Bus Management System

Denisa Costinas

Sarah Dolan

Keira Gatt

Sean Langan

1 Introduction

Application Requirements

The objective of this project is to develop an interface using techniques learnt in class which will act as a bus information system for Vancouver public transport. For the purpose of this project, we have used three text files including stops.txt, transfers.txt and stop_times.txt. Using these input files, we have implemented the following functionalities –

- Finding shortest paths between 2 bus stops (as input by the user), returning the list of stops en route as well as the associated "cost"
- Searching for a bus stop by full name or by the first few characters in the name, using a ternary search tree (TST), returning the full stop information for each stop matching the search criteria (which can be zero, one or more stops)
- Searching for all trips with a given arrival time, returning full details of all trips matching the criteria (zero, one or more), sorted by trip id
- A graphical user interface that incorporates user input validation, GUI event management, HTML-based rendering of search results, runtime environment configuration and the integration of the search APIs for the above mentioned functionality.

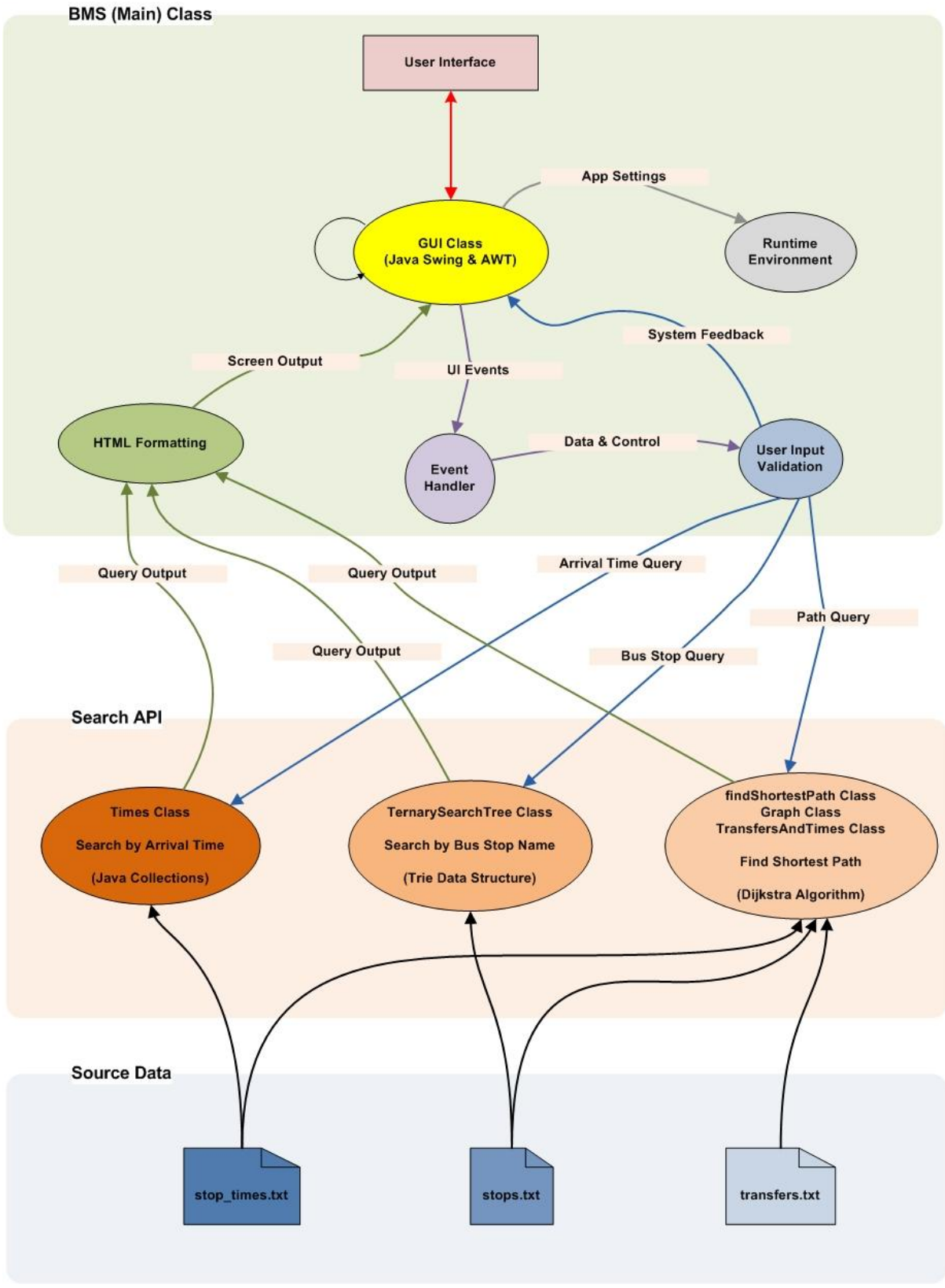
Project Organisation

Project decomposition involved assigning specific tasks and responsibilities to each individual as shown below –

Denisa Costinas	The design and development of the Ternary Search Tree algorithms and the implementation of the Bus Stop Name search API (Part 2) and documentation
Sarah Dolan	The implementation of the Search by Arrival Time API, based on Java Collections (Part 3) and documentation
Keira Gatt	The design and development of the GUI and supporting routines and the integration of the search APIs (Part 4), documentation and video demo
Sean Langan	The implementation of the Shortest Path API, based on the Dijkstra algorithm (Part 1) and documentation

2 Design

BMS Logic Flow Diagram



3 Implementation

Tools, Libraries, Platforms

- Eclipse for Java IDE on MacOS
- Eclipse for Java IDE on Windows
- Java Swing & AWT for GUI development
- GitHub (repo to store and share code)
- Instagram messaging for discussions
- Zoom for group meetings

Algorithms

The following are the design decisions in which we discussed and made, relative to each of the three main features:

Part 1:

- Consisted of three .java files, containing 8 classes, (findShortestPath, Dijkstra, Graph, Path, Node, Edge, and transfersAndTimes)
- Getting from one stop to another, consists of several interlinked transfers, where each single transfer is referred to as 1 Stop. For a transfer to exist, it needed a fromStopID, a toStopID, a minTransferTime, and a transferType. Each transfer has this very information stored about it in the transfersAndTimes class.
- Each stop was represented by a node, and each 'journey' was represented by an edge. From stop 1817, to 1818, there would be two nodes (1817,1818) and one edge connecting them. Travelling from one to the other is called a path, and when the weight is factored in, it is a transfer.
- To create the graph, I used a graph class, which takes in a string array containing our three files: "Stops.txt", "transfers.txt" and "stop_times.txt". Using these files, we sort them into arrayLists and add them to our graph by creating edges and nodes from the data.
- We used Dijkstra algorithm to traverse the graph and help to find the shortest path between two stops. I used Dijkstra because this was its intended purpose of creation, finding the shortest distance between nodes.
- The findShortestPath class creates a very simple was of finding the shortest path between two stops, to be implemented in the main. The two methods, (numberOfStops and stopsAlongTheWay) each return the number of stops between two entered stops, and the tickers and cost associated with these stops, respectively.
- Dijkstra algorithm has time complexity O(ELogV) and space complexity of O(V) where E= number of edges, and V=number of vertices.

Part 2:

- Ternary search tree was used.
- A ternary search tree is a type of prefix tree where nodes are arranged as a binary search tree.
- There are two classes, a class for the Node and a class for the TernarySearchTree.
- The node class has a constructer for the nodes in the tree.
- I have an insert function in the TernarySearchTree class which is used to insert the list of bus stops into the tree
- The average Time Complexity for the insert function is of O(log n) and the worst case is O(n)
- There is a search function in the TernarySearchTree class which is used to search for the word (bus stop name).
- The average Time Complexity for the search function is of O(log n) and the worst case is O(n)
- There is a function in TernarySearchTree class to traverse the tree to be able to find the searched word
- The average Time Complexity for the transverse function is of O(log n) and the worst case is O(n)
- This algorithm has a Space Complexity of O(1).

Part 3:

- This part entailed searching for all trips with a given arrival time, returning full details of all trips matching the criteria, sorted by trip ID.
- The data structure used in this part of the project includes the use of Array List. We choose to use an array list here as we are using an input file of over 1.7 million entries and array lists have a space complexity of O(N) .
- As well as that Array lists work well with Collections.sort() which I initially thought implemented quick sort but after research I found it implements Tim sort which is a hybrid stable sorting algorithm derived from merge sort and insertion sort.
- Collections.sort() also sorts elements presented in a specified list of collection in ascending order which is ideal as This part of the project the trips are needed to be sorted by ID.
- We found it ideal to use Collections.sort() as it is fast and has a guaranteed run time of O(N log N), it is stable meaning the order of stops is preserved.

GUI & Integration

- This area constitutes **Part 4** of the project and is made up of several components that are ultimately responsible to broker the services requested by the end-user from the Search APIs.
- The GUI framework is based on Java Swing and AWT and consists of several graphic components, layers and event handlers that translate user actions into a sequence of logical tasks. The human communications interface maintains a consistent layout and logic to simplify the user experience.
- Central to the GUI is the validation engine to ensure that all input does not exceed the system-imposed limits and to verify that data is always within range and in the correct

format before it is passed to the search APIs. A good part of the validation engine is based on regex to enforce strict conformance with pre-determined input patterns.

- The core module responsible for the presentation of search results was developed as a HTML generator, where the output from the search APIs is encoded and formatted for rendering in a similar way as web content on a browser agent. This provides the required flexibility to handle and display dynamic content.
- Another module incorporated with the GUI takes care of the runtime environment, where it is initialised and restrictions are applied, depending on the availability of resources. For instance, individual search functions are locked if the required data files are not available and facilities are provided to the end-user to customise the source location of these files.

Runtime Environment

- BMS can be invoked either from the *Eclipse for Java IDE* by running the main class file *BMS.java* or by executing the compiled bytecode classes directly in *JRE* from the command line as follows -

  java -Xms128M -Xmx512M -classpath <location of .class files> BMS

  Note the inclusion of the *-Xms128M -Xmx512M* options to adjust the heap memory size allocated for the runtime environment. These settings are also required if the application is executed in *Eclipse for Java IDE* and can be set as follows :

  - Right click on *BMS.java* and choose *Properties*
  - Select *Run/Debug Settings*, choose *BMS* and click *Edit*
  - Select tab *(x)= Arguments* and declare the 2 options as *VM arguments*

- The application requires access to all data files that were provided for this project, i.e. *stop_times.txt*, *stops.txt* and *transfers.txt*. It is usually sufficient to have these files in the same folder as the runtime code, for which the full path is displayed at the bottom of the application main screen. However, a different location can be specified during runtime by using the *Set Data Path* option, which works for both MS Windows and Unix/Mac file systems.