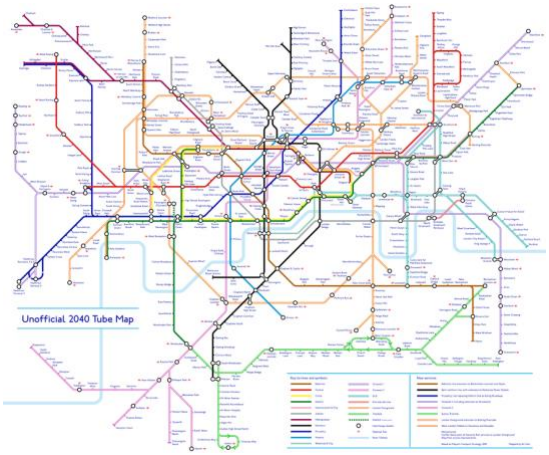
# Algorithms and Data Structures

*Searching for paths in a public transport map*

*Assignment-5*

*Version: 30 November 2019*

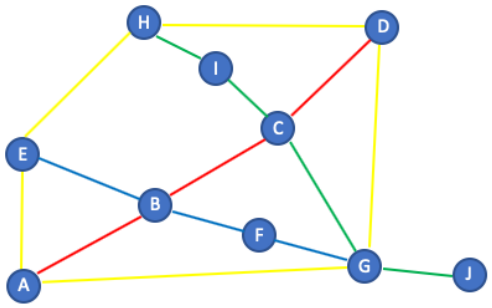



## Introduction

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<Station, Integer> holds the same information, but vice versa, so you can easily find the index of a Station. 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!

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

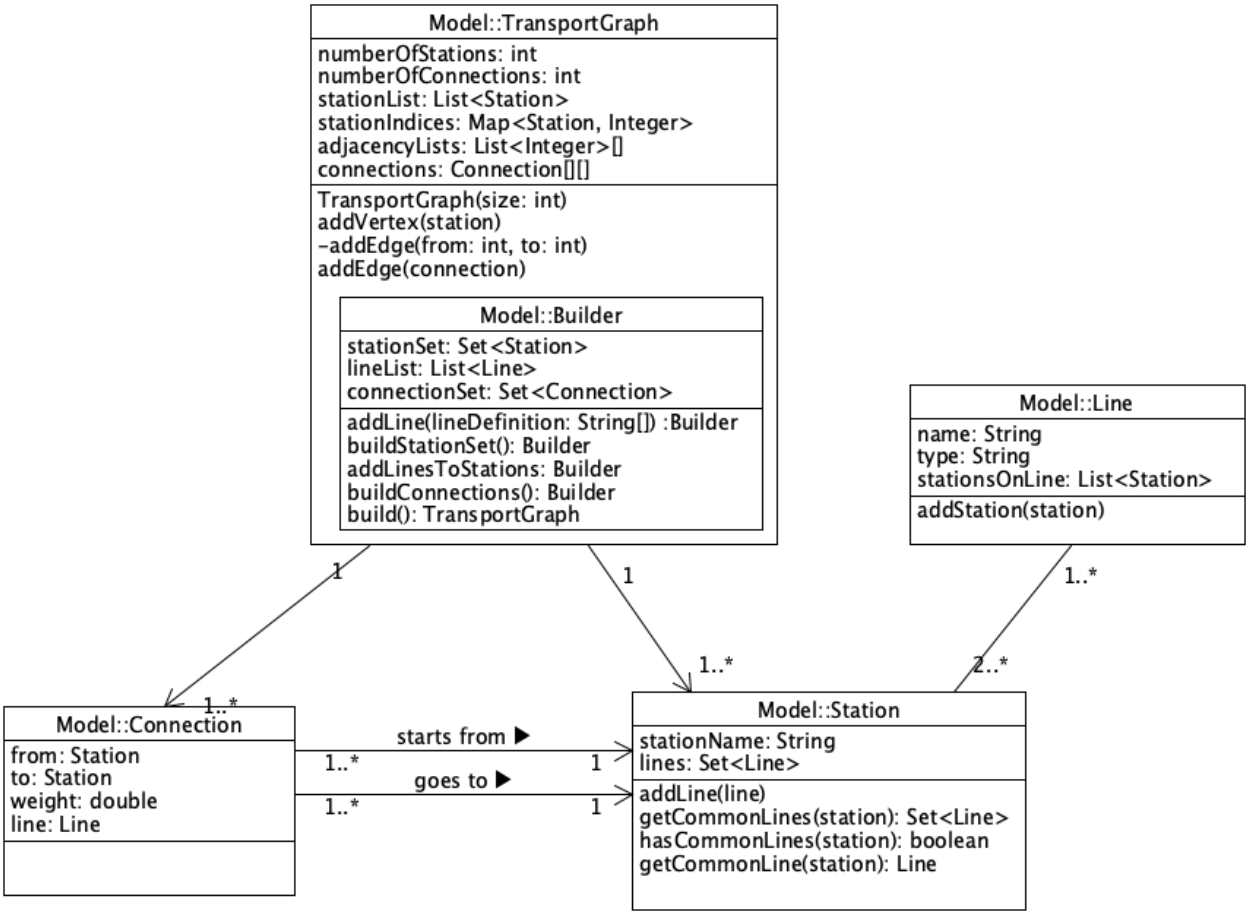
Now the data structures are in place, we will focus on the algorithms. Following the approach of Sedgewick/Wayne there will be classes for every graph-processing algorithm, see page 528.

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.

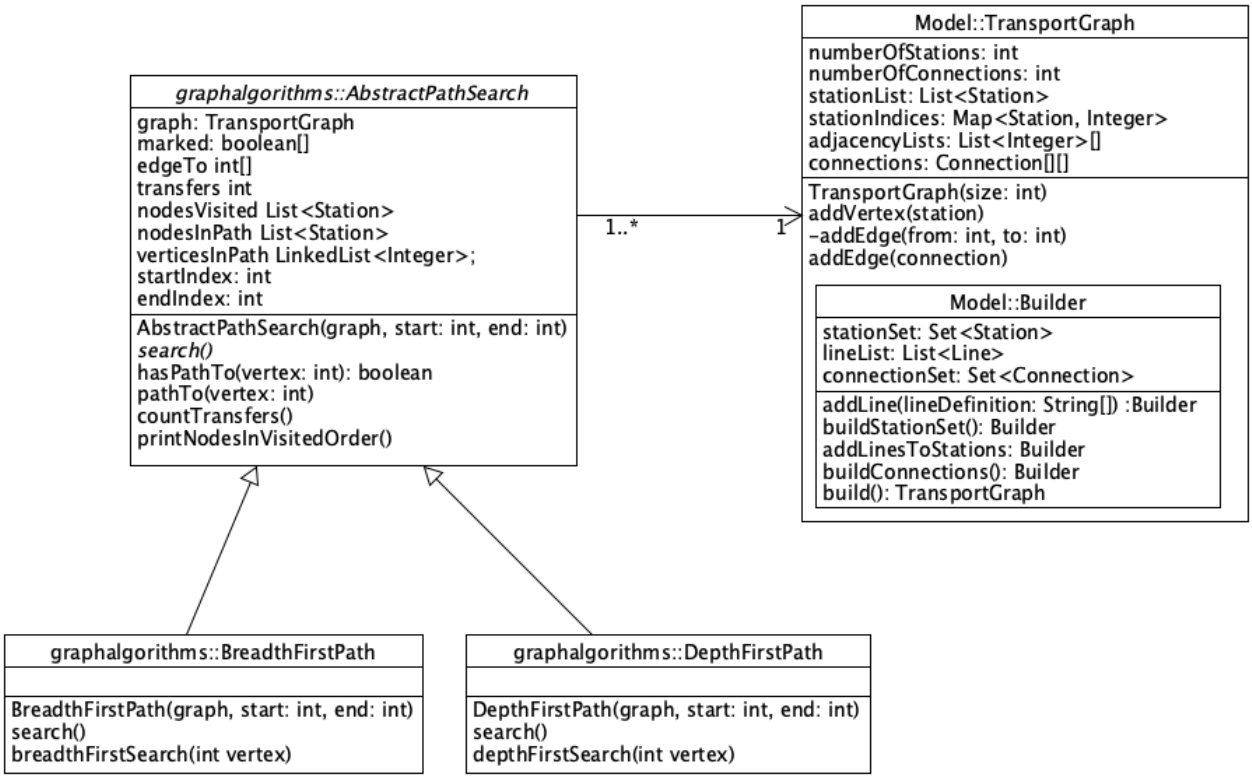
## Assignment A.

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().
3. Then you make the DepthFirstPath class. You have to adjust the code on page 536 of Sedgewick/Wayne to match the class diagram above. Main differences:
    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 depthFirstSearch() 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 depthSearch() method.
    c. The constructor should not call the depthFirstSearch() method. The search() method should implement the invocation of the actual method. So, in the main, after initializing a DepthFirstPath object, the search() method must be called to start the algorithm.
4. Finally make the BreadthFirstPath class. The same adjustments will have to be made as in the DepthFirstPath class and depthFirstSearch() method.

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
```