Artificial Intelligence Concepts

Assignment1

HONG Yifan
22043798g

GitHub Repository:
https://github.com/EvanHong99/GeneticAlgorithm.git

A COMP5511 Homework Assignment

THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

October 21, 2022

1 Introduction

1.1 Travelling Salesman Problem (TSP)

The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" [5]

1.2 Genetic Algorithm (GA)

A genetic algorithm is a search heuristic that is inspired by Charles Darwinâ€™s theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. The algorithm generally includes procedures as below [4]:

- Selection: The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation.
- Crossover: Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. And generate two children by swapping genes.
- Mutation: To simulate the mutation in natural environment, GA allows gene to mutate under a certain probability, which introduce new vivid genes into population.

1.3 Introduction to My Code

To ensure low coupling of the code, I imitated the code structure of ‘geatpy‘.

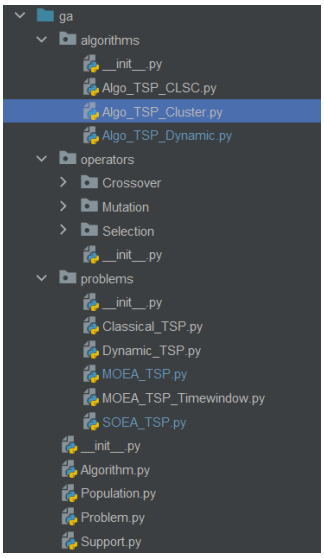


Figure 1: project structure

- Class - ‘Algorithm‘: records program status, information, and executes genetic operations.
- Class - ‘Population‘: defines chromosomes and preserve basic information such as the coordinate of each customer.
- Class - ‘Problem‘: defines fitness functions and objective functions, as a consideration that different problem will have different targets, shortest distance or highest profits, for example.
- subpackage ‘algorithms‘: specific implementation of ‘Algorithm‘.
- subpackage ‘problems‘: specific implementation of ‘problems‘.
- subpackage ‘operators‘: specific implementation of ‘operators‘.

2 Methodology

```
START
Generate the initial population
Compute fitness and other information
REPEAT
    Reintroduce elites
    Weighted Fitness Selection
    Crossover
    Mutation
    Compute fitness
    Preserve elites
UNTIL population has converged or reaches max generation
STOP
```

General Pseudocode

2.1 Encoding

Chromosomes are encoded by Customers' NO.

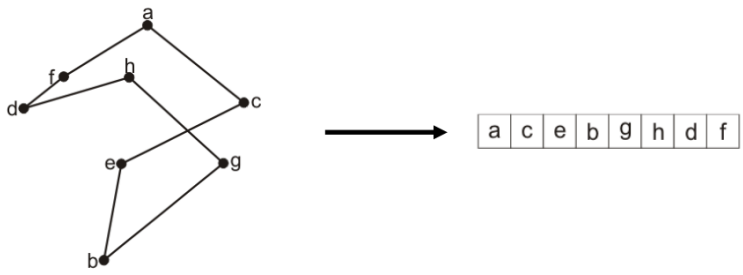


Figure 2: Encoding

2.2 Task1: Classical TSP

2.2.1 Fitness

In Classic TSP, lower distance means better fitness score, so I use the inverse of distance to represent fitness.

Coefficient 1e6 makes fitness closer to 0, preventing from losing precision.

$$
\begin{aligned}
coefficient &= 1e6 \\
total\_distance &= \Sigma_{i=0}^{n} EuclideanDistance(customer_i, customer_{i+1})^2 \\
fitness &= coefficient/total\_distance
\end{aligned}
$$

Note that I use the sum of squared distances to calculate total distance. This is better performed than simple summation of distance

$\Sigma_{i=0}^{n} EuclideanDistance(customer_i, customer_{i+1})$, because the former will urge our algorithm to prevent from long distance paths.

2.2.2 Selection

Weighted Fitness

Calculate the fitness of each chromosome, and the greater chance to be selected if the fitness is larger.

Note that in consequence of the transformation function $fitness = coefficient/distance$, the transform is nonlinear, so it is easier for my program to converge.

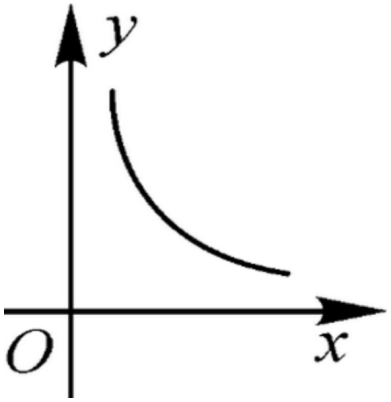


Figure 3: fitness

Elitism

Select individuals with the highest fitness, reintroduce them by replacing the worst individuals in the next loop.

Higher proportion of elites means our algorithm can reach the local optimum faster, but lower probability to jump out of local optimum.

2.2.3 Crossover - Recombination

My algorithm is the same as Professor showed in the class.

I add some features that the probability of crossover and proportion of genes are self-defined, so it will be easier to converge.

2.2.4 Mutation - Swap

Classic TSP not allows repeating points, so swap two genes is a good way to perform mutation.

Also, a parameter 'pom(probability of mutation)' is added, in order to control the program.

2.2.5 Objective Function

To evaluate whole population.

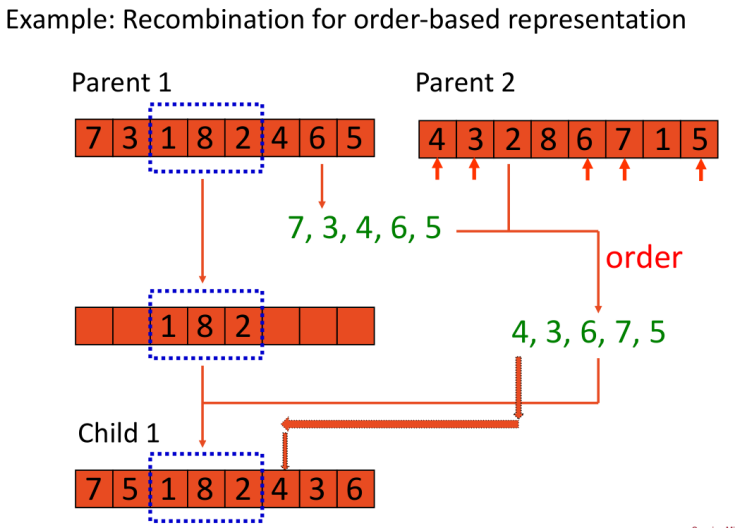


Figure 4: recombination

I choose mean of fitnesses to represent it.

2.3 Task2: Dynamic TSP

2.3.1 Fitness

Same as previous task.

2.3.2 Operators (Selection, Crossover, Mutation)

Same as previous task.

2.3.3 Objective Function

Same as previous task.

Because of the objective function is the mean of fitness, so it is scale insensitive, which means that the obj value is comparable with task1.

2.3.4 Changing Customers

```
class Algo_TSP_Dynamic(Algorithm):
    def run(self, if_reuse=False):
        ...
            if generation % 100 == 0 and env <= 5:
                if env != 0:
```

```
            if if_reuse: # do not regenerate
             ↪  individuals
                self.population.add_individuals(10)
                self.population.update_info_env(env)

                 ↪  self.problem.init_cost(self.population.info)
            else: # regenerate individuals
                self.population.pop_size += 10
                self.population.chrom_len += 10
                self.population.init_info() #
                 ↪  regenerate individuals
                self.population.init_individuals()
                self.population.update_info_env(env)

                 ↪  self.problem.init_cost(self.population.info)
            env += 1
```

2.4 Task3: Large-scale Optimization Problem

Sometimes large-scale problem will cause exceptions such as overflow and low computing efficiency. We can solve this problem by using larger data type, but this will extremely increase storage demand.

So the idea of map and reduce comes out. There are many advantages of decomposing a large-scale problem. First, decomposition can reduce the problem scale and avoid precision overflow. Second, parallel computing is allowed to accelerate program.

2.4.1 Task decomposition

I choose K-means as the clustering technique. The result of clustering will affect chromosome encoding. Customers at the same cluster will construct a gene segment.

2.4.2 Encoding

Slightly adjust original encoding by adding cluster constrains. For example:

```
[0,1,2,3,4,5,6,7] -> [[0,1],[2,3,4],[5],[6,7]]
```

Where '[0,1]' means customer0 and customer1 are clustered in the first region. And the only path from region1 to region2 is 1- >2, so my program obeys the requirements that the salesman only can go to next region after visited all the customers in this region.

Besides, every individual's chromosome clusters are the same, which means we only improve our solution under the same cluster result.

e.g. Population is consist of four clusters

```
individual1: [[0,1],[2,3,4],[5],[6,7]]
individual2: [[3,2,4],[0,1],[5],[7,6]]
individual3: [[5],[6,7],[3,2,4],[1,0]]
...
```

2.4.3 Fitness

The same as task 1.

One difference is that when calculating fitness, we need to flatten the chromosome.

2.4.4 Selection

Weighted fitness selection and elitism are the same as task1.

2.4.5 Crossover - ClusterRecombination

Chromosome Level Crossover

```
[[0,1],[2,3,4],[5],[6,7]] x [[2,3,4],[6,7],[1,0],[5]]
-> [[1,0],[2,3,4],[5],[6,7]] x [[2,3,4],[6,7],[0,1],[5]]
```

Gene Level Crossover

```
[[0,1],[2,3,9,4,8],[5],[6,7]] x [[2,8,3,4,9],[6,7],[1,0],[5]]
-> [[0,1],[2,3,8,4,9],[5]},[6,7]] x [[2,9,3,4,8],[6,7],[1,0],[5]]
```

2.4.6 Mutation - ClusterSwap

Chromosome Level Mutation Simply swap two clusters' order.

```
[[0,1],[2,3,4],[5],[6,7]] -> [[0,1],[5],[6,7],[2,3,4]]
```

Gene Level Mutation

Simply swap two customers' order. [[0,1],[2,3,4],[5],[6,7]] - > [[0,1],[4,3,2],[5],[6,7]]

2.4.7 Objective Function

Same as task1.

2.5 Task4: Multi-objective Optimization Problem

```
Note: To get familiar with high performance package 'geatpy',
↪ the rest of two tasks are implemented by it.
```

2.5.1 Fitness

The multi-objective optimization problem can be formulated as $<min\ distance,\ max\ profit>$.

Transform it to single-objective optimization problem

$$min(distance - \lambda profit)$$

The advantage of this method is very simple to adapt from other SOEAs (Single-Objective Evolutionary Algorithms). I choose $soea\_SEGA\_templet$ to solve this problem, which implements simple GA with elitism.

```
def __init__(self, testName, fpath):
    M=1 # objective function dimension
    ...

def evalVars(self, Vars):
    # objective function implementation
    profit=np.sum(self.data.loc[X[i],"PROFIT"])
    ObjV.append(distance-self.weight*profit)
```

Multi-objective optimization problem

```
def __init__(self, testName, fpath):
    M=2 # objective function dimension
    maxormins = [1, -1] # tell the program that we need to
    ↪ minimize the first target and maximize the second.
    ...

def evalVars(self, Vars):
    ...
    all_dist = []
```

```
    all_profit = []
    for i in range(N):
        ...
        distance = np.sum(np.sqrt(np.sum(np.diff(journey.T) **
        ↪  2, 0)))
        profit = np.sum(self.data.loc[X[i], "PROFIT"])
        all_dist.append(distance)
        all_profit.append(profit)
    all_dist=np.array([all_dist]).T
    all_profit=np.array([all_profit]).T
    # objective function implementation
    f = np.hstack([all_dist, all_profit])
```

2.5.2 Inspection of Algorithm *soea_SEGA_templet*

```
START
Generate the initial population
Compute fitness and other information
REPEAT
    Preserve elites (which is the whole population)
    Weighted Fitness Selection
    Crossover
    Mutation
    Gather parents and children, then the population is 2N
    Compute fitness
    Select N individuals as next generation
UNTIL population has converged or reaches max generation
STOP
```

General Pseudocode

I am glad to see differences between my code and the official code (highlighted in red), which enlightens my mind. They just preserve all the parents as elites.

```
offspring = population[ea.selecting(self.selFunc,
↪  population.FitnV, NIND)] # NIND==population.size
```

So larger crossover ratio and mutation ratio are allowed without making the program divergent. But my program only preserves 10% of best parents. Even though it decreases the computing power, but more generations I have to iterate and I need to consider the problem of divergence.

Let's take a look of the details of their code.

Selection - Elites Tournament

```
START
Select n individuals (elites included)
Select the best one and return it
FINISH
```

General Pseudocode

Crossover - Partially matched crossover [2]

```
START
Partially change genes of parents
WHILE conflict:
    replace it with the gene on the same index of another chromosome
FINISH
```

General Pseudocode

Mutation - Invertion mutation [3]

```
START
Randomly choose a chromosome segment
Reverse it
FINISH
```

General Pseudocode

2.5.3 Inspection of Algorithm *moea_NSGA2_templet [1]*

This algorithm adopts *Elites Tournament*, *Partially matched crossover* and *Invertion mutation* as selection, crossover and mutation operators as well.

However, this algorithm will consider the diversity of individuals.

```
class moea_NSGA2_templet(ea.MoeaAlgorithm):
    def reinsertion(self, population, offspring, NUM):
        dis = ea.crowdis(population.ObjV, levels)
```

2.6 Task5: Time Window Constraint Problem

Similar to task4, we just need to calculate time penalty and add it to the optimization targets.

```
class MOEA_TSP_Timewindow(ea.Problem):
    def evalVars(self, Vars):
        ...
```

```
        all_time_penalty=[]
        for i in range(N):
            ...
            time_window_l=self.data["READY_TIME"]
            time_window_r=self.data["DUE_TIME"]
            distance_intervals=np.sqrt(np.sum(np.diff(journey.T)
            ↪  ** 2, 0))
            time_penalty=0
            current_time=0
            for i in range(len(distance_intervals)):
                current_time+=distance_intervals[i]
                temp=time_window_l[path[i + 1]]-current_time
                if temp>0:
                    time_penalty+=temp
                    continue
                else:
                    temp=current_time-time_window_r[path[i + 1]]
                    if temp>0:
                        time_penalty+=temp
                        continue
            ...
            all_time_penalty.append(time_penalty)
            ...
        all_time_penalty=np.array([all_time_penalty]).T
        f = np.hstack([all_dist, all_profit,all_time_penalty])
        return f
```

3 Experimental Results

3.1 Task1: Classical TSP

No.	target	required	completed
1	find the shortest round-trip route of these 100 customers	✓	✓
2	visualize the round-trip route	✓	✓
3	calculate total distance	✓	✓

I did a grid search to find the local optimum parameters. Each parameter set was tested three times.

Table 1: Grid Search

	pos	poc	proportion	pom	distance
0	0.05	0.05	0.4	0.1	588.515705
1	0.05	0.05	0.4	0.1	613.043183
2	0.05	0.05	0.4	0.1	559.741788
3	0.05	0.05	0.4	0.2	628.880424
4	0.05	0.05	0.4	0.2	597.325714
			...		
45	0.10	0.10	0.5	0.2	524.917625
46	0.10	0.10	0.5	0.2	569.319894
47	0.10	0.10	0.5	0.2	562.480359

Take mean of these parameter sets.

Table 2: Grid Search Mean

				distance
pos	**poc**	**proportion**	**pom**	
0.05	**0.05**	**0.4**	**0.1**	587.100225
			0.2	609.131925
		0.5	**0.1**	555.574904
			0.2	593.422366
	0.10	**0.4**	**0.1**	543.343484
			0.2	554.606586
		0.5	**0.1**	529.843958

Table 2 continued from previous page

				distance
			0.2	575.513898
0.10	**0.05**	**0.4**	**0.1**	586.246860
			0.2	604.424385
		0.5	**0.1**	629.545166
			0.2	605.295807
	0.10	**0.4**	**0.1**	523.340314
			0.2	528.871191
		0.5	**0.1**	519.786874
			0.2	552.239293

From grid search, we can draw the conclusion that (0.1, 0.1, 0.5, 0.1) is optimal parameter.

The shortest round-trip is :

```
[15, 10, 68, 39, 21, 74, 48, 83, 81, 45, 28, 73, 94, 14, 36, 29, 49, 78, 9,
    33, 60, 0, 44, 43, 82, 97, 89, 1, 18, 3, 99, 46, 76, 34, 12, 5, 22, 63,
    66, 30, 71, 16, 55, 91, 84, 86, 20, 25, 92, 41, 93, 70, 4, 38, 31, 53, 69,
     58, 80, 75, 7, 23, 88, 56, 64, 47, 37, 24, 42, 2, 6, 79, 72, 90, 32, 85,
    8, 62, 96, 95, 11, 65, 67, 87, 57, 98, 17, 61, 19, 51, 35, 52, 50, 54, 13,
     26, 27, 59, 40, 77]
```

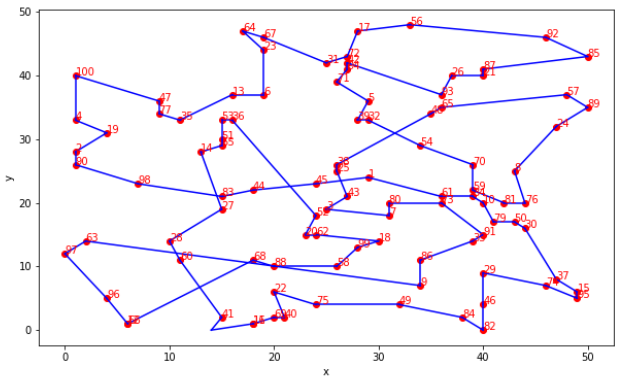
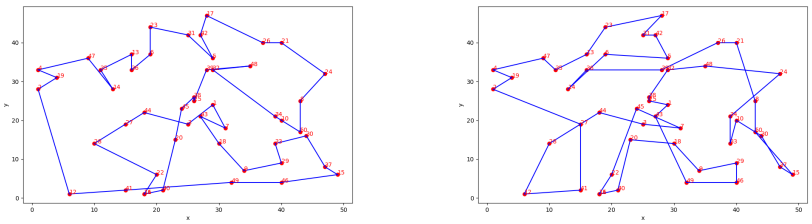


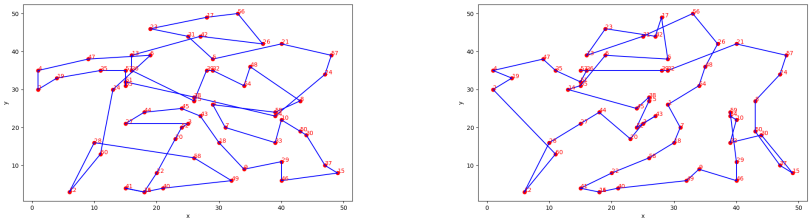
Figure 5: task1_best

3.2 Task2: Dynamic TSP

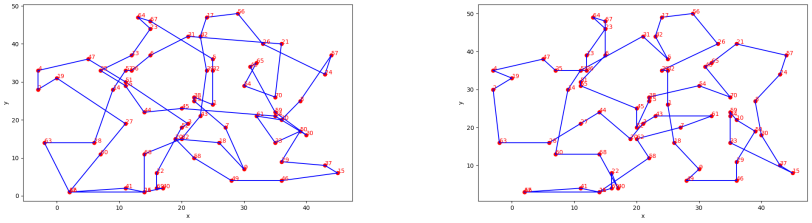
I didn't run too many generations, it is just an illustration of adding new customers into our problem.



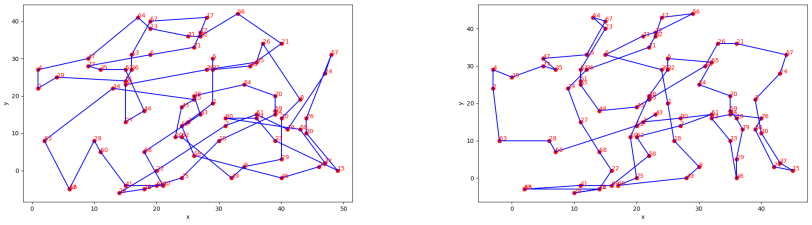
1000 generation



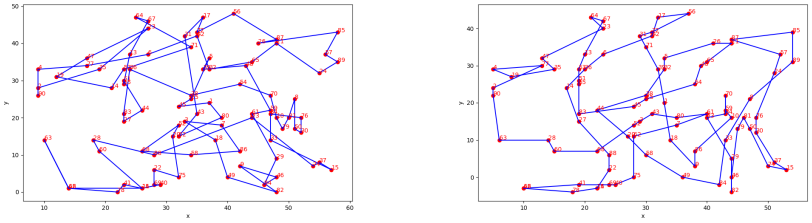
2000 generation



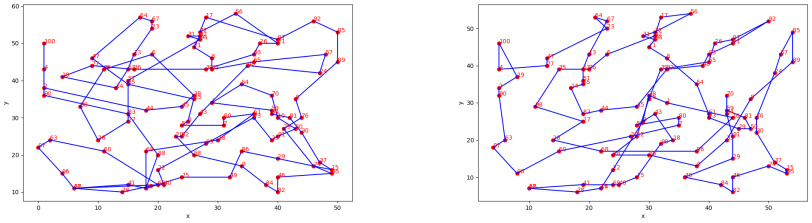
3000 generation



4000 generation



5000 generation



7000 generation

Figure 6: task2: different generations of round-trip

No.	target	required	completed
1	changing customer number and locations	✓	✓
2	compare results between reusing and not reusing the solutions from the last environment	✓	✓
3	visualize the round-trip route	✓	✓
4	calculate total distance	✓	✓

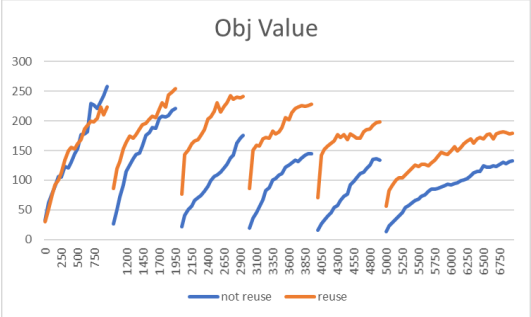


Figure 7: task2

We can see from Figure 7 that reusing the last population will extremely accelerate the procedure of evolution.

3.3 Task3: Large-scale Optimization Problem

No.	target	required	completed
1	clustering	✓	✓
2	visualize the round-trip route	✓	✓
3	calculate total distance	✓	✓

The easiest k-means cluster result must be two clusters on the left and right. We can see from Figure 8 that the salesman only travels to the other region once. Even though I didn't run too many generations, the trip path is relatively shorter than origin, which is randomly walking.

When we cluster custemers into four region, then we get Figure 9. The salesman obey the "once a rigion rule" and the path is apperantly shorter than 2 clusters. So suitable cluster number will help our program converge quickly.

3.4 Task4: Multi-objective Optimization Problem

As we can see from Figure 10 that these two methods are quite similar. This is the consequence of f_2 being a linear function. So the value of λ doesn't

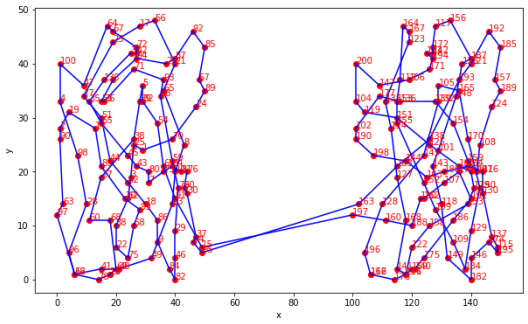


Figure 8: task3: 2 clusters

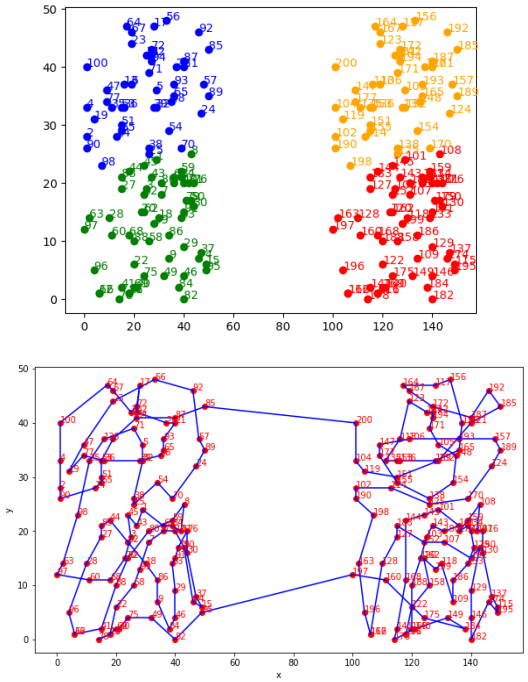
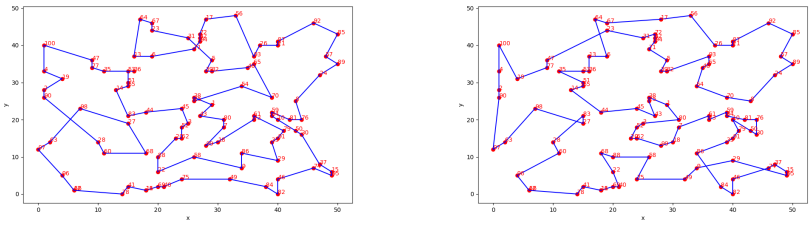


Figure 9: task3: 4 clusters

No.	target	required	completed
1	weighting objective functions-based method	✓	✓
2	Pareto dominance selection-based method	✓	✓
3	discuss the advantages and disadvantages	✓	✓

affect the result (because we have to visit all the customers so the total profit is fixed).

But when f_1 and f_2 are both nonlinear, λ indicates the importance of these two targets. We need to choose the parameter λ carefully, which is one of the weakness of weighting objective functions-based method. On the contrary, the biggest strength is that we do not need set hyper-parameters.



weighting objective functions-based Pareto dominance selection-based

Figure 10: task4

Because the TSP is quite a simple problem, so geatpy's algorithm converges quickly and accurately, and all the results are the same, which means that our pareto set shrinks into two points, shown in Figure 11.

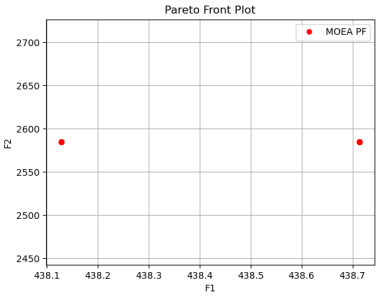


Figure 11: pareto

3.5 Task5: Time Window Constraint Problem

By adding another optimization target, our problem get much harder to optimize. So the pareto frontier (Figure 12) lies along the axis and the round-trip is different from the previous ones.

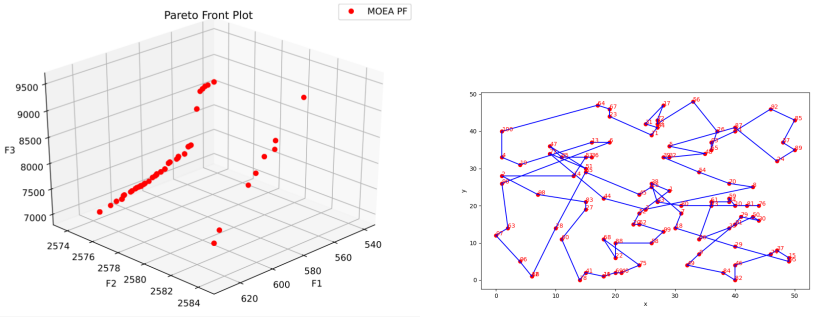


Figure 12: Time Window Constrain

4 Conclusion

In this project, I find it really hard to choose the right algorithms/operators to optimize our problem.

And I learn from geatpy that by preserving all the population to the next generation, then we can have a larger crossover and mutation probability, so that the program can converge faster.

References

[1] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[2] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial intelligence review*, 13(2):129–170, 1999.

[3] Jung Y Suh and Dirk Van Gucht. Incorporating heuristic information into genetic search. In *Genetic Algorithms and Their Applications*, pages 100–107. Psychology Press, 2013.

[4] Vijini Mallawaarachchi. Introduction to genetic algorithms â€" including example code, 2017.

[5] Wikipedia contributors. Travelling salesman problem — Wikipedia, the free encyclopedia, 2022. [Online; accessed 17-October-2022].