Assignment 6

Aim: Optimization of Travelling Salesman Problem using Simulated Annealing

Approach

- For the given problem I have provided a single .py file which performs the `Simulated Annealing` optimization.
- The function first takes a text file in the format provide by sir.
- Then I set the required essentials for the Annealing proocess

```
- Temperature = 10000 which can changed. High because more number of iteration
- Decay Rate = .995
- Iterations = 30000 if this is changed then change temp as well
```

Note

- For high number of cities the optimum distance is variable everytime you run the programme as it highly depends on the initial guess and the no. of iterations.
- The programme will output the optimized order, and it will also save the Animation with the filename Animation.gif.
- The TA's are just required to update the path in the start oof the .py file.
- The Animation will take like 15 to 20 seconds to get saved.
- The programme will output the `best_order` as well as the `Percaentage Improvement`.

Simulated Annealing

- The programme starts with a `random initial order` called as the `best_order`.
- It first creates a variable named `current_order` and `near_order.`
- It then randomly swaps order of any two of the cities, using the `random.sample()` function from the random module into the `near_order`
- `Near_order` contains the swapped order.
- Now if the distance from the `near_order` is less than the `current_order` then the update the `current_order`.
- `current_order` also updated if the value generated from the function `np.random.random_sample()` less than that of `np.exp(-delta_distance / temp)` where `delta_distance` is the difference in the distance.
- This essential signifies that a greater distance is taken with some probability to explore more paths unlike gradient descent.
- The `best_order` is updated if and only if the `current_distance` less than the `best_distance`.
- At the end Temperature is updated via `Temperature = Temperature * decayrate`

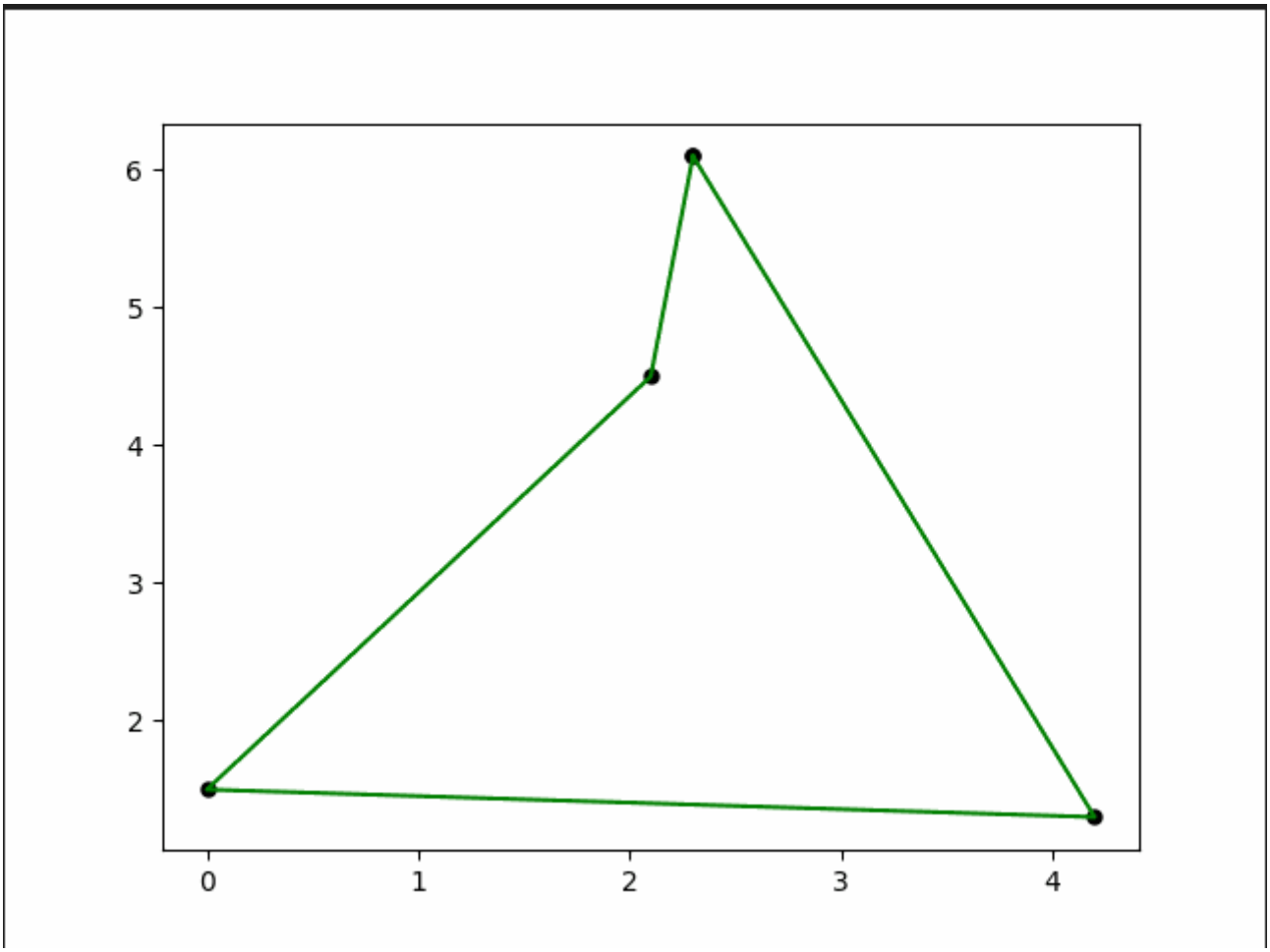
- For the animation theres another list called as the `order_list` which stores the orders from all iterations.
- At the end of all the iterations we get the `best_distance`.

Results

- For the 4 cities problem -

```
4
0.0 1.5
2.3 6.1
4.2 1.3
2.1 4.5
```

- The Best order is `[1 2 0 3]` with cyclic rotations.
- `Minimum Value = 14.64154124236167`
- The path is -

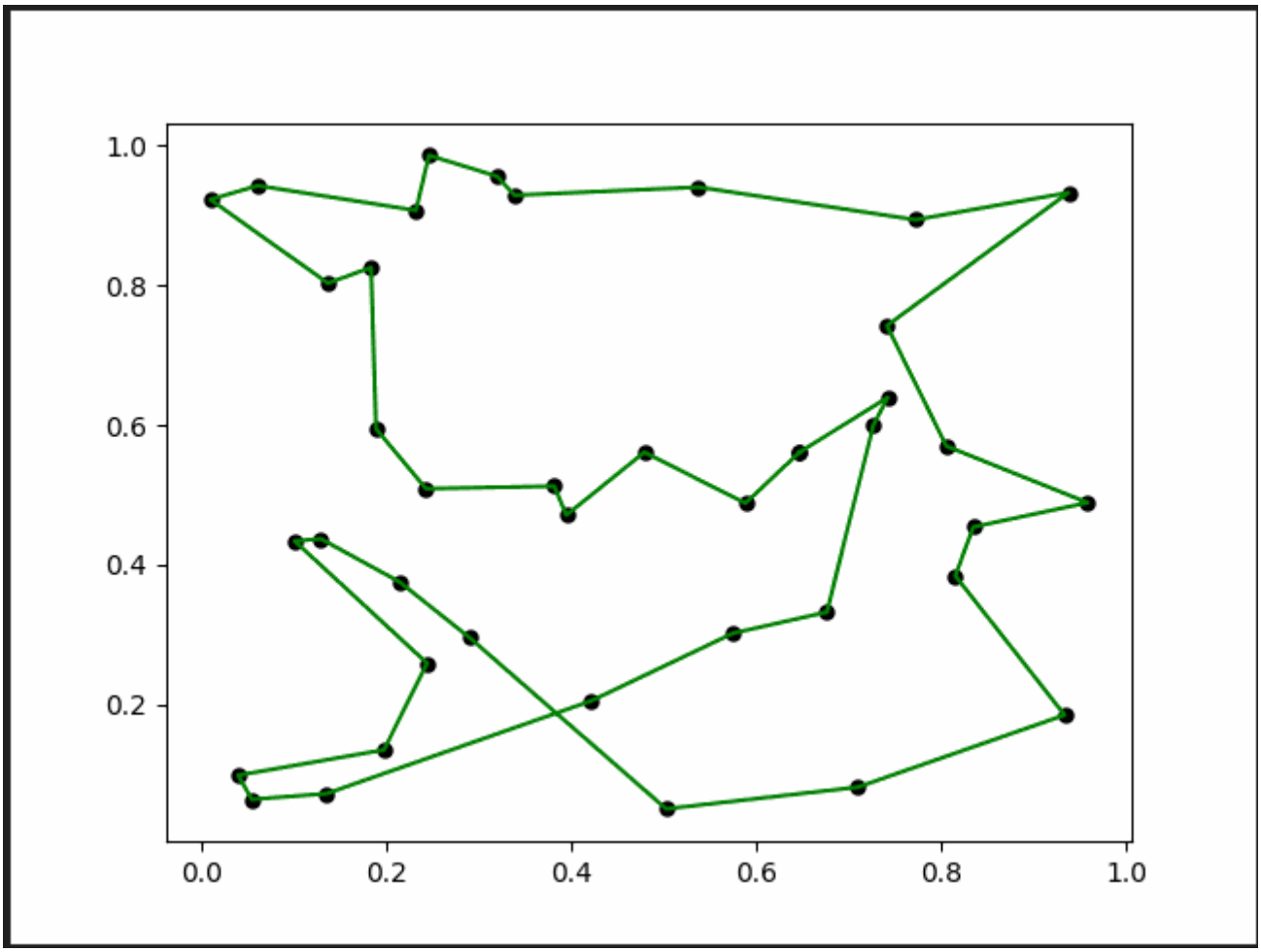


- For the 40 cities problem -

```
40
0.060562 0.942934
0.394229 0.471144
```

```
0.937219 0.932889
0.243591 0.259056
0.338060 0.929149
0.245617 0.986246
0.587850 0.488260
0.771818 0.894098
0.741888 0.639771
0.009128 0.923038
0.054042 0.064156
0.289258 0.296086
0.319255 0.956349
0.805218 0.569889
0.213761 0.375533
0.128785 0.437189
0.381072 0.512592
0.420950 0.205079
0.814813 0.384201
0.502155 0.050541
0.709357 0.081568
0.574503 0.302022
0.039253 0.098582
0.100408 0.434016
0.834228 0.454153
0.242461 0.508993
0.675943 0.332270
0.726183 0.599843
0.136350 0.803325
0.231589 0.907479
0.933070 0.184902
0.646404 0.561210
0.537232 0.940763
0.958013 0.488955
0.196986 0.135021
0.740817 0.742469
0.478621 0.561161
0.183073 0.825718
0.134909 0.072343
0.188435 0.594701
```

- There are different values for the number of iterations, temperature and the number of times the programme is run.
- The least value I once got was `Minimum Distance = 5.888701771803248`
- Best path - `[31 6 36 1 16 25 39 37 28 9 0 29 5 12 4 32 7 2 35 13 33 24 18 30 20 19 11 14 15 23 3 34 22 10 38 17 21 26 27 8]`
- The path is -


1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0