Assignment 4: Paris Metro: Riding Paris Subway Network

CSI 2110 Section B

Lucia Moura

Fall 2017

Madison Morgan, 8287926

Due: December 6th, 2017

Introduction

This assignment requires us to create a graph, a provide a number of operability's on this graph, to find directed paths, shortest paths, and shortest paths with exceptions. This graph is modeled by the Paris Metro Network, as provided to us in "metro.txt". Which was then read using the static method `readMetro(filename)`. We read the stations into vertices and the travel lines and travel time into weighted directed edges for our intended graph.

The main objectives of this assignment is to find the shortest path in a real graph, using the subway of the city of Paris as an example.

Data Structure

The data structure used to hold this directed, weighted representation of a graph using an Adjacency List structure as implemented by myself (however the supporting Vertex class was modified from [3]). Essentially this data structure contains instance variables:

- Array of vertices, holding all the vertices
- Integer declaring the number of vertices in the graph
- Integer declaring the number of edges in the graph

Each vertex will hold an edge-list specifying the directed point of travel from the source (vertex in array) to destination (edge list's specified vertices).

This data structure was used as getting/setting vertices and edges could be provided in linear time… As our intended operability required a lot of access and comparison to vertices and edges, this was the most efficient data structure to implement our graph in- to make sure our algorithms run at the most efficient time as possible!

Classes

To create this representation and support the necessary operability, the following classes were created:

- **Vertex (adapted from [3]):** This class was used to encapsulate a vertex in our graph, or a station in our subway network.

  This class had the following instance characteristics:

  - String declaring the station name
  - List of Edges declaring all outgoing edges for which the Vertex is the source of
  - List of Edges declaring all incoming edges for which the Vertex is the destination of
  - Integer declaring the position of the Vertex
  - Double declaring the minimum distance from the source node (used for Shortest Path Algorithms, as explained later)
  - Vertex to represent the previous Vertex visited on the shortest path

  This class also implemented the comparable interface so that it may be compared to other Vertices when calculating the shortest path. As well as a method for conversion of a Vertex to a string for printing purposes.

- **Edge:** This class was used to encapsulate a weighted, directed edge in our graph, or a travel path from one station (source, specified by Vertex) to another (destination, specified by Edge) and its travel time.

  This class had the following instance characteristics:

  - Double declaring the weight of the edge
  - Vertex declaring the destination of travel
  - Boolean declaring whether the edge is broken or not

  Otherwise, this class contained well as a method for conversion of an Edge to a string for printing purposes.

- **AdjacencyList:** This class holds our graph implementation, represented as an Adjacency List, as described under “Data Structures”.

  Additionally, we can see getter methods for the variables tracking the total number of edges and vertices; as well as a method for a conversion of a list to a string.

- **ParisMetro:** This class is meant to hold the subway network, represented as graph (or more specifically an Adjacency List). And provide/direct operability’s unto this network.

  This class has the following instance characteristics:

  - Adjacency List providing a representation of the Paris Subway Network

Additionally this class contains a method to read a text file, readMetro- as called by the constructor of this class, it takes a text file (namely metro.txt) and converts it to an adjacency list representation of the subway network. This code was modified from the Lab 10, WeightedGraph.java [1].

Also, this class will have its main method to direct the program as specified by the User input to initialize a Paris Metro object and perform the correct operation on the object as designated by the command prompt input.

Additionally we have the `identifyAllStations(N1)`, method to implement an algorithm to identify all stations belonging to the same line of a given station, N1 (explained in more detail in the Algorithms section).

We also have a method to break vertices and edges when the station is broken for Q 2iii).

- **Dijkstra (adapted from [2] and [3]):** This class implements the Dijkstra algorithm to calculate the shortest path from one station to another- with perhaps an exception. This class was modified from [2] and [3].

  This class contains the following algorithms: an algorithm to compute all the shortest paths from vertex source to all reachable vertices in a graph by the source

  Another method is contained to calculate the shortest path from the source to the target.

Algorithms

2i. Path of Station

This algorithm is implemented in the ParisMetro class`identifyAllStations`. Given a position, we seek to find all stations on the same line as the station in pos. The algorithm used for this implementation uses a form of recursion, as described in detail in the class file…

We begin by iterating through the outgoing edges/paths of the vertex/station called – lets call it the source vertex (ignoring all walking edges), we print these edges. We know visit all target vertices that stem from the source vertex- and so on until all vertices belonging to the path has been visited. we do this recursively by calling the function again with the target vertices and performing the same action. In the end, all visited vertices of the path indicated by pos are printed.

It is also noted that we track the position of the vertices visited to account for bidirectional edges/ cycles.

2ii. Shortest Path from one Station to Another

This algorithm use Dijkstra's algorithm as studied in class to determine the shortest path from one vertex to another, as adapted from [2] and [3]. It can be found in the Dijkstra.java file. It encompasses two helper methods to be called consecutively.

`computePaths`: To calculates the shortest path from the source vertex to all reachable vertices using Dijkstra's algorithm. It will take the unvisited vertex with the minimum weight, visits all of its neighbors (comparing their distances), update the distance for all the neighbors (in queue/cloud), and repeats this process until all reachable nodes are visited. This algorithm will also set a path from each reachable vertex to the source vertex, for the next helper method to acknowledge this path. This algorithm will also account for already visited nodes (in the case of cycles) and broken edges/stations.

`getShortestPathTo`: This method will be called consecutively after the above, this algorithm will calculate the shortest path from the source vertex to the target, these two methods are used conjunctively. The algorithm behind this is it will iterate through the target Vertex to the previous Vertex , to the previous and so on…(the path set by the premier method), until it reaches the end of the path (marked by a null). This produces a path from the target to the source vertex- so we must reverse this path to get the shortest path from the source to the target vertex and return it, where it will be printed in main.

2iii. Shortest Path from one Station to Another with Broken Line

This question uses the same methods as described above in 2ii). However, before these methods are conjunctively called- we use another method found in the ParisMetro.java, `breakEdges`. This method will be called before the methods defined in the Dijkstra.java class.

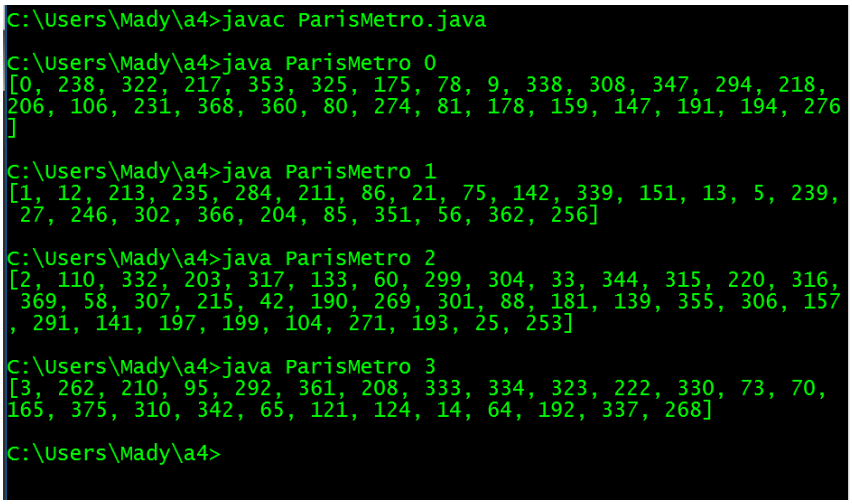
This method algorithm is as follows: given a vertex v, it will iterate through all the incoming and outgoing edges of the vertex- setting them to be broken (i.e. broken=true). Therefore when consecutively `computePaths` is called, it will not recognize these edges as part of the path and will continue the algorithm as described, as well `getShortestPathTo`.

Testcases

For the purposes of this section please refer to Lucia's corrected test solutions.

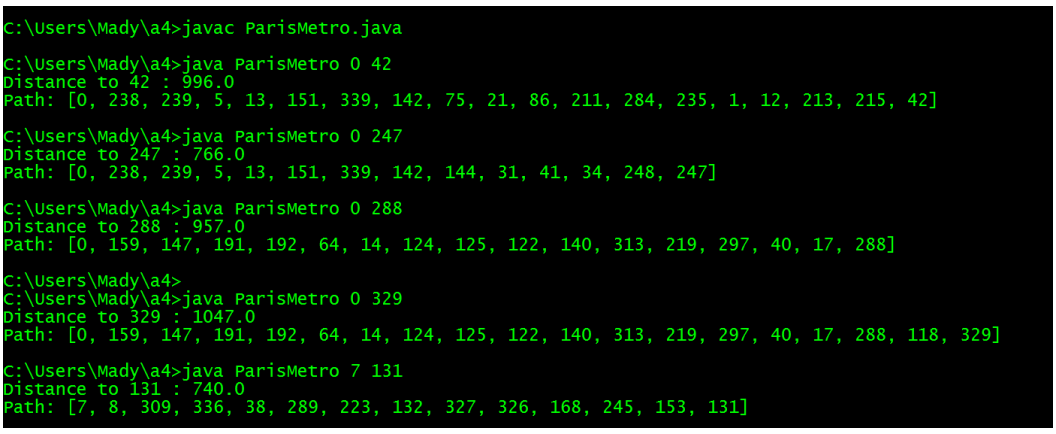
2i. Path of Station

As follows, here are some test case screenshots of some correct inputs and outputs, as checked by the corrected test solutions documents, with some discrepancies (see discussion).



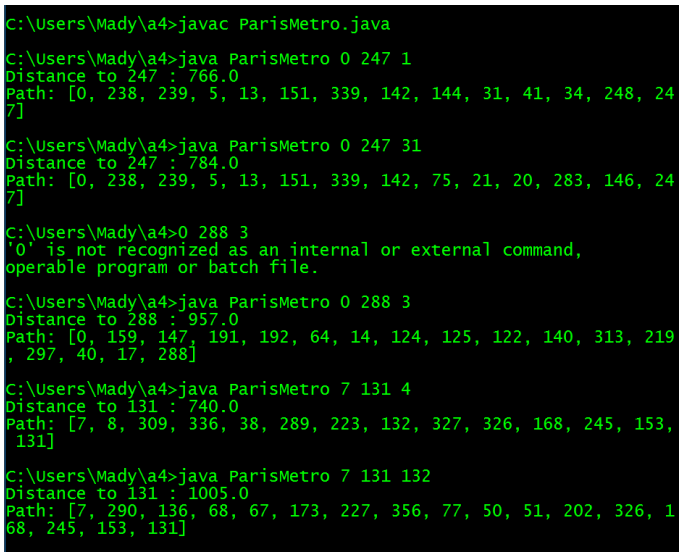
2ii. Shortest Path from one Station to Another

As follows, here are some test case screenshots of some correct inputs and outputs, as checked to the corrected test solutions document.



2iii. Shortest Path from one Station to Another with Broken Line

As follows, here are some test case screenshots of some correct inputs and outputs, as checked by the corrected test solutions documents, with some discrepancies (see discussion).



Discussion

There were some discrepancies for my output results and the given results for question 2i)- particularly for higher values of N1, the path lists were smaller and smaller. I found they remained approximately the same length. Upon further inspection of the metro.txt and my code I was not able to find the source of error. I can only speculate that the error may have stemmed from an edge that shouldn't have been taken- however I did account for walking edges and already covered edges in my algorithm.

There were also discrepancies for Question 2iii), the shortest path would only be altered from one end to another if N3 was part of the actual path. Other then that the original path would be displayed. I believe this is most likely to the inability of the function to "relax" the edges. In other words if the path [u,v] > path [u,w,v] – the shorter path is not taken. Perhaps it is just a simple error of the priority queue, for still recognizing the broken vertex despite my best attempts to fix this.

Conclusion

In conclusion, we can see that most algorithms are implemented correctly (as indicated by the test cases) or have a sound backing theory- completing the objectives of this assignment. Due to the large nature of this graph, it was slightly difficult to ensure that all test cases were completely correct and to debug our mis-cases.

References

[1] WeightedGraph.java. CSI 2110: Lab 10 Shortest Paths. Virtual Campus (2017).

[2] https://stackoverflow.com/questions/17480022/java-find-shortest-path-between-2-points-in-a-distance-weighted-map

[3] https://github.com/vaishnav/Dijkstra/blob/master/Dijkstra.java