Adam Carcione 12/10/18

The Effectiveness of Genetic Algorithms on the Travelling Salesman Problem

Abstract

This paper details the effectiveness of the application of genetic algorithms on the travelling salesman problem. I will cover the problem at hand regarding the travelling salesman problem, my approach to solving it, the experimental design, the data, and the results.

Introduction

In this paper I will approach the problem of finding the optimal solution to various travelling salesman problems or TSPs through the use of genetic algorithms. Genetic Algorithms are very good when attempting to search through large problem spaces and due to this nature are ideal for the task of trying to find the correct path out of any n! amount of paths, where n is the number of cities in the TSP. This paper will cover the results of my analysis on the effectiveness of a genetic algorithm and will document the various design choices that I made to maximize its performance. It will include all of the data that I accumulated over running my algorithm with various mutation methods on 10,17,26, and 48 - TSPs which I then used to determine which mutation method would give the best result.

TSP Encoding

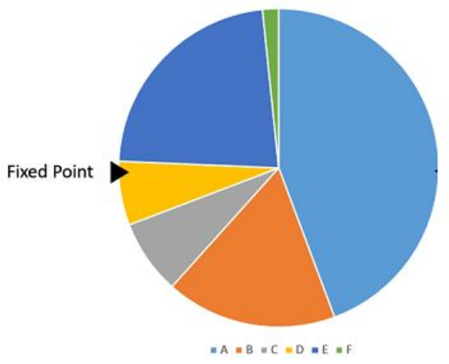
The travelling salesman problem is a problem that involves the question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" The approach that I took for encoding this problem would be to represent each city in my problem as a number, and then represent the path that I take with an array of numbers that correspond to their unique cities. For example, a possible individual in a population doing 10-TSP might look like this: [0,3,9,6,4,5,2,8,1,7]. Notice that city 0 corresponds to the first city because 0 is the index of the city in my list of cities.

Fitness

In order to measure the fitness of a certain individual (i.e. a solution) I found the sum of the distances from following the complete path. However this presents us with a problem because our most fit individual is going to have the lowest fitness score. In my **selection** section I will describe this problem in detail and what approach I took to solve it.

Selection

My original approach to selecting the parents for the next generation was to use fitness proportional selection, which works by assigning a probability of being selected as a parent proportional to an individual's fitness score in comparison to the sum of the populations fitness score. The problem mentioned previously is that the individuals we want to select have the smallest fitness and would therefore have the smallest probability. This problem was overcome by simply assigning the fitness of each individual the inverse of its original fitness value. For example, an individual with a fitness score of 1,580 after summing its distances would have its fitness score assigned to 1/1,580 or 0.00063291139. I then sum up all of these inverse fitness' to find the total fitness and from there on out this method works just like normal fitness proportional selection. The individuals with the smallest distance and smallest original fitness will now have the highest inverse fitness score and will have the highest probability of being selected as a parent. Once I have all of the inverse fitness score in a sorted array, I create an accumulated normalized list where each value is equal to the sum of the values that come before it. The last value in this accumulated normalized array should always equal 1, unless you have done something wrong. The values in this array will act as the "slices" of a pie and each value contains the range of numbers between itself and the value before it. Finally, I generate two random numbers and select the individuals whose range of values the random numbers fall into. The diagram below is a good presentation of how this selection method is supposed to work.

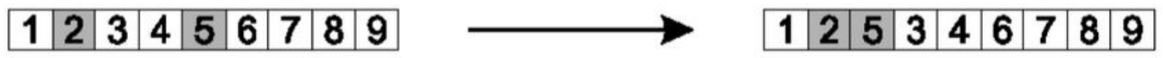


Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Crossover
It was imperative for our crossover function to make it impossible for an individual to have duplicate numbers contained within its encoding, as this would imply that the same city will be visited multiple times, and other cities will be excluded from the path outright. Because of this, I decided to use position based crossover to ensure that each value in an individual was unique. It works by selecting two of the parents that were picked by the inverse proportional selection, and choosing k points for crossover. It then copies the values at those k points from the first parents into the same indexes in the first child array. The next step is to fill in all of the missing indexes in the child from the values in the second parent in the order that they appear, skipping any values that were copied over initially from the first parent. The same is done to generate the second child, except the k values copied will come from the second parent and the missing values will be filled in by the first one.

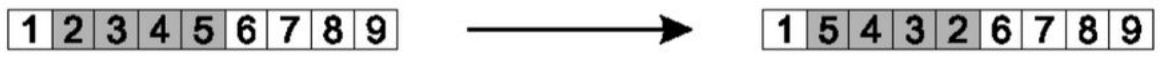
Mutation
Insert Mutation
One of the mutation methods that I tested my genetic algorithm on was insert mutation. This mutation method is performed by picking two unique and random indexes in the individual, and

inserting the value at the second index directly after the value at the first index, which causes everything after these values to be shifted over by one. This mutation method preserves most of the order and the adjacency information within the individual and thus is not a very drastic mutation.



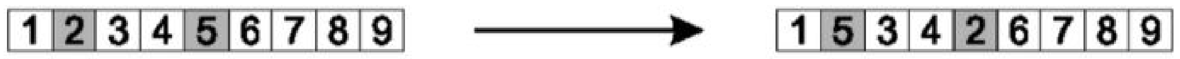
Inverse Mutation

Another mutation method that I used for my genetic algorithm was inverse mutation. This method is performed by picking two indexes within the individual and reversing the substring in between them. This mutation is quite different from insert mutation because it preserves most of the adjacency information but is very disruptive to the order of information.



Swap Mutation

I also experimented with swap mutation for my genetic algorithm. This mutation method is performed by picking two indexes within an individual and swapping the values at those locations. This type of mutation preserves most adjacency information and is not too disruptive to the order of the information.



Elitism

Elitism is the act of copying a predetermined percentage of the fittest individuals from each generation into the next generation without the possibility of being subjected to mutation. Elitism played a very important role in my project because before I had implemented elitism I was finding that the average fitness of my population was never improving over time. Once I had implemented elitism I immediately noticed a very drastic improvement to the average, minimum, and maximum fitness of the population from generation to generation.

Results

The results from my experiments were quite satisfactory as I was able to produce optimal routes to various TSPs. I limited my experimentation to only the mutation methods because the obvious choice for crossover was position based crossover and because inverse fitness proportional selection offers good diversity as well as the prospect that over time the best individuals will be picked more often than not. The three mutation methods that I gathered data on were swap mutation, insert mutation, and inverse mutation. The data gathered from the smaller TSPs were quite easy for my genetic algorithm to solve and the differences between the various mutations method were amplified as the number of cities increased.

From *Table I* we can see that for 10 city TSP all of the mutations work essentially equally well. Invert and insert mutation were able to find the optimal route of 1127 100% of the time and swap mutation was able to find it 90% of the time, finding a path of 1128 once instead.

From *Table II* we can see that for 17 city TSP inverse mutation is starting to become noticeably more effective, finding the optimal route of 2085 50% of the time, whereas insert and swap only find it 40% of the time. However, the difference in their average fitness' is not insignificant as invert mutation's average fitness was 2087.3. This value is within 2 miles of the optimal distance, whereas both swap and insert mutation have distances of over 2100.

From *Table III* we begin to see the differences between each mutation method more clearly. None of the methods were able to find the optimal route of 937 miles 100% of the time, in fact inverse and swap mutation were only able to find it 20% of the time and insert mutation never found it in any of the 10 tests. While all of the convergence generations are quite similar among all three methods, the average fitness of them is quite drastic. Inverse mutation had an average fitness of 956.8 miles, insert mutation had an average fitness of 975.3, and swap mutation had an average fitness of 988.8 miles. At this point is starting to become clear that inverse is the most effective mutation method of the three.

From *Table IV* none of the methods were able to find the optimal distance. Inverse mutation clearly comes out on top of these tests as it had an average fitness of 35,266.3 miles which is a 363% improvement from the average fitness of the first generation. Swap mutation performed the second best with an average fitness of 39,687.6 miles and a fitness improvement of 329% over the 300 generations. Insert mutation, following the trend of being comparable with swap mutation performed relatively equally with an average fitness of 40,799.7 and an average improvement of 338%.

<u>Conclusion</u>

The results of my experimentation not only validate the effectiveness of genetic algorithms on the travelling salesman problem, but also make it clear that inverse mutation is the best choice for the mutation method in my algorithm. In every single test that was run, inverse mutation had either the highest or was tied for the highest average fitness. The differences between inverse mutation and the other two methods were not as apparent on the smaller TSPs, however on the 26 and 48 city TSPs the differences in performance were quite clear. The usefulness of genetic algorithms proved themselves capable in the implementation of the travelling salesman problem, but also work very well in any problem that involves searching large problem spaces. For example, another area of genetic algorithms that I am familiar with is its use for breaking substitution ciphers. In the future, things that could be done to expand on this study would be to implement various selection methods such as tournament selection, and compare its effectiveness to that of inverse fitness proportional selection. In conclusion, I would claim that my experimentation on the use of genetic algorithms on the TSP proved to work quite well, and it is my hope that in the future people apply this approach to solving other similar problems that could have an impact on society.

Data:
Avg. Fit is the average fitness score of the best fit individuals over ten tests
Avg. Conv. is the average generation that the algorithm converged on over ten tests
Avg. Impr. is the average improvement of fitness values from generation 1 to the generation it converged

Parameters Used for Testing:
Population Size = 300
Number of Gen. = 500
Mutation Rate = 0.4
Crossover Rate = 0.9
Percent Elitism = 0.3

These parameters were chosen after testing various different values for each parameter on all of the datasets and deciding which values worked the best.

Table I (Opt. Dist. = 1127)

10 TSP	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Avg. Fit	Avg. Conv.	Avg. Impr.
Inv. Mut	1127	1127	1127	1127	1127	1127	1127	1127	1127	1127	1127	15.7	169%
Ins. Mut	1127	1127	1127	1127	1127	1127	1127	1127	1127	1127	1127	15.1	172%
Swap Mut.	1127	1127	1127	1127	1127	1127	1127	1127	1127	1128	1127.1	17.2	168%

Table II (Opt. Dist. = 2085)

17 TSP	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Avg. Fit	Avg. Conv	Avg. Impr.
Inv. Mut	2085	2085	2090	2085	2085	2090	2090	2085	2088	2090	2087.3	73.2	189%
Ins. Mut	2085	2090	2123	2085	2146	2085	2118	2090	2085	2152	2105.9	59.9	185%
Swap Mut.	2158	2085	2090	2085	2153	2116	2090	2090	2085	2085	2103.7	65.7	181%

Table III (Opt. Dist. = 937)

26 TSP	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Avg. Fit	Avg. Conv.	Avg. Impr.
Inv. Mut	953	937	962	940	972	956	953	955	1003	937	956.8	131.8	236%
Ins. Mut	1003	975	993	994	974	1013	989	978	974	995	988.8	133.8	237%
Swap Mut.	978	937	959	970	1001	955	1035	978	1003	937	975.3	144.3	228%

Table IV (Opt. Dist. = 10,628)

48 TSP	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Avg. Fit	Avg. Conv.	Avg. Impr.
Inv. Mut	34737	35626	35205	34544	35270	35471	35371	36934	35248	34257	35266.3	340.7	363%
Ins. Mut	35107	36218	41204	39237	40679	47543	43761	42398	35826	46024	40799.7	330.9	338%
Swap Mut.	39523	37282	36882	36018	40046	43114	42455	48760	34772	38024	39687.6	366.9	329%

(10 TSP Dataset made by me manually)
(17, 26, & 48 TSP Datasets from http://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html)