Using a Genetic Algorithm in TSP

Presented by: Marco Elumba, Nº 20210982

Course: Computational Intelligence for Optimization 2021/2022

GitHub link: https://github.com/marcoelumba/GeneticAlgorithmTSP

Submission: 31st May 2022

Table of Contents

Introduction

This project is an adaption of a Genetic Algorithm (GA) for a computer optimization project. There are many optimizations problems that can be solved using GA. In this paper, I will discuss the classic problem of a Traveling Sales Person (TSP) using a database of 152 cities.

The TSP problem asks the following question: “Given a list of cities and the distance between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” There are many other similar implementations and use cases that a TSP optimization can be applied to. Various operations in GA can be used in solving TSP problems but not all GA operations would apply in TSP optimisation. I have used only the following operations:

- in selection - fitness proportion, tournament and ranking;
- in crossover - partially matched crossover, single point crossover and two-points crossover;
- in mutation - swap mutation, inversion mutation, and insert mutation.

I will discuss the distance model I have used to evaluate the fitness of the route. The report will explain some selection, crossover, and mutation models that I have added from the original Charles model given in the class. Additionally, I will also highlight the challenges that I faced when implementing the new and existing functions. Finally, I will present the experiments that I have done using a combination of functions.

This GA TSP problem was inspired by the mathematical programming TSP test data published at http://elib.zin.de. Therefore, the code can only calculate and accepts datasets of data that can calculate Euclidean distance between points. As mentioned, there are many other implementations of TSP, with different sets and types of data, for example, a TSP with “cost of travel” on top of “distance between points” but this type of problem is not in the scope of this project.

Implementation

Given the initial set of codes in Charles with the basic problem of TSP, I have managed to reuse some of its library. The initial challenge was reading the published TSP file(s) online with x and y coordinates. To solve this, I created a function that reads and loads the locations into my program as a directory class with name, x and y coordinates as attributes. Additionally, I parsed the original location list as a directory every time I calculated the distance between points.

Once I had the parsed list of locations, the next task was to be able to calculate the distance between the locations (cities) given a specific order or what I called a route.

The project code can be found in the following GitHub link:

LINK: https://github.com/marcoelumba/GeneticAlgorithmTSP

Fitness functions

I used Euclidean distance commonly known as the distance formula. The formula calculates the change between the two points in a coordinate and finds the square root of the sum of each changes and adds it to the change of two points in another coordinate and squares each change.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Figure 1: Distance formula (Euclidean Distance)

I implemented two other formula distances in the code - Minkowski distance and Manhattan distance.

$$\boldsymbol{distance} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{n-1}} |(\boldsymbol{x}[\boldsymbol{i}] - \boldsymbol{y}[\boldsymbol{i}])|$$

Figure 3: Manhattan distance

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

Figure 2: Minkowski distance

With the distance function I was able to generate routes with fitness by calculating the total sum of each point distance in a route, which allowed me to move on to my selection process. Since generally, the TSP problem is a minimization problem, I added the minimization instance in my fitness proportion selection (fps) operation and then added ranking as a third selection operation.

After completing the 3 selection operations, I started working on the crossover operations. I added the two-point crossover operation to the library. For the mutation operations, I added insert mutation into the library to complete 3 mutations.

Finally, I implemented a *write_result* function that calculates the average fitness of each generation across each run. The results are then processed by a function called *evaluate* that generates a line-chart to compare the performance of each combination of algorithm.

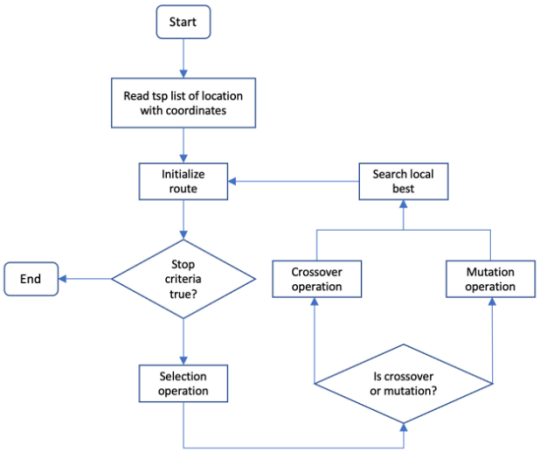

Figure 4: Algorithm workflow diagram

Fitness Proportion Selection

I recoded the fps method to calculate the percent share of each route's fitness as my *weight*. In minimization, to ensure the smallest distance will have the highest percentage of the share, I subtracted the distance from the maximum distance and divided by the total distance.

Ranking

The other selection method that I added is the ranking method, which ranks the order of the fitness and calculates the probability of the rank by dividing the total of the rank. An example representation is shown in figure 2. Ranking selection is mainly advised if the fitness of the routes are closely similar to each other.

Rank	Fitness	Sel.prob.
1	3.00	4.76%
2	7.00	9.52%
3	9.00	14.29%
4	10.00	19.05%
5	15.00	23.81%
6	85.00	28.57%

Figure 5: Example ranking method

Two-points Crossover

Th two-points crossover is a multi-point crossover with two crossovers. It is equivalent to two single-point crossovers but with different crossover points. It is done by picking two random crossover points from the parent. This strategy can be generalised to k-point crossover for any positive integer k, picking k crossover points [https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)].

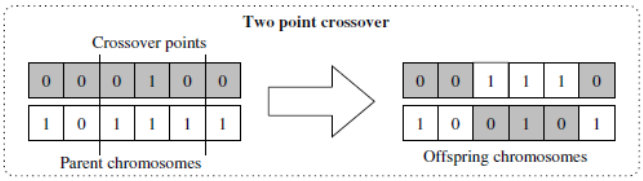

Figure 6: Two-point crossover

Insert Mutation

Insert mutation, is a selection of a random point from the parent to insert in another randomly selected point different from the original point. A representation of insertion mutation can be seen in fig.4 and the other mutation algorithms.

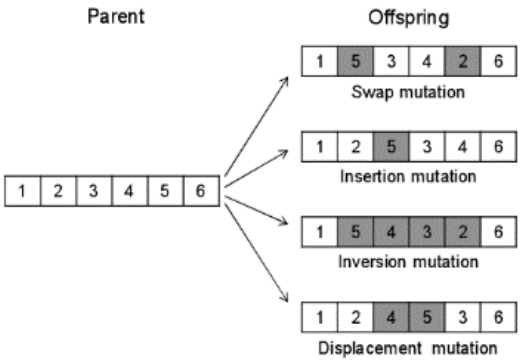

Figure 7: Mutation

Experimental result

To decide which combination of algorithms I wanted to use in my experiment, I decided to run a combination of all 3 selections, 3 crossovers and 3 mutations with 50 runs and 100 generations as an initial benchmark. In total I had 27 algorithms run 50 times with 100 generations each using Euclidean distance and Manhattan distance. Based on this initial run I found the figure 8 result for Euclidean distance. The top 5 algorithms for both distance formulas were the same, namely: 1.) tournament selection, inversion mutation and single point crossover, 2.) fitness proportion selection, inversion mutation and single point crossover, 3.) tournament selection, insertion mutation and two-point crossover, 4.) fitness proportion selection, insertion mutation and two points crossover, and lastly 5.) tournament selection, insertion mutation and single point crossover.

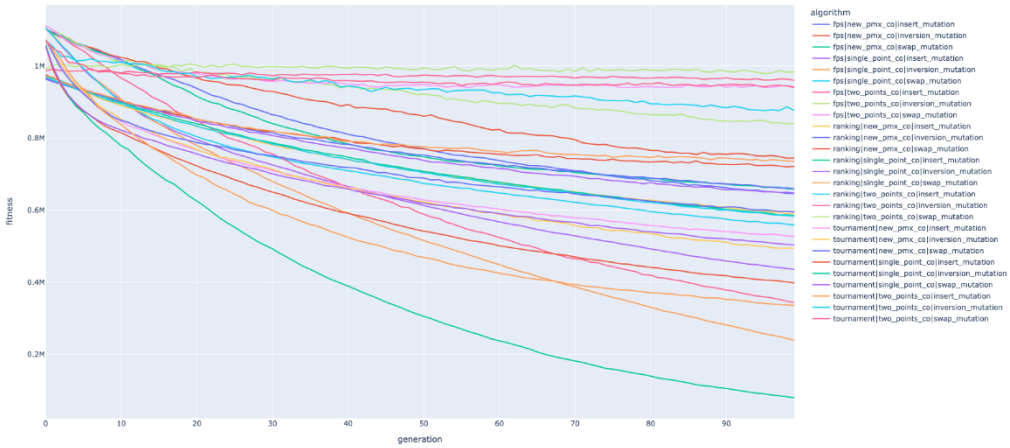

Figure 8: Initial run (50 runs by 100 generations) with 0.9 crossover probablity and 0.15 mutation probability using Euclidean distance formula to calculate fitness

After finding the above result, I decided to further run the top 5 combination algorithms with a higher number of generations and with other datasets. The result of the final test showed that of the 5 selected algorithms, the combination of tournament selection, single point crossover and inversion mutation performed better than the rest. At a 100th generation almost all combinations have reached the average minimum travel per generation or the fittest routes per 1000 generation across 50 runs.

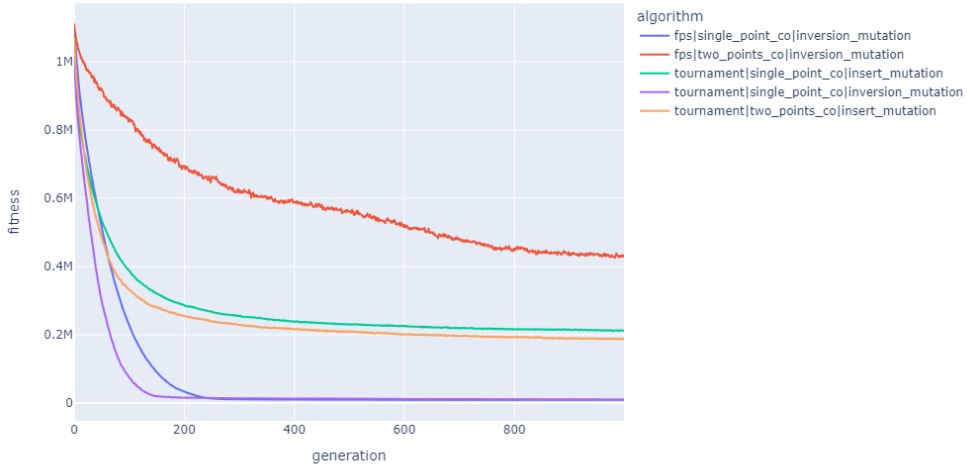

Figure 9: Top 5 algorithm combo run (50 runs by 1000 generations) with 0.9 crossover probablity and 0.15 mutation probability using Euclidean distance formula to calculate fitness

Conclusions

In this report we can see how genetic algorithms work in solving a problem such as TSP or a best shortest route. In the first test of 100 runs the top performing combination of algorithms has shown that it reached the average fitness solution to the problem between about 100 and 200 generation. When executed with 1000 generations there was only a marginal increase in fitness.

There is a huge combination of routes possible with 152 cities. This genetic algorithm approach for TSP served well in finding a solution with a limited number of generations and runs. Using this method allowed me to arrive quickly at an answer to the shortest possible route. The table below shows the best algorithm combination produced an Average Minimum Fitness of 31K and took only 7 minutes to complete 1000 generations.

Selection	Crossover	Mutation	Generation	Average Minimum Fitness	Time Completed (1000 generation)
tournament selection	single point crossover	inversion mutation	130	31K	7mins
fitness proportion selection	single point crossover	inversion mutation	130	134K	9mins
tournament selection	two-point crossover	insertion mutation	130	293K	14mins
fitness proportion selection	two points crossover	inversion mutation	130	764K	1hr+
tournament selection	single point crossover	insertion mutation	130	342K	16mins

This analysis and result are probabilistic - in other problem data sets it is possible that other combinations of algorithms could perform better than tournament selection, single point crossover and inversion mutation.

References

1. https://en.wikipedia.org/wiki/Euclidean_distance
2. https://machinelearningmastery.com/distance-measures-for-machine-learning/
3. https://pianalytix.com/k-nearest-neighbour/
4. https://medium.com/@samiran.bera/crossover-operator-the-heart-of-genetic-algorithm-6c0fdcb405c0
5. https://www.researchgate.net/figure/Mutation-operators-applied-to-chromosomes-in-the-proposed-genetic-algorithm_fig8_272093243
6. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html