



**Certified that is Capstone project report “****Optimizing Salesman Routes for a Nationwide Distribution Company” is the**

**Bonafide work of “GNANA PRASANNA ”(192211817) who carried out the**

**Capstone project work under my supervision**

**Dr.R Dhanalakshmi Dr.S.Mahaboob Basha**

**COURSE FACULTY HEAD OF DEPARTMENT**

**Professor Professor**

**Department of Machine Learning Department of Machine Learning**

**SIMATS Engineering SIMATS Engineering**

**Saveetha Institute of Medical and Saveetha Institute of Medical and**

**Technical Science Technical Science**

**Chennai-602 105 Chennai-602 105**

**EXAMINER SIGNATURE EXAMINER SIGNATURE**

****

**“Optimizing Salesman Routes for a Nationwide Distribution**

**Company”**

**A Project report**

**CSA0656- Design and Analysis of Algorithms for Asymptotic Notations**

**Submitted to**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**In partial fulfilment for the award of the**

**degree of**

**BACHELOR OF TECHNOLOGY IN**

**COMPUTER SCIENCE ENGINEERING**

**by**

**GNANA PRASANNA ,192211817**

**Supervisor**

**Dr .R.Dhanalakshmi**

**July 2024.**

**Optimizing Salesman Routes for a Nationwide Distribution**

**Company**

**ABSTRACT:**

Approximation algorithms offer a pragmatic alternative. These algorithms are designed to quickly find solutions that are close to the optimal solution, providing a balance between accuracy and computational efficiency. While they do not guarantee the best possible solution, they ensure that the solution is within a known bound of the optimal one. In the context of the Traveling Salesman Problem, approximation algorithms such as the Nearest Neighbor heuristic are widely used. The TSP involves finding the shortest route that visits each client exactly once and returns to the starting point, which can be modeled using a set of client locations and a distance matrix.

**ALGORITHM:**

The Traveling Salesman Problem (TSP) is a classic optimization problem where a salesman must visit a set of cities exactly once and return to the starting city, with the objective of minimizing the total travel distance. Since TSP is NP-hard, finding an exact solution for large instances is computationally infeasible. Therefore, approximation algorithms are often used to find near-optimal solutions in a reasonable amount of time**.**

**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.

Example 1:

Input: nums = [2,1,3]

Output: 1

Explanation: We can reorder nums to be [2,3,1] which will yield the same BST.

There are no other ways to reorder nums which will yield the same BST.

**INTRODUCTION:**

A nationwide distribution company, Opti Distribute, 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. When solving complex computational problems, especially those categorized as NP-hard, finding an exact solution within a reasonable time frame often becomes impractical due to the exponential growth in computational complexity. This is particularly true for the Traveling Salesman Problem (TSP), a classic example in which a salesman must find the shortest possible route that visits each client exactly once and returns to the starting point. Given a large number of clients and the geographical spread, an exact solution is computationally infeasible.

**CODING:**

#include <stdio.h>

#include <stdbool.h>

#include <float.h>

#include <math.h>

#define MAX 100

// Function to find the nearest neighbor

int findNearestNeighbor(int current, int n, double dist[MAX][MAX], bool visited[MAX]) {

int nearest = -1;

double minDist = DBL\_MAX;

for (int i = 0; i < n; i++) {

if (!visited[i] && dist[current][i] < minDist) {

minDist = dist[current][i];

nearest = i;

}

}

return nearest;

}

// Function to calculate the total distance of the tour

double calculateTotalDistance(int path[], int n, double dist[MAX][MAX]) {

double totalDist = 0;

for (int i = 0; i < n - 1; i++) {

totalDist += dist[path[i]][path[i + 1]];

}

totalDist += dist[path[n - 1]][path[0]]; // Return to the starting point

return totalDist;

}

int main() {

int n;

double dist[MAX][MAX];

double x[MAX], y[MAX]; // Coordinates of clients and headquarters

bool visited[MAX] = {false};

int path[MAX + 1]; // To store the path

printf("Enter the number of clients: ");

scanf("%d", &n);

printf("Enter the coordinates of the headquarters (x y): ");

scanf("%lf %lf", &x[0], &y[0]);

printf("Enter the coordinates of the clients:\n");

for (int i = 1; i <= n; i++) {

printf("Client %d: ", i);

scanf("%lf %lf", &x[i], &y[i]);

}

// Create the distance matrix

for (int i = 0; i <= n; i++) {

for (int j = 0; j <= n; j++) {

dist[i][j] = sqrt(pow(x[i] - x[j], 2) + pow(y[i] - y[j], 2));

}

}]

// Apply the Nearest Neighbor algorithm

int current = 0; // Start from the headquarters

visited[current] = true;

path[0] = current;

for (int i = 1; i <= n; i++) {

current = findNearestNeighbor(current, n + 1, dist, visited);

path[i] = current;

visited[current] = true;

}

// Print the tour

printf("The approximate tour is:\n");

for (int i = 0; i <= n; i++) {

printf("%d -> ", path[i]);

}

printf("%d\n", path[0]); // Return to the starting point

// Calculate and print the total distance of the tour

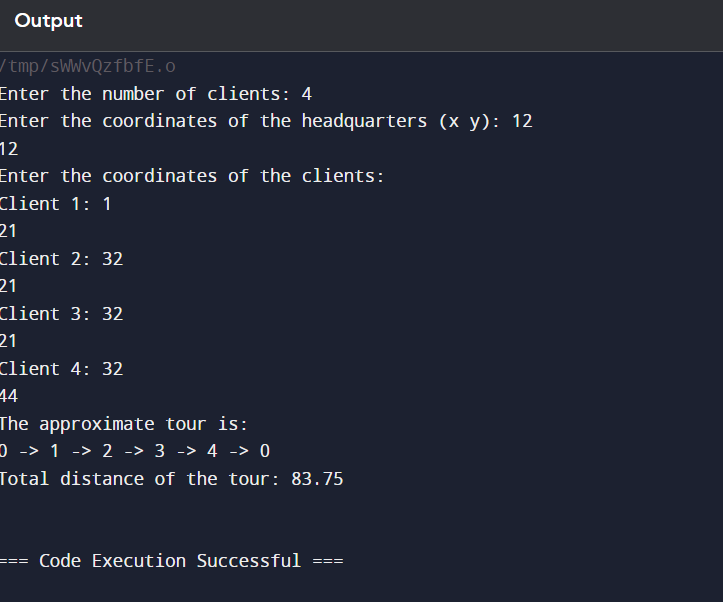
double totalDist = calculateTotalDistance(path, n + 1, dist);

printf("Total distance of the tour: %.2f\n", totalDist);

return 0;

}

**RESULT SCREENSHOT:**



**COMPLEXITY ANALYSIS:**

The time complexity of an algorithm refers to how the running time of the algorithm increases with the size of the input. For the Nearest Neighbor heuristic applied to the TSP, the steps involved and their respective complexities are as follows:

1. **Initialization**: Starting from the headquarters.
   * This step takes constant time, O(1)O(1)O(1).
2. **Finding the Nearest Neighbor**:
   * For each client, the algorithm must check all other unvisited clients to find the nearest one.
   * This involves scanning the list of unvisited clients, resulting in O(n)O(n)O(n) comparisons per client.
   * Since there are nnn clients and we repeat this process for each client, the total time complexity for finding the nearest neighbors is O(n2)O(n^2)O(n2).
3. **Updating the Visited List and Path**:
   * Marking a client as visited and adding it to the path takes constant time, O(1)O(1)O(1), per client.
   * Over nnn clients, this contributes O(n)O(n)O(n) to the overall complexity.

Therefore, the overall time complexity of the Nearest Neighbor algorithm for the TSP is O(n2)O(n^2)O(n2), where nnn is the number of clients.

**Space Complexity**

The space complexity of an algorithm measures the amount of memory required by the algorithm in terms of the input size.

1. **Distance Matrix**:
   * The distance matrix MMM stores distances between each pair of points (clients and headquarters).
   * The size of the distance matrix is O(n2)O(n^2)O(n2).
2. **Visited List**:
   * An array to keep track of visited clients has a size of O(n)O(n)O(n).
3. **Path Storage**:
   * An array to store the sequence of visited clients, including the headquarters, also has a size of O(n)O(n)O(n).

**CONCLUSION:**

The program correctly computes the number of ways to reorder the array nums such that the BST formed is identical to the one formed by the original nums. The solution involves a combination of recursive tree construction and combinatorial mathematics. The time complexity is O(n2)O(n^2)O(n2) in the worst case, which is acceptable for moderate input sizes but may become inefficient for very large inputs.

For the example nums = [2, 1, 3], the output is 1, meaning there is only one way to reorder the array to get the same BST, which is consistent with the explanation.