Data Structures and Algorithms

Integrated RailHub Dynamics Management System (IRHDMS)

Course Project Report

School of Computer Science and Engineering

2023-24

Contents

1. Course and Team Details

1.1 Course details

Course Name	Data Structures and Algorithms
Course Code	23ECSC205
Semester	III
Division	A
Year	2023-24
Instructor	Mr. Prakash Hegde

1.2 Team Details

Si. No.	Roll No.	Name
1.	113	Vishwa M Badachi
2.	115	Sai Satya B V
3.	148	Shreya Inamdar

1.3 Report Owner

Roll No.	Name
148	Shreya Inamdar

2. Introduction

All of us have heard of the Mumbai local trains. The life of Mumbai- as they are called by most of Mumbaikars. But as convenient as the local trains are, they have some demerits too. A major one being ill crowd management in local trains as well as stations. As Mumbai local trains are cost efficient and faster compared to any other commute available so, they are preferred by everyone. Thus, the only way is to manage the crowds efficiently.

Also, there is a lot of confusion for a person when visiting the city for first time. There is need to have an efficient guide/recommendation software for the passengers/tourists that suggests places in Mumbai and lets the person know the time required, distance to be travelled and nearest station to travel to.

One can also think of the fact that if there is so much crowd and around 88 stations in Mumbai, there must be a huge employee database and a vast amount of revenue. Thus, it is necessary to manage employees and calculate revenue efficiently.

These needs and difficulties were noticed when we went through the white paper named 'Streets' about life in Mumbai. After reading about the commute in Mumbai, a need was felt that there must be something which could make the commute easier and smoother. Thus, the choice of need statement for our project.

3. Problem Statement

3.1 Domain

The inefficiency in crowd management during peak hours of boarding/alighting from local trains in Mumbai poses significant inconvenience to passengers. Additionally, the chaos hinders first-time visitors from exploring the city. Addressing this, there is a pressing need for an effective recommendation system that guides passengers to optimal places and stations. Furthermore, the extensive data related to railway staff, trains, maps, and revenue necessitates the development of an efficient data management system to streamline operations and enhance overall railway functionality.

Addressing problems with the most significant impact on a citizen's life is crucial. Among the services offered in the city, railway services are the most widely used by commoners. Therefore, we believe that enhancing this system would be very beneficial.

3.2 Module Description

Owing to the vast size of the city along with the complicated local train system; navigating the ends of the city is a cumbersome task, especially for visitors who are not used to travelling in the bustling trains and are unfamiliar with the routes and the destinations.

Visitors often want to see particular locations but lack the knowledge of how to get there, what are the expected costs, travelling time and what other similar locations are present near-by that are worth visiting.

4. Functionality Selection

Si. No.	Functionality Name	Known	Unknown	Principles applicable	Algorithms	Data Structures
	(Name the functionality within the module)	(What information do you already know about the module? What kind of data you already have? How much of process information is known?)	(What are the pain points? What information needs to be explored and understood? What are challenges?)	(What are the supporting principles and design techniques?)	(List all the algorithms you will use)	(What are the supporting data structures?)
1	Collecting the information about the average distance, time and cost of visiting a place from a particular location	We can acquire a list of the most popular locations in the city which are a hub of tourist activity and store them in a text file. These values can be read from the files and stored in an AVL tree	Finding out the data required for the travel costs and time as this is very subjective.	Transform and conquer	Using self-balancing AVL (Adeleson - Velsky and Landis) tree	Binary Search Tree
2	Sort the locations into groups where each group defines an activity that the user wants to enjoy	Different activities are listed and are assigned to each of the locations in the files	Reading the particular category from the text file and copying it at a required location to be inserted into the AVL tree	Brute force	InOrder traversal of a Binary Search Tree	Binary Search Tree
3	Based on the starting point and the destination selected, provide the shortest path with respect to the distance, cost and time.	The cost matrices for the different attributes have been created by reading the values from the tree	Applying a suitable shortest path algorithm and finding the route and the weight from it	Greedy Technique	Dijkstra's Algorithm	Minimum heap, arrays, structures
4	Provide the intermediate travel costs between particular locations	The entire cost matrix is available	Creating an array for the cost of locations in the shortest path array by running a loop and changing the indices in the Fenwick query	Dynamic programming	Fenwick Tree	Arrays
5	Display the maximum cost between intermediate locations in the route	The entire time matrix is known	The lookup indices need to be adjusted as the	Space-time Trade-off	Lookup Table	2D-arrays, matrices

6	Display the nearest location from the destination for the user to explore	We have the distances between the locations from the arrays	Creating a adjacency matrix from the distance matrix and performing cautious traversal on the graph	Decrease and conquer	Breadth-First Search	Adjacency Matrix(2D-array), queues
7	Sort the near-by locations based on ratings, distance, time or cost	The various sorting techniques are known which can be employed for this purpose	The information about the weights is stored in trees so there is a need to extract this and store in a suitable structure for sorting	Brute force, Decrease and Conquer, Divide and conquer	Bubble Sort, Insertion Sort, Merge Sort, Quick Sort	Structures, Arrays
8	Provide a matrix of what the lowest costs could be to travel the near-by locations	The individual weights are present so there is a need to find the distances between particular locations closest by distance and provide a matrix for lowest cost. These are available in the matrix created from the tree	Providing the shortest possible weight from every node to every other node	Dynamic Programming	Floyd's Algorithm	2D- arrays (Matrices)
9	Find particular locations based on their name	The names of the location are present in the tree	Finding out the intuition behind using the length of the longest prefix-suffix to reduce the number of comparisons as compared to using brute force	Space-Time trade-off	KMP (Knuth-Morris-Pratt) InOrder Traversal	Arrays, Binary Search Tree
10	Update the location information	The new updates and changes are available from the user	Performing tree traversal to find and update the correct node and performing respective changes in the matrices as well. If the user wishes to delete a location, then setting all the weights to and from it to infinity and re-balancing the tree	Brute force	In-Order Traversal	Arrays, Binary Search Tree

5. Functionality Analysis

1. Reading the information from the files and storing it in an AVL tree

The file that was created to store the information of the various location has the following pattern:

50 49
00 Name category 5
1 2 3 4…
1 2 3 4…
1 2 3 4…

Here, the first number in the file represents the total number of locations present currently. The second number is the last location number in the file (this is used for incrementing while adding the next location). From the next line the location information starts. Th has the location number, followed by the name and the category and finally the rating. The next 3 lines have the distance, time and cost respectively from that location to every other location.

This information is read from the file and stored in a node. This node is then inserted into a Binary Search Tree based on the location number.

However, the resulting tree will be skewed as the location numbers are in a particular order (ascending thus the tree will be right skewed). To solve this issue and decrease the retrieval time of the nodes later, a self-balancing AVL tree was employed.

AVL trees have the time complexity for retrieval similar to the binary search of an array. At every comparison, we discard half the nodes. Thus,

$$\frac{n}{2^k} < 1 \implies k = \lfloor \log_2 n \rfloor + 1$$

Though rebalancing needs to be performed at instances, the overall growth remains Θ log n

Space complexity is linear; as the number of nodes increase, the space increases with it as well.

2.Sort the locations based on activities and display the results

Each location in the file was provided with a category. The tree needs to be traversed and all the locations that fall within a particular category need to be displayed. The technique that was followed was In Order traversal of the tree and then using the strcmp () function to see if the category was matched.

This is a brute force way to find all the location as every node needs to be traversed and compared. Thus, the efficiency class of this module will be linear.

Time complexity: Θ (n) where n is the number of nodes/locations in the tree.

3.Calculating the shortest path based on different parameters from the source to the destination

From the data in the tree, a cost matrix can be created which holds the weights from one node to the other. Applying Dijkstra's algorithm with a min-heap to find

the shortest path gives the location numbers, which can be used to traverse the tree for the names.

[1] The given graph G=(V, E) is represented as cost matrix and the priority queue Q is represented as a binary heap;

it takes O(|V|) time to construct the initial priority queue of |V| vertices.

Iterating over all vertices' neighbors and updating their distance values over the course of a run of the algorithm takes O(|E|) time.

The time taken for each iteration of the loop is O(|V|), as one vertex is removed from Q per loop.

The binary heap data structure allows us to extract-minimum (remove the node with minimal dist) and update an element (recalculate dist[u]) in O(log|V|) time.

Therefore, the time complexity becomes O(|V|) + O(|E| x log|V|) + O(|V| x log|V|), which is O((|E|+|V|)xlog|V|) = O(|E|xlog|V|), since |E| >= |V| - 1 as G is a connected graph.

From the distance and path arrays, calculating the shortest path takes Θ V

The space complexity class is linear as 2 arrays will be formed.

4. Provide the intermediate travel costs between particular locations

The result of the Dijkstra's algorithm would be the path that the user can follow. The distance between the intermediate nodes could be calculated based on the Fenwick Logic. The Binary indexed tree gives a complexity of Θlog(n) similar to that of binary search. It uses a clever logic of calculating the partial sums of the array segments at the indices at a distance of x&-x where x is the previous index.

This array query method was used to find the distance between intermediate nodes however some changes needed to be implemented in the query as the user input was the location numbers while the arrays store the intermediate costs.

5.Display the maximum cost between intermediate locations

A maximum look-up table can be employed for this. It is based on the principle of space-time trade-off. The answers to the range queries are already computed and stored in the matrix. Thus, query has a constant time. The table can be created using 3 loops

$$\sum_{i=0}^{n} \sum_{j=i+1}^{n} \sum_{k=i}^{j} 1 \approx n^2$$

The efficiency class for creation of the table is quadratic. The space efficiency is also quadratic as a nxn matrix needs to be created for an array of n elements.

In the implemented program, the indices of the lookup have been adjusted as the user mentions the location numbers but the distances are intermediate. Thus, the indices have been shifted.

6.Display the locations close to the destination to be explored

At the destination, all the tourist locations near-by need to be displayed. Thus, an adjacency matrix was created from the distance matrix and Breadth First search was employed to find other locations in the vicinity.

BFS uses a queue data structure and performs cautious traversal as opposed to Depth First Search. [2] Breadth First Search has a running time complexity of

O(|V|+|E|) since every vertex and every edge will be checked once. Depending on the input to the graph, O(|E|) could be between O(1) and O(|V|²).

7. Sorting the near-by locations based on rating, distance, time, cost

Based on the location number of the close locations found in the previous functionality, an array of structures needs to be created for sorting the locations. This will take Θ(l*logn) time as every time the tree needs to be traversed to find the required information for sorting (l is the number of near-by locations and n is the total number of locations). Various sorting techniques have been employed for the above purpose. All of them are in-place except for merge sort which needs extra space to copy the elements and sort them. The following are the time complexities of the sorting techniques.

- Bubble sort:

$$\sum_{i=0}^{n-2} \sum_{i=0}^{n-2-i} 1 \approx n^2 \Rightarrow B(n) \in \Theta(n^2)$$

- Insertion sort:

$$\sum_{i=1}^{n-1} 1 \approx n-1 \Rightarrow I_{best}(n) \in \Omega(n)$$

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 \approx n^2 \Rightarrow I_{worst}(n) \in O(n^2)$$

$$\sum_{i=1}^{n-1} \frac{i+1}{2} \approx n^2 \Rightarrow I_{avg}(n) \in \Theta(n^2)$$

- Merge sort:

$$M(n) = \begin{cases} 0 & \text{when } n=1 \\ 2(M(n/2)) & \text{otherwise} \end{cases}$$

$$\Rightarrow M(n) \in \Theta(n \log n)$$

- Quick Sort:

$$Q_{best}(n) = \begin{cases} 0 & \text{when } n=1 \\ 2Q_b(n/2) + cn & \text{otherwise} \end{cases}$$

$$\Rightarrow Q_{best}(n) \in \Omega(n \log n)$$

$$Q_{worst}(n) = \begin{cases} 0 & \text{when } n=1 \\ Q_w(0) + Q_w(n-1) + cn & \text{otherwise} \end{cases}$$

$$\Rightarrow Q_{worst}(n) \in O(n^2)$$

$$Q_{avg}(n) = \begin{cases} 0 & \text{when } n = 0 \\ 0 & \text{when } n = 1 \\ \frac{1}{n}\sum_{s=0}^{n-1} Q(s) + Q(n-s-1) + C(n+1) & \text{otherwise} \end{cases}$$

$$\Rightarrow Q_{avg}(n) \in \Theta(n \log n)$$

8. Provide the lowest cost matrix for the locations

The above functionality employs Floyd's algorithm. It uses dynamic programming to calculate the shortest path from every node to every other node.

It uses three 'for loops' thus the time efficiency class is cubic.

9.Finding a particular location based on its name

Traversing the entire tree will take linear time as every node will be checked. Further, the name of every location in the tree needs to be compared in the substring from the user. The Knuth-Morris-Pratt Algorithm is used to accomplish this.

[3]For analysing the time complexity of the KMP string matching algorithm, let's define a variable k that will hold the value of i - j.

k ← i - j

So, for every iteration through the while loop, we can expect one of three things to occur:

We find a match, and hence increment both i and j, hence k remains the same

There is a mismatch but j > 0, so i does not change and hence we can say that k increases by at least 1 since k changes from i - j to i - LPS [j - 1]

There is a mismatch but this time j = 0, hence i is incremented and k increases by 1.

So, each iteration of the loop, either i increases by 1, or k increases by 1. Hence the greatest possible number of loops would be 2n.

Now talking about the LPS compute function, our table is computed in linear time complexity so we can say that the time complexity is O(m).

Hence, the total time complexity of KMP algorithm is O (n + m).

10.Updating the location information

Finding the correct node in the tree will take logarithmic time. Further changing the respective matrices will be at a constant time. Writing each node back to the files will be nxm where n is the number of current nodes present while m is the maximum nodes that were present. This method was adopted as the relative distances need not change and the matrix and the location numbers need not be changed in cases of deletion.

6. Conclusion

Structured data management has evolved over time to address specific needs and challenges encountered in storing and accessing information efficiently. Working on this

project gave me a better understanding of what were the incentives and needs that led to the creation of these structures.
Different algorithms that have been devised and used for ages have an idea associated with their solution. Working on these algorithms like the different sorting techniques, sub-string/ pattern matching algorithms and spanning tree algorithms gave me the intuition that led to their discovery. It also made me realize how we are constantly surrounded by algorithms and how they can be learnt and implemented even in the normal course of life.

7. References

[1] Understanding Time Complexity Calculation for Dijkstra. Link:
https://www.baeldung.com/cs/dijkstra-time-complexity
https://www.baeldung.com/cs/dijkstra-time-complexity
[2] BRILLIANT- Breadth First Search link:
https://brilliant.org/wiki/breadth-first-search-bfs/
[3] SCALAR KMP Algorithm Link:
https://www.scaler.com/topics/data-structures/kmp-algorithm/

~*~*~*~*~*~*~*~