**Solving problems using genetic algorithm**

**21900239 노재영**

**22000741 차신국**

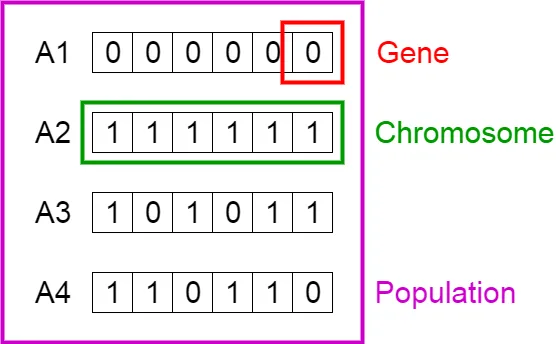
**1. Introduction of genetic algorithm**

Genetic algorithm is one of the most popular and widely used biologically inspired heuristic algorithms. It is inspired by natural selection, the fittest individuals survive and reproduce more similar offspring while weak ones are eliminated with the passage of time. The theory is based on genetic recombination and genetic mutations. They are commonly used to generate high-quality solutions for optimization problems and search problems[1],[2],[3],[4]. Genetic algorithms are applicated at RNN(Recurrent Neural Network), mutation testing, data mining and clustering, image processing, traveling salesman problems(TSP), vehicle routing problems, mechanical engineering design, etc.[5]

**2. Steps of genetic algorithm with Python code**

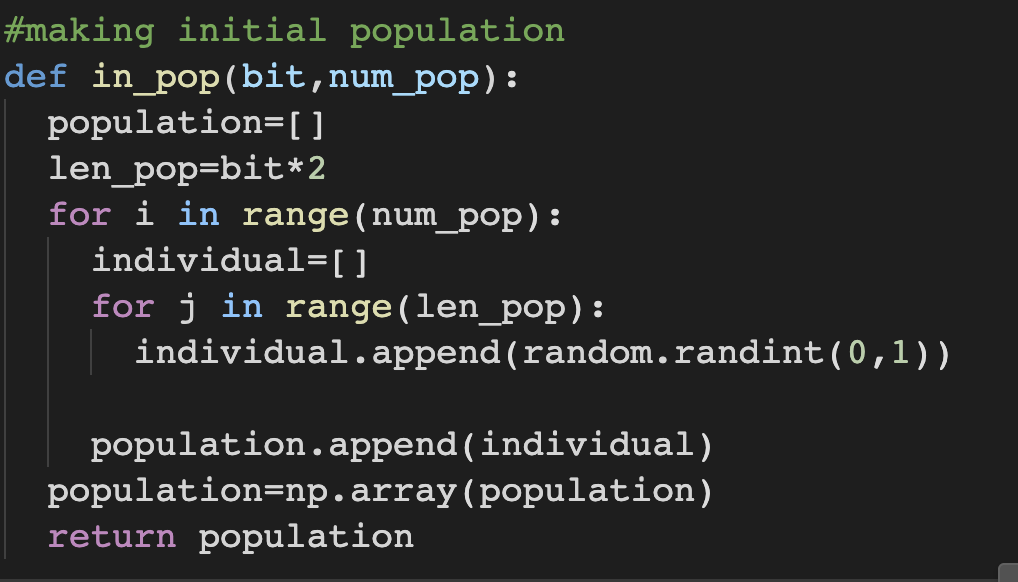
**Step 1. Initialization**

Genetic algorithm starts with processing a set of individuals using the random bit-string. Each of the values in the bit-string is genes, and the string is the chromosome. They are referred to as initial populations. The length of each population is defined by (number of bits \* number of design variables). Figure1 shows gene, chromosome, and population in genetic algorithm.



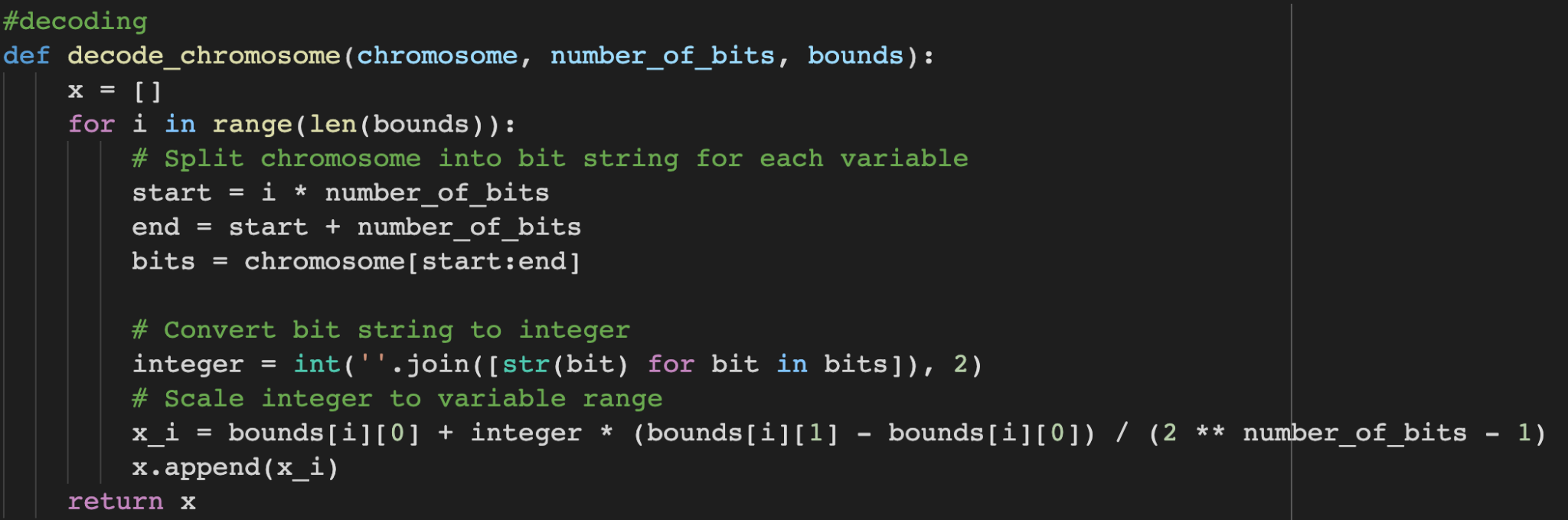
**Fig 1. Gene, chromosome, and population in genetic algorithm [6]**

The most used number of bits is 16, so for making the initial population in this code, we used 16 bits. And as the number of design variables is two, 0 and 1 are used. Bit-string is then made by appending 0 or 1 randomly as much as the length. We will repeat making the bit-string until the specified population number. Figure 2 shows the Python code making the initial population.

****

**Fig 2. Python code function for making the initial population.**

Before the selection step, we need to decode the bit-string to scaled values. For decoding, specific bounds are needed. After decoding the bit-string, each result is put into the fitness function, and evaluate each variable. Figure 3 shows the Python code function for decoding the bit string to scaled values. The function base is made using chat gpt.

****

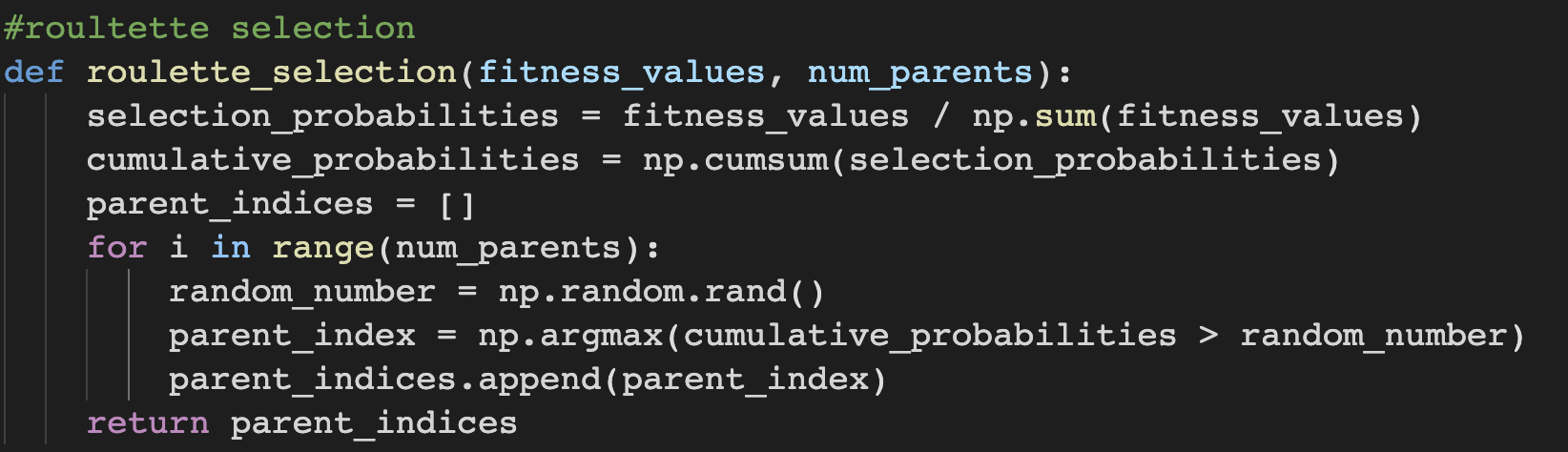
**Fig 3. Python code function for decoding the bit string to scaled values**

**Step 2. Selection**

Selection is the step that selects some individuals to act as parents. It selects the fittest individuals and pass the genes to the next generation.[6] There are 2 kinds of most common selection ways: roulette selection, and tournament selection.

**1)Roulette selection**

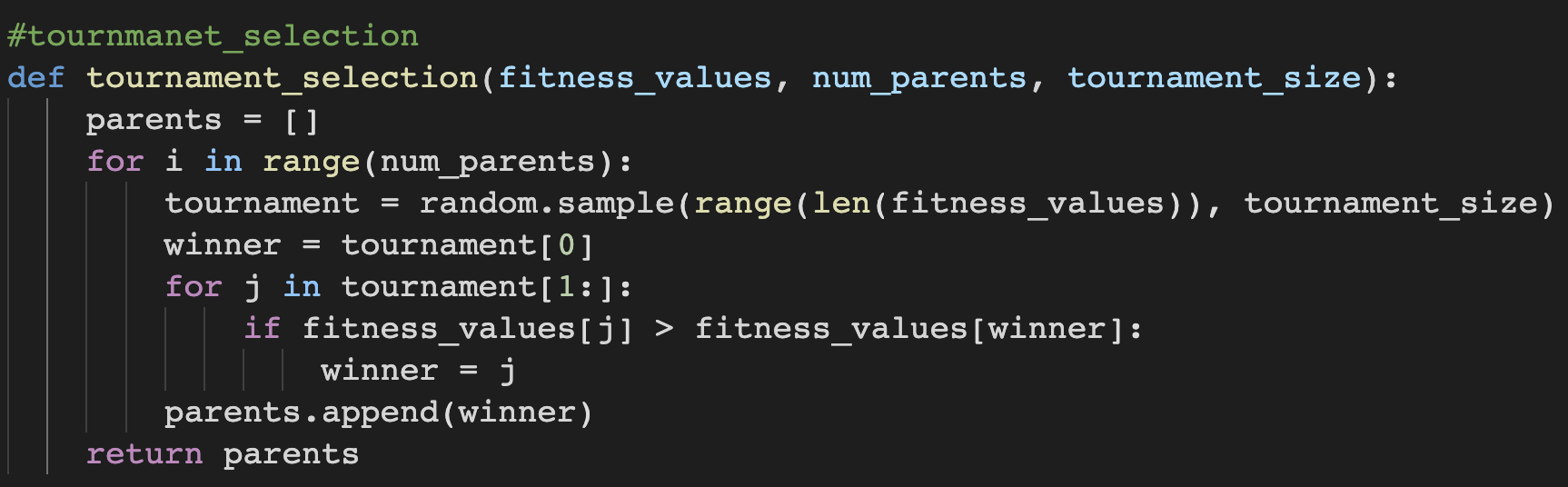
Roulette selection is a stochastic selection method. This means that the probability of an individual is proportional to each fitness. The larger fitness of an individual is, the more likely its selection.[7] Figure 4 shows the Python code function for roulette wheel selection. The function base is made using chat gpt.

****

**Fig 4. Python code function for roulette wheel selection**

**2)Tournament selection**

Tournament selection uses head-to-head competition and selects the winner.[8] The function base is made using chat gpt.

****

**Fig 5. Python code function for tournament selection**

**Step 3. Reproduction**

At the reproduction step, the genetic algorithm leverages two variation operators which are applied to the parent population.

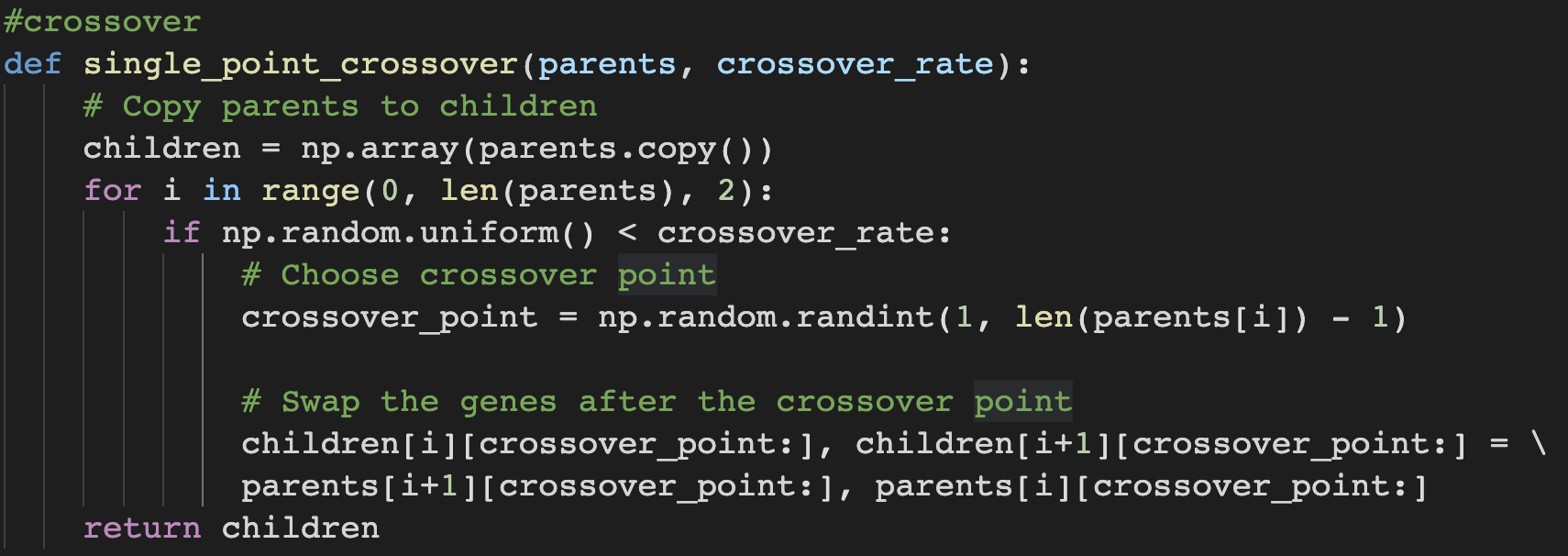
**1)Crossover**

Crossover swaps the genetic information of two selected parents. Figure 6 shows how the crossover works. The point where crossover occurs is called crossover point. This point is chosen randomly.



**Fig 6. Crossover example [6]**

Figure 7 shows the Python code for crossover. Variable ‘crossover\_rate’ is the percent of crossover we are going to make. The maximum rate usually used is around 80 percent. The function base is made using chat gpt.

****

**Fig 7. Python code function for crossover**

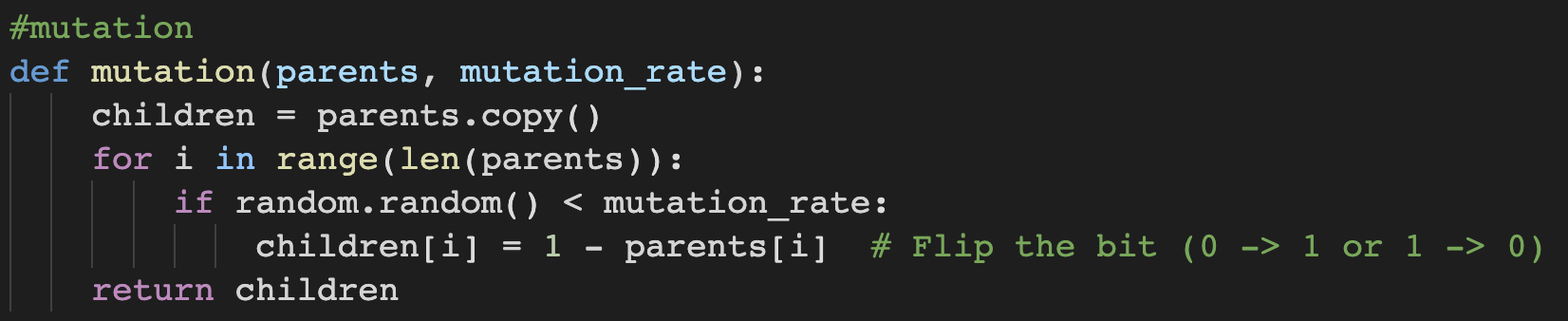
**2)Mutation**

Mution inserts or flips random genes to maintain the diversity and prevent convergence. [6]



**Fig 8. Mutation example [6]**

Figure 9 shows the Python code for crossover. Variable ‘mutation\_rate’ is the percent of mutation we are going to make. The maximum rate usually used is around 10 percent. The function base is made using chat gpt.

****

**Fig 9. Python code function for mutation**

**Step 4. Replacement & Termination**

When the offspring is produced, the population is replaced with the new produced bit-string. The whole population can be replaced, but also we can remain part of the parents which have high fitness rate. This is called elitism. In this report, we brought top 20 parents. The whole step is repeated continuously until the termination is given. Replacement can be terminated in lots of ways. In this report, we designed termination in 2 ways. First, we designed it to terminate when the population does not produce offspring which are significantly different from the previous generation. This means the population is gathered to one point, which is the optimal solution to the fitness function. The second way is by applying the stopping criterion of replacement iteration. Figure 10 shows the Python code of the replacement and termination using the first termination way and Figure 11 shows the Python code using the second termination way.



**Fig 10.Python code of replacement & termination using the first termination way(no significant difference with the previous population)**

****

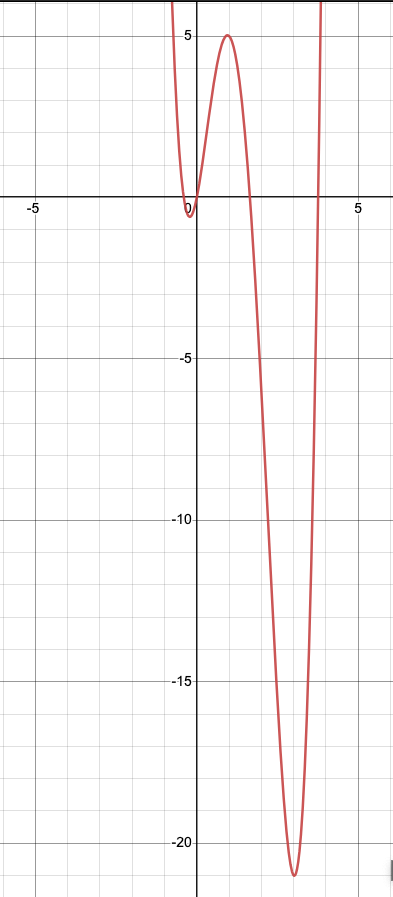
**Fig 11.Python code of replacement & termination using the second termination way(giving iteration limitation)**

**3. Genetic algorithm examples 1\_ Optimizing a continuous function using the genetic algorithm**

**1)Optimization with unconstrained optimization**

Let’s suppose fitness function is:

Figure 12 shows the plot of the function within the range **.** The plot is drawn using desmos program.



**Fig 12. Plot showing the objective function within**

From the plot, we can observe that the global minimum is made when at 3, and a local minimum is made at between 0 and 1.

The input variables for the genetic algorithm were set as the following:

bit=16, number of population=100, crossover rate=0.8, mutation rate=0.1.

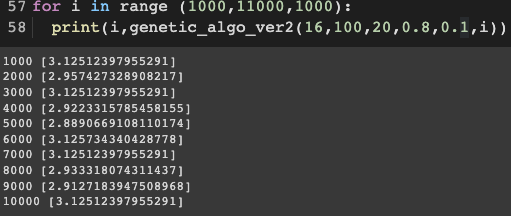
We set the crossover rate and mutation rate to the commonly used maximum value. When we tried it with lower values, there was no significant difference when setting it to the maximum rate. So we chose the commonly used maximum value.

Using the termination way 1, which is the way stopping when there is no significant difference with the previous population, showed different results when trying every run time. However, it still returned  around 2 to 3. Figure 13 shows different results of **.**



**Fig 13. Each runtime difference when using termination way 1**

Using termination way 2, giving iteration limitation, we started the iteration from 1000, and increased it until 10000, giving an increase of 1000. Still, there was difference of the answer at each run time, higher iterations showed the more accurate answer to the global minimum. Figure 14 shows the results of way 2.

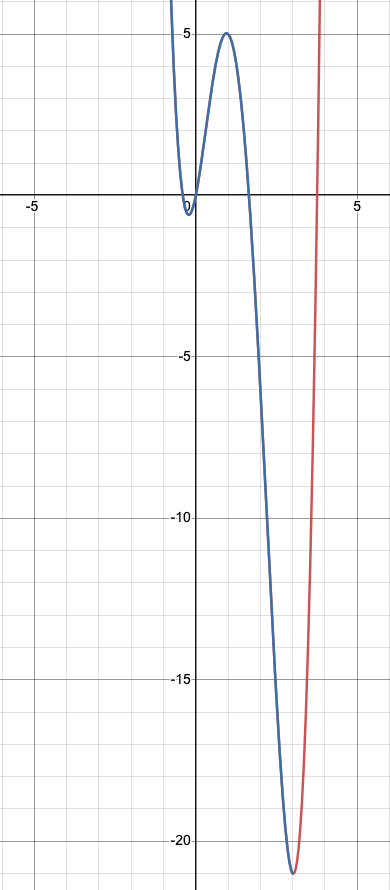


**Fig 14. Increasing the iteration 1000 to 10000 and each result**

From these results, we can conclude that the optimal solution is around 3, and the optimal design variable is crossover rate 0.8, mutation rate 0.1. For termination, we used 2 ways: stopping when no difference with the previous population and giving 10000 iterations.

**2)Optimization with constrained optimization**

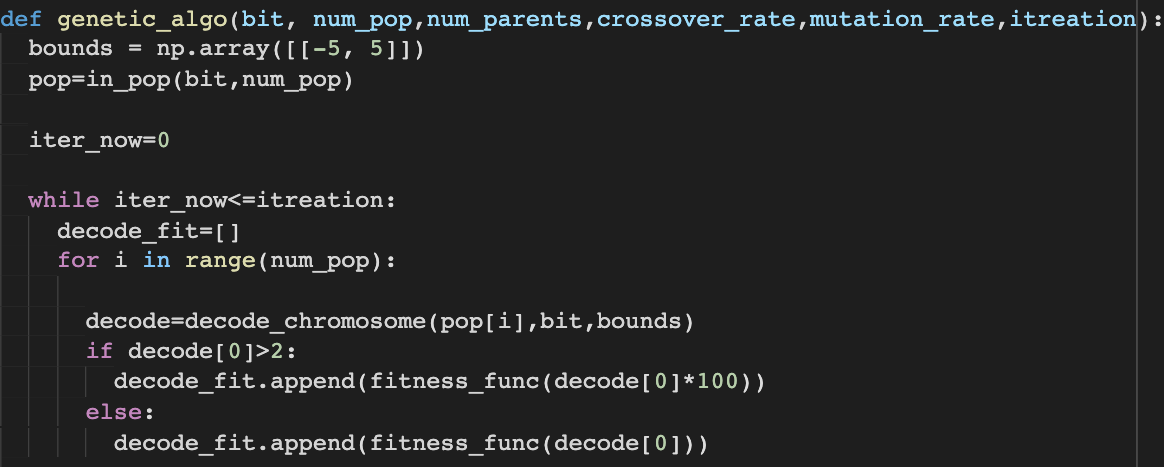
At the previous chapter, the optimization is made at the range of with no constraint. In this chapter, we will give a constraint **,** and observe the difference. Figure 15 shows the plots. The red plot shows the plot within the range **,** while the blue plot shows when **.** The plot is drawn using desmos program.



**Fig 15. Plot showing the objective function within (red)**

**and constrained range (blue)**

For the constraint, we gave a penalty(ex. multiplying 100 by the value) when the fitness function value is bigger than 2. By giving the penalty, the value fitness rank gets lower and the chance of being chosen gets lower too. We will use termination way 2, using iteration, because this way gives a more accurate answer than way 1. Figure 16 shows part of the Python code when a constraint range is given.

**Fig 16. Part of the Python code to regulate the results using penalty when the constraint range given**

Other values are given same with the precious chapter. (bit=16, number of population=100, crossover rate=0.8, mutation rate=0.1) Iteration is given 10000.



**Fig 17. Results of the algorithm when the constraint range is given**

Figure 17 shows the returned results. We got the optimal solution is around 2, and the optimal design variable is crossover rate 0.8, mutation rate 0.1. For termination, we gave 10000 iterations.

From referring result, the returned answers are almost close to 2. This is because when the fitness value is bigger than 2, they get a penalty and their rank gets low, which means the percentage of being selected gets lower. However, referring to Runtime 3, we can see it sometimes shows a different result, which seems to be the local optimum.

**4. Genetic algorithm examples 2\_ Traveling Salesman Problem(TSP)**

**1)Introduction of TSP**

Traveling salesman problem which is called TSP, is made to find the shortest possible route that visits every city. TSP is one of the real-life combinatorial optimization problems that were solved using genetic optimization. It helps in finding an optimal way in a given map with the distance between two points and with the routes to be covered by the salesman.[5] ,[9]

**2) Problem statement**

Suppose we are currently working for the United Parcel Service (UPS) as an operations research analyst. We need to provide the shortest route, which visits each delivery point exactly once from the origin point, to the UPS truck drivers. We are given seven different cities: Atlanta, Boston, Cincinnati, Denver, New York, Philadelphia, and San Francisco. We used all 7 cities for solving the problems, but 3 cities: Atlanta, Boston, and Cincinnati results will be only shown in this report.

**3) Problem solution**

**Step 1. Generating a plot showing delivery locations for selected cities**

Before we start to solve the TSP problem, we need to check the delivery locations for selected cities. Using the plot, we will mark each delivery locations based on their longitude and latitude. Figure 18 shows each location plot.



**Fig 18. Delivery locations for each city**

**Step 2. Constructing distance adjacency matrix of great circle distances between cities**

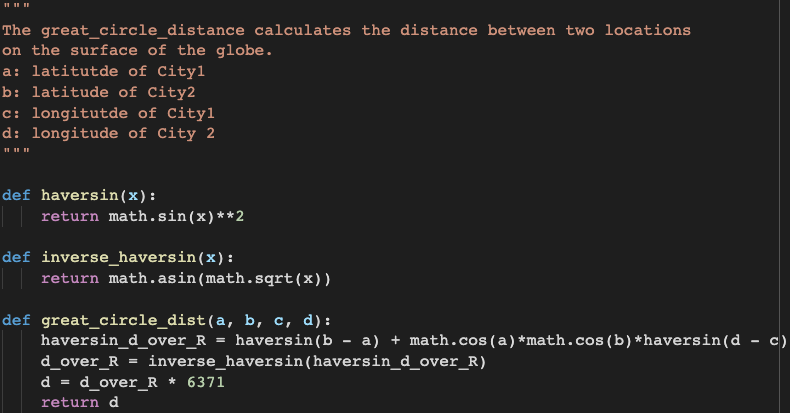
When we calculate the great-circle distance between two points on a sphere when the longitudes and latitudes are given, we use the Haversine formula.

the Haversin formula is as follows:

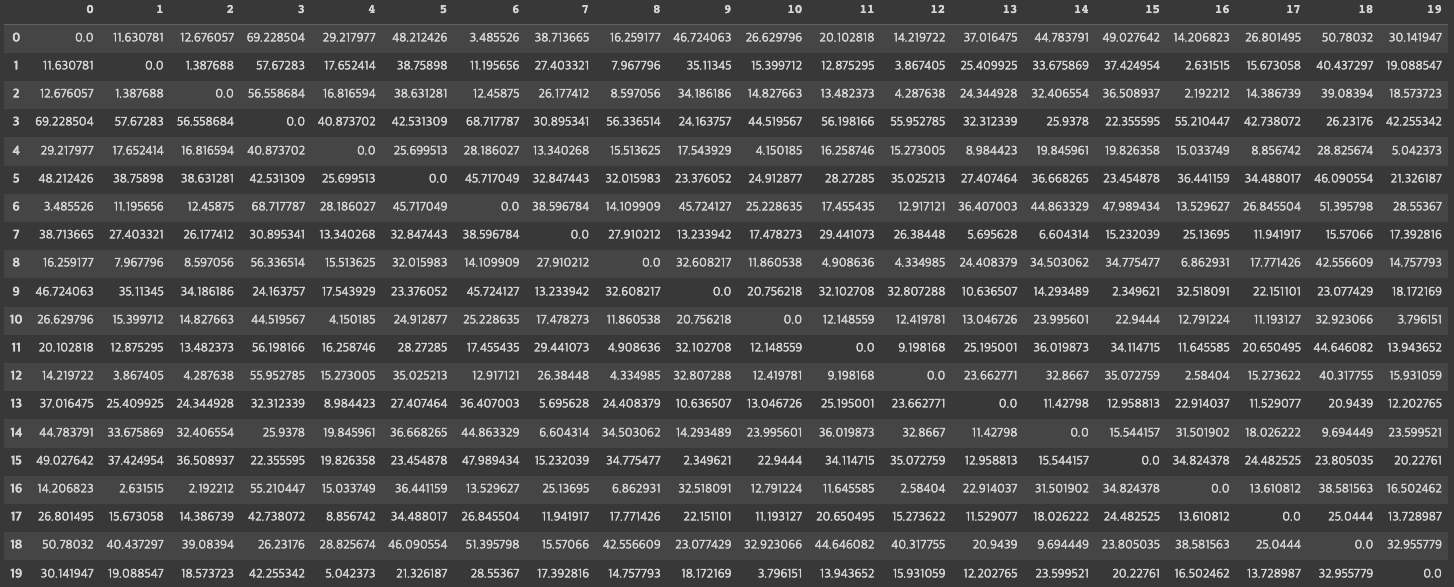
City P and Q have coordinates of () and () respectively.

To calculate the great circle distance, rewrite the equation with respect to d (Great Circle Distance). In other words, multiply the *inverse of haversin* to both sides of the haversin formula, followed by multiplying *R* (radius of earth; approximately 6371km). [10]

Using the Haversine formula, we will calculate the distance of each delivery location and make them into an adjacency matrix. Figure 19 shows the Python code of the Haversine formula and Figure 20 shows an example of the distance adjacency matrix of each location in the city.



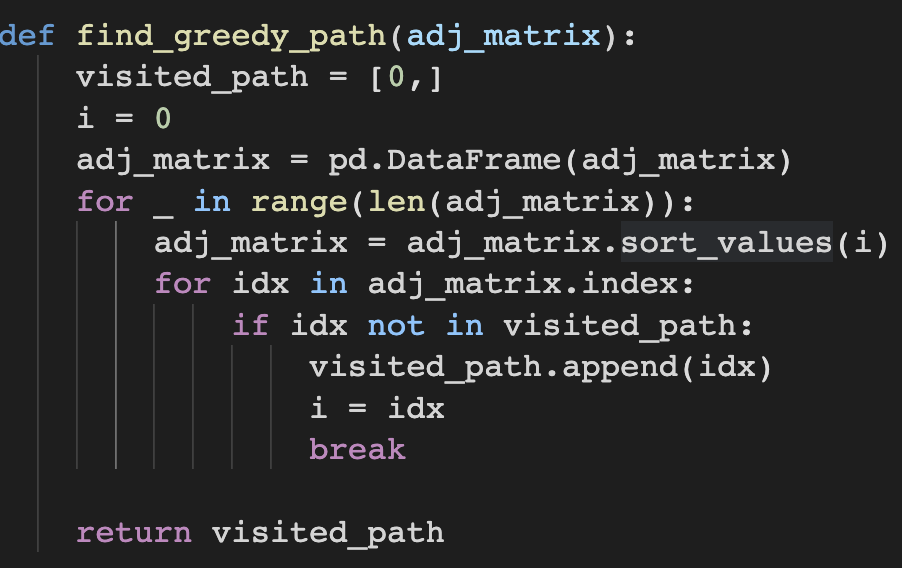
**Fig 19. Python code of Haversine formula**

****

**Fig 20. Example of the distance adjacency matrix of great circle distances between each location**

**Step 3. Finding the initial solution using the greedy algorithm**

Before using the genetic algorithm, we will use the greedy algorithm first to save time of the genetic algorithm. Greedy algorithm selects the best option available at the moment, without consideration of the overall optimal result. [11] When we apply this algorithm, it will choose the closest location from the current location. Figure 21 shows the Python code of greedy algorithm, and Figure 22 shows the initial route made by greedy algorithm.

****

**Fig 21. Greedy algorithm Python code**

**Fig 22. Shortest route generated by the greedy algorithm for each city**

The route generated by the greedy algorithm is :

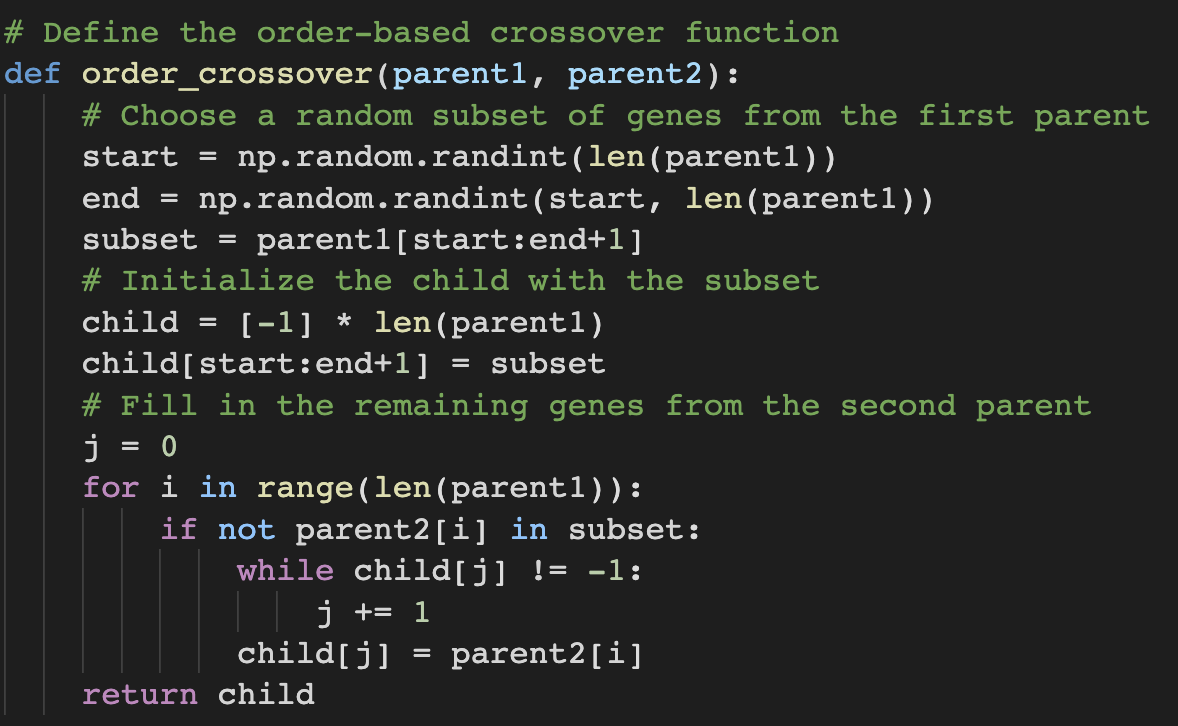
Atlanta: [0, 6, 1, 2, 16, 12, 8, 11, 10, 19, 4, 17, 13, 7, 14, 18, 9, 15, 3, 5]

Boston: [0, 9, 20, 10, 34, 1, 17, 26, 39, 18, 3, 21, 19, 29, 36, 38, 24, 30, 27, 12, 37, 28, 22, 35, 6, 5, 7, 14, 16, 13, 2, 15, 8, 33, 32, 11, 25, 31, 4, 23]

Cincinnati: [0, 2, 1, 8, 4, 5, 7, 6, 3, 9].

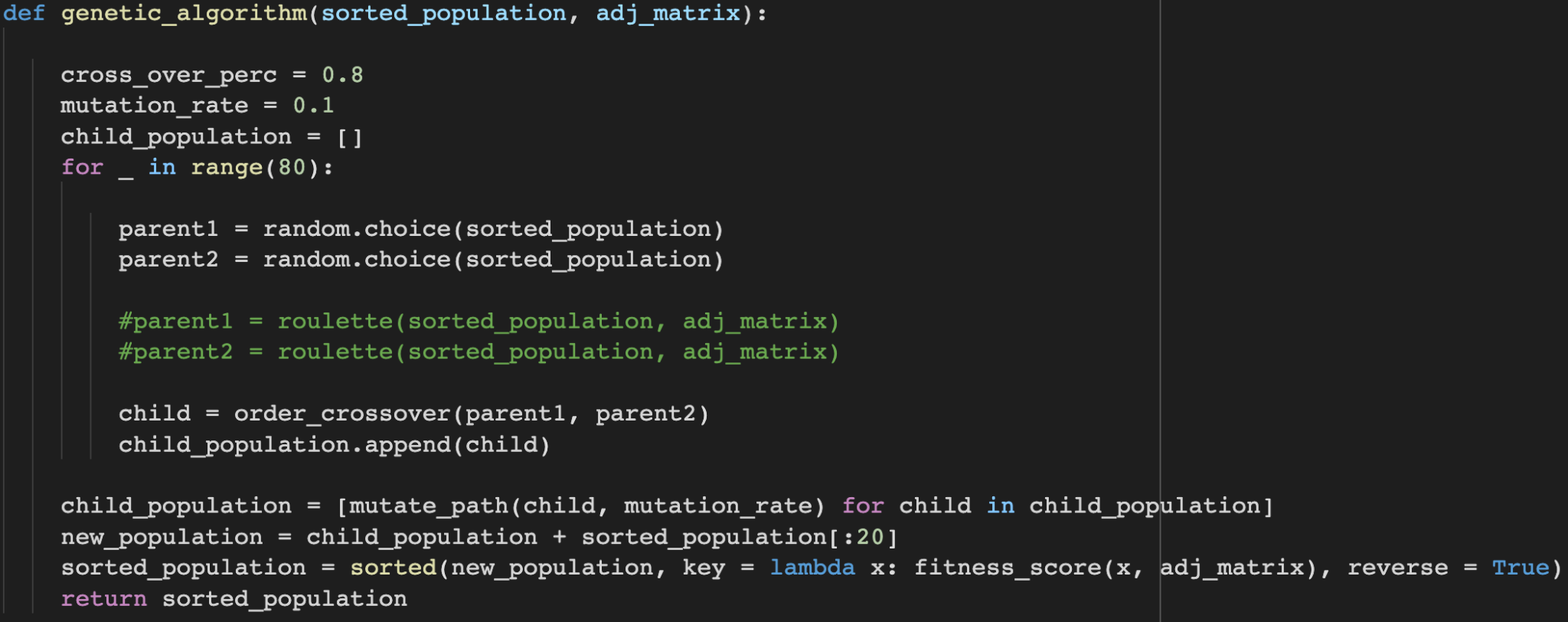
**Step 4. Finding the solution using the genetic algorithm**

Using the genetic algorithm principle explained on the previous chapters, we made a new genetic algorithm function for TSP problem. To give higher efficiency, we used greedy algorithm answer, not from starting at random. We made the initial population by giving mutation to the greedy algorithm answer. One important thing we had to consider is, in the TSP problem there can be no repetition. So when making crossover, we needed to use order-based crossover. Figure 23 shows the order-based crossover function code made by Python.



**Fig 23. Python code function for order-based crossover**

Figure 24 shows the genetic algorithm function applied for the TSP problem. There is an aspect the user must note: the code does not implement the *Roulette Selection* when selecting for a parent sample for crossover. The *Roulette Selection* is an implementation that allows a fitter sample to have a higher chance of selection compared to other samples that have lower fitness. However, we observed that population size is decently large (100 samples in one population) and that the fitness score is not extraordinarily different between samples. Due to these two factors, we discovered that there is no significant difference between selecting the parents using random sampling and the *Roulette Selection*. In addition, we observed that the convergence time for the genetic algorithm is significantly slower when the *Roulette Selection* method is used. Thus, based on these two reasons, in the actual implementation of the code above, we simply used random sampling to select parents for crossover.



**Fig 24. Python code for genetic algorithm function applied to TSP**

The shortest route generated by the genetic algorithm is :

Atlanta: [0, 6, 1, 2, 16, 12, 8, 11, 10, 19, 4, 17, 13, 7, 14, 18, 3, 15, 9, 5]

Boston: [20, 9, 34, 1, 26, 18, 3, 21, 39, 17, 6, 10, 14, 16, 13, 2, 15, 7, 8, 5, 0, 35, 28, 37, 22, 29, 36, 38, 24, 30, 12, 27, 19, 25, 33, 32, 11, 31, 4, 23]

Cincinnati: [9, 2, 0, 1, 8, 4, 6, 5, 7, 3]

At figure 25, the plot shows the shortest path route generated by the genetic algorithm.

****

**Fig 25. Shortest route generated by the genetic algorithm for each city**

The output of the greedy algorithm and the genetic algorithm shows a significant difference in path traveled. This is largely because the greedy algorithm simply considers the next nearest destination respective to its current city when looking for the shortest path. The Greedy Algorithm’s strategy may seem like a reasonable one but sometimes, taking the longer route could possibly pay you off much more in the long run. For example, consider cheating in a classroom exam. Getting a good score by cheating will give you short term payoffs but in the run, it won’t do you much good. However, a diligent student who keeps himself away from underhanded tricks, will get himself further in life. This analogy is a bit of an extreme example, but the key idea is the same. The Greedy Algorithm’s strategy is like a student who cheats in exams. It will pay off in the short run, but since it won’t consider all possibilities, it has a higher chance of missing an optimal solution. On the other hand, the Genetic Algorithm considers multiple options from its population. In its population, some solutions are worse than others and some, better. Yet, by exploring a wider range of possibilities, it has a higher chance of encountering the optimal solution by taking detours and exploring a wider search space, looking for paths that pays off in the long run.

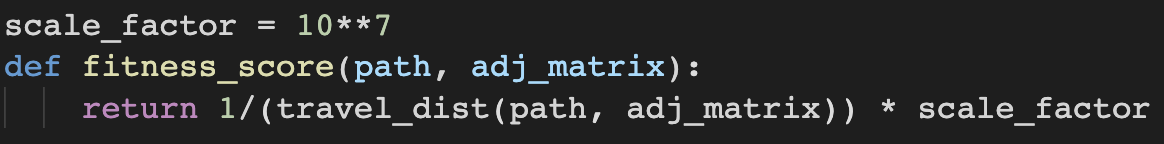
To check if the optimum value generated by the genetic algorithm is converged as the number of iterations increases, we made the convergence history plot.

The fitness score function is based on the total distance of the path traveled.

Since the fitness score should increase as the total distance traveled decreases, we set the fitness score function as the inverse of the travel distance which is then multiplied by an appropriate scaling factor. The scaling factor prevents the fitness score from becoming too large or too small. The default scale factor is set to .

Figure 26 shows the Python code for calculating the fitness score of a given path, and Figure 27 shows the convergence history plot of the fitness score.

We can see the fitness score increases as the iteration increases, which means we found the optimal pathway for each city using the genetic algorithm.

****

**Fig 26. Python code for calculating the fitness score of each iteration**

**Fig 27. Convergence history plot of fitness score**

**4) Results discussion**

**(Q1)Do you think that the solution from the genetic algorithm is a global optimum or not?**

It is hard to confirm that the solution is the global optimum. We need to check every path combination to be sure that our output is the global optimum. However, the genetic algorithm does not consider the whole combinations. Therefore, although the solution from the genetic algorithm might be the global optimum, we can not be sure that it is the global optimum.

**(Q2)Do you think that you will always get the same solution (e.g., the total travel distance is always identical) when you execute the algorithm multiple times?**

When we execute the algorithm multiple times, we can not confirm that we will get the same results each time. This is due to the aspect of the randomness of the genetic algorithm. There are three steps in the algorithm that is highly dependent on randomness: Initial population generation, parent selection, and reproduction. Therefore, it is difficult to be confident that we will get the same output every execution. This is also reflected in the fact that different x values were outputted in the ‘Genetic Algorithm Examples 1’ in page number 9.

**(Q3)Suppose that your algorithm is performing slowly for some reasons. From the perspective of designing the algorithm, how would you be able to make it more computationally efficient?**

It is possible to make the algorithm computationally more efficient by utilizing an initial population created from the greedy path. By making random mutations in the greedy path through swapping two random numbers in the list, we can generate a population derived from the greedy path. By using a sub-optimal path as the initial population, there is a higher chance that the genetic algorithm converges faster to the optimal solutions, since the sub-optimal path has a higher chance of being closer to the global optimum than a randomly generated path.

**Citation**

[1] Brownlee, J. (2021, October 11). *Simple genetic algorithm from scratch in Python*. MachineLearningMastery.com. Retrieved April 7, 2023, from https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/

[2]AlgoDaily. (n.d.). *Daily coding interview questions. full programming interview prep course and software career coaching.* AlgoDaily. Retrieved April 7, 2023, from https://algodaily.com/lessons/introduction-to-genetic-algorithms-in-python

[3]Team, T. A. I. E. (2021, April 2). *Genetic algorithm (GA) introduction with example code*. Medium. Retrieved April 7, 2023, from https://pub.towardsai.net/genetic-algorithm-ga-introduction-with-example-code-e59f9bc58eaf

[4]GeeksforGeeks. (2022, September 29). *Genetic algorithms*. GeeksforGeeks. Retrieved April 7, 2023, from https://www.geeksforgeeks.org/genetic-algorithms/

[5]Chaudhary, S. (2022, November 18). *Genetic algorithm applications in machine learning*. Genetic Algorithm Applications in Machine Learning. Retrieved April 7, 2023, from https://www.turing.com/kb/genetic-algorithm-applications-in-ml

[6]Mallawaarachchi, V. (2020, March 1). *Introduction to genetic algorithms - including example code*. Medium. Retrieved April 9, 2023, from https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[7]Luca, W. by: G. D. (2022, November 11). *Roulette selection in genetic algorithms*. Baeldung on Computer Science. Retrieved April 9, 2023, from https://www.baeldung.com/cs/genetic-algorithms-roulette-selection

[8]GeeksforGeeks. (2022, July 11). *Tournament selection (GA)*. GeeksforGeeks. Retrieved April 9, 2023, from https://www.geeksforgeeks.org/tournament-selection-ga/

[9]GeeksforGeeks. (2023, February 6). *Travelling salesman problem using Dynamic Programming*. GeeksforGeeks. Retrieved April 8, 2023, from https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/

[10]*The Great Circle Distance Add to Your Resource Collection remove from your resource collection add notes to this resource view your notes for this resource*. The great circle distance | Trigonometry: Compound Angles | Underground Mathematics. (n.d.). Retrieved April 9, 2023, from https://undergroundmathematics.org/trigonometry-compound-angles/the-great-circle-distance

[11]*Greedy algorithm*. Programiz. (n.d.). Retrieved April 9, 2023, from https://www.programiz.com/dsa/greedy-algorithm