Data Structures and Algorithms

[Optimizing Commuting Route]

Course Project Report

School of Computer Science and Engineering
2023-24

Contents

Si. No.	**Topics**
1.	Course and Team Details
2.	Introduction
3.	Problem Definition
4.	Functionality Selection
5.	Functionality Analysis
6.	Conclusion
7.	References

1. Course and Team Details

1.1 Course details

Course Name	Data Structures and Algorithms
Course Code	23ECSC205
Semester	III
Division	B
Year	2023-24
Instructor	Prakash Hegade

1.2 Team Details

Si. No.	Roll No.	Name
1.	206	Nagendra Hanchinale
2.	212	Dsouza Mark Xavier
3.	215	Hithin Chitriki
4.	236	Shridhar Anigolkar

1.3 Report Owner

Roll No.	Name
206	Nagendra Hanchinale

2. Introduction

Our cities are bursting with people, and getting around can be a nightmare. Traffic jams waste time, pollute the air, and make everyone grumpy. But what if there was a way to make commutes smoother and less stressful?

This project digs into a research paper that explores cool ideas for optimizing commuting routes in big cities. Imagine millions of people zipping around on buses, trains, bikes, and scooters, all using the fastest and most efficient paths. By making even small improvements, we could dramatically change the way we travel!

The research uses real-time traffic data and fancy computer algorithms to figure out the best routes for everyone. It's like having a personalized GPS that adapts to traffic jams and road closures. This wouldn't just be good for individuals – it could also mean cleaner air, happier people, and even a boost to the economy!

Our job is to take the key findings from this research and make them practical. We'll explore how these ideas could actually be implemented in cities, what kind of technology we might need, and who needs to work together to make it happen. Optimizing commutes isn't just about traffic lights and apps – it's about smart planning, clever technology, and working together to build better cities for everyone.

3. Problem Statement

3.1 Domain

The white paper highlights the significant issue of long commutes and traffic congestion in Metro City, impacting over 2 million daily commuters. The goal of this project is to create an automated system that offers personalized route suggestions based on real-time traffic data and predictive algorithms. Even small improvements in routes could save millions of hours for commuters each year. The aim is to enhance transportation efficiency, alleviate congestion, and ultimately improve the overall quality of life.

This project focuses on addressing a widespread problem with the potential for substantial positive impact. By tackling urban transportation challenges, it aligns with broader societal objectives such as sustainability, increased productivity, and the creation of more livable cities. The proposed automated routing system is designed to be scalable and applicable to numerous major metro areas facing similar transportation issues.

3.2 Module Description

I'm developing a navigation system that helps users find the optimal route to their destination. It doesn't just provide directions but also keeps them entertained in traffic, informs them about interesting places along the way, and reveals unique characteristics of those locations. Users can even refer crowd-sourced ratings to discover hidden location and choose their final destination based on what attracts them the most.He can even find the efficient route for this trip.The system prioritizes efficient routes to save time and ensure a pleasant ride.

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	Graph representation and storage	We have the data about the places and their roots and distance between them.	Pain point is to read it from files and arrange it in matrix. We have to go through the their root connection and distance 1.Storage optimization 2.Time and space complexity	1.Adjacency matrix storage	There are many like(Dijkstra's algorithm Bellman, Floyd,Warshall) but I am goin to use Dijkstra's algorithm.	rout e[], map[][] (possible alternatives: adjacency matrix, list)
2	Printing the spanning roots of the graph	we know the connection of cities	the pain piont is that we have assign one route for one city without revisting it . We have go through routes data and come with correct path	Minimum cost network, efficient traversal greedy technique would be best.	DFS-algorithm	Graph data structures (arrays, matrix)
3	Search for places	Checks if a type string exists within a city name.	More efficient alternatives to brute-force search for large datasets. Handling case sensitivity and variations in city names.	String matching, search optimization	bfss algorithm	Arrays (city[])
4	Search for places	It allows users to search for places based on type	Logic for matching user input with place types. Handling ambiguous or misspelled searches.	User input validation, pattern matching	Brute force	Arrays (places[])

5	Display city information: city_root, inorder	Builds binary search tree (BST) for city names. Performs an inorder traversal to print city names.	Specific implementation details of BST operations (insertion, searching).	Data organization, efficient search	BST	city_root BST
6	User interaction and additional features	Prompts users for input and controls program flow.	Specific user interface and error handling logic. Making the program work for their data without effecting ours will be difficult.	User interface design, modularity	N/A (For this there is no needof using algorithm as we have provide only space where a user insert their data .)	structures places,city,routes etc;
7	Game while in Traffic	User is asked if they are stuck in traffic and input an estimated time. If the time is above 15 minutes, they are offered a choice of three games: Rock-Paper-Scissors, Number Guessing, or Watch Instagram Reels	Specific validation logic for user input (traffic time).	User input validation, error handling,using random numbers;	inbuild function where function chooes randomly	srand(), rand(),
8	Display the places acoording asked ratting	Places data is available. User can input a desired rating. Places are displayed based on the requested rating.	How to handle places with the same rating as the requested one.	Data Filtering and Retrieval: Efficiently finding places that match the requested rating criteria	Sorting Algorithm (QuickSort)	Array
9.	Customised map	User can create his own map by inputting data in runtime	Specific user interface and error handling logic. we have arrange the code in the way that this data of user doesn’t effect our main data in file.	User interface design, modularity	N/A	Structure.

5. Functionality Analysis

Function 1:Functionality Analysis: load_from_file1
Input: city_root: The root node of the Binary Search Tree (BST).
Output : TREE*: The new root of the BST with loaded place names.

Time Complexity : O(n log n), where 'n' is the number of places (pcount).

Space Complexity :O(n), where 'n' is the number of places (pcount).

Functionality Analysis: insert_into_bst
Input: root: The root node of the Binary Search Tree (BST).
p_name: The place name to insert.

Output:TREE*: The new root of the BST.

Time Complexity :O(log n), where 'n' is the number of nodes in the BST.

Space Complexity : O(1) - The function uses a constant amount of space regardless of the input size.

Functionality Analysis: check_greatest
Input: place1: The first place name.
place2: The second place name.
Output: 1 if place1 is lexicographically greater, 0 otherwise.

Time Complexity : O(min(len(place1), len(place2))) - The time complexity is determined by the length of the shorter string.

Space Complexity : O(1) - The function uses a constant amount of space regardless of the input size.

Functionality Analysis: inorder
Input: root: The root node of the Binary Search Tree (BST).
Output : Prints node place names.

Time Complexity:O(n), where 'n' is the number of nodes in the BST.

Space Complexity:O(h), where 'h' is the height of the BST.

Function 2:Functionality Analysis: print_route
Input:None
Output:void (No return value, prints routes to the console)
Time Complexity : O(n log n), where 'n' is the number of routes (global_count).

Space Complexity:O(n), where 'n' is the number of routes (global_count).

Functionality Analysis: merge
Input params:
  routes: The array of optimalroot structs to be merged.
  left: The index of the first element in the left subarray.
  right: The index of the last element in the right subarray.

Output : void (No return value, modifies the original array)

Time Complexity : O(n), where 'n' is the total number of elements being merged.

Space Complexity : O(n), where 'n' is the total number of elements being merged.

Functionality Analysis: mergeSort
Input params:
- routes: The array of optimalroot structs to be sorted.
- left: The index of the first element in the subarray to be sorted.
- right: The index of the last element in the subarray to be sorted.

Output : void

Time Complexity : O(n log n), where 'n' is the total number of elements in the array.

Space Complexity:O(n), where 'n' is the total number of elements in the array.

This overall complexity is based on the fact that the mergeSort function is the dominant factor in terms of time complexity, and its time complexity is **O(n log n)** .

Function 3: Dijkstra's Algorithm
Input:
nn: The total number of nodes in the graph.
Matrix: The adjacency matrix representing the graph, where Matrix[i][j] stores the distance between node i and node j.
sourceNode: The name of the starting node.
destNode: The name of the destination node.

Output:

Prints the shortest path and distance between the source and destination nodes.
Prints the travel time based on user-selected speed.

Time Complexity:
The main loop of Dijkstra's algorithm runs for nn iterations.
In each iteration, it searches for the unvisited node with the minimum distance, which takes O(n) time.
The overall time complexity is O(n^2), where 'n' is the number of nodes in the graph.

Space Complexity:Uses arrays (dist, path, visited) of size 'nn' to store distances, paths, and visited status.
The space complexity is O(nn), where 'nn' is the number of nodes in the graph.

Function 4: Sorting and Searching
1. sort Function:
Input:None.

Output:None.

Time Complexity:The sort function uses quicksort, which has an average-case time complexity of O(n log n).
In the worst case, when the array is already sorted, it becomes O(n^2).

Space Complexity:
The quicksort algorithm is an in-place sorting algorithm, so the space complexity is O(1).

2. quicksort Function:
Input:
low: The starting index of the portion to be sorted.
high: The ending index of the portion to be sorted.

Output:None.

Time Complexity : Same as the sort function, O(n log n) on average and O(n^2) in the worst case.

Space Complexity : O(1) since quicksort is an in-place sorting algorithm.
3. partition Function:
Input:
low: The starting index of the portion to be partitioned.
high: The ending index of the portion to be partitioned.

Output:
Returns the index of the pivot element after partitioning.

Time Complexity:O(n), where n is the number of elements in the portion being partitioned.

Space Complexity:O(1) since it operates in-place.

4. Find_location Function:
Input:None.
Output:None.

Time Complexity:
The function involves sorting and iterating through the places array (O(places_count)).

Space Complexity:
O(1) since the function doesn't use additional space beyond local variables.

5. bfss Function:
Input:
i: Unused index
pattern: The pattern to search for.
city: The city name to search within.

Output : Returns 1 if the pattern is found in the city name, 0 otherwise.

Time Complexity:O(n * m), where n is the length of the city name and m is the length of the pattern.

Space Complexity : O(1) since the function uses a constant amount of space.

Function 5: Depth-First Search (DFS) and Minimum Spanning Tree
1. dfs Function:
Input:
node: The starting node for the DFS traversal.
size: The number of nodes in the graph.
Total_distance: A pointer to an integer to store the total distance of the spanning tree.

Output:None.

Time Complexity:The dfs function has a time complexity of O(V + E), where V is the number of vertices (nodes) and E is the number of edges.
In each recursive call, it visits each adjacent node once.

Space Complexity:O(V) for the visited array and recursive call stack.

2. SpanningTree Function:
Input:
size: The number of nodes in the graph.

Output:
None.

Time Complexity:The time complexity is determined by the dfs function.
Overall time complexity is O(V + E), where V is the number of vertices and E is the number of edges.
Space Complexity:
O(V) for the visited array and additional space for variables.

Function 6: Adding New Cities and Routes
1. addPlace Function:
Input:None.
Output:None.

Time Complexity:The time complexity of this function is O(N * M), where N is the number of cities to be added, and M is the average length of city names.

Space Complexity:
O(1) for variables and arrays

Function 7: Adding New Routes
1. addroute Function:
Input:None.
Output:None.

Time Complexity:The time complexity of this function is O(N * M), where N is the number of routes to be added, and M is the average length of city names.

Space Complexity:O(1) for variables and arrays, excluding the space required for the new route details.

Function 8: Adding New Buildings
1. add_building Function:
Input:None.
Output:None.

Time Complexity:The time complexity of this function is O(N * M), where N is the number of buildings to be added, and M is the average length of city names and building names.

Space Complexity:O(1) for variables and arrays, excluding the space required for the new building details.

Function 9: Identifying Important Places
1. important_places Function:
Input:None.
Output:None.

Time Complexity:
The time complexity of this function is O(N * M), where N is the number of places and M is the average length of place names and types.
Iterating through the places array

Space Complexity:O(1) for variables and arrays, excluding the space required for the existing places array.

2. check_for_importance Function:
Input:
rating: The rating string to evaluate.
Output:
int (1 if important, 0 if not important).

Time Complexity:O(1).The function only checks the length of the rating string to determine importance.

Space Complexity:O(1).

Function 10: Displaying Places with a Specific Rating
1. Display_wrt_rating Function:
Input:None.
Output:None.

Time Complexity:O(N * M), where N is the number of places and M is the average length of place names and types.

Space Complexity:O(1) for variables and arrays,

Function 11: Finding Nearby Places
1.bfs Function:

Input:
start_node: The index of the starting city in the 'map' and 'City' arrays.
size: The total number of cities.
distance_limit: The maximum distance to explore from the starting city.

Output:None.

Time Complexity:O(V+E), where V is the number of vertices (cities) and E is the number of edges.

Space Complexity:
O(V) for the 'visited' array.
O(V) for the 'queue' array.

2.findNearbyPlaces Function:

Input:None.
Output:None.

Time Complexity: the time complexity of the bfs function is O(V+E).

Space Complexity:Depends on the space complexity of the bfs function, which is O(V).

Function 12: Route-Finding Process
dijkstra1 Function:
Input:
nn: Number of nodes (cities).
Matrix: Adjacency matrix representing the graph.
sourceNode: Starting place.
destNode: Ending place.
remainingRoutes: Number of remaining routes (recursion depth).
Output:None.

Time Complexity:O(N ^2), where N is the number of cities
The function implements Dijkstra's algorithm with a time complexity of O(N ^2).

Space Complexity:O(N) for the 'dist,' 'path,' and 'visited' arrays.
Recursive calls contribute to the function call stack, potentially reaching a depth of remainingRoutes

Function 13: Nearest Place
nearest_place Function:

Input:None.
Output:None.
Time Complexity:O(V ^3), where V is the number of vertices (cities).The function calls the Floyd Warshall algorithm, which has a time complexity of O(V ^3).

Space Complexity:
O(V ^2) for the 'dist' matrix.

floyd_warshall Function:
Input:
graph: The adjacency matrix representing the graph (distances between vertices).
V: The number of vertices in the graph.
source: The index of the source vertex for which we want to find the nearest vertex.
Output:
int - The index of the nearest vertex to the source.
Time Complexity:O(V ^3) - The triple-nested loop iterates through all vertices to update shortest paths.
Space Complexity:O(V ^2) for the 'dist' matrix - Stores shortest distances between all pairs of vertices.

Function 14: create_map
Input:None.
Output:None.
Time Complexity:O(N+M+P), where N is the number of cities,M is the number of routes, and P is the number of places.

Space Complexity:
O(N+M+P) for arrays storing information about cities, routes, and places.
Additional space is required for the 'City', 'route', and 'places' arrays.

Function 15: calculateAverageRating
Input:None.
Output:None.
Time Complexity:O(P), whereP is the number of places.The function iterates through the 'places' array to calculate the overall average rating.

Space Complexity:O(1) for variables, excluding the space required for the existing 'places' array.

Function 15: calculate_city_avg
Input:None.
Output:None.

Time Complexity:O(P), P is the number of places.The function iterates through the'places' array to calculate the average rating of a specific city.

Space Complexity:O(1) for variables, excluding the space required for the existing 'places' array.

Function 17: check_and_print_connected_cities
Input:None.
Output:None.
Time Complexity:O(P), where P is the number of places.
The function calls the isvalid function, which iterates through the 'City' array to check for the validity of the entered city name.

Space Complexity:O(1) for variables, excluding the space required for the existing 'City' array.

Function 17: print_connected_cities

Input:
cityName: The name of the city to check for connections.
map: The adjacency matrix representing connections between cities.
Output:None.

Time Complexity:O(P ^2), where P is the number of places.
The function initializes and fills a matrix 'dist' based on the adjacency matrix 'map' and then iterates through it to print directly connected cities.

Space Complexity:O(P ^2) for the 'dist' matrix, where P is the number of places.O(1) for other variables, excluding the space required for the existing 'City' array.

6. Conclusion

Optimizing Commuting Routes t was created by integrating a number of clever features. Several algorithms and data structures were used to offer user-friendly search choices, efficient station access, and optimised trip routes
this project has been a valuable learning experience, equipping me with valuable technical skills, problem-solving abilities, and personal growth. I am confident that these takeaways will serve me well in my future both within the field of software development and beyond.

MY LEARNING OUTCOMES:

- Invaluable learning experience translating theory into practical applications.
- Stepped out of comfort zone to master and apply new concepts for project functionalities.
- Realized programming's broader applicability in solving everyday non-programming issues.
- Prioritized maintaining overall code functionality during the addition of new features.
- Encountered and resolved error warnings, turning initial frustrations into a sense of accomplishment.
- Understanding of algorithms and data structures : Implementing efficient algorithms like Dijkstra's and Floyd,Warshall for route optimization solidified my knowledge and boost my practical skills in handling complex data structures like graphs and trees.

7. References

Cite your references.

zeelo.co/blog/efficient-routes-and-budget-friendly-commutes
ask.com/lifestyle/optimizing-commute-time-saving-strategies-go-bus-routes-schedules

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