EE2703: Assignment 6

Nikhil Deepak - ee22b020

October 2023

1 Introduction

1.1 Problem Statement

The Traveling Salesman Problem (TSP) involves finding the shortest possible route that visits a set of cities and returns to the starting city.

1.2 Objective

The objective of this report is to document the implementation of a solution to the TSP using the simulated annealing algorithm.

2 Approach Used (Simulated Annealing)

The approach used to solve the TSP is simulated annealing. This approach is based on the principles of annealing in metallurgy, where a material is heated and then slowly cooled to remove defects and optimize its structure. Similarly, simulated annealing optimizes a solution by starting with a random order of cities, exploring nearby solutions, and probabilistically accepting worse solutions to escape local optima.

3 Implementation

3.1 Data Input

- Read city coordinates from an input file.
- Convert the data into a list of city coordinates.

3.2 TSP Algorithm

- Implement the TSP using simulated annealing.
- Start with a random initial solution (random permutation of city indices).
- Explore neighboring solutions and use temperature-based acceptance criteria.
- Maintain the best solution found.

4 Code/Algorithm explained

4.1 tsp(cities) function

- The function starts by initializing variables, including the current and best solutions, current and best distances, and parameters for simulated annealing.
- The algorithm runs for a specified number of iterations (controlled by num_iterations).

- In each iteration, it randomly swaps two cities in the current solution to explore a neighboring solution.
- It calculates the new distance for the updated solution and evaluates whether to accept or reject it based on temperature and the change in distance.
- The temperature decreases in each iteration, controlled by the cooling rate (decay_rate).
- The algorithm calculates the percentage improvement between the initial random order and the best solution found.
- The function returns the best solution (a list of city indices) and the percentage improvement achieved.

```python
def tsp(cities):
    num_cities = len(cities)
    # Initial solution: a random permutation of city indices
    current_solution = list(range(num_cities))
    best_solution = current_solution.copy()

    current_distance = distance(cities, current_solution)
    best_distance = current_distance

    # Simulated Annealing parameters
    initial_temperature = 2000
    decay_rate = 0.995
    num_iterations = 10000

    temperature = initial_temperature

    for k in range(num_iterations):
        # Randomly swap two cities in the current solution
        i, j = random.sample(range(num_cities), 2)
        new_solution = current_solution.copy()
        new_solution[i], new_solution[j] = new_solution[j], new_solution[i]

        new_distance = distance(cities, new_solution)
        if k==1:
            solution1 = new_distance
        delta_distance = new_distance - current_distance

        # Accept the new solution if it's better or with a certain probability if it's worse
        if delta_distance < 0 or random.random() < math.exp(-delta_distance / temperature):
            current_solution = new_solution
            current_distance = new_distance

            # Update the best solution if needed
            if current_distance < best_distance:
                best_solution = current_solution.copy()
                best_distance = current_distance

        # Cooling schedule
        temperature *= decay_rate

    # Percentage improvement between first random order and best solution
    improvement = (abs(best_distance - solution1) / solution1) * 100
```

```
43
44      return best_solution, improvement
```

Listing 1: Code snippet used for tsp(cities)

4.2 distance(cities, cityorder)

- The function starts by initializing the total distance to 0.
- The function iterates through the city indices specified in the cityorder list.
- For each pair of adjacent cities, it calculates the Euclidean distance using the coordinates of the cities.
- The distance is added to the total distance.
- The function returns the total distance, representing the length of the TSP route.

```
1  def tsp(cities):
2      num_cities = len(cities)
3      # Initial solution: a random permutation of city indices
4      current_solution = list(range(num_cities))
5      best_solution = current_solution.copy()
6
7      current_distance = distance(cities, current_solution)
8      best_distance = current_distance
9
10     # Simulated Annealing parameters
11     initial_temperature = 2000
12     decay_rate = 0.995
13     num_iterations = 10000
14
15     temperature = initial_temperature
16
17     for k in range(num_iterations):
18         # Randomly swap two cities in the current solution
19         i, j = random.sample(range(num_cities), 2)
20         new_solution = current_solution.copy()
21         new_solution[i], new_solution[j] = new_solution[j], new_solution[i]
22
23         new_distance = distance(cities, new_solution)
24         if k==1:
25             solution1 = new_distance
26         delta_distance = new_distance - current_distance
27
28         # Accept the new solution if it's better or with a certain probability if
           ↪  it's worse
29         if delta_distance < 0 or random.random() < math.exp(-delta_distance /
           ↪  temperature):
30             current_solution = new_solution
31             current_distance = new_distance
32
33             # Update the best solution if needed
34             if current_distance < best_distance:
35                 best_solution = current_solution.copy()
36                 best_distance = current_distance
37
38         # Cooling schedule
```

```
        temperature *= decay_rate

    # Percentage improvement between first random order and best solution
    improvement = (abs(best_distance - solution1) / solution1) * 100

    return best_solution, improvement
```

Listing 2: Code snippet used for distance(cities, cityorder)

5 40 city testcase

- Run the cell in the the notebook under the path section to get an optimized path.
- Run the following code cell to get the distance and the percentage improvement.
- And run the last cell for visualisation.

6 Result

I have been getting distances that range from 6 to as high as 8.5 but more often that not it is in the 7-8 range.
The percentage improvement is in the 60-70 range.

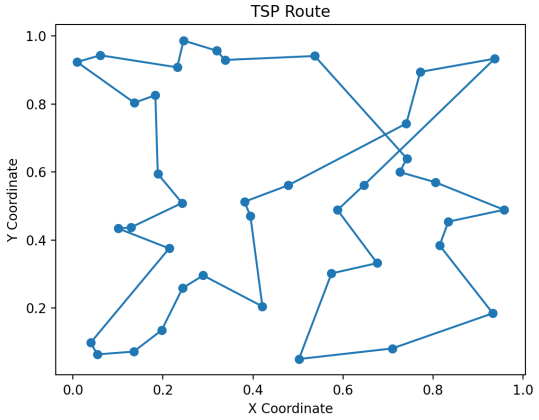

Figure 1: Path for one of the 'good' solutions

7 Conclusion

The approach of using simulated annealing provides a balance between exploration and exploitation to find near-optimal TSP solutions. The percentage improvement calculation helps assess the quality of the solution compared to a random initial order.