CSU22012 Algorithms & Data Structures II Design Document

Matthew Dowse, James Fenlon,
Matthew Grouse and Ciara Lynch

For the shortest path between two bus stops, we chose to implement Dijkstra's shortest path instead of Floyd Warshall's because we only wanted to find the shortest path from one bus stop to another, using the first input as the source vertex and the second input as the destination. This would make Dijkstra the better choice for this problem as it's time complexity is far better suited as Floyd Warshall would take much more time trying to calculate all possible shortest paths from the source vertex. We did not implement Bellman-Ford Shortest Paths as we were already familiar with Dijkstra and there are no negative edges to take into consideration for this problem so Dijkstra seemed like a more appropriate algorithm. We did not use A* shortest path algorithm as we were more familiar with Dijkstra and it was easier to implement. Also, A* uses greedy best first search which would take more space than Dijkstra. Even though A* is faster than Dijkstra, we felt Dijkstra's Shortest Path algorithm was more appropriate for this problem.

One important method we had was finding the total amount of bus stops so that we knew how many Vertices there were for the Edge Weighted Directed Graph. This consisted of a basic for loop that would add one to a counter for every line in the file "stops.txt" as each line represented a bus stop's information, remembering to skip the first line as it is the details.

The exact logic was applied to find the total amount of edges, but this time using "transfers.txt" as each line represented a directed edge. Then for "stop_times.txt" Matthew Dowse had to compare two lines, ensuring that they shared the same trip_id and then add one to count. After discussing this together, we thought this could have been improved using a buffer reader.

Now, we had the total amount of Stops (Vertices), and total amount of Edges, we can make the Edge Weighted Directed Graph. Then, we added the edges using "stop_times.txt" making sure to only add an edge if two lines shared the same trip_id, again making sure to not skip a line in the file. The weight of the edges for this file was 1. Then for "transfers.txt" adding each line as an edge, the difference being if the transfer type was equal to 0 then the cost/weight would be 2, and if the transfer type was equal to 2 then the cost/weight would be minimum transfer time divided by 2.

Following on from this we needed to return all the bus stops between the two stops that the user inputted. We made a method that added the stops to a List when "from" bus stop was found in the file, it would then keep adding the next bus stop on the next line to the List until it reached the "to" stop inputted by the user and then return immediately.

To approach the second part of the project it was decided to break the part down into the different functionalities required to create. We understood that section 2 would require us to be able to take user input whether it was the full bus stop name or just a few characters and return all the information about the bus stops that match the inputted criteria.

After the team fully understood what this part of the project required Ciara began to tackle the certain sections. The class would be an implementation of the TST algorithm and would be working primarily with the "stops.txt" data file. From accessing the way the data was structured in the file we came to the conclusion that Ciara would have to implement some form of string manipulation to move the keywords specified in the project specification to the end of the "stop_address". Here we researched online and in the Algorithms 4th Edition

Textbook by Robert Sedgewick and Kevin Wayne how to create a basic Ternary Search Tree.

In the class the basic TST code alongside the string manipulation was implemented; Which was a for loop iterating through the inputted string from the data file and splitting it where necessary. The newly adapted bus addresses were placed into a Ternary Search Tree and a for loop was created to search and compare against the user's inputted strings/characters. This was all coded into a function that could be easily called by Part 4 in the creation of the User Interface. Each of the team members in this project ensured that all of the code and functionalities could easily be called throughout the project to maintain a high level of consistency and legibility for future version control and changes.

Ternary Search Trees are an industry implementation standard when it comes to searching, iterating and storing string based data. It is much more space-efficient than multiway trees and other algorithms and therefore for this project can support more advanced searches if necessary. As a result it was decided to limit the amount of changing to the data structure containing the addresses as possible as this would slow down the algorithm as a whole and take up more space than necessary. Therefore Ciara maintained a consistent usage of an Array List taking in the input then the structure of a Ternary Search Tree to store.

For the third part we broke the question down and put together a loose plan on how to attack it. It would need a fully error handled user input to take in the desired arrival time. This should account for times exceeding 24:59:59, and any typos or strings entered by the user. Next, "returning full details of all trips" stood out to the team. It was interpreted as it must return the full details including the locations and areas. The output should be all organised by the tripID in ascending order.

Once the team was happy that we understood the brief James set out the loose plan. The plan was to convert both files necessary into their own 2D array as it will save time and memory when searching for times to just use the column that holds the times. It would scan through the arrival times column to match every occurrence of the desired time. Both files share the stopID value so by pulling that value from the line that there was a match we could fetch all the stop information from the line that matches the same stopID. Then format all this information nicely by ordering the output to display by tripID. Quicksort is a great algorithm if there are a lot of numbers in the array to sort through, so the team decided to prepare for the worst case of many times matching the user input which is why James chose quicksort. This would then be formatted to output nicely by separating the information on different lines with titles stating what they are.

James worked by completing the tasks in the order it is written above. James created functions to turn the files into 2D arrays with the only trouble being allocation of special characters, quickly sorted with small error handling to ensure the input reads correctly and reads what is in the text file.

Next up, James got the user input of the arrival time and sent it to a function which searches in the stop_times file for all the matches. This is done by just two simple for loops as the worst case will be the best case because it must go to the end of the column to find every single match. When a match is found it pulls the stopID and the tripID. The stopID is so that the program can search in the other file and return the stop information regarding the time given by the user. Following the same process in this function it uses for loops to loop through the 2D array searching for the line that matches the stopID. This line is then returned.

Finally, the last task James had to do to complete part number three was to format what would be returned from the function as a string. A for loop was created and it continues to print the amount of matches there are ordered neatly and in ascending tripID number. This makes it very easy for the user to read through all available stops that have an arrival at the time of the user's input.

To conclude, we implemented the algorithms in the most efficient way possible for us as a team in regards to the space and time complexity of the project problem. The team tried to ensure a consistent approach to the space and time complexities of the algorithms throughout the program alongside a subtle and concise user interface to complete the project created by Matthew Grouse.