Representation and Rationale

Our implementation includes three interfaces, MultiGraphADT, Edge, and Node. The MultiGraph class implements the MultiGraphADT, and the BostonMetro creates a new MultiGraph.

MultiGraph provides descriptions for the methods outlined in the MultiGraphADT interface. It has two additional methods responsible for finding the shortest path between two nodes, and getting the path between two nodes with line changes called findRoute and getPath respectively. Although this is somewhat against what a MultiGraph does conceptually, it reduced semantic coupling between the BostonMetro class, and MultiGraph.

As a group we decided that an ArrayList of Nodes and ArrayList of Edges would be the easiest, but not the most efficient way of implementing our MultiGraph. We could’ve used a HashMap representation if we had more time, but an ArrayList serves our purpose fine.

Station class implements Node, and Line implements Edge. The method signatures in the interfaces receive their descriptions from the classes that implement them. All Nodes and Edges are represented globally, and only actually modified in the addEdge and addNode methods.

The Parser class is responsible for parsing information from a text file. This information is then used to create the subsequent Line and Station objects that are added to the MultiGraph. This MultiGraph is then used by BostonMetro find the desired route from the provided source station to the provided destination station, which are given via user input.

We thought it would be best to have Nodes as an object with an ID and a Name, and Edges with a String label and two Nodes. This saved us time as well as helping abstract the concept of the MultiGraph further.

The usage of interfaces is justified by the fact that it provides a good level of abstraction. Alongside that it separates the BostonMetro from the MultiGraph. It means that the core functionality of the MultiGraph is retained even if the scenario or requirements are different for the application. It is for this same reason we have Station implement Node and Line implement Edge; Stations and Lines are essentially just nodes and edges in a multigraph so the same functionality can be applied, again with the occasional additional method as required. It also means that BostonMetro can send Stations to the MultiGraph it’s using, as station implements Node meaning Station is (like Node) an accepted parameter for MultiGraph’s methods.

Changes made to design based on feedback:

We changed Edge to use actual Nodes instead of ints for its representation of nodes.

Parser’s createMap method is now a void which takes in a MultiGraph object and has it filled with nodes and edges. This reduced coupling between the Parser and the BostonMetro class.

We removed the nNodes and nEdges methods as they were not used. The return type of addEdge and addNode were changed to void as their previous boolean return type were not used. We also reduced the number of global variables in the MultiGraph and BostonMetro classes.

1. Interfaces:

MultiGraphADT- This interface contains method declarations that allow for the handling of data in the context of a MultiGraph.

Edge-The Edge interface contains method declarations for retrieving data for an edge between two Nodes.

Node-The Node interface represents a typical vertex in a graph, it only has two methods for retrieving its name and unique identifier.

Classes:

MultiGraph- Implements MultiGraphADT and provides implementation of each function from the interface. It creates a collection of edges and nodes, additional methods are also provided to aid the functionality for the searching and creation of routes. This is used by BostonMetro.

BostonMetro- Creates a new MultiGraph using the Parser createMap (see below) method, this class also contains all the console interaction for a user to find a route between stations. BostonMetro gives two Stations to MultiGraph to find a route between. BostonMetro then gets returned a list of directions as a String, and prints them to the user.

Line- This class implements Edge, taking in the name of the line, the source vertex and the destination vertex. BostonMetro sends Lines to the MultiGraph it uses, as a Line implements Edge, and, as such, is an accepted parameter for its methods (e.g addEdge). Line has an overridden hashCode and equals method.

Station-Implements Node. A station has the exact same information “blueprint” as a Node. Station also has an overridden equals and hashCode method, use for future changes involving hashmaps. BostonMetro also sends Stations to the MultiGraph it uses, as a Station implements Node, and, as such, is also an accepted parameter for its methods (e.g addNode). Node has an Overridden hashCode and Equals method for usage in HashMaps.

Parser- Parses information from a given file about stations and lines. BostonMetro uses this to fill a MultiGraph which is used later on for when a user wants to find routes. The classes main createMap method also checks for errors in the grammar of the file.

b) One relationship is that BostonMetro uses a MultiGraph which implements MultiGraphADT, as the BostonMetro and MultiGraph have the same concept of nodes and edges (in the context of the BostonMetro; stations and lines).

Line also implements Edge, as they contain the same components but Line in the context of subway lines. This is the same case for Station implementing Node, the stations of the Subway being like the vertices of a graph. Line and Station have some extra overridden hashCode and equals methods, for testing and future implementation.

c)

MultiGraph:

addEdge- Adds the given edge to the list of edges.

addNode- Adds the given node to the list of nodes

isEdge- Verifies whether or not two nodes exist in an edge in the list

getNeighbours- Returns a list of nodes that are neighbours to a supplied node.

getNodeByName- Returns a node with a matching name.

getNodeById- Returns a node with that ID. Used for when we need to get a St.PaulStreet

findRoute- Finds the route between two different nodes, returns list of nodes indicating the shortest path

getPath- Given a route, returns directions including when to change lies

BostonMetro:

run- runs program

printHelp- prints instructions for user

printDirection- Prints list of directions

handleStPaul- deals with event of user entering StPaul as an origin or destination station, asks for clarification between the two

Line:

getLabel- Returns the colour of the line

getSource- Returns the station that a train is coming from.

getDestination- returns the station that the train will be going to next.

hashCode- Used so that a HashMap (or any method which uses hashing) correctly knows how to compare two values to another

equals- Compares 2 to check if they are equal.

toString- converts to string

Station:

getName – Returns the stations name/label.

getId – Returns the stations ID.

hashCode- Used so that a HashMap correctly knows how to map values to another

toString- converts to string

equals- Compares 2 to check if they are equal.

Parser:

createMap- Reads a file of lines and stations to fill a MultiGraph which is passed as a parameter.