[CSU22012 Final Project] [Darragh Nolan]

[10/04/22]

Student ID: [20335426]

Bus Management System

The bus management system is represented using a Graph object. This is a directed, edge-weighted Graph. The Graph contains an Adjacency List of directed Edge objects.

I used an Adjacency List for this representation rather than an Adjacency Matrix as the input files are large. An Adjacency List takes up Θ(V + E) space whereas an Adjacency Matrix takes up Θ(|V|^2) space. Although, in the worst case (dense) an Adjacency List can take up the same amount of space. In an Adjacency List it is faster to add/delete a node than in an Adjacency List(O(n^2) - complex operation). Furthermore, Adjacency Lists find adjacent vertices much faster than an Adjacency Matrix, which is useful here.

Each Edge has an origin, destination, and distance (direction and weight), along with their respective getters.

The Graph uses HashMaps to track pairs of stop information for easy retrieval later. In general, HashMap is preferred over HashTable if thread synchronization is not required. In a HashMap, threads do not need to wait so performance is better.

This Graph receives the input file names as parameters. The files are read in using BufferedReader rather than FileReader as wrapping a FileReader in a BufferedReader gives improved performance. BufferedReader can also read one line at a time using readLine(). As the input files took up a large amount of space, BufferedReader was used instead of Scanner (larger buffer memory, etc.).

To achieve functionality number 1, I had intended (I had it mostly implemented but could not troubleshoot in time) to use Dijkstra’s shortest path algorithm with a Priority Queue, rather than Floyd Warshall’s algorithm as I had discovered its benefits in the previous assignment. Floyd Warshall's worst-case runtime of O(N^3) cannot compare to Dijkstra's O(NlogN) in this particular case.

For the second requested functionality (search for stop by its first few characters), a Ternary Search Tree was specified. This was appropriate for the given task as its insert, delete and search methods have a worst-case runtime complexity of just O(n). The TST implementation I chose did not have unnecessary complexity or bloat for its purpose. Also, Ternary Search Trees are effective in autocompletion. The TST was filled with stop objects.

The stop class contains a method to find stops that match a given string (using a TST). This class also contains a stop constructor that moves keywords(WB, SB etc.) from the start of stop names to the end of the string, in order to provide meaningful search results, as was specified. The ArrayList listOfAllNames tracked each stop's line number for easy access later. For speed of access, an ArrayList was chosen.

A scanner in the main method of app.java takes user input and passes their search into a TST. The stop class contains a method to print every stop that matches a given string. It separates each of the properties of a stop using .split(","), as the input file is comma-separated. Again, however, if no matching stops are found, an appropriate incorrect input message is given to the user.

The 3rd functionality I did not successfully implement in time.

Functionality number 4 involved creating an interface for the user to select one of the functions or exit the program. I used a command line interface. In the case of user input, I used a switch and its "default" is an error message in order to deal with any input outside of the range (handling incorrect input was also specified). As a way of exiting the program was required, a user input of "0" terminates the running JVM using System.exit(0);