CSE246 – Analysis Of Algorithms - 2017/18 Spring - Homework 3

Ufuk Çetinkaya 150113824 Sezin Gümüş 150113841

1. Problem Description:

We want to find a solution for the Traveling Salesman Problem (TSP) as close as possible. The problem can be summarized like as follows:

Inputs: n cities with city id's $c_1, c_2, \dots, c_n$ and n coordinate sets for those cities (x_1, y_1) , (x_2, y_2) , ... , (x_n, y_n) , in a 2D-Grid.

Output: A tour, so that total covered distance is minimum. Output file will display total distance, followed by all covered city id's.

Aim: Find the shortest possible route that visit each city one time and returns to city of origin.

General formula to calculate distance between two cities:

$d(c1,c2) = round(\sqrt{(x_1-x_2)^2+(y_1-y_2)^2})$

Total Tour = Σ of all distances covered.

2. Research about TSP and Citations:

To gain initial ideas, we used Chapter 12 from the course book , " Heuristics for the Traveling Salesman Problem" [1] article . At the end we decided to use Nearest Neighbor Algorithm by creating adjacency Matrix, this method will give us a good closure to optimal solution.

Heuristic	% excess over the Held-Karp bound	Running time (seconds)
nearest neighbor	24.79	0.28
multifragment	16.42	0.20
Christofides	9.81	1.04
2-opt	4.70	1.41
3-opt	2.88	1.50
Lin-Kernighan	2.00	2.06

Table 1. Heuristics Table – Average Tour Quality and Running Times for a 10000-city Grid

3. Methodologies and Ideas:

We will use Nearest Neighbord Heuristic Algorithm. Our initial idea was to improve Lin-Kerninghan Problem . However , due to being difficult to implement and improve Lin-Kerninghan Algorithm and running time of this algorithm , we decided to use and

improve Nearest Neighbor algorithm. Firstly , we imported the informations that are x , y coordinates and ID of cities by using these coordinates , distances between cities were computed and these distances were put in an adjacency matrix .By using distance in this matrix , neighbors were bound to each city . Each of cities with their neighbors were stored in “list_of_cities” array . We started with first element of the array to discover shortest path.We selected a next city which has minimum distance to current city as a next point . ID of next city and distance to a previous city were added to “path” array for calculating total cost and to store the minimum path. After reaching to next city , previous city will be current city and current city will be next city , previous city will be deleted from neighbors of each city to prevent next city visit previous city . We can assign false to true for visited cities instead of deleting , however , we thought , since ,we store visited city in an array , keeping visited is unnecessary hence , we decided to delete visited cities to provide space efficiency .

4. Coding Part:

We have decided to implement the methodology we developed in Python. Some parts of code is as follows.

TSP.py

```
def nearestNeighbor (list_of_cities):
    path = []
    min_edge = 0
    total_cost = 0
    current_city = list_of_cities[0]
    path.append([current_city.ID,min_edge])
    next_city_id , min_edge = min(current_city.getNeighbors(),key=lambda x: x[1])
    next_city_index  = findCity(next_city_id,list_of_cities)
    next_city = list_of_cities[next_city_index]
    total_cost += min_edge
    print(current_city.ID,"-->",next_city.ID)
    prev_city = current_city
    removeNeighbor(prev_city.ID,list_of_cities)
    current_city = next_city

    while (current_city.getNeighbors()):

        path.append([current_city.ID,min_edge])
        next_city_id , min_edge = min(current_city.getNeighbors(),key=lambda x: x[1])
        next_city_index  = findCity(next_city_id,list_of_cities)
        next_city = list_of_cities[next_city_index]
        print(current_city.ID,"-->",next_city.ID)
        prev_city = current_city
        removeNeighbor(prev_city.ID,list_of_cities)
        total_cost += min_edge
        current_city = next_city

    path.append([next_city.ID,min_edge])
    index = findCity(next_city.ID,list_of_cities)
    last_city = list_of_cities[index]
    last_edge = calculateDistance(list_of_cities[0].x_coordinate,last_city.x_coordinate,list_of_cities[0].y_coordinate,last_city.y_coordinate
    total_cost += last_edge
    return total_cost , path
```

5. Output & Results

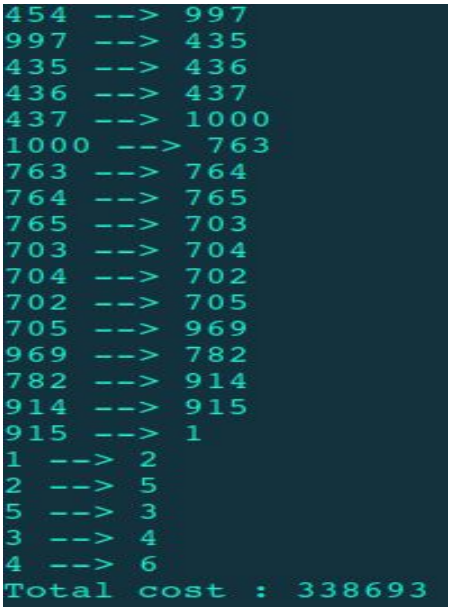

Figure 1. Results of “test-input-2.txt

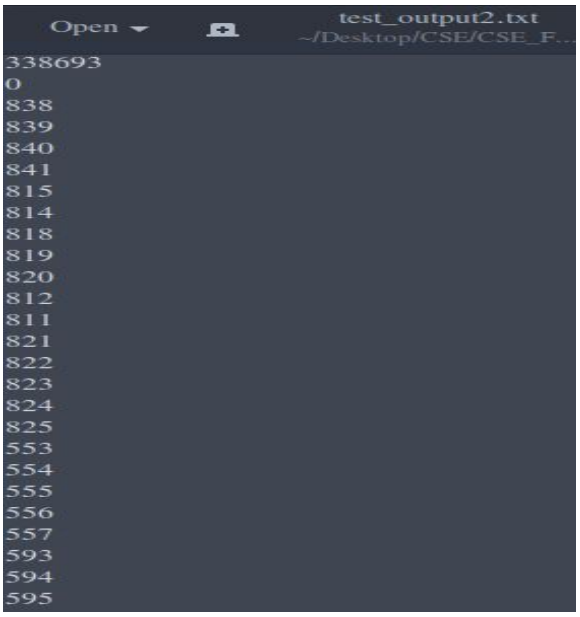

Figure 2. “test-output-2.txt” file that includes total cost and min path

6. Test and Evaluation of Results:

We used the tsp-verifier.py file supplied to test if our code gives us a correct total distance. If it does, we can then check how close our results are to the optimal solution.

As seen in the output screen, our code calculated the total path distance correctly and gave us a tour which is quite close to optimal solution.

When we run tsp-verifier.py with “test-input-2.txt” and “test-output-2.txt ” , the output is :

According to result of this output , the distance we computed is matched with result of tsp-verfier.py , we can conclude that Nearest Neighbor Algorithm we redesign gives correct solution .

6. References:

[1] Heuristics for the Traveling Salesman Problem Christian Nilsson January 2003 www.reseachgate.com