Assignment 6: Simulated Annealing

Simulated Annealing

- The code uses Simulated Annealing to find a close-to-optimum solution to the Travelling Salesman Problem.
- The method introduces the concept of a *Move*. This is a change from the present order to some other order, where we hope the solution is better than it is here. The move performed in this solution is swapping two random points in the order of cities to be traversed by the travelling salesman.
- We do this by introducing a notion of *Temperature*. We accept a path with a greater distance with the probability given below as:

$$P(\Delta d) = e^{-\frac{\Delta d}{T}}$$

- Here Δd is the distance increase as a result of swapping two points, and T is the present temperature. Clearly, as T decreases, the probability calculated above will tend towards $e^{-\infty} = 0$. We decrease T by a factor $decayrate = 0.99$, in this case
- What this means is, after a number of iterations, the probability of accepting worse values become very small, so that only swaps that result in smaller distances are chosen.
- The number of such iterations done in this solution is 50000, and the initial Temperature and Decay Rate are 1000 and 0.99. This code will also work for much larger number of cities
- The reason that all three are relatively large numbers is to "walk around" the solution space even more and find a lot of potential, but not-so-optimum paths in the beginning and explore around so that more paths are discovered. This is done by increasing Decay Rate and Number of Iterations. This would force an increase in decayrate so that you keep accepting such values for a reasonably high number of iterations.
- Simulated annealing reduces the complexicity of the solution greatly. This is because the number of solutions to the problem is

$$\frac{(n-1)!}{2}$$

where n is the number of cities

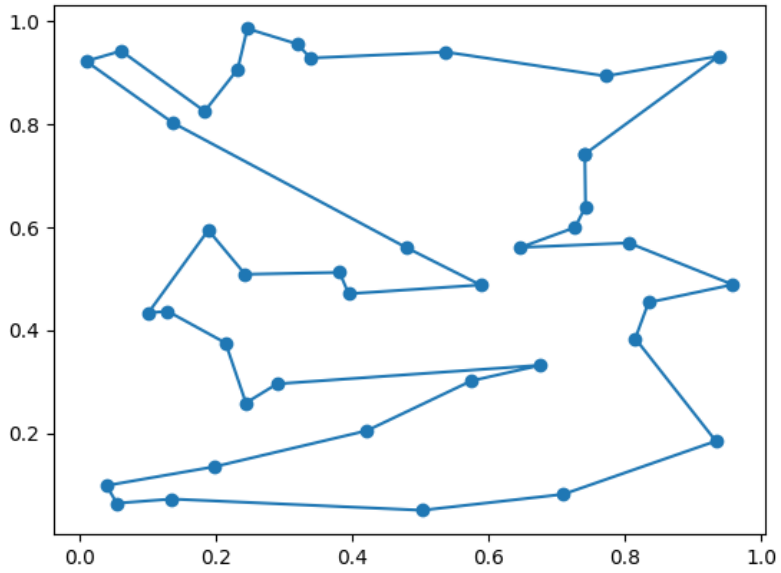
- This means that the number of potential paths goes to a very large values as value of n increases. Simulated annealing aims to reduce the computations required for the same.

Approach

- The code uses Simulated Annealing as mentioned above.
- Initially, a random path is chosen and the distance is stored in a variable
- Then the initial values of Temperature, Decay Rate and Number of iterations are set.
- Then simulated annealing is performed and the final value after the said number of iterations is printed.
- Also, the plot is made.
- I did not animate for each iteration and it increases the execution time substantially, which sort of contradicts the whole idea of simulated annealing over bruteforcing through all the potential paths to reduce time of execution.

Results

The final plot for one of the best simulations of 40 cities is given below:



The stats for this run:

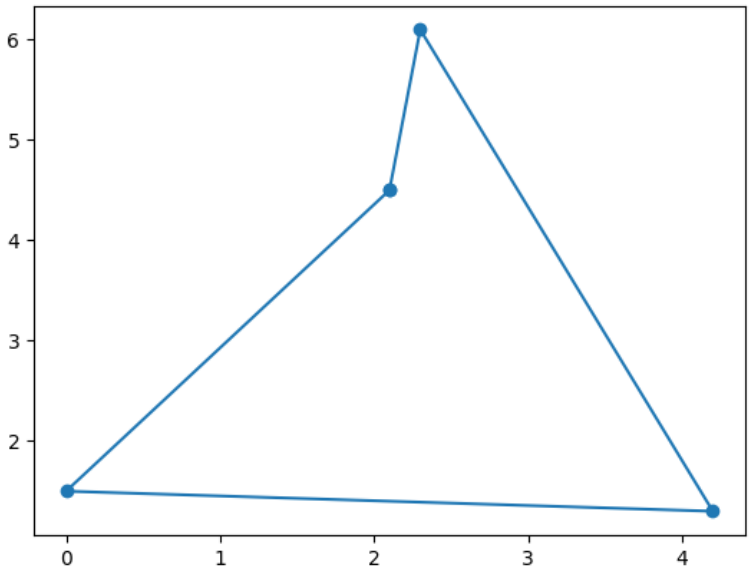
- Initial Guess: 21.845852437776433
- Best Distance: 6.10745869449431
- Improvement= 72.04293715756712
- Order: [35 2 7 32 4 12 5 29 37 0 9 28 36 6 1 16 25 39 23 15 14 3 11 26 21 17 34 22 10 38 19 20 30 18 24 33 13 31 27 8]

I ran the code 1000 times and these were the values obatined:

- The Best Distance was found to be 5.763177841548179.
- The Worst Distance was found to be 9.391555162568514
- The Average was found to be 7.4381060504492975

The animation is saved in the same folder as the code

The final plot for the set of 4 points given:



- Initial Guess: 14.787698948035882
- Best Distance: 14.64154124236167
- Improvement= 0.9883735541804763
- Order: [3 1 2 0]

I did not do multiple runs as in the above problems since the number of iterations was large enough to reach the best solution in all the runs I did.

How to Run

- The code takes the txt file as a command line argument. You can run the code with the following command:

`python3 EE22B086_Assignment6.py <data_file_name>`

- `<data_file_name>`: The file in which the data as given in the problem statement is stored.

Functions in Code

1. `tsp(cities)`

- Arguments:
  - cities: A list (or tuple, or other iterable datatypes) of 2- element arrays as input (the arrays have to be numeric).
- Return Values:
  - cityorder: List of the best order obtained for the TSP after Simulated Annealing.

2. `distance(cities, cityorder)`

- Arguments:
  - `cities`: A list (or tuple, or other iterable datatypes) of 2- element arrays as input (the arrays have to be numeric).
  - `cityorder`: A list containing the order in which the cities are to be traversed
- Return Values
  - float totaldistance: The distance in the TSP, calculated as given in `cityorder`

The code also saves the best path obtained in the end to the file named "Sim_Annealing.jpg"