CS451/551 – Introduction to Artificial Intelligence

Assignment-2

Solving Travelling Salesman Problem with Genetic Algorithm

Author: Ertuğrul Özvardar / S012366

Introduction

In this assignment, travelling salesman problem was solved by using Genetic Algorithm. Implementing the Genetic Algorithm was the main requirement of this assignment. Because of that, 4 major components of Genetic Algorithm which are crossing-over, mutation, evolving and tournament were implemented in order to solve the TSP. During the process, the interdependence between those components was observed and reflected to the code structure.

Algorithms

Genetic Algorithm (GA)

Mostly, Genetic Algorithms start with a population of feasible solutions to an optimization problem and apply iteratively different operators to generate better solutions. These operators, based on random processes, allow Genetic Algorithms to explore the search space in different directions. In our problem, our Genetic Algorithm evaluates each individual of the population using a fitness function. Most fitted individuals are selected for reproduction with the aim of getting better feasible solutions. The reproduction process includes crossover and mutation operators. The evaluation, selection and reproduction are repeated until some stopping conditions are reached.

The four different components used to develop the genetic algorithm are detailed in the next subsections:

I. Evolve
II. Crossover
III. Mutate
IV. Tournament

Evolve

In this section of our Genetic Algorithm, core idea and desired solution is mainly implemented with the help of other subsections; crossover, mutate, tournament. The aim of this section is constructing new routes in an advanced manner. These advanced routes can be called as evolved routes or evolved paths. Thus, it is basically evolving our predecessor routes to reach the best optimal solution of our problem. As it is mentioned, evolve function is implemented using other three functions. Firstly, we obtained P/2 set of parents as proute_1 and proute_2 from the current population using tournament function. Then our child is created with implementing crossover function to our parent routes. After applying child route to the structure of new routes, we tried to establish new and altered route structure with the help of mutate function. As a result, we obtained the desired routes as a final solution to our problem with repeating those assistive subparts.

Crossover

In this part of our Genetic Algorithm, two parent routes were randomly selected in order to create a superior child route. At the end of this process, new routes that are more likely to be better than the parents were created owing to the exchange of genetic material between parents.

Mutate

In this code fragment of our Genetic Algorithm, we maintained and introduced diversity in the genetic population of our routes. There are different types of mutation functions in Genetic Algorithms. But we implemented the Swap Mutation because it is easily applicable to the mutation rate variable. In our function, we selected two routes at random, and interchanged the values. As a result, we maintained our genetic diversity between route generations.

Tournament

In this section of our Genetic Algorithm, we tried to make an election between candidate routes. To be more specific, we applied a selection strategy to choose the fittest candidates from the current generation. These selected candidates are then passed on to the next generation.

We followed mainly 3 steps:

1) Select k individuals from the population and perform a tournament amongst them.
2) Select the best individual from the k individuals.
3) Repeat process 1 and 2 until you have the desired amount of population.

Results

Different set of routes were obtained in the running process of the algorithm. Since “random” and different variables and hyper-parameters were used in the test phase, it was expected to encounter distinct results as different set of routes.

For instance, we encountered different results from the obtained outcome scenarios.

Default Scenario
(population_size=50, mutation_rate=0.2, tournament_size=10, elitism=False)

```
1905.2715721412055
1644.3317021194043

Route:
City_3(140, 180) --> City_7(140, 140) --> City_12(120, 80) --> City_18(60, 20) --
> City_14(20, 40) --> City_8(40, 120) --> City_17(20, 20) --> City_15(100, 40) --
> City_4(20, 160) --> City_9(100, 120) --> City_6(200, 160) --
> City_5(100, 160) --> City_11(60, 80) --> City_19(160, 20) --
> City_16(200, 40) --> City_10(180, 100) --> City_13(180, 60) --
> City_2(80, 180) --> City_0(60, 200) --> City_1(180, 200)
```

Scenario II. (population_size=75, mutation_rate=0.2, tournament_size=10, elitism=False)

```
1925.7364054112766
1673.0788230926087

Route:
City_3(140, 180) --> City_18(60, 20) --> City_17(20, 20) --> City_19(160, 20) --
> City_15(100, 40) --> City_14(20, 40) --> City_16(200, 40) --
> City_13(180, 60) --> City_10(180, 100) --> City_9(100, 120) --
> City_8(40, 120) --> City_4(20, 160) --> City_12(120, 80) --> City_6(200, 160) -
-> City_1(180, 200) --> City_7(140, 140) --> City_5(100, 160) --
> City_2(80, 180) --> City_11(60, 80) --> City_0(60, 200)
```

Observation 1: According to the results obtained from Default Scenario and Scenario II, we can conclude that if all of the hyper-parameters except population_size remain unchanged and if we increase the value of population_size, our algorithm tends to find a longer path. So, we cannot optimize our algorithm with incrementing the value of our hyper-parameter; population_size.

Scenario III. **(population_size=50, mutation_rate=0.3, tournament_size=10, elitism=False)**

```
1781.699365169133
1774.2977831242679

Route:
City_17(20, 20) --> City_4(20, 160) --> City_11(60, 80) --> City_1(180, 200) --
> City_7(140, 140) --> City_12(120, 80) --> City_16(200, 40) --
> City_6(200, 160) --> City_0(60, 200) --> City_2(80, 180) --> City_8(40, 120) --
> City_5(100, 160) --> City_10(180, 100) --> City_3(140, 180) --
> City_9(100, 120) --> City_15(100, 40) --> City_18(60, 20) --
> City_19(160, 20) --> City_13(180, 60) --> City_14(20, 40)
```

Observation 2: According to the results obtained from Default Scenario and Scenario III, we can conclude that if all of the hyper-parameters except mutation_rate remain unchanged and if we increase the value of mutation_rate, our algorithm tends to find a longer path. So, this cannot lead to optimize our algorithm with incrementing the value of our hyper-parameter; mutation_rate.

Scenario IV. **(population_size=50, mutation_rate=0.2, tournament_size=20, elitism=False)**

```
1660.0980107439714
1545.6433531353387

Route:
City_0(60, 200) --> City_5(100, 160) --> City_2(80, 180) --> City_1(180, 200) --
> City_3(140, 180) --> City_16(200, 40) --> City_9(100, 120) --
> City_13(180, 60) --> City_19(160, 20) --> City_8(40, 120) --> City_17(20, 20) -
-> City_11(60, 80) --> City_14(20, 40) --> City_18(60, 20) --> City_15(100, 40) -
-> City_12(120, 80) --> City_10(180, 100) --> City_6(200, 160) --
> City_7(140, 140) --> City_4(20, 160)
```

Observation 3: According to the results obtained from Default Scenario and Scenario IV, we can conclude that if all of the hyper-parameters except tournament_size remain unchanged and if we increase the value of tournament_size, our algorithm tends to find a shorter path. So, this can lead to optimize our algorithm with incrementing the value of our hyper-parameter; tournament_size.

Scenario V. **(population_size=50, mutation_rate=0.2, tournament_size=10, elitism=True)**

```
1712.363464247331
1136.2875260813314

Route:
City_0(60, 200) --> City_4(20, 160) --> City_6(200, 160) --> City_1(180, 200) --
> City_3(140, 180) --> City_9(100, 120) --> City_8(40, 120) --
> City_12(120, 80) --> City_15(100, 40) --> City_11(60, 80) --> City_14(20, 40) -
-> City_17(20, 20) --> City_18(60, 20) --> City_19(160, 20) --
> City_16(200, 40) --> City_13(180, 60) --> City_10(180, 100) --
> City_7(140, 140) --> City_5(100, 160) --> City_2(80, 180)
```

Observation 4: According to the results obtained from Default Scenario and Scenario V, we can conclude that if all of the hyper-parameters except elitism remain unchanged and if we set the value of elitism as True, our algorithm tends to find a shorter path. So, this can lead to optimize our algorithm with changing the value of our hyper-parameter; elitism.

Final Scenario **(population_size=30, mutation_rate=0.1, tournament_size=20, elitism=True)**

```
1983.7202504617032
897.530521988939

Route:
City_6(200, 160) --> City_1(180, 200) --> City_3(140, 180) --> City_7(140, 140) -
-> City_5(100, 160) --> City_2(80, 180) --> City_0(60, 200) --> City_4(20, 160) -
-> City_8(40, 120) --> City_9(100, 120) --> City_12(120, 80) --
> City_11(60, 80) --> City_14(20, 40) --> City_17(20, 20) --> City_18(60, 20) --
> City_15(100, 40) --> City_19(160, 20) --> City_16(200, 40) --
> City_13(180, 60) --> City_10(180, 100)
```

In the Final Scenario, we analyzed the results from the 4 observations. Then, we adjust our hyper-parameters in order to obtain the best optimal solution as optimized set of routes. As a result, we end up with a desired solution to our problem.

Implementation Details

As it is mentioned in the introduction, there was a dependence and interrelation among those four functions. Since our evolve function is composed of our three functions, it was impossible to implement it initially. Hence, I started with crossover function. After implementing this, I implemented mutation and tournament functions respectively. Finally, I created the evolve function with the help of other three components.

Summary

As a conclusion, 4 different functions such as evolve, crossover, mutate and tournament were implemented as components of a Genetic Algorithm to solve Travelling Salesman Problem. Consequently, we end up different types of results as set of routes in each running phase of the code.