COMP5511-Project1-Report

XU,Jiahao

Department of Applied Mathematics
21042344G

1 Introduction

1.1 Problem description

We[1] are solving TSP(Traveling Salesman Problem) in this report. TSP is a classical NP-Hard problem. Here is how the TSP is being defined: assume a hard-working salesman is required to recommend goods to customers who are living in different cities. Any two cities are connected by highway. This salesman has to drive his car to find a path which goes through all the cities he needs to visit and make this path as shorter as possible.

In this report, we will discuss many different kinds of TSP problems, such as ATSP(Asymmetric Traveling Salesman Problem), TSPTW(Traveling Salesman Problem with Time Windows), TSPC(Traveling Salesman Problem with Clusters), and so forth.

1.2 Genetic Algorithm

This salesman is lucky because nowadays there are plenty of algorithms that can be used to help him solve this problem. GA(Genetic Algorithm) is one of the best. GA was introduced by John Holland in 1967. This smart algorithm applies Nature Selection to the optimization problem. A feasible solution(a path for TSP) has been known as a chromosome in GA, and any cities in the chromosome are called genes. GA mimics the selection, crossover, and mutation operations to find latent optimal solutions. In most cases, GA has the ability to find the global optima.

[1]In fact, all the contents in this report are completed by myself, but for the sake of expressing my thoughts more suitably, I use 'We' to denote 'I'.

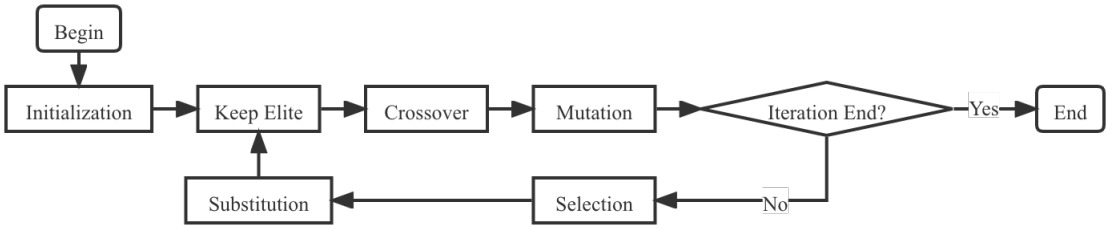


Figure 1: This flow chart is the main structure of GA. In general, after GA initializes the population, we need to keep the elite chromosomes in the initial population. When crossover and mutation finish, we will select chromosomes based on a specific strategy, and next, we use the elite chromosomes from the last generation to substitute the worse chromosomes in the current generation.

2 Methodology

2.1 Structure of GA

The basic structure of our algorithm is shown by **Figure 1**. We first generate the initial population, then begin the iterations. In each iteration, some chromosomes in the population will crossover with each other, or mutate by themselves. Then we apply the *keep elite* strategy, which can preserve the next generation never degraded. This strategy will also keep the Cost function non-increasing(Assume that we minimize the Cost function).

2.2 The Coding strategy

2.2.1 Coding of cities

To solve TSP by GA, the first we need to do is design the representation of chromosomes. The salesman needs to visit all the cities on the map, so we naturally use a specific integer number to represent a specific city. Assume the number of cities that salesman needs to travel to is n, so the chromosome is defined by lists which lengths are n, and all the elements in chromosomes are unique. This relation can be shown in the mathematical way as follow.

$$\forall chromosome \in P, chromosome = [\text{shuffle}(\text{range}(0, n))],$$

where P is the whole population and the shuffle() function means the integer list range$(0, n)$ is consisted of out of order. In our definition, every city has its own number, for example 0, 1 or 2. For simplicity, We denote the number of city$_i$ is i.

2.2.2 Cost function

We need a Cost function to determine whether our model performs well or not. Because we are solving TSP, it is good and acceptable to define the Cost function by formula (1). For

simplicity, we just use $ch[]$ to denote $chromosome[]$.

$$Cost = \sum_{\substack{i=0 \\ j=(i+1)\bmod n}}^{n} d_{ch[i]ch[j]}, \tag{1}$$

where i and j are the index of the neighboring two cities in *chromosome*, and the $d_{ch[i]ch[j]}$ is the distance between $\text{city}_{ch[i]}$ and $\text{city}_{ch[i]}$. Typically, formula(1) also called *fitness function* in GA. By formula(1), we can conclude the mathematical model of TSP by (2):

$$\begin{aligned} &Minimize \quad Cost = \sum_{\substack{i=0 \\ j=(i+1)\bmod n}}^{n} d_{ch[i]ch[j]} \\ &s.t. \qquad\quad \forall i \in [0, 1, \ldots, n-1] \text{ appears and only appears once.} \end{aligned} \tag{2}$$

2.2.3 Distance matrix

Here comes the problem: how to compute and store the d between any two cities in the map? The first problem is easy to solve. We have X-coordinate and Y-coordinate for each city, so the distance between city_i and city_j is defined by formula(3).

$$d_{ij} = ||\text{coordinate}_i - \text{coordinate}_j||_2, \tag{3}$$

where coordinate_i and coordinate_j denote the transpose of the coordinate of city_i and city_j. Then we consider using the adjacency matrix to store the distance. We are clear that the map of cities can be realized as a connected graph. So adjacency matrix helps us define the distance between any two cities. We already defined that every city has its own number from 0 to $n-1$, this implies the distance adjacency matrix $D \in \mathbb{R}^{n \times n}$. The element in i_{th} row and j_{th} column represent the distance between city_{i-1} and city_{j-1}. For symmetric TSP, we take $O(n)$ to construct D and D is symmetric, and $O(n^2)$ for ATSP.

2.3 Selection operator

From **section 2.2.2** and **section 2.2.3**, we are able to compute the fitness of every chromosome in the population, then we need to design an algorithm to select some chromosomes and put them into the population of the next generation. Note that we must ensure that the better chromosome has a greater probability to be chosen than the worse one. There are many different kinds of selection strategies, and in this project, we use RWS(Roulette Wheel Selection) which is also known as Fitness Weighting Selection.
After getting the fitness of chromosomes, the probability of ch_i being selected is defined by formula(4).

$$P_i = \frac{f_i}{F}, \tag{4}$$

where f_i is the fitness of ch_i and F is the total fitness in the population. By formula(4), we can easily calculate the probability of being selected for each chromosome. Obviously, the

sum of all probability is 1 and the bigger fitness means bigger probability. Now we introduce the CP(Cumulative Probability) which is the cumulative sum of all probability. This way just like we map the probability of each chromosome into the range $[0, 1]$, every city has its own sub-range, if we give a random number r from $(0, 1)$ and r belongs to one of the sub-range, so this means that city is being chosen.

RWS is an efficient and effective method. It ensures that the better chromosome has a bigger chance to be selected. It also keeps the probability for a worse chromosome because the worse chromosome may carry some useful genes fragments.

2.4 Crossover operator

Crossover is an important operation that exchanges two sub-chromosomes from two chromosomes to generate new chromosomes. From the definition of chromosomes in **section 2.2.1**, we know that after crossover operation, there may be some collision cities in the exchanged chromosomes. Therefore, the essential part of the crossover operation is to tackle the collision.

2.4.1 Obtain fragment

If we want to exchange two fragments of chromosomes, we need to get the fragment first. This is very easy that we generates two integer number *begin* and *end* from $[0, \ldots, n-1]$, we should exchange their values if *end* is bigger than *begin*, and we also need to ensure the difference between *end* and *begin* larger than 1 because if the difference between them is 1 or 0, that means crossover become mutation or the length of fragment is 0, respectively.
After getting *begin* and *end*, we can derive two fragments from two parents' chromosomes, and exchange them. Here is a simple example to illustrate how to get the fragments and exchange them.

We have chromosome1 and chromosome2 as the parents and assume *begin* is 2 and *end* is 5, after exchanging, we have new chromosomes: child1 and child2.

$$\begin{aligned} \text{chromosome1} = [0, 1, 2, 3, 4, 5, 6] &\implies \text{child1} = [0, 1, \{4, 3, 2, 1\}, 6] \\ \text{chromosome2} = [6, 5, 4, 3, 2, 1, 0] &\implies \text{child2} = [6, 5, \{2, 3, 4, 5\}, 0], \end{aligned}$$

where $\{4, 3, 2, 1\}$ in child1 is from chromosome2 and $\{2, 3, 4, 5\}$ in child2 is from chromosome1. We will put child1 and child2 into a temporary list in case they are selected to be parents.

2.4.2 Handle collision

We get child1 and child2, however, they are not valid chromosomes because some cities appear more than once. This is the reason why collision comes up with. To solve collision, we implement so-called *the same position substitution* strategy. We know that all the collision genes in chromosome1 must exist in the fragment from another chromosome, this means if a gene exists in fragment1(is a part of chromosome2), but does not exist in fragment2(is a part of chromosome1), so this gene collides.

For example, fragment1= $\{4, 3, 2, 1\}$ and fragment2= $\{2, 3, 4, 5\}$, gene 1 is exists in fragment1 but not in fragment2, so 1 is a collision gene in child1. Traverse fragment1 giving us all

collision genes, this means we have collision1 = {1} and collision2 = {5}. Then we define remain1 which is the remaining part of chromosome1 as we ignore the fragment2. **Algorithm 1** shows how we tackle the collision.

The main idea of **Algorithm 1** is we do not change the gene in fragment1, we just modify the collision genes in remain1. In the first beginning, we select the last gene in collision1 as collision_gene, whereby we locate the position of collision_gene in fragment1. Regard the gene which is at the same index in fragment2 as the temp. Then we modify the collision_gene in remain1 by temp and pop it from collision1. However, this operation may cause another new collision, so we have to check that whether temp is a collision gene. Repeat the loop until collision1 contains nothing. While the collisions are all eliminated, we reconstruct chromosome1 by fixing the remain1 and fragment1 together.

Algorithm 1 Tackle Collision Algorithm

Input: collision1;remain1;fragment1;fragment2.
Output: No output. It is an in-place algorithm.

```
1: while len(collision1) ≠ 0 do
2:     collision_gene ← collision1[-1]
3:     temp ← fragment2[fragment1.index(collision_gene)]
4:     remain1[remain1.index(collision_gene)] ← temp
5:     if remain1.count(collision_gene) = 0 then
6:         collision1.pop()
7:     end if
8:     if temp ∈ fragment2 & temp ∈ remain1 then
9:         collision1.append(temp)
10:    end if
11: end while
```

To illustrate this algorithm more vividly, we still take the simple example we used before. Now, we have child1=[0, 1, {4, 3, 2 ,1}, 6] and child2=[6, 5, {2, 3, 4, 5}, 0] and 1 is a collision in child1. So we find the gene by the same index in child2, which is 5, and use 5 to substitute the 1 beyond {4, 3, 2, 1}. After that, child1 becomes [0, 5, {4, 3, 2, 1}, 6]. Luckily, 5 is not a collision and this means the child1 is a valid chromosome now.

2.5 Mutation operator

The mutation is a very critical part of GA. Because of mutation, GA has the chance to find a better solution or even help GA jump out of local optima. Nevertheless, the rate of mutation should be limited because if the probability of mutation is very high, the high mutation rate will destroy our population and the GA degrades to random selection.

It is easy to design a mutation algorithm. We just randomly generate two integer number $first$ and *second* from $[0, \ldots, n-1]$, then we exchange the gene in chromosomes by the index $first$ and *second*.

2.6 Keep elite

A GA performed with crossover and mutation may generate worse children. Recall that we apply the RWS strategy in the selection, so the better chromosomes have a probability that not being selected. These two situations make the next generation may worse than their parents. This is the reason why we introduce the keep elite strategy.

Before GA performs crossover and mutation, we record some elite chromosomes. After crossover and mutation, we add the elite individuals to the population by substituting the worse chromosomes. **Algorithm 2** show that how we make substitution.

Algorithm 2 Elite Substitution Algorithm

Input: elite_chromosome, population, fitness_elite, fitness_pop.
Output: No output, it is an in-place algorithm.

```
while len(elite_fitness) ≠ 0 & min(fitness_pop) < max(fitness_elite) do
    index_min_pop ← fitness_pop.index(min(fitness_pop))
    index_max_elite ← fitness_elite.index(max(fitness_elite))
    population[index_min_pop] ← elite_chromosome[index_max_elite]
    fitness_pop[index_min_pop] ← M                ▷ M is a large enough number
    fitness_elite.remove(fitness_elite[index_max_elite])
end while
```

In **Algorithm 2**, elite_chromosome is a list of excellent chromosomes from the last generation, while fitness_elite is a list of fitness for each chromosome in elite_chromosome. As the same as fitness_elite, fitness_pop is a list of fitness for all chromosomes in the population. When **Algorithm 2** done, we have finished once evolution. Then, GA will back to select operation after recording some elite chromosomes in the population.

3 Experimental results and analysis

3.1 Basic results and analysis

Now we put our eyes into experiments. In this section, we will test our model by changing the parameters' values. Also, we will give a fundamental analysis.

In the most simple version of our model, we just solve TSP with 10 cities. Firstly, we introduce the hyper-parameters in our model. **Table 1** gives all parameters and their explanation.

Table 1: Hyper-parameters and corresponding explanation

hyper-parameters	explanation
population_init_size	the population size we fix
iteration_times	iteration times of GA
elite_keep_rate	the number of elite chromosomes model keep in each iteration
crossover_rate	the number of chromosomes will crossover
mutation_rate	the number of chromosomes will mutate

Base on these parameters, we can execute some tests repeatedly and figure some useful information out. In our test, we choose a group of hyper-parameters and run our model 100 times with it. Actually, we have the optimal cost of this simple TSP is 2.58. So we will count how many times our model will achieve the optimal situation in different groups of hyper-parameters. We will not change population_init_size because we are solving a simple TSP with only 10 cities, we fix this number with a proper value so that we can analyze the details in our model more efficient. We will also not vary iteration_time and elite_keep_rate because of the same reason.

Table 2 gives some results of our experiment. We fix population_init_size, iteration_times and elite_keep_rate with 150, 150 and 0.15, respectively. By changing crossover rates and mutation rates, we can figure out that our model performs better when the crossover and mutation operation happened more frequently.

Table 2: Several results about varying crossover and mutation rates

Groups[1]	Rate[2]	Groups	Rate	Groups	Rate
crossover_rate = 0.6 mutation_rate = 0.3	0.75	crossover_rate = 0.6 mutation_rate = 0.4	0.79	crossover_rate = 0.6 mutation_rate = 0.5	0.87
crossover_rate = 0.7 mutation_rate = 0.3	0.72	crossover_rate = 0.7 mutation_rate = 0.4	0.77	crossover_rate = 0.7 mutation_rate = 0.5	0.8
crossover_rate = 0.8 mutation_rate = 0.3	0.78	crossover_rate = 0.8 mutation_rate = 0.4	0.8	crossover_rate = 0.8 mutation_rate = 0.5	0.81

[1] Just two variables with three fixed parameters.
[2] Rate is the algorithm that achieves optimal times divided by 100.

By **Table 2**, we can easily find that the rate will be higher if we continue to enlarge the mutation rate, predictably. Recall that in **section 2.5**, we say that if the mutation rate is too high, the selection operation will degrade to random selection, nevertheless, it does not seem to happen in our model. The reason why our model keeps robust with a high mutation rate is we implement keep elite strategy in our algorithm. With the help of this significant strategy, the high mutation rate gives the model a greater probability to escape from local optima instead of destroying the population.

3.2 Dynamic adjustment of parameters

Inspired by the things we talk about above, we naturally come up with an idea, if we regulate the rate of crossover and mutation by the performance of GA, we are likely to have a better solution. If GA cannot break away from the local optimal, we try to expand the probability of crossover and mutation occurring. Otherwise, we reset crossover_rate and mutation_rate to the initial value. We design **Algorithm 3** which can ensure GA adjusts crossover rate and mutation rate dynamically.

Algorithm 3 Dynamic Adjustment Algorithm

Input: crossover_rate, mutation_rate, stuck_times.
Output: No special output, it is an in-place algorithm.

```
if GA stuck in local optima then stuck_times + 1
else
    stuck_times ← 0
    reset crossover_rate and mutation_rate to initial value
end if
if stuck_times = crossover_tolerance then
    if crossover_rate + step-size < 1 then crossover_rate + step-size
    end if
end if
if stuck_times = mutation_tolerance then
    if crossover_rate + step-size < 1 then crossover_rate + step-size
    end if
    if mutation_rate + step-size < 1 then mutation_rate + step-size
    end if
    stuck_times ← 0
end if
```

Algorithm 3 shows if GA falls into local optima, we will adjust the crossover rate higher, if GA is still stuck in local optima, we enlarge the mutation rate higher. Now, we apply **Algorithm 3** into our model, taking the three entries in the first row of the **Table 2**. We have **Table 3** which shows that by implementing **Algorithm 3**, the converge rate of GA improved.

Table 3: New results with Algorithm 3

Groups	Rate	Rate′[1]	Improve	Groups	Rate	Rate′[1]	Improve
crossover_rate = 0.6 mutation_rate = 0.3	0.75	0.9	20.00%	crossover_rate = 0.7 mutation_rate = 0.3	0.72	0.94	30.56%
crossover_rate = 0.6 mutation_rate = 0.4	0.79	0.96	21.52%	crossover_rate = 0.7 mutation_rate = 0.4	0.77	0.94	22.08%
crossover_rate = 0.6 mutation_rate = 0.5	0.87	0.94	8.05%	crossover_rate = 0.7 mutation_rate = 0.5	0.8	0.93	16.25%

Average improved: **19.74%**

[1] Rate′ is the new Rate after applying **Algorithm 3**.

In a short conclusion, we figure out that by fixing population size, iteration times, and elite keep rate, higher rates of crossover and mutation especially for the mutation will help GA get rid of local optimization. This is because if GA finds a feasible solution that is very close to optima, then crossover will lose its magic because it is hard to change several unsuitable genes in chromosomes to get the optima by exchanging a whole part of sub-chromosome from

another, however, the mutation can do that. So after we apply **Algorithm 3**, with ascending mutation rate, the probability of GA finding global optima is also ascending.

4 Extensions

We have 6 required extensions to solve. we will discuss everyone in this chapter. And a figure including all extensions' results will be shown.

4.1 More cities

This problem is so simple that we just need to generate more cities and still run the same model we talk about above. One way the generate cities is to use random values from 0 to 1, but we just use the cities' information in the offered data set. We only need to normalize the X-coordinates and Y-coordinates for every city. this simple extension can be solved by the original model.

4.2 ATSP

ATSP is also very simple, we do not need to modify our codes because we use an adjacency matrix to store the distance between every two cities. Assume that the salesman drive to a city will cost c, we just let the return route will cost $c \times 1.2$, by a more mathematical way to represent, in the ATSP' adjacency matrix, if the entry e_{ij} is c, then the entry e_{ji} is $1.2c$. So, up to now, we are already constructed an ATSP. ATSP can also be solved by the original model.

4.3 Constraints of start and end cities

Now, assume the salesman is required to depart from a specific city and return to this city in another specific city. This problem is also not difficult to solve because if we regulate those two cities, all we need to do is modify 3 parts of the codes of our model.

The first is the initialization of the population. In the normal version of our model, the chromosomes in the population are randomly generated, but now we fix the first and last gene in case our algorithm makes chaos. Then the left two parts are the operator of crossover and mutation, respectively. In both of them, we ignore the first and last genes so that we can fulfill this problem. The left part is the same as the normal version, we just run our model and if the salesman puts proper parameters into the model, the model will give an optimal path to him.

4.4 Sequential ordering

The poorly salesman meets a new problem again: this time, he is required to visit a special guest(Let us just use sg to denote this city) before some special cities(Let us just us sc to

denote these cities). We have two choices to help him, one is designing a new crossover and mutation algorithm to fulfill this strange requirement, and another one is keeping the operators we used now, and repairing chromosomes when this chromosome does not satisfy the constraint. Here we choose the second one by designing a new algorithm.

Algorithm 4 Sequential Repair Algorithm

Input: chromosome, sg, sc.
Output: repaired chromosome.

```
position ← a list.                          ▷ store the genes' position in chromosome
for item ∈ sc do
    position.append(chromosome.index(item))
end for
min_position ← min(position)
if chromosome.index(sg) < min_position then
    return chromosome
else
    swap(chromosome[chromosome.index(sg)], chromosome[min_position])
end if
return chromosome
```

Algorithm 4 shows how we repair the wrong gene in the chromosome. After crossover or mutation, the chromosome may not satisfy the requirement, so **Algorithm 4** will check whether this chromosome is wrong or not. If this is a wrong chromosome, the algorithm will locate the position of the first gene which appears in sc, and exchange it with sg. By doing that, we satisfy the requirement.

4.5 TSPTW

Here comes a more challenging problem. Assume that every city have their customs and each of them has its own work time. So we have three situations now, one is the salesman arrive a city in time, so all things are OK, however, if the salesman does not meet the time window, then he needs to call the staff back and let him get into the city or he must wait for opening. With the help of our TA(Mr.Wang Zhenzhong) and [1], we modify our Cost function by adding a penalty to cost when the salesman misses the time window.

Algorithm 5 TW Penalty Algorithm

Input: chromosome, TW, penalty_rate. ▷ TW is a list of tuple stores time window
Output: penalty.

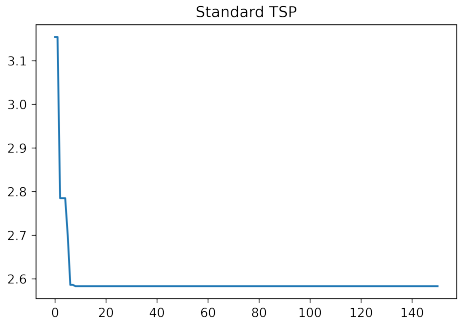
```
penalty ← 0 , current ← 0                                          ▷ current time
for each gene ∈ chromosome do
    gene_tw ← TW[gene]
    if current <= gene_tw[1] then
        if current <= gene_tw[0] then                               ▷ arrive early
            current = gene_tw[0]                                    ▷ wait for customs
        end if
    else                                                            ▷ arrive late
        penalty ← penalty + (gene_tw[1] - current) × penalty_rate
    end if
end for
return penalty
```

We design **Algorithm 5** to compute the penalty of one specific chromosome. In the above algorithm, we can see that the salesman will wait for a longer time if he arrives in a city later. After we get the penalty, we add it to the true cost. In the simulation stage, we use *TSPTW_dataset*.txt as our data set. Other parts of our original model do not need to be modified except the codes of how we handle the cities' location because we need to make a normalization with the raw data.

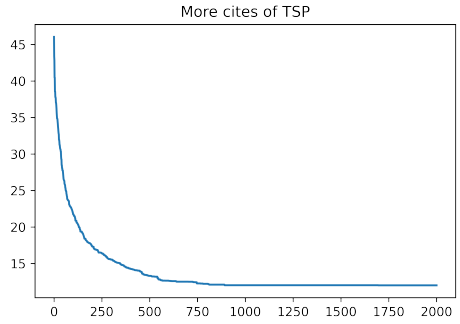
4.6 TSPC

This poor guy finally meets the last requirement which is to let him visit several cities first, then go to other clusters of cities. OK, this is a more challenging problem. The first thing we need to do is divide cities into several city groups. We use *Cluster_dataset*.txt as our data set, and following the instruction, we decide to divide these cities into 3 clusters. The method we use is KMeans which is a very classical machine learning clustering algorithm, but we do not introduce it in this report.

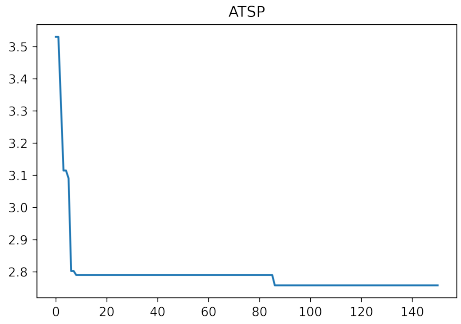
Now we have 3 clusters divided by KMeans, the next question is how should we design our model to complete the TSPC. Inspired by [2], we first find the shortest path in every two clusters and fix the cities in every sub-chromosome. Because we have 3 clusters, so when the chromosome crossover or mutation, we split the chromosome into 3 pieces, and these 3 sub-chromosomes will be regarded as the normal chromosomes and doing crossover or mutation. Recall that we have 3 clusters, so we need to fix 3 pieces' the first and the last gene ensure the distance between every two clusters is shortest, then we apply the same crossover operator and mutation operator we have mentioned in **section 4.3** to solve these sub-problems. In the last, we assemble pieces into the normal chromosome.



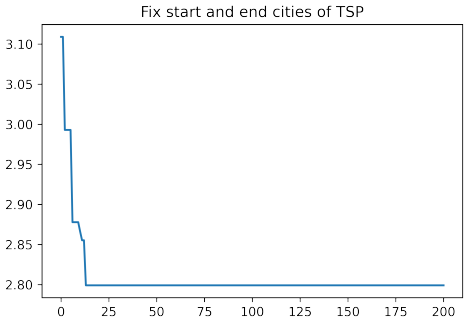
(a) Standard TSP, converged at 2.58



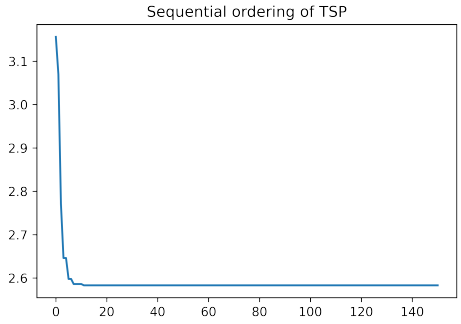
(b) More(100) cities, converged at 12.02



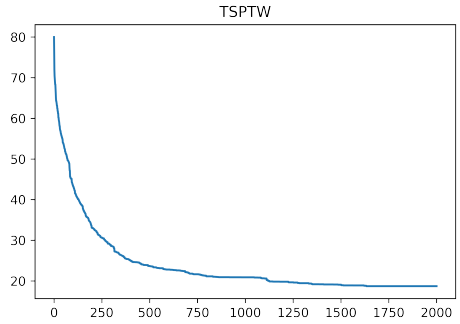
(c) ATSP, converged at 2.76



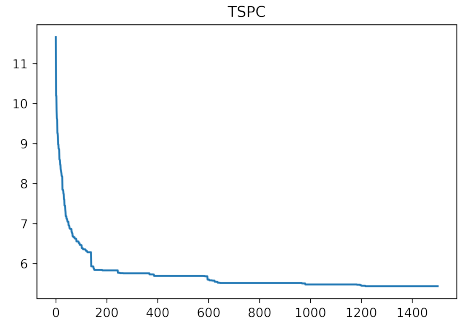
(d) Constraints of TSP, converged at 2.80



(e) Sequential ordering TSP, converged at 2.58



(f) TSPTW, converged at 18.72



(g) TSPC, converged at 5.43

Figure 2: Convergence of all extensions

4.7 A short conclusion

Up to now, we have already introduced how all the extensions are be done, After simulation, it is feasible for our model to accomplish the target though some of them converge slowly. **Figure 2** shows all the convergence curves of all the TSP problems, we can see that all the curves are strictly descending. All models attain the minimum value of Cost function.

5 Conclusion

5.1 Results

In the first part of this report, we implement the GA and consider some extensions of basic TSP, we first design the representation of chromosomes, then solve the collision problem in crossover. Enlightened by the experimental results and the keep elite strategy, we design the dynamic adjustment algorithm to modify crossover_rate and mutation_rate dynamically. Proved by results, after using this algorithm, the performance of our model is enhanced.

Then, we begin to settle down the extensions. More cities in TSP, ATSP, Constraints of the first and the last cities, and sequential ordering problems are easy to solve. we just need to modify the codes of our original model a little. TSPTW requires us to design a penalty algorithm to evaluate how late or early the salesman arrives at customs. TSPC asks us to cluster data set first, then we find the shortest path between every two clusters, then we split chromosomes into three parts and fix the first and the last genes in every sub-chromosomes, by doing these, we change the complex problem into three easy problems though it may cause some bias because we fix the position of some genes.

5.2 Shortcomings

Obviously, our project has some problems. In this sub-chapter, we will discuss some inevitable(or, maybe, can be solved) problems.

5.2.1 Shorts of RWS

In **Section 2.3**, we introduce RWS as our selection strategy, however, RWS performs really well? Recall that RWS will give the chromosomes with higher fitness a higher probability to be chosen, this is the right thought of our goal because we always want the next generation can be more excellent. However, the problem is also here, when a so-called super chromosome appears, this chromosome will dominate RWS, which means, GA will get into a local optimal value.

How to deal with this? We have some preliminary thoughts and just for thoughts because we do not implement and verify them. The first is after a chromosome be chosen, we can modify the fitness of this chromosome to reduce its impact, nevertheless, it can be foreseen that this will cost so much time to recompute the new fitness of all chromosomes in the population. The second way is we can set an upper limit for the number of chromosomes. For example, we set

every chromosome in population can be chosen at most 10 times, if a super chromosome was already attained this boundary, then we generate another number until the super chromosome not be chosen.

5.2.2 Lack of codes optimization

I[2] do not think my codes perform well in time consumption because there is much redundancy on them, however, because I also have other 3 subjects and most of them have their midterms and assignments so that I have no enough time to optimize my codes before the deadline. I am very sorry for that.

6 Reference

[1] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

[2] Wang Xing. Research on traveling salesman problemof cluster class featured large city groups. *Process Automation Instrumentation*, 34(3):4, 2013.

[2]To avoid ambiguity, I use 'I' to denote myself here.