CS308 Boston Metro System

Our implementation attempts to keep the Multigraph separate from the Boston Metro system. The multigraph itself consists of three interfaces, one for the edges, one for the nodes and one for the graph itself. The edge interface only allows you to get the label and the nodes either side of the edge without seeing the information about the edge class itself, with its implementation in the MGEdge class. The node interface allows you to get the ID and label of the node without seeing information about the node class itself, with its implementation in the MGNode class. The graph interface allows for adding nodes and edges, getting nodes from labels and ID’s, finding the path to a node based on its ID and getting a nodes neighbours. Its implementation is in the Multigraph class. These three interfaces can be seen on the diagram, connected to their implementations through dashed lines.

This use of interfaces as specifications is important as the Multigraph class creates lists of Nodes and Edges with the node and edge interfaces, and with the specifications their implementations are hidden. Similarly, The Boston metro system itself uses the Multigraph Interface to create and populate the graph and also to search it. This is a good level of abstraction as the Boston Metro doesn’t know about how the multigraph is implemented, it just uses its methods. This successfully decouples the Multigraph itself from the Boston Metro system.

The Boston Metro class operates by creating a graph using the graph interface and then using the Map Parser to load the data from the file in the read method. This can be seen through their connection on the diagram. It then uses the graph interface to add all the nodes and then all the edges with the information parsed with the Map Parser with the addAllEdges and addAllNodes methods. With the multigraph created using the data, the init method uses the terminal to request input, call the search of the graph and print informative output accordingly. The getStation method is used here to receive input and check for nodes with the same label and handle errors.

This diagram and design is different to our initial design due to the feedback we received. Initially, our edge class simply contained a label, while the node class only contained a list of another list of edges. We realised that having the edge class hold two node neighbours would make the searching algorithm easier to implement. We previously planned to use the array index as the node ID but soon realised this was memory inefficient as it involved more looping through lists. The multigraph was changed to have a separate list of nodes and edges instead of a list of nodes (which was in turn a list of edges). This made searching and populating far easier as two separate lists are easier to traverse. A method to set the number of nodes was removed from the node interface and a number of nodes field removed from the graph interface as the new design did not require them. The set number of nodes method was also quite complex due to our design of an array list of an array list, removing the method made the edge and node classes a lot more concise.

The searching algorithm for finding the shortest path between two given nodes was also modified in its planned implementation as well. It still uses the Breadth-First search, as this method, by its nature finds the shortest path. By looking at every node at each depth before probing deeper into the graph, the destination will be found at the point closest to the root, thus finding the shortest path. The algorithm had previously been planned to have been written using stacks to maintain a current path searched, but this architecture was replaced by the backtracking map and a queue of nodes to test (which used first in-first out to stay within appropriate depths). This proved successful as the search works well and provides the intended functionality of finding the shortest path between two given nodes.