**Deadline:08.06.2023**


# Marmara University Faculty of Engineering

## CSE 2046 Analysis of Algorithms, Spring 2023

## Homework 2 Report

## Group Members:

| Name-Surname | Student ID | Contribution |
|---|---|---|
| Serkan Korkut | 150119036 | We did it together from the discord application. |
| Şule Koca | 150119057 | We did it together from the discord application. |
| Ömer Faruk Kışlakçı | 150119689 | We did it together from the discord application. |
| Bihter Nilüfer Akdem | 150119810 | We did it together from the discord application. |

**Instructor:**
Doç.Dr Ömer Korçak

**Solving the Half TSP Problem using Heuristic Approaches**

Abstract:
The Half Traveling Salesman Problem (Half TSP) is a variant of the well-known Traveling Salesman Problem (TSP), where the objective is to find the shortest possible route that visits exactly half of the cities and returns to the starting city. This project aims to develop an algorithmic solution for the Half TSP using heuristic approaches. We explore various approximation algorithms, including local search heuristics, to design a solution that produces near-optimal tours. Our implementation is tested on multiple instances of the problem and evaluated based on the quality of the solutions obtained.

**1. Introduction:**
The TSP is a classic NP-hard problem with numerous real-world applications. The Half TSP extends the problem by introducing the constraint of visiting only half of the cities. Solving the Half TSP poses significant computational challenges due to its inherent complexity. Therefore, this project focuses on developing heuristic algorithms to find suboptimal solutions efficiently.

**2. Literature Review:**
In this section, we review existing literature on the TSP and Half TSP, exploring different approximation algorithms and heuristic techniques proposed by previous researchers. Some notable approaches include the nearest neighbor algorithm, genetic algorithms, ant colony optimization, and simulated annealing. These techniques serve as the foundation for our algorithm design.

**3. Algorithm Design:**
We propose a heuristic approach based on a modified version of the nearest neighbor algorithm. The algorithm starts with an initial city and iteratively selects the nearest unvisited city until half of the cities are visited. The tour is completed by returning to the starting city. We incorporate local search heuristics to improve the quality of the solution obtained by the nearest neighbor algorithm.

**4. Implementation Details:**
Our algorithm is implemented in Python, utilizing its rich libraries for data manipulation and mathematical computations. We read the input cities from a text file and write the output tour to another text file. The code is modularized into functions, allowing for easy extension and modification.

```python
# find distance between cities
def euclidean_distance(city1, city2):
    x1, y1 = city1[1:]
    x2, y2 = city2[1:]
    return round(math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2))
```

This function returns distance between two cities, so to get distance we are used that function.

```python
# find nearest city to current city
def find_nearest_city(current_city, unvisited_cities):
    min_distance = float('inf')
    nearest_city = None
    for city in unvisited_cities:
        distance = euclidean_distance(current_city, city)
        if distance < min_distance:
            min_distance = distance
            nearest_city = city
    return nearest_city
```

This function takes current_city and unvisited_cities list as parameter. To find the nearest city to the current city, it searches the closest city to the current_city in the list of unvisited_cities. It returns this city after finding it.

```python
# main function that runs the program
def half_tsp(cities, solutionfile):
    # set the variables
    start_city = cities[0]
    unvisited_cities = cities[1:]
    current_city = start_city
    visited_cities = [current_city]

    # in while loop finds all nearest cities
    while len(visited_cities) < math.ceil(len(cities) / 2):
        nearest_city = find_nearest_city(current_city, unvisited_cities)
        visited_cities.append(nearest_city)
        unvisited_cities.remove(nearest_city)
        current_city = nearest_city

    # calculate total distance
    total_distance = sum(euclidean_distance(
        visited_cities[i], visited_cities[i+1]) for i in range(len(visited_cities)-1))
    total_distance += euclidean_distance(visited_cities[-1], start_city)

    # Write output to a text file
    with open(solutionfile, 'w') as file:
        file.write(str(total_distance) + '\n')
        for city in visited_cities:
            file.write(str(city[0]) + '\n')
```

This function takes cities list and output file name as parameter. Here it creates unvisited_cities and visited_cities lists. Firstly it visits the first city in the cities list and add all other cities to unvisited-cities. After that, it calls find_nearest_city() until (n+1)/2 cities are visited. In each loop, adds the city that returns from the function to the list of visited_cities and removes it from the list of unvisited_cities. After the loop ends, it calculates the distance between visited cities and generates the output file.

```python
def main(instancefile, solutionfile):
    start_time = time.time()
    # Read input from a text file
    cities = []
    with open(instancefile, 'r') as file:
        for line in file:
            city_id, x, y = map(int, line.strip().split())
            cities.append((city_id, x, y))

    half_tsp(cities, solutionfile)
    end_time = time.time()
    print('Execution time:', end_time-start_time, 'seconds')


main(sys.argv[1], sys.argv[2])
```

Here it reads the input file and creates cities list. Then it calls the half_tsp function.

**5. Experimental Results:**

To evaluate the performance of our algorithm, we conduct experiments on multiple instances of the Half TSP problem. The instances vary in the number of cities, ranging from small to large-scale problems. We compare the quality of the obtained tours with the optimal solutions, whenever available, to assess the effectiveness of our heuristic approach.

```
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp.py .\test-input-1.txt test-output-1.txt
Execution time: 0.0219419002532959 seconds
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp.py .\test-input-2.txt test-output-2.txt
Execution time: 0.2902989387512207 seconds
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp.py .\test-input-3.txt test-output-3.txt
Execution time: 347.8767740726471 seconds
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp.py .\test-input-4.txt test-output-4.txt
Execution time: 2.274736166000366 seconds
```

This is the execution time of the 4 test input files.

```
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp_verifier.py .\test-input-1.txt .\test-output-1.txt
Your solution is VERIFIED.
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp_verifier.py .\test-input-2.txt .\test-output-2.txt
Your solution is VERIFIED.
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp_verifier.py .\test-input-3.txt .\test-output-3.txt
Your solution is VERIFIED.
PS C:\Users\srknk\OneDrive\Belgeler\Visual Studio Code\Python\AlgorithmHW2> python .\half_tsp_verifier.py .\test-input-4.txt .\test-output-4.txt
Your solution is VERIFIED.
```

Here, we verified the output files.