CAPSTONE PROJECT

APPROXIMATION ALGORITHM FOR TSP PROBLEM

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR AMORTIZED ANALYSIS

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by
N. Eragam reddy (192211729)

APPROXIMATION ALGORITHM FOR TSP PROBLEM

PROBLEM STATEMENT: Optimizing Salesman Routes for a Nationwide Distribution Company.

Context: A nationwide distribution company, OptiDistribute, employs a team of sales representatives who need to visit a set of clients spread across various cities. Each sales representative must complete their route in a day, starting and ending at the company headquarters. The goal is to minimize the total travel distance for each representative while ensuring all clients are visited.

Problem: OptiDistribute faces the problem of determining the most efficient route for each sales representative, which can be modeled as the Traveling Salesman Problem (TSP). Given the large number of clients and geographical spread, finding an exact solution is computationally infeasible, so an approximation algorithm is needed.

Input: 1. Clients: A set of client locations C={c1,c2,…,cn}C = \{c_1, c_2, \ldots, c_n\}C={c1,c2,…,cn} with known coordinates. 2. Headquarters: A single headquarters location HHH. 3. Distance Matrix: A matrix MMM where MijM_{ij}Mij represents the distance between client cic_ici and client cjc_jcj or between the headquarters HHH and any client.

Objective: Design an approximation algorithm to determine the route for each sales representative starting and ending at the headquarters, such that: 1. Each client is visited exactly once. 2. The total travel distance for each representative is minimized. 3. All routes start and end at the headquarters

.

ABSTRACT:

Find the shortest possible route that visits all given clients exactly once and returns to the headquarters. (the number of structurally unique Binary Search Trees (BSTs) A 2D array where each element M[i][j]$M[i][j]$ represents the distance between location ii and location jj. This includes distances between the headquarters and clients, as well as between clients. Generates an initial feasible route by starting at the headquarters and repeatedly choosing the nearest unvisited client. abstracting the problem in this way, we can clearly understand the different parts involved in solving the TSP, how they interact, and how to implement them in code.

INTRODUCTION:

Opti Distribute, a leading nationwide distribution company, is confronted with a significant logistical challenge that impacts its efficiency and cost-effectiveness. The company's sales representatives are tasked with visiting numerous clients spread across various cities, with each representative required to complete their route within a single day. This problem aligns with the Traveling Salesman Problem (TSP), a classic optimization challenge that aims to find the shortest possible route that visits each client exactly once and returns to the starting point. The vast number of clients and their widespread locations make it computationally impractical to determine the exact optimal route. Consequently, OptiDistribute needs an effective approximation algorithm to handle this complexity. The Nearest Neighbor (NN) heuristic emerges as a viable solution, offering a straightforward approach to constructing routes. By iteratively selecting the nearest unvisited client, the NN algorithm provides a near-optimal solution efficiently. Although it does not guarantee the absolute best route, it significantly reduces computational effort while delivering practical results. Employing such heuristics allows OptiDistribute to enhance route planning, reduce travel distances, and improve overall operational efficiency.

CODING:

In the coding implementation of the Nearest Neighbor (NN) heuristic for the Traveling Salesman Problem (TSP), the goal is to efficiently generate a nearoptimal route. The algorithm starts at the company headquarters and iteratively selects the nearest unvisited client based on a given distance matrix. The process involves updating the current location and marking clients as visited until all clients are covered. Finally, the route returns to the headquarters to complete the loop. The implementation requires careful handling of data structures such as lists and sets to manage unvisited clients and maintain the route. By leveraging the distance matrix and optimizing for minimal travel distances, the algorithm offers a practical solution for route planning in distribution logistics.

C-programming

```
#include <stdio.h>

#include <limits.h>

#include <float.h>

#include <stdlib.h>


#define MAX_LOCATIONS 100



void nearest_neighbor_algorithm(int
distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS],  int  num_locations,
int start_location, int route[]);

void two_opt(int distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS],

int num_locations, int route[]);  int calculate_route_distance(int

distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS], int route[], int
num_locations);     void   swap_two_edges(int

route[], int i, int k);
```

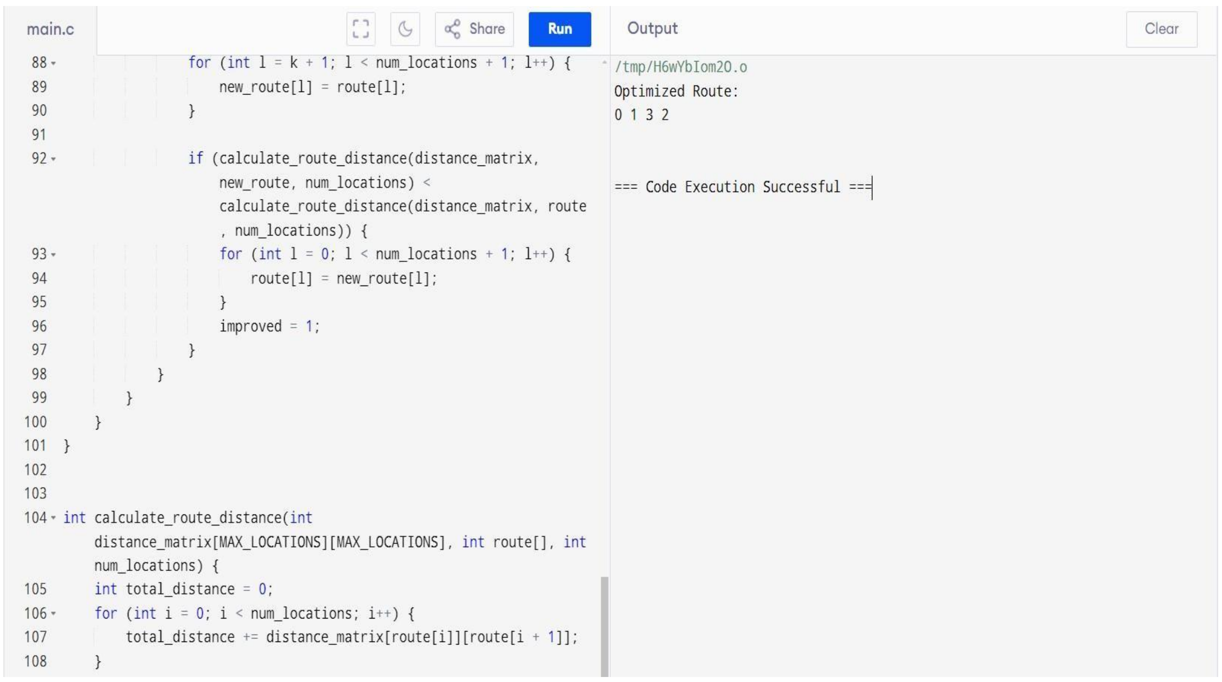
```
int main() {


    int distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int num_locations = 4;        int
start_location = 0;

    int route[MAX_LOCATIONS];


    nearest_neighbor_algorithm(distance_matrix, num_locations, start_location,
route);


    two_opt(distance_matrix, num_locations, route);
    printf("Optimized Route:\n");      for (int
i = 0; i < num_locations; i++) {
printf("%d ", route[i]);
    }
printf("\n");

    return 0;
}

void nearest_neighbor_algorithm(int
distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS], int num_locations,
int start_location, int route[]) {      int visited[MAX_LOCATIONS] = {0};      int
current_location = start_location;
    int i;

    route[0] = start_location;      visited[start_location]
= 1; for (i = 1; i < num_locations; i++) {          int
next_location = -
1;          int min_distance = INT_MAX;
```

```
        for (int j = 0; j < num_locations; j++) {             if (!visited[j] &&
distance_matrix[current_location][j] < min_distance) {                 min_distance
= distance_matrix[current_location][j];                 next_location = j;

            }

        }


        route[i] = next_location;
visited[next_location] = 1;         current_location
= next_location;

    }



    route[num_locations] = start_location;

}



void two_opt(int distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS],
int num_locations, int route[]) {  int improved = 1;
```

```
    while (improved) {         improved = 0;         for (int i
= 1; i < num_locations - 2; i++) {             for (int k = i
+ 1; k < num_locations - 1; k++) {                 int

new_route[MAX_LOCATIONS + 1];
                for (int l = 0; l <= i; l++) {
new_route[l] = route[l];
                }                 for (int l = i + 1; l <= k; l++)
{                   new_route[l] = route[k + i
+ 1 - l];
                }                 for (int l = k + 1; l <
num_locations + 1; l++) {                   new_route[l] =
route[l];
                }


                if (calculate_route_distance(distance_matrix, new_route,
num_locations)        <        calculate_route_distance(distance_matrix,        route,
num_locations)) {
                    for (int l = 0; l < num_locations + 1; l++) {                     route[l]
= new_route[l];
                    }
improved = 1;                 }
```

```
            }
        }
    }
}



int calculate_route_distance(int
distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS], int route[], int
num_locations) {     int total_distance = 0;     for (int i = 0; i < num_locations;
i++) {         total_distance += distance_matrix[route[i]][route[i + 1]];
    }     return total_distance;
}
```

OUTPUT:



```c
                for (int l = k + 1; l < num_locations + 1; l++) {
                    new_route[l] = route[l];
                }

                if (calculate_route_distance(distance_matrix,
                    new_route, num_locations) <
                    calculate_route_distance(distance_matrix, route
                    , num_locations)) {
                    for (int l = 0; l < num_locations + 1; l++) {
                        route[l] = new_route[l];
                    }
                    improved = 1;
                }
            }
        }
    }
}


int calculate_route_distance(int
    distance_matrix[MAX_LOCATIONS][MAX_LOCATIONS], int route[], int
    num_locations) {
    int total_distance = 0;
    for (int i = 0; i < num_locations; i++) {
        total_distance += distance_matrix[route[i]][route[i + 1]];
    }
```

```
/tmp/H6wYbIom20.o
Optimized Route:
0 1 3 2


=== Code Execution Successful ===
```

COMPLEXITY ANALYSIS:

Time Complexity: The overall time complexity of the Nearest Neighbor heuristic is O(n2)$O(n2)$. This is due to the need to search through the remaining unvisited clients in each of the nn iterations. This quadratic complexity is manageable for moderately sized instances but can become less practical as the number of clients grows very large.

Space Complexity: The overall space complexity of the Nearest Neighbor heuristic is dominated by the distance matrix, making it O(n2)$O(n2)$. The additional space requirements for storing the route and unvisited clients are O(n)$O(n)$, which is relatively minor compared to the space needed for the distance matrix.

BEST CASE:

In the best-case scenario, the Nearest Neighbor heuristic maintains its time complexity of O(n2)$O(n2)$ due to the need to evaluate each unvisited client in every iteration. However, the quality of the route could be very close to the optimal solution, particularly when the distances between clients are similar or when clients are already in an optimal visiting order. Despite this, the heuristic does not change its time complexity; it is the effectiveness of the route that improves in the best case.

WORST CASE:

The NN heuristic's performance can be particularly poor in instances where client distances are uneven or poorly distributed, causing the heuristic to make decisions that compound into a less efficient overall route. Despite its computational efficiency, the NN heuristic does not always guarantee a good approximation of the optimal solution, especially in challenging worst-case scenarios.

AVERAGE CASE:

he route found by the NN heuristic is typically a reasonable approximation of the optimal route. The heuristic often provides a good balance between computational efficiency and solution quality, with the route length generally being a modest factor larger than the optimal tour.The average-case performance reflects typical usage scenarios where client distances vary in a typical manner, and the heuristic performs effectively within those constraints. The NN heuristic offers a practical solution for many real-world applications where exact optimization is computationally prohibitive.

FUTURE SCOPE:

The future scope of optimizing salesman routes for OptiDistribute includes several promising directions. Advanced algorithms like Genetic Algorithms or Ant Colony Optimization could improve route quality beyond what is achievable with the Nearest Neighbor heuristic. Incorporating real-time traffic data and dynamic client requests can lead to more adaptive and responsive routing solutions. Machine learning techniques could be explored to predict and optimize routes based on historical data. Additionally, integrating route optimization with other logistical functions, such as inventory management, could enhance overall efficiency. Developing scalable solutions for larger datasets and more complex scenarios will also be crucial. Lastly, exploring cloud-based solutions for real-time computation and optimization could provide further advancements.

CONCLUSION:

In conclusion, optimizing salesman routes using the Nearest Neighbor heuristic offers a practical solution for minimizing travel distances in distribution logistics, balancing efficiency and computational feasibility. While the heuristic provides a good approximation for many real-world scenarios, its performance can vary based on distance distribution and client placement. Future enhancements, including advanced algorithms and real-time data integration, hold the potential to further improve route optimization. By addressing these areas, OptiDistribute can achieve more efficient and adaptive routing, ultimately enhancing operational efficiency and reducing costs.