20160025 고건호

CS454 AI Based Software Engineering

Assignment #2: Stochastic Optimisation

Due by 23:59, 4 October 2021

Problem Description

These days, people are more likely to catch taxi with mobile application such as 'kakao taxi', rather than just waiting. Currently, taxi drivers can choose which passengers to pick up, but we can imagine all matching will be achieved automatically in the near future. Time complexity to gain the exact solution of the problem (complete search algorithm) is O(n!), which means it is almost impossible to get the right answer in time. Greedy algorithm may work, but sometimes it is not enough. I adopted genetic algorithm to suggest effective matching algorithm that leads to better result than greedy algorithm, but takes less time than complete search algorithm.

Class

There are three classes in the solver, which are 'Customer', 'Taxi', and 'World'

Customer

Customer has attributes that represents the initial location(*src*), destination(*dest*), waiting time since they called for taxi(*wait*), and distance between initial location and destination(*distance*). Src and dest is a tuple with x and y value.

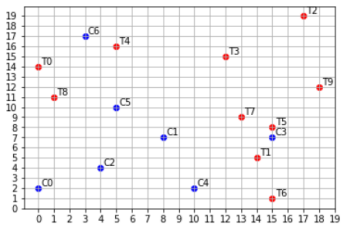
Taxi

Each taxi can be picked up by one group of passengers who have same source and destination (*passenger*), and each taxi can be allocated to at most one customer (or one group of customers) (*allocate*). A taxi can have both passenger and allocated customer, so that it can take the passenger first and move to allocated customer. Taxi can earn money in proportion to the distance traveled by passenger(*earn*). It is initialized with its initial location(*loc*)

World

World are initialized with world size, customers, and taxis in the world. You can give the location of taxis and customers, or just make random taxis and customers with given numbers(*num_taxis, num_customers*). World contains three different algorithms that solve their best : complete, greedy, genetic.

```
world = World(world_size=(20, 20), random=True, num_taxis=10, num_customers=7)
world.show("plain")
```
✓ 0.2s



```
customers = [(2, 2), (5, 5), (8, 6), (3, 7), (10, 9), (11, 7)]
dests = [(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
taxis = [(1, 9), (2, 9), (8, 8), (9, 3), (11, 2), (6, 1), (4, 2), (6, 1)]

world = World(world_size=(15, 15), random=False)
world.taxis(taxis)
for c, d in zip(customers, dests):
    world.add(c, d)
world.show("plain")
```
✓ 0.1s

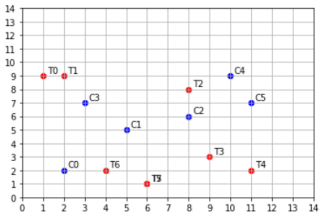


Figure 1 Two ways of initializing the World

Solver

There are three solvers in class World, in order to compare the performance among the solvers. The solution of each algorithm will be returned into list type, whose length is number of taxis. It represents the matching between taxis and customers. If an element of the list is −1, it means the taxi are not allocated to a customer yet.

```
world.greedy()
```
✓ 0.4s

```
[-1, 3, 2, 4, 5, 1, 0, -1]
```

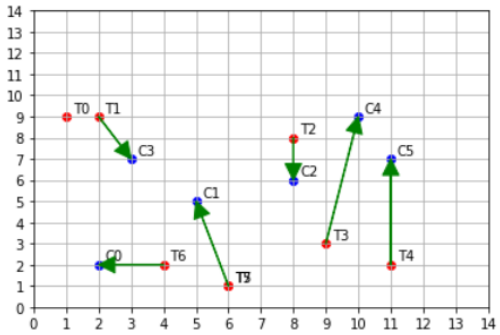


Figure 2 Solution of greedy algorithm solver and matching diagram

Fitness Function

The first goal of the solver is to minimize the total waiting time of customers. Also, it is important to guarantee the fairness to both customers and taxis. As a result, variance of waiting time of customers and variance of earned money of taxis will be added. Final goal of the solver is to minimize the fitness value.

$$fitness = (\sum cus.wait\_time)^2 + var(cus.wait\_time) + var(taxi.earned)$$

Complete Search Algorithm Solver

Complete Search Algorithm Solver will search every possible combination between taxis and customers, and choose the sequence that minimize the fitness function. The solver has to compute fitness value N!/(N-M)! (N: number of taxis, M: number of customers) times, so time complexity of the solver is O(n!).

Greedy Algorithm Solver

In Greedy Algorithm Solver, every customer is allocated to its closest available taxi. 'closest' means the shortest distance to go until the taxi reaches the customer. If there is a passenger in the taxi, it is the sum of the distance that takes the passenger to the destination and the distance travels to the new customer.

Genetic Algorithm Solver

It gets the 'max_generation', 'selected_num' as input. The first generation is number of mutated greedy solutions. In the beginning of every generation, individuals are selected by their fitness value. Selected parents generate new children by conducting crossover with each other, and each child are mutated. If there are same solutions in one generation, only one solution remains. Repeats these steps for each generation.

Crossover

In this case, each individual is array type. And they cannot have a duplicate numbers except '-1'. As a result, cross over will take place in the following process.

1. Choose cross-point among the indices that the elements of two parents are different.
2. The next cross-point is determined such that the second element of the current cross-point is equal to the first element of the next cross-point.
3. Repeat step2 until the second element of last cross-point is equal to first element of first cross-point.
4. Switch the elements on the selected cross-point.

Figure 3 Crossover Strategy

Mutation

For each element in one individual, the probability of mutation is 1/L (L: length of solution). When mutation occurs, it switches element with another element in random location.

Figure 4 Mutation Strategy

Result

To test the performance of three Solvers, I first generated a small world with size 20x20, number of taxis 10, number of customers 7.

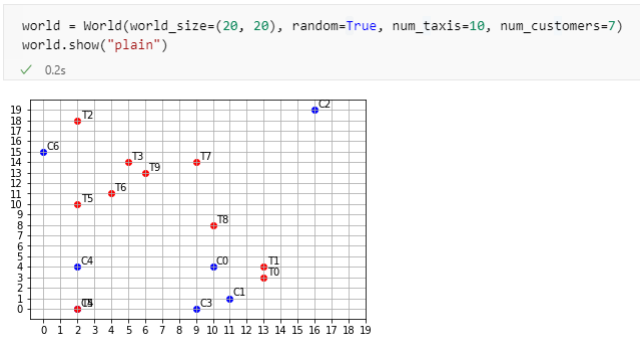


Figure 5 small world generation

It took 33 seconds to run Complete Search Algorithm Solver. Greedy Algorithm Solver took only 0.2 seconds in running, but its fitness value and total waiting time is bit higher than Complete Search Algorithm Solver.

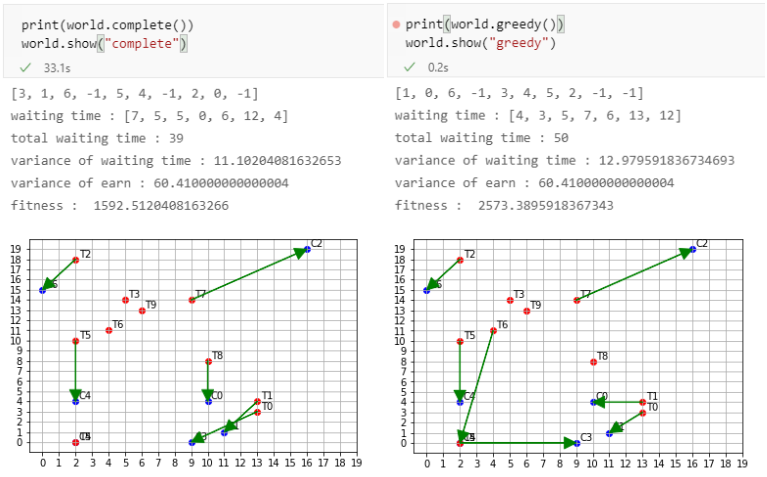


Figure 6 Complete Search Algorithm Solver and Greedy Algorithm Solver for small world

Figure 7 is a graph of fitness value for generation. We can see its fitness value goes below Greedy Algorithm Solver's fitness value around 20th generation, and reaches fitness value of Complete Search Algorithm Solver before 30th generation. And it took 0.3 seconds, which is not far ahead of Greedy Algorithm solver's running time.

```
print(world.genetic(graph=True, max_generation=50, selected_num=4))
world.show("genetic")
```

✓ 0.3s

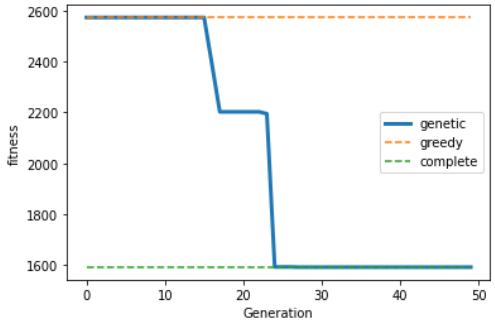


Figure 7 fitness value of Genetic Algorithm Solver compared to Complete and Greedy Solver in small world

I tried bigger world with size 50x50, number of taxis 50, number of customers 40. From now on, I won't use Complete Search Algorithm Solver anymore, because it takes too much time in running. Figure 8 is the solution of Greedy Solver and Genetic Solver(max_gen=200, selected_num=10). We can visually confirm that the Genetic Algorithm Solver is more efficient than Greedy Algorithm Solver. It showed 22% decrease in fitness value, and 11.5% decrease in total waiting time.

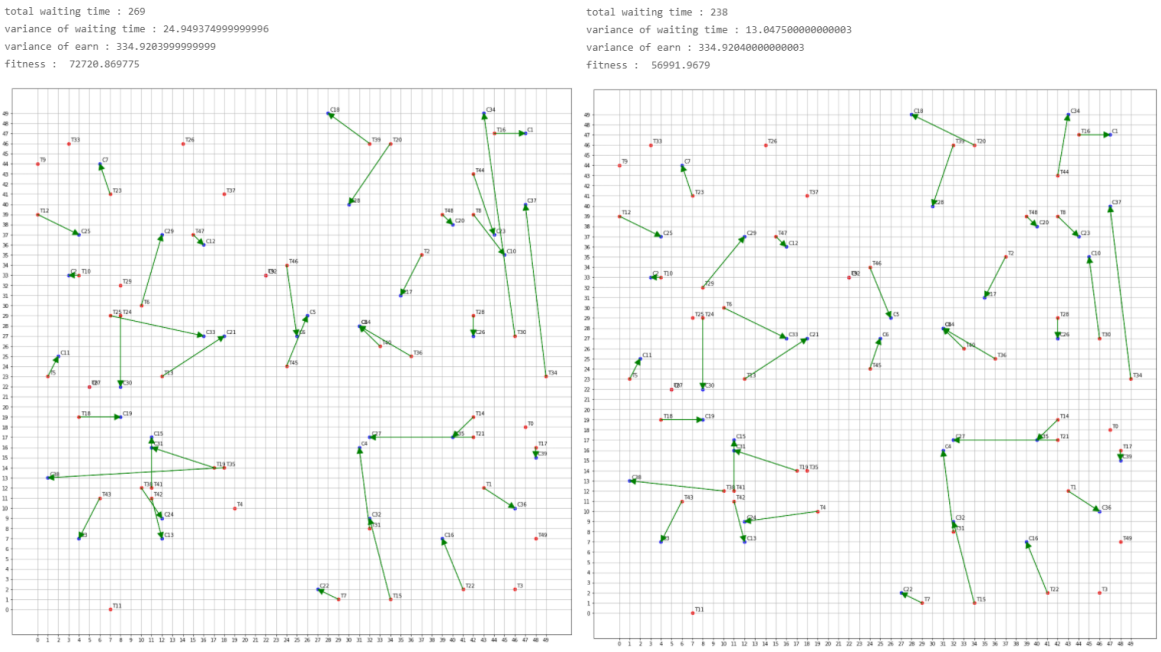


Figure 8 Greedy Algorithm Solver(left) and Genetic Algorithm Solver(right) of big world.

Figure 9 is a graph of fitness value of Genetic Algorithm Solver compared to Greedy Algorithm Solver. It seems to reached optimal point before 100th generation. Running time of genetic algorithm solver is now 3.3 seconds.

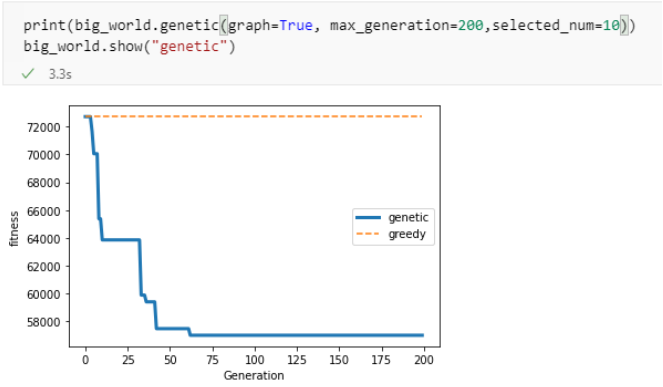


Figure 9 fitness value of Genetic Algorithm Solver compared to Complete and Greedy Solver in big world

Finally, I tested solver in huge world with size 200x200, number of taxis 500, number of customers 350. Max_generation and selected_num of Genetic Algorithm Solver was 10,000 and 15. It seems to reach optimal solution around 8000th generation, and it took 1354 seconds to improve solution. Improvement of the solution was 18.8% decrease in fitness value, and 9.9% decrease in total waiting time.

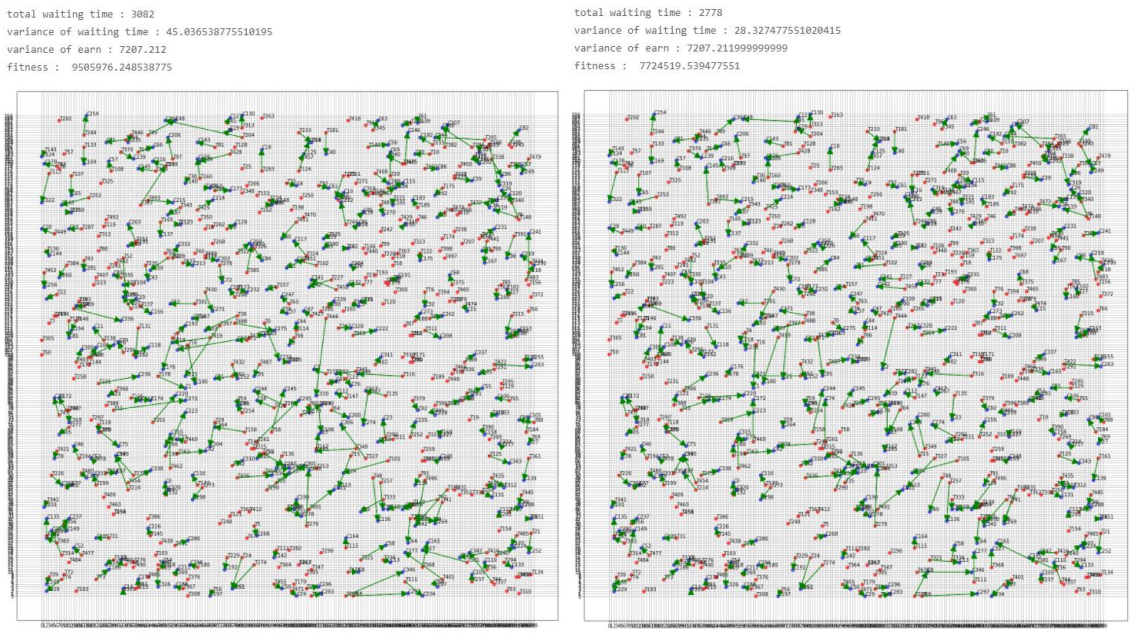


Figure 10 Greedy Algorithm Solver(left) and Genetic Algorithm Solver(right) of huge world.

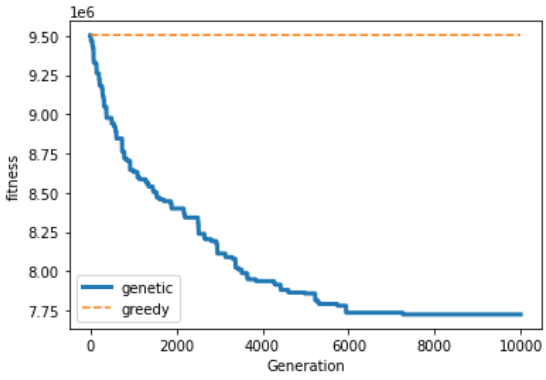


Figure 11 fitness value of Genetic Algorithm Solver compared to Complete and Greedy Solver in huge world

Conclusion & Discussion

Genetic Algorithm showed very surprising results more than I expected. It reached optimisation point very quickly when N(number of taxis) is small. However, it took considerably large amount of time to make a noticeable improvement when N is large. Also, finding appropriate hyperparameters, max_generation and selected_num was difficult. The values I used in experiment is “best practices”. I couldn’t find a nice relationship between N and appropriate hyperparameters.

However, there is still room for positive thinking. Genetic Algorithm Solver is not a complete independent solver, but it is a process of improving solution. So when the initial solution is good enough, it will take a few time to get to the optimal solution. More specifically, taxi-customer matching is not a static problem. Taxis are moving, and a new customers can appear at any time, but a new solution will not be changed dramatically from the previous solution. So when we feed a previous solutions as individuals of initial generation, effectiveness of Genetic Algorithm Solver will be even better than now.