MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING

CSE2246

Analysis of Algorithms

2-TSP

Project Report File

Ayşenur Tüfekçi - 150119699

Muhammet Görkem Gülmemiş - 150119785

Mert Efe Karaköse - 150119805

Introduction

The Two Travelling Salesman Problem (2-TSP) is a variation of the classic Travelling Salesman Problem (TSP). In the 2-TSP, two salesmen start from an initial city and must visit a set of cities such that each city is visited exactly once by either of the salesmen. The goal is to minimize the total distance traveled by both salesmen. This project aims to solve the 2-TSP using the A* search algorithm, which leverages heuristics to efficiently find the shortest path.

Methodology

1. Distance Calculation

The Euclidean distance between each pair of cities is calculated using the formula:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

These distances are stored in a distance matrix for quick lookup during the A* search.

2. A* Algorithm Implementation

The A* algorithm is used to find the shortest paths. The key components of the A* algorithm include:

- Open Set: A priority queue that stores nodes to be evaluated.
- Closed Set: A set of nodes that have already been evaluated.
- Heuristic Function: An estimate of the cost to reach the goal from a given node. The heuristic function used is the minimum distance to any unvisited city.

Detailed Explanation of Each Function

is_city_visited(city, visited_cities)

- Purpose: Checks if a city has been visited.
- Parameters:
    - **city**: The city to check.
    - **visited_cities:** A list indicating whether each city has been visited.

```
def is_city_visited(city, visited_cities):
    return visited_cities[city] == 1
```

find_min_distance_city(last_city, visited_cities, distance_matrix, heuristic_values, total_cities)

- Purpose: Finds the next city with the minimum distance and heuristic value.
- Parameters:
  - **last_city:** The last city visited.
  - **visited_cities:** A list indicating whether each city has been visited.
  - **distance_matrix:** A matrix of distances between cities.
  - **heuristic_values:** A list of heuristic values for each city.
  - **total_cities:** The total number of cities.
- Returns: The city with the minimum distance and heuristic value.

```
def find_min_distance_city(last_city, visited_cities, distance_matrix, heuristic_values, total_cities):
    min_distance = float('inf')
    min_distance_city = -1
    for i in range(total_cities):
        if i != last_city and not is_city_visited(i, visited_cities):
            distance = distance_matrix[last_city, i]
            h_value = heuristic_values[i]
            f_value = distance + h_value
            if f_value < min_distance:
                min_distance = f_value
                min_distance_city = i
    return min_distance_city
```

a_star_algorithm(start_city, visited_cities, visited_cities_flags, distance_matrix, heuristic_values, salesman_city_count, total_cities)

- Explanation:
  - This function is the core of the algorithm, finding the optimal path for a salesman.
  - It initializes by marking the start city as visited.
  - It uses a loop to find and visit the next city based on the minimum distance and heuristic value until all required cities are visited.
  - It updates the visited cities and flags lists to reflect the path taken by the salesman.
  - The function ensures that the salesman visits each city exactly once and in the most efficient order.

```
def a_star_algorithm(start_city, visited_cities, visited_cities_flags, distance_matrix, heuristic_values, salesman_city_count, total_
    visited_count = 1
    current_city = start_city
    visited_cities[0] = current_city
    visited_cities_flags[current_city] = 1
    while visited_count < salesman_city_count:
        next_city = find_min_distance_city(current_city, visited_cities_flags, distance_matrix, heuristic_values, total_cities)
        visited_cities[visited_count] = next_city
        visited_cities_flags[next_city] = 1
        visited_count += 1
        current_city = next_city
```

main()

- The **main** function orchestrates the entire process of solving the 2-TSP.
- It starts by recording the start time to measure execution time.
- It reads city coordinates from input files and stores them in a list.
- It calculates the distance matrix for all city pairs.
- It determines heuristic values for each city to be used in the A* algorithm.
- It executes the A* algorithm separately for both salesmen, ensuring all cities are visited efficiently.
- It writes the results to output files in the required format.
- Finally, it calculates and prints the total execution time.

```python
def main():
    start_time = time.time()

    input_files = ['test-input-1.txt','test-input-2.txt','test-input-3.txt', 'test-input-4.txt']
    output_files = ['test-output-1.txt','test-output-2.txt','test-output-3.txt', 'test-output-4.txt']
    max_rows = 50000

    for input_file, output_file in zip(input_files, output_files):
        # Initialize city coordinates array
        city_coordinates = np.zeros((max_rows, 3), dtype=float)
        total_cities = 0

        # Read city coordinates from the input file
        with open(input_file, 'r') as file:
            for line in file:
                if total_cities < max_rows:
                    city_data = list(map(float, line.split()))
                    if len(city_data) == 3:
                        city_coordinates[total_cities] = city_data
                        total_cities += 1
                    else:
                        print(f"Warning: Invalid city data format in line: {line.strip()}")

        # Calculate distance matrix for all cities
        distance_matrix = np.zeros((total_cities, total_cities), dtype=int)
        for i in range(total_cities):
            for j in range(total_cities):
                dx = city_coordinates[i, 1] - city_coordinates[j, 1]
                dy = city_coordinates[i, 2] - city_coordinates[j, 2]
                distance = int(round(math.sqrt(dx ** 2 + dy ** 2)))
                distance_matrix[i, j] = distance
```

Division of Labor

Ayşenur Tüfekci: Input Handling and Distance Calculation, A* Algorithm Core Logic, Testing and Debugging

Mert Efe Karaköse: Heuristic Function and Next City Selection, Algorithms Optimization for Speed Up, A* Integration

Muhammet Görkem Gülmemiş: find_min_distance_city Implementation, A* Algorithm Development, Verification