }

    //My implementation of Dijkstra's algorithm
    //returns a list of transfer stations and the distances between them along a path from sour
    public List<Graph.Vertice> Dijkstra(Graph G, Graph.Vertice s, Graph.Vertice f){
        double inf = Double.POSITIVE_INFINITY;
        List<Graph.Vertice> path = new ArrayList<Graph.Vertice>();        //this is the shortes
        List<Graph.Vertice> finishPath = new ArrayList<Vertice>();        //this is the shortes
        List<Graph.Vertice> transfer = new ArrayList<Vertice>();          //this is the list of

        //set the distance from the source vertex to itself to 0
        //set the distance from the source vertex to all other vertices to infinity
        for(Graph.Vertice v : Vertices){
            if (v.code.equals(s.code)){
                v.setDistance(0);
            }

            else{
                v.setDistance(inf);
            }
        }

        //copy all the vertices to keep track of which vertices have been visited
        List<Graph.Vertice> Q = Vertices;

        //visit each vertex and relax its adjacent vertices
        while(!Q.isEmpty()){
            //remove the vertex with the smallest distance from the source node that has not
            Graph.Vertice u = removeMin(Q);

            //break out of the loop if the minimum distance node is infinity
            //this will only happen if the graph is disconnected
            if (u.distance == inf)
                break;

            //relax all of the removed vertex's adjacent vertices
            //update the distances and add the minimum spanning graph
            for(Graph.Vertice v: G.incidentVertice(u)){

                if(v.getDistance() > u.getDistance() + G.getEdgeDistance(u,v)){
                    v.setDistance(u.distance + G.getEdgeDistance(u,v));
                    replaceKey(v,u.getDistance() + G.getEdgeDistance(u,v), Q);
                    v.setPrevious(u);
                    if(path.contains(v))
                        path.remove(v);
                    path.add(v);
                }
            }
        }

        //find the destination station in the minimum spanning graph created by dijkstra's alg
        Graph.Vertice target = findVertice(f.code, path);
        Graph.Vertice counter = target;

        //create the minimum distance path from the source to destination by "backtracking" al
        while(counter.previous != null){
            finishPath.add(counter);
            counter = counter.previous;
        }
        finishPath.add(s);

        //record the transfer stations along the path
        transfer = transferfy(finishPath, s);
        return transfer;

    }

    //removes the minimum distance vertex from the graph and returns it
    public static Graph.Vertice removeMin(List<Graph.Vertice> V){
        Graph.Vertice min = V.get(0);
        for(Graph.Vertice v : V){
            if(v.distance < min.distance)
                min = v;
        }
        V.remove(min);
        return min;
    }

    //sets the distance for a specific node in a list of nodes
    public void replaceKey(Vertice v, double d, List<Graph.Vertice> V){
        findVertice(v.code, V).setDistance(d);
    }

    //searches for a specific node by its station code in a list of nodes
    public Vertice findVertice(String c, List<Graph.Vertice> V){
        for(Vertice v : V){
            if(v.code.equals(c))
                return v;
        }
        return null;
    }

    //returns the list of transfer stations along a path
    //it also calculates the distance between each transfer station and assigns it to the dista
    public List<Graph.Vertice> transferfy(List<Graph.Vertice> V, Vertice start){
          List<Graph.Vertice> transfer = new ArrayList<Graph.Vertice>();
          Graph.Vertice t = V.get(0);
          transfer.add(t);
          while(t.previous != null){
               while(t.previous != null && t.previous.getLine() == t.getLine()){
                    t=t.previous;
               }
               if(t.previous == null)
                    break;
               t = t.previous;
               transfer.add(t);
               if(t.previous == null)
                    break;
               t = t.previous;
          }

          for(int i = 0; i < transfer.size() - 1; i++){
               transfer.get(i).setTransferDistance(transfer.get(i).getDistance() - transfer.get(
          }
          transfer.get(transfer.size()- 1).setTransferDistance(transfer.get(transfer.size() - 1)

          return transfer;
     }

     //displays the line, code and distance information for each vertex in a list
     //this was used A LOT for debugging purposes
     public void displayVertices(List<Graph.Vertice> V){
          for(Graph.Vertice v : V){
               System.out.println(v.line + " " + v.code + " " + v.transfer_distance);
          }
     }

}
```