THE COMPUTER PROGRAMMER: USING GENETIC ALGORITHMS TO WRITE CODE

AARON KAUFER, DANIEL TAN

1. Introduction

Ever since the dawn of the Industrial Revolution some 300 years ago, human labour has been increasingly replaced by machine labour, and routine tasks have been automated. At first, machines were clunky and rigid, and unsuitable for all but the most menial of tasks. But as technology has progressed, the variety of tasks that can be automated has expanded dramatically, growing to encompass not just physical tasks, but also intellectual tasks like weather forecasting and email classification. Against this backdrop of automation there seems to be one exception: ultimately, even the best programs must be designed, written and maintained by human programmers. As of now, computers are unable to write their own code. But what if we could change that?

With the rise of Artificial Intelligence, computers are starting to get an edge on what were previously thought to be necessarily human tasks – image classification and self-driving cars are among the most popular examples. The reason programming is so hard to teach to a computer is that unlike various other automated jobs (e.g. cashier) which consist of routine, methodical actions, programming requires a certain degree of problem-solving, and in some instances, creativity. In this paper, we investigate whether computers can learn to write nontrivial programs using a genetic algorithm.

2. Task Definition

The basic task for the algorithm is to receive, in some format, a description of a desired program for it to write, and subsequently return, in some language, the source code for that program. To be able to be more specific, we need to answer:

1. What format should we use to describe the desired program to the algorithm?
2. What constitutes a valid language for the algorithm to write its code in?
3. What counts as a program?

In this section, we will elaborate on (3) and (1), and in the Approach section we will elaborate on (2). We begin with defining what we mean by “program”.

For this project, we have split the notion of a ”program” into two distinct categories: Procedural Programs, and Functional Programs:

1. A **Procedural Program** is a program whose job is to perform a specific task. For example, a program which outputs ”Hello World!” upon running it would count as a procedural program.
2. A **Functional Program** is a program whose job is to model a function. Specifically, it is a program which takes in some sort of input and returns an output that depends solely on the input in a deterministic manner. For example, a program which inputs an integer n and outputs $5n^2 + 1$ would count as a functional program.

Of course, in reality, most programs are mixtures of the two; for example, a program might take in a string as input, perform a hash on it (applying a function), and send the result to a network (performing a procedure). For this project, however, we have opted to specifically focus on just procedural and functional programs.

The individual task definitions vary slightly according to which type of program is attempting to be created.

2.1. Procedural Programs.

Since the idea of a procedural program spans an incredible amount of possible programs, we have chosen to narrow in specifically on procedural programs whose sole task is to print out a string. The naive version of the task definition is:

Task. *Given a target string S, generate a program which, when run, outputs the string S.*

Of course this task is trivial if, for example, the algorithm a priori:

1. knows the target string,

Date: June 2018.

(2) knows the syntax of the language of the program it is trying to write.

For example, if it knew the target string S and that it was generating a python program, then it could just come up with the simple program:

```
print("S")
```

where S gets replaced by the target string. The task that the algorithm is really trying to accomplish as follows:

Task. *Given nothing other than the ability to compare a program's (string) output to the desired output S, generate a program which outputs S.*

This task is now much harder. With no knowledge of the syntax of a language, how can you possibly write programs in that language? The key to this comes from two important factors: (1) a well-chosen language, and (2) the use of randomness. The details of this are outlined in the approach section.

2.2. **Functional Programs.** The setup for the task of generating functional programs is similar to that of procedural programs in terms of overall design. Additionally, with the purpose of serving as a stepping stone to a later progression to general n-ary functions, we have decided to restrict to just unary functions. Additionally, general mathematical functions allow for any type of input, but for the sake of practicality, we restrict the domain of the function to just be (real) numbers. We may formulate a similar naive setup:

Task. *Given a unary function f, generate a program that takes in an input x and outputs the value $f(x)$.*

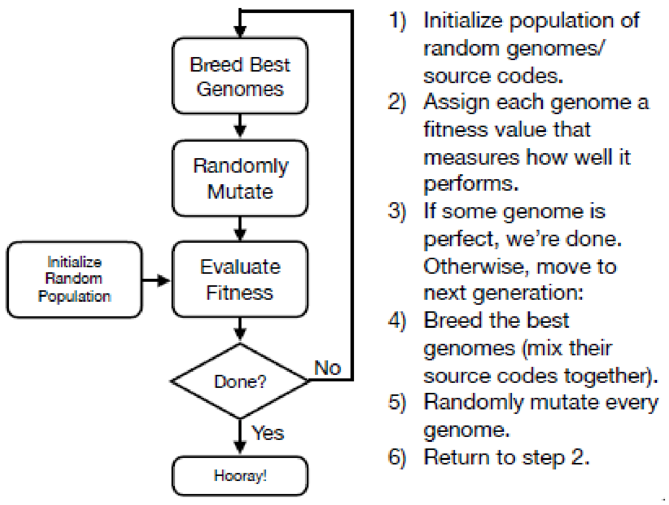
As with procedural programs, it is crucial that the algorithm not know the syntax of the language in which it is writing the function, and it should not be able to interact with the function f other than comparing the output values of a given program to that of f.

Additionally, since there is no way to say with complete certainly whether or not the desired function f is equal to the generated function g by simply comparing output values (because you would need to compare their outputs at infinitely many input values), instead of feeding an actual function to the algorithm, it suffices to feed it a finite set of input-output data points (x_i, y_i). For example, to model the function $f(x) = x^2$, the input could be $\{(0,0),(1,1),(2,4),(3,9),(4,16)\}$. Thus we can refine the task definition:

Task. *Given a finite set of data points $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^2$, generate a program, which takes an input $x \in \mathbb{R}$ and outputs $g(x) \in \mathbb{R}$, where $g(x_i) = y_i$ for all i. Ideally, if the data points (x_i, y_i) were generated as $(x_i, f(x_i))$ for some function f, then $g = f$.*

3. Approach

Our approach to this task was to use a genetic algorithm. Broadly speaking, we treated the source code of a program as its "genome" and then applied principles of natural selection to organically evolve a program that accomplishes the desired functionality. An overview of how genetic algorithms work is given in Figure 1 below:



1) Initialize population of random genomes/ source codes.
2) Assign each genome a fitness value that measures how well it performs.
3) If some genome is perfect, we're done. Otherwise, move to next generation:
4) Breed the best genomes (mix their source codes together).
5) Randomly mutate every genome.
6) Return to step 2.

FIGURE 1. A brief overview of genetic algorithms

In order for programs to be able to be generated using a genetic algorithm, there must be (1) a way to encode the program as a "genome", and (2) a way to measure how "fit" a program is. For encoding a program as a genome, we simply use its source code, and to measure how "fit" a program is, we pass it into a fitness function which runs it and compares its functionality to that of a perfect program's functionality. The specifics of each vary between procedural programs and functional programs, but the overall outline of using genetic algorithms to produce programs is the same, and can be broken down into the following six components:

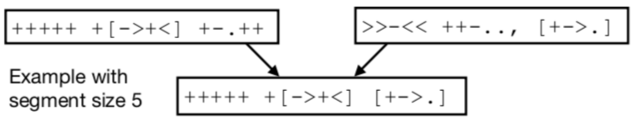
(1) **Overall Strategy:** The overall strategy controls the different populations and how they interact with each other (e.g. How many different populations are there? How big are they? Do stagnant populations die off? Do different populations interact with each other?).
   - For all of our tests, we have chosen to use a `ManyPopulations` overall strategy. This means that a fixed number (`numPopulations`) of populations exist, each of a fixed size (`populationSize`), which do not interact with each other. The purpose of this is to allow for enough populations to ensure that they don't all get stuck at local maxima. In practice, we generally set `numPopulations=10` and `populationSize=100` for an optimal tradeoff between speed and accuracy. In addition, we have added a parameter `accuracyCutoff` between 0 and 1, which stops stops the algorithm when the highest-fitness genome has larger than an `accuracyCutoff` fraction of the best possible fitness. In practice, we set `accuracyCutoff` to either 0 or 1.

(2) **Interpreter:** The interpreter decides how to take the genome (source code), and run it. Effectively, it controls the language in which the genetic algorithm writes code. For consistency, the setup we have built requires (1) all genomes must consist of solely ascii characters that can fit into a string, and (2) the input and output must be a sequence of bytes. The interpreter and fitness functions are free to interpret these bytes as they please (e.g. the procedural programs generally interpret them as strings, while functional programs recast them as floats).
   - Implementations vary across procedural and functional programs. Specifics will be discussed later.

(3) **Fitness Function:** The fitness function takes a genome and evaluates how "fit" it is, which is just a measure of how close its performance is to the desired performance. Higher fitness value means better performance.
   - Implementations vary across procedural and functional programs. Specifics will be discussed later.

(4) **Breeding Selector:** The breeding selector is in charge of looking at all the genomes and their fitness values and deciding which genomes breed in order to create the next generation.
   - A common breeding selector is roulette selection, in which genomes are chosen randomly from the population, where the probability that a given genome is chosen is proportional to its fitness. For time efficiency, we chose a simpler strategy: elite selection. We fix a percentage value of the variable `elitePercentage`, and then in each generation, the top `elitePercentage` of any given population are copied identically to the next population. Then, genomes in the top `elitePercentage` are chosen to breed randomly with other genomes in the population. We generally fix `elitePercentage=20%`.

(5) **Breeding Method:** The breeding method takes two parent genomes and produces a child genome for the next generation.
   - For breeding, we have implemented what we call **Segment Breeding**. Specifically, we partition both parent genomes into segments of length `segmentSize`, and for each segment, we randomly choose which parent the child inherits from. This is demonstrated in Figure 2.



```
+++++ +[->+<] +-.++
>>-<< ++-.., [+->.]
+++++ +[->+<] [+->.]
```

FIGURE 2. Segment Breeding Example

(6) **Mutation Method:** The mutation method is responsible for taken a genome and randomly mutating it.
   - For our tests, we have chosen to use a `ReplaceInsert` mutation method. The algorithm works as follows. A variable `mutationRate` is fixed (one out of every `mutationRate` genes are mutated). Then, let `dna` be the string of genes in a genome, let C be the set of valid ascii characters (according to the interpreter), and $S = C \cup \{\star\}$, where $\star$ is some arbitrary character not in C. Then the algorithm works as follows:

```
for i in 0..len(dna)-1:
    with probability 1/mutationRate, do:
        let c be a random element of $S$
        if c == $\star$:
            insert a random character from $C$ before dna[i]
        else
            replace dna[i] with c
```

For procedural programs, we have found `mutationRate=100` to be effective, and for functional programs, we have found `mutationRate=10` to be effective.

3.1. **Procedural Programs.** For procedural programs, we have chosen to use the esoteric programming language BrainFuck (BF). The specifics of the language can be found at its Wikipedia page. We chose BF for two main reasons. Firstly, it is proven to be turing complete, so in theory it should be capable of accomplishing any computable task. Secondly, BF programs are comprised entirely of the eight characters `+-<>,.[]`. This makes it so that BF programs can easily and conveniently be thought of as a type of genome. The lack of diversity of characters makes breeding and mutations much more likely to be effective.

For a fitness function, we do a basic character by character comparison between the outputted string and the target string, where we compute the distance between each character in the corresponding strings and weight small distances largely, and we sum up the results. Additionally, after processing an index `i` such that `output[i]!=target[i]`, we stop computing the fitness. This makes it so that the genetic algorithm builds the string up from the start. Specifically, the algorithm is as follows:

```
fitness = 0
for i in 0 ... min(len(target), len(output))-1:
    fitness += 256 - abs(target[i] - output[i])
    if target[i] != output[i]: break
```

Finally, we add on a small reward (`lengthReward`) if the length of the output string is equal to the length of the target string. This is so that the algorithm isn't considered perfect in the event that it prints the target string and then some more characters. Typically, we set `lengthReward=100`.

3.2. **Functional Programs.** To construct a language to represent functions, we considered the tree-like nature of mathematical functions. As such, we named the language **TreeFunction**. The idea is that n-ary operators can be thought of as roots of trees, with their branches the arguments. With this framework in mind, we encode a function as a pre-order traversal of its tree. This is seen in Figure 3.

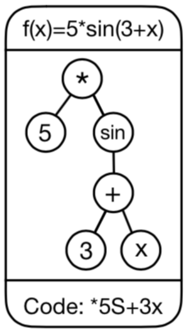


FIGURE 3. Encoding a Function as a Tree

For a fitness function, we recall that the algorithm is given a finite set of input-ouput pairs $(x_1, f(x_1)), \ldots, (x_n, f(x_n))$ that it is hoping to match. Thus, to measure how successful an outputted function g is, we compute each of $g(x_1), \ldots, g(x_n)$, and we compare each $f(x_i)$ to $g(x_i)$. Roughly speaking, we compute the relative error between the two, reward pairs with low relative error, and add up the results. Specifically, we assign a fitness to each input point x_i, and add them up. We compute the fitness at x_i as follows: suppose for the sake of convenience that $f(x_i) > 0$ (if it is not, flip the sign of $f(x_i)$ and $g(x_i)$). Then, we give a fitness of zero to x_i if $g(x_i) < 0$ or $g(x_i) > 2f(x_i)$. If $g(x_i)$ is between 0 and $2f(x_i)$, then we apply the following logic:

- If $g(x_i) = f(x_i)$, give x_i a fitness of 1.
- If $g(x_i) = 0$ or $g(x_i) = 2f(x_i)$, give x_i a fitness of 0.
- Otherwise, linearly interpolate the fitness of x_i according to where $g(x_i)$ falls in the range 0 to $2f(x_i)$ (for example, if $g(x_i) = \frac{1}{2}f(x_i)$ or $g(x_i) = \frac{3}{2}f(x_i)$, then x_i gets a fitness of $\frac{1}{2}$).

Additionally, we have a parameter `lengthPunishment`, which we multiply by the length of the program (number of nodes in its tree) and subtract from the fitness. This discourages long and overly complex programs.

We chose this fitness method because it computes a form of relative error instead of absolute error, so that large functions like e^x and small functions like $\frac{1}{x}$ can both be approximated.

Finally, we note that for breeding, since this language less naturally falls into segments like BF does, we used a segment breeder with segment size 1, so individual genes are chosen from the parents.

4. Related Work

The choice of the language BF was inspired by programmer-blogger Kory Becker, who embarked on a similar experiment of using genetic algorithms to generate BF programs in 2013. Her work can be found on her blog here. We largely recreated many of her results, and improved on them in various ways. Specifically, our breeding algorithm differed from Becker's (which was a more standard crossover algorithm), and our overall strategy improved upon Becker's (Becker ran a single-population strategy while we ran multiple populations). Additionally, in our results section, we quantified a variety of factors that Becker did not touch.

The functional program generation aspect of our algorithm is very similar to a larger topic known as Genetic Programming. Genetic Programming also treats programs and functions like trees, as we do in our algorithm; however, Genetic Programming generally doesn't then take the next step of encoding the tree as a string of characters and mutating/breeding it as such. Generally, the genomes of Genetic Programming are still trees, and the mutation and breeding operations happen between different branches of the trees.

5. Results

We ran a suite of experiments for both procedural programs and functional programs. For each experiment, we tested several experimental settings, and measured the number of generations required to obtain a correct answer.

5.1. Procedural Programs.

For each of the following experiments, we have fixed the setup such that there are 10 populations, each of size 100. For breeding, we use a segment size of 20, and for mutating, we use a mutation rate of 100. In general, the two biggest factors that can effect the number of generations and time it takes to generate a string are (1) the length of the string, and (2) the complexity of the string. For this section, we test generation numbers and time against length of target string as well as two different measures of complexity of the string: character span and entropy.

5.1.1. *Length.*

In this section, we test the generations required of the program for target outputs of varying length. For each experiment, the listed generation counts and times taken are averages of three trials. The results are summarized in Table 1.

Table 1. Results for Strings of Veryving Length

Target output	Average Generation Reached	Average Time Taken (s)
`h`	216	0.93
`he`	320	1.22
`hel`	1243	4.16
`hell`	1497	5.53
`hello`	1767	6.34
`hellow`	2417	8.73
`hellowo`	3047	11.71
`hellowor`	3448	13.52
`helloworl`	4286	17.37
`helloworld`	7415	32.20

Interestingly, both the generation count and the time taken grow roughly linearly with the length of the target string, with the exception of the tenth character, which in all trials required longer to get. We applied a linear regression, and got an R^2 value is 0.929 with a corresponding p-value of 0.000106, indicating that we have a strong linear correlation[1].

[1] Refer to Appendix B for an actual plot

5.1.2. *Character Span.* One feature of the BF language is that it outputs characters by calculating their ascii values, and it does so through chains of `+` and `-`. Thus, intuitively, strings with characters that are far apart on the ascii table will be harder to reach. To test this out, we introduce the concept of **character span**, which is meant to intuitively capture how much "up and down" occurs between pairs of consecutive characters:

Definition 5.1. *The* ***character span*** *of a string w with n characters is defined as $\sum[\frac{|x-y|}{n-1}]$, where the difference is calculated using ASCII values of characters and the sum is taken over all pairs of consecutive characters x, y contained in w.*

In Table 2, the results of fixing a string's length while varying its character span are shown.

TABLE 2. Results for Common Words of Different Character Spans

Target output	Span	Average Generation Reached	Average Time Taken
`loops`	1.75	1368	5.05
`hello`	3.25	1804	6.57
`world`	6.25	3508	13.99
`fires`	9.75	4007	15.69
`zebra`	14.25	4416	17.45

We ran a linear regression and found that the R^2 value is 0.93, with a corresponding p-value of 0.0191. Again, this indicates a strong linear relationship[2].

5.1.3. *Entropy.* Another measure of string complexity is it's Shannon Entropy. This is defined as:

Definition 5.2. *The* k^{th}***-order entropy*** *of a string w is defined as the Shannon entropy of the distribution of $k-$grams of the string, i.e. $\sum_{a_k \in w} -p_{a_k} \log(p_{a_k})$, where the sum is taken over all k-character substrings a_k of w, and p_{a_k} is the probability of a_k appearing in the string.*

The **first-order entropy** of a string represents the evenness of the distribution of unigrams (characters) in the string. Because strings with higher first-order entropy are usually more complex, the minimum required length is higher, and so we hypothesize that there is an increasing relationship between first-order entropy and the number of generations required. The results are summarized in Table 3.

TABLE 3. Results for Varying First-Order Entropy

Target output	First-order entropy	Average Generation Reached	Average Time Taken
`aaaaa`	0	583	1.95
`aafaa`	0.50	1080	3.55
`aaffe`	1.05	819	2.81
`aabcd`	1.33	687	2.41
`abcde`	1.60	796	2.57

The results show that there is no correlation between the first-order entropy of a string and the generation time required. It is still possible that there is a relationship, but perhaps the effect is negligible given our current parameters.

5.2. **Functional Programs.** For functional program generation, rather than establish trends, we chose to explore the capabilities and limitations of the algorithm, as well as some applications. As with procedural programs, each experiment has 10 populations of size 100 each. The breeding is segment breeding with segment size 1, and the mutation rate is 10. As before, all results are averages of 3 trials.

5.2.1. *Simple Function Guessing.* For this section, we gave the algorithm its original task: given a function, attempt to program that function.

For each of the following tests, the constants of the TreeFunction language are $\{0, 1, \ldots, 9, \pi\}$ and the operations are the basic operations $\{+, -, \times, \div\}$, the trigonometric operators $\{\sin(x), \cos(x), \tan(x)\}$, and miscellaneous operations $\{x^y, \frac{1}{x}, e^x, \log(x)\}$. Functions were evaluated on inputs from 1 to 10, with an accuracy cutoff of 1 (so they were guessed perfectly) and the length punishment was 0. The results for various functions are presented in Table 4.

[2]Refer to Appendix B for an actual plot

Table 4. Results for Various Simple Functions

Target Function	Average Generation Reached	Average Time Taken (s)
$9x+2$	25	0.11
$3x^2+2x+3$	52	0.23
$\sin(5x+7)$	479	2.54
$e^x+\tan(2x)$	100	0.44
$15x\log(x)+3$	1277	7.16
$\frac{2x+3}{4x+5}$	276	1.48

Remarkably, it is able to consistently get simple enough functions in under 10 seconds. For more complicated functions or functions with high constants, it is generally unable to exactly guess the function. For example, if we replace $\frac{2x+3}{4x+5}$ with $\frac{29x+31}{14x+50}$, then the algorithm quickly settles on some long, complex local maxima and is unable to recover from there.

5.2.2. *Function Approximation.* Although the algorithm generally fails to exactly recover complicated functions, it nonetheless produces incredibly accurate approximations, since maximizing fitness on a set of data points is equivalent to minimizing error or those points. Additionally, since the TreeFunction language can be modified to include whatever operations or constants necessary, the approximations can be constrained. To demonstrate this, we restrict the operations of TreeFunction to just $\{+,-,\times,\div\}$ and we remove π from the set of constants. With this inventory, it is solely capable of producing rational functions (quotients of polynomials). If the target functions is, say, $f(x)=\sin(x)$, then the function that the algorithm outputs will be a rational function approximation to $\sin(x)$. For these experiments, since the goal is to model functions with high precision, the accuracy cutoff was set to 0.999999. Table 6 shows the results of performing rational function approximation on some common functions. In addition, we had the algorithm approximate the constant function $f(x)=\pi$, which it did to a high degree of precision (with a remarkably simple fraction). All functions were trained on data points from $1-20$ inclusive, except e^x which was trained on inputs of $1-8$ because it grew too quickly, and $sin(x)$ which was trained on 20 values from the interval $[0,2\pi]$ because it is a periodic function.

5.2.3. *Constant Guessing.* One of the applications of the algorithm is that it can be used to guess simple representations for complex mathematical expressions. For example, a famous mathematical problem is the evaluation of the sum $\sum_{n=1}^{\infty}\frac{1}{n^2}$. Leonhard Euler shocked that mathematics community in the eighteenth century when he showed that it was actually equal to $\pi^2/6$. Such a seemingly bizarre answer would have been nearly impossible to guess at the time. With this algorithm, however, asking it to generate the constant function $f(x)=\sum_{n=1}^{10^6}\frac{1}{n^2}$, it is capable of easily guessing the correct answer. The results of it guessing that and other constants from numerical approximations of more complicated expressions is shown in Table 5. For this experiment, the TreeFunction language was allowed access to the constants 0 through 9 as well as π, and its operations consisted of $\{+,-,\times,\div,e^x,\log(x)\}$.

Table 5. Evaluations of Common Sums

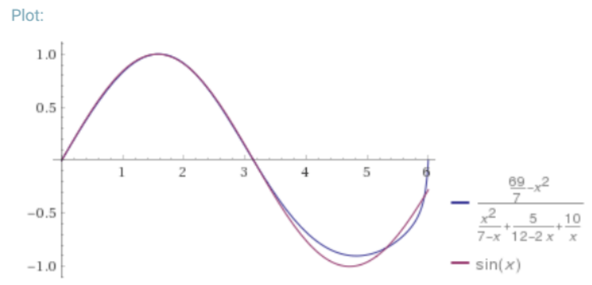
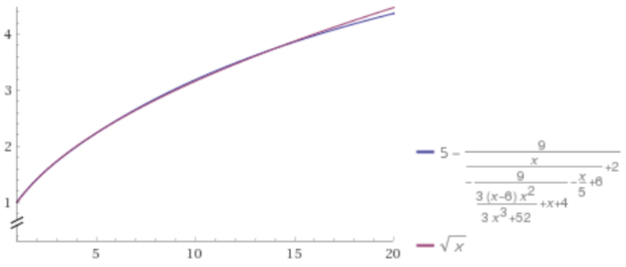
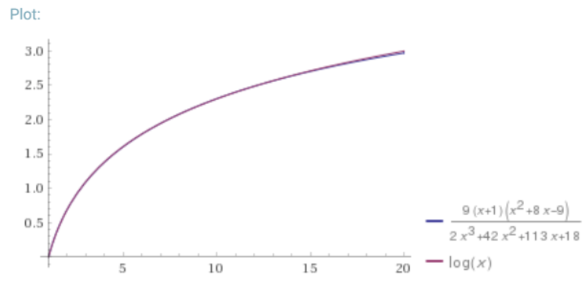
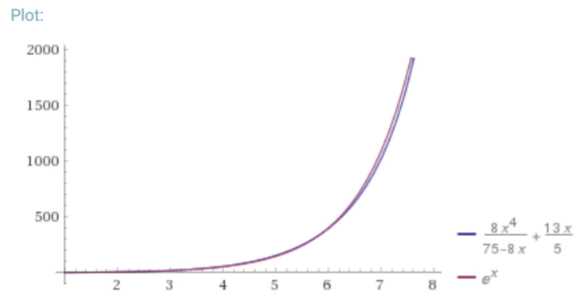
Expression	(Simplified) Output	Generations Reached	Time Taken
$\sum_{n=1}^{10^6}\frac{1}{n^2}$	$\frac{\pi^2}{6}$	272	0.45
$\sum_{n=1}^{10^6}\frac{(-1)^{n+1}}{n^2}$	$\log(2)$	3	0.01
$\sum_{n=1}^{10^6}\frac{2^n}{n!}$	e^2	1	0.01

5.2.4. *Polynomial Interpolation.* There is one class of functions that the algorithm is particularly good at guessing: low-degree polynomials with integer coefficients. Given a set of data points, determining the polynomial that runs through them is precisely the problem of polynomial interpolation, so alternatively phrased, the algorithm is capable of performing polynomial interpolation for low degree polynomials with integer coefficients.

To make the algorithm only produce polynomials, we restrict its constants to 0 through 9 and its operations to $\{+,-,\times\}$. Additionally, we test each polynomial on the input points 1 to 10 inclusive. The results for performing polynomial interpolation on a variety of randomly chosen polynomials is shown in Table 7.

The results demonstrate that the algorithm is capable of doing exact polynomial interpolation consistently in under 30 seconds for polynomials of low degree and reasonably small coefficients. In fact, by comparing $111x^2+223x+139$

Table 6. Modeling Common Functions with Rational Functions

Expression	Algorithm's Output (With Correct Output for Reference)	Generation Reached
π	$\frac{335}{115} \approx 3.1415929$ versus $\pi \approx 3.1415927$	960
$\texttt{sin}(x)$		20000
$\sqrt{x}$		10000
$\log_e(x)$		35492
e^x		10082

and $5x^5 + 12x^3 + 4$, we can see that having coefficients of high magnitude often takes longer than having polynomials of high degree. For high degree polynomials with many terms and distinct coefficients, the algorithm generally isn't able to accurately identify every coefficient, but almost always correctly identifies the degree.

As a baseline for polynomial interpolation, we compare our algorithm against the Lagrange Interpolation Formula[3] (LIF), a standard way of recovering a polynomial from its evaluation at some values. In particular, given n data

[3]A brief description of the Lagrange Interpolation Formula can be found in Appendix 7.1. For additional details, visit https://en.wikipedia.org/wiki/Lagrange_polynomial

Table 7. Results for Polynomial Interpolation

Target Function	Generation Reached	Time Taken (s)
$2x + 1$	4	0.02
$57x + 23$	20	0.09
$1234x + 5678$	410	2.03
$x^2 + 2x + 3$	16	0.07
$10x^2 + 15x + 12$	132	0.69
$57x^2 + 86x + 39$	717	3.67
$111x^2 + 223x + 139$	4418	26.90
$x^3 + 2x^2 + 7x + 9$	94	0.5
$12x^3 + 20x^2 + 15x + 31$	854	4.53
$5x^5 + 12x^3 + 4$	1407	7.15
$98x^7 + 103x^6 + 19$	5768	36.03

points, the LIF typically returns a polynomial of degree $n - 1$, regardless of the degree of the polynomial that generated the data. In this sense, despite being deterministically always capable of generating a polynomial that accurately fits the data, the LIF is at risk of overfitting the data when, for example, it is given 8 data points that were generated by a cubic polynomial.

In contrast, as seen in Table 7, despite being given 10 data points for every polynomial, the genetic algortihm was correctly able to guess the smallest degree necessary to fit the data points. Thus, the algorithm has generalization capabilities that the LIF lacks.

5.2.5. *General Remarks.* In general, we observe from our experiments that the algorithm performs a sort of implicit regularization, and intrinsically avoids "over-complex" solutions when simple ones suffice. This can be observed most clearly in the section on polynomial interpolation, when the program is able to find relatively simple polynomials that match the data well, as compared to the LIF which finds an unnecessarily high-degree polynomial. Intuitively, this is because as the length of a given expression grows, the probability of it being discovered by the algorithm decreases. Hence, a shorter expression is more likely to be found than a longer one. Another way of thinking about it is that the genetic algorithm builds complex programs using simpler ones as building blocks. Thus, it must explore the solution space of "simple" programs before it can explore more complicated ones, and so short programs that are a good approximation of the function will be returned before the algorithm has any chance to explore more complicated programs.

Additionally, we find that in order for the algorithm to evolve complex programs, there must be a "path of increasing fitness" that leads from random programs to the desired program. Otherwise, the algorithm tends to get stuck in local optima. This is quite reminiscent of evolutionary biology, as the changes made by natural selection must often be incremental - even slight deviations from the locally optimal solution are heavily penalized. Thus, we argue that our algorithm's tendency to get stuck in local optima is not an implementation quirk, but representative of the class of genetic algorithms as a whole.

6. Conclusion

Overall, we have presented a genetic-algorithm based approach to spontaneously evolving computer programs. For procedural programs, we found that the generation time grows linearly with length and character span, but not with the first-order entropy. For functional programs, the function solver easily finds simple functions, and can approximate more complex functions to a high degree of precision. In particular, the program is good at (constrained) function approximation and fitting data to low-degree polynomials.

For this project, we built a shell that allows for complete customization of the six components of the genetic algorithm outlined in the Approach section. With this, we are free to experiment with improving on any of the six components. Here are some ways that each could be improved:

(1) **Overall Strategy:** One aspect of our design was that the different populations never interacted with each other. Instead, a design could be tried in which, for example, many different populations created successful genomes, and then those genomes bred with each other. Additionally, features like killing off stagnant populations could be implemented to diminish the effects of getting stuck in local maxima.

(2) **Interpreter:** Though BF was a nice language for its short alphabet, it was certainly not designed for the purpose of being used in a genetic algorithm. It would be interesting to see what type of language could be designed to function well with a genetic algorithm (i.e. could use a small character set, but have high functionality).

(3) **Fitness Function:** For functional programming, the fitness function just calculated a linear random error. With this model, functions that are fairly bad and extremely bad both get fitness scores of 0, despite one being better than the other. Choosing a smoother model for error calculations could aleviate this issue.

(4) **Breeding Selector:** Though elite selection was time efficient, implementing Roulette Selection (fitness proportionate selection) would likely increase genetic diversity and cause less local maxima issues.

(5) **Breeding Method:** The biggest issue with breeding in BF programs (and programs in general) is that internal structure, such as the BF loop structure (balance of `[` and `]`) is generally thrown off, rendering the children programs useless. An interesting future path would be to devise a breeding function that respected the internal structures of programs.

(6) **Mutation Method:** Similar to breeding methods, mutation methods also often destroy the internal structures of programs. While this is necessary to maintain a certain level of genetic diversity, it could be explored whether there was a balance to strike between mutations that destroy structure and preserve it.

The code can be found at https://github.com/ASKProducts/BFAI

7. Appendix A

7.1. Lagrange Interpolation Formula.

Definition 7.1. *Given $k+1$ points $(x_0, y_0), \ldots, (x_k, y_k)$ where all the x_i are distinct, the* ***Lagrange interpolation polynomial*** *is defined as $L(x) = \sum_{i=0}^{k} y_i l_i(x)$, where $l_i(x)$ is defined as:*

$$l_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \tag{1}$$

We observe that each of the polynomials $l_i(x)$ is a product of k linear terms, and so is a $k-$degree polynomial. Thus, the interpolation polynomial $L(x)$ is the sum of degree k polynomials, and so has degree at most k. In practice, however, the Lagrange Interpolation Formula usually returns a $k-$degree polynomial, and is unable to return a low-degree polynomial.

8. Appendix B - Linear Regression for Procedural Programs

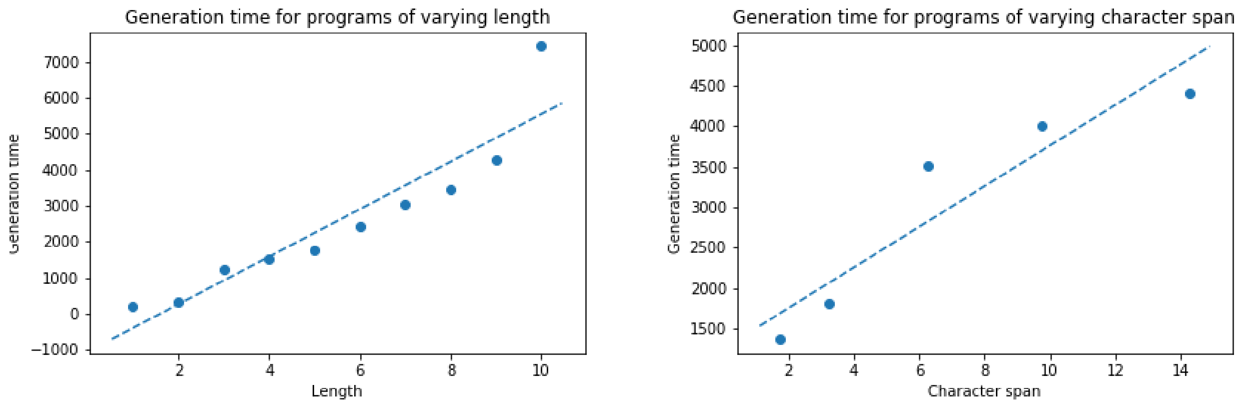


Figure 4. Linear regressions of generation time against various characteristics