Name: Hany Hamed,
Group Number: BS18-Robotics
Course: Statistical Techniques for Data Science and Robotics (STDSR) - Spring 2022,
Assignment 3 Code

Contents

1 Structure of the code

There are two main folders:

- src: it has all the source codes as following:
  - utils.py: this includes all the useful functions for importing the csv data with transformation from geolocation to Cartesian coordinates and many other functions for plotting.
  - main.py: it includes the main components of the code to parse the arguments, and parse the csv file filled with the data. Moreover, it runs the main algorithm on the data with plotting.
  - SA.py: it includes the main implementation for SA function that has the Simulation Annealing algorithm implementation. Furthermore, it contains other useful functions to propose the new path and to compute the cost of this path.
- $\text{tmp}_plot.py : itwasusedtocollectthedatafromdifferentexperimentsandplotthem.$
- city: it is a sub-module from the original repository for the cities data
- results: it has all the figures from the results

2 Travelling Salesman Problem

In this problem, we intend to find the optimal path with the lowest cost for the path, which is the cumulative summation for the distances between the cities in the path. The path is starting from a city and ends by the same city.

The path contains order of cities with their geolocation (longitude and latitude) and the transformation to their x and y Cartesian Coordinates.

We get the data of the Russian cities from this repository and parse the csv and extract the most 30 populated of this list.

Then, we compute the distances between each city and the other cities. We use the function from geopy.distance to compute the distance based on the geolocation and use the distance in Kilometer (KM)

3 Simulated Annealing

This algorithm is one of the simple algorithms that optimizes the functions without any required computation for the gradients of the function. Therefore, it is considered as a gradient-free algorithm. The algorithm is based on initialize a solution, then proposing a new solution and comparing the proposed solution based on a specific cost and then comparing the cost and act greedy based on the cost.

Here, we define the cost as the cumulitive summation of the distances through the path from the starting city till the end of the path that returns to the same city back.

The algorithm is as following:

```
Initialize the temperature
Initialize the cooling rate
Initialize the initial solution
Compute the cost of the initial solution, mark it as the current cost
while temp > 0.1:
    Generate new solution based on swapping two random cities
    Compute the cost of the new solution, mark it as the new cost
    Compute \textbf{p} = e^(current cost−new cost)/temp
    if(new cost < current cost or random.uniform(0,1) < p):
        current solution = new solution
        current cost = new cost
    temp *= cooling rate
```

The following is the implementation in python (it can be found in SA.py)

```
def SA( cities, initial_temp=10000, cooling_rate=0.95,
        visualize=False, visualization_rate=0.01, fig=None, ax=None):
    cities_dict = cities
    cities = to_list(cities)
    time = 0
    temp = initial_temp
    costs = []
    temps = []
    new_solution_cost = 0
    current_solution = cities
    current_solution_cost = compute_cost(cities)
    new_solution = None
    while temp > 0.1:
        # Get new solution
        new_solution = generate_solution(current_solution)
        # Calculate the cost for the new solution
```

```
        new_solution_cost = compute_cost(new_solution)
        # Calculate p
        p = safe_exp((current_solution_cost - new_solution_cost)/temp)#(math.exp()
        # print(p)
        # if new solution is better or random less than p
        if(new_solution_cost < current_solution_cost or random.uniform(0,1) < p):
            current_solution = new_solution
            current_solution_cost = new_solution_cost
            costs.append(new_solution_cost)
            temps.append(temp)
        if(visualize):
            path = generate_path(current_solution)
            title = f"Temp={temp:.3f}, Cost={current_solution_cost:.3f}\nInitial_temp={initial_
            plot_animation(fig, ax, cities_dict, path, pause=visualization_rate, title=title)

        temp *= cooling_rate
        time += 1
    return costs, temps, new_solution, new_solution_cost
```

For choosing the new solution, we use the following method of swapping two different cities in each iteration.

4 Experiments and Results

We have tested with the folloing configurations:

- Initial temperature: 100, 500, 1000, 10000
- Cooling rate: 0.1 (fast), 0.5, 0.9, 0.99 (slow)

We have added some of the graphs, and the rest of the graphs are available here, such that the figure name: $\text{final}_s ol_< cool_r ate >_< init_t emp > indicates the map for the final solution and tem$ $init_t emp > indicates the temp and cost vs steps plot.$

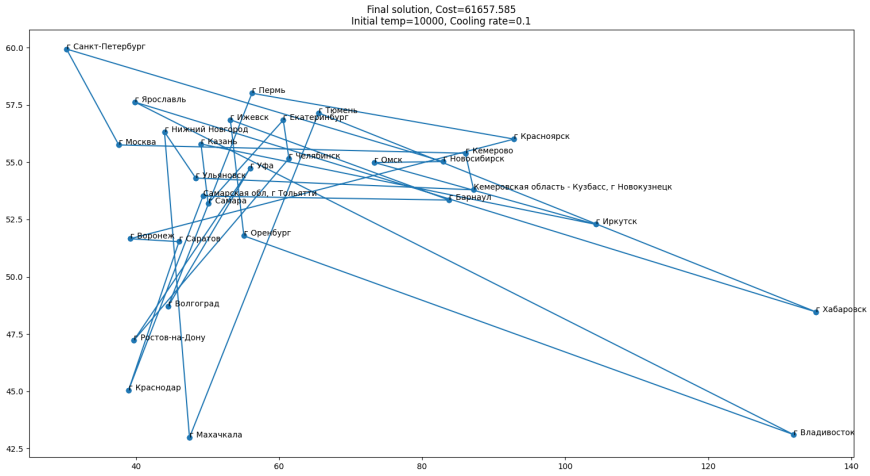

Figure 1: Final solution for 0.1 and initial temp: 10000

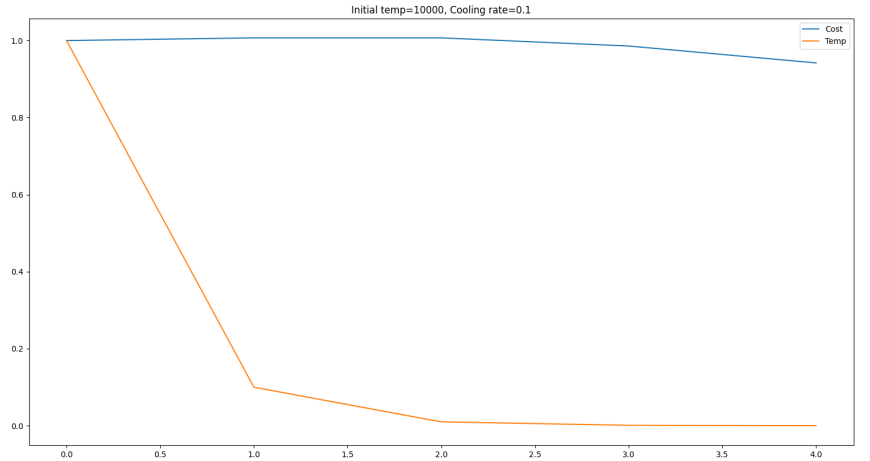

Figure 2: Temperature and Cost 0.1 and initial temp: 10000

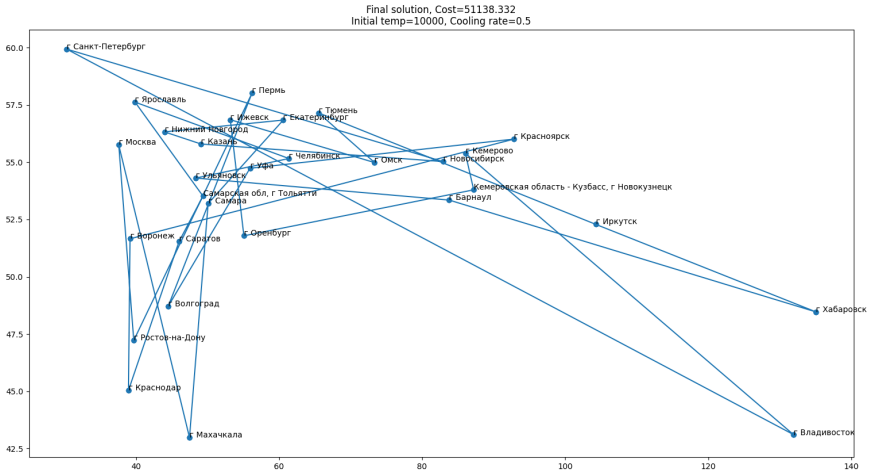

Figure 3: Final solution for 0.5 and initial temp: 10000

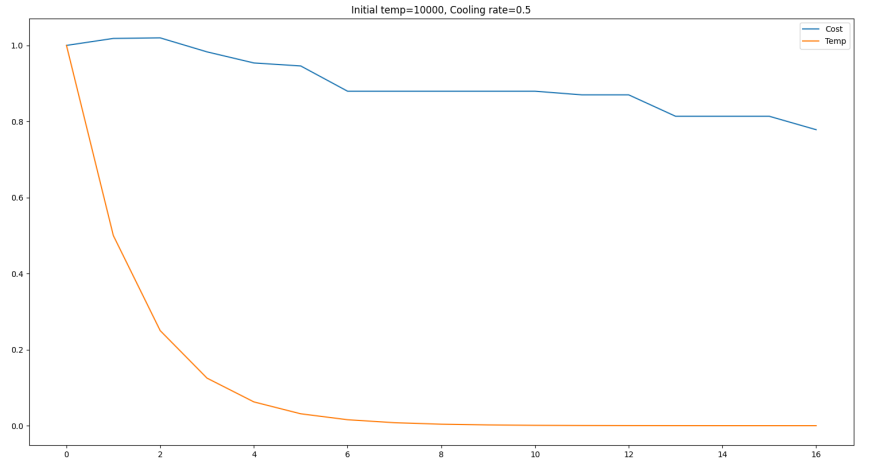

Figure 4: Temperature and Cost 0.5 and initial temp: 10000

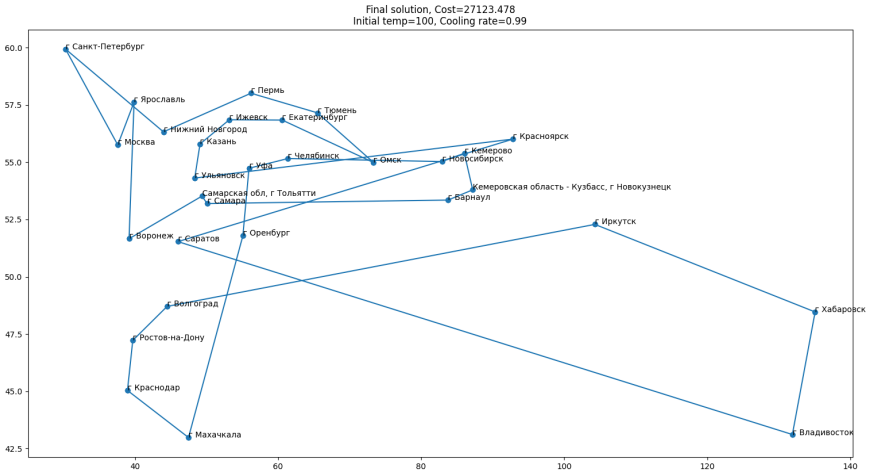

Figure 5: Final solution for 0.99 and initial temp: 100

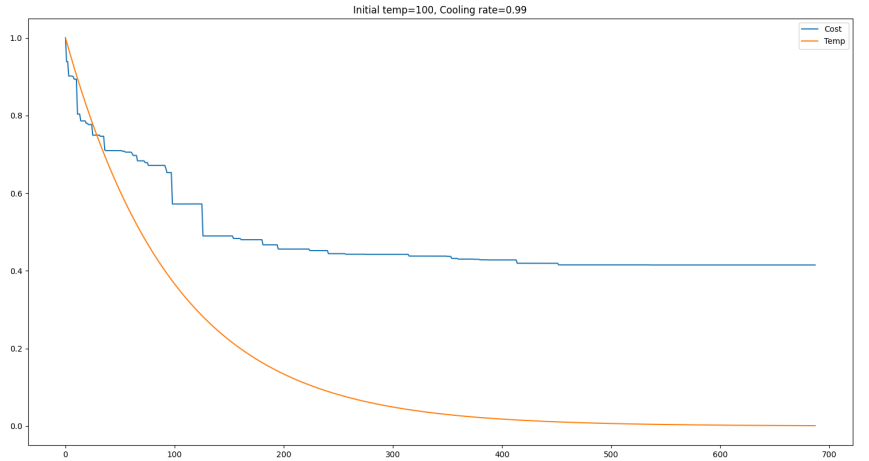

Figure 6: Temperature and Cost 0.99 and initial temp: 100

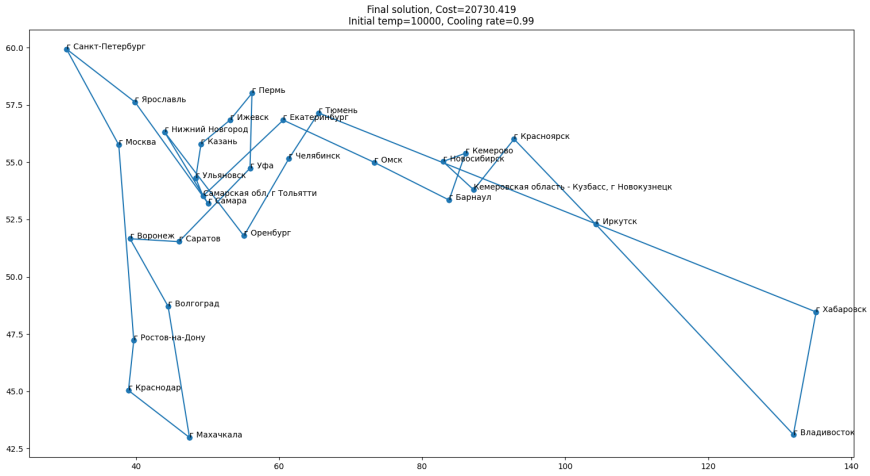

Figure 7: Final solution for 0.99 and initial temp: 10000

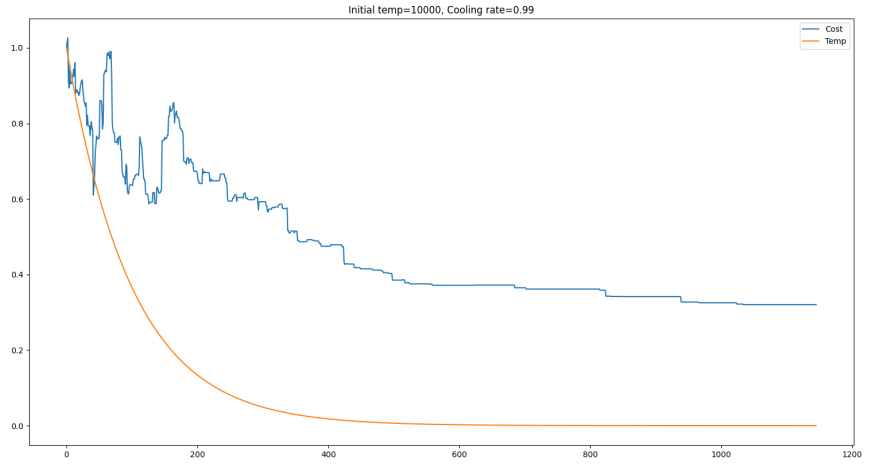

Figure 8: Temperature and Cost 0.99 and initial temp: 10000

Notice that the temperature and the cost functions are normalized in order to fit in the same graph such that the maximum = 1 and minimum = 0

From the previous graphs, we can see the convergence of the cost based on different combinations of temperatures and different cooling rate, and we can see the associated converged optimal distance with the respected parameters.

And the following graphs to indicate the different experiment obtained results.

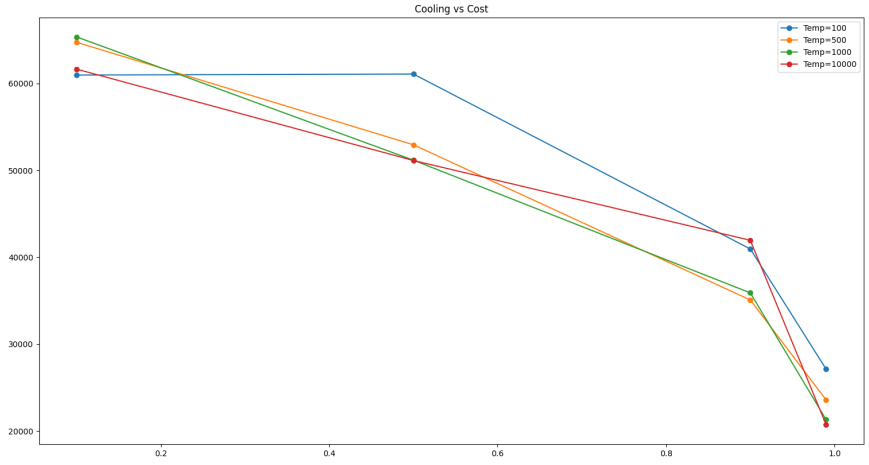

Figure 9: Cooling vs cost with different temperature

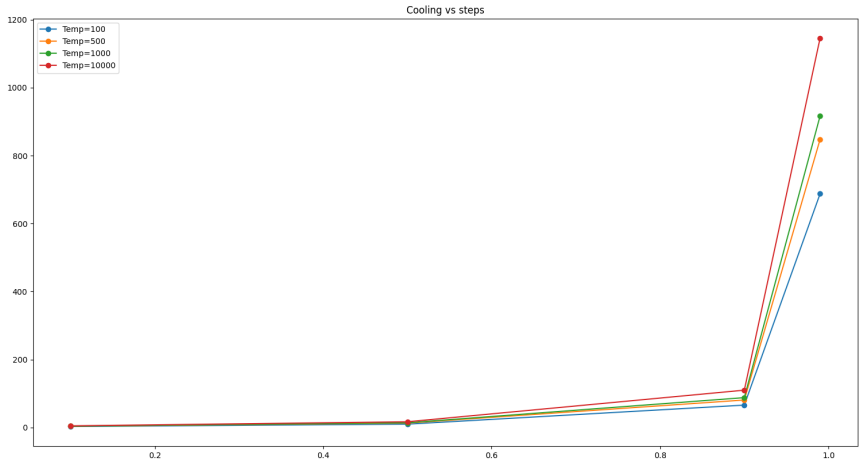

Figure 10: Cooling vs steps with different temperature

Notice that higher initial temperature with slow cooling rate contributes to more number of steps and vice-versa.

Also, you can see an animation of the solved solution while running the code