Table of Contents

1 Introduction:

The purpose of this project is to solve the Half Traveling Salesman Problem (Half TSP) by designing and implementing an algorithm that finds a tour which visits exactly half of the cities and returns back to the origin city. The Half TSP is a variant of the Traveling Salesman Problem (TSP) where the salesman needs to visit each city exactly once, but only half of the cities are selected for the tour.

In this report, we will discuss the design and implementation of our algorithm for solving the Half TSP. We will also describe the process of running the code and analyzing the results.

1.1 Designing the Experiment

To design our algorithm, we started by studying approximation algorithms and local search heuristics commonly used for solving TSP-like problems. We focused on the simulated annealing algorithm, which is known for its effectiveness in finding near-optimal solutions.

Based on our research, we identified the following key steps for solving the Half TSP:

1. Reading the input file and extracting the city coordinates.
2. Calculating the distances between each pair of cities using the Euclidean distance formula.
3. Generating an initial tour by randomly selecting half of the cities.
4. Applying the simulated annealing algorithm to improve the tour.
5. Writing the final tour and its length to an output file.

1.2 Approach and Algorithm Design

To tackle the Half TSP, we researched various approximation algorithms and local search heuristics commonly used for solving TSP-like problems. This allowed us to gain insights into existing techniques and develop our own approach.

Our algorithm follows a simulated annealing approach, which is known for its effectiveness in finding near-optimal solutions for combinatorial optimization problems. The main steps of our algorithm are as follows:

Input Processing: We read the input file, which contains the city coordinates, and store them as a list of cities.

Initial Tour Generation: We randomly select half of the cities to form an initial tour.

Simulated Annealing: We apply the simulated annealing algorithm to improve the initial tour. The algorithm iteratively explores neighboring tours by randomly swapping two cities and accepts or rejects the new tour based on the acceptance probability, which depends on the temperature and the difference in tour lengths.

Output Generation: Once the algorithm converges or reaches a stopping criterion, we write the final tour and its length to the output file, following the specified output format.

We implemented our algorithm in Python programming language. The code consists of several functions that handle different tasks, such as reading the input file, calculating distances, generating neighbor tours, and writing the output file.

To run the code, the user needs to provide the input file containing the city coordinates. The algorithm then processes the input, applies the simulated annealing algorithm, and generates an output file with the final tour.

2 General Description

2.1 Problem Statement

The Half TSP problem aims to find the shortest possible route that visits exactly half of the cities and returns to the origin city. Given the city coordinates, the goal is to minimize the total distance traveled.

2.2 Algorithm Overview

Our algorithm utilizes a local search heuristic to approximate the optimal solution for the Half TSP problem. The algorithm starts with an initial tour and iteratively explores the neighborhood of the current tour by generating neighboring tours through pairwise city swaps. It accepts a neighbor tour if it improves the tour length or based on a probability computed using the difference in tour length and a temperature parameter. The process continues until a termination condition is met.

2.3 Initialization

The algorithm begins by reading the city coordinates from the input file and storing them in a suitable data structure. It initializes the current tour by selecting the first half of the cities in a sequential manner, forming the initial tour.

2.4 Local Search Heuristic

The local search heuristic is employed to improve the current tour iteratively. In each iteration, two random cities from the current tour are selected, and their positions are swapped to create a new neighboring tour. The tour length is computed for the new tour, and a decision is made to accept the new tour based on the improvement in tour length or the acceptance probability.

2.5 Termination Condition

The algorithm continues the local search process until a termination condition is met. This condition can be a predefined temperature threshold or a maximum number of iterations. The termination condition ensures that the algorithm explores the search space adequately without excessive computation.

2.6 Complexity Analysis

The time complexity of the algorithm depends on the number of iterations in the local search process and the problem size. As the problem size increases, the execution time will also increase due to the combinatorial nature of the TSP problem.

2.7 Performance Evaluation

To evaluate the algorithm's performance, experiments were conducted using different problem instances with varying city numbers. The execution time and solution quality were measured by comparing the obtained tour lengths to the optimal tour lengths when available. The trade-off between solution quality and computation time was analyzed to assess the algorithm's effectiveness.

3 Decisions Variables

3.1 Choice of Programming Language: Python

Python was chosen as the programming language for its simplicity, readability, and versatility. Python offers a clean and intuitive syntax, making it easier to write and understand code. Additionally, it has a large and active community, which provides extensive support and resources for problem-solving. Python also has a wide range of built-in data structures and libraries, which simplifies the implementation of complex algorithms. These factors made Python a suitable choice for developing the solution to the Half TSP problem.

3.2 Algorithm Selection: Simulated Annealing

Simulated Annealing (SA) is a search algorithm used to solve global optimization problems. The algorithm gets its name from the process of cooling and heating found in nature. As SA searches for a solution, it 'moves' randomly throughout the search space, and this mobility is subject to a particular cooling schedule (that is, the 'temperature' decreases over time). Initially, the SA algorithm is allowed to 'roam' widely in its search for a solution. However, as time progresses, the search narrows increasingly towards an optimal solution.

There are several reasons why we chose this algorithm:

Global Optimization: SA tends to perform well in global optimization problems because it is capable of finding both local minima and the global minimum. This is important for a problem like Half TSP where we're trying to find its shortest path, which is an optimization problem.

Flexibility: The SA algorithm is adaptable to various problems and is effective in solving this one. This flexibility means we can solve TSP problems of different dimensions and different numbers of cities.

Simplicity and Applicability: The SA algorithm is relatively simple to understand and apply. This allows it to be implemented quickly and for its results to be observed.

Diversity: The SA algorithm uses randomness in its search for a solution. This encourages diversity in the solution space and reduces the risk of getting stuck in local minima.

For these reasons, we chose to use the Simulated Annealing algorithm in solving the Half TSP problem.

4 Implementation

4.1 Functions

- **read_cities(filename):** This function reads city coordinates from a file. It opens the file, reads each line, converts the line to a list of integers representing the city's ID, x-coordinate, and y-coordinate. It appends each city to a list and finally returns the list of cities.
- **dist(city1, city2):** This function calculates the Euclidean distance between two cities based on their coordinates. It subtracts the x-coordinates and y-coordinates of the two cities, squares the differences, adds them together, and takes the square root of the result. The calculated value represents the distance between the two cities.
- **tour_length(tour, cities):** This function computes the total length of a given tour. It takes a tour (a list of city IDs) and the list of cities. It iterates over the tour, calculates the distance between each consecutive pair of cities using the dist function, and sums up the distances. The final result is the total length of the tour.
- **generate_neighbour(tour):** This function generates a new tour by swapping the positions of two randomly selected cities in the current tour. It takes the current tour, selects two random indices, and swaps the cities at those indices in a copy of the tour. The new tour is returned.
- **write_tour_to_txt(tour, cities, filename):** This function writes the given tour and its length to a text file. It opens the file in write mode, writes the tour length as the first line, and then writes each city ID of the tour on separate lines.
- **checksolution(cities, value, cityorder):** This function verifies the correctness of the solution. It takes the list of cities, the expected tour length (value), and the tour order (cityorder). It performs several checks:

It ensures that the tour contains exactly half of the cities.

It checks for duplicate cities in the tour.

It checks if the city IDs are valid (within the range of the number of cities).

It calculates the length of the tour given by the cityorder.

It compares the computed tour length with the expected value and prints a verification message.

- **main():** This is the main function that serves as the entry point of the program. It defines a list of input-output file pairs. For each pair, it reads the cities from the input file using read_cities, initializes the current tour, and sets the parameters for simulated annealing. It then executes the simulated annealing algorithm, updates the tour based on temperature and length comparisons, and writes the resulting tour to the output file using write_tour_to_txt. Finally, it calls checksolution to verify the solution's correctness.

4.2 Sample Outputs in Different Machines

First Machine:

```
PS C:\Users\Enes\Desktop\algo proje> & C:/Users/Enes/AppDat
Execution time for test-input-1.txt: 6.266300916671753
Your solution is VERIFIED.
Execution time for test-input-2.txt: 24.22670269012451
Your solution is VERIFIED.
Execution time for test-input-3.txt: 724.9062869548798
Your solution is VERIFIED.
Execution time for test-input-4.txt: 64.56407809257507
Your solution is VERIFIED.
PS C:\Users\Enes\Desktop\algo proje>
```

Second Machine:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
[Running] python -u "c:\Users\kujk\Desktop\algoproject\main.py"
Execution time for example-input-1.txt: 0.9852004051208496
Your solution is VERIFIED.
Execution time for example-input-2.txt: 3.3605637550354004
Your solution is VERIFIED.
Execution time for example-input-3.txt: 193.9229612350464
Your solution is VERIFIED.

[Done] exited with code=0 in 198.38 seconds
```