DSA Report

Introduction:

We identified our project essentially consists of 2 parts: one where you see the graph and edit it (by adding, deleting edges) and the other just simply using the application to get the shortest path. So we proceeded to make **2 logins**, one '**Admin**', who can access the first part, change graphs, and '**User**' who simply enters source and destination and gets desired output.

Explanation of thought-process:

Seeing some real-world data (we considered Bangalore as our model city, which has, on average, 300 vehicles per kilometre), we divided the congestion ratio (simply traffic/edge length) into **blocks of speed**. This speed denotes the max speed a vehicle can drive on the road. We then calculated the time as simply **edge length/appropriate speed**.

We then ran our **dijkstra algorithm** on the **time taken** and not the shortest distance. We understand that the shortest path need not always be take the lowest time (because of a higher congestion) and hence we want to output the **shortest time instead of path**.

Our final output for user is the **distance to be travelled, minimum time, time of departure,** and the **estimated time of arrival** designed with a neat interface. We are also showing the **path for shortest time** from source to destination.

Here, we are taking input for the graph from a **file**. As explained previously, only an admin will be able to access and change the graph temporarily through code, and for permanent changes, they would have to change in the input file.

Right now, we have **5 input files** with different kinds of graphs, each representing a city. We have formed the graphs in such a way so that we can have the data for **dense** (several edges, on average, for one node) as well as **sparse graphs**. These input files were **randomly generated** and the **5 cities** that we have are:

- city-1 contains only 10 nodes so it a **small, sparse graph**.
- city-2 contains 46 nodes and it's a **small, dense graph**.
- city-3 contains 380 nodes and it's a **medium, dense graph.**

- city-4 contains 2952 nodes and it's a **medium, sparse graph.**
- city-5 contains 1154 nodes and it's a **medium, highly-dense graph.**

The **map folder** contains the **visualization of map** of the city 1,2,3 and 4 using **graphviz**. City 5 is highly dense and a graph visualization could not be generated for this graph.

Data Structures Used:

We have **stored our graph** in the form of an **adjacency list**, thus using the concepts of a **Singly Linked List.** We started by having a **structure of the node**, containing information like the destination, edge length, congestion, time and a pointer to the next node.

Similarly, we had a **structure for the graphs**, which contained an **array of node pointers** along with the number of nodes in that graph.

We are using a **priority queue**, to select the edge taking minimum time. This **priority queue** was in the form of **min-heaps** and this was another data structure that we used.

For this implementation, we had a **structure of the queue node**, containing information like vertex and time taken. We also had a **structure for the priority queue** itself, which contained information like the number of nodes in the priority queue as well as an array of pointers to the priority queue.

We can run the required functions, which are **insertion, extraction** (extracting the min), **heapify, and decrease key**, on our priority queue.

Algorithms Used and Complexity:

Since we are using the **Dijkstra algorithm** with a **min-heap (priority queue)**, the time complexity reduces to $O((E+V)log(V))$ from $O(V^2)$. In a dense graph, the number of edges is more than the number of vertices **(E>V)** and that is why, in the worst case (a dense graph), the time complexity reduces to $O(Elog(V))$. The first part $(Elog(V))$, arises due to the **decrease key** operation (relaxation) for each edge. The second part $(Vlog(V))$ happens because of the **extract min operation** for each vertex. The **heapify function** takes $log(V)$ time and hence is a common factor in both terms that can be taken out common. Our function to **choose the minimum** is implemented in $O(1)$. To check the validity of our **Dijkstra algorithm**, we have written also written a function to **choose minimum** in $O(V^2)$.

However, this function took a longer time to run compared to our implementation for all the input files, thus showing a faster algorithm that we have implemented.

We are also **inserting** and **reading** from the **adjacency list.** In the worst case, inserting into the adjacency list is $O(V)$. We have implemented in this way to avoid time complexity issues that arise from an **adjacency matrix** (which is $O(V^2)$).

Additional Features:

- The user interface that we have implemented in our program is user-friendly and easy on our eyes.
- We are giving the user the choice to navigate in 5 cities, whose graphs are of both sparse and dense nature.
- There is a clear distinction in the login profiles and the UI helps the user navigate the application easily. The admin profile is password-protected for the simple reason that everybody should not have access to the intricacies of the graph.
- In the admin profile, we have given the user the choice to see the graph and make temporary changes according to his will. However if he would like to make permanent changes, he would have to make those changes in the file.
- We have implemented a time feature, which takes the time of execution and shows an estimated time of arrival after once the path is calculated.
- A unique feature we have implemented is visualizing the graph. We have used the tool graphviz.dot and it shows edges between a graph.

Division of work:

We tried to ensure that our teammates worked in parallel, on different parts of the code. At every given point of time, at least 2 of our teammates worked together to increase understanding of code as well as to avoid monotonosity.

Anusha:

Was crucial to the understanding and implementation of Dijkstra using min-heaps and related functions like heapify, decrease key and function related to choosing the minimum. Also worked on implementing the brute force method. Was crucial in designing and implementing the User Interface.

Arnav:

Worked on implementation of min heaps. Helped in understanding congestion and further calculations. Worked on implementation of User Interface.

Geet:

Worked on the implementation of the graph and related functions. Was crucial in the understanding of congestion and further calculations. Worked on implementing and designing the User Interface.

Sai Pranav:

Worked on the implementation of graph and related functions. Worked on implementing the User Interface.

Srijan:

Worked on understanding and implementation of Dijkstra using min-heaps and the brute force method. Was crucial in understanding congestion and further calculations. Worked on implementing the User Interface

Scope for Improvement:

- Our visualization using graphviz.dot works for small graphs (100 odd nodes). It takes fairly long for graphs with 1000 nodes and above that, it is unable to generate the graph. Hence a scope for improvement is to use better and stronger tools for faster and easier visualization.
- We also searched for better heuristics. We found that a better time complexity can be executed if Fibonacci heaps are implemented. The time complexity of the program then becomes $O(E + Vlog(V))$.
    1. This reduces the time complexity from $O((E + V)log(V))$ because of the difference in time complexity of the decrease function. As explained previously, min heaps do the decrease function in $O(log(V))$ time whereas the Fibonacci heap does the operation in $O(1)$ time, which induces a difference in the time complexity.