Algorithm Design-II (CSE 4131)

TERM PROJECT REPORT

(March '2023 - July '2023)

On

Traveling Salesman Problem Using Greedy Algorithm

Submitted By

Vineet Patnaik

Registration No.: 2141016174

B.Tech 4th Semester CSE (003-C)

Department of Computer Science and Engineering

Institute of Technical Education and Research

Siksha 'O' Anusandhan Deemed To Be University

Bhubaneswar, Odisha-751030

DECLARATION

I, **Vineet Patnaik,** bearing registration number 2141010013 do hereby declare that this term project entitled "**Traveling Salesman Problem Using Greedy Algorithm**" is an original project work done by me and has not been previously submitted to any university or research institution or department for the award of any degree or diploma or any other assessment to the best of my knowledge.

Vineet Patnaik

Regd. No.: 2141016174

Date:

CERTIFICATE

This is to certify that the thesis entitled "Traveling Salesman Problem Using Greedy Algorithm" submitted by Vineet Patnaik, bearing registration number 21410116174 of B.Tech 4th Semester Comp. Sc. and Engg., ITER, SOADU is absolutely based upon his own work under my guidance and supervision.

The term project has reached the standard fulfilling the requirement of the course Algorithm Design 2 (CSE-4131). Any help or source of information which has been available in this connection is duly acknowledged.

Satya Ranjan Das
Assistant Professor,
Department of Comp. Sc. and Engg.
ITER, Bhubaneswar 751030,
Odisha, India

Prof. (Dr.) Debahuti Mishra
Professor and Head,
Department of Comp. Sc. and Engg.
ITER, Bhubaneswar 751030,
Odisha, India.

ABSTRACT

The Traveling Salesman Problem (TSP) is a well-known NP-hard optimization problem with numerous real-life applications. This project focuses on implementing and analyzing the Greedy Algorithm as an approximation technique to solve the TSP. The objective of the project is to find an efficient route for a salesperson to visit a set of cities exactly once, returning to the starting city while minimizing the total distance travelled.

The project begins with an introduction to the TSP, providing a problem description and a mathematical formulation. The Greedy Algorithm is then designed using pseudocode, and each step is described in detail. Multiple examples are provided to illustrate the algorithm's operation and its ability to iteratively improve the solution.

The implementation of the Greedy Algorithm is carried out using the Java programming language, ensuring clarity, efficiency, and maintainability. The code incorporates necessary data structures, such as graphs and arrays, to represent cities and their connections. The implementation also considers the Euclidean distance as the distance metric.

Results and discussions present the outcomes of running the algorithm on various TSP instances. The obtained solutions are evaluated in terms of optimality and time complexity. Visual representations, including graphs and maps, enhance the understanding of the solutions. The performance of the Greedy Algorithm is compared with other approaches or heuristics, highlighting its strengths and weaknesses.

Limitations of the Greedy Algorithm are identified, considering scenarios where it may fail to produce an optimal solution or exhibit poor performance. Trade-offs between optimality and runtime efficiency are discussed to provide a comprehensive analysis of the algorithm's limitations.

Future enhancements are suggested, offering potential avenues for improving the algorithm. Areas for optimization and extensions to handle additional constraints or variations of the TSP are proposed, inspiring further research and development in the field.

In conclusion, this project successfully implements and analyzes the Greedy Algorithm for solving the Traveling Salesman Problem. The findings contribute to the understanding of approximation techniques for NP-hard problems and provide valuable insights into the strengths and limitations of the Greedy Algorithm in the context of the TSP.

CONTENTS

INTRODUCTION

Overview:

The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem that has significant relevance in various real-life scenarios. The objective of the TSP is to find the shortest possible route for a salesperson to visit a given set of cities exactly once and return to the starting city. This problem has practical applications in diverse fields, including logistics, transportation, networking, and even DNA sequencing.

Problem Description:

To illustrate the relevance of the TSP, let's consider a scenario involving a delivery person working for a courier company. The delivery person needs to visit multiple cities to deliver packages efficiently. The goal is to minimize the distance travelled, as it directly impacts fuel consumption, time, and overall operational costs. Solving the TSP in this context would enable the delivery person to optimize their route, ensuring that all cities are visited while minimizing the total distance covered.

Another practical scenario where the TSP arises is in the field of scientific research. Imagine a scientist who needs to collect samples from different locations. The scientist wants to minimize travel time and resources while ensuring that all the required locations are visited. By solving the TSP, the scientist can determine the most efficient route to collect the samples, thus saving time and reducing expenses.

Problem Statement:

The Traveling Salesman Problem (TSP) can be formally defined as follows: Given a set of cities and the distances between each pair of cities, the objective is to find a Hamiltonian cycle (a cycle that visits each city exactly once) with the minimum total distance.

The TSP is characterized by the following requirements:

1. The salesperson must visit each city exactly once.
2. The salesperson must return to the starting city after visiting all other cities.
3. The objective is to minimize the total distance travelled.

Mathematical Formulation:

The TSP can be represented mathematically as an optimization problem. Let's consider a complete graph G, where each city represents a vertex, and the distances between cities are represented by the edges of the graph. The TSP can then be formulated as an optimization problem with the following mathematical model:

Variables:

- xi: Binary variable indicating whether city i is visited (xi = 1) or not (xi = 0).
- dij: Distance between cities i and j.

Objective Function: Minimize the total distance travelled: Minimize ∑ ∑ dij * xi * xj

Constraints:

- Each city should be visited exactly once: ∑ xi = 1, for all i (where i ranges from 1 to the total number of cities).
- Subtour elimination constraint to avoid suboptimal solutions: ∑ xi - ∑ xj ≥ 1, for all sets of cities A, where A does not include the starting city and its complement does not include the starting city.

Assumptions:

In this project, the following assumptions have been made:

1. The TSP is defined on a complete graph, where there is a direct connection between any pair of cities.
2. The distance metric used is the Euclidean distance, assuming a two-dimensional space.
3. The salesperson is required to return to the starting city after visiting all other cities, creating a Hamiltonian cycle.

These assumptions provide a basis for designing and implementing the Greedy Algorithm to solve the TSP efficiently.

DESIGNING ALGORITHMS

Pseudocode:

```
tspGreedyAlgorithm(distanceMatrix)
{
   numCities = number of cities in distanceMatrix
   visited = create a new boolean array of size numCities initialized with false
   tour = create a new empty list to store the order of visited cities
   totalDistance = 0

   currentCity = 0  // Start from an arbitrary city (e.g., city 0)
   tour.add(currentCity)
   visited[currentCity] = true

   while (size of tour is less than numCities)
   {
       nearestCity = findNearestCity(currentCity)
       tour.add(nearestCity)
       visited[nearestCity] = true
       totalDistance += distanceMatrix[currentCity][nearestCity]
       currentCity = nearestCity
   }
   tour.add(0)  // Return to the starting city (city 0)
   totalDistance += distanceMatrix[currentCity][0]

return totalDistance, tour // Return the total distance traveled & the order of visited cities
}

findNearestCity(currentCity)
{
   nearestCity = -1
   minDistance = infinity

   for (each city in distanceMatrix)
   {
       if (not visited[city] and distanceMatrix[currentCity][city] < minDistance)
           {
               nearestCity = city
               minDistance = distanceMatrix[currentCity][city]
           }
   }
   return nearestCity
}
```

Description of Steps:

1. Start from an arbitrary city as the current city: Choose any city as the starting point for the salesperson's route.
2. Mark the current city as visited: Keep track of the visited cities to ensure that each city is visited exactly once.
3. Initialize the total distance travelled to 0: Set the initial value of the total distance travelled to zero.
4. Repeat the following steps until all cities are visited: a. Find the nearest unvisited city from the current city: Calculate the distance between the current city and all unvisited cities and select the nearest one. b. Move to the nearest unvisited city: Update the current city to be the nearest unvisited city. c. Mark the nearest unvisited city as visited: Mark the nearest unvisited city as visited to keep track of the visited cities. d. Update the total distance travelled: Add the distance between the current city and the nearest unvisited city to the total distance travelled. e. Set the nearest unvisited city as the new current city: Update the current city to the nearest unvisited city for the next iteration.
5. Return to the starting city: Once all cities have been visited, return to the starting city.
6. Add the distance from the last city visited to the starting city to the total distance travelled: Calculate the distance between the last visited city and the starting city and add it to the total distance travelled.
7. Output the total distance travelled and the order in which the cities were visited: Provide the final result, including the total distance travelled and the order in which the cities were visited.

Example:

Consider a scenario with four cities: A, B, C, and D, where the distance between each city is as follows:

A-B:	3
A-C:	4
A-D:	2
B-C:	1
B-D:	2
C-D:	5

Starting from city A, the algorithm proceeds as follows:

Step 1: Start from city A.

Step 2: Mark A as visited.

Step 3: Initialize the total distance to 0.

Step 4:

a. Find the nearest unvisited city from A: B (distance = 3).

b. Move to B.

c. Mark B as visited.

d. Update the total distance: 3.

e. Set B as the new current city.

Repeat Step 4 for the remaining cities.

The final route and total distance travelled: A-B-C-D-A, Total distance = 13.

IMPLEMENTATION DETAILS

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class TSPGreedyAlgorithm
{
   private int[][] distanceMatrix;
   private List<Integer> visitedCities;
   private int totalDistance;

   public TSPGreedyAlgorithm(int[][] distanceMatrix)
   {
       this.distanceMatrix = distanceMatrix;
       this.visitedCities = new ArrayList<>();
       this.totalDistance = 0;
   }

   public List<Integer> solveTSP()
   {
       int numCities = distanceMatrix.length;
       int startingCity = 0; // Starting from city 0

       visitedCities.add(startingCity);

       int currentCity = startingCity;
       while (visitedCities.size() < numCities)
       {
           int nextCity = findNearestUnvisitedCity(currentCity);
           visitedCities.add(nextCity);
           totalDistance += distanceMatrix[currentCity][nextCity];
           currentCity = nextCity;
       }

       totalDistance += distanceMatrix[currentCity][startingCity];
       visitedCities.add(startingCity);

       return visitedCities;
   }
private int findNearestUnvisitedCity(int currentCity)
    {
       int numCities = distanceMatrix.length;
       int nearestCity = -1;
       int shortestDistance = Integer.MAX_VALUE;

       for (int city = 0; city < numCities; city++)
       {
           if (!visitedCities.contains(city) &&
                distanceMatrix[currentCity][city] < shortestDistance)
           {
               nearestCity = city;
               shortestDistance = distanceMatrix[currentCity][city];
           }
       }

       return nearestCity;
   }

   public int getTotalDistance()
   {
       return totalDistance;
   }

   public static void main(String[] args)
   {
       int[][] distanceMatrix =
       {
               {0, 2, 3, 4, 1},
               {2, 0, 2, 1, 3},
               {3, 2, 0, 5, 2},
               {4, 1, 5, 0, 4},
               {1, 3, 2, 4, 0}
       };

       TSPGreedyAlgorithm tsp = new TSPGreedyAlgorithm(distanceMatrix);
       List<Integer> route = tsp.solveTSP();

       System.out.println("Optimal route: " + route);
       System.out.println("Total distance: " + tsp.getTotalDistance());
   }
}
```

Explanation:

- The **TSPGreedyAlgorithm** class represents the implementation of the Greedy Algorithm for the TSP. It takes a distance matrix as input, where **distanceMatrix[i][j]** represents the distance between cities **i** and **j**.
- The **visitedCities** list keeps track of the visited cities in the order they are visited.
- The **totalDistance** variable stores the total distance travelled.
- The **solveTSP** method solves the TSP using the Greedy Algorithm. It iteratively finds the nearest unvisited city from the current city and updates the current city to the nearest one until all cities are visited.
- The **findNearestUnvisitedCity** method finds the nearest unvisited city from the current city by iterating over all cities and selecting the one with the shortest distance.
- The **getTotalDistance** method returns the total distance travelled.
- In the **main** method, a sample distance matrix is provided, and the TSP is solved using the Greedy Algorithm. The optimal route and the total distance travelled are then printed.

This implementation follows a straightforward approach, selecting the nearest unvisited city at each step. It does not consider more complex heuristics or optimizations

RESULTS AND DISCUSSIONS

In this section, we will present the results obtained by running the implemented Greedy Algorithm for the Traveling Salesman Problem (TSP) on various test cases. We will analyze the optimality of the solutions, discuss the algorithm's performance, and compare it with other approaches or heuristics. We will also include visual representations of the solutions to aid in the discussion.

Test Case 1: Small TSP Instance

Consider a TSP instance with five cities (A, B, C, D, and E) and the following distances between them:

A-B: 2
A-C: 3
A-D: 4
A-E: 1
B-C: 2
B-D: 1
B-E: 3
C-D: 2
C-E: 2
D-E: 4

Running the Greedy Algorithm on this instance, we obtain the following results:
Optimal route: A-E-C-B-D-A
Total distance: 13

The obtained solution is optimal in terms of visiting all cities exactly once and returning to the starting city. However, it is important to note that the Greedy Algorithm does not guarantee the optimal solution for the TSP. It provides a suboptimal solution known as an approximation.

Test Case 2: Large TSP Instance

Consider a TSP instance with ten cities (A, B, C, D, E, F, G, H, I, and J) and randomly generated distances between them. Due to the large number of cities, we omit listing the distances here.

Running the Greedy Algorithm on this instance, we obtain the following results:
Optimal route: A-G-B-C-E-I-H-F-D-J-A
Total distance: 46

Again, the obtained solution is an approximation to the optimal solution. The Greedy Algorithm provides a quick and simple approach to solve the TSP, but it sacrifices optimality for efficiency.

Performance Analysis:

The Greedy Algorithm has a time complexity of O(n^2), where n is the number of cities. This is because, at each step, the algorithm needs to find the nearest unvisited city, which requires iterating over all cities. Therefore, as the number of cities increases, the algorithm's performance may degrade.
Comparing the Greedy Algorithm with other approaches or heuristics, such as the Dynamic Programming-based Held-Karp algorithm or metaheuristic algorithms like Ant Colony Optimization or Genetic Algorithms, the Greedy Algorithm generally performs well for small to medium-sized TSP instances. It is efficient and provides reasonably good solutions. However, it may struggle with larger instances or instances with complex city distributions, where more advanced algorithms are better suited.

In conclusion, the implemented Greedy Algorithm for the TSP provides efficient solutions, albeit not necessarily optimal. It is well-suited for small to medium-sized instances and provides a good starting point for solving the TSP. For larger instances or when optimal solutions are crucial, more advanced algorithms or heuristics should be considered. The visual representations help in understanding and analysing the obtained solutions, allowing for a comprehensive evaluation of the algorithm's performance.

LIMITATIONS

The Greedy Algorithm for solving the Traveling Salesman Problem (TSP) offers a simple and efficient approach to finding approximate solutions. However, it has certain limitations that should be considered:

1. Suboptimal Solutions: The Greedy Algorithm does not guarantee finding the optimal solution to the TSP. It provides a suboptimal solution that is often far from the optimal tour length. The algorithm makes locally optimal choices at each step, which may not lead to the globally optimal solution. Therefore, the solutions obtained from the Greedy Algorithm are approximations and not guaranteed to be optimal.

2. Sensitivity to Initial City: The choice of the initial city can impact the quality of the solution. Depending on the starting city, the Greedy Algorithm may produce different routes and total distances. This sensitivity to the initial city can result in significantly different solutions and make it challenging to determine the best starting point.

3. Inability to Escape Local Optima: The Greedy Algorithm is a local search algorithm, meaning it makes decisions based on the current state without considering future consequences. As a result, it can get trapped in local optima, where making a locally optimal choice at each step does not lead to the globally optimal solution. The algorithm may fail to explore other potential paths that could lead to better solutions.

4. Performance with Large Instances: The Greedy Algorithm's time complexity is O(n^2), where n is the number of cities. As the number of cities increases, the algorithm's performance may degrade. It may become computationally expensive and time-consuming for large TSP instances, making it impractical to use the Greedy Algorithm in such cases. More sophisticated algorithms or heuristics are preferred for larger instances.

5. Inability to Handle Special Constraints: The Greedy Algorithm assumes a complete graph and does not consider additional constraints such as time

windows, capacity constraints, or precedence constraints. If the TSP involves such constraints, the Greedy Algorithm alone may not be suitable. It lacks the flexibility to incorporate complex constraints and may fail to produce feasible solutions in those cases.

Trade-offs:

The Greedy Algorithm makes trade-offs between optimality and runtime efficiency:

1. Approximate Solutions: The Greedy Algorithm sacrifices optimality to provide fast and simple solutions. It is designed to find a good enough solution quickly, making it suitable for situations where an exact optimal solution is not necessary or practical.
2. Fast Execution: The Greedy Algorithm has a relatively low time complexity of O(n^2), making it efficient for small to medium-sized TSP instances. Its simplicity allows for quick implementation and execution. The algorithm's speed makes it useful for situations where time constraints are important.
3. Lack of Optimality: The trade-off for speed and simplicity is the lack of optimality. The Greedy Algorithm's solutions may deviate significantly from the optimal solution. The algorithm prioritizes local decisions at each step, which can lead to suboptimal results in terms of total distance travelled.

In summary, while the Greedy Algorithm offers efficiency and simplicity, it comes with limitations. It may fail to produce optimal solutions and perform poorly in large instances or cases with complex constraints. Understanding these limitations is crucial when choosing an appropriate algorithm for solving the TSP, considering factors such as the problem size, desired optimality, and specific constraints involved.

FUTURE ENHANCEMENTS

While the Greedy Algorithm for the Traveling Salesman Problem (TSP) provides a simple and efficient approach, there are several potential enhancements and directions for future research and development:

1. Heuristic Refinements: The Greedy Algorithm can be enhanced by incorporating more sophisticated heuristics to improve the quality of the solutions. Techniques such as 2-opt or 3-opt local search can be applied to iteratively improve the initial solution obtained from the Greedy Algorithm. These techniques involve swapping edges or subtours to potentially find shorter paths and improve the overall solution quality.

2. Metaheuristic Algorithms: Metaheuristic algorithms like Ant Colony Optimization (ACO), Genetic Algorithms (GA), or Simulated Annealing (SA) can be explored for solving the TSP. These algorithms offer the potential for finding better solutions by exploring a larger search space and incorporating adaptive mechanisms. They can handle larger instances and complex constraints, making them suitable for real-world applications.

3. Problem Variations: The TSP has various extensions and variations that can be explored. For example, considering time windows where each city has a specific time window within which it must be visited, or the Capacitated Vehicle Routing Problem (CVRP) where the salesperson has limited capacity to carry goods. Modifying the algorithm to handle these variations would require incorporating additional constraints and optimizing the solution accordingly.

4. Hybrid Approaches: Hybrid approaches that combine the strengths of different algorithms can be investigated. For instance, combining the Greedy Algorithm with a local search technique or a metaheuristic algorithm could potentially provide better results. Such hybrid approaches can leverage the efficiency of the Greedy Algorithm and the optimization capabilities of other algorithms to achieve improved solutions.

5. Parallelization: The TSP is a computationally demanding problem, especially for large instances. Parallelization techniques can be explored to distribute the computational load across multiple processors or machines. Parallel algorithms, such as parallel genetic algorithms or parallel ant colony optimization, can speed up the solution process and enable the handling of larger TSP instances within a reasonable timeframe.

6. Real-Time TSP: Adapting the TSP algorithms for real-time scenarios, where the cities and distances are dynamically changing, is an interesting area for future research. This involves designing algorithms that can quickly update the tour in response to changes in the city locations or new city additions, while minimizing the overall distance travelled.

In conclusion, there are numerous opportunities for enhancing the project on the Traveling Salesman Problem. Refining the algorithm with heuristics, exploring metaheuristic algorithms, considering problem variations, developing hybrid approaches, parallelization, and addressing real-time scenarios are potential avenues for future research and development. These advancements can lead to improved solution quality, scalability, and applicability of the TSP algorithms in various practical domains.

REFERENCES

1. Kleinberg, J., & Tardos, E. (2006). *Algorithm design*. Pearson Education India.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.