12/3/21

MET CS 767 Assignment 5T: GA’s

*Nathan A. Horak*

Please fulfill the requirements in the gray text, handing in this filled-out template and the accompanying source code, leaving the gray text and the headings unchanged.

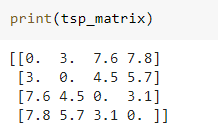
You are to implement the *traveling salesman problem* using a genetic algorithm in the manner outlined below, effectively described in the “*Crossover”* section—which you are advised to read first. You can assume that there is a route connecting every pair of cities, and that no two distances are equal.

Eric Braude created this technique independently as a way to perform trouble-free crossover for Traveling Salesman but it may be published. If so, please do not use the work of others in responding.

# Representation of the Data

## Taking into account how crossover below will be defined, explain how the application will *represent the data* (e.g., “a list consisting of …”). As an example show the following is to be represented: B(oston) to L(ondon) 3(k miles), L to M(umbai) 4.5, M to S(hanghai) 3.1, S to L 5.7, B to M 7.6, and B to S 7.8.

A genetic algorithm is a stochastic global optimization algorithm that works by performing an optimization procedure inspired by the biological theory of evolution by means of natural selection, according to Jason Brownlee of machinelearningmastery.com[3]. The first step to creating a genetic algorithm is picking an appropriate method of genotype representation. This is also one the most important steps of creating a genetic algorithm, according to tutorialspoint.com[4]. Taking into account the crossover functionality described in Section 3, I will represent the data in two parts. The first a list of integers utilizing integer representation (chromosome) and the second a distance matrix of floats (showing distance between cities). I will establish ‘0’ for Boston, ‘1’ for London, ‘2’ for Mumbai, and ‘3’ for Shanghai as the respective city starting and ending points. Given the established numerical values for each city shown above, here is an adjacency matrix with indices represented by each city’s equivalent number:

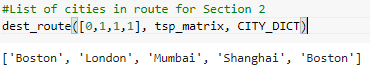


It is interesting to note that of the representation options available to machine learning engineers for genetic algorithms, I am using integer representation instead of permutation representation. According to tutorialspoint.com, permutation representation represents a chromosome with an order of elements[4]. While this seems like a strong choice for a genetic algorithm implementation of the traveling salesman problem, we will not use it in this implementation (see Appendix A for more information).

# Representation of a Route

## Explain how the application will represent *a route* starting and ending in Boston. Include, as an example, how the following route is to be represented: *B to L to M to S to B*.

I will represent a route consisting of four cities with a list of four elements. The initial (zero) element will reference the starting city (from Section 1). Each subsequent element of the list will, in accordance with Section 3, represent the nth closest city. ‘1’ will represent the closest unvisited city, ‘2’ the second closest unvisited city, and so on. The final aspect of the route, returning to the starting location, will not be represented in the chromosome. Instead, it is assumed (and coded as such) that the route returns to the city represented by the first (zero) element. As mentioned in Section 1, the above representation is not enough to stand on its own. A distance matrix is needed to show the distance between each city. The route starts in Boston, represented by ‘0’- that is the first integer of the list. The route goes to London (‘1’) next. Given that London is the closest location that hasn’t been visited, the second index is ‘1’. The route continues to Mumbai (‘2’). Mumbai is closer than Shanghai to London, thus the third element of the list is ‘1’. There is only one city left to visit (Shanghai) and the final element is ‘1’. Using this methodology, the following chromosome represents the described route: [0,1,1,1], see below:



# Crossover

## Define a crossover function. The implementation should create a child from two parents as in the following example. Bold and italics are added to clarify what part of the child comes from what part of the parent.

## Parent 1 Boston 🡪 2nd closest unvisited city[[1]](#footnote-0) 🡪 2nd closest unvisited city[[2]](#footnote-1) 🡪 closest unvisited city 🡪 Boston

## *Parent 2 Boston 🡪 closest unvisited city 🡪 2nd closest unvisited city 🡪 closest unvisited city 🡪 Boston*

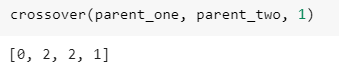
## If the crossover point is at 1, the child route will be

## Boston 🡪 2nd closest unvisited city *🡪 2nd closest unvisited city 🡪 closest unvisited city 🡪 Boston*

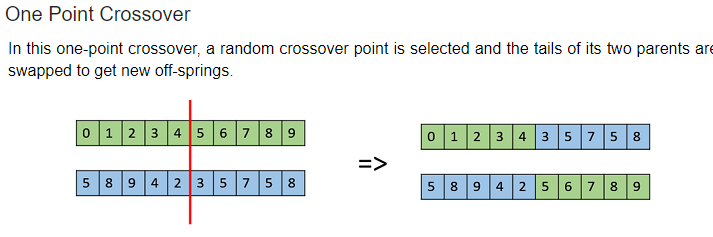
A crossover function is a function that creates a child chromosome from two existing parent chromosomes. According to towardsdatascience.com, crossover is the most significant phase in a genetic algorithm[1]. When a pair of parents mate, a crossover point is chosen at random within the genes. From the Section 3 prompt, crossover points will be marked in accordance with python indexing (index ‘0’ the first gene of the chromosome). The function takes genes from each parent to pass on to the child and selects the genes it will pass down by the defined crossover point. All genes with an index smaller or equal to, or the left visually and including the crossover point are passed down from the first parent to the child. All genes greater, or the right visually are passed down to the child from the second parent. See below for how the crossover function performs with a stated crossover point:



With crossover point at 1:



According to tutorialspoint.com, this method is known as one point crossover and is demonstrated in the figure below[4]:

**

*Visual representation of One Point Crossover, per tutorialspoint.com[2].*

# Mutation

## Explain (clearly) how the application will perform mutation.

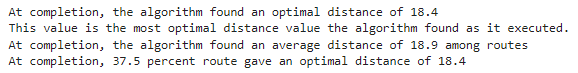
According to geeksforgeeks.org, a mutation operator inserts random genes in offspring of parents to maintain diversity in population and avoid premature convergence[2]. Per Vijini Mallawaarachchi, mutation usually occurs with a low, random probability[1]. I will use this methodology in this problem. Mutation will occur with a small, random chance after each child is created. Each epoch, every chromosome will be paired with another chromosome in a weighted random fashion. The weights will reflect how short the route distance is for the chromosome, with shorter distance chromosomes prioritized as partners. This method is very similar to the method described by David E. Goldberg in his book *Genetic Algorithms in Search, Optimization, and Machine Learning*[7]. It is also close to the roulette wheel selection described by tutorialspoint.com[4]. However, there are a few differences between the roulette wheel selection method (see more in Appendix B). A crossover point will be chosen randomly for each creation instance and then there is a small chance that a mutation will occur on one or more of the child chromosome’s genes. There are some constraints to how mutation can occur given the construction of our genetic algorithm model (see more in Appendix C).

Because of how routes are represented within the model, each indexed element of the chromosome has a different number of possible values it can mutate to (see Appendix C). I decided to give genes a chance to mutate for each possible value it could mutate to. While it may seem like I am inappropriately favoring genes to mutate over others, I think this was a justifiable choice given how routes are represented in the model. The indices of chromosomes are not equal in importance throughout the data structure. The starting location and the first jump are much more important than the penultimate and last index (which is always ‘1’). In addition to importance, there are more possible values for the first two indices (7, 4 and 3, in total for a route of 4 cities) compared to the last two (3, 2 and 1, for a route consisting of any number of cities). However, I did include code in my mutation function that would move to the next index in the element whenever a mutation fired. This would prevent a gene from mutating from one value to another, and then back to the original value within the same generation.

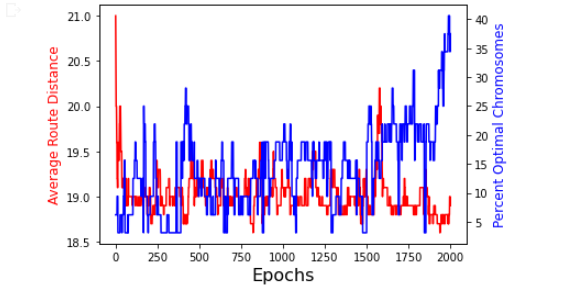
# Result on the Given Data

## Describe the result from executing on the data … *B(oston) to L(ondon) 3(k miles), L to M(umbai) 4.5, M to S(hanghai) 3.1, S to L 5.7, B to M 7.6, and B to S 7.8*. What do you think of this result? Explain.

Before executing the algorithm on the data, I wanted to determine a way to evaluate the performance of my functions (see Appendix D). Based on how I approached crossover, the parent selection method, and mutation process, I wanted to give the data a chance to improve itself while maintaining genetic diversity. I decided to use a population size of 32, number of epochs at 2000, and a mutation rate of 0.001 (one tenth of 1%). Then I created 32 chromosomes representing traveling salesman problem (TSP) routes randomly, in accordance with the constraints related to Section 3. The model proceeded through 2000 epochs of selection, reproduction, and mutation as outlined in Sections 3 and 4. See the optimal distance, average distance, and percent optimal below:



First, it’s worth noting that the algorithm found the true optimal distance. Given that 37.5% of routes at the end had the optimal distance, it seems likely that this optimum value was found early. Even if 62.5% of routes did not have a route with the optimal distance, most of them were likely close given the average distance value of 18.9. See below for a graph of the results over time:

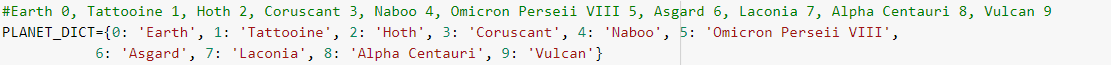


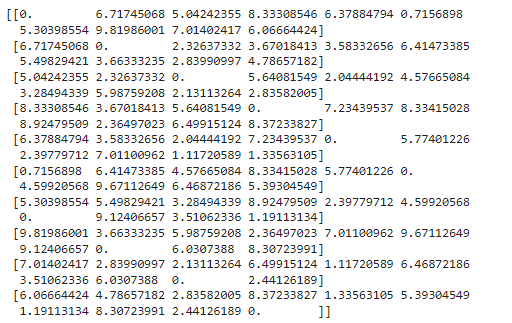
While there was a decent amount of noise in the middle epochs for Average Route Distance (red) and Percent Optimal Chromosomes (blue), some obvious patterns stick out. The red line starts in a drastic freefall at the very start from around 21.0 down to 19.0, reflecting that the algorithm was quick to lower the average route distance. Afterwards, the red line goes up and down repeatedly for almost 2000 epochs but not before it slowly starts trending even lower at the very end, dipping below 19.0. The blue line starts under 10% and fluctuates repeatedly for more than 1500 epochs between 5 and 20%. However, at the very end the blue line soars past its previous highs, hitting 40%. Corresponding with the red line, this indicates that in the last few hundred epochs, the chromosomes drastically improved with many converging to the optimal distance value.

# Result on Your Data

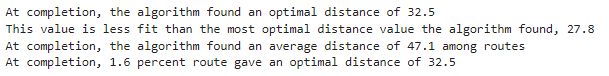
## Describe the result from executing on illustrative data of your choice. What do you think of this result? Explain.

Doggel is a spaceship engineer who has two months off work. He wants to visit the 10 most famous planets in the galaxy and spend the least amount of time traveling possible. I was hired as a consultant to help him identify the optimal route to minimize travel time. The planets and the distances between each planets is represented by the following (distance in googolplons):

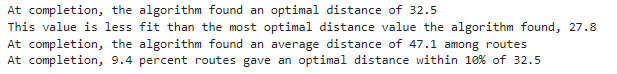




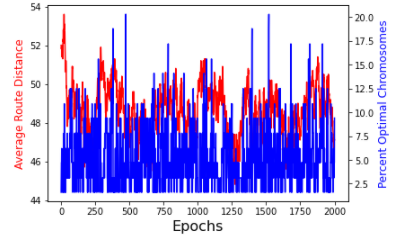
I was pleased with the constants I used in Section 5 but I wanted to adjust them given the new data I was working with. The data from Section 5 had 4 locations, 6 connecting routes between any two cities ((½)(4)(3)), and 24 unique routes (4!) satisfying the traveling salesman problem (TSP). However, this data has 10 locations, 45 linked paths between any two cities ((½)(10)(9)), and 3628800 unique routes (10!) for TSP. Given this, I increased the population by a factor of 2 to 64 from 32. I also increased the mutation rate fivefold from 0.001 to 0.005 (0.5%). By increasing the population size and number of epochs, I will give the algorithm more chances to mutate and pass on desirable genes and by raising the mutation rate, I’ll increase the speed that the data changes outside of replication. I made these changes within the framework of the design choices I laid out in Sections 1, 2, 3 and 4. See below the optimal distance, average distance, and percent optimal:



Comparing this to the results from Section 5, the first thing to note is that because I don’t know what the true optimal route is, I can’t say for sure that the algorithm identified the value. The optimal value at completion (distance of 32.5) wasn’t even the best route found over the execution of the model (distance of 27.8). Secondly, there is a huge difference between the stated optimal distance and the average distance- over 50%. This demonstrates that there are still routes with significantly larger distances compared to the optimal remaining in the population. Lastly, the percentage of routes with an optimal distance is very, very small. Given the population size and this value, there is only one chromosome with a distance metric measuring 32.5. This seems justifiable when compared to the data from Section 5 because there are over a hundred thousand times more unique routes possible comparing Section 6 data to Section 5 data. There are just so many possible routes for the algorithm to evaluate and learn from. The metric percentage of routes with an optimal distance was very low and I wanted to adjust this. I wanted to learn more about how well values were evolving and fitting over time, so I decided to add a cushion of 10% towards considering a chromosome ‘optimal’ (See more on the original implementation results in Appendix E). Here are the optimal distance, average distance, and percent optimal metrics given this adjustment:



The addition of the cushion mentioned above improved the percent optimal from 1.6% to 9.4%. This reflects 6 points fitting the criteria, up from 1. I was curious about seeing the graph of these metrics over epochs. Below a graph of the results over time:



The first thing that sticks out is the reduction of visual clarity pertaining to the blue Percent Optimal Chromosomes line. This line unfortunately jumps around very erratically to the point of distraction. However, the red Average Route Distance line has more of a pattern to it with it sharply falling off from the start. The drop is not nearly as steep as the graph from Section 5, with the line trending steadily downward from above 52.0 to below 46.0 within the first 250 epochs. After this point, the line erratically moves up and down in almost equal waves before it dips to around 44.0 just after 1250 epochs. But the red line surrenders its gains and creeps back upwards as the algorithm runs to completion. The blue line jumps maniacally around between 1.6% and 20% with little noticeable trend beyond a trend increase at the very beginning as the red line fell. It seems that there were multiple points throughout the algorithm’s execution where the algorithm had a chance to improve even more, but it’s progress was halted by either an unfortunate mutation or strong genes lacking a significant role in replication and the results regressed. The above results raise a decent amount of doubt concerning the performance of my genetic algorithm. I attempted to strike a balance between allowing for genetic diversity and rewarding strong genes, but it is possible that my mutation and replication methods hampered the ability of the genetic algorithm to operate on a more complex and robust dataset.

# Source Code

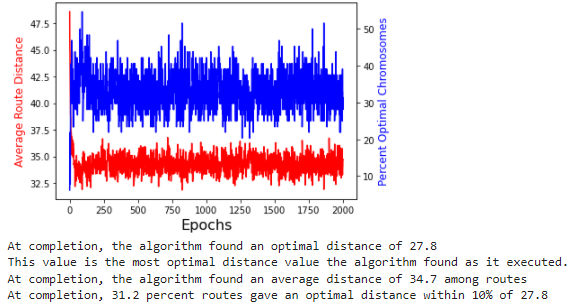
## Paste your source code below. It should accompany this doc as well.

<https://colab.research.google.com/drive/1smINGqPqpiR8tYct--l9FBEApO4EXzfi#scrollTo=ktIz2T7Rrzah>

# Comments on Performance

## Compare the performance of this TSP GA with at least one known TSP GA implementation.

I decided to compare my results to an example I found online, published on towardsdatascience.com by Eric Stoltz[6]. Reading through his reasoning, two things stuck out to me regarding how he approached the problem. First, was his use of elitism. Per Stoltz, elitism allows the best performing individuals from the population to automatically carry on to the next generation, ensuring the most successful individuals persist[6]. In my implementation, each generation would entirely replace the subsequent, whereas Stoltz allows the most successful chromosomes to continue on. I was concerned about poor convergence when I made this choice, but it seems likely that with a proper elitism parameter, the algorithm can progress and improve without poor convergence. The second was his use of tournament selection. I debated using this method but decided I wanted to use a weighted metric similar to Goldberg’s from the module notes (see more about tournament selection in Appendix F). After incorporating Stoltz’ code, I made a slight alteration in the geneticAlgorithmPlot() function to allow a visual output similar to the plots in Section 5 and 6. I set the elitism pool at the population size divided by 5, matching the ratio in Stoltz’s example (20 eliteSize for 100 popSize). See the results below:



The first thing that I noticed is that the results are much better than the results I obtained. There was no shorter distance found than the optimal value in the last generation. All of the performance metrics are better: the optimal distance (27.8 vs 32.5), average distance (34.7 vs 47.1), and percentage optimal (31.2% vs 9.4%). This algorithm clearly performed better. I think that in retrospect, some of my ideas concerning the parent selection method led me astray. My reasons for selecting the method stated in Sections 4 and 5 were solid but the approach I took unnecessarily sacrificed performance of the algorithm. I could have maintained good genetic diversity and avoided poor convergence with a measured approach similar to Eric Stoltz’s. The last point to note is how most epochs in the above plot appear to be unnecessary. There is a sharp initial drop in the red line and a drastic increase in the blue line, before a long period of fluctuation. There are no meaningful improvements in metrics past the first hundred or so- as such, I wanted to analyze the results with a smaller epoch size (see Appendix G).

# References

Show that you used a wide variety of resources by listing them below and clearly indicating in the body above where you used. Make sure to use proper referencing in your paper. We suggest using the APA format, but other formats are fine as long as they clearly distinguish your work from the work of others in your response. In general, observe the stated plagiarism rules.

[1] Mallawaarachchi, Vijini. (2017, July 7). *Introduction to Genetic Algorithms - Including Example Code*. towardsdatascience.com. Retrieved December 4, 2021. <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>

[2] (2021, September 27). *Genetic Algorithms*. geeksforgeeks.org. Retrieved December 4, 2021. <https://www.geeksforgeeks.org/genetic-algorithms/>

[3] Brownlee, Jason. (2021, October 12). *Simple Genetic Algorithm from Scratch in Python*. machinelearningmastery.com. Retrieved December 5, 2021. <https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/>

[4] *Genetic Algorithms - Quick Guide*. tutorialspoint.com. Retrieved December 6, 2021. <https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_quick_guide.htm>

[5] Sugihara, Kazuo. (1997). *Measures for Performance Evaluation of Genetic Algorithms (Extended Abstract)*. 3rd Joint Conference on Information Sciences (JCIS ‘97). Retrieved December 6, 2021. <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=C823C029F34116EF98B86EBC41D410EA?doi=10.1.1.49.8611&rep=rep1&type=pdf>

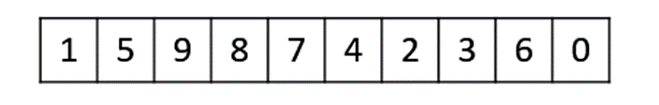
[6] Stoltz, Eric. (2018, July 17). *Evolution of a salesman: A complete genetic algorithm tutorial for Python*. towardsdatascience.com. Retrieved December 7, 2021. <https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>

[7] Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning* (13th ed. 1988).

# Appendix

[A]

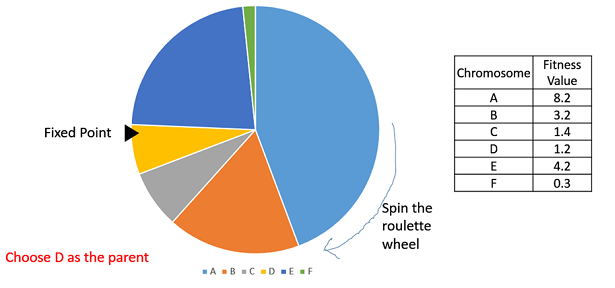
From Section 1, permutation representation is a method to represent elements by order and is known to be useful in the traveling salesman problem (TSP) according to tutorialspoint.com[4]. While known as a “classic example of this representation”, permutation representation was a poor choice for this assignment given the project requirements. Specifically, the way that routes are defined according to Section 3 causes issues with this method. Stops in the journey past the starting location are noted by the closeness of the stop relative to others, not by a marker for location. It is not possible to square up the use of this method with the way that routes are to be expressed from Section 3.



*An example of Permutation Representation, from tutorialspoint.com[4]. Note how constructing a route by ordered stop location is not feasible with the requirements of Section 3.*

[B]

As mentioned in Section 3, my genetic algorithm will utilize a parent selection method similar to David E. Goldberg’s method described in Module 5 and the roulette wheel selection method[7]. My design is as follows: each epoch, every chromosome will be the parent of at least one child as a child replaces their parent each generation. Chromosomes with a low evaluated distance (and high fitness value) are more likely to be chosen as mates and thus high fitness chromosomes are likely to have more children compared to a chromosome with low fitness. According to tutorialspoint.com, parent selection methods are chosen to allow subsequent generations to fit a better and better solution while maintaining good genetic diversity[4]. A population getting overrun by an extremely fit solution can lead to premature convergence, where the model settles on a non-optimal solution. This is an undesirable condition in a genetic algorithm[4]. I wanted to avoid this by maintaining high genetic diversity throughout the process but also allow for the chromosomes to improve by crossover and mutation.



*Visual representation of the Roulette Wheel Selection method, very similar to the method used in this problem, per tutorialspoint.com[4].*

It is also worth mentioning that due to children replacing all adults each generation, there are no survivors. Per tutorialspoint.com, survivor selection methods are used so fitter individuals are not removed from the population while diversity is maintained[4]. I opted to not include survivor selection and instead replace the population each generation. I thought that because of my strategy of choosing mates by fitness and letting each chromosome mate at least once, I can maintain fit genes while maintaining strong genetic diversity.

[C]

Due to how I approached the problem (described in Section 2), we can’t mutate an element into a random number. The only feasible value given our construction of the last value of a chromosome list is ‘1’. The only fitting values for the second-to-last value of a chromosome list are ‘1’ and ‘2’. This is because when there is only one unvisited city left, there is only one chance to be made. Likewise, at the second-to-last point, there are only two choices to be made. Given this, these are the following rules for mutation: first, the initial (zero) index element can be any element from to the length of the chromosome, minus one [\*range(0,len(chromosome))]. Given our problem looking at four cities, the initial element can be ‘0’, ‘1’, ‘2’, or ‘3’- the route can start from any of our four cities (Boston, London, Mumbai, or Shanghai). Secondly, the next element (one) represents which closest city we will travel to. Given we are already at our starting city, we can visit the closest, second-closest, or third-closest city. This element can be ‘1’, ‘2’, or ‘3’ [\*range(1,len(chromosome))]. Third, the next element (two) represents the third city we will visit and this city is either the closest or second-closest. This element can be represented by ‘1’ or ‘2’ [\*range(0,len(chromosome)-1)]. And as mentioned above, the last element can only be ‘1’ [\*range(0,len(chromosome)-2)].



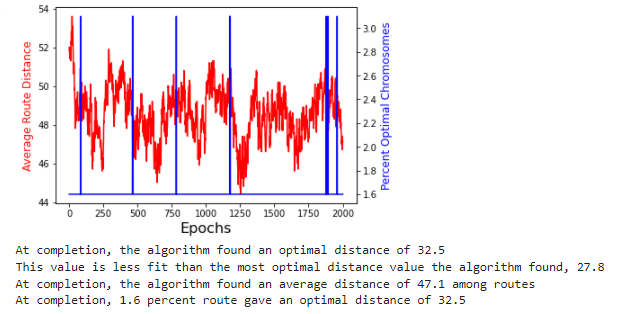
*How mutation functions in a chromosome, with mutated genes in light green, from geeksforgeeks.org[2].*

[D]

Evaluation methods are important tools that allow a machine learning engineer to examine the effectiveness of an algorithm. I wanted to identify metrics that I could compare results between different setups. According to Kazuo Sugihara, two metrics that are useful are likelihood of optimality and average fitness value[5]. Sugihara writes that the likelihood of optimality ‘k’ is the estimated percentage of runs that produced an optimal solution within ‘k’ generations. This metric sounds great in theory and I would be able to use it for Section 5 given that I knew the true value of the lowest possible distance, but I did not know what the optimal distance was for the data in Section 6. Given this, I chose to define this metric slightly differently. I would find the shortest distance among the chromosomes per generation, and then calculate the percentage of chromosomes in the generation whose routes have that distance. Average fitness value was much more straightforward to apply to the problem. Per Sugihara, the average fitness value at the ‘k’th generation is the average of the best fitness values obtained within ‘k’ generations in the ‘n’ runs[5]. Given that I was executing a single run, the average fitness value is the average distance of all chromosomes for each generation.

[E]

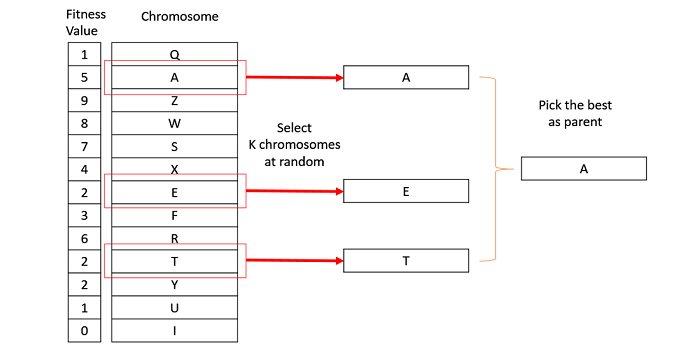
<https://colab.research.google.com/drive/1smINGqPqpiR8tYct--l9FBEApO4EXzfi#scrollTo=91s33J4G0Udv>



As mentioned in Section 6, I opted to include a cushion when identifying optimal distances due the extremely low percentage for percent routes giving an optimal distance. In the above graph, you can see this low percentage bear out in the blue line. The blue line is essentially flat at 1.6% over 2000 epochs except for a few locations where it hits 3.2%. Visually, these locations look like thin blue vertical lines. It was my hope that adding a cushion to this metric would provide improved clarity, but the updated results turned out to not be much better.

[F]

Tournament selection is a parent selection method that is commonly used in genetic algorithms. With this method, a set number of individuals are randomly selected from the population and the one with the highest fitness is chosen as the first parent. This method is repeated to choose the second parent[6]. According to tutorialspoint.com, tournament selection is extremely popular in online literature as it can work with negative fitness values[4].

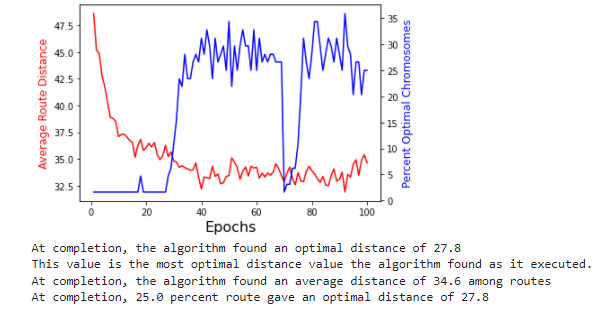


*Visual representation of Tournament Selection, via tutorialspoint.com[4].*

[G]

<https://colab.research.google.com/drive/1smINGqPqpiR8tYct--l9FBEApO4EXzfi#scrollTo=RHePZ6rYN8du>

I wanted to analyze the results of the traveling salesman problem on Doggel’s prospective vacation but with a smaller epoch number. I also wanted to make another change by removing the cushion applied to the percentage optimal calculation. Given the drastic improvement going from my algorithm to Eric Stoltz’s, I surmised that this cushion may not be necessary. Here are the results and plot output:



My hunch in Section 8 proved to be correct. The last 1900 epochs are unnecessary for this genetic algorithm applied to Doggel’s problem. At completion, the optimal distance was found to be the same, the average distances are almost equal, and 25% percent of routes gave an optimal distance of 27.8. Given that 31.2% percent of routes gave an optimal distance within 10% of 27.8 in Section 8, 25% with no cushion seems like an excellent result.

There are two trends I wanted to mention related to the graph beyond the initial drop of the red line or rise of the blue line and how this ties in to some aspects missing from my model. The rise of the blue line does not happen at the same time as the drop of the red line. To me, this indicates that a very strong chromosome was sticking around through successive generations replicating it’s genes. Over this period, other genes were becoming more and more fit- lowering average distance. After around 25 epochs, the elite gene had spread his genes far enough that the percentage of optimal chromosomes shot up from 1.6% to 25%. The rise of the blue variable also corresponded with a drop (and improvement of fitness) for the red variable. The second instance is the drastic drop of the blue variable around the 70th epoch. Through either an unfortunate round of mutations or poor replications, a multitude of distance optimal chromosomes were replaced. This coincides with an uptick in the red variable. However, the memory of all the strong genes were not lost as the blue variable rebounds after a few epochs. In my opinion, these two trends speak to not only the powerful fitting force of Stoltz’s method, but also its resiliency to withstand unlucky mutations and poor replications. If I were to construct a genetic algorithm from scratch again, I would be strongly encouraged to include tournament selection and definitely include elitism.

# Evaluation



1. i.e., compared to all other direct hops from the city just visited (Boston in this case) [↑](#footnote-ref-0)
2. i.e., compared to all other direct hops from the city just visited [↑](#footnote-ref-1)