**Project 4 – Genetic Algorithm TSP**

Stone Barrett

Computer Science Engineering

Speed School of Engineering

University of Louisville, USA

[Scbarr04@louisville.edu](mailto:Scbarr04@louisville.edu)

1. **Introduction**

In this project, I was asked to adapt the idea of a genetic algorithm to be implemented in solving the Traveling Salesperson Problem. The Traveling Salesperson Problem (TSP), as we’ve come to learn, is a variation of an NP-complete computer science problem. The problem entails a salesperson – or in this case, a program – being given a list of cities that need to be visited on his or her business route. The route they must take will complete a Hamiltonian cycle of the graph, meaning that every city will be visited once and only once, sans the starting city which must also be the ending city. The goal of the problem is to find the shortest possible route to take through all the cities. Thus far in class, our projects have been slowly tipping the scales between optimization and speed, where higher optimization requires lower speed and vice versa. For example, brute forcing a solution by generating every single possible permutation of the list of cities and comparing every total cost gives a certainly best solution but takes a very long time to compute; meanwhile, variations of the greedy algorithm can return a solution somewhat quickly, but by nature of the algorithm they may not be very good cost-wise. This time around, the goal was to implement what is known as a genetic algorithm and find a very good solution in very good time, especially considering the significant increase in the number of cities provided.

1. **Approach**

As stated previously, the algorithm being implemented in this project is a genetic algorithm, or one that mimics the logistics of natural selection with its data to provide better solutions as iterations continue to spawn. Levels of relevant data are chosen to represent individual organisms among populations of organisms. Data with fewer levels of abstraction is chosen to represent these organisms’ genetic makeup, with certain data structures representing chromosomes and the data within representing genes. In nature over lots of time, external factors influencing what makes it easier or more difficult to stay alive for a species’ individuals wind up changing the genetic code of what is considered successful by killing off those that are not successful. For example, if the temperature in a region climbed by .5 or 1 degree Celsius over the course of a few hundred years, mammals in the region with less fur may survive longer and have better chances at reproduction. Because of this, genetic code that results in less fur would become more common among the population and the solution to the problem that was increasing temperatures has become more optimized. When genetic code is changed, the outcome of a developing organism or population can potentially change; the same applies when dealing with populations made up of computational information. What’s really the difference…?

Anyway, for this case involving the TSP described earlier, an individual organism is to be represented by a route through every city or an order in which to visit the cities. Populations are to be represented by lists of these lists or lists of routes. Chromosomes are to be represented by subsections of routes. For example, slots 52 through 68 could be a chromosome. A chromosome could be longer or shorter than that example subsection; the lengths are chosen at random when its time for chromosomes to be considered, but that will be discussed later. Genes are to be represented by city placements in the route. If slot 27 in the route to be traversed is taken up by City 89, that is considered a gene. Now that all the data structures and their representations are described for this approach, the methods employed to mimic natural selection are next.

To simulate natural selection, a program needs to be able to create a population of individuals to begin with. This is done by randomly generating routes through the provided list of cities and creating a list of those routes. The number of routes, or individuals, generated for the population can vary, and the effects of different population sizes on the results will be documented in Section 3.

This program also needs to have inherent rules on what is considered “fit” for our desired outcome. In this case, genetic fitness would be considered city placements and combinations of city placements in the route that lead to a lower minimum cost. So, the fitness of each individual in the first and every subsequent generation is determined by finding the inverse of the route’s total cost. This provides a floating point value that will allow for easy handling of calculations for mating pool selection.

To begin the selection of which members of the population get to reproduce, the aforementioned fitness scores are inserted into a “data frame” structure handled by the *Pandas* Python 3.x library. This then generates a weighted likelihood of being chosen in the selection process, with individuals with higher fitness scores having a higher chance of being selected. There are a certain number of individuals that are exempt from this selection process though, as genetic “elites” (individuals that are in the top *x* fitness scores) help genetic algorithms improve towards the goal more quickly than without. I chose to collect data sets using exactly 25 elites in each population, as through trial and error in development it seemed any higher or lower would result in less optimization in the end. Elites have their genetic information automatically transferred to the mating pool, bypassing the roulette wheel selection. The remaining slots in the mating pool are then filled up via random number generation selecting from the remaining members of the population.

Next comes potentially the most important piece of a genetic algorithm – the breeding method. Breeding – or crossover – is the part of sexual reproduction when genetic information from each parent is combined into the resulting child’s chromosomes. The genes from one parent are “crossing over” the genes from another parent. The crossover method used in this program selects a random subsection of one parent route, transfers it to the child route, then fills in the remaining slots from the other parent route using Python’s list comprehension capabilities. If there is a city that has been double-booked by being in both parent subsections, it is replaced with a different city until a free city is reached. The mating pool is divided into pairs of parents and this crossover function is applied to each pair to produce a child until the population limit for the next generation is reached. The structure of this program can be attributed to Eric Stoltz of *TowardsDataScience*. After the next generation has been produced, it’s time to throw in evolution’s chaotic factor: mutation.

Mutation generally occurs in nature when something happens – whether it be radiation, chemical imbalance, or other unforeseen thing – that changes or damages genetic code. Since there are no nuclear weapons targeting my routes or UV rays affecting their skin, the program will need to use some factor to imitate damage to the “DNA”. In this scenario, damage to the genes will be represented by a random chance to have two city placements swapped. Each entire generation is pushed through the mutation method route by route. Each route has a chance to have a random city swapped with another random city in the route. For example, the city in slot 48 could be chosen to be swapped with the city in slot 99. This obviously has the potential the derail the route entirely, increasing the cost significantly; it also has the chance to introduce a new order that could be beneficial to the total cost that will then be carried on through subsequent generations via the crossover method and breeding. Much like real genetic mutation, these changes can be very harmful to our outcomes or beneficial to some degree. As for the chance of mutation, I found a middle ground to stay consistent in trial and error during development. I ran the algorithm multiple times with a maximum mutation chance of 3% and a minimum of .0001%. These data sets varied wildly, but the cost seemed to reach a minimum average at .001%. Anything lower would almost never cause a mutation and anything higher would result in lots of fluctuation that would increase the minimum cost in the ending generation. This will be discussed more in Section 3.

All of these procedures were repeated over and over for thousands of generations per run until a satisfactory dataset was compiled. In order to monitor progress of the algorithm’s runs, the current generations number was printed to the terminal; I also implemented a progress bar to help with the patience of the longer runs using the library *TKinter*.

Graphical user interface

Description automatically generated

Figure 2.1

The improvement curve and shortest route are visualized using *MatPlotLib* and the location data for cities is stored using *NumPy* arrays. Multithreading is used to run 10 iterations of the algorithm at the same time to help quickly build the datasets required. Finally, let’s see how it performed…

1. **Results** 
   1. **Data**

The data used was part of the provided files with the project assignment. Obviously there have been many lists of cities provided for this class, and any of them could have been used to test. However, I decided to stick with the list of 100 cities provided with this project.

The algorithm cannot run until an initial generation is produced by randomly generating routes between the cities provided. These (expectedly) look chaotic and extremely unoptimized; a human eyeballing the map could easily deduce a shorter route.

Figure 3.1

Chart

Description automatically generated

* 1. **Results**

To find and communicate results that represent well what the program is capable of doing with a given set of starting data, I decided to use four sets of non-controlled variables. The number of genetic elites carried to next generations stayed constant as well as the mutation chance; as mentioned previously, trial and error in development showed at least a local minimum path cost with the values I landed on. The testing variable values will be as follows: population size at 100, population size at 200, generation count at 3000, and generation count at 5000.

Table

Description automatically generated Figure 3.2

Now let’s fill in the table.

p = 100, g = 3000

For each of these datasets, I ran the list of 100 cities through the algorithm, ten instances at a time. (Mentioned above using threading.) This gave me a quick set of 10 results in one run. I began with what would be the shortest runs and then moved to the sets that would take far longer to generate. Sheer numbers would dictate that larger population size and generation count would take more computational time. This run, with the lowest of each, clearly would be the first to go. The first generation of this run looked chaotic, as they all will.

Figure 3.3

Chart

Description automatically generated

As the first thread finished, I captured a screenshot of what the improvement curve looked like as evolution progressed.

Figure 3.4

Chart

Description automatically generated

As can be seen in Figure 3.4, there is initially an exponential decline in route minimum cost. This levels out as fewer optimization choices become available towards the end of the evolution. We’ll see similar curves with slight variations in all of these datasets. When the program finished at 3000 generations reached in all 10 threads, some shortest routes found looked much better than the mess we saw in the random beginning.

Figure 3.5

Chart, line chart

Description automatically generated

In Figure 3.5, the visual may still look messy at first glance. This is because *MatPlotLib* color coded the minimum cost path found in all 10 threads. Following a specific color will lead you through a route. Though, it may be a better representation of the optimization this program does to show a path graphically separated from the rest.

Figure 3.6

Chart

Description automatically generated

Something to be noted here is the appearance of paths crossing where the route has already been. We’ll see how this can change as the variables do in later datasets. As for the numbers, this run took the final cost decently low.

Figure 3.7

Graphical user interface, application, table

Description automatically generated with medium confidence

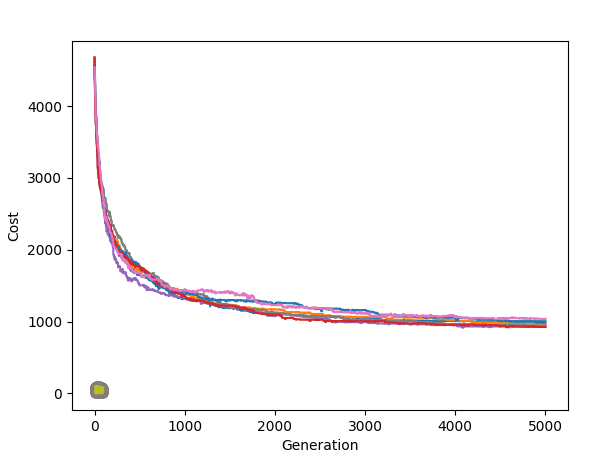
p = 100, g = 5000

Next up is a dataset with the same size population but more generations to be produced. I arrived at these values similarly to how I decided upon the control values – through trial and error during development. I noticed that setting a dynamic cutoff condition (to stop evolution on the current generation) – such as a percentage change requirement to continue – would result in drastically different numbers of generations produced. So, I decided the lower bound of hard-coded number of generations had to be something that gave significant change over a previous arbitrary value (2000, 2500) but would be difficult to notice change from then after. I ran this program a LOT during development, so I got a good feel for when the plateau at the end of the curve would hit. I decided that the upper bound would be the maximum number that I had patience to run, plain and simple. The last dataset took only 40 minutes to run. This one took roughly an hour and a half. I knew as values climbed, so would computational time at a faster rate.

As with last time, we can see the improvement curve, but this time for all threads at the same time. We can also see the values returned by these threads.

Figure 3.8 Figure 3.9

Table

Description automatically generated

p = 200, g = 3000

Now we’ll take the generation count back down to 3000 but increase the population size to 200. This dataset took roughly an hour and a half to run. Just as before, we can see the initial random path, the improvement curve, some final paths, and the total costs.

Figure 3.10

Chart

Description automatically generated

Figure 3.11

Chart

Description automatically generated

Figure 3.12 Figure 3.13

Chart, line chart

Description automatically generated

Table

Description automatically generated

p = 200, g = 5000

Here in the final dataset, we will see the upper bound of both parameters with population size of 200 and generation count of 5000. This run took three and a half hours to complete. Also in this run, just for curiosity’s sake, I managed to capture initial generation minimum path costs from each of the ten threads. This will allow comparison between the randomized paths and the final evolved paths.

Figure 3.14 Figure 3.15

Chart

Description automatically generated

Table

Description automatically generated

Figure 3.16 Figure 3.17

Chart

Description automatically generated

Table

Description automatically generated

As you can see in Figure 3.16, there are still crossovers in this minimum path, but far fewer than when the population size and generation count were much lower. This demonstrates these two variables being increased (at least to the point that I felt was a reasonable computational time limit) will continue to chip away at the minimum possible route cost.

Using the information from Figures 3.14 and 3.17, we’re able to calculate that the average path cost was improved from 4516 at random to 943.3 post-algorithm. That’s a 79% decrease calculated in just over three hours! This may not be as optimal as brute forcing or as fast as greedy algorithm, but it’s almost as close as a solution can get to both.

So now that all the datasets have been collected and analyzed, how does the final table look?

Finished Table

Figure 3.18

Graphical user interface, table

Description automatically generated with medium confidence

This table shows the minimum, maximum, average, and standard deviation of each dataset.

1. **Discussion**

Seeing as how the maximum value for the final dataset is less than the minimum value from the first, it’s safe to say increasing the number of generations and individuals in each population have a positive effect on route optimization. Again, the solutions graphically presented could certainly be improved (as evident by crossovers), even if just by letting the algorithm continue to evolve the dataset for a few thousand more generations.

If I were to redo this project from scratch or even adapt this one more in the future, there are a few glaring changes I would make. To begin with, I wouldn’t be looking for four specific datasets determined by the parameters used, so I would implement a dynamic cutoff condition. I wouldn’t want it to run needlessly forever, but I would also like to get as close as possible to optimal. There would be no hard-coded number of generations to produce; there would be a requirement that to reproduce, generation *n* would need to have an *x* percent decrease in minimum route cost from generation *n-1*. Another thing I would spend more time on is eliminating crossovers in the final route. This issue may be taken care of simply by allowing more generations to evolve, but if not, there could be contingencies in place to solve it. The biggest thing I would change if redoing this project or if granted much more time would be to implement a crossover method of my own design. I did design one but could not implement it in this program in time. The idea was very similar to the one created by Mr. Stoltz, but without complete randomness. The heuristic employed could allow for more efficient evolution. Each route would be broken into pairs of cities. Slots 1 and 2, 3 and 4, and so on would be made into pairs. The shortest *x* paths of these pairs would automatically be carried over to the child route, while the shortest *y* paths between pairs would be carried over from the other parent. If there are double bookings, the remaining child route slots are filled in by the remaining unused cities at random. The problem with this approach is that it requires an even population size. An easy solution to that problem is to check if the population size is even and run this heuristic if so. Otherwise, another crossover method would be employed that works for odd-numbered population sizes.

The biggest problem I had in creating this project was creating a dynamic graphical user interface (GUI) that accurately and simply displayed the data in the end. This is absolutely my first time handling this amount of data in a program – trying to tie all up in a nice bow in the end was the most difficult part. It didn’t help my case that in the first couple projects in this class I didn’t bother to create a GUI whatsoever. That was a mistake!

This project taught me something about genetic algorithms that snuck up on me and I didn’t expect. I think I’ve always sort of idealized the concept of artificial intelligence and its capabilities. When hearing media like television shows, books, and movies and people fluent in pseudoscience discuss the potential of artificial intelligence, “genetic algorithms” were contextualized as what could make machines living things. Obviously, neural networks and machine learning are closer to what these representations of AI were trying to convey, but still not it. These search heuristics that I have either learned anew or learned to implement in practical problem solving are incredibly useful and powerful tools. However, the data is still just data and unblurring the line between cognitive evolution and lists of numbers just being picked because they’re smaller made this experience humbling – there’s so much further to go in this field.

Genetic algorithms certainly show true power and efficiency in problem solving, at least for TSP. I imagine this kind of solving could be applied in all sorts of situations. It crossed my mind a lot over the course of this project that there was something poetic about the idea. We’ve learned so much these last few years: calculus logic, data structures, sorting algorithms, search heuristics and more. It was interesting to see all these awesome tools developed in the field of computer science and other STEM branches come together to more efficiently solve a problem than ever by modeling the design process that brought about all life around us. Given its speed, philosophical implications, and optimization power, GA has become one of my favorite techniques I’ve gotten to work on to date.

1. **References**

NumPy. (n.d.). Retrieved October 22, 2021, from <https://numpy.org/>

*Operator - standard operators as functions¶*. operator - Standard operators as functions - Python 3.10.0 documentation. (n.d.). Retrieved October 22, 2021, from <https://docs.python.org/3/library/operator.html>

*Pandas*. pandas. (n.d.). Retrieved October 22, 2021, from <https://pandas.pydata.org/>

*Pathlib - object-oriented filesystem paths¶*. pathlib - Object-oriented filesystem paths - Python 3.10.0 documentation. (n.d.). Retrieved October 22, 2021, from <https://docs.python.org/3/library/pathlib.html>

*Random - generate pseudo-random numbers¶*. random - Generate pseudo-random numbers - Python 3.10.0 documentation. (n.d.). Retrieved October 22, 2021, from <https://docs.python.org/3/library/random.html>

Stoltz, E. (2021, March 18). *Evolution of a salesman: A complete genetic algorithm tutorial for python*. Medium. Retrieved October 22, 2021, from <https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>

*Time - Time Access and conversions¶*. time - Time access and conversions - Python 3.10.0 documentation. (n.d.). Retrieved October 22, 2021, from <https://docs.python.org/3/library/time.html>

*Visualization with python¶*. Matplotlib. (n.d.). Retrieved October 22, 2021, from <https://matplotlib.org/>