ARTIFICIAL INTELLIGENCE
ASSIGNMENT-3

Group-10
CS11B018
CS11B019

Q4. Allow users to choose the number of cities. Display cities on the monitor. Implement deterministic algorithms: Greedy nearest neighbour, Greedy Heuristic (shortest edges first), Savings Heuristic. Display the process of tour construction on the screen.

PROGRAMMING LANGUAGE:
Python

FILE STRUCTURE:
The folder consists of 6 source files:
-> main.py
The main file
-> buildgraph.py
This file declares the classes and functions for a city,edge and Graph respectively.
-> greedy.py
This file is for the implementation of greedy algorithm
-> input.py
This file is for building the distance matrix for the given number of cities
-> nearest.py
This file contains the implementation of Nearest Neighbour algorithm
-> savings.py
This file contains the implementation of Savings Heuristic
It consists of 2 input files euc_7 and euc_100. euc_7 has 7 cities and euc_100 has 100 cities

INPUT FORMAT:
Input should be from a file and the file should be in the following format
The first line will be the number of cities. The next lines should be the coordinates of the each city. The numbering of the cities start from 1 till the number of cities. This will be followed by the distance of each city to other cities in the respective order.

OUTPUT FORMAT:
For the Nearest Neighbour it prints only the order of the cities seperated by a space, for the greedy it prints the edge it is considering, and the partial tours it has constructed so far and then later it prints the path it found. For savings it prints the two cycles it is merging with and the result of the merging and then later it prints the path it found.

ALGORITHMS:
Nearest Neighbour:
We choose a start city and from the start city it chooses the nearest city which it hasnt already visited and tries to complete the tour.

Nearest Neighbour:
tour <- start city
city <- start city
while length of tour is not equal to total number of cities:
    neighbours <- neighbours of city sorted in increasing order of costs to visit from city to that node
    node <- first neighbour
    while node is not in tour

```
            node <- next city from neighbours
        city <- node
        tour <- append city
```

Greedy Algorithm:

We sort the edges in the order of their weights and then we keep choosing the edges such that they dont form a cycle or degree of each node is not more than 2. We basically here are avoiding to add those edges which will form a circle by checking if both the cities of the edge belong to same partial tour which we already have before. If they do then we avoid them else we update the set of partial tours such that the degree of each node will be less than or equal to 2 after updating. We are doing this by considering edges only which have one of the cities at the extremes of one of the partial tours constructed otherwise it will change the degree of that city to 3.

Greedy Algorithm:

```
edge_list <- edges sorted in increasing order of their magnitude
partial_tours <- []
while length of edge_list is not zero
        remove the shortest edge from edge_list
        update_partial_tours()
        if length of partial_tours is one and its element's length is total number of cities
                break

update_partial_tour()
        city1 <- one node of the edge
        city2 <- other node of the edge
        i <- index of the partial_tours in which city1 is present
        j <- index of the partial_tours in which city2 is present
        //Initially i and j are intialised to 0. So if one of them is 0 then it either is in the first element or not at
        //present in the tours
        if i and j are equal
                if both are not equal to 0
                        return
                else if both are equal to 0
                        if both cities are present in that element
                                return
                        if both cities are not present in that element
                                partial_tours <- append the edge
                                return
                        if one city is present in that element
                                if that city is at the extremes of that tour
                                        tour <- partial_tours[0]
                                        delete it from partial_tours
                                        tour <- append the other city to this city
                                        partial_tours <- append tour
                                        return
        else
                if city1 and city2 are present in the extremes of the partial tours
                        if i = 0
                                if city1 is present in the extreme of that element
                                        merge partial_tours[i] and partial_tours[j] and delete them from
                                        partial_tours and add new tour to partial_tours
                                        return
                                if city1 is not present in the element
                                        tour <- partial_tours[j]
                                        delete it from partial_tours
```

```
                    tour <- append the city1 to this city
                    partial_tours <- append tour
                    return
        else if j =0
            repeat the above process
        else
            merge partial_tours[i[ and partial_tours[j] and delete them from
            partial_tours and add new tour to partial_tours
            return
```

Savings Heuristic:

We first construct cycles from a base city to all the other cities. Then we merge cyles with edges which will give us the maximum savings. We keep on doing it till a complete cycle of all the cities is done. So first construct cyles from each city to the base city. Now calculate the savings matrix for each city with each and every other city which is not itself and neither of the cities is equal to the base city. So if k is base node and c(i,k) is the edge from city i to k then savings matrix entry can be calculated as follows

s(i,j) = c(i,k)+c(k,j)-c(i,j) if i is not equal to j and they are not equal to k
= sentinel, elsewise

Then choose the upper triangular matrix of the savings matrix and sort them in decreasing order. Now keep merging the cycles with the first element and then delete the first element until required cycle is formed.

Savings Heuristic:

```
savings_list <- edges sorted based on the savings they give in decreasing order
result <- []
k <- base city
construct cycles from base city to every other city
while length of result is not equal to number of cities +1
    city1 = savings_list[0][0]
    city2 = savings_list[0][1]
    if length of result is 0
        result <- merge(cycle(k,city1),cycle(k,city2))
    else
        if city1 in result and city2 not in result
            result <- merge(result,cycle(k,city2))
        else if city2 in result and city1 not in result
            result <- merge(result,cycle(k,city1))
    delete savings_list[0] from savings_list

cycle(city_1,city_2)
    cycle = [city_1,city_2,city_1]
    return cycle

merge(path1,path2)
    new_path = []
    i = 0
    while i < length(path1)-1
        new_path <- append(path1[i])
    i = 1
    while i < length(path2)
        new_path <- append(path2[i])
    return new_path
```