Solving Travelling Salesman Problems with Genetic Algorithms

Junshuai Zhang
Aberystwyth Univeristy, UK
juz1@aber.ac.uk

***Abstract*—In this report, we talk about the application of genetic algorithms (GAs) in classic travelling salesman problems (TSPs). In the beginning, we introduce travelling salesman problems and genetic algorithms. Then we discuss how to apply genetic algorithms in sequencing problems, introducing different options of genetic operators. The methods in this report have been implemented as a framework so that we can run a couple of experiments with different configuration. We did some experiments with the framework, managing to understand what affects the performance of GA and how it does. The program under the framework can solve the problem for ATT48 dataset which is an example of 48 cities in one minute. Also, the solutions for a TSP with 200 random cities have been shown later. After that we explain how the framework was implemented. In the end, we discuss some other methods of GAs applied in TSPs and the future work we can do for this project.**

I. TRAVELLING SALESMAN PROBLEM

The travelling salesman problem (TSP) describes that, a travelling salesman is going to travel a number of cities and sell goods by the way. He only wants to visit each city exactly once and return his home lastly. Then what is the route with the shortest distance? TSP can be considered as an abstract of many complicated problems in real life, especially in the field of transportation. For example, what is the shortest path for a school bus to travel from the school, pick up every students and return to the school? Or how shippers deliver goods to retailers with a minimum of time and distance?

In graph theory. It can be also described as: Given a complete undirected weighted graph, find a Hamiltonian cycle with the least weight. As the Hamiltonian cycle has been proved NP-hard, TSP is an NP-hard problem. TSP has $n!$ solution space. In general, it is nearly impossible to find the optimal solution with simple search algorithm in limited time. Consequently, many people usually use a stochastic algorithm to find out a near-optimal solution.

In this project, we will only discuss 2D Euclidean TSPs. This means all cities are in a two-dimensional plane, and the distances between two cities are their Euclidean distances. It simplifies the problem. But a solution for 2D TSPs is still very meaningful to solve some real-life problems.

II. GENETIC ALGORITHM

Genetic Algorithm (GA) is an evolutionary algorithm. It is usually used to find out near-optimal solutions of optimisation problems. It was inspired by biological evolution. To address problems, A GA generates a large number of plans randomly, called population. Then it simulates the process of natural evolution to perform the operations such as natural selection, chromosome crossover and chromosome mutation on the population, making population evolve (i.e. get better answers) continuously. After a number of iterations, pick up the best answer as the solution.

In common, a GA consists of the following steps:

1) Initialise the population of individuals for the problem.
2) Select better individuals evaluated by a fitness function as candidates.
3) Use the candidates to generate the next generation with crossover and mutation operator.
4) Until the result is good enough, go back to step 2.

Most studies focus on generic operators of GA, which are the methods to make the population "evolve". They includes: selection, crossover and mutation.

A. Chromosome and encoding

A chromosome stands for a solution in the population. It is also called an individual in GA. It consisted of a series of genes. A common encoding of chromosomes is using the same form as the solution of problems.

B. Selection and Fitness

A GA simulates the natural selection to choose some "good" individuals in the population for crossover or mutation so that the population will evolve and iterate to get the best answer. When evolving, selection makes the decision according to the fitness of a chromosome.

Fitness is a value to evaluate how well the result of a individual performed in the problem. It is often the value which the problem tries to optimise. The formula to calculate the fitness is called a fitness function.

C. Crossover

Crossover operation merges two or more selected individuals into some new individuals to make up the next generation. It is the most basic approach in GA to make a population evolve.

D. Mutation

Compared with crossover, mutation is usually an approach to keep GAs away from local optimal solution. It does import some new genes in chromosomes so that GA can still be stochastic rather than converging.

III. Methods

In this project, we chose to attempt several methods to solve TSPs and do experiments. In this section, we introduce all methods which we apply into solving problems.

A. Encoding

As we mentioned before, the encoding of gene in chromosome has commonly the same form of solutions. For example, if we try to solve a TSP with 10 cities, a possible chromosome is as form of:

$$1 \quad 2 \quad 0 \quad 9 \quad 8 \quad 7 \quad 4 \quad 5 \quad 3 \quad 6,$$

which means we will visit the city 1, 2, 0, 9, 8, 7, 4, 5, 3, 6 in order.

B. Fitness

The fitness of chromosome can be the travelling distance of route. That is easy to understand and evaluate. However, in GA, we tend to use a low fitness to stand for an terrible answer of the problem literally. Also, the value of fitness is commonly positive. Hence, we will use the reciprocal of a distance as shown as the following fitness function:

$$fitness(x) = 1/total\_distance(x).$$

C. Selection

There are a number of selection methods for GA. In this project, three selection methods have been used and experimented. They are: roulette-wheel selection, tournament selection and elitist selection.

1) Roulette-wheel Selection: Roulette-wheel selection makes the probabilities of choosing each individuals exactly equal to their fitness. So the probability to choose individual i will be:

$$P(x) = fitness(x) / \sum_{i=1}^{n} fitness(i)$$

2) Tournament Selection: Tournament selection will choose a certain number of individuals completely randomly and select the fittest individuals.

3) Elitist Selection: Elitist selection is different than the former selection methods, it does selection not for genetic operators. It just keep some best individuals into next generation each iteration. This can improve the efficiency of GA markedly. Because it will absolutely prevent some good candidates from eliminating in evolving. However, a suitable parameter of retention rate (i.e. a elitist rate) is important. A overlarge parameter may cause GA lost randomness, be easy to sink into local optimums.

D. Crossover Operator

A usual crossover operator just exchange two parts of chromosomes in the same position. As shown following:

$$\begin{array}{ll}
P1: & A\ B\ C\ D \mid E\ F\ G \mid H\ I\ J\ K \\
P2: & K\ A\ G\ F \mid B\ D\ H \mid I\ J\ C\ K \\
 & \\
C1: & A\ B\ C\ D \mid B\ D\ H \mid H\ I\ J\ K \\
C2: & K\ A\ G\ F \mid E\ F\ G \mid I\ J\ C\ K
\end{array}$$

However, this does not work to product a valid chromosome. Because, as we know, it is not allowed to have visiting twice to the same city. Therefore, we need to do some special crossover operators for TSPs.

TSP is a sequencing problem, which is how we call the problem with this situation. There are some well-known crossover methods working for sequencing problems [1]. In this project, we utilise order crossover, cycle crossover and partially mapped crossover.

1) Order Crossover: Order Crossover(OX) tries to keep the same order and relative positions of genes in parents as in children. It chooses a fragment of genes randomly, copying the genes from a parent into a child at the same position. Then, starting just after the fragment, copy each gene which did not appear inside the fragment from another parent into the child until all positions has been filled in.

$$\begin{array}{ll}
P1: & A\ B\ C\ D \mid E\ F\ G \mid H\ I\ J\ K \\
P2: & E\ A\ G\ F \mid B\ D\ H \mid I\ J\ C\ K \\
 & \\
C1: & A\ B\ D\ H \mid E\ F\ G \mid I\ J\ C\ K \\
C2: & C\ E\ F\ G \mid B\ D\ H \mid I\ J\ K\ A
\end{array}$$

2) Cycle Crossover: Cycle Crossover(CX) always keeps the positions of genes from one of parents. CX finds out a cycle from parents by the following method: choose the first gene in a parent, observe the gene in the same position of another parent, then jump to the same gene in the former parent; repeat this process until the first gene of the former parent is found in the latter parent. After finding a cycle, just exchange each genes of the cycle between two parents, then we got two children. A question of this method is, if the parents are the same loop in a graph just with different start points. Only one child can be produced and it is exactly as same as the former parent.

$$\begin{array}{lcccccccccccc}
P1: & H & K & C & E & F & D & B & L & A & I & G & J \\
 & | & & & & & & & | & | & | & & | \\
P2: & A & B & C & D & E & F & G & H & I & J & K & L
\end{array}$$

$$H - A - I - J - L - H$$

$$\begin{array}{ll}
C1: & H\ B\ C\ D\ E\ F\ G\ L\ A\ I\ K\ J \\
C2: & A\ K\ C\ E\ F\ D\ B\ H\ I\ J\ G\ L
\end{array}$$

3) Partially Mapped Crossover: Like OX, Partially Mapped Crossover(PMX) tends to keep the orders and relative positions of genes in parents as in the children. It is like a combination of OX and CX. PMX will choose two cutpoints, then copy the alleles within two cutpoints from parents into children. In order to sort out the genes-conflict problem, it does some ”map” operators, which are similar to the cycle finding of CX: for each gene which is not appear in ”cutpoints area” of another parent, keep tracking the gene in the same position of the another parent until the gene appear out of cutpoints; fill the gene into the position found. After all genes have been filled in, copy remaining alleles directly from parents into children.

```
P1 :  I E H | D G C F B | J A
P2 :  A B C | D E F G H | I J

C1 :  _ _ _ | D G C F B | _ _

          E − G − F − C

C1 :  _ _ E | D G C F B | _ _

          H − B

C1 :  _ H E | D G C F B | _ _
C1 :  A H E | D G C F B | I J
```

E. Mutation Operators

Three mutation methods are used and analysed in this project. They are inversion mutation, exchange mutation and shift mutation.

1) Inversion Mutation: Inversion mutation select a range in chromosome, then invert genes inside the range.

```
A B C | D E F G | H I J
A B C | G F E D | H I J
```

2) Exchange Mutation: Exchange mutation is a simple operator which just exchanges the position of two genes in the chromosome.

```
A B C D E F G H I J
    |       |
A B G D E F C H I J
```

3) Shift Mutation: Shift mutation is easy to understand. It just generates a random value p, then cycle shift the positions of genes in chromosomes by p positions.

```
A B C D E F G H I J

J A B G D E F C H I
```

IV. Experiments

There are lots of parameters and choices of generic operators in a generic algorithm. We have to choose the value of population size, elitist rate and mutation rate. And also, the choices of methods for selection, crossover, mutation is important for solving problems without doubt. But due to the time limit of the project and the length limit of the report, we cannot do a quite specific search for determining the best parameters like a grid search. Hence, we decided to use some usual reasonable values for the parameters which do not have much negative effect to the results. Also, we did some experiments to see which values of some parameters will bring the best results for us.

A. Parameters Choice and Experimental Setup

1) Population Size: We use **1000** as the population size, because in experiments, we found that a overlarger population size cannot provide much better answers at all. And it is acceptable that a population with 1000 chromosomes can usually get the results in 1 minute for 1000 iterations.

2) Elitist Rate: We applied elitist selection in almost all of experiments. We got the application of elitist selection can improve the rate of convergence greatly. However, we only define the value of the elitist rate as **0.001**. It means that we only keep the fittest chromosome in the population. Because under this setup, the elitist selection still works well. And a larger elitist rate usually leads to a rapid convergence of the local minimum.

3) Selection: In addition to **elitist selection**, we also applied **roulette-wheel selection** in GA. Because it is fair enough and much more frequently applied than tournament selection in most researches. But it is still worth attempting in future. So it has been implemented in the project, just not being used.

4) Mutation: We decided to use **inversion mutation** rather than exchange mutation and shift mutation. The reason is: exchange mutation whose range of influence is only two genes seems not make much sense when the scale of the TSP is large (i.e. the TSP with a large number of cities); shift mutation might not be very useful because the start point of this problem does not matter at all. By contrast, inversion mutation can bring a scaled effect and different operators than crossover. So it is be chosen.

B. The Impact of the Mutation Rate

Generally, a mutation operator tries to keep randomness of a GA so that the GA is less possible to sink in the local optimal value. So it is considered that a low mutation rate will let the convergence be quick but get into a bad local optimal value early. A high mutation rate is more possible to find out a better result but it makes convergence slow. However, in experiments, I found the impact of the mutation rate depends on many aspects like different mutation operators, different selection operators and which problem we solve.

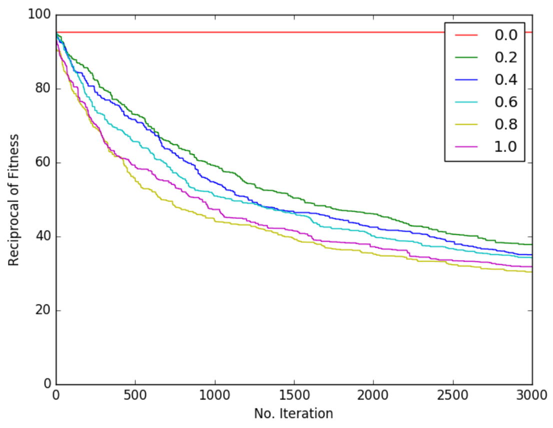


Fig. 1. The Impact of Different Mutation Rate of 0.0, 0.2, 0.4, 0.6, 0.8 and 1.0 with the setup of 200 cities, a population size of 1000, 3000 iterations, elitist selection with 0.001 rate, roulette-wheel selection, partially mapped crossover and inversion mutation.

As shown in Fig. 1, the interesting thing is, sometimes a extra high mutation rate does really perform better than other lower mutation rates do. My guess of this is, in some situations, the efficiency of mutation operators is better than crossover operators of two chromosomes. After that, I also found some people have the same view in some research [2].

In the end, I chose to be prudent to use **0.2** as the mutation rate. And I think it is worthy to have research in the future to see the influence of the mutation in GA for travelling salesman problems.

C. Comparison of Different Crossover

For applying GA to solve TSP, the most important part must be the crossover operator. It is not only because we have to choose a suitable crossover operator for TSP (sequence problems), but also various crossover operators have obviously different performances in solving TSP.

We did experiments for every crossover methods three times each. The result has shown in Fig. 2.

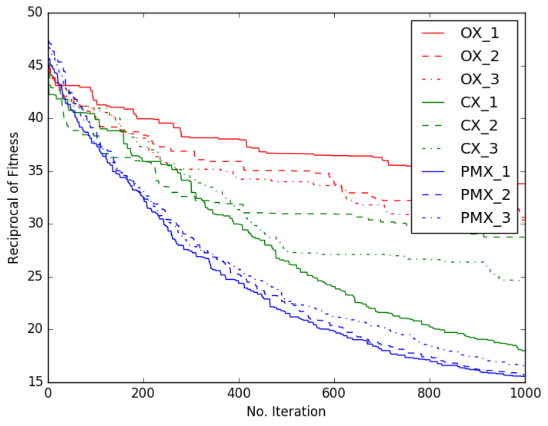


Fig. 2. Comparison between different crossover operators OX, CX and PMX with the setup of 200 cities, a population size of 1000, 3000 iterations, elitist selection with 0.001 rate, mutation rate of 0.2, roulette-wheel selection and inversion mutation.

As shown in Fig. 2, PMX is stable and better than OX and CX. So we choose **PMX** in our final program.

V. Results

We have already got a decision on which parameters and methods are applied in GA. In this section, we use the program to solve TSPs and observe the results.

A. A Real Problem

We use a dataset named ATT48 from TSPLIB [3] for testing the performance of our algorithms. The dataset ATT48 describes a simple abstract problem of travelling US state capitals. The reason to choose it is that, it is a real problem so that the result may be more meaningful than random problems. Also, the optimal solution of ATT48 is known, so that we can know how well the algorithm perform. Besides, it is commonly used, so that it can be a benchmark of this project and the methods that other people used.

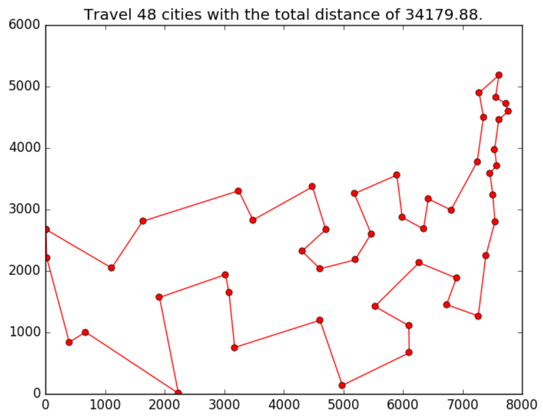


Fig. 3. The result of 5000 iterations with 1000 chromosomes where the mutation rate is 0.2, the elitist rate is 0.001, roulette-wheel selection, partially mapped crossover and inversion mutation.

With a poor performing 1.3GHz Intel Core i5 CPU, it took 44.2s to run 5000 iterations and got a route with the length of 33867.40 shown in Fig. 3. I think it is very good because there is only 1% difference between the result and the optimal solution, whose length of route is 33523.71, as shown in Fig. 4.

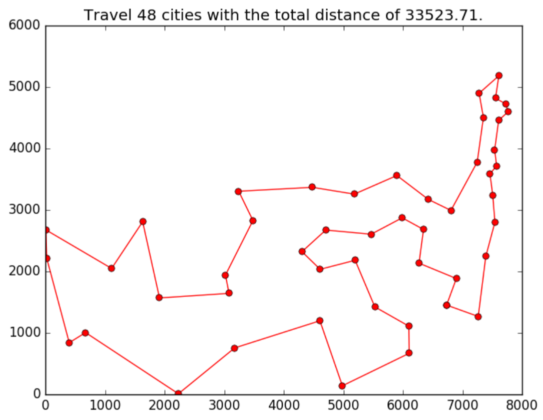


Fig. 4. The Optimal Solution of ATT48 dataset in TSPLIB[3]

B. Random 200 Cities

In order to know how well the program can perform, we also use a random TSP to test the program. This time we can not know what the best solution is, but we will draw the route to try to understand that.

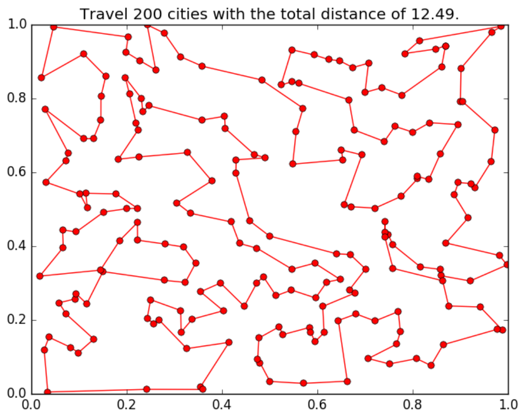


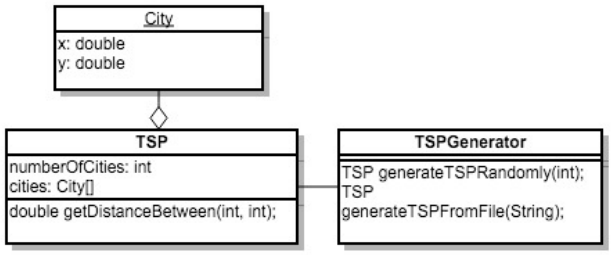
Fig. 5. A result of solving a TSP with 200 cities. The population with 1000 individuals has evolved 20000 times by roulette-wheel selection, partially mapped crossover and inversion mutation with the mutation rate of 0.2 and the elitist rate of 0.001.

The result has been shown in Fig. 5. It took 673.1s (20000 iterations) to get this result. I think it is not bad because we can see there are no joint in the route.

VI. Implementation

To do the experiments easily, I developed a GA for TSP framework in Java 8. In this framework, it is easy to use different kinds of selection, crossover and mutation methods. Besides, it is easy to add a new method and do the experiments. In this section, we talk about how classes in this framework have been designed. And through this, we learn of the implementation of the framework.

A. TSP Class and TSPGenerator Class

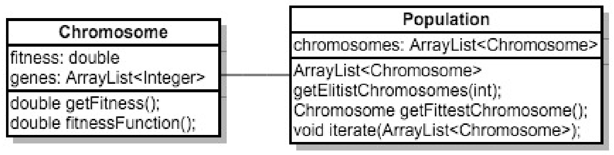


City
x: double
y: double
TSP
numberOfCities: int
cities: City[]
double getDistanceBetween(int, int);
TSPGenerator
TSP generateTSPRandomly(int);
TSP
generateTSPFromFile(String);

Fig. 6. Class Diagrams of City, TSP and TSPGenerator

The `TSP` class is the Travelling Salesman Problem entity class, which stores the information of cities and provides the calculating method of the distance between two cities. To create a `TSP`, instead of new a instance directly, we use `TSPGenerator` class which provides two static methods so that people can create a `TSP` instance randomly or by reading from files.

B. Chromosome and Population



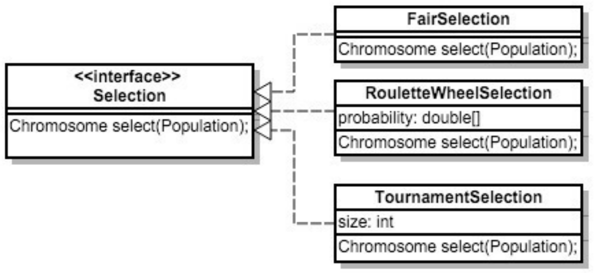
Chromosome
fitness: double
genes: ArrayList<Integer>
double getFitness();
double fitnessFunction();
Population
chromosomes: ArrayList<Chromosome>
ArrayList<Chromosome>
getElitistChromosomes(int);
Chromosome getFittestChromosome();
void iterate(ArrayList<Chromosome>);

Fig. 7. Class Diagrams of Chromosome and Population

`Chromosome` class defines how to encode and how to compute the fitness. `Population` class is a set of `Chromosome` and can be used to find the elitist chromosomes and fittest chromosome.

To use a different fitness function, we can just extend the `Chromosome` class and reload the `fitnessFunction()` method(). In fact, the `fitnessFunction()` is a protected function so that people will not call it directly. And the user should always call the `getFitness()` function to get the fitness. In other words, we a template design pattern to avoid the fitness being recalculated each time the function is called.

C. Selection, Crossover and Mutation Interface



<<interface>>
Selection
Chromosome select(Population);
FairSelection
Chromosome select(Population);
RouletteWheelSelection
probability: double[]
Chromosome select(Population);
TournamentSelection
size: int
Chromosome select(Population);

Fig. 8. Class Diagrams of Selection

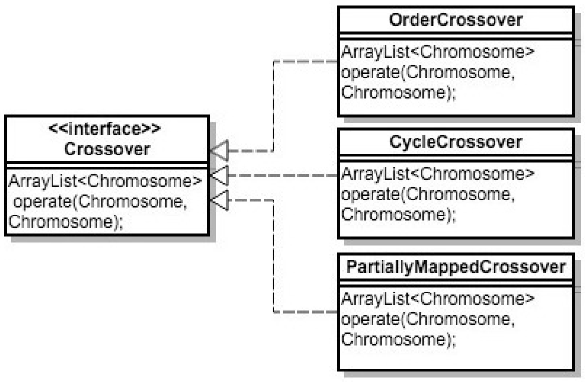
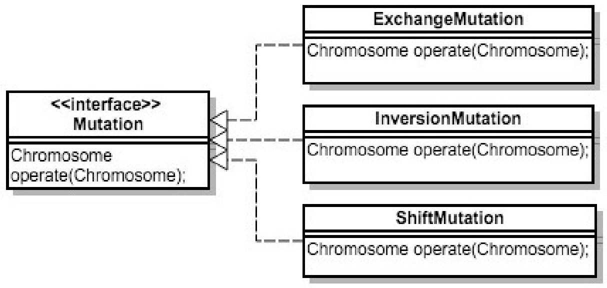


Fig. 9. Class Diagrams of Crossover



<<interface>>
Mutation
Chromosome operate(Chromosome);
ExchangeMutation
Chromosome operate(Chromosome);
InversionMutation
Chromosome operate(Chromosome);
ShiftMutation
Chromosome operate(Chromosome);

Fig. 10. Class Diagrams of Mutation

These three generic operators are interfaces in the framework. For defining a kind of generic operators, people have to implement the method.

In the framework, we have implemented all generic operators mentioned before, a list of all implemented classes as shown in Fig. 8, Fig. 9 and Fig. 10.

D. Parameters Class

`Parameters` class contains three parameters in GA: population size, mutation rate and elitist rate.

As we mentioned before, the population size decides how many individuals in the population of GA. The mutation rate will affect the probability of mutation and crossover. If mutation rate is 0.2, then there will be 20% for mutation occurring, and 80% for crossover occurring. The elitist rate is about elitist selection. It decides how many individual with best performance will be kept into the next generation.

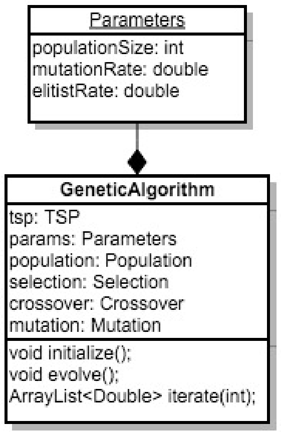


Fig. 11. Class Diagrams of Parameters and GeneticAlgorithm

E. GenericAlgorithm Class

`GenericAlgorithm` class is the implementation of generic algorithms, it will initialise the population and configure the algorithm parameters (`Parameters` class) and genetic operators. It provides a method called `evolve()` which makes population evolve exactly once.

F. Usage

There are two ways to use this framework. The first way is, because the test driver programs have been written and be made as jars, we can just use jvm to run the code directly with the following command:

```
# solve the ATT48 TSP
java -jar jar/TSPSolver.jar data/ATT48.txt

# solve a random TSP of N_CITIES cities with
# N_ITERS iterations
java -jar jar/TSPSolver.jar N_CITIES [N_ITERS]

# do the mutation rate experiment
java -jar jar/MutationRateExperiment.jar

# do the crossover experiment
java -jar jar/CrossoverExperiment.jar
```

What is more, after executing the above programs, users will be noticed to run some python programs to visualise the results. The commonds are like following:

```
python scripts/draw_route.py res/last.res
python scripts/draw_mutationrate.py
python scripts/draw_crossover.py
```

Note that this requires a python `matplotlib` library. Whichever python2 and python3 can both work well.

The second way to use the framework is the way to make the code more ”framework”, not just to type some commands. If we want to adjust some parameters in GAs, even add some new genetic operators without modification of existing code. We can set the parameters and implement the genetic operator interfaces by ourselves. Then we write some general code like the following:

```
TSP tsp = TSPGenerator.generateTSPRandomly(50);

Parameters params = new Parameters();
params.mutationRate = 0.2;
params.populationSize = 1000;
params.elitistRate = 0.001;

GeneticAlgorithm ga = new GeneticAlgorithm(
    tsp,
    new RouletteWheelSelection(),
    new PartiallyMappedCrossover(),
    new InversionMutation()
);

ga.iterate(2000);
```

Note that `iterate(numIter)` always restarts and initialises the population. We can call `ga.evolve()` to iterate the population exactly once and do anything we want between iterations.

VII. Conclusion

In summary, GA is a good solution for TSP if we just need to find a good enough solution rather than exact best answer. It performs much better than just brute force search. And it is very easy to be implemented.

However, there are many options in GA to solve TSP. I believe they are still lots of possibilities in the future. More crossover and mutation operators are worthy to attempt in the future. For example, a LR crossover, which I did not try though, is common for sequence problems too. And there are some modern methods people have figured out which can deal with TSPs with thousands of cities. Before it comes to the end, I would like to introduce some other methods in the application of GA for TSP.

A. Methods Based on Priority Algorithm

When I just started in this project, my first idea is to use a priority algorithm. It is a idea to use floating numbers in encoding of chromosome standing for priorities of cities. Then to visit the city with higher priority earlier than cities with lower priorities.

The above mentioned fitness function and selection methods can be also applied in this form of chromosomes. And crossover and mutation is easy: we can calculate the weighted average of parents chromosomes to crossover a new chromosome; each gene has a possibility to occur a mutation, getting a new value between [0, 1).

There are several improvements I have done. For example, there is a problem of crossover, using weighted average makes the value field decreasing continually. For solving this problem, I extend the value fields each time after crossover. Besides, I set a fixed starting point to reduce redundant solutions.

Although I spent much time attempting various crossover and mutation, even doing some improvements on chromosomes every time after genetic operations. It is still awful. (I got an idea to consider the geographical locations of cities because this is a specific 2D TSP. And I figured out a interesting ”gravitation” algorithm with the formula of universal gravitation. However, I did not make it work in limited time.)

Finally, I gave up this idea and removed it from the framework. (This algorithm needs a floating type encoding genes, so I used generic types and reflection of java to implement it. Later on, after I had given up, I realised it bring more complexity of the framework, so I abandoned it.)

B. Better Solutions

Another population modern methods is to use a greedy strategy to take the geographical locations of cities into account in crossover. It selects more than two parents for crossover each time, choose a gene in the chromosome, calculate the distance between the cities of this gene and its neighbouring gene in each parent chromosome and always join the nearest gene behind it.

This method is so popular and powerful that I have seen lots of paper using it with different programming language. And even in javascript, it can runs so well to solve TSPs with thousands of cities.

However, I kind of doubt the performance of greedy strategies on some special case, whose cities are not uniformly randomly distributed on a 2D plane. But I have to admit, it is a fast and not bad way to find a good enough solution.

References

[1] Fox, B. R., and M. B. McMahon. ”Genetic operators for sequencing problems.” *Foundations of genetic algorithms 1* (1990): 284-009.

[2] Abdoun, Otman, Jaafar Abouchabaka, and Chakir Tajani. ”Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem.” *arXiv preprint arXiv:1203.3099* (2012).

[3] Reinelt, Gerhard. ”TSPLIBA traveling salesman problem library.” *ORSA journal on computing* 3.4 (1991): 376-384.