CS308: BostonMetro System Analysis

Group 27: Benjamin Garside (201730178), Nicole Haggerty (201708009), Robbie Wilson (201707872), Tony Sharkey (201720351), Koni Watson (201710203)

Brief Overview-

Our implementation of a MultiGraph consists of Nodes connected by undirected weighted Edges, both classes of their own right. The Multigraph class stores the Nodes in a HashMap using the nodeID as the key and a second HashMap stores the Edges spanning from each Node: data types chosen to improve efficiency when searching.

The MultiGraph remains completely decoupled from the BostonMetro system but to obtain a path with a minimal number of line changes a second search function findAllPaths() was added to the MultiGraph. This function returns multiple paths that are not necessarily optimal in terms of weight but allows the line changing minimisation to be handled within the BostonMetro class itself and thus keeping the MultiGraph implementation completely general for use in other applications.

Weights were assigned to Edges based on the line they were on and were thought of as cost in pence per station of travel. As such lines with large distances between Edges are assigned larger weights and vice versa. Users are given a choice after input to minimise line changing or cost.

Interface Roles-

The interfaces *NodeADT*, *EdgeADT* and *GraphADT* provide a way to specify the methods which classes that implement them must include. All the interfaces are made to be general with the aim of decoupling the Boston Metro from the *MultiGraph* and thus making the code reusable for any program that requires an implementation using a graph.
Classes in the UML diagram always instantiate *Node*s, *Edge*s and the *MultiGraph* as their respective interface to introduce a layer of abstraction and open up the possibility for polymorphism (particularly with respect to the *findPath* method which would likely change in different graph implementations).

Class Roles-

Node: Data object allowing a node (or in the case of the Boston Metro, a station) to be easily represented to make up a graph.

Edge: Similarly, a data object allowing an edge (or in the case of the Boston Metro, a connecting metro line) to easily represent connections between nodes on a graph.

MultiGraph: Composed of nodes and edges, used to complete the representation of a graph and implementing all the methods defined by the *GraphADT* interface.

BostonMetro: Essentially the 'Driver' class using an instantiation of the *GraphADT* to implement all the functionality that's specific to the Boston Metro route finder.

MetroMapParser: Used to read in the Boston Metro map from a file.

Relationships-

Association: Used to indicate that classes need to know about each other. In our case all associated relationships are directed, indicated by the open arrowhead: the class in which the arrow is pointing to is the target class which will contain an instance of the source class.

Interface Implementation: Indicated by the dotted arrows, this is a relationship between a classifier and a provided interface, where the classifier will obey the structure of the given interface.

Composition: This relationship is thought of as a "part of" relationship: without the whole the parts would not exist. For both these connections the *MultiGraph* is the "whole" composed of *Nodes* and *Edges*.

Searching-

The search algorithm used in our implementation will be Dijkstra's algorithm which finds the shortest path between nodes on a weighted MultiGraph. This will be implemented with a priority queue and a custom comparator to expand nodes into an agenda and remove the shortest weighted path from the queue until the goal node is reached.

Method Descriptions-

Java Class	Method Name	Description
NodeADT	getLable()	Will return the given lable (String) of a given Node.
	getID()	Allows for the unique ID number given to a Node to be retrieved.
EdgeADT	getLabel()	Will return the given label i.e. line colour (String) of a given Edge.
	getNodeA ()	Returns the node that is attached to one end of the edge
	getNodeB ()	Returns the other node that is attached to the other end of the edge
	getWeight()	As each line is given a specific weight, this will allow this weight to be retrieved.
GraphADT	addNode(NodeADT node)	This adds a given node to a HashMap of all current nodes.
	getNode(int id)	Retrieves and returns the node with the passed in unique ID.
	addEdge(NodeADT node, EdgeADT edge)	Allows for a new Edge to be added between two Nodes on the graph
	findPath(NodeADT nodeA, NodeADT nodeB)	Finds a path from nodeA to nodeB, returning the path as a list of Edges.
	getNodes()	Returns HashMap populated with all the nodes in the metrosystem
	isConnected(NodeADT nodeA, NodeADT nodeB	Returns true if the two specified Nodes are connected.
	getConnected(NodeADT node)	Returns a list of all edges spanning from the specified Node.
	findAllPaths(NodeADT nodeA NodeADT nodeB)	Provides all paths from nodeA to nodeB and returns them as a listed list of edges
	getNodes()	Allows for all the nodes to be accessed by returning a map containing all of them.
BostonMetro	readFile()	Uses MetroMapParser class to read map data in from file.
	getInput(I)	Used by the findRoute() method to get user input for the stations with error handling.
	getRoute()	Used to print out the route between the two stations input by user.
	getCharInput()	This method allows for the used to make decisions i.e. which St.PaulStreet station and cheapest or least switching path.
	findMinLineChanges(List<List<EdgeADT>>)	Provides the path (list of edges) that has the least line switches based off the list of paths that are passed into it.
MetroMapParser	generateGraphFromFile()	Reads in the text file of the Boston metro system and parses the data into nodes and edges and it stores them in a graph form.
	assignEdgeWeight(String line)	Method which assigns the weightings to the right coloured lines (edges).

Changes Made-

Throughout this project multiple changes have occurred. These changes being due to feedback received from a TA and functionality purposes.

One thing that changed often was our UML and the relationships between classes and the methods within them. These changes were made upon the advisement of our TA

New methods were added throughout for functionality purposes. These methods not being a part of our original UML but were necessary for our system. For example findAllPaths() and findMinLineChanges() are two methods that were thought of/added later on to allow for the option for the user choice if they wish to travel the fastest route (i.e. minimum line changes)/cheapest (minimum weighted path).