**README**

To develop the genetic algorithm for solving the Travelling Salesman Problem, we implemented 3 Classes: Genetic Algorithm class, Person class and Configuration class. Person class mainly focus on defining the DNA and how the DNA was translated to real point of locations and the Crossover/ Mutate process. The Genetic Algorithm class mainly contains the whole list of salesmen, maintains a map of all locations and focuses on the evolution process. It has the method to evaluate the fitness, select the population and determine the way to reproduce the next generation. Configuration class stores the parameters for running the program.

**Version 1 of our implement**

Considering the Genetic Algorithm Class, it contains five major methods: Constructor, fitness, sort, select and evolve. It also contains some other supplement methods like logging and selectMin/Max methods.

* The Constructor initialize the process of TSP, it builds a map of all points and initialize the population using the pre-defined list and parameters including the DNASize defined in Person class and populationSize.
* The Fitness method calculate the total distance of each person and save it into the person, the number of the fitness is the total distance.
* Sort using the fitness of each person and make the population in order. The shortest distance represents the best fitness.
* Select method select the top 60% fitness and if the population grows too large (which in this problem, we arbitrary set it to 4 times of the original population size), it will cut down half the number of the population
* Evolve is the main method of the genetic algorithm, it combines the above methods and reproduce the next generation.

In the Person class, we are using three different constructors two differ the type to initialize a person. The class contains translate methods, crossover method and mutate method.

* We are using 5 digits of binary number to represent 32 different genetic type, each type represents a location point. All the DNA is composed by different genetic type without duplicate.
* The constructor without parameter is used to generate the first generation, it randomly decides the DNA. The constructor with two parameters is used for crossover, it uses the DNA from father and mother to combine and mutate the child’s DNA. The constructor with only one parameter is used for asexual reproduction, it receives its father’s DNA and then mutate to generate its own DNA.
* The translate method help translate the DNA to the location point
* Crossover method receive the DNA from father and mother, combine their DNA order and generate the DNA of the child. First, (start from 0) setting the even number of DNA from the father DNA’s corresponding location. Then, the odd number of child DNA will be set according to the sequence of the non-duplicate DNA from mother’s. This process ensures the child inheritance the tendency of parents’ DNA and in the same time have the possibility to try new path.
* Mutate method happens in a certain possibility predefined, which will exchange one DNA point with another DNA point.

**Version 2 of our implementation**

* Store the weight instead of actual path for each vertex.
* Example: (N = 4); Index: 0, 1, 2, 3; Weight: 3, 1, 2, 0; Weight 0 is at index 3, so the path begins from 3, then 1, 2, and 0.
* In the initialization stage, generate M salesman and there path randomly.
* Mutation: If random value < Mutation Rate, perform mutation before crossover. Mutation swaps the weights of 2 different vertex.
* Crossover: Mother x Father. Children's weight is average of (Mother.weight[i] + Father.weight[i]) /2. Mother has slight advantage compared to Father so if average weight is same, follow Mother's priority: The actual formula used is: (Mother.weight[i] \* 1.01 + Father.weight[i] \* 1.0001) /2. Crossover is performed for the entire population (size M) so the children size is M \* M; Also include the M parents in case parents is better currently; So, the result matrix size is (M + 1) \* M.
* Fitness: Intuitively, sum of the distance in the path.
* Culling: From (M + 1) \* M population, drop worse M \* M and leave better M.

**Conclusion**

Our problem is to find a shortest path to access all the points without duplicate by using genetic algorithm. We have developed two methods for solving this problem. The result of two methods are relatively close and both methods runs a reasonable result. According to our tests, the genetic algorithm will generate a result which is quite close to the optimal result. In a relative small size of problem (16 locations), the program generates the result which is close to the optimal result (or maybe is the optimal result) in less than 50 generations (rare improvement after that) but in a larger size of problem (20 locations), the program will need more generations to reach a result which is close to the optimal. According to the log, we can see the result can gradually approach the optimal result but can hardly be the actual optimal result. After several generations, the best DNA remained unchanged and the crossover between other DNA type influenced little to the result which is a huge problem remaining unsolved.

The genetic algorithm in some cases can help generate a relative optimal result for those with a high time complexity. But as we are hard to know whether the result is optimal or not, it would require extra effort to prove that. We consider the genetic algorithm will easily sink into the situation that its result is only local optimal solution not overall optimal, the population will easily be precocity and hard to generate new genetic, it will require a lot of space and calculation to approach a optimal result. Its result is also not stable, usually we will need to run the program several times to know whether the result is close to the optimal.

In conclusion, we believe the genetic algorithm is a wonderful way to get a result close to the optimal solution. With other constrains and optimization, this algorithm will help deal with various kinds of problems.



