Genetic Algorithms

Asia Morgenstern

December 14, 2023

Abstract

Genetic algorithms are a particularly useful optimization method. Although often used to search a neural network, we implement a genetic algorithm in Python for three different applications: a method to match a random string to a target string, a method to solve the transcendental equation $\cos x = x$, and a method to find the roots of a polynomial-like equation (provided that there exists at least one real, non-negative root). By running each implementation for 1000 runs, we find that the average numbers of generations the algorithms vary greatly. For the target string algorithm, it takes 346.7 generations to converge on average. For the transcendental equation algorithm and the root finding algorithm, it takes significantly fewer generations to converge on average: 4.0 and 5.2 generations, respectively.

1 Introduction & Background

Genetic algorithms (GAs) are a subclass of evolutionary algorithm that is based on the concept of natural selection, similar to how simulated annealing is based on the annealing process in metallurgy. Also like simulated annealing, GAs are used to find global extrema. However, unlike with simulated annealing, GAs assign a fitness score, which determines how able an individual is able to compete. Individuals with "better" fitness scores are then allowed to mate and produce "better" offspring, while individuals with "worse" fitness scores die off. The goal therefore is that new generations will have on average more genes that are "better" than the best individual of the previous generation. In other words, partial solutions should improve after each generation until it converges to the "best" solution [1], [2].

In order to better understand how GAs work, we must first define some terminology. The search space of a GA is the set of all possible results. The population is a subset of the search space with the size of which remaining constant. Each individual of the population is uniquely identified by its chromosome, which represents a single possible result. Each chromosome is made up of genes. (Examples of each can be found in Section 2.)

Implementing a GA is then straightforward. To start, we first initialize a population p. Then, we determine the fitness for each individual in the population using some predefined cost function $c(x)$. Finally, we generate new generations until the algorithm converges. To generate a new population p', we apply the following steps: elitism, crossover, mutation, and sorting based on fitness [1].

The idea behind elitism is that we want to give the best individuals preference, allowing them to pass their genes down to the next generation. Combining the crossover and mutation steps, we have a mating process that mirrors "survival of the fittest," in that we only allow an upper percentage of the population to mate and to pass their combined genes to the next generation. Ultimately, including these three steps is meant to ensure that future generations are on average "better" than past generations [1].

Based on these ideas, we can simulate a similar process. Elitism occurs when we select an elite percentage of population p to pass on their chromosomes to the next generation p'. In the mating process, we randomly select two individuals from population p, which we call Parent 1 and Parent 2. Then, we iterate through their genes, choosing either the ith gene from Parent 1, the ith gene from Parent 2, or a random gene. Doing so includes the effect of mutations and thereby minimizes the chances of the GA getting stuck in local extrema instead of finding the global extrema. The resulting chromosome gives us a new individual in population p', which we call a child. After repeating the mating process until the size of p' is equal to that of p, we apply the same cost function $c(x)$ to sort the population p' from lowest fitness to highest fitness [1].

We choose to order the population in this manner since we want to minimize the fitness score. Sorting the population means we only have to check just the individual at index 0. Thus, we say the algorithm converges if the minimum fitness is either equal to 0 or within ε of 0. The choice of the convergence criterion is based on whether the GA allows for definitive convergence. In some implementations, it is not possible to get a fitness of exactly 0, thus the need for an ε-based convergence criterion. Such an instance occurs when we convert floating-point numbers to binary; for irrational numbers, we cannot get an exact binary representation that has a finite number of decimal places.

Despite this limitation, GAs are still particularly useful due to how robust they are. GAs work regardless of the size of the search space, or inputs. Furthermore, if we vary two inputs slightly, similar to how we would vary initial conditions when testing for chaos in a system, the GA should still converge to the same value. Moreover, the GA will still work even if the input has some noise. In addition to being robust over varied inputs, GAs are also robust over different types of optimization problems. In other words, GAs will work on cost functions that are time-dependent, independent of time, linear, non-linear, continuous, discontinuous, or that contain noise [1], [2].

However, GAs are not without their drawbacks. They are extremely dependent on the selected cost function and the size of the population. They are also dependent on the percentage of the population that we consider to be the elite, the percentage of the population we consider to be the fittest, and the likelihood of a mutation occurring. If any of these five values are suboptimal choices, the GA could either take much longer to converge (both in runtime and in number of generations) or it could yield unhelpful results [2].

Despite these disadvantages, GAs have a wide range of applications; many of which are outside of the scope of this project. GAs are often used with neural networks due to their ability to work on large search spaces. Additionally, GAs can be applied to problems of graph coloring and pattern recognition. They also work on both discrete and continuous systems. Furthermore, they even work with optimizing financial markets [1], [2].

2 Methods

In our implementations, we focus on three different applications; the first of which is to match to a target string. We choose this application because code was provided in *Genetic Algorithms* [1]. As such, we can write our own GA using the provided code as an outline and as a way to assist with debugging potential errors. The second application is to solve the transcendental equation $\cos x = x$, which we choose because, even though the search space is the real number line, the implementation is not that much more complicated than the former. Finally, the third application is to find the roots of polynomial-like equations. We choose this application because it is a generalization for the second.

2.1 Match to a Target String

To set up our GA, we must determine how we will represent a population, an individual, an individual's chromosome, and a single gene in a chromosome. We represent a gene as a single character, or a string of length 1. Additionally, we will restrict all possible genes to a `GENES` constant, which is a string of all valid characters. To represent a chromosome, we will use a list, converting each string to a list using `list(chromosome)`. Doing so improves efficiency since strings are immutable. To represent an individual, we will create an `Individual` class, where each `Individual` object has a chromosome property, which stores the string as a list, and a fitness property, which stores the fitness score as either an integer or floating-point number. Finally, we represent a population as a list that stores `Individuals`.

Now that we have the structure of the GA defined, we need to set our initial constants. First, we select the size of the population `POPULATION_SIZE`, which we arbitrarily define to be 100. Next, we select the valid genes for the `GENES` constant, which we define to be the upper- and lowercase letters, the numbers, and all other special characters on a standard computer keyboard. Finally, we define our target string `TARGET` to be `"I love Geeks for Geeks!"`, which is the same target string as in *Genetic Algorithms* [1].

To implement the GA in Python, we start by initializing our population. To do so, we generate `POPULATION_SIZE = 100 Individuals` and append them to the list `population`. To generate an `Individual`, we first generate a chromosome and then calculate the fitness score of that chromosome. To generate a chromosome, we create a list `chromosome`. Then, using `random.choice(GENES)`, we iteratively append a random gene from `GENES` to `chromosome` until `len(chromosome)`, the length of the list, is equal to `len(TARGET)`, the length of the target string.

To calculate the fitness score of the chromosome, we iterate through the list and the target string, comparing the character at index i in the list with the character at index i in the target string. If the characters are not the same, then we increment the fitness of the chromosome. Thus, using this calculation as the cost function $c(x)$, if there exists a fitness score with $c(x) = 0$, then the chromosome as a string is an exact match to the target string.

Once the population is filled, we sort the population in order of increasing fitness. To do so, we make use of the `sorted()` function. Since we to sort on fitness and not alphabetically, we must include as an argument the custom key function `key = lambda x : x.fitness`.

Now, we can apply the optimization portion of the GA. First, we check if the algorithm has converged. Since nothing hinders finding the string that is an exact match to the target

string, the algorithm should converge definitively. Furthermore, since we make no approximations in evaluating the cost function, there should not be any errors in an individual's fitness score. As such, we can get a fitness score of exactly 0. Therefore, our GA can indeed converge definitively. Thus, our convergence criterion is when the minimum fitness score of the population is exactly 0.

If the algorithm has not converged, we then create the next generation, another population. To start, we create a list `new_generation`. Then, we want to perform elitism and the mating process. To perform elitism, we must define what we want the elite percentage `elite_percent` to be. We need to choose a value that allows for some of the best individuals to pass their chromosomes down to the next generation without copying so many individuals that we have too many suboptimal individuals showing up in multiple generations. Thus, based on the value used in *Genetic Algorithms* [1], we choose `elite_percent` to be the top 10% of the population. Using this value, we determine the number of the elite `n_elite = int(elite_percent*POPULATION_SIZE)`. We then select the `n_elite Individuals` from `population` and append them to `new_generation`.

To perform the mating process, we must decide what is considered the top percentage of the population `top_percent`. Just as with determining what the elite percentage should be, we need to consider that we want only the best of the population to pair off and pass their crossed genes down to the next generation. We choose `top_percent` to be the top 50% of the population, again based on the value used in *Genetic Algorithms* [1]. Then, we have that the number of the top percentage is `n_top = int(top_percent*POPULATION_SIZE)`.

Next, we want to choose two parents to mate. To do so, we randomly select two parents from the top 50% of `population` using `random.choice(population[:n_top])`. Using the `mate()` function, we create a child, which we then append to `new_generation`. Since the size of the population must stay constant, we need to produce `POPULATION_SIZE - n_elite` new offspring.

In our `mate()` function, we must determine the likelihood that a gene mutates; call the probability of a mutation `mutation_percent`. We must consider that if the percentage that the gene mutates is too low, the GA is more likely to get stuck in a local extremum. However, if the percentage is too high, the GA will take much longer to converge, as each generation will not necessarily be on average "better" than the previous generation. We choose `mutation_percent` to be 10%. The reason for this choice is twofold: that is the value used in *Genetic Algorithms* [1], and based on a few tests, a 10% mutation probability led to quicker convergence.

Then, the remaining percentage is divided evenly between Parent 1 and Parent 2. Therefore, for each gene in the chromosome, we will use the gene from Parent 1 45% of the time, the gene from Parent 2 45% of the time, and a mutated gene 10% of the time. Since we cannot directly tell the computer to pick an option with a certain percentage, we will generate a random variable `r = random.random()`. Let `p1_percent = p2_percent/2` with `p2_percent = 1 - mutation_percent`. Then, if `r < p1_percent`, we will select the ith gene from Parent 1. If `r` is not less than `p1_percent` but `r < p2_percent`, we will select the ith gene from Parent 2. Otherwise, we will choose a mutated gene using `mutated_gene()`, which returns a random gene from `GENES`, again making use of the `random.choice()` function.

We repeat this process until the GA converges. At such a point, after n generations,

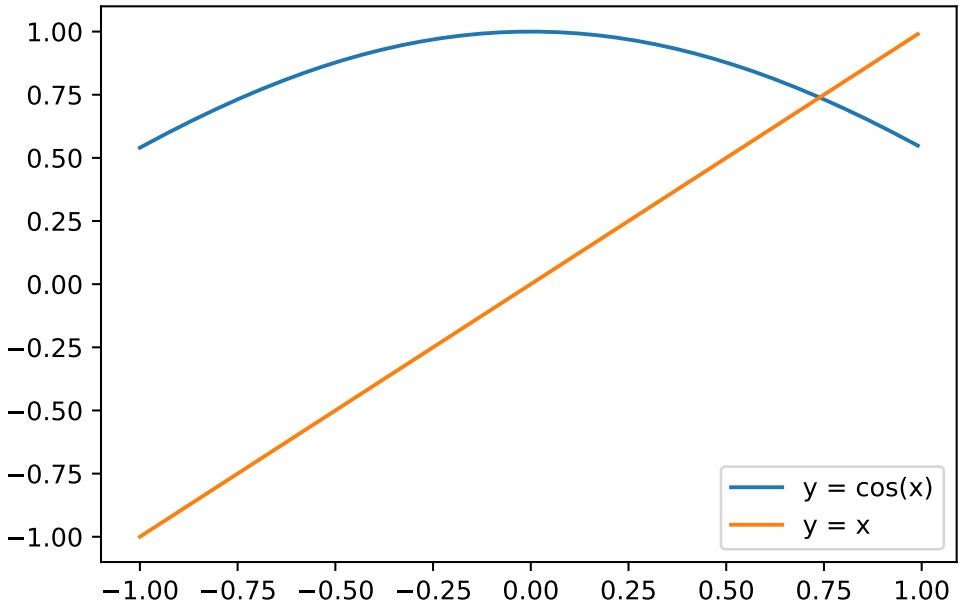

Figure 1: Notice that when $x \in [-1, 0)$, $\cos x > 0$.

`TARGET` should be the same as the string representation of the chromosome of the individual with lowest fitness score, `population[0].chromosome`.

2.2 Solve $\cos x = x$

Transcendental equations are equations that include polynomials, trigonometric functions, exponential functions, and logarithmic functions. We solve for $\cos x = x$. The implementation to solve this transcendental equation is very similar to that of matching to a target string.

In the set up, we no longer need a `GENES` constant. However, we make use of a `MAX_VAL` constant with an implied `MIN_VAL` constant equal to 0. This is because $\cos x$ is non-negative when x is between -1 and 0; see Fig. 1. When x is strictly less than -1, $\cos x$ is always greater than or equal to -1. Thus, the intersection of $y = \cos x$ and $y = x$ must be when $x \leq 0$; see Fig. 2.

These constants restrict the search space down from the entire real number line to just the range [0, `MAX_VAL`). Since we expect that $\cos x = x$ is true when $x \approx 0.739 \leq \frac{\pi}{4} \approx 0.785$ (see Fig. 3), we choose to bound the search above by `MAX_VAL = math.pi/4`. To generate a random variable in that range, we use `random.random()*MAX_VAL`.

Since we want to change genes in our chromosomes, we cannot strictly use floating-point values as chromosomes. Instead, we will convert our floating-point values to binary and use their corresponding binary strings as our chromosome with each bit value representing a single gene. To convert a floating-point number to binary, we utilize the ideas given in the

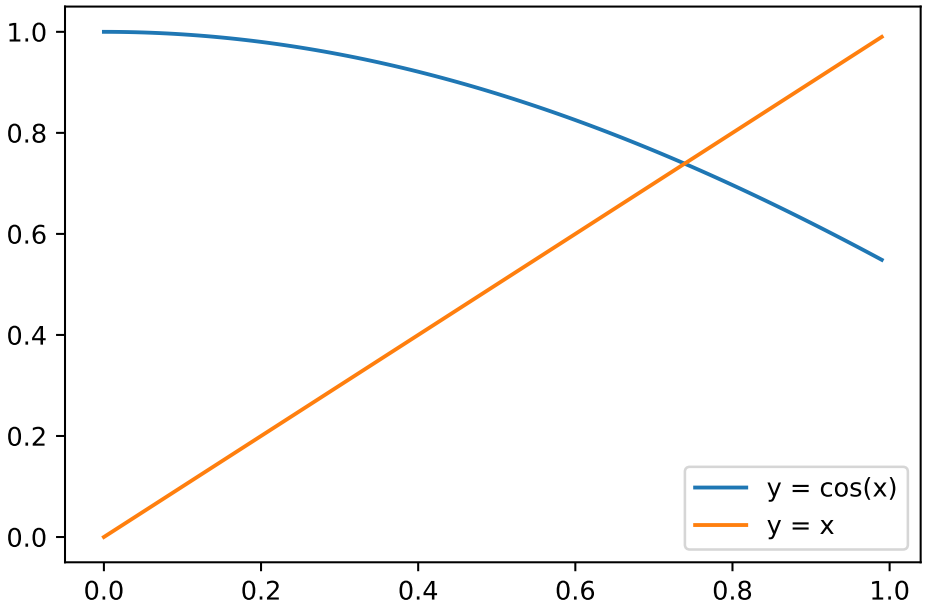

Figure 2: Notice that when $x \in [-1, 0)$, $\cos x > 0$.

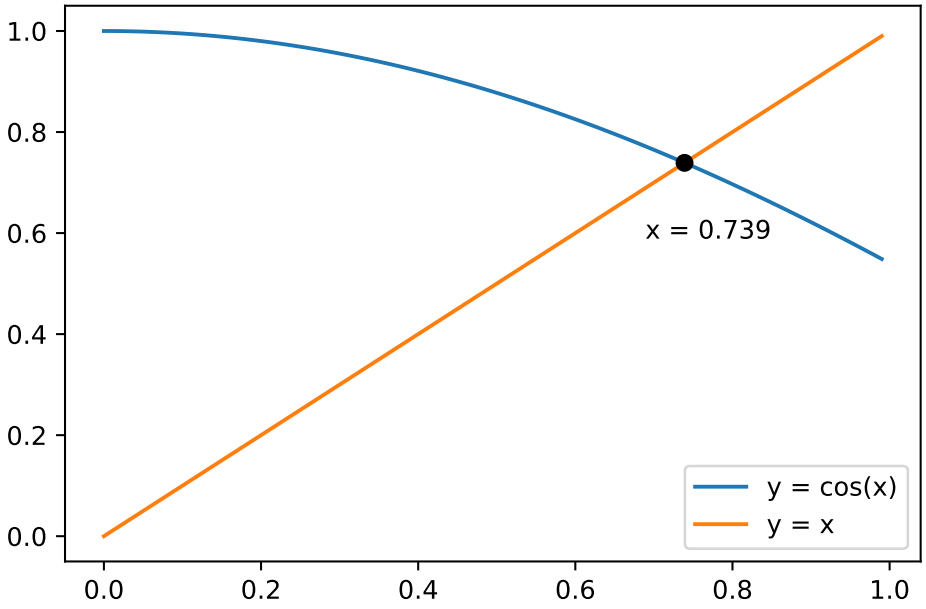

Figure 3: The functions $y = \cos x$ and $y = x$ intersect when $x \approx 0.739$.

Stack Overflow forum posts in *How to convert float number to Binary?* [3]. To summarize the majority of these posts, we will convert the integral portion first as normal. To convert the fractional portion, we will iteratively multiply m by 2, append the integer portion of $2m$ to the binary representation, and subtract 1 if $2m > 1$. This process will repeat until it converges when $2m = 1$.

Since we cannot computationally allow for an infinite number of decimal points, we make use of a `PRECISION` constant, which cuts off the binary representation after so many decimal places, effectively controlling how precise the binary representation will be. We choose to use `PRECISION = 10` since a precision of 10 decimal places leads to a relative error of about 0.1%. As a consequence of our choice, the maximum value we can represent is $\sum_{n=1}^{10} \frac{1}{2^n} \approx 0.999023$. Note that even though $0.999 > \frac{\pi}{4}$, we are not able to represent every real number in the range [0, `MAX_VAL`) since a finite binary representation can only represent to rational numbers but not the irrational numbers.

To implement this in Python, we write a function `float_to_bin(n, precision)`, which takes in either an integer or floating-point number for `n` and an integer for `precision` and returns the binary representation for `n` to `precision` decimal places. If `n` is an integer, we will first convert it to a floating-point number. Then, we convert the floating-point number into a string, which we split into a whole number and decimal portion using `whole, decimal = n.split(".")`.

To convert the whole number portion, we convert `whole` into an integer. Then, we use the `bin()` function paired with the `str()` function, which results in the binary representation of `n` but with a `"0x"` prefix. To remove it, we simply slice off the first two indices of the string and add the remaining string to `bin_rep`, a string that stores the binary representation of `n`, using the following line of code: `bin_rep = whole[2:]`.

To convert the decimal portion, we must first add `"0."` to the beginning of `decimal` so that we have a decimal rather than an integer. Then, we convert `decimal` into a floating-point number. After adding a decimal point to the end of `bin_rep`, we can begin converting the decimal portion to binary. Since we want to calculate the binary representation to the predefined precision `precision`, we make use of a `for` loop, iterating from 0 to `precision`. In the body of the loop, we first double `decimal`. Then, we add the integer portion to the binary representation, `bin_rep += str(int(decimal))`. Finally, we subtract 1 if `decimal` is greater than or equal to 1. Thus, the `float_to_bin()` function will return the binary representation of a floating-point number `n` with a predefined degree of precision, including any necessary zeros for padding.

Once we have generated a random variable in the range [0, `MAX_VAL`), we can convert it to a binary string. Then, to store it as a chromosome, we simply cast the string to a list using the `list()` function. However, whereas generating a chromosome requires converting a floating-point number to binary, calculating the fitness requires converting binary back to a floating-point number. To calculate the fitness, we use $c(x) = |\cos x - x|$ since we want to find where $\cos x = x$. Since we want $c(x) = 0$, we again seek to minimize our cost function.

To convert to a floating-point number, we create a function `bin_to_float(bin_rep)` that takes in either a string or a list for `bin_rep`. If `bin_rep` is a list, we will convert it to a string using `"".join(bin_rep)`. We again split the binary string into whole number and decimal components using the `whole, decimal = bin_rep.split(".")` function. Then, we initialize a variable `ft_num` to calculate the floating-point number equivalent.

Additionally, we initialize an `exponent` variable to 0, which determines by which power of 2 we multiply. For the binary whole, we iterate backwards through the string using a `for` loop with `range(len(whole) - 1, -1 , -1)` and multiply the digit at index i by `2**exponent`, adding 1 to `exponent` after each iteration. For the binary decimal portion, we reset `exponent` to -1. Then, we iterate forwards through the string using a `for` loop with `range(0, len(decimal), 1)` and multiply the digit at index i by `2**exponent`, subtracting 1 from `exponent` after each iteration.

From here, implementing the rest of the GA is nearly identical to the GA when matching to a target string in Section 2.1. Only two minor things change: the convergence criterion and how genes mutate. Since the binary representation has an absolute error of approximately 0.001, we cannot use a definitive convergence criterion. Thus, we say the algorithm converges when the minimum fitness is less than ε, where ε is the absolute error of the binary representation $\varepsilon = 0.001$. Since we are using binary strings instead of strings of random characters, we need to modify our `mutated_gene()` function to only return 0's or 1's. To do so, we use `random.randint(0, 1)`. Finally, since our binary strings represent floating-point numbers and not integers, if the ith gene in the chromosome from Parent 1 is the decimal point `"."`, then we just append it to our child chromosome and `continue` through the loop.

Just as with the first implementation, we generate new generations until the GA converges. At such a point, after n generations, the value represented by the chromosome of the individual with lowest fitness score, `population[0].chromosome`, should be the x value where $|\cos x - x|$ is minimized. Note that while the GA will stop once the minimum fitness is within ε, there exists cases when the fitness is exactly 0, but there is no guarantee of that occurring.

2.3 Find Roots of a Polynomial-Like Equation

We define a "polynomial-like" equation to be a polynomial equation that may include one or more trigonometric functions. We focus on the function $f(x) = (x-10)^3 - \cos x + \sin x$, which has three real roots; see Fig. 4. Since the algorithm can only search for one root at a time, we will search for the root located at $x \approx 11.002$; see Fig. 5. Note that not only does this algorithm rely on the existence of at least one real root, but it also relies on the root being non-negative.

Implementing this algorithm is even more similar to the transcendental algorithm (see Section 2.2) than that algorithm was to the target string algorithm (see Section 2.1). Since the algorithm relies on a polynomial-like equation $f(x)$, we must define equation as the function `f(x)`, which returns the value of our function $f(x)$. Like before, we use a `MAX_VAL` constant. However, we also explicitly use a `MIN_VAL constant`. Not only do they serve to restrict the search space, they also bracket the root we want to find. As such, these values are dependent on both the function and the particular root we seek to find. Therefore, some previous understanding of how the function behaves is required for the algorithm to converge properly. Thus, to generate a random variable in the range [`MIN_VAL`, `MAX_VAL`), we use `random.random()*(MAX_VAL - MIN_VAL) + MIN_VAL`.

Again, we will convert that random variable to binary. However, since it is possible that the lengths of integer portion of the binary representation can vary, we will pad the front with 0's in the `float_to_bin()` function. Since we assume that `MIN_VAL` is non-

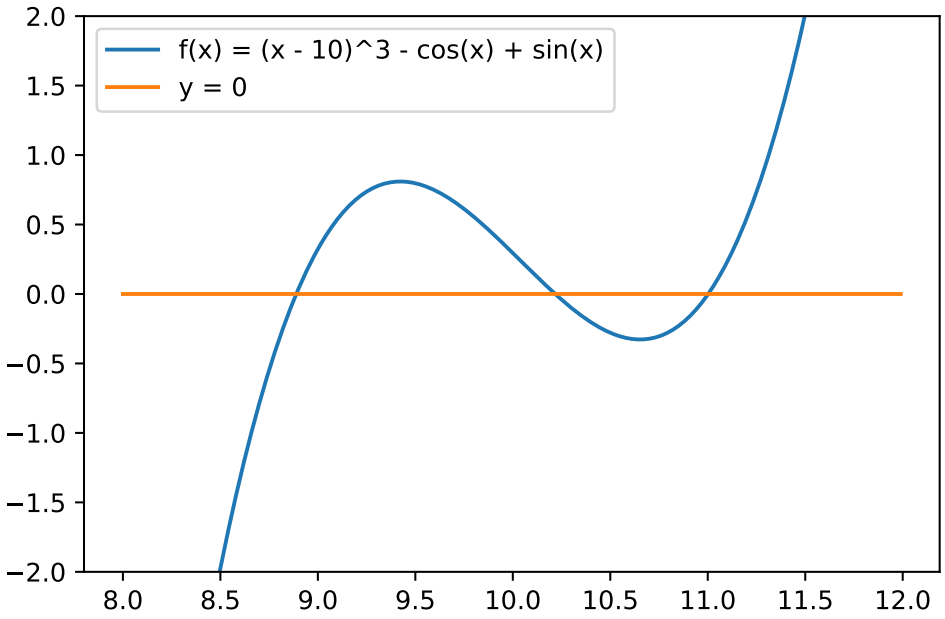

Figure 4: The function $f(x) = (x-10)^3 - \cos x + \sin x$ has 3 real roots.

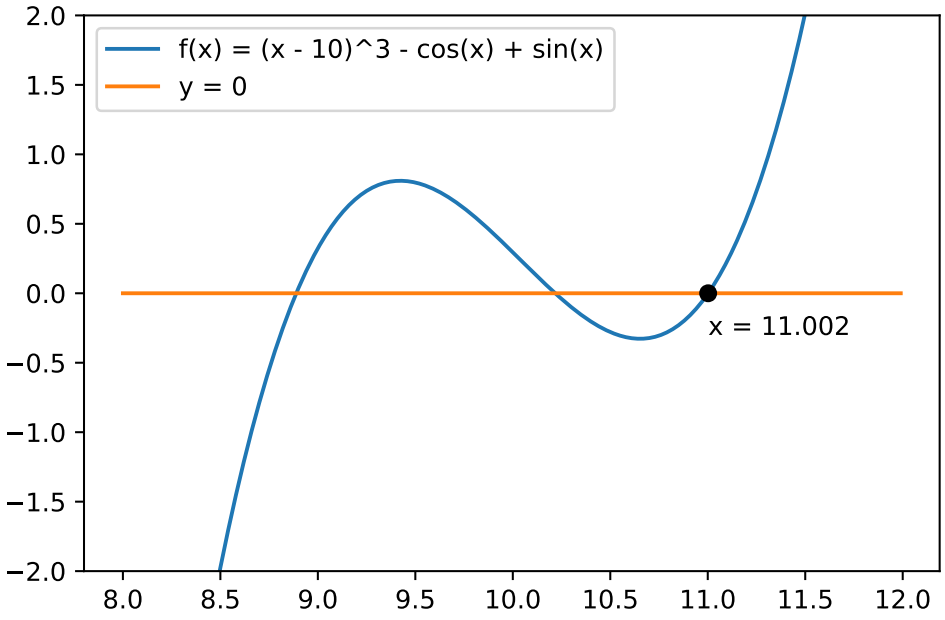

Figure 5: The function $f(x) = (x-10)^3 - \cos x + \sin x$ has a root at $x \approx 11.002$.

negative, `int(MAX_VAL)` must have the longest binary representation. Therefore, if, after converting the whole number portion to a binary string, the length of `bin_rep` is less the length of the binary representation of `int(MAX_VAL)`, then we will add `"0"` to the front using `bin_rep = "0" + bin_rep` until the strings are the same length.

For the cost function $c(x)$, we use $c(x) = |f(x)|$ since $|f(x) - 0| = |f(x)|$. Since we want $c(x)$, this is another minimization problem. Once again, we convert back to a floating-point number to calculate the fitness of each chromosome. Since converting to binary yields an absolute error of approximately 0.001 with `PRECISION = 10` decimal places of precision, calculating this fitness will have some error. Due to other computational errors that may arise when calculating cubes and trigonometric functions, we define our $\varepsilon = 0.002$. Thus, our algorithm will converge when the minimum fitness is less than ε.

The final change we must make is to ensure that every child chromosome generated is still within our given range [`MIN_VAL`, `MAX_VAL`). Intuitively, we do this by checking after every bit if the chromosome is within range. To accomplish this, we modify the `mate()` function. First, we convert both `MIN_VAL` and `MAX_VAL` to their respective binary strings, `min_str` and `max_str`. Then, we define an index pointer variable $i = 0$. Finally, we define two boolean flags, `check_min` and `check_max`, which determine whether we still need to check if the chromosome is within range. As such, `check_min` and `check_max` are initially set as `True`.

Inside of the `for` loop that creates a child chromosome, we select a `gene` as usual: randomly choose between the gene at index j from Parent 1, the gene at index j from Parent 2, or a gene normally distributed between 0 and 1. Before appending `gene` to `child_chromosome`, we now check if it is in range. If we still need to check if the the chromosome is within the maximum range — that is, if `check_max` is `True` — then there exist three cases: `gene` is (strictly) less than, (strictly) greater than, or equal to the ith gene in `max_str`. First, we check if `gene` is less than the ith gene in `max_str`. If it is, then we set `check_max` to `False`. This effectively says that, no matter what bits follow, the value represented by `child_chromosome` is strictly less than `MAX_VAL`, so we no longer need to check if it is within the maximum range.

In addition to checking if `gene` is less than `max_str[i]`, we also need to check if `gene` is greater than `max_str[i]`. If it is, then we set `gene` to `max_str[i]`. This effectively says that, up to i bits, `max_str` and `child_chromosome` are equal, so `child_chromosome` still needs to be checked. Notice that we do not need to handle the third case in which `gene == max_str[i]` since this is exactly the outcome of the second case.

A similar argument can be made for checking that `gene` is within the minimum range but switching `max_str` to `min_str`, as well as switching "less than" to "greater than" and vice versa. After each iteration through the loop, we also increment i so that we continue to check the bits in `min_str` and `max_str` that correspond to the bit in `child_chromosome`. Finally, we increment i when we add the decimal point to `child_chromosome` before we `continue` the iteration.

Table 1: Generations Until Convergence for the Target String Algorithm

Statistical Quantity	Value
Number of Runs	1000
Total Number of Generations	346728
Average Number of Generations	346.7
Minimum Number of Generations	51
Maximum Number of Generations	2041

Table 2: Generations Until Convergence for the Transcendental Equation Algorithm

Statistical Quantity	Value
Number of Runs	1000
Total Number of Generations	4033
Average Number of Generations	4.0
Minimum Number of Generations	1
Maximum Number of Generations	14

3 Results & Discussion

By running each algorithm multiple times, we see that every algorithm successfully converged. When we run each algorithm 1000 times to collect statistical data, we see some interesting results. Each of the three algorithms took on average different numbers of generations to converge. To match to the target string `"I love Geeks for Geeks!`, it took on average roughly 347 generations to converge, ranging from 51 to over 2000 generations; see Table 1. To solve $\cos x = x$, it took on average 4 generations to converge, ranging from 1 to 14 generations; see Table 2. Finally, to find the root at $x = 11.002$ for $f(x) = (x-10)^3 - \cos x + \sin x$, it took on average 5 generations to converge, ranging from 1 to 10 generations; see Table 3.

Notice how in at least one instance both the transcendental equation algorithm and the root finding algorithm (Tables 2 and 3, respectively) converged in one generation. Notice also that in every instance the target string algorithm took over 50 generations to converge. This

Table 3: Generations Until Convergence for the Root Finding Algorithm

Statistical Quantity	Value
Number of Runs	1000
Total Number of Generations	5288
Average Number of Generations	5.2
Minimum Number of Generations	1
Maximum Number of Generations	10

significant difference is likely due to the probability of randomly selecting the individual with a convergent fitness score. Since all 92 genes in `GENES` are equally as likely, the probability of randomly creating the target string of length 23 is $\frac{1}{92^{23}} \approx \frac{1}{1.469 \times 10^{45}}$, which is very nearly 0.

Calculating the probability of picking the value where $c(x) = 0$ based on some function $g(x)$ (either $\cos x = x$ or a polynomial-like equation) is easy, as picking a given element out of an uncountable set is 0. However, understanding the probability over a range is more complicated. Instead, we provide a very brief explanation based solely on intuition. The probability of selection a range of elements out of an uncountable is (paradoxically) non-zero. Therefore, the probability of finding the values that are within ε of 0 in the first generation is also non-zero. Thus, this provides a loose explanation to the difference in the minimum numbers of generations that the three algorithms take to converge.

4 Future Modifications

We now discuss some possible future modifications. Our algorithms work, but we want to make them both more accurate and more efficient computationally. The first modification would be to increase the precision of the binary representation. This would, in turn, reduce our ε values, leading to more accurate convergent values. Additionally, we could modify any number of our initial constants and values, including the following: population size, selection of the elite, selection of the top percent, and rate of mutation.

Finally, we would like to explore the effects on accuracy and computation time that adding a δ-based convergence criterion would have. Currently, we only base convergence on being less than ε. However, we would also base convergence on $|x-y| < \delta$, where x is the minimum fitness of the current generation and y is the minimum fitness of the previous generation. These two convergence criteria have the effect that not only must the minimum fitness be less than ε, it must also be within δ of the minimum fitness of the previous generation.

5 Summary

Thus far, we have seen three different applications of a genetic algorithm: the target string algorithm, the transcendental equation algorithm, and the root finding algorithm, which was a generalization of the transcendental equation algorithm. Thus, we can group the three algorithms into two subsets with the first subset containing only the target string algorithm and the second subset containing the other two algorithms. Notice that for the first subset, its search space contained strings, whereas the search space for the second subset was the real number line. Based on our statistical results, the second subset converged in much fewer generations compared to the first subset. Furthermore, we see a correlation among search spaces, convergence criteria, and the minimum, maximum, and average numbers of generations until convergence. Finally, we have provided some future modifications that we would like to implement to see their effects on the average number of generations until convergence.

References

[1] "Genetic algorithms," Geeks for Geeks. (n.d.), [Online]. Available: https://www.geeksforgeeks.org/genetic-algorithms/ (visited on 12/13/2023).

[2] X.-S. Yang, "Nature-inspired optimization algorithms," in 2nd ed. Academic Press, 2021, ch. Chapter 6: Genetic Algorithms, pp. 91–100. DOI: https://doi.org/10.1016/B978-0-12-821986-7.00013-5.

[3] "How to convert float number to binary?" Stack Overflow. (2010), [Online]. Available: https://stackoverflow.com/questions/3954498/how-to-convert-float-number-to-binary (visited on 11/17/2023).

A Appendix

The code for the three implementations are located in the `genetic-algorithm` repository in on my GitHub page.