Documentation

October 26, 2023

1 ASSIGNMENT - 6

1.1 Traveling Salesman Problem Solver

Mithesh M | EE22B060

This code solves the Traveling Salesman Problem (TSP) using the Simulated Annealing algorithm. It reads a list of city coordinates from a file, finds an optimized order to visit all cities that minimizes total travel distance, calculates how much this optimized order improves upon a random order, and visualizes the cities and travel path using matplotlib.

1.2 Dependencies

This code requires the following Python libraries:

- numpy
- matplotlib

1.3 How to run

To run this code, replace `"cities.txt"` with the input file in the following line:

```
with open("cities.txt", "r") as file:
```

Then, simply run the Python script.

1.4 Code Explanation

The code consists of three main functions:

- `distance(cities, cityorder)`: This function calculates the total distance of a given order of cities.
- `tsp(cities)`: This function solves the TSP using Simulated Annealing. It generates a random initial order of cities, then iteratively improves upon it.
- `get_neighbor(current_order)`: This function is defined inside the `tsp` function. It generates a neighbor order by swapping two cities in the current order.

The main loop of the `tsp` function generates a neighbor order, calculates its cost (total distance), and decides whether to accept it based on the acceptance probability. If the neighbor order is accepted, it becomes the current order. If it's better than the best order found so far, it also becomes the new best order.

1.5 Assumption

I have assumed the followings things in my code

- The input file is named as `cities.txt`
- initial_temperature = 1000
- cooling_rate = 0.995
- Number of iterations = 100000

1.6 Algorithm

- Initialization: The code starts by randomly generating an initial order of cities.
- Neighbor Generation: In each iteration, it generates a 'neighbor' solution by swapping two cities in the current order. This is done in the `get_neighbor` function.
- Cost Calculation: The 'cost' of an order is the total distance traveled for that order of cities, calculated in the distance function.
- Acceptance Probability: The `acceptance_probability` function determines whether to accept a neighbor solution. If the neighbor's cost is less than the current cost, it is always accepted. If not, it may still be accepted with a certain probability, depending on how much worse the neighbor is and how high the 'temperature' is.
- Annealing Schedule: The temperature starts at a high value and gradually decreases over time, according to a cooling schedule. This allows the algorithm to accept worse solutions at first but become more selective as time goes on.
- Iteration: The main loop of the algorithm involves generating a neighbor, deciding whether to accept it, and updating the current and best solutions found so far.
- Result: After a set number of iterations, the code returns the best order of cities it has found.
- Visualization: Finally, the code plots the cities and the best path found, and saves this plot as an image.

1.7 Results

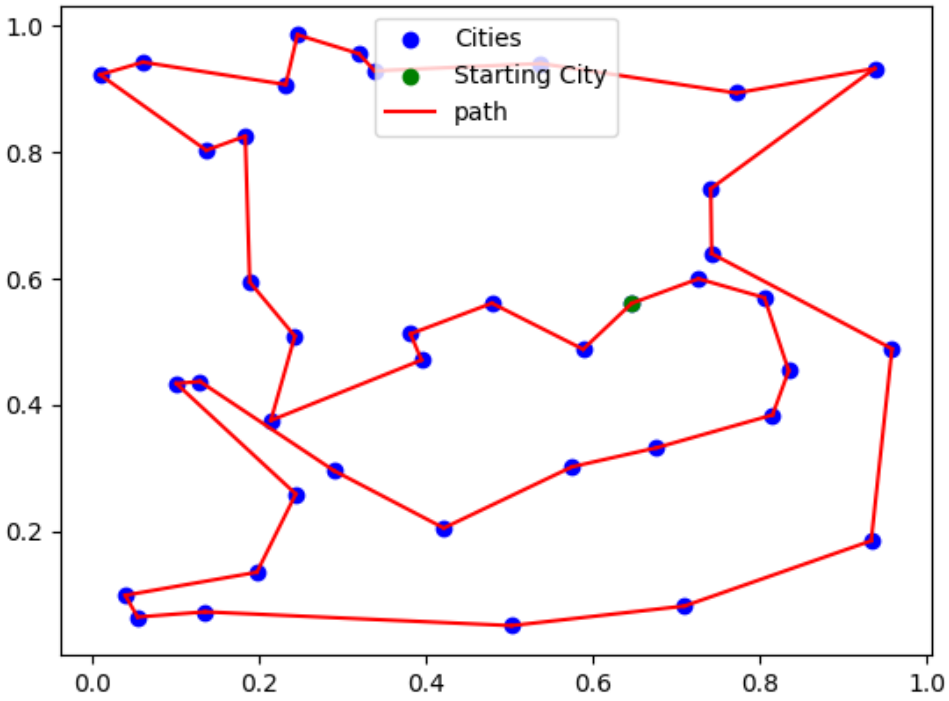
After running the file `tsp40.txt` given, we will see an output like this:

```
The Optimized Order is [31, 27, 13, 24, 18, 26, 21, 17, 11, 15, 23, 3, 34, 22, 10, 38, 19,
20, 30, 33, 8, 35, 2, 7, 32, 4, 12, 5, 29, 0, 9, 28, 37, 39, 25, 14, 1, 16, 36, 6] with
distance = 5.92. It has 71.16% improvement from the starting order [0, 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39].
```

This tells the optimized order of cities, its total distance, and how much it improves upon a random order.

In addition, a plot will be generated showing all cities as blue dots, the starting city as a green dot, and the travel path as red lines. The plot will be saved as "assg6.png".

The plot for the given input `tsp40.txt` file for the optimised order mention above is:


Cities
Starting City
path
1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.4
0.6
0.8
1.0