Project 4: TSP – Genetic Algorithm

Hunter Ellenberger

CSE-545-02-4248

Speed School of Engineering

University of Louisville, USA

[hlelle01@louisville.edu](mailto:hlelle01@louisville.edu)

**Introduction**

To start developing the code for this project, I started by importing my usual functions from the tspUtil.py file I have been building all semester. From this file I imported the following functions: get\_coordinates, calc\_distance, and dynamic\_plotter; dynamic\_plotter will be elaborated on later. get\_coordinates was imported so that I could translate the nodes and coordinates located in the TSP file into a dictionary so they may be later parsed through functions. calc\_distance is used to calculate the distance between two nodes. To start this project, I identified the parts that needed to be created to complete the project relating to the GA algorithm. The following parts I identified were: creating the base generation, crossing over members within the generation, mutating crossovers, loading data into something to analyze it, and dynamically graphing routes.

I started tackling these parts in this order. As for the parts relating to the actual implementation of the genetic algorithm, I heavily relied on the video by James Cutajar for reference. This video showed me how to implement the following functions: make\_paths, weed\_generation, offspring, crossover, and mutation. These functions will later be described in the approach section of this report. The following functions I had to implement took some research to figure out how they worked, but were mostly my own doing: compute\_stats\_of\_generation, run\_genetic\_algo, and dynamic\_plotter.

The first function listed is compute\_stats\_of\_generation. This function takes an entire generation and loads the statistics of the generation into a dictionary. This dictionary is then converted to a pandas data frame, which is then loaded into an Excel file for analysis of the generation. In all cases, each generation has the following statistics recorded: maximum path, minimum path, maximum path distance, minimum path distance, average path length, and standard deviation of path lengths. The function itself only takes care of computing this data and loading it into a dictionary, but the rest is handled in the global scope. This data that I obtained will be further discussed in the results section.

The second function, run\_genetic\_algo, was used so that two separate variations of the algorithm could be evaluated. The variations in this case are the number of members a generation will have along with how many iterations that population will go through. This is used to analyze which variation of the algorithm is better. One that has fewer members but goes through more iterations. Or one that has more members but goes through fewer iterations. This will be later discussed in the discussion section.

The final function that was core to completing this project was dynamic\_plotter. A separate function was used to plot the route taken in project three, but this could not be transferred over to plot this graph as it had to be a dynamic GUI. For this, I still used the Matplotlib library, but I instead used animations to graph every minimum path for a generation as opposed to plotting the final route. To figure out how to do this, I heavily referred to the video credited below by Corey Schafer. These animations would plot the minimum path taken for a generation before moving on to the next. These animations were then saved locally to my computer as .mp4 files. This element alone contributed greatly to the runtime of the program. Without this functionality, the runtime of the algorithm would be much quicker.

**Approach**

For this project, I am using the Genetic Algorithm which gives a better solution the more it runs. It does this by crossing over the genes of two “parent” paths and evaluating this child with other children to see the best and most fit. In this instance, fitness is defined as the shortest path that must be taken. This child is then given the chance of “mutating”. In this implementation of the algorithm there is a 1% chance that a child has two random nodes in their path randomly swapped. It then passes this most fit, and possibly mutated, child into the next generation. This is then repeated until the completion criteria are met. The completion criteria, in this instance, is defined by how many generations are produced.

To start implementing this algorithm, I needed to generate the first generation of paths that would be parsed to start the process. To accomplish this, I made the function make\_paths, which takes the number of members you want in the generation and the number of nodes you want per member. With these two parameters given, it returns an initial generation of randomly generated paths.

This generation is then passed into the weed\_generation function that weeds out half of the members of this population. It does this by cutting the generation in half and then comparing a member from one half with another member from the other half. The comparison is made using the distance, whichever member has the fewest distance is returned as it is considered fit enough to move on to the next generation. The child that is not found fit is not returned, eliminating it from the population.

This surviving generation now produces offspring through the same methodology. The generation is now split in half and combines genes. In this case, subarrays of nodes are combined. This is done by randomly selecting a segment from the genes of one parent and passing those genes over to another parent. This goes both ways as two children are created for each parent pair. One child takes the genes from one parent and one child takes the genes from another.

This returned generation is then finally modified through the mutation function. This function emulates mutations that take place within a generation. In this implementation, the chance that a member mutates is 1/100. I found that this is a good number as increasing their opportunity to mutate usually results in a lack of consistency between generations, however, a lower chance usually results in too many local conversions where all members of a generation are the same. If a member of the generation mutates, two random points within the route are swapped.

**Results**

*Data*

For my analysis of the genetic algorithm, I decided to have one variation that has more members with half the generations. The other variation has the opposite, half the members with more generations.

In the case of the algorithm that has more members, we see that it does not converge until much later generations. This is indicated by the standard deviation being around 200 for most generations. We only see a decline in this deviation after the 2,200 generation where we see a rapid increase in improvement, indicated by the lower minimum distance. The only theory I can reason this late convergence into a more optimal solution would have to be due to the mutation of members. Since there are more members in this instance of the algorithm, there is a higher chance that the members will mutate, possibly ruining a more optimal member. However, we see this breakthrough at around generation 2,200 where it rapidly improves, even getting close to the distance held by the solution presented in the other variation. Overall, this solution has a lot more diversity which does end up resulting in a less optimal solution in this case.

In the other variation, we see many more local traps, that is places where the algorithm gets stuck and there is little diversity between members, essentially convergence towards the same path. In its most extreme form, it results in the standard deviation of a generation being zero meaning all of the members have the same paths. This extreme convergence does not appear until generation 802 and is not seen again until generation 1018. This is not a recurring theme that is seen though, unlike previous generations where the mutation rate is lower, and the generation quickly makes it out of this extreme convergence within one generation. As compared to the other variation of the algorithm, that of more members and fewer generations, it slowly improves. This variation reaches a distance below 2000 in just less than 450 generations whereas it takes the other more than 2000 to achieve. Around this point, we also see the standard deviation of each generation reaching a point of around 20 which indicates sluggish improvements from this point until it reaches its final most optimal path at generation 5000.

These algorithms took a little less than 30 minutes to run. This extreme runtime was mainly due to the rendering of the videos shown in Figures 1 and 2. The variation that had more members had a final path distance of 1041.201 and went through a total of 2500 generations. The other that had fewer members, but more iterations had a final path distance of 1003.293 and went through 5000 generations. In both cases, the algorithm does very well at reducing the original ~4000 length of a path down to a more optimal one but in this case, the one with fewer members seems to be more optimal with this problem.

*Results*

Figure 1



Figure 2



**Discussion**

The largest problem I had with this project was working with all the needed elements. Whereas with the last project we had to only worry about getting the algorithm to work and graphing the results, we now had to work with a much more difficult algorithm along with incorporating many more systems such as statistical analysis and a dynamic GUI. It no doubt helped me become a better programmer and have a much more intricate understanding of these tasks, but it was a daunting task, nevertheless.

If I were to do this project over again, I would analyze something different like a different mutation function where it mutates more nodes or a different crossover function where multiple segments are crossed over instead of just one. I believe that an analysis of these factors would be much more interesting compared to that of just more iterations with less members.

From using the genetic algorithm, I have gained a deeper consideration of the algorithms that have been developed for the computer science field. Many algorithms are already stunning with how optimized they are, but in this case, we are emulating something from real life. This emulation is where my admiration comes from, instead of using an algorithm that essentially only relates to the CS field we are using something that we can view the direct product of in everyday life. We ourselves are a product of evolution and I think it is very intriguing that we can even emulate this aspect of life through code.

Although it has been a lot slower than our other algorithms, I do really appreciate this algorithm. It can be applied to solve any problem that we may have, even if we do not have the compute for it now. The emulation of real-life behavior is also something I really appreciate. This has definitely been my favorite algorithm to learn so far.

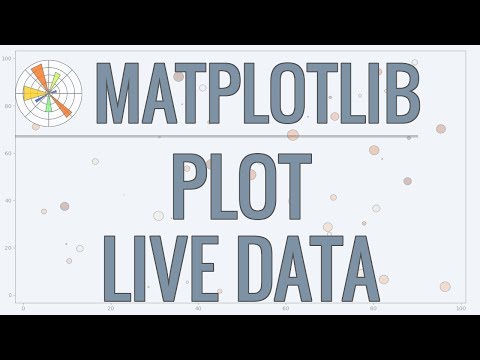
**Sources**

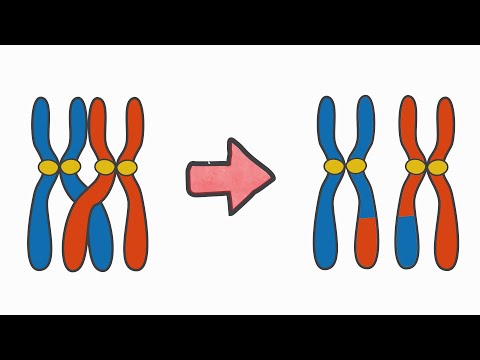
My sources included the following:

<https://www.geeksforgeeks.org/exporting-a-pandas-dataframe-to-an-excel-file/>

<https://youtu.be/Wgn_aPH3OEk?si=nRDEE53VWvci0WbT>

<https://www.youtube.com/watch?v=Ercd-Ip5PfQ&t=976s>

[](https://www.youtube.com/watch?v=Ercd-Ip5PfQ&t=976s)

[](https://youtu.be/Wgn_aPH3OEk?si=nRDEE53VWvci0WbT)