"Cash Me A Taxi"
Design Specifications Document
SFRWENG 2XB3 - L03 Group 7

Albert Zhou, zhouj103
Areeba Aziz, aziza11
David Bednar, bednad1
Dylan Smith, smithd35
Himanshu Aggarwal, aggarwah

Department of Computing and Software, McMaster University

April 12, 2020

Revisions

Member	Student Number	Role
Albert Zhou	400196651	GUI
Areeba Aziz	400070863	Data Structure Designer
David Bednar	400177661	Algorithms
Dylan Smith	001314410	Client
Himanshu Aggarwal	400200223	Testing

By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through SE-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes

Contributions

Name	**Role**	**Contributions**
Albert Zhou	GUI	GUI Report Structure GraphFinder Diagrams Requirements traceback
Areeba Aziz	Data Structure Designer	GUI and Cmd-line Controllers GraphSearch File Writer Helper GraphFileController Report finalization
David Bednar	Algorithms	Sorters Models (Graph, Node, Edge)
Dylan Smith	Client	Parser
Himanshu Aggarwal	Testing	Test classes Trip ADT

Executive Summary

This project intends to provide a solution for the problem of taxi drivers in New York City struggling financially as modern ride-sharing apps like Uber currently dominate the market. The solution is to calculate the most optimal routes between NYC zones that earn drivers the most profit, based on 10 years' worth of taxi trips datasets provided by the City of New York. The datasets contain taxi trip data such as the distance travelled per trip, the amount of fare made, the tips earned, and the start/end location and times.

Our program first parses the intensive data that contains the taxi trip records and builds a directed graph with nodes representing a NYC taxi zone. The user then can select from various functions to run on the graph, such as search for the most optimal path between two given zones, or sort the edges (routes) by the given field (fare, tip, distance, etc). This project implements graph traversal, sorting and searching algorithms that were taught throughout the year.

Contents

1 Deployment Guide

1.1 Installation and Setup

1. Download the Eclipse project and open it on Eclipse.
2. To run the GUI Application, open the `GUI.java` file and then click on the Run button on Eclipse.
3. To run the Command-line Application, navigate to the `Main.java` file. From the top menu on Eclipse, go to Run → Run configurations → Arguments. Enter your Program arguments which must match the format specified in the command-line interface specifications.
4. To run automated tests, navigate to the `AllTests.java` file, and then click the Run button on Eclipse.

1.2 Prerequisite Knowledge

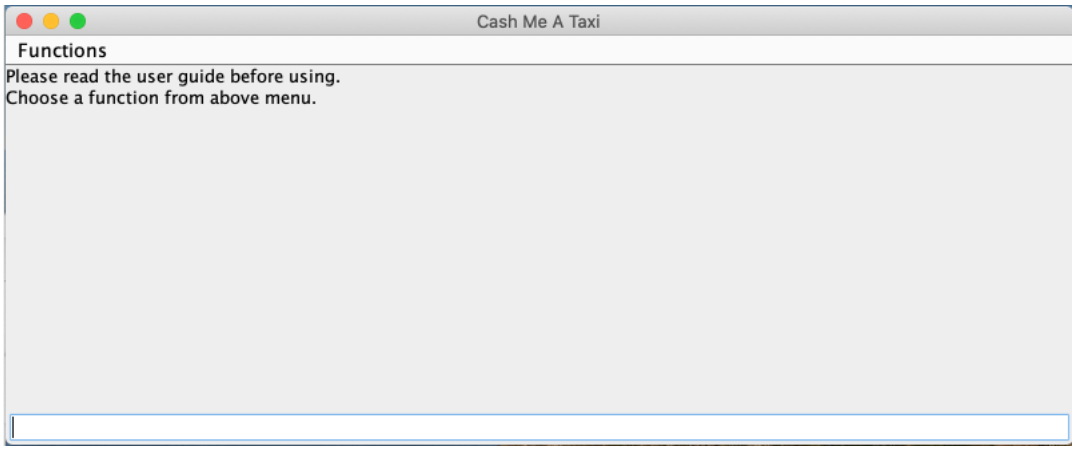
In this section we will outline some basic information that you'll need to know in order to use our application.

- This application uses datasets on taxi trip records provided by the City of New York. There are 10 years' worth of extensive data available to the public here: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.
- By default, there is 1 dataset that exists within the project, which is the 2019 Yellow Taxi Trip Records. This is the dataset that will be used by default. If you wish to use another dataset from the same website, you may download it and then enter that dataset's file location into the application to use that instead.
- A "zone" is a NYC taxi zone. Each zone has a unique integer zone ID that ranges from 1 to 263. More information about these zones can be found on the end of this webpage: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
- In our application, we build the graph while parsing the input dataset. The graph contains nodes that represent a NYC zone, and directed edges that represent the aggregate data of trips made between the source zone and the destination zone. Since the input dataset is very large, we save the graph that is constructed onto a csv file for future use. So, if you've run the application once and the graph has been constructed, and then later you want to re-run the application but use the same input dataset as before, instead of re-building the entire graph, you can simply specify the filepath of the graph data file that was generated at the previous application run. This could save a lot of time if you want to aggregate all 10 years' worth of data to run multiple functions on the application.

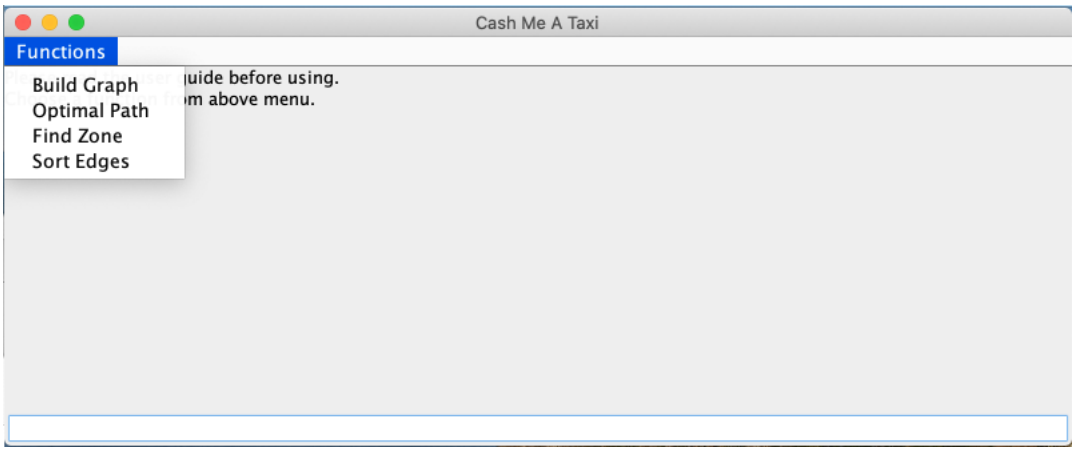
1.3 GUI Application

In this section we will provide a complete walk through of the Cash Me A Taxi GUI application.

1. When the app is launched you will be greeted with the following screen.



2. To begin using the app, click on "Functions". A drop-down menu of function options will appear.



3. Select the function you would like to use.

   · **Optimal Path:** Compute the most optimal path from a given start zone and end zone.

   · **Find Zone:** Given a zone ID, find the zone in the list of all zones and get its outgoing edges.

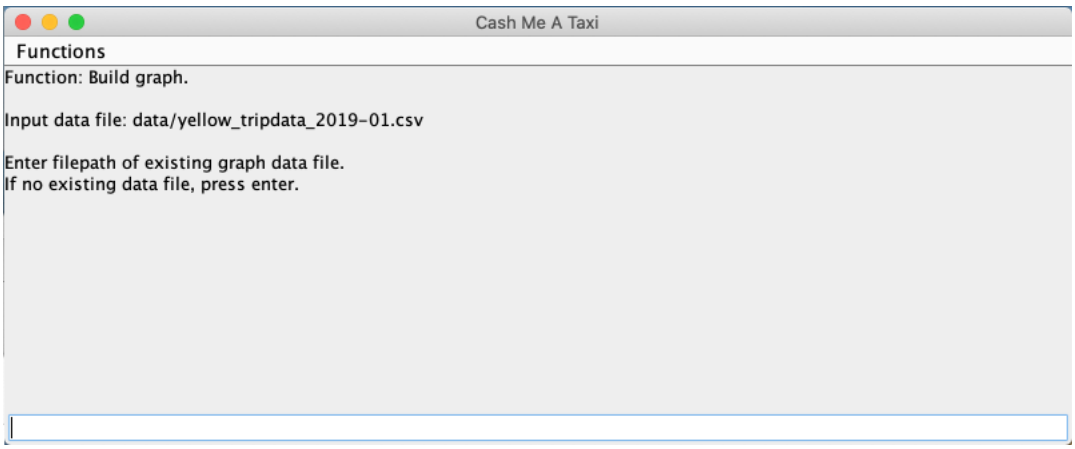
   · **Sort Edges:** Sort all the edges in the graph by a field such as average fare, average tips, etc.

   · **Build Graph:** Only build the graph and save its data onto a file.

   · Note that all functions will build the graph first and then compute the function specified. If you wish to change the data set you must restart the app.

1.3.1 Build Graph

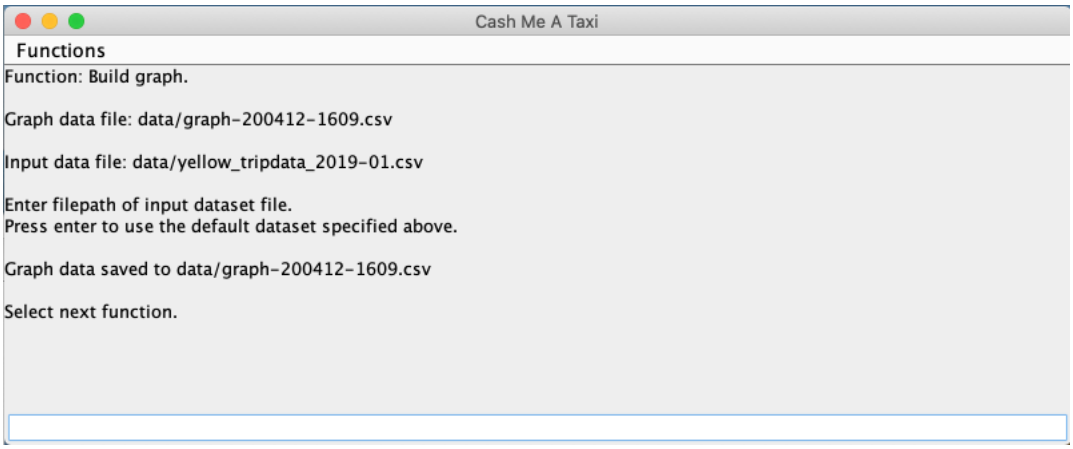
1. When "Build graph" is chosen you will be given the following options.



2. If you have already run the app, and have created a graph data file, you may enter the location of the file. If not, hit enter in the input textbox in the bottom of the screen to skip this step.

3. If you had not specified an existing graph data file, you will now be prompted to enter the file path of the input dataset you would like to use. If you want to use the default input dataset that is included with the application, simply hit enter to skip this step.

4. When the input file has been chosen, the app builds a graph and stores the edges in the file specified. Note this can take around 20 seconds.

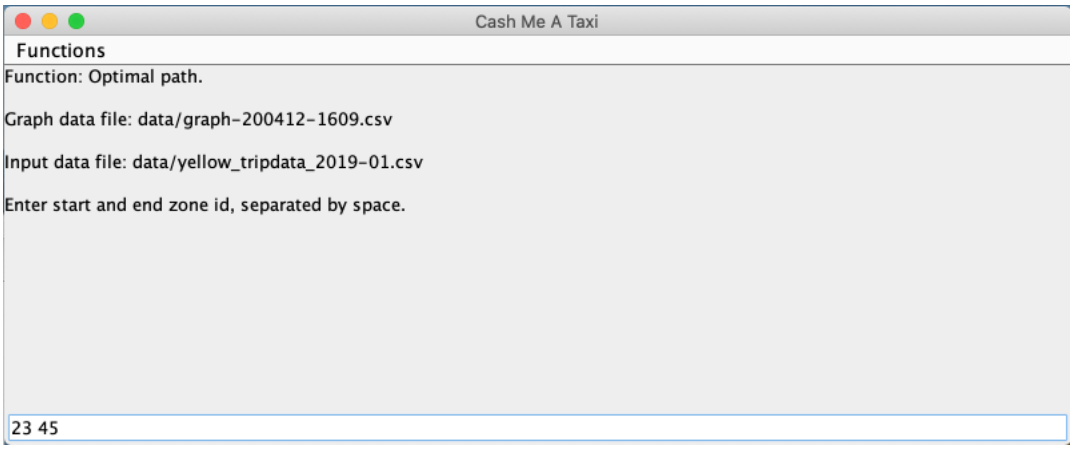


5. You are now able to choose another function from the "Functions" menu on the top menu bar.

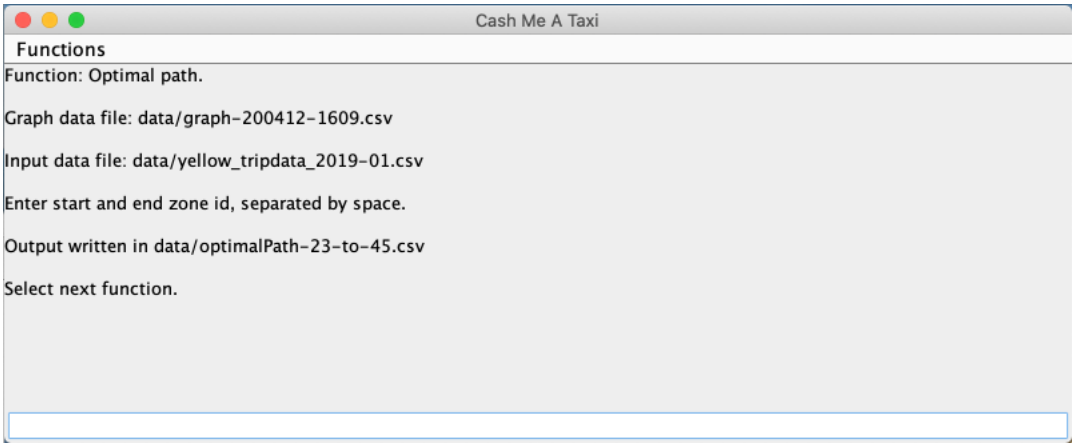
1.3.2 Optimal Path

1. When "Optimal Path" is selected, you will be given the following options. (Assuming you have already run the "Build Graph" function. If you have not, you will be prompted to input the file path. Return to 1.3.1 and repeat steps 2 and 3, before proceeding).

2. You are prompted to enter a start and end zone ID, separated by a space. It is assumed that the user will be familiar with the zone IDs of the city. Zones range from 1-263 and can be found at https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, under Taxi Zone Lookup Table.



3. Once the zone IDs are entered, the app calculates the optimal route and stores it in the file specified. .

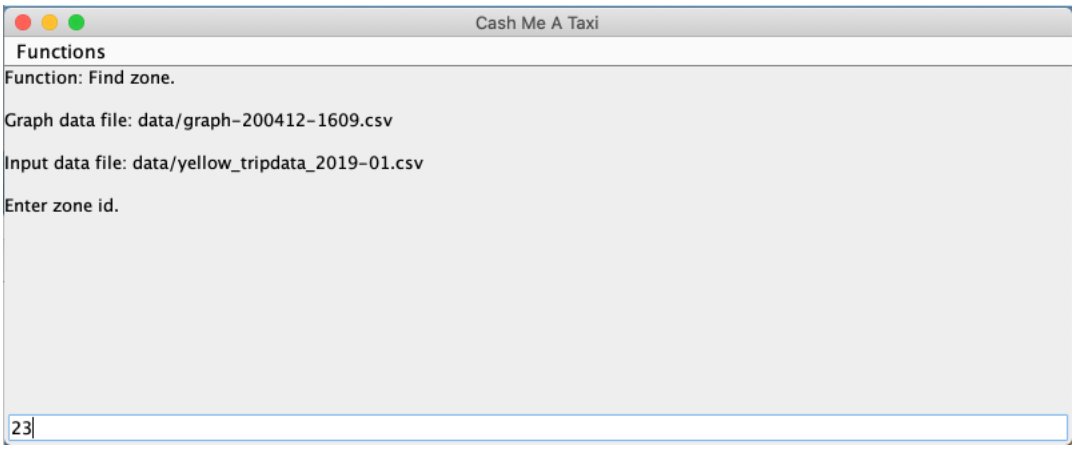


4. You are now able to choose another function from the "Functions" menu on the top menu bar.

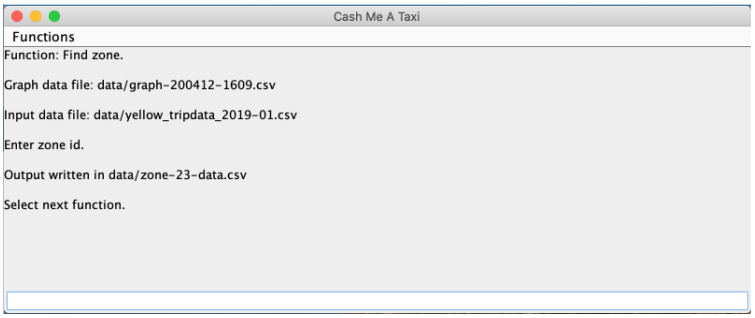
1.3.3 Find Zone

1. When "Find Zone" is chosen you will be given the following options. (Assuming you have already run the "Build Graph" function. If you have not, you will be prompted to input the file path. Return to 1.3.1 and repeat steps 2 and 3, before proceeding).

2. You are prompted to enter the zone ID you are looking for. It is assumed the user will be familiar with the zone IDs of the city. Zones range from 1-263 and can be found at https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, under Taxi Zone Lookup Table.



3. Once the zone ID has been entered, the app searches for the data stored in the node and writes the results to the specified file.

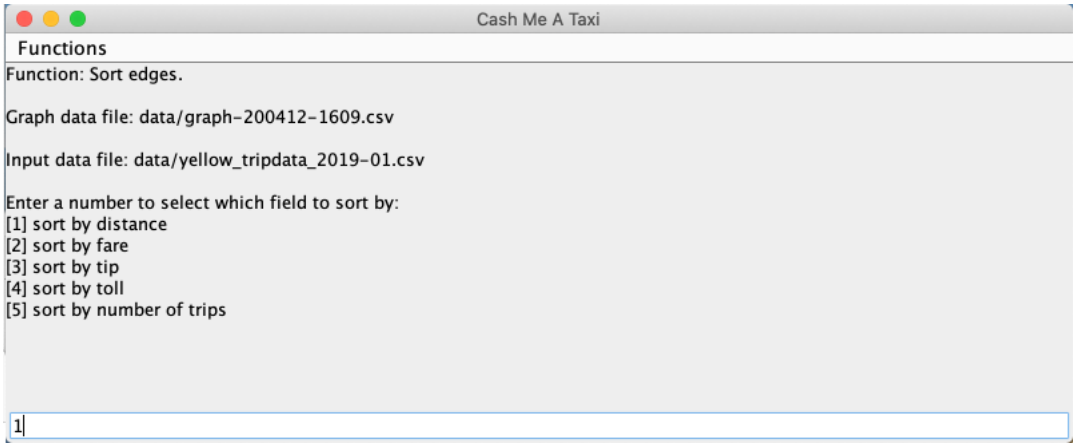


4. You are now able to choose another function from the "Functions" menu on the top menu bar.

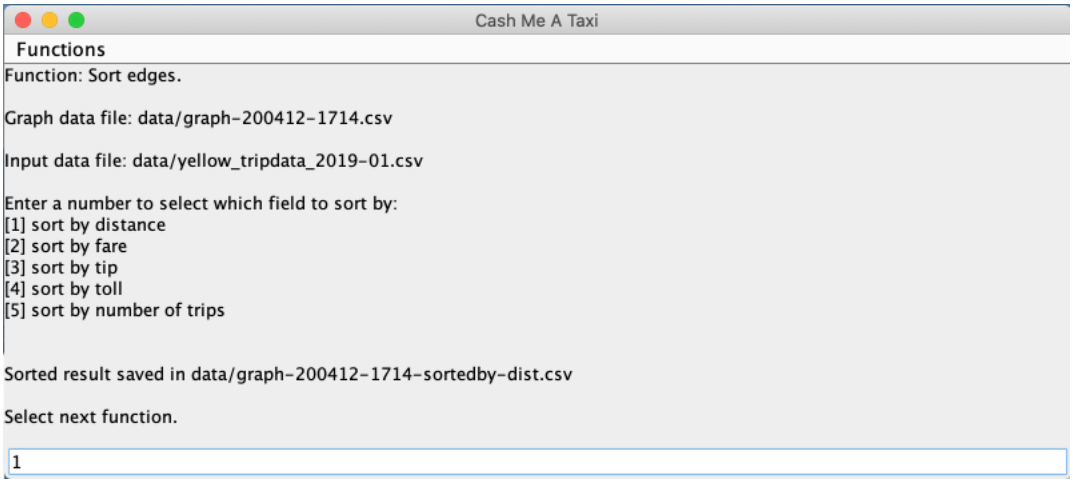
1.3.4 Sort Edges

1. When "Sort Edges" is chosen you will be given the following options. (Assuming you have already run the "Build Graph" function. If you have not, you will be prompted to input the file path. Return to 1.3.1 and repeat steps 2 and 3, before proceeding).

2. You will now be prompted to choose how you would like the edges to be sorted. Enter the number, corresponding to the option you would like to choose, in the text box at the bottom of the window.



3. When you have chosen your option, the app sorts the data and stores them in the file specified.



4. You are now able to choose another function from the "Functions" menu on the top menu bar.

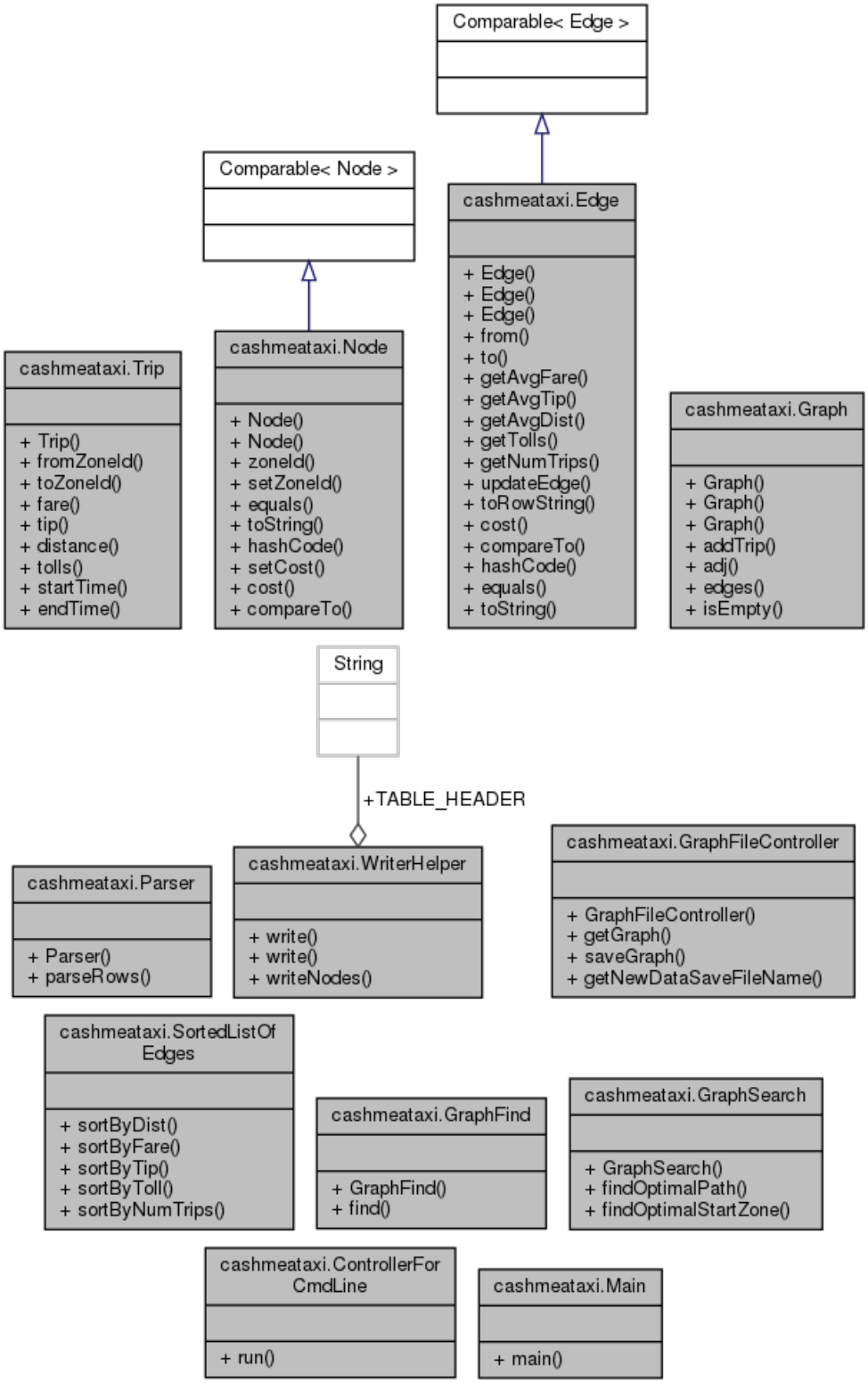
2 Design Overview

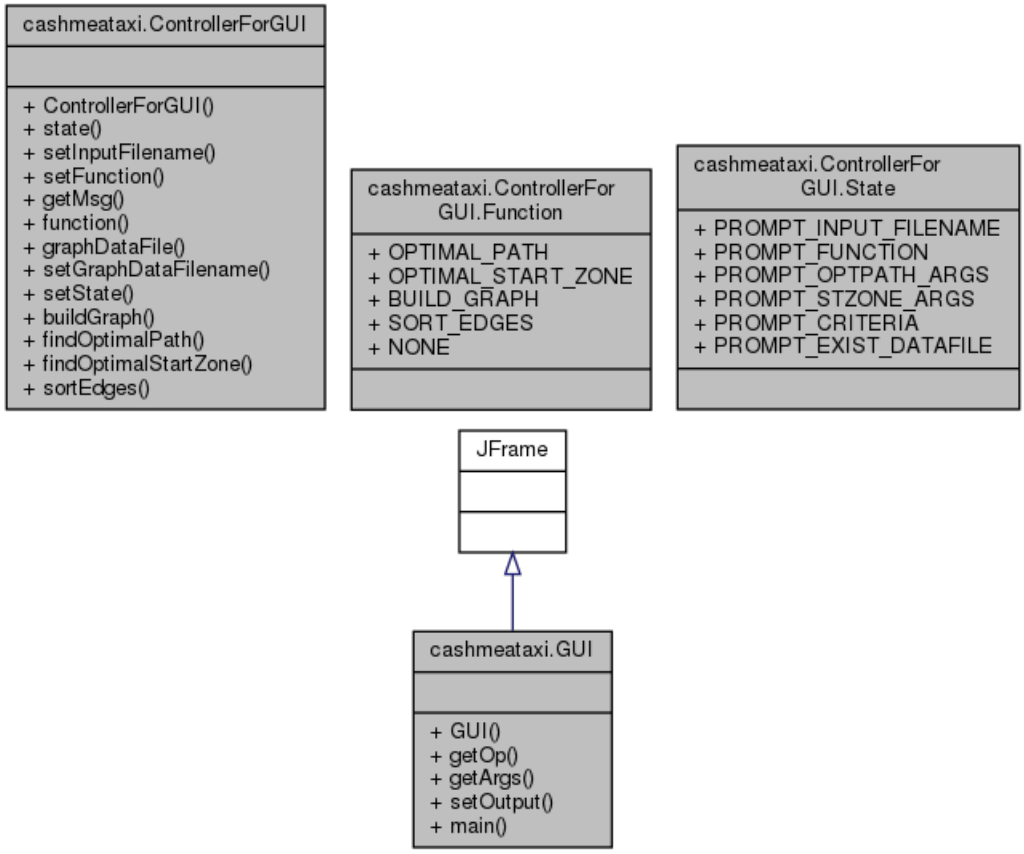
2.1 Module Decomposition

We wanted to emphasis the one module one secret idea so that many modules could be split up and implemented independently. We did not want the Graph module to be in charge of many tasks of the project, so we split them up into modules that use the Graph. The Parser is the only module that interacts with the input dataset to creates Trip objects that encapsulates the data of each dataset row. The Graph takes the Trips and creates Edges and Nodes to represent the graph. The SortedListOfEdges, GraphFind, and GraphSearch each contain the sorting, searching, and shortest path algorithms respectively. The GUI communicates with the user. The Controller takes the commands from the GUI, calls the appropriate tasks, then provides the GUI with the requested information.

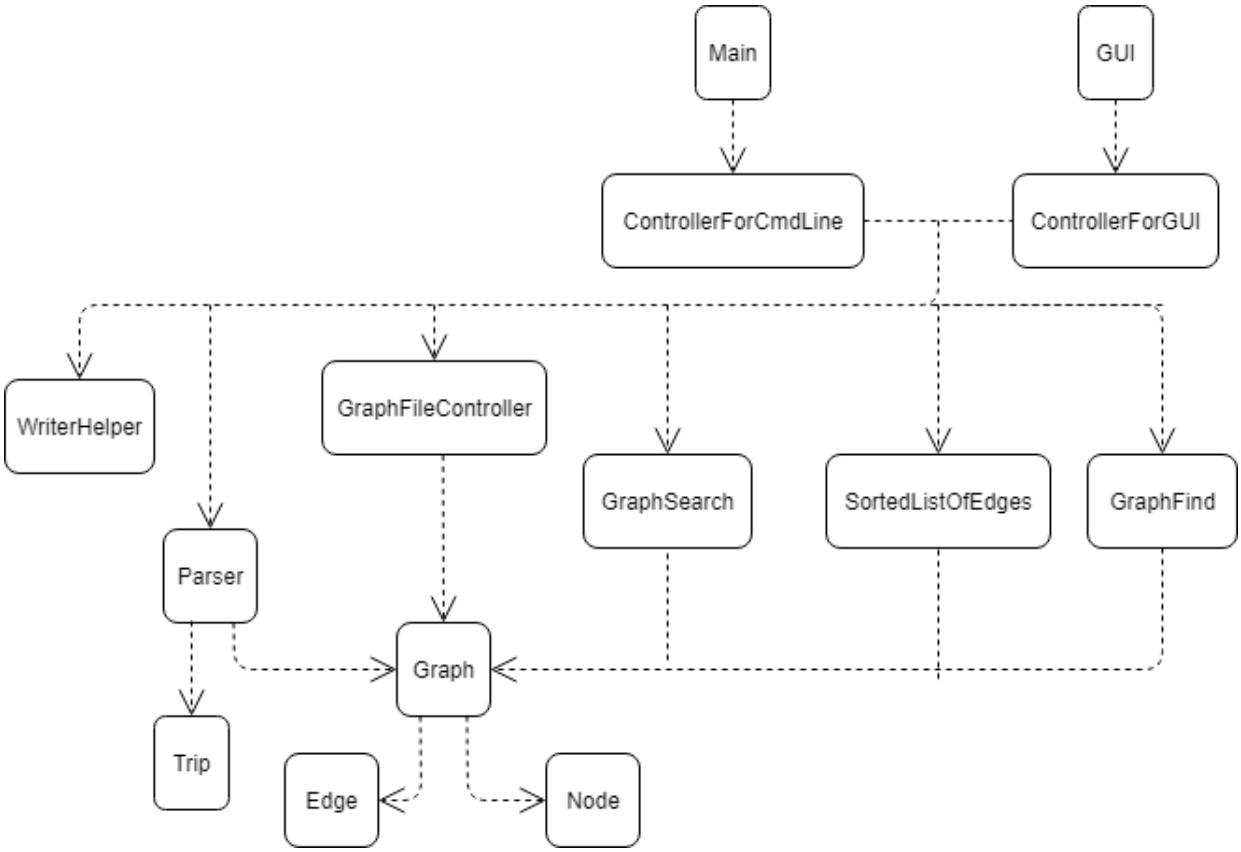
2.2 UML Diagram

Note that the relationships between modules are shown in the Uses diagram as there wasn't enough space to show all the uses relationships here.




cashmeataxi.ControllerForGUI
+ ControllerForGUI()
+ state()
+ setInputFilename()
+ setFunction()
+ getMsg()
+ function()
+ graphDataFile()
+ setGraphDataFilename()
+ setState()
+ buildGraph()
+ findOptimalPath()
+ findOptimalStartZone()
+ sortEdges()
cashmeataxi.ControllerFor GUI.Function
+ OPTIMAL_PATH
+ OPTIMAL_START_ZONE
+ BUILD_GRAPH
+ SORT_EDGES
+ NONE
cashmeataxi.ControllerFor GUI.State
+ PROMPT_INPUT_FILENAME
+ PROMPT_FUNCTION
+ PROMPT_OPTPATH_ARGS
+ PROMPT_STZONE_ARGS
+ PROMPT_CRITERIA
+ PROMPT_EXIST_DATAFILE
JFrame
cashmeataxi.GUI
+ GUI()
+ getOp()
+ getArgs()
+ setOutput()
+ main()

2.3 Uses Relationship Diagram



3 Modules

3.1 Parser

Parses the input dataset values into Trip data and then a Graph.

The parser has no state variables and one function. Its purpose is to read the dataset and create a set of `Trip` objects representing the data of every row.

3.1.1 Interface

- `Parser(String inputDataFile)`
  - Create a new Parser given the filename for the dataset.
- `long parseRows(Graph graph, long startRow, long numRowsToRead)`
  - Ignore the first row (header) + `startRow` rows of the file.
  - Reads rows up to `numRowsToRead`. For every row, parses the data, creates a `Trip` object, and adds it to `graph`.
  - Returns the number of rows read.

3.1.2 Implementation Details

Private entities:

- `final String inputFilename` The filename for the input dataset.

Notes:

- Uses a `BufferedReader` to read from the file.
- The reason why we want to specify the `startRow` when parsing data from the dataset is because if the dataset is very large and we've partially created a graph, we can continue from where we left off instead of starting all over again.

3.1.3 Requirements Traceback

The parser meets the original requirements. It properly reads the file, but instead of creating a graph it grows it.

3.2 Node

A node represents a zone in New York City, with its id being the NYC taxi zone id.

3.2.1 Interface

- `Node(int zoneId)`
  - Create a new Node given only the zone id.
- `Node(int zoneId, String borough, int numStartTrips, int numEndTrips)`
  - Create a new Node.
- `int zoneId()`
  - Get zone ID.
- `void setZoneId(int zoneId)`
  - Set zone ID.
- `boolean equals(Object other)`
  - Compare with another object.
  - Returns `true` if the other object is a Node instance and its `zoneId` is the same is this object's `zoneId`.
- `String toString()`
  - Return the string representation.
- `int hashCode()`
  - Return the hash code.
- `void setCost(Double cost)`
  - Set the cost.
  - Used for the graph search algorithm.
- `Double cost()`
  - Get the cost.
  - Used for the graph search algorithm.
- `int compareTo(Node other)`
  - Compare with another `Node`.
  - This object is greater than the given object if its cost is greater than the given Node's cost.

3.2.2 Implementation Details

Private entities:

- `int zoneId`
- `double cost` // for use in the graph search alg

Notes:

- `cost` is initially positive infinity. This is used for the graph search algorithm.

3.2.3 Requirements Traceback

The node meets the original requirements. It properly contains the contents of a New York City zone.

3.3 Edge

An edge represents a connection from one `Node` to another. Its state variables include ID, starting `Node`, ending `Node`, average fare, average tips, average distance, tolls, trip count, and weight. It has a function `update` that updates its state when given a new `Trip`.

3.3.1 Interface

- `Edge(Node nodeFrom, Node nodeTo, long numTrips, double fare, double tip, double dist, double tolls)`
  - Create an Edge with the given data.
- `Edge(Node nodeFrom, Node nodeTo, double fare, double tip, double dist, double tolls)`
  - Create an Edge with the given data.
  - This constructor doesn't require the `numTrips` field - it initializes the `numTrips` state variable as 1.
- `Edge(Trip t)`
  - Create a new Edge given a Trip object.
  - This constructor calls the second constructor above using the data retrieved from the given Trip object.
- `Node from()`
  - Get source Node.
- `Node to()`
  - Get destination Node.
- `double getAvgFare()`
  - Get average fare.
- `double getAvgTip()`
  - Get average tips.
- `double getAvgDist()`
  - Get average distance.
- `double getTolls()`
  - Get tolls.
- `long getNumTrips()`

    - Get the number of trips.
- `void updateTrip(Trip t)`
    - Updates the state variables when given a new `Trip` object.
    - Recalculates its average fare, average tip and average distance fields given the new Trip object data.
- `String toRowString()`
    - Return the string representation.
- `double cost()`
    - Return the cost in distance / money earned.
    - This is used for the graph search algorithm.
- `int compareTo(Edge other)`
    - Compare with another `Edge`.
    - This object is greater than the given object if this cost() is greater than other.cost()
- `int hashCode()`
    - Return the hash code.
- `boolean equals(Object other)`
    - Compare with another object.
    - This object is considered equivalent to the given object if its source nodes and destination nodes are equal.
- `String toString()`
    - Return the string representation.

3.3.2 Implementation Details

Private entities:

- `Node nodeFrom` // Source Node
- `Node nodeTo` // Destination Node
- `double avgFare` // Average fare
- `double avgTip` // Average tips earned
- `double avgDist` // Average Distance

- `double tolls` // Tolls
- `int numTrips` // Number of trips along the edge

Notes:

- `updateTrip(Trip t)` assumes the given `Trip` has the same source and end Nodes.
- The hash code is the integer equivalent of `nodeFrom.zoneId()` concatenated with `nodeTo.zoneId()`.

3.3.3 Requirements Traceback

The edge meets the original requirements. It properly represents the connection between 2 zones and updates when given new data.

3.4 Trip

Wrapper class to encapsulate a single Trip record, whose data is retrieved from a single row in the dataset.

3.4.1 Interface

- `Trip(int fromZoneId, int toZoneId, double fare, double tip, double distance, double tolls, String startTime, String endTime)`
    - Create a `Trip` with the given data.
- `int fromZoneId()`
    - Get source zone ID
- `int toZoneId()`
    - Get destination zone ID
- `double fare()`
    - Get the trip fare
- `double tip()`
    - Get the tips
- `double distance()`
    - Get distance
- `double tolls()`
    - Get tolls
- `String startTime()`
    - Get the start time
- `String endTime()`
    - Get the end time

3.4.2 Implementation Details

Private entities:

- `int from` // Starting zone
- `int to` // Ending zone
- `double fare` // Fare
- `double tip` // Tip
- `double distance` // Distance
- `double tolls` // Tolls
- `String startTime` // Start time
- `String endTime` // End time

3.4.3 Requirements Traceback

The trip meets the original requirements. It properly contains the contents from a row of the file.

3.5 Graph

A collection of nodes and edges. Represents the directed graph with nodes representing a NYC zone, and edges between zones representing that taxi trips were made between these zones.

3.5.1 Interface

- `Graph(ArrayList<Node> nodes)`
    - Create a `Graph` from the given `Node` list.
- `Graph(List<Edge> edges)`
    - Create a `Graph` from the given `Edge` list.
- `Graph()`
    - Create an empty `Graph`.
- `void addTrip(Trip t)`
    - Add an `Trip`. Add or update an edge to the graph given the new `Trip` object.
- `Iterable<Edge> adj(Node node)`
    - Get the adjacency list of the given `Node`.
- `ArrayList<Edge> edges()`
    - Get the list of all `Edge`.
- `boolean isEmpty()`
    - Determine if the `Graph` is empty.

3.5.2 Implementation Details

Private entities:

- `Map<Node, HashMap<Node, Edge>>` adj - Adjacency map.

Notes:

- `adj` is a map with keys representing each node in the graph, and the values are a hashmap of `Node, Edge` where the `Node` represents a node it is connected to. The reason why we chose to use a hashmap instead of a linked list for each node is so that retrieving the edge between a given pair of nodes would be O(1). Retrieving the edge between nodes is done quite frequently, so it is a good idea to use a hashmap for the adjacency list instead of a linked list, which would have resulted in an O(n) retrieval time for an edge.

3.5.3 Requirements Traceback

The graph meets the original requirements. It properly contains a collection of edges. Instead of updating with an edge it updates with a trip. A particular edge between a given pair of nodes can be fetched in O(1) time.

3.6 GraphFind

Find a node in the graph and retrieve its edges.

3.6.1 Interface

- `GraphFind(List<Edge> edges)`
    - Create a `GraphFind` object from the given `Edge` list.
- `ArrayList<Edge> find(Node node)`
    - Return the list of of outgoing edges from the given node.

3.6.2 Implementation Details

Private entities:

- `Map<Node, HashSet<Edge>>` graphAdjList - Adjacency map.

Notes:

- Searches using hashing, which takes O(1) time.

3.6.3 Requirements Traceback

The searcher doesn't meets the original requirements. It originally searched for an edge, but a single edge is not very useful so instead it searches for all the outgoing edges of a node.

3.7 SortedListOfEdges

Sorts a list of edges by a given field such as average distance, average fare, tips, tolls, and the number of trips made between the nodes in that edge.

3.7.1 Interface

- `static void sortByDist(ArrayList<Edge> edgeList)`
    - Sorts the list of edges by distance
- `static void sortByFare(ArrayList<Edge> edgeList)`
    - Sorts the list of edges by fare
- `static void sortByTip(ArrayList<Edge> edgeList)`
    - Sorts the list of edges by tip
- `static void sortByToll(ArrayList<Edge> edgeList)`
    - Sorts the list of edges by toll
- `static void sortByNumTrips(ArrayList<Edge> edgeList)`
    - Sorts the list of edges by number of trips

3.7.2 Implementation Details

- Sorts using a private class `Mergesort` that creates an `Edge[]` copy, sorts using standard merge sort with comparisons depending on the desired value, and copies the sorted array to the original.

3.7.3 Requirements Traceback

The sorter meets the original requirements. It properly sorts a list of edges.

3.8 GraphSearch

Traverses through the graph to search for the most optimal route between the given source node to the given destination node.

3.8.1 Interface

- `GraphSearch(List<Edge> edges)`
    - Create a new `GraphSearch` object from the given `Edge` list.
- `LinkedList<Node> findOptimalPath(int s, int e)`
    - Returns the most optimal path from the node specified by the zone id `s` to the node specified by the zone id `e`.
    - The resulting linked list includes the start node (if the given start node exists in the graph), and then the nodes to the destination node that specify the most optimal route. If no optimal route is found, then only the start node is returned.

3.8.2 Implementation Details

Private entities:

- `Map<Node, HashSet<Edge>> adj` // Adjacency map that represents the graph.

Notes:

- `findOptimalPath` employs Dijkstra's Algorithm:
    1. First find the start and end nodes in the graph that are specified by the given integer values that indicate the nodes' zone IDs.
    2. If either the start or end node don't exist, return an empty linked list.
    3. Set the "cost" value of the start node to 0 (as per Dijkstra's algorithm).
    4. Initialize a hashmap called `immediateParents<Node, Edge>` that will be used to track the immediate parents of each node to get the most optimal path.
    5. Set all immediate parents to `null` (the keys in immediate parents are all the nodes in the graph).
    6. Create a priority queue. Add the start node to the priority queue (as per Dijkstra's algorithm).
    7. Enter a `while` loop. Remove the minimum cost node from the priority queue. If this minimum cost node has already been visited (we keep track of visited nodes in the set `visitedNodes`), then we `continue` in the while loop.

8. Add the minimum cost node to the set of `visitedNodes`. Now visit this node's outgoing edges. For each outgoing edge whose destination node we have not visited yet, calculate a new cost (as per Dijkstra's algorithm) to this destination node, and if this new cost is less than its current cost, then set this node's cost to the new cost, and put the current outgoing edge as this node's immediate parent. Add the destination node of the outgoing edge to the priority queue to process later.
9. Once we exit the while loop (when the priority queue is empty), we have found the most optimal ("shortest") path. This path data is stored in the `immediateParents` hashmap. We now start with the destination node, and add its immediate parent to an array list. Then we add the parent's node's immediate parent to the array list, and so forth until we reach the start node.
10. The last thing to do is to reverse the array list and return it as a linked list which now has the correct optimal path from the source node to the destination node.

· To support the priority queue used in Dijkstra's algorithm described above, we modified our `Node` class to implement `Comparable<Node>` and have a function `compareTo(Node other)` that returns whether this node is of higher cost, lower cost or equal cost to the given node `other`. Because of this modification of the `Node` class, we can now easily use a `PriorityQueue` to hold a min heap of the nodes so that the retrieval of the minimum cost node is O(1) after every pass in the Dijkstra's algorithm.

3.8.3 Requirements Traceback

The graph searcher was originally part of the controller. However we found that it is best to separate the graph search module from the controller to maintain separation of concerns. This module successfully finds the optimal path between the given two nodes.

3.9 WriterHelper

Class to help write data onto csv files, specifically a list of edges or nodes from a graph.

3.9.1 Interface

- `void write(String filename, List<Edge> edges)`
    - Write the list of `Edge` objects to a file, in csv format, with information of each edge such as the source/destination nodes, number of trips, average fare, average tips, etc.
- `void write(String filename, String pre, List<Edge> edges)`
    - Similar to above, but also writes a message before the table and then writes the csv-formatted table.
- `void writeNodes(String filename, List<Node> nodes)`
    - Write the list of `Node` objects to a file, including each node's zone id. Writes each node on a separate line.

3.9.2 Implementation Details

Private entities:

- `final static String TABLE_HEADER` // The table header for the edges table.

Notes:

- Uses a `BufferedReader` to read from the file.
- Uses a `BufferedWriter` to write to the file.

3.9.3 Requirements Traceback

The writer helper wasn't originally part of the design. It was made to make file writing easier, which was also not part of the original design.

3.10 GraphFileController

Manages the file IO to save graph data and read graph data from the graph data file. This module is used to save a graph onto a file so that we don't have to go through the process of building a new graph with the same input dataset; instead we could simply "load" an existing graph if we want to run some more functions.

3.10.1 Interface

- `GraphFileController(String dataFileName)`
  - Create a `GraphFileController` with the given filename.
- `Graph getGraph()`
  - Creates a new `Graph` object given the data on the graph data file. If there's no existing data, then return a new empty `Graph` object.
  - Assumes that this data file is valid. A valid graph data file must have the first line be "numRowsReadFromInput=<number of rows read from the input data file>", the second line being the table header: "fromNode,toNode,numTrips,avgFare,avgTip,avgDistance,tolls", and each following line containing the above fields for each edge in the graph.
- `void saveGraph(Graph graph, long deltaNumRowsRead)`
  - Save the given graph in the file determined by the state variable `filename` (which was initialized at the constructor).
- `String getNewDataSaveFileName()`
  - Generate a new data file name, based on the current timestamp to avoid rewriting to existing graph data files.

3.10.2 Implementation Details

Private entities:

- String filename - Graph file name

Private functions:

- `void setupFile()`
  - Sets up the graph data output file. If the file given by the state variable `filename` already exists, then verify its contents to make sure its in the correct format.
  - If the file given by the state variable `filename` does not exist or has invalid contents, then create a new file (overriding the existing one) and write the skeleton contents (i.e. the table header).

- `long getNumRowsRead()`
    - Get the number of rows that have been read from the input dataset. This number is saved in the first line of the graph data file specified by the state variable `filename`.
- `static void writeInitialContents(Writer fwriter)`
    - Given the file writer `fwriter`, write the initial skeleton contents for the graph data file (i.e. the table header and the number of lines read from input dataset value to 0).
- `static boolean verifyFile(File file)`
    - Verify the given file by making sure its contents meet the required format for the graph data file.

Notes:

- Uses a `BufferedReader` to read from the file.
- Uses a `BufferedWriterer` to write to the file.

3.10.3 Requirements Traceback

The graph file controller wasn't originally part of the design. It was created to save graph data so that we wouldn't have to repeatedly re-build the same graph based on the same input datasets.

3.11 ControllerForCmdLine

Controller for the command-line application.

3.11.1 Interface

- `void run(String[] args)`
  - Perform the appropriate functions based on the arguments.
  - Command-line arguments: csvFilename functionToRun args loadExistingDataFile
    - functionToRun (required):
      1. Find optimal paths from zone A to zone B. Args required: startZone endZone
      2. Find zone. Args required: zone id
      3. Sort edges based on given a critera. Args required: criteriaId. criteriaId can be:
         - 1 - Sort by distance
         - 2 - Sort by fare
         - 3 - Sort by tip
         - 4 - Sort by toll
         - 5 - Sort by number of trips
      4. Build graph. Required args: none
    - loadExistingDataFile (optional): load existing graph data that was previously saved by this program. Eg. raphdata-200309-1619.csv

3.11.2 Implementation Details

Private entities:

- `String inputDataFile` // Input data
- `String saveDataFile` // Save data
- `Graph graph` // The graph

Private functions:

- `void buildGraph(boolean readInputData)`
  - Build a graph and save the graph data into a file. This also sets the state variable `graph` with the `Graph` object created.
  - If `readInputData` is `true`, then read the input dataset specified by the state variable `inputDataFile` to build the graph.
  - If `readInputData` is `false`, then build the graph based on an existing graph data file specified by the state variable `saveDataFile`.

- `void findOptimalPath(String startZone, String endZone)`
    - Find the optimal path from one zone to another. Writes the result into a file.
- `void findZone(int zoneId)`
    - Find a zone given by the zone ID. Writes the result into a text file.
- `void sortEdges(int criteria)`
    - Sort the `graph` edges based on a criteria.
- `void buildGraph()`
    - Builds the graph.
    - First checks if there is an existing graph data file specified by a non-empty `saveDataFile`. If there is, then build the graph without reading the input dataset. If there is no existing graph data file specified, then builds the graph by reading the input dataset.

3.11.3 Requirements Traceback

The controller meets most of the original requirements. It properly makes the graph on startup and performs the functions. It doesn't update the graph as the graph doesn't change, or use the GUI as the GUI is the higher entity.

3.12 Main

Main class for the command-line application.

3.12.1 Interface

- `static void main(String[] args)`
  - Run the `ControllerForCmdLine`.
  - The `args` passed to `main` must follow the rules specified in the `ControllerForCmdLine` specifications.

3.13 ControllerForGUI

Controller for the GUI application.

3.13.1 Interface

- `ControllerForGUI()`
  - Create a new `ControllerForGUI` object.
  - Initialize the state variable `state` to `State.PROMPT_FUNCTION`.
  - Initialize the state variable `function` to `Function.NONE`.
- `enum State`
  - Current state of the GUI application.
  - Values:
    - `PROMPT_INPUT_FILENAME`
    - `PROMPT_FUNCTION`
    - `PROMPT_OPTPATH_ARGS`
    - `PROMPT_FZONE_ARGS`
    - `PROMPT_CRITERIA`
    - `PROMPT_EXIST_DATAFILE`
- `enum Function`
  - Current function that is to be run by the user.
  - Values:
    - `OPTIMAL_PATH`
    - `FIND_ZONE`
    - `BUILD_GRAPH`
    - `SORT_EDGES`
    - `NONE`
- `State state()`
  - Return the current state.
- `void setInputFilename(String filename)`
  - Set the input dataset filename.
- `void setFunction(Function f)`
  - Set the function to run.
- `String getMsg()`

    - Return the message to display on the GUI depending on the current state and function being run.

- `Function function()`
    - Return the current function that is to be run.

- `String graphDataFile()`
    - Return graph data file.

- `void setGraphDataFilename(String filename)`
    - Set the save filename to save the graph data.

- `void setState(State state)`
    - Set the current state.

- `String buildGraph(boolean readInputData)`
    - Build `graph` and return the filename that the graph data is stored in.
    - If `readInputData` is `false`, then that means we don't want to read the input dataset, and we can build the graph from an existing graph data file.
    - If `readInputData` is `true`, then we must read the input dataset and build the graph based off of that.

- `String findOptimalPath(int startZone, int endZone)`
    - Find the optimal path given the start and end zone IDs.
    - Writes the resulting output to a file, and returns the filename of where the output is saved.

- `String findZone(int zoneId)`
    - Find the zone given by its id.
    - Writes data on the zone's outgoing edges in a file. Returns the name of this file.

- `String sortEdges(int criteria)`
    - Sort the graph's edges by a field specified by the `criteria`.
    - Sorted in ascending order.
    - Writes the sorted edges output on a file. Returns the name of this file.

3.13.2 Implementation Details

Private entities:

- `String inputDataFile` // Input dataset file
- `String saveDataFile` // Graph data file to save the graph
- `Graph graph` // Graph
- `State state` // Current state of the GUI application
- `Function function` // The current function that is to be run

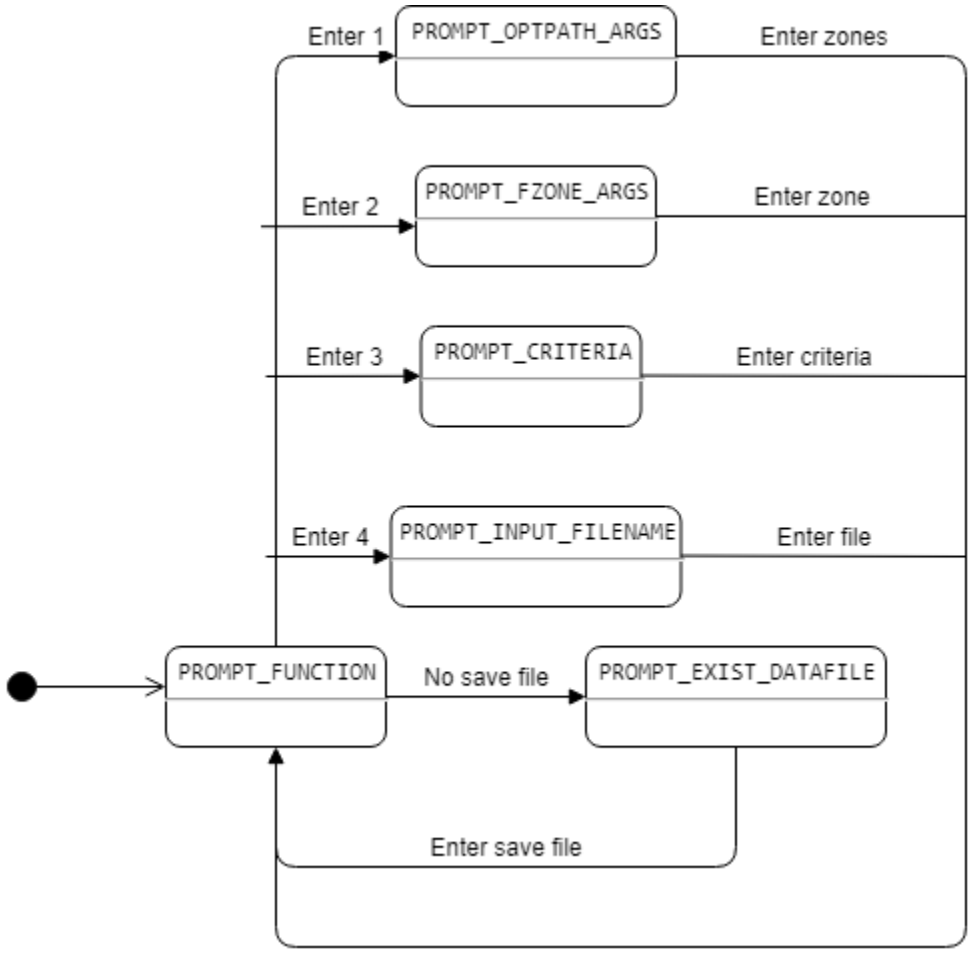
Private functions:

- `void buildGraph()`
  - Builds the graph.
  - First checks if there is an existing graph data file specified by a non-empty `saveDataFile`. If there is, then build the graph without reading the input dataset. If there is no existing graph data file specified, then builds the graph by reading the input dataset.

3.13.3 Requirements Traceback

The controller meets most of the original requirements. It properly makes the graph on startup and performs the functions. It doesn't update the graph as the graph doesn't change, or use the GUI as the GUI is the higher entity.

3.13.4 UML State Machine



3.14 GUI

Main class for the GUI application. Extends the Java `JFrame` module.

3.14.1 Interface

- `GUI()`
    - Create the GUI with a menu bar at the top, the input text at the bottom, and the output label between them.
    - When a menu option is selected, set the appropriate option and display the prompt.
    - Run the appropriate function and save the results to a file.
- `static void main(String[] args)`
    - The main function to run when running the GUI application.

3.14.2 Implementation Details

Private entities:

- `JTextField input` // Input textbox
- `JLabel output` // Output label
- `JMenuBar menu` // Menu bar
- `ControllerForGUI controller` // Controller

Private functions:

- `static void setOutput(String text)`
    - Set the text of `output` line by line.
- `static void validateZone(int zone)`
    - Validates the zone ID inputted by the user. The zone must be between 1 and 263, which is the range of valid NYC taxi zones.
    - Throws a new IllegalArgumentException if the zone is not valid. The calling function then catches the exception and displays an error message on the GUI, and prompts the user to select a new function.
- `static void validateSortOption(int option)`
    - Validates the sorting criteria option inputted by the user. The option must be between 1 and 5.

- Throws a new IllegalArgumentException if the option is not valid. The calling function then catches the exception and displays an error message on the GUI, and prompts the user to select a new function.

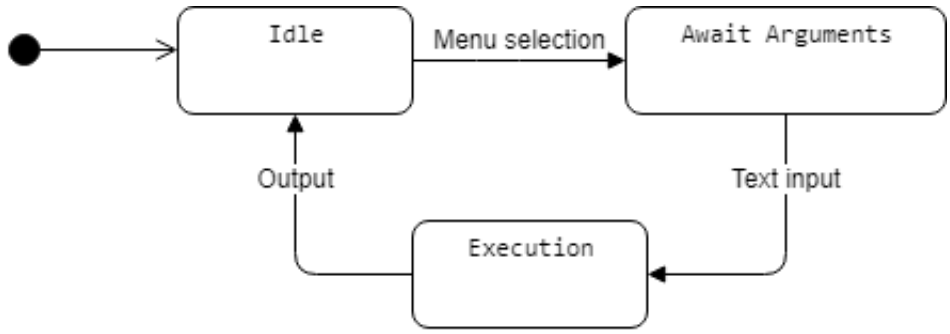
Notes:

- The GUI is implemented using Java Swing.
- It is a frame containing a menu, input textbox, and output label.
- When a menu option is selected the corresponding function is called via `controller`. If arguments are needed the user is given a prompt first.
- The function's output is displayed on `output` line by line.

3.14.3 Requirements Traceback

The GUI meets most of the original requirements. It properly makes accepts user input. Since most of the output will be too large to display, it is instead written to a file.

3.14.4 UML State Machine



4 Design Critique

The non-functional requirements set out by the SRD have been met. The application is always available as it only relies on files on the computer. It is secure as there is no personal data required. It is completely safe to use. Since the dataset is static, it is up to the user to update it to ensure the results are accurate. The performance is relatively fast compared to the size of the data. The memory usage is large as the dataset is large. The user interface could have some improvements such as shortcuts so that entire file paths don't have to be typed or a legend to map zones with real-world locations. The program is still constrained by the presence of the large dataset, but it can be used for other city datasets as long as they meet the same format. Running only on a computer with Java correctly installed is still a big constraint, but it shouldn't be a problem as it is not heavily used by the small audience.

Although the graph "saving" parts aren't completely necessary as the graph can be re-built everytime, we believe that as the dataset grows, it become significantly faster to run multiple functions over multiple runs of the program using the same dataset, since the graph will only be built once and subsequent program runs can use the already-constructed graph data to perform computations.