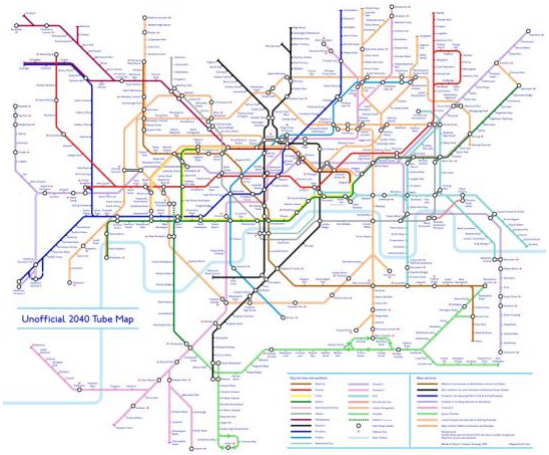
Algorithms and Data Structures

Searching for paths in a public transport map

Assignment-5

Version: 7 December 2019



READ THIS FIRST:

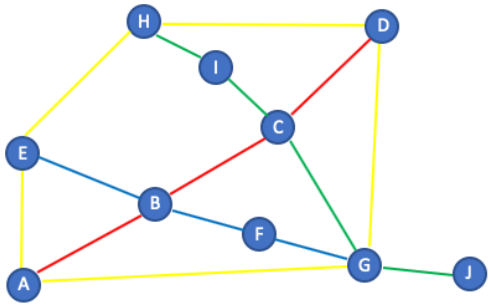
There are some minor changes in the first part compared to the previous version. These changes are described at the beginning of section A.

Introduction

In this assignment you will use different search algorithms in the context of a public transport system. These algorithms will try and find paths between stations using connections between stations. Stations are on lines, for instance metro lines and bus lines. You will build a small graph from a small transport map, that shows lines, station and connections between the stations. You will implement code to build the graph data structure and code to use the search algorithms.

In the book of Robert Sedgewick and Kevin Wayne many examples of algorithms are shown. Your task is to use an adapt the code within the context of the transport map.

A simple example of such a map is shown below.



We will define the information of this map by defining the following arrays of Strings:

```
String[] redLine = {"red", "metro", "A", "B", "C", "D"};

String[] blueLine = {"blue", "metro", "E", "B", "F", "G"};

String[] greenLine = {"green", "metro", "H", "I", "C", "G", "J"};

String[] yellowLine = {"yellow", "bus", "A", "E", "H", "D", "G", "A"};
```

As you can see a line is defined by it's name (in this case colours), by it's type (metro of bus), and by the stations listed in order.

Part of the assignment is to implement a TransportGraph and a Builder that extracts the information of the arrays and builds the graph, which contains vertices, the stations, and edges, called connections.

Specification of the Public Transport System

On the next page you find two UML class diagrams specifying the Public Transport System. For sake of simplicity, no visibility indicators are included for attributes or methods and Java Collection interface types are specified for relevant data types.

The public transport system runs metro trains and busses on lines from station to station.

Line, Station, Connection

A Line is uniquely identified by a name. It has a type (metro, bus) and it holds an ordered list of stations.

A Station is uniquely identified by a stationName. It holds a set of lines that run through the station.

A Connection is uniquely identified by the stations it connects, the 'from' station and the 'to' station. It is a directed connection. All connected stations are connected both ways. So, there are two connections connecting the stations. The connection holds a Line attribute and for future use it holds a weight. The weight will express the travel time in minutes as a double.

TransportGraph

The TransportGraph will have stations as it's vertices and connections as it's edges. The stations, vertices, will be stored in an ArrayList, so they are indexed.

For the sake of convenience a Map<String, Integer> holds the same information, but vice versa, so you can easily find the index of a Station by means of it's stationName. So for example, if we want to know the index of Station C in the list of stations, we can use the map to find the correct index. Note that this is redundant information! There is a method getIndexByStationName(String name) to address this.

Furthermore we will store an array of all the adjacency lists. For every station, we use it's index to find the connected stations in its' adjacency list. Note that we will store the indices of the connected stations.

See also the listing on page 526 of Sedgewick/Wayne and compare this to the class diagram at the end of this document.

Again for the sake of convenience, the TransportGraph has a 2D-array to find a Connection objects more easily by using the indices of the connected station. Say, we have a connection from station C to G and C and G have indices 3 and 6 resectively. Then the 2D-array stores the Connection(C, G) at Connection[3, 6] and we can use a method getConnection(int from, int to) get the proper connection by using the array.

The graph has a numberOfConnections attribute. We will count both connections between two stations as one. So the numberOfConnections matches the number of coloured 'lines' in the picture of the graph.

Builder

To build the TransportGraph there is a Builder inner class. This class has an addLine() method that uses a line information String array to add the a Line object to a list of lines and add all the stations of the line to the list of stations in the Line object.

Then you can use the buildStationSet() method to make a Set of stations. Using a set ensures that stations will be added only once as a vertex in the list of Stations of the Graph. You will have to loop through all lines and for every line loop through the stations on the line to make the set of stations complete.

After building the set of stations, you add lines to the stations. More than one line can run through a station. The method addLinesToStations() adds the correct lines to all stations.

Next you will have to use a buildConnections() method that uses the ordered list of stations in a line to make connection object of consecutive stations. Again loop through all the lines and it's stations to make the set of connections complete.

Finally the build() method uses the set of stations and the set of connections to add all vertices and edges to the graph. Of course after initializing the Graph object with the proper size.

Use of the Builder

In the main method you will define the lines by it's line definitions. Then you will have to use a Builder object to do the following in the specified order:

- Call addLine() for all the lines
- Call buildStationSet()
- Call addLinesToStations()
- Call buildConnections()
- Make an TransportGraph object by calling the build() method of the Builder object.

Algorithms

Now the data structures are in place, we will focus on the algorithms. Following the approach of Sedgewick/Wayne there will be classes for every graph-processing algorithm, see page 528.

We will start by implementing DepthFirstPath (page 536) and BreadthFirstPath (page 540). Because these classes share common methods and attributes there is an AbstractPathSearch class from which the two classes will inherit. Take a look at the class diagram and see that the approach here is slightly different from the approach of Sedgewick and Wayne.

The algorithm classes will hold a reference to the graph. They will keep track of the order of the nodes visited. They will try and find a path from a station to another station and they will keep track of transfers from one line to another. And they use a LinkedList of the station indices to build the path found and use that to build a List of stations in the path found. So you have to adopt this approach and make necessary changes to the listings on page 536 and 540.

Section A.

Changes

There are some changes made to this version compared to the first version. Below you find the changes, which you can apply to your own code, so it is compliant to this version. Or you copy-paste your code to the unzipped new version.

In the class **`TransportGraph`** the following changes were made:

- The signature of the `stationIndices` attribute changed from `Map<Station, Integer>` to `Map<String, Integer>`.
- The following method is added:

```
public int getIndexOfStationByName(String stationName) {
    return stationIndices.get(stationname);
}
```

In the class **`AbstractPathSearch`** the following changes were made:

- The constructor now expects station names instead of indices, so the first three lines of the constructor should look like:

```
public AbstractPathSearch(TransportGraph graph, String start, String end) {
    startIndex = graph.getStationIndexByName(start);
    endIndex = graph.getStationIndexByName(end);
```

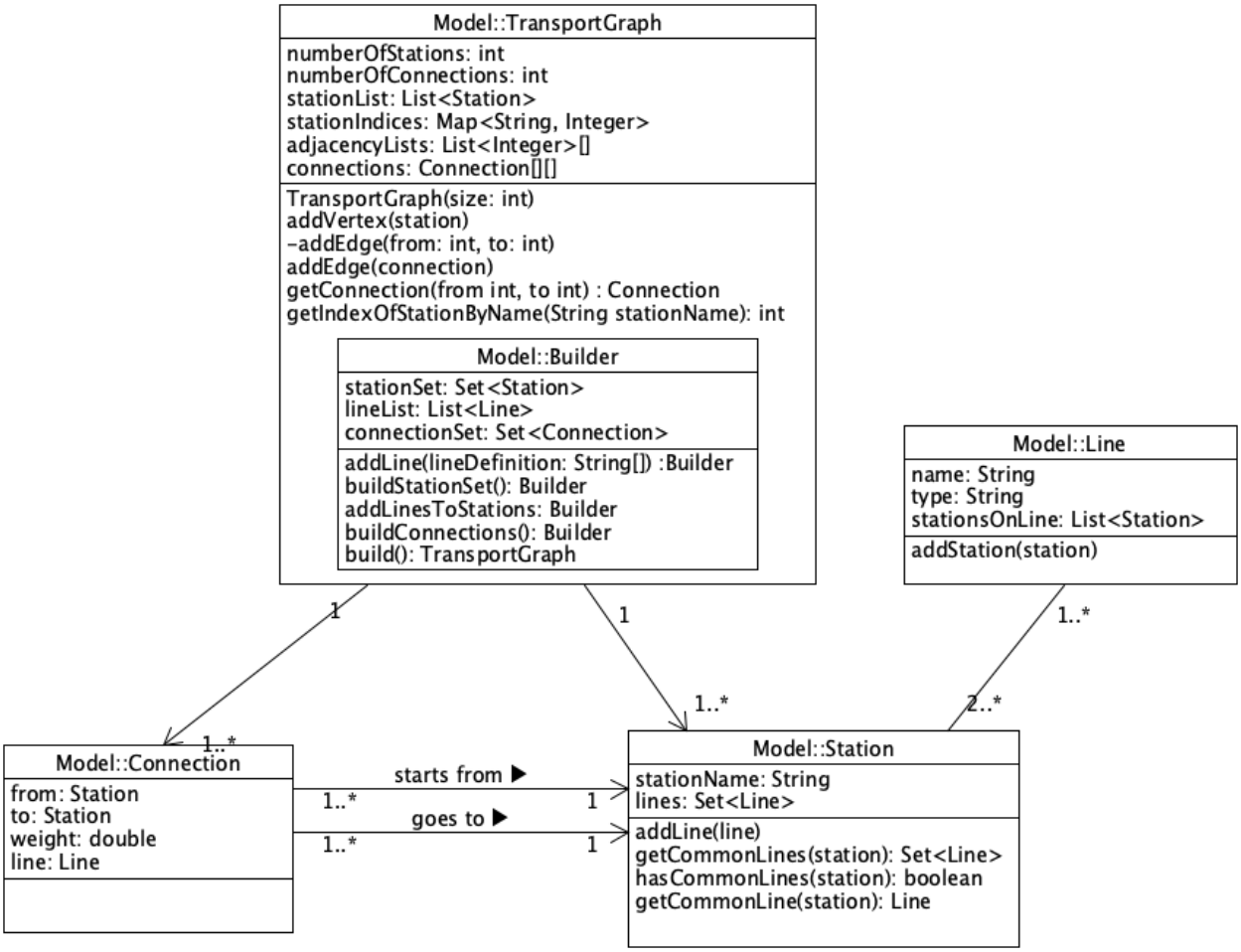
In the class **`TransportGraphLauncher`** the following changes were made:

- The construction for the dfs uses station names: `new DepthFirstPath("A", "J");`
- The construction for the bfs uses station names: `new BreadthFirstPath("A", "J");`

There is a new class **`IndexMinPQ`** in the package **`model`**, the source code for it can be found on the DLO.

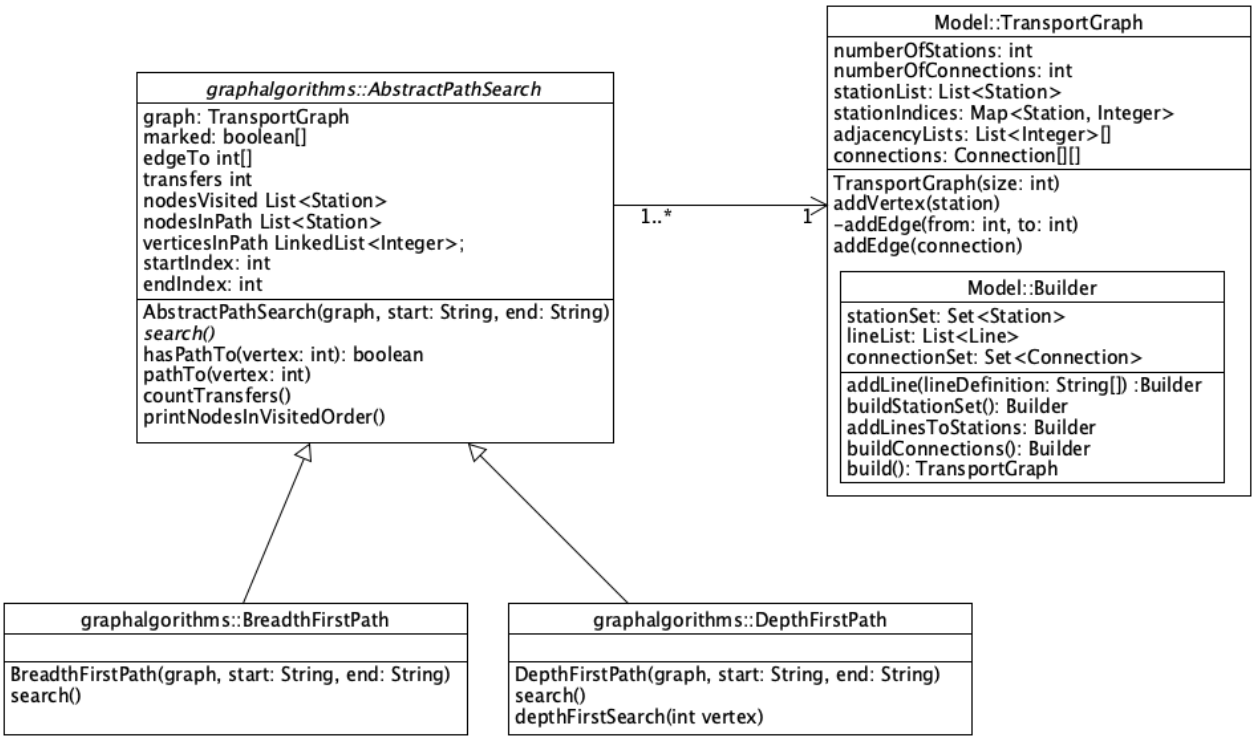
Assignment A

1. In the first part of this assignment you will have to implement the methods of the Builder class and part of the methods of the TransportGraph itself. See the To Do's in the code, the JavaDoc comments above the methods and the class diagram below.



You can print the information of the graph. See the toString() method of the TransportGraph.

```
Graph with 10 vertices and 15 edges:
A: B-E-G
B: A-C-E-F
C: B-D-G-I
D: C-G-H
E: A-B-H
F: B-G
G: A-C-D-F-J
H: D-E-I
I: C-H
J: G
```



2. Next you will have to implement several methods in the AbstractPathSearch class. Note that this class has an abstract method search(). In the subclasses the search() method will implement the actual algorithm. Also note that the constructor of all the different graph algorithm classes use start and end parameters with datatype String and not int. This is convenient for the use in client classes. The client can use the name of the Station and does not need to know the index of the vertex in the underlying graph. Make use of the getIndexByStationName(String name) method to set the startIndex and endIndex as integers.

3. Now make the BreadthFirstPath class. You can adjust the code on 540 of Sedgewick/Wayne to match the class diagram. Main differences are:
    a. As you can see in the constructor you can also set the end vertex of the search algorithm.
    b. The end vertex should be used in the search() method. As soon as the end vertex is reached, the pathTo() method should be called to build the path that connects the start vertex with the end vertex. And also the nodesVisited list has to be built in the search() method.
    c. The constructor should not call the search() method. The search() method should implement the invocation of the actual method. So, in the main, after initializing a BreadthFirstPath object, the search() method must be called to start the algorithm.

4. Then you make the DepthFirstPath class. You have to adjust the code on page 536. In this case however the search() method should call a private helper method that is recursive (as you have seen before). The recursive method resembles the code on page 536. Again the end vertex should be used and the nodesVisited list has to be built.

Using the search classes you should get a result as follows:

```
Result of DepthFirstSearch:
Path from E to J: [E, A, B, C, D, G, J] with 3 transfers
Nodes in visited order: E A B C D G F J H I

Result of BreadthFirstSearch:
Path from E to J: [E, A, G, J] with 1 transfers
Nodes in visited order: E A B H G C F D I J
```

5. Make an overview of all the path from all stations to all other stations with the least connections.

Assignment B.

In this assignment you will use another transport map, showing stations on a virtual grid of 15 by 15 squares. Each station has a location with two coordinates in the range of 0 to 14.

The locations can be used to find the estimated travel time when riding in a straight line from station to station. This will be used later on in the A* algorithm.

Furthermore the map shows the actual travel time in minutes from station to station. Minutes are given in decimals. You do not need to work with seconds.

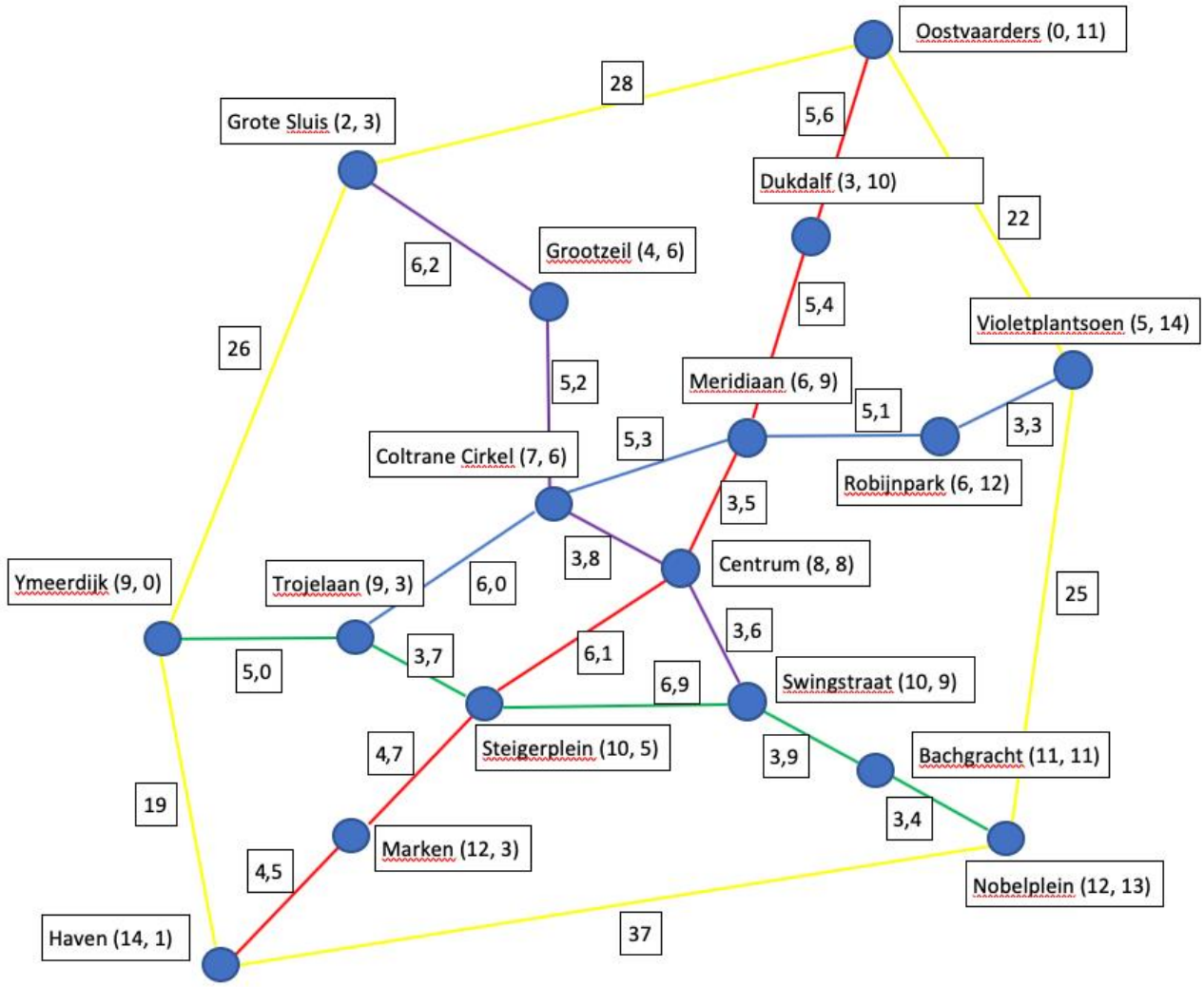
There are four metro lines and one bus line.

Metro lines:

- Red: Haven – Marken – Steigerplein – Centrum – Meridiaan – Dukdalf – Oostvaarders
- Blue: Trojelaan – Coltrane Cirkel – Meridiaan – Robijnpark – Violetplantsoen
- Purple: Grote Sluis – Grootzeil – Coltrane Cirkel – Centrum – Swingstraat
- Green: Ymeerdijk – Trojelaan – Steigerplein – Swingstraat – Bachgracht – Nobelplein

Bus Circle line:

- Yellow: Grote Sluis – Ymeerdijk – Haven – Nobelplein – Violetplantsoen – Oostvaarders – Grote Sluis



1. Build the graph similar to the way we used in assignment 1.

2. Test the DFS and BFS methods on the new graph.

3. Add a convenient method to the TransportGraph to add the weights of the connections to all the edges (connections) in the graph. Hint: use an array of doubles in the main method with all weights of a line. Make sure the method sets the weight of the correct connection.

4. Implement the class DijkstraShortesPath() that contains a search() method to find the path from one station to another with the shortest travel time. You may use the code in Sedgewick/Wayne.
    a. You need an attribute distTo, an array of doubles.
    b. You need to initialize the values of the array distTo in the constructor.
    c. You need a Priority Queue. The start project offers the IndexMinPQ used in the book.
    d. Note that you need to override the hasPathTo() method.
    e. You need a method getTotalWeight() to find the weight of a path.

5. Test your algorithm in the main method.

When a traveler has to change lines, make a transfer, this will take time. Use the following rules:

- Changing from metro line to another metro line will cost 6 minutes.
- Changing from metro line to bus line and vice versa, will cost 3 minutes.

6. Add an attribute edgeToType with datatype Line[]. So this is an array of lines. In the search algorithms the edgeTo attribute holds the 'from' index of the connection to a certain index, when the vertex is in the path. So this holds the connection. In this new attribute you will hold the Line information of that connection.

7. Implement a getTransferPenalty(int from, int to). You can use the edgeToType array to find the Line that is used to get to the from vertex and check whether the (from, int) connection gives a difference in line and what transfer penalty should be applied.

8. Adjust the search() algorithm in order to use the transfer penalty to find the path with shortest travel time.

Assignment C.

To use an heuristic in the A* algorithm you have to find a heuristic function.

1. Add a location class to your project. The location class should have a method that finds the estimated travel time from one location to another location.

2. The estimated travel time for one square is 1.5 minutes. So from location (3, 5) to (3, 6) will take 1.5 minutes. Implement a travelTime() method in the Location class to calculate the travel time form one location to another.

3. Add a convenient method to the TransportGraph to add the locations of the stations to all the vertices (stations) in the graph. Hint: use an array of integers in the main method that lists all the coordinates of the stations of one line in order of the stations of that line.

4. Implement the class A_Star, with a search() method that uses the heuristic to find the shortest path from one station to another.

5. Test the search() in the main method.

6. Make an overview of all the shortest paths from all stations to all other stations. Use both Dijkstra and A*. Show that the results are the same, but that A* is more efficient.

Grading criteria.

Sufficient: You have to implement BFS, DFS, Dijkstra and A* without transfer information and transfer penalties.

Good: You have implemented BFS, DFS, Dijkstra and A* with transfer information and transfer penalties.

Excellent: You have improved the way the graph is built. The information of the Transport map may be stored differently so you can build the complete graph, including weight and location, in a very efficient and straightforward way.

Of course you should hand in a report showing your work and results.