Name: Liu Rongxing
UUN:s1810054

Introduction to Algorithm and Data Structure

Coursework 3

1.Algorithm:

There is another heuristic method called "insertion heuristic" which runs in polynomial time and also gives a rather close approximation to the Travelling Salesman Problem. I did not come up with the algorithm on my own. Instead, I gained insights from **Professor Z.Max.Shen** (details can be found at https://aswani.ieor.berkeley.edu/teaching/FA13/151/lecture_notes/ieor151_lec17.pdf).

The basic idea of the algorithm is that we choose a random city to start with, and then try to insert other cities into the tour one by one, with the resulting total distance being minimal after each insertion. The algorithm can be executed in the following steps:

Step1: Initialize an empty set **T**, and choose any city **v** as the starting city (in my implementation of the algorithm, I just chose the city indexed "0"). Put **v** into the S so that **T** will contain a single element **v** now. Then remove **v** from the set of all cities **S**.

Step2: Consider all the remaining cities in **S**. Each of them can be inserted in different positions of **T** (i.e. if there are currently n cities inserted in **T**, then there will be n+1 possible positions that the new city can be inserted in). Of all possible cases, find the one such that the total distance obtained after the insertion is minimal. Record the city`s index and the position it should be inserted in.

Step3: Insert that city in that specific position.

Step4: Remove that city from **S**.

Step5: If S is empty (i.e. all cities have been inserted), stop the algorithm and output **T** as the answer. If not, return to Step 2 and continue the algorithm.

The pseudocode of the algorithm is given as follows:

Algorithm insertion-Heuristic (G):

Initialize **S** as the set containing the indices of all cities
Initialize **T** as the set containing the city indexed "0" and remove this city from **S**.
Initialize n as the total number of cities.

while (**S** is not empty) **do**
    $min\_dist = 999999$ ###min_dist is an arbitrarily large number
    $min\_pos = 0$ ###min_pos denotes the position of insertion
    $min\_idx = 0$ ###min_idx denotes the city inserted
    for i **from** 0 **to** n-1
        for j **from** 0 **to** length of **T**
            if (**T** does not contain i) **then**
                Insert i into j^{th} position of **T**
                value=total tour value of **T** #This tour value is different from that in Part A, since it only calculates the tour value for a shorter path
                if (value<min_dist) **then**
                    $min\_dist = value$
                    $min\_pos = j$
                    $min\_idx = i$
                Remove i from **T**
    Insert min_idx into the position min_pos
    Remove the city min_idx from **S**
Return **T** as the final answer

The time complexity of insertion_Heuristic is about $O(n^3)$, which is the same as the Greedy algorithm.

2.Experiments:

I wrote two functions that generate random graph both in the Euclid setting as well as in the metric setting.

The function **generate_euclid_graph(n,g)** takes in a parameter **n**, which denotes the number of cities to be generated, and a parameter g, which is an existing graph.
To run this function, I just run **g=graph.Graph(-1,"cities50")** first to initialize **g** as a graph. However, I am not interested in the data stored in the file "cities50". Instead, I am changing the parameters of **g** soon.
Firstly, I change **g.n** to the input parameter **n.** Then I create two arrays of length **n** with entries randomly generated between 1 and n*20 (with no repetition). One of the arrays provides the x-coordinates while the other provides the y-coordinates. After creating all the coordinates, I calculate the distances between each pair of cities and change the entries in **g.dist** accordingly.

Similarly, the **function generate_metric_graph(n,g)** generates a random metric graph, with weights of edges being positive integer.
Certainly there are limitations to my algorithm because theoretically, the output of **min_inter(a)** could be the empty set, which gives no possible value for the weight of an edge being taken.
However, after doing several testing, I found out that this algorithm can almost guarantee to output a metric graph with 40 cities. Moreover, as the entries of the initialized **g.dist** gets larger, the algorithm will 'guarantee' to output a metric graph with more cities. Nevertheless, the algorithm will he much slower in that case.

After being able to generate random graph for the two settings, it is time to check the accuracy of the approximation of the three algorithms, namely the combination of swap-Heuristic and TwoOpt-Heuristic, the Greedy approach and the insertion-Heuristic.

For a graph with **n** number of cities, there are **n!** number of possible permutations of the tour path. The time complexity of using the exhaustive method will be **O(n!)**, which is super slow as **n** grows larger. However, when **n** is kept below 8, **n!** will be around 40000, which can still be handled by computer. Hence, I wrote the exhaustive method **brute_force(g)** that can find the minimal solution with small **n**.

After that I wrote test functions for all three algorithms and two settings. Each function takes in a parameter **n**, which denotes the number of trial wish to be tested, a parameter **g**, which is the graph to be generated later, and a parameter **size**, which is the number of cities. For each function, I calculate the accuracy (which is given as the result obtained using the proposed algorithm divided by the result obtained by the exhaustive method) for each trial, and calculate the average accuracy for all the trials.

After testing for the three algorithms under both the Euclid setting and the metric setting, **I have reached the same conclusion that the approximation using the combination of swap and TwoOpt Heuristic is the best, followed by the insertion-Heuristic, while the greedy approach yields the worst result.**

Now for large input (that cannot be solved by the exhaustive method), I just considered the Euclid setting. In order to test the accuracy of the three algorithms, I generate graphs such that all cities are **aligned on four sides of a square**. I also set one city at each vertex. **Intuitively, the best solution in this scenario is to travel around the square and the tour value will be the perimeter of the square.**

Then I calculate the accuracy of the three algorithms and I have reached the following conclusion. **The combination of swap and TwoOpt Heuristic almost guarantees to get the best solution every time, while greedy approach and the insertion method do not. For the other two methods, neither of them is more efficient than the other. Nevertheless,**

al three algorithms have higher accuracy due to the special features of the graph.

Hence after doing the tests for both small and large inputs, I have reached the conclusion **that the combination of swap and TwoOpt Heurustic is the most efficient algorithm of all three. The insertion method is slightly better than the greedy approach.**