T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE2246 – Analysis of Algorithms

Homework-2 Report

<u>Group Members</u>

150119639 – Erdem PEHLİVANLAR

150120528 – Samet CAN

150120063 – Hamza Ali İSLAMOĞLU

150120031 – Şükrü Can MAYDA

Half Traveling Salesman Problem (Half TSP)

Two algorithms have been developed for the **Half Traveller Salesman Problem** requested from us. One of them is the **Clustering Algorithm** and the other is the **Nearest Neighbor Algorithm**. By combining these two algorithms, an attempt was made to approach the optimal result. During our studies, we first developed and tested the nearest neighbor algorithm, then we developed the clustering algorithm and added it to our code. Our code first enters the clustering algorithm and selects the densest regions to provide at least half the number of cities. Then it enters the Traveler Salesman class, where the distance is calculated by applying our other algorithm and the cities are listed. Our code receives a filename from the user and a percentage for the clustering algorithm. Our code outputs an output file for the file that is input to the user. We ran the 3 examples given during the homework and the 4 test files given later in our code and got the verified result. We also designed and tried different development algorithms for our code, but either we failed in the implementation phase or the result did not provide the improvement we expected. Our code is really perfect in terms of running time, it managed to produce verified results for the input we created and contained 50K cities in just 10-15 seconds. On the other hand, we are not that ambitious in terms of space, especially the matrix contained in our code includes a lot of space. You can look for running code details to Readme file

- ### <u>Clustering Algorithm:</u>

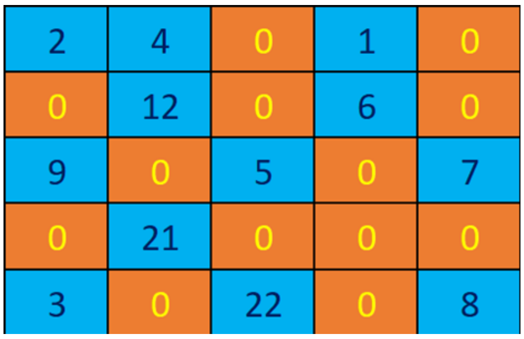
Definition of a **clustering algorithm** heaps up cities to a specific region. First of all, the algorithm determines the upper x coordinate and y coordinate, then determines the lower x coordinate and y coordinate. After determining these positions, the algorithm defines a **virtual rectangle** and it creates regions like a **square matrix** according to the given percentage called (***rowColumnPercentageForRegion***) which is in **Management class**. This attribute provides an algorithm to a **dynamic structure**. For instance, when the percentage is 50, the algorithm creates 4 regions like a 2x2 matrix or when the percentage is 25, it creates 16 regions like a 4x4 matrix. **Formula of that is the (100/percentage) X (100/percentage) matrix**. Moreover, when the percentage is changed, the result also changes. After creating the regions in a virtual square, **suitable cities** are added to **suitable regions based on (x, y) coordinates**. an appropriate environment is created for the clustering **algorithm** up to here. After that, the algorithm determines the **number of cities for each region** and **it iterates each region from largest to smallest with respect to number of cities**. **<u>Purpose of iterating from largest to smallest with respect to the number of cities is that cities are near each other in regions which have a lot of cities</u>**. While iterating each region, **Nearest Neighbor Algorithm** is used and it visits all cities in that region. When the cities are finished, the algorithm passes the next region. As stated first, this is the **Half TSP** algorithm. Therefore, **the clustering algorithm determines the half of the city via the regions.**

<u>Sample Run of Clustering Algorithm:</u>

Definitions : Number of total cities: **100**, rowColumnPercentageForRegion: **20**,

Half of city: 100 / 2 = **50**

Because of that the percentage is 20, a 5x5 matrix **(100 / 20 = 5)** is created.



2	4	0	1	0
0	12	0	6	0
9	0	5	0	7
0	21	0	0	0
3	0	22	0	8

Orderer Tour For Regions = [22, 21, 12] , 3 regions are enough for half of the city. 22 + 21 = 43 cities that are visited up to the last region. It does not provide half of the city. Thus, one more region is added to the ordered tour according to the number of cities from largest to smallest.

Nearest Neighbor Algorithm iterates each city of these 3 regions. However, the algorithm only iterates necessary cities that are 7 **(halfOfCities – visitedCitiesUpToLastRegion)** to **complete in half of cities in the last region**.

As a result, the algorithm does not visit unnecessary regions.

Time complexity is:**Θ(n)= n.m** Space complexity is:**Θ(n)= n**

n is number of city

m is number of region

Our Percentages for best results are:

- **example-input-1: %19**
- **example-input-2: %8**
- **example-input-3: %5**
- **test-input-1: %8**
- **test-input-2: %20**
- **test-input-3: %27**
- **test-input-4: %19**

- **Nearest Neighbor Algorithm:**

The **nearest neighbor algorithm** executed in the function named **orderCities** takes List<City> cities as an argument. The function **generateMatrix** creates an n x n matrix where each element represents the distance between two cities. The rows and columns of the matrix are labeled from 0 to n-1, corresponding to the indices of the cities. The value at the intersection of row index i and column index j represents the distance between the city with ID i and the city with ID j.

We start our journey from the city with index 0 in the created matrix, considering it as the starting point. We should return to this city after traveling. We traverse the rows as long as the row and column indices are not the same, and find the minimum distance.

The column containing the minimum distance in the row will be the city we will visit next, and we will use the column index containing the minimum value for the next row.

The city with the ID corresponding to the row index we will be visiting will be the next city we will visit.

The main algorithm works as follows: Starting from 0, we set the entire 0th row to 0 because we have already traveled from the city with ID 0 and cannot travel there again, unless all the cities have been visited. Similarly, we set the column that contains the minimum value to 0 because we cannot access that city again. The crucial point at the beginning is to disable the access to the 0th column when starting from 0, as traversing back to the starting city would result in visiting a city multiple times, which is incorrect. For the subsequent steps, we periodically set the row of the city we depart from to 0 and the column of the city we arrive at to 0. The column that is set to 0 is used for the next row.

At the end of the algorithm, it attempts to create the shortest route starting from the first element of the given city list as an argument. It continuously updates the total cost of the journey in the **distance** variable. Finally, we can access the total distance we will travel from there.

Time complexity is:**$\Theta(n)=2n^2$** Space complexity is:**$\Theta(n)=2n^2$**

n is number of city

Division of Labor - Who did what?

Erdem Pehlivanlar: Clustering Algorithm Design and Implementation, Report, Merging Classes and testing

Samet Can: I/O operations, Creating Project Structure, Nearest Neighbor Algorithm Design, Report, Merging the classes and getting the code working, Menu, testing

Hamza Ali İslamoğlu: Nearest Neighbor Algorithm Design and Implementation, Report, Calculations, Merging Classes, testing

Şükrü Can Mayda: generateMatrix - calculateDistance - twoCityDistance methods, Menu design and implementation, Report, Working on algorithm improvement, testing