Name: Majdi Alali

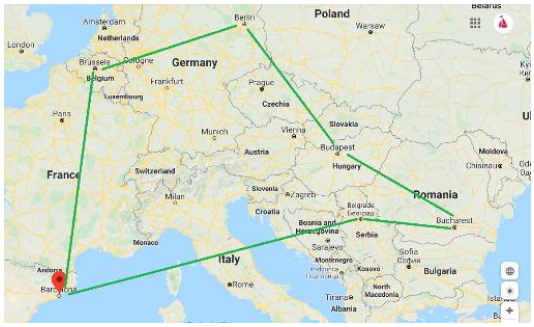
username: majdioa

Assignment1, IN3050

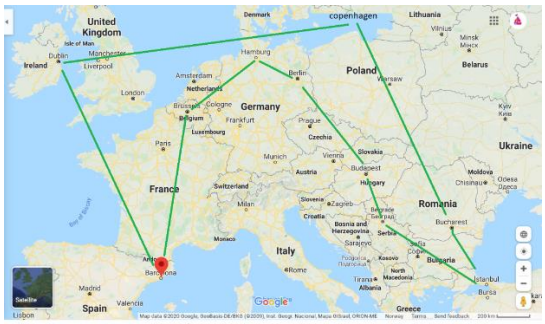
Pre-processing:

First, I will talk about the csv file. I have read it, divided it via delimiter ';' and save all the data in a list called *Mainlist*. Then I made a String list, called *cities,* which represents the names of the available cities in our file. While the float list *distances* takes care of the distances between the cities.

A graph which show the shortest path between the 6 cities:



An another graph shows the shortest possible path among 10 cities.



Excusive search:
-Shortest path via salesman_exhaustive_search for 6 cities is :', ['Barcelona', 'Belgrade', 'Bucharest', 'Budapest', 'Berlin', 'Brussels', 'Barcelona'], 'total distance among these cities is: ', 5018.8099999999995, 'km'
-the estimated time for 6 cities: 0:00:00.002952

-Shortest path via salesman exhaustive search for 10 cities is :', ['Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'Berlin', 'Copenhagen'], 'total distance among these cities is: ', 7486.309999999999, 'km')
-the estimated time for 10 cities: 0:00:21.446047

-I have made a method called estimate_**En_per_for_24cities()** which calculates one permutation/order of 24 cities. It gives 0,00063 sec.

One approach To estimate the time for 24 cities with 24 :
0,00063 * 24! = 3,90882493091940E20
Then we divided the above on 3.17098e-11 to get the number years it will take to execute this on my computer. It seems it takes million if not billions years!
Estimated number for on permutation %1000 (msec.) %60(sec.) %60(min.)%24(hour)%4(week)%52(month)%12(year)
Another approach is the following form:
time(n_cities) * (24!/n_cities!)

Obs: I have used normal lists/arrays, not numpy's. And you konw the nump's array is much faster than the normal one. That can help to process the code faster. However, I have a modern pc which helps the processing goes somehow fast.

.....

Task 2: hill climbing

Functionality of my method hill_climber()**:**
This method returns a list. This list consists of two elements. The first element represents the total distance among the shortest possible path. The second element is a list of indexes of that path, in other words, the indexes of the cities for the shortest path. First, I generated a random permutation of n cities and calculated the distance of this permutation. Then I made a copy of this permutation an put in a variable called term_perm. This copy/variable will help me to compare between improved version of the permutation to get the best case (i.e. best improved version of a permutation. Afterwards, I made a loop which tries to improve the current version of the perm n times by swapping between to cities randomly.
Helping methods:
-I made a method called *calculate_distanse().* This takes one permutation and return an integer which represents permutation's total distance. Observe that I assume that we deal with 24 cities.
-I made another helpful method called *swapToCities()* which takes to random integers(=indexes of to cities) and a permutation as arguments. Then we manipulate this permutation via swapping these to random integers/cities.

Answer on the questions:

- **How well does the hill climber perform, compared to the result from the exhaustive search for the first 10 cities?**
  It's not as precise as exhaustive search. But for 10 cities, it can give the best result like exhaustive search if we pay attention to the number iterations we run.

Obs. Pay attention that my method returns a list which has the total distance to the shortest possible path and another list which gives the indexes of the cities.

- **Report the length of the tour of the best and worst for 10 and 24 cities after running 20 times.**
  -The best result for 10 cities is 7486.3099999999995 km [2, 6, 8, 3, 7, 0, 1, 9, 4, 5]
  -The worst result for 10 cities is: 16871.170000000002 km [9, 6, 8, 2, 3, 0, 4, 5, 1, 7])
  -After 20 iterations and every iteration goes in a loop 3000000 times, here is the best result I have gotten for the total distance of 24 cities is 12384.22km.
  [11, 7, 16, 12, 0, 18, 13, 15, 17, 2, 23, 22, 5, 1, 20, 9, 4, 10, 14, 19, 21, 6, 8, 3]
  - while the worst result is:
  40925.73000000001 km, [3, 17, 6, 22, 15, 10, 0, 8, 14, 20, 19, 5, 13, 16, 12, 7, 2, 18, 11, 21, 9, 23, 4, 1])
  Obs: To get the worst solution I reversed the operator < in if-check when I compared between the distance before and after swapping to tell the program that I want that solution to be as worse as possible.
  -the standard deviation is for 10 cities is 373.8397929239489
  -the standard deviation is for 24 cities : 873.2307058110791

- It is very clear that exhaustive method gives the best result (the shortest path). But hill climbing algorithm is much faster!

……..

Task3: Genetic algorithm:

-I made also a method called *crossover().* It takes to parents and make to new children which the method returns. It's good to mention that this mixing between the features of the given parents is very random. That will give high variations in the new generations. Consequently, we will have high exploration.

- I have made another important method also. The method is called *population_and_totalDistanses().* This method generates a population of size 100 and calculate the total distance every permutation in this population. And then it sorts the population and keep the best 50 solutions/permutations. In other words, this method does parents selection. And return a list of 50 elements (i.e. survivals). Every element in this list has to elements, the total distance for a permutation and its related permutation.
-The third important method is *offspring_mutation_newGen()*. It takes the previous method as an argument. I will remind you that the prev. method returns a list of 50 survivals. This method makes offspring or children, improve them and then extend the list of their parents/ the survivals. Observe that the new big list has 100 elements which form so-called new generation. And at the end we sort the list before to return it.

-*Generate_generations()*, The fourth method, completes what we have done in the last three methods. It generates 300 generations out of the generation nr. 0. I made this method like nodes. Every node is related to the previous one. And then pick up the best solution (shortest path) among all generations.

Answers on the questions:

-choose different values for the population size

1) For 24 cities After running 20 iterations on 300 generations, the best result is

the best result is: 12561.220000000001km for the following permutation: [9, 3, 18, 13, 0, 12, 7, 11, 16, 20, 8, 6, 21, 19, 14, 10, 23, 2, 17, 15, 22, 5, 1, 4]]

2) the best result for 6 cities is: [5018.8099999999995, [0, 1, 4, 5, 2, 3]]

3) the best result for 10 cities is: [7486.3099999999995, [0, 1, 2, 4, 5, 9, 6, 8, 3, 7]]

-the standard deviation for 24 is: 1841.7613078081165

- Among the first 10 cities, did your GA find the shortest tour (as found by the exhaustive search)? Did it come close?
Yes, it did but I intended to run the code many times in a way I can get the best possible result.

-How did the running time of your GA compare to that of the exhaustive search?
For 10 cities the running time was relatively close between GA and the exhaustive search. But for 24 cities the difference is huge where GA is much faster because the exhaustive search tries alle possible solutions/permutations (24!). Consequently, the code will never terminate!

-How many tours were inspected by your GA as compared to by the exhaustive search?
They are very few in comparison with the exhaustive search.
I assume the following equation:
Size of population+(children+constant*n_generations)
Constant is a number permutations which are generated by swapping in mutation stage.
In my case:

100 +