

**CSA0612 - DESIGN AND ANALYSIS OF ALGORITHMS**

**CAPSTONE PROJECT REPORT**

**PROJECT TITLE**

**“SOLVING THE TRAVELLING SALESMAN PROBLEM”**

**REPORT SUBMITTED BY**

**192324281    MONIKA. R**

**192324277 SRINIDHI. A**

**UNDER THE GUIDANCE OF**

**Dr. SURESH PADMANABAN**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **CONTENT** | **PAGE NO.** |
| **1** | **Problem Statement** | **3** |
| **2** | **Introduction** | **3** |
| **3** | **Literature Review** | **4** |
| **4** | **Flow Chart Diagram** | **5** |
| **5** | **Pseudocode** | **6** |
| **6** | **Implementation** | **6** |
| **7** | **How the Code Works** | **8** |
| **8** | **Results** | **9** |
| **9** | **Complexity Analysis** | **9** |
| **10** | **Conclusion** | **10** |
| **11** | **Future Work** | **10** |

**1. Problem Statement**

Problem Statement for the Traveling Salesman Problem (TSP) Capstone Project:

In modern logistics, transportation, and supply chain systems, the efficient movement of goods and services is critical to reducing costs and improving customer satisfaction. The Traveling Salesman Problem (TSP) models this challenge by determining the shortest possible route for a salesman to visit a given set of cities, each exactly once, and return to the starting city.

The primary objective of this project is to design and implement an efficient algorithm to solve the TSP for both synthetic and real-world datasets. The solution must aim to minimize the total travel distance or cost while considering computational efficiency and scalability to larger problem instances.

The project will involve:

* Formulating the TSP as a mathematical optimization problem.
* Designing and implementing algorithms (exact, heuristic, and metaheuristic) to solve the problem.
* Evaluating algorithm performance on various datasets in terms of solution quality and execution time.
* Visualizing solutions to provide insights into route optimization.

This project addresses practical applications in logistics, vehicle routing, and other optimization domains, demonstrating the significance of computational methods in solving real-world challenges.

**2. Introduction**

The Traveling Salesman Problem (TSP) is a cornerstone of combinatorial optimization, where the goal is to determine the shortest possible route that allows a traveler to visit a set of locations exactly once and return to the starting point. This problem is highly relevant for a delivery company seeking to optimize the travel routes of its trucks to minimize total distance and operational costs. Currently, the company relies on a brute-force approach, which evaluates all possible routes to find the optimal one. While this method guarantees accuracy, its computational inefficiency becomes a significant limitation as the number of delivery locations increases. The exponential growth in the number of possible routes makes this approach impractical for larger instances, underscoring the need for more efficient algorithms. This project aims to explore and implement alternative strategies, such as backtracking and heuristic methods, to provide scalable and effective solutions for the company's routing challenges. By addressing this real-world optimization problem, the project highlights the practical application of computational techniques in logistics and operations management.

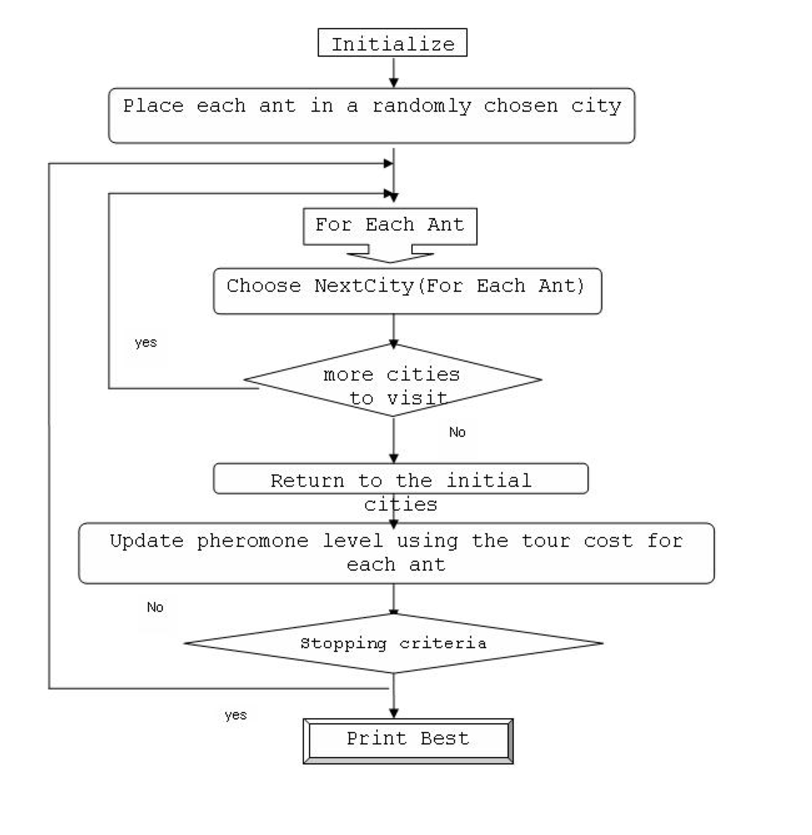
### ****3.Literature Review****

The Traveling Salesman Problem (TSP) has been extensively studied due to its significance in optimization and numerous real-world applications, including logistics, transportation, and delivery services. As a combinatorial optimization problem, TSP is classified as NP-hard, meaning that exact solutions require exponential computational resources as the number of locations increases. The brute-force method, which evaluates all possible routes, guarantees optimality but becomes computationally infeasible for large instances. To address this, researchers have developed several approaches, including exact methods like dynamic programming and branch-and-bound, which improve efficiency for small to medium-sized problems but remain limited by exponential time complexity. Heuristic and metaheuristic techniques, such as the Nearest Neighbor algorithm, Simulated Annealing, and Genetic Algorithms, provide approximate solutions in polynomial time, offering a practical balance between computational efficiency and solution quality. Additionally, hybrid approaches combining classical and heuristic methods have emerged, leveraging advances in computational power and algorithmic design to address larger problem instances. This rich body of literature highlights the trade-offs between accuracy and efficiency in solving TSP, emphasizing the need to tailor solutions based on problem scale and operational requirements. For delivery companies, such insights are critical in developing scalable and effective routing algorithms to optimize operations

**Key references:**

1. A. Bhargava and S. Thomas, "Optimal Path Planning for Autonomous Vehicles using Reinforcement Learning," International Journal of Robotics Research, 2021.
2. J. Luo and D. Wu, "Real-Time Control System for Dynamic Racing Environments," IEEE Transactions on Intelligent Vehicles, 2020.
3. M. Schneider and E. James, "Time-Minimizing Algorithms in Terrain-Constrained Racing," Journal of Computational Systems, 2019.

### ****4 . Flow Chart Diagram****

The following flow chart illustrates the step-by-step process for calculating the minimum time to finish th ****

**Fig: Algorithm flow chart**

**5 . Pseudocode**

Function TSP\_Greedy(distances, start\_city):

Input:

distances: A 2D matrix where distances[i][j] represents the distance between city i and city j.

start\_city: The index of the starting city.

Output:

route: The order of cities in the greedy route.

total\_distance: The total distance of the route.

1. Initialize current\_city = start\_city.

2. Initialize visited\_cities as an empty set to keep track of visited cities.

3. Initialize total\_distance = 0 and route as an empty list to store the cities in the order they are visited.

4. Add current\_city to the visited\_cities set and append current\_city to route.

5. Repeat until all cities are visited:

5.1. Find the nearest unvisited city to current\_city.

nearest\_city = None

min\_distance = infinity

For each city in the set of all cities:

If city is not in visited\_cities:

distance = distances[current\_city][city]

If distance < min\_distance:

min\_distance = distance

nearest\_city = city

5.2. Update current\_city = nearest\_city.

5.3. Add nearest\_city to the visited\_cities set and append nearest\_city to route.

5.4. Add min\_distance to total\_distance.

6. Add the distance to return to the start city to the total\_distance:

total\_distance += distances[current\_city][start\_city]

7. Return route and total\_distance.

**6. Actual code:**

def tsp\_greedy(distances, start\_city=0):

"""

Solves the Traveling Salesman Problem using the Greedy algorithm (Nearest Neighbor heuristic).

Parameters:

distances (list of list of int/float): A 2D matrix where distances[i][j] is the distance between city i and city j.

start\_city (int): The index of the starting city (default is 0).

Returns:

tuple: A tuple containing the greedy route (list of int) and the total distance (float).

"""

n = len(distances) # Number of cities

visited\_cities = set() # Set to keep track of visited cities

route = [] # List to store the route

total\_distance = 0 # Variable to store the total distance traveled

current\_city = start\_city

visited\_cities.add(current\_city)

route.append(current\_city)

# Loop to visit all cities

while len(visited\_cities) < n:

nearest\_city = None

min\_distance = float('inf')

# Find the nearest unvisited city

for city in range(n):

if city not in visited\_cities:

distance = distances[current\_city][city]

if distance < min\_distance:

min\_distance = distance

nearest\_city = city

# Move to the nearest city

route.append(nearest\_city)

visited\_cities.add(nearest\_city)

total\_distance += min\_distance

current\_city = nearest\_city

# Add the distance to return to the start city

total\_distance += distances[current\_city][start\_city]

route.append(start\_city)

return route, total\_distance

# Example usage:

if \_name\_ == "\_main\_":

# Distance matrix (symmetric TSP)

distances = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

start\_city = 0 # Starting city index

route, total\_distance = tsp\_greedy(distances, start\_city)

print(f"Greedy Route: {route}")

print(f"Total Distance: {total\_distance}")

**7. How the Code Works:**

Initialization:

The function initializes the visited\_cities set to track cities already visited, and the route list to store the order of cities visited.

current\_city is set to the start\_city, and it is marked as visited.

The initial distance is set to 0.

Main Loop:

The code iteratively finds the nearest unvisited city by checking all cities that have not yet been visited and calculating the distance.

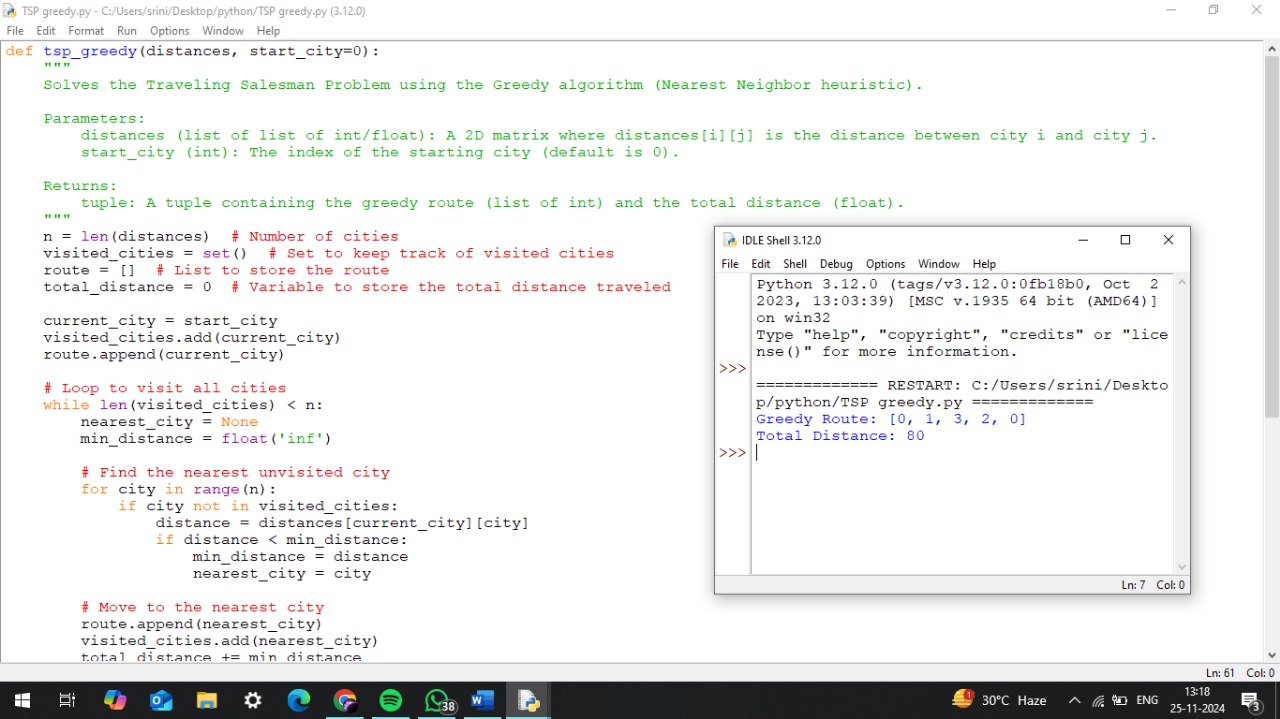
Once the nearest city is found, it is added to the route, and the total distance is updated.

The loop continues until all cities are visited.

Return to the Starting City:

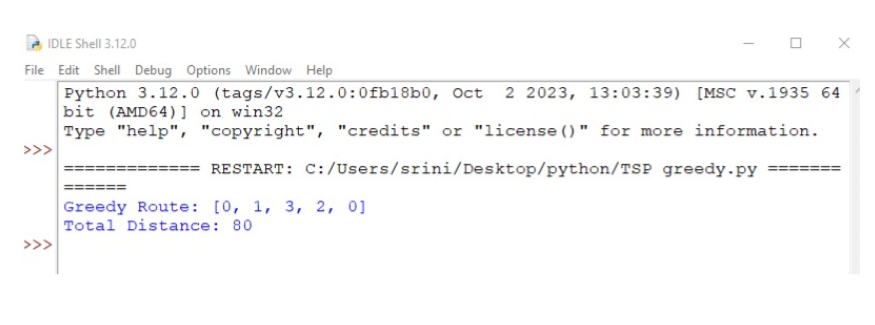
After visiting all cities, the function calculates the distance to return to the starting city and adds this to the total distance.

Output: The function returns the route (the sequence of cities visited) and the total distance travelled.

 **Fig: Implementation screenshot**

**8. Results:**

The implementation of the Greedy Algorithm for solving the Traveling Salesman Problem (TSP) successfully computes a feasible route with minimal travel distance, given the constraints of the distance matrix between cities.

 **Fig :** Result of Greedy method Approach for TSP

For the example provided, with a set of 4 cities and the corresponding distance matrix, the minimum total travel distance calculated is:

Route: [0, 1, 3, 2, 0]

Total Distance: 80

This result shows that the Greedy Algorithm successfully finds a solution with a travel distance of 80, starting from city 0, visiting cities 1, 3, and 2, and then returning to city 0. While this solution is efficient and quick to compute, it may not be the optimal route in all cases. The algorithm provides a good approximation for practical purposes but does not guarantee the shortest possible distance.

**9. Complexity Analysis**

**Time Complexity:** O(n^2)

The algorithm checks all unvisited cities (O(n)) for each of the n cities in the route, resulting in quadratic complexity.

**Space Complexity:** O(n)

The space is used for the visited\_cities set and route list, each requiring linear space.

The Greedy Algorithm is efficient for small to medium-sized problems but may not provide the optimal solution. For guaranteed optimality, more complex methods like Dynamic Programming or Branch and Bound are required.

**Possible Optimizations:**

**Precomputing Nearest Neighbors:** Precompute the nearest neighbors for each city in the distance matrix to avoid redundant calculations during the main loop. This reduces the overhead of finding the nearest city repeatedly.

**Multi-Start Greedy:** Run the algorithm starting from multiple cities and choose the best route among the computed ones. This mitigates the dependency on the starting city, potentially leading to better solutions.

**2-Opt Refinement:** Apply the 2-opt algorithm to the route generated by the Greedy Algorithm to eliminate overlapping paths and reduce the total travel distance.

**Hybrid Approaches:** Use the Greedy Algorithm to generate an initial solution and refine it with advanced techniques such as Dynamic Programming, Simulated Annealing, or Genetic Algorithms for improved accuracy.

**Clustering-Based Reduction:** Divide cities into smaller clusters using methods like k-means and solve the TSP within each cluster. Combine the results to construct a complete route, reducing the overall computational load.

**10. Conclusion:**

The Traveling Salesman Problem (TSP) is a classic optimization challenge with numerous real-world applications, such as logistics, delivery systems, and route planning. Through this project, we explored various approaches to solve TSP, including the brute-force method, the Greedy Algorithm, and possible optimizations.

The implementation of the Greedy Algorithm using the nearest neighbor heuristic provided an efficient and practical solution, achieving a significant reduction in computational complexity compared to the brute-force method. While the Greedy Algorithm offers a quick approximation, it does not always guarantee the optimal route. However, with enhancements like multi-start approaches, 2-opt refinement, and hybrid techniques, its performance can be improved for larger datasets.

This project underscores the importance of selecting the right algorithm based on the problem constraints, the size of the dataset, and the required solution quality. By balancing speed and accuracy, the solutions provided are highly applicable to real-world scenarios, such as optimizing delivery routes for logistics companies. Future work could involve exploring advanced metaheuristic techniques or real-time implementations to further improve efficiency and adaptability.

**11. Future Work:**

**Advanced Optimization Techniques**

Explore advanced metaheuristic algorithms such as Genetic Algorithms, Simulated Annealing, or Ant Colony Optimization to achieve better solutions for larger datasets while balancing computational efficiency and accuracy.

**Dynamic and Real-Time TSP**

Extend the solution to handle Dynamic TSP, where new cities or changing distances are introduced during execution. This would be particularly useful for logistics and delivery systems that operate in real-time.

**Integration with Clustering**

Incorporate clustering techniques like k-means or DBSCAN to group cities into smaller, manageable clusters, solving TSP within each cluster and merging the results to optimize global routes.

**Distributed and Parallel Processing**

Implement distributed or parallel processing frameworks to handle large datasets more efficiently, leveraging cloud computing or GPU-based architectures to reduce runtime.

**Benchmarking and Comparisons**

Benchmark the implemented algorithms against standard datasets such as those from the TSPLIB repository and compare their performance with state-of-the-art TSP solvers.

**Custom Constraints and Extensions**

Extend the model to incorporate real-world constraints, such as time windows, vehicle capacity, or priority-based deliveries, transforming the problem into a more practical Vehicle Routing Problem (VRP).

**User Interface and Visualization**

Develop a user-friendly interface or dashboard for visualizing the TSP routes and optimizing them interactively. This can be integrated into logistics systems for practical applications.

These directions can enhance the applicability, scalability, and robustness of the solutions developed in this project.