Oliver Benning
CSI 2110 Fall 2017
7798804

Assignment 4 report

Chosen data structures

Global

Graph metro:
The metro is stored in a graph. This is to reflect the structure of the metro system which contains the stations as vertices and the directed line segments between them as weighted edges corresponding to the travel time. The net.datastructures.AdjacencyMapGraph implementation was chosen as it implemented all the required functionality, including the ability to specify a directed graph on creation and to return iterators of edges and vertices.

String[] stationNames:
This is simply an array of station names to print the stations' names during the output. The structure is an array because we are using all the natural numbers within a [0 – n] range and both set and get operations are constant time.

ArrayList vertices:
A java.util.ArrayList of vertices. The vertices are stored in an ArrayList in addition to being accessible itertively through metro.vertices() because since our vertices follow a range from [0-n] we can get constant time sets and gets from the underlying array rather than having to iterate in linear time to find a vertex.

stationsAlongLine(...) and propagateAdd(...)

ArrayList stationsAlongLine
Stores the visited stations along the line during method execution. Uses ArrayList for O(n) contains() method to check if we already visited a vertex.

Queue q
Queue to return. The ArrayList is emptied into this queue and the queue is returned, as later on all we need is the dequeue operation to get the vertices, either to print in 2 i) or to exclude in 2 iii).

ShortestPath(...) and dijkstraRecursive(...)

Stack D.edges
The edges we retrieve from the Dijkstra algorithm are stored in a LinkedStack for efficient retrieval in reverse order and unrestricted storage size.

Map D.disc and Map D.cloud
The Dijkstra discovery and cloud vertices are each stored in a Map to associate them with their variable minimum distances as values. This allows for efficient checking if the Maps contain specified vertices, and then retrieval and setting of the associated minimum distance.

PriorityQueue D.pq

Each time we discover a vertex, we also have a priority queue where the minimum value gets added with it's discovery edge as value. This gives us a map structure with the shortes distances as the keys this time. An unsorted priority queue is used for two reasons. The unsorted property lets us add a lot of elements in each run in O(1), and then pull the minimum in one O(n) operation. PriorityQueue is the most suitable Map ADT in this case since we're only concerned with pulling the minimum value.

Stack D.s

At the end of our run we have our stack D.edges with our top edge being the leading edge to our destination. We want to reverse this structure and also filter out the edges not part of our path from origin to destination. To do this we just empty edges into the new stack D.s, excluding edges we don't want. The emptying of one stack into another reverses the order from beginning with destination, to beginning with origin – which is the order we would like to finish with.

Description of algorithms

Question 2 i):

This is a simple recursive algorithm to find all vertices along a stations path. It is extremely short as it makes extensive use of the methods in AdjacencyMapGraph. The algorthm is as follows.

- For each edge of a vertex, whether incoming or outgoing:
- Check whether the edge is a walking edge (value of -1), if so, skip it.
- Identify the vertex opposite to the initial vertex on the edge,
- if it is not already in our list of identifies vertices, add it and call the method recursively for the new vertex.

When this terminates we have a list of vertices along the same line of our paramter vertex, which we transfer into a queue and return.

Question 2 ii) and iii)

ii) and iii) are similar with one extra step for iii) so I will describe it as one algorithm. This is essentially Dijkstra's algorithm and will be described assuming familiarity to terms such as "discovery edge" and "cloud". It is as follows:

- *Before recursion:*
- If a third parameter is specified, meaning a line to exclude, call the algorithm from 2 i) on the vertex. Empty the returned queue of vertices into the cloud, this ensures that the algorithm will skip them during its run and the vertices won't be considered.
- If one of those vertices was our start or destination vertex, return a predetermined value to signal that we won't be able to get to our destination.
- If our start vertex is our destination vertex then return a distance of 0.
- *Recursion*:
- Put the current vertex we're looking at in the cloud, ensuring we don't visit it again.
- For each iteration through the outgoing edges only, (we just care about our path **to** the destination):
- If the edge is a walking edge, consider it's weight the specified constant.
- Add the weight of the edge to the previous minimum, we will consider this shortest path value
- If the vertex isn't in our cloud and if it's either not yet in our discovery map, or if it's there but with a larger value:
    - Add the vertex to our discovery map with the new shortest distance as its value
    - Add the shortest value to the priority queue with the discovery edge as its value

- Obtain the minimum from our priority queue with its associated edge
- Push the edge onto a stack to consider later, it may be part of our path
- Identify the destination vertex of the edge, if its the destination vertex for our whole algorithm then return the minimum.
- If not, repeat recursion with the new vertex and minimum value
- *After recursion:*
- Empty the edge stack into a new stack D.s with the following logic
  - We always want the first edge, it's the last edge to our destination. Pop it and push it into the next stack saving the source vertex
  - Keep popping edges until we encounter the edge with our saved vertex as the destination
  - Save its source vertex and push the edge onto our stack s
  - Repeat this until the edges stack is empty

When we empty out stack D.s afterwards, we receive in order the vetices forming the path from source to destination, excluding the third vertex's path if specified. We can determine the distance a number of ways, in this case it is the last min value returned by our recursive method

Sample Outputs - obtained using openjdk-8-jre (the open-source variant to Oracle JRE 8)

java ParisMetro 1 1
You're already there!

java ParisMetro 28 193 199
Is is impossible to reach your destination

java ParisMetro 28 192
28 Bobigny, Pablo Picasso
29 Bobigny-Pantin, Raymond Queneau
374 Église de Pantin
134 Hoche
270 Porte de Pantin
226 Ourcq
160 Laumière
143 Jaurès
340 Stalingrad
125 Gare du Nord
124 Gare du Nord
14 Barbès Rochechouart
64 Château Rouge
192 Marcadet Poissonniers
With a travel time of 705 seconds
which is 11 minutes and 45 seconds.

java ParisMetro 192 28
192 Marcadet Poissonniers
64 Château Rouge
14 Barbès Rochechouart
124 Gare du Nord

125 Gare du Nord
340 Stalingrad
143 Jaurès
160 Laumière
226 Ourcq
270 Porte de Pantin
134 Hoche
374 Église de Pantin
29 Bobigny-Pantin, Raymond Queneau
28 Bobigny, Pablo Picasso
With a travel time of 705 seconds
which is 11 minutes and 45 seconds.

java ParisMetro 148 37
148 Jussieu
47 Cardinal Lemoine
195 Maubert Mutualité
74 Cluny, La Sorbonne
221 Odéon
174 Mabillon
346 Sèvres Babylone
358 Vaneau
99 Duroc
349 Ségur
154 La Motte Picquet, Grenelle
11 Avenue Émile Zola
54 Charles Michels
145 Javel
373 Église d'Auteuil
196 Michel Ange Auteuil
259 Porte d'Auteuil
36 Boulogne, Jean Jaurès
37 Boulogne, Pont de Saint-Cloud, Rond Point Rhin et Danube
With a travel time of 823 seconds
which is 13 minutes and 43 seconds.

java ParisMetro 37 148
37 Boulogne, Pont de Saint-Cloud, Rond Point Rhin et Danube
36 Boulogne, Jean Jaurès
198 Michel Ange Molitor
52 Chardon Lagâche
201 Mirabeau
145 Javel
54 Charles Michels
11 Avenue Émile Zola
154 La Motte Picquet, Grenelle
349 Ségur
99 Duroc
358 Vaneau

346 Sèvres Babylone
174 Mabillon
221 Odéon
74 Cluny, La Sorbonne
195 Maubert Mutualité
47 Cardinal Lemoine
148 Jussieu
With a travel time of 813 seconds
which is 13 minutes and 33 seconds.
java ParisMetro 28 193
28 Bobigny, Pablo Picasso
29 Bobigny-Pantin, Raymond Queneau
374 Église de Pantin
134 Hoche
270 Porte de Pantin
226 Ourcq
160 Laumière
143 Jaurès
340 Stalingrad
125 Gare du Nord
122 Gare de l'Est
123 Gare de l'Est
250 Poissonnière
44 Cadet
162 Le Peletier
59 Chaussée d'Antin, La Fayette
60 Chaussée d'Antin, La Fayette
133 Havre Caumartin
317 Saint-Augustin
203 Miromesnil
332 Saint-Philippe du Roule
110 Franklin D. Roosevelt
2 Alma Marceau
139 Iéna
355 Trocadéro
306 Rue de la Pompe
157 La Muette
291 Ranelagh
141 Jasmin
197 Michel Ange Auteuil
199 Michel Ange Molitor
104 Exelmans
271 Porte de Saint-Cloud
193 Marcel Sembat
With a travel time of 1695 seconds
which is 28 minutes and 15 seconds.

java ParisMetro 44
250: Poissonnière

123: Gare de l'Est
63: Château Landon
169: Louis Blanc
341: Stalingrad
300: Riquet
87: Crimée
82: Corentin-Cariou
277: Porte de la Villette
10: Aubervilliers-Pantin, Quatre Chemins
108: Fort d'Aubervilliers
152: La Courneuve, 8 Mai 1945
44: Cadet
162: Le Peletier
59: Chaussée d'Antin, La Fayette
224: Opéra
282: Pyramides
228: Palais Royal, Musée du Louvre
255: Pont-Neuf
71: Châtelet
254: Pont-Marie
345: Sully Morland
149: Jussieu
241: Place Monge
49: Censier Daubenton
164: Les Gobelins
244: Place d'Italie
352: Tolbiac
184: Maison Blanche
161: Le Kremlin-Bicêtre
364: Villejuif, Léo Lagrange
365: Villejuif, P. Vaillant Couturier
363: Villejuif, Louis Aragon
260: Porte d'Italie
266: Porte de Choisy
261: Porte d'Ivry
237: Pierre Curie
179: Mairie d'Ivry

java 28 193 44
28 Bobigny, Pablo Picasso
29 Bobigny-Pantin, Raymond Queneau
374 Église de Pantin
134 Hoche
270 Porte de Pantin
226 Ourcq
160 Laumière
143 Jaurès
340 Stalingrad
339 Stalingrad

151 La Chapelle
13 Barbès Rochechouart
5 Anvers
239 Pigalle
27 Blanche
246 Place de Clichy
302 Rome
366 Villiers
204 Monceau
85 Courcelles
351 Ternes
56 Charles de Gaulle, Étoile
57 Charles de Gaulle, Étoile
150 Kléber
30 Boissière
354 Trocadéro
355 Trocadéro
306 Rue de la Pompe
157 La Muette
291 Ranelagh
141 Jasmin
197 Michel Ange Auteuil
199 Michel Ange Molitor
104 Exelmans
271 Porte de Saint-Cloud
193 Marcel Sembat
With a travel time of 1786 seconds
which is 29 minutes and 46 seconds.

Reference -

package: net.datastructures by Goodrich et al.
CSI 2110 Lab 9 by Jochen Lang