TSP solved with Genetic Algorithm

Eduardo Saldana

November 11 2021

Introduction

The TSP or Travelling Salesman Problem presents the situation of given a number of cities and their coordinates to find the shortest path possible in a way that every city is visited only once and then return to the original city. This problem was solved using genetic algorithms which rather than just brute forcing to every possible solution since the number of possible routes grow exponentially. A map with 55 cities will have 55! possible routes.

To understand Genetic Algorithms we have to establish a few concepts:

- Chromosomes are possible solutions to a problem, in this case a possible route
- A Generation simply refers to a group of chromosomes
- Crossover is any algorithm two Chromosomes can go through to create a new Chromosome
- The 2 Chromosomes used to create the 2 new ones are referred to as the parents and the resulting Chromosomes are called the off-springs
- Mutation is process a Chromosome goes through which changes random aspects of a Chromosome in random way
- A Chromosome is said to be "Valid" if the solution is feasible, in this case meaning the Chromosome includes every city just once except for the starting and ending city which are the same

Background

In order to even begin the to go through the Genetic Algorithm itself we need an initial population or size X which means we create code that will give us X random chromosomes. Then for a set number of generations we will go through a process to make the general population better than the last.

This is done by selecting the best chromosomes of the generation at the start of the process, we call this selected chromosomes "Elites". Elites are separated from the rest of the population. After than we select a pair of chromosomes and determine by random chance if they should go through crossover, if they are selected for crossover we obtain 2 new chromosomes which will take their place in the new generation. If they are not selected for crossover then the 2 chromosomes simply advance to the next generation. After that each new chromosome (Excluding elites) has a chance of "mutating" which means going through random changes. At the end of this process the elites are attached back to the generation and that concludes the loop, this is repeated a defined number of times.

Here is some pseudo-code to better understand:

```
For number of generations:
    Evaluate all chromosomes and select elite, then separate them
    Select pairs of chromosomes then decide if they will undergo crossover
    Apply crossover or simply pass them to the next generation
    Select chromosomes to apply mutation and apply it
    Insert elites back in population
```

Experimental Setup

To run these tests specific parameters were used. The parameters that were changed to see how it affected the outcome were crossover rate and mutation rate, this were set to either 0.9 (90%) or 1.0 (100%) for crossover rate and 0.0 (0%), 0.1 (10%) or 0.2 (20%) for mutation rate. The parameters that remain constant through the 22 and 55 cities were the generation size and number of generations, 200 generation size and 400 generations for the 22 cities and 60 generation size and 400 generations for the 55 cities.

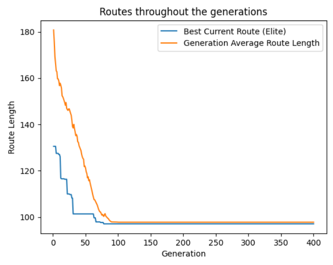
In order to replicate the experiments or try different parameters simply run the program and when asked about a specific parameter input the number. To change between running the 22 cities or 55 cities program simply change the file to be opened near the top of the program from "22locations.txt" to "55locations.txt" and vice versa. Additionally to change between UOX crossover with bit-mask and the 2-point one refer to the Readme.txt file or code comments to see how to change between the two.

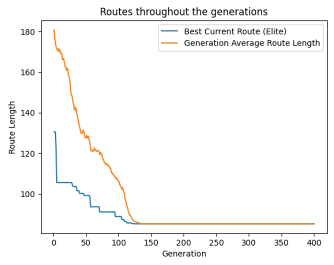
The crossover operators used were two who were used independently, the first one being a 2-point crossover. The 2-point crossover works in a rather simple manner, it selects a starting an ending index excluding the beginning and the end and then takes out those sections of each parent and swaps it with one another, after that it checks for repeated numbers in each child as well as for repeated numbers. Then it replaces the repeated numbers with the ones that are yet to exist based on the order of the other parent.

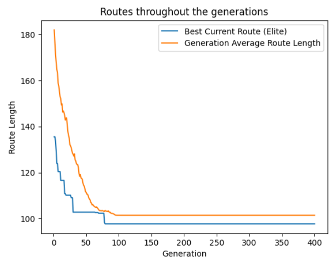
The second crossover operator used was a UOX crossover with bit-mask. This crossover works by creating a randomized list of numbers of 0's and 1's in a list which is the same size as a chromosome minus 2 since the starting and ending cities wont participate in this to avoid invalid chromosomes. After generating the bit-mask each child receives the same number as their parents in each location where the bit-mask is 1 and receives the other parent number when the number is 0. After that the ending process as the 2-point operator is used, the code checks for repeated numbers and for missing numbers and then replaces a copy of the repeated numbers with a missing one in the same order as the second parent.

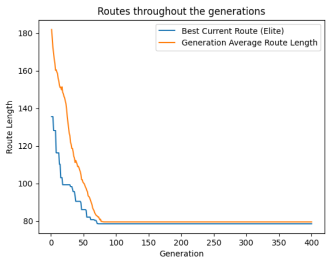
Results

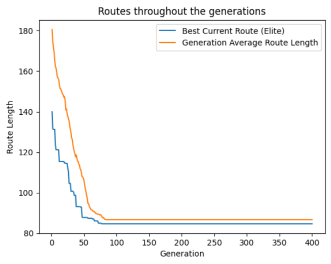
As for the results with the different parameters, right from the start patterns started emerge that presented themselves throughout the whole experiment regardless of the random seeds selected. The first and most obvious pattern is how the general fitness of a generation worked when mutation was set to 0. When mutation was set to 0 the general fitness of a population starts improving by having the average distance of a route decrease but really quickly flattens out and does not go up or down anymore. This is because since no new variants are introduced and the only material the Genetic Algorithm has to work is the initial population once the crossovers figure out the best route the can with what they have improvement is no longer possible without introducing new possibilities to the chromosomes. The resulting routes are clearly not near maximum efficiency

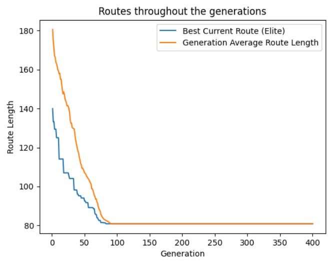


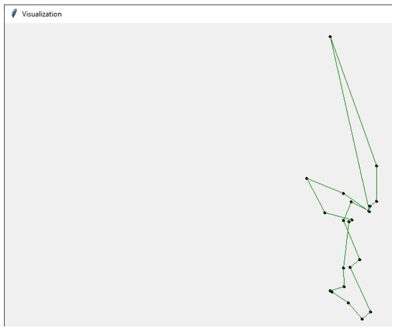


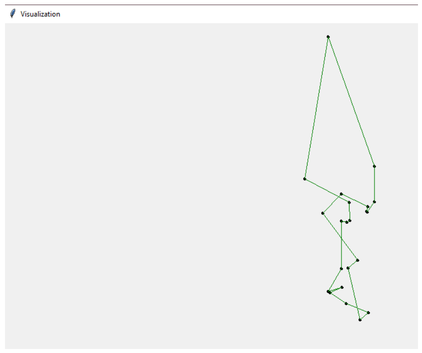






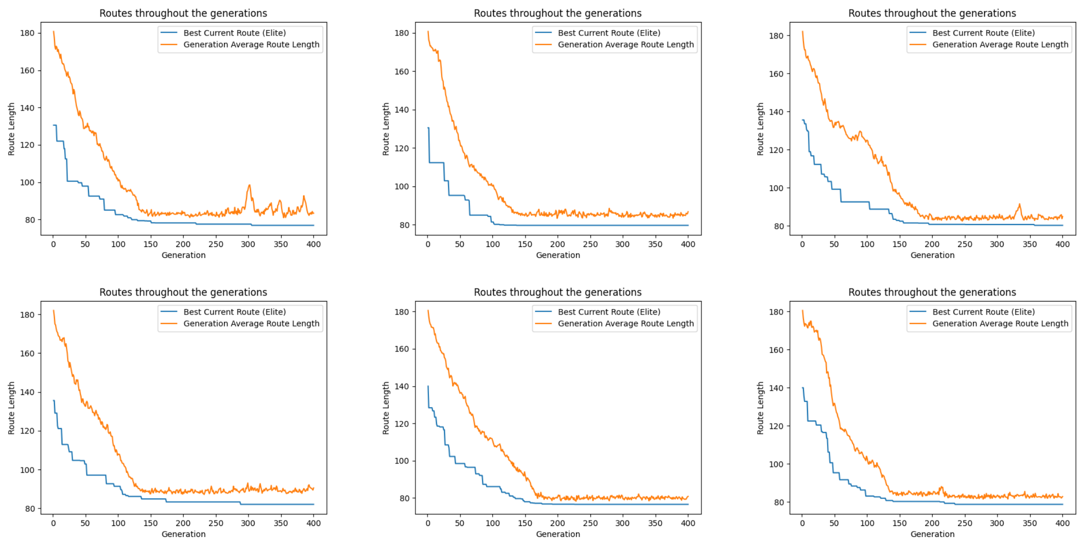


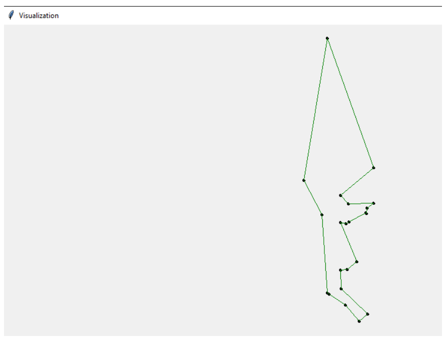


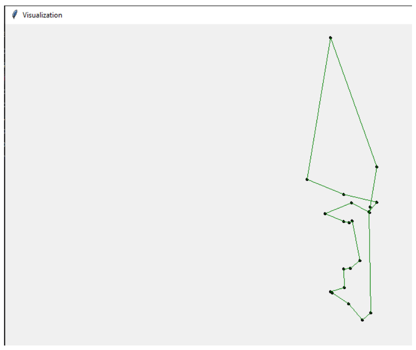


As it is observable while the average of the general population decreases it very quickly flattens out and it stops ever going up or down and this is because with no mutation all chromosomes eventually start to be the same since the chromosomes with lower fitness all end up disappearing and the ones with the higher fitness reproduce until every chromosome is virtually identical. This also makes the best chromosome which is obtained through elitism to also flatten out early into the program since it no longer has material to keep evolving to have a higher fitness.

That issue however is solved when mutation is included into the algorithm, when mutation is included we can observe that the population average fitness is always fluctuating to the very end. This happens because mutation ensures that new variations of chromosomes are always introduced to any chromosome (excluding elites) which keeps the program in a constant state of change, this change can only be positive for the currently best solution (blue line on the graph) since elitism prevents it from ever getting worse. Below are some graphs of runs with mutation included and we can observe how the average population fitness never stops changing, also the routes it produced are more efficient than the ones without mutation.

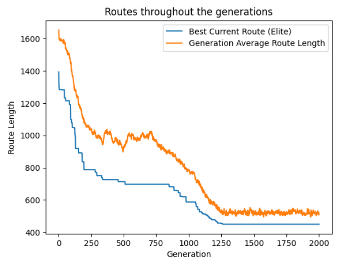


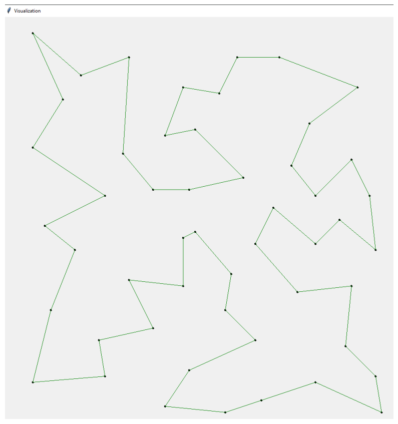




As for statistics such as mean, median, min, max and standard deviation, if the code was run without mutation the standard deviation is extremely small with standard deviation constantly being below 1, at times being even below 0.00001. As for runs with mutation, the standard deviation stayed around 10 since the chromosomes were in a constant state of change. The min and max values for runs with no mutation were almost identical if not identical in some runs for the same reason that eventually all chromosomes started to look like each other or simply being copies of each other. When there was mutation however the min and max always had some space between each other which really varied since there was always a chance that in the last generation a mutation would occur that would make a chromosome highly inefficient resulting in a big gap between the min and max.

As a final additional experiment to show the power of a genetic algorithm I decided to run the code with a generation size of 300 and for 5000 generations for the 55 cities to show how it is possible to get extremely efficient results without brute forcing all possibilities. This additional run lasted for over 10 minutes which is the reason it was a one of a kind type of experiment however it still a good showcase of how far Genetic algorithms can go, this final run has a route of distance 449.32. Below the graph and visual representations are shown.





To check all experiments done with go to the folders attached where the raw data, graphs and visual representations are included.

Discussions and Conclusions

Based on the results we can safely conclude that while mutation adds an element of randomness to the algorithm it is ultimately necessary for the genetic algorithm to become efficient. Without mutation it was shown that the algorithm will end up relying purely on its initial population to make the most of if which is not consistent way to obtain results. As for the crossover methods, the UOX with bit-mask method proved to be far superior to the 2-point crossover.

In terms of crossover rates there did not seem to be a big difference between running the code with either 0.9 (%90) or 1.0 (%100) chance for crossover. Having said that when crossover was set to 1.0 the program did seem to find the best solution it had faster, especially when mutations were non-existent

References

[1] Dr. Beatrice Ombuki-Berman, GA.ppt [PowerPoint Slides] 1-61. Computer Science Department, Brock University.