Analysis Document

Introduction

The shortestPath and searching by stop name functionality of the project reference the Princeton Book code, and modify it for the purpose of the project.
FileReading and the command-line functionality implemented on the main were imperative to the project and may affect the runtime of the system, however they are not described in the analysis document as it was not asked for in the project specification.

1. Shortest Path between two stops

For the graph, reference has been made to EdgeWeightedDigraph, DirectedEdge, and Bag classes from the Princeton book. For the shortestPath, references have been made to DijkstraSP and IndexMinPQ from the Princeton book. Modifications have been made to most classes to make it work for the project specifications.

The graph uses an adjacency-list representation, using (E + V) space. It takes V time to create a graph with V vertices. I encountered one problem with the adjacency list representation, the nodes are labelled from 0 to V, and this is how they are indexed in the list, however bus stop IDs can have numbers higher than V. I created an array called stopMapping, and as the stops were read in from stops.txt, stopMapping[i] was assigned the value of the stopID of the current line (i) from the file. This was the only way I could think of translating stopIDs to nodes on the graph, however it's worst case time is V for searching for a stop map, with V/2 being the average case. Also due to the necessity of stop mapping, creating an Edge between two nodes is no longer a constant runtime, as when creating an edge from v to w, you must find the nodes in the graph corresponding to v and w, therefore takes (2V/2) average time per edge creation.

Dijkstra was the chosen shortest path algorithm for a few reasons. Firstly, it was evident from the project specifications that we would be looking for a single target shortest path. This ruled out Floyd warshall as the space needed for all paths (V^2) was unnecessary. Bellman-ford was unneeded as there would be no negative weights present, as shown in the project specifications.
Since the graph would consist of directed edges, using topological sort for shortest path would be impossible. I determined that Dijkstra would be ideal, as the worst case runtime of (ElogV) and the space complexity of (V) are ideal for this situation. I thought about the efficiency of A* shortest paths, however I do not know how I would compute a heuristic in a reasonable time for this assignment

2. Searching for a bus stop by using TST

In this project, TST has a basic implementation, which was taken from the Princeton book. However, a series of steps were made within the TST class in order for it to work properly and provide the user with the correct output.

Firstly, Queue class was added, using the slightly shortened version of the code from the Princeton book. It provided TST with queue objects and various methods to manipulate them.

Secondly, readBusStation was created. It contains functions, which fill two 2D arrays in the TST class with the information provided in the stops.txt file. While arrays are fixed and therefore may cause a crash of the code, they were still considered a better alternative to the array lists, as the number of stops in the stops.txt was limited and constant. Moreover, 2D arrays allow to easily store both stop id and relevant stop names in a convenient way. Another advantage of the 2D arrays in this case is the fact that they can be compared in the for nested loops with a very small amount of code.

Although nested for loops might seem as not the most efficient solution, it was still used due to its simplicity and the fact that the number of stops is limited.

3. Searching for stop_times by arrival time

Originally I had planned on pushing each line of stop_times.txt into an arraylist, which I could sort, and create subarrays through the search function. However, stop_times.txt is a very large file, with around 1.7 million entries, so when trying to push into an a arrayList it would produce a error from lack of memory. This also made me realise that it is not necessary to read in and store every item, since the data is stored in a txt and can be read out as Strings which is exactly what is needed.

Currently the algorithm first runs through stop_times.txt and finds all entries with the matching arrival time, and adds the associated trip ID's to a list. This list is then used when we run through stop_times.txt again to pull all the associated entries out and push them into a separate List. This list is then sorted and printed out to the command line.

The asymptotic worst case running time for this function right now is O(N^2), where N is the size of stop_times.txt, because of the function to pull the trip ID's from the stop_times file. In practice the size of the tripID list will never be close to the size of stop_times.txt, but since it is directly based off the size of it, it will still scale with N. The first loop for reading in the trip ID's will always be cost N running time since it is simply iterating over the text file in its entirety.

I'm not entirely happy with this solution, as it can be unbearably slow when returning a lot of entries. The one thing that this solution is good for is that we are saving a lot of space by not having to read in the entirety of stop_times.txt, and only storing the data we want to output to the user as variables within our code.