Aidan Bonnar

Daniela Filipova

Liam Joyce

Filip Lejhanec

Heather McLean

# CS308 Project Design

# Multigraph Representation

Our implementation of multigraph, **AdjacencyMapMultiGraph** implements the MultiGraph interface and does not provide any additional functionality than the one specified in the interface. It uses Java generics to allow user to input any type of objects as nodes and edges, as long as the edge implements LabeledEdge interface. The nodes are stored as keys in a Map and are mapped to Lists of edges adjacent to them.

**LabeledEdge** has three fields, namely label, nodeIn and nodeOut. Label is used in findPath method to prioritize paths with same labels and nodeIn and nodeOut store references to adjacent nodes.

# Class Descriptions

The **Metro**, **Controller** and **Display** represent a simplified MVC pattern.

* **Display** is the input/output mechanism, reading input specifically from the user and displaying output to the terminal on request by Controller.
* **Controller** is the core menu loop, processing the instructions from the user and responding by giving commands to the output and requesting data from Metro.
* **Metro** represents the model of the metro system. It can load the graph data from a file through the parser into the multigraph. It also takes requests from the controller to provide information on routes from one destination to another.

The classes used to make objects which represent the data in the multigraph are **Route** and **Station**.

* An instance of the **Route** class represents one line on the Boston metro system.
* An instance of the **Station** class represents one specific station in the Boston metro system.

**LabeledEdge** is the interface from which Route extends. Using this interface allows for interacting with the Route objects within the multigraph while also maintaining the decoupling of the Metro and Multigraph systems.

There is also a **Parser** which is responsible for reading data from the external file provided, returning Station and Route objects to the Metro.

The **Multigraph** consists of the interface itself and the **AdjacencyMapMultiGraph** which extends from it.

* The **Multigraph** is the interface which provides a connection between the multigraph system and the metro system whilst keeping the two from knowing about each other. In doing this, decoupling of classes is achieved, making different parts of the system less dependent on each other.
* **AdjacencyMapMultiGraph** is the concrete multigraph class which stores all nodes and edges of the graph, and carries out the operations to determine a path from the initial station to the destination.

The program is run from the **Application** class which simply creates a Controller object

# Method Descriptions

Methods which were renamed have been annotated with one asterisk (\*)

All methods added since the initial design have been annotated with two asterisks (\*\*)

## AdjacencyMapMultiGraph:

* AdjacencyMapMultiGraph()- Constructor which creates the multigraph in the form of a HashMap
* addNode(N node)- adds a given node to the graph and returns true if successfully added.
* addEdge(E edge)- Takes in two nodes already in the graph and the edge between them. The method adds the edge to its associated nodes and returns true is successful.
* removeNode(N node)- Deletes a given node from the graph and return true if successful.
* removeEdge(E edge)- Deletes a given edge from the graph and returns true if successful.
* getNodes()- Returns a list of all nodes in the graph.
* getEdges()- Returns a list of all edges in the graph.
* containsEdge(E edge)- Checks if an edge already exists within the graph and returns true if it does.
* getPath(N source, N destination)- Returns the shortest route between two given nodes of the graph.
* getAdjacentNodes(N node)- Returns a list of all nodes adjacent to a given node \*\*

## Metro:

* Metro()- Creates a new MultiGraph on which the Boston Metro system will be modelled
* populate(String fileName)- Creates a new Parser using the given filename and populates the MultiGraph with the data from the Parser
* findPath(int choiceStart, int choiceEnd)- Retrieves the first and last stations of a journey along with the fastest route between them
* getStationsOfSameName(String name) – Returns a list of integers, which are ids of stations with the specified name\*\*
* getNearbyStations(int id) – Returns adjacent stations for the station of the specified id\*\*
* getStation(int id) – Returns a station object with the same id as the id passed in\*
* getStation(Route<Station> route1, Route<Station> route2) - Takes in two route objects and returns the station common to both. \*\*

Station:

* Station(int id, String name) - Constructor for Station class. Sets the name and ID of a station
* Rest of the methods are getters and setters needed

## Route:

* Route(String label) - Constructor for Route class
* Other methods are getters and setters for the fields in Route

Controller:

* Controller() - Constructor for Controller class. Sets up Display and Metro and begin the menu loop
* mainMenu() – Continuous loop of the outer menu, asking whether the user wants to search a route or exit
* clarifyStationChoice(List<Integer> stationIDs, String stationName) - checks for two stations with the same name\*

## Display:

* Display() – Constructor for Display class, creates a new scanner
* readLine() – Retreive the next line which the user inputs
* getMenuChoice(int limit) – Display method for the outer menu
* requestStation(String line)- Request a station from the user\*
* output(String s) – Called by the display to handle the outputs
* printPrompt()- Prints out a prompt for the user to type next to

## Parser:

* Parser(String fileName) – Creates the Parser for the file the data is going to be read from by the scanner
* parseFile(String fileName) – Reads through the file and populates a list of stations and routes to be sent to the multigraph\*\*
* getStations()- Returns the list of stations \*\*
* getRoutes() – Returns the list of routes\*\*

# Changes Made to Original Design

* Some methods have been renamed to better clarify their purpose
* Parser methods have been altered-only the parseFile() method is concerned with any parsing of the data. The getter methods in Parser are for returning the resulting lists after the parseFile() method has been used
* Methods getNearbyStations() and getAdjacentNodes() in Metro and Multigraph respectively were added to identify Stations next to a Station specified
* getStationsOfSameName() was added to find cases where two or more Stations had the same name