BF-Programmer: A Counterintuitive Approach to Autonomously Building Simplistic Programs Using Genetic Algorithms

Kory Becker
kbecker@primaryobjects.com

Justin Gottschlich
justin.gottschlich@intel.com
Intel Labs

Abstract

Over the last decade there have been significant advances in computation, data management, and algorithms, which have enabled large-scale use of machine learning (ML). Many fields of research and practice now use ML as a core vehicle for progress, yielding results that are bordering on revolutionary (e.g., computer vision and natural language processing). Yet, somewhat ironically, advances using ML for programming have not seen progress on a similar scale.

In this paper, we present the BF-Programmer system, which uses a genetic algorithm, a specific type of ML, to autonomously develop software programs given minimal human direction. We show that using a well-suited programming language as its foundation, the BF-Programmer system can autonomously generate a wide variety of simplistic programs. We also discuss numerous attempts to autonomously generate software programs, prior to the development of BF-Programmer, which had varying degree of success.

Keywords Genetic algorithm, Program synthesis, Genetic programming, Evolutionary computation, Esoteric, Brainfuck, Artificial intelligence, Machine learning, Programming languages

1. Introduction

Since the invention of the computer, having the ability to correctly and efficiently develop software programs has been a principle challenge [4]. To help address this, countless breakthroughs have been made in the field of software development. Some of these include safety and flexibility advances in static, dynamic, and gradual type systems [3, 27]; simplification, safety, and robustness advances using automatic memory management and garbage collection systems [5, 12]; generality and specificity progress in both general-purpose and domain specific languages [26, 29]; and, of course, a plethora of tools aimed at assisting programmers in nearly every way imaginable [9, 10, 24].

Yet, simultaneous advances in hardware innovation have occurred with similar frequency, such as increasingly performant general purpose multi-core CPUs with advanced hardware extensions [18, 31], low power system-on-chip (SoC) edge compute devices [13], high-performance pluggable coprocessors with near supercomputing performance of yesteryear [11], wide data-parallel graphics processing units (GPUs) [22], and application specific integrated circuits (ASICs) for deep neural networks and computer vision [1, 28], to name a few. While such hardware advances continue to broaden and deepen the space of what is computationally tractable, they have the fracturing side-effect of complicating and exacerbating the tension between the ease of software programmability and the ability for human programmers to generate maximally efficient code.

Within this context, it can be argued that the process of human-focused software development has moved from a problem of reasonable complexity to one of intractability due to the sheer volume of heterogeneous hardware components and software complexity. It is for this reason, that we believe a fundamental shift must be made in the way we develop software. Humans may no longer be capable building software that manages both the correctness associated with the complex hardware computational interactions while simultaneously achieving near peak performance for such devices. Instead, we believe that computers should drive the development of software, with humans specifying only the minimal amount necessary to achieve the goal. In this paper, we present early research along these lines.

1.1 The Evolution of Programming Languages

Over the last several decades, programming languages have followed a steady path of providing more simplistic and higher-level programming abstractions, aimed at reducing the challenge of software development for humans. From the lowest level of binary code to the increasingly higher-level abstractions (e.g., Assembly to BASIC to C to Java to Python, etc.), programming languages have proliferated a design goal of facilitating human use. Although this trend

is natural in an era where humans perform the majority of software development, we argue that it is suboptimal in an environment where the majority of programming is performed by machines.

The notion of computers automatically creating their own software programs has been a long-standing goal of computing technology and artificial intelligence [25]. By removing humans from the time-intensive and error-prone process of software development, computer software has the potential to be generated in a completely automated, streamlined, and more correct and optimized fashion [6, 16].

As we show through experiments presented in this paper, we believe languages that are best aligned for ML-based programming are fundamentally different than those for humans. We propose that ML-aligned languages are ones that *(i)* not unexpectedly, provide support for the broadest number of possible program solutions (i.e., Turing completeness [30]), *(ii)* facilitate a high likelihood of generating functional programs from random sequences of their instruction set, and *(iii)* are exceptionally succinct, containing the minimal number of instructions to satisfy *(i)* and *(ii)* making them uniquely unfit or ill-advised for use by humans. We provide a deeper analysis of this exploration throughout the paper.

This paper makes the following technical contributions:

1. We present BF-programmer, a system that autonomously generates software using an ML-suitable programming language along with genetic algorithm techniques.
2. We provide technical exploration and evaluation criteria for programming languages that make them less or more amenable to ML-based software generation.

2. The Problem with Programming Languages

At the highest level, we believe one of the fundamental limitations using today's programming languages for autonomous program generation is that they were intended for human use, not machine use. This *intention of design* is a fundamental restriction in the expressiveness of many programming languages when used in concert with ML-based program generation. In this section we highlight some of these details by providing a brief exploration of some of our early experiments in this space.

2.1 An Early Failure with BASIC

In the early stages of this project, we began researching software program generation using the programming language BASIC, with nothing more than if-then conditionals. In the end, this experiment was ultimately a failure, however, it inspired and refined our thinking and pushed us to consider alternative, and eventually, workable solutions. We brief discuss the highlights of our experimentation with BASIC below.

Lack of generality First, conditional branches are unable to handle a large variety of computing tasks. In addition, static branching cannot anticipate the changing needs of a dynamic programming environment. As such, the approach we first explored which used embedded logical conditionals could only provide a fixed solution for a specific set of predetermined tasks and would be incapable of generalizing on different or evolving program spaces. While useful for fixed programs, such solutions are limited in terms of generality.

Large syntactical combinations To accommodate the composition of prose, many programming languages are defined with flexible syntaxes that provide numerous legal combinations and can be read much like complete sentences (e.g., if-else statements) or even paragraphs (e.g., flow control like for or while loops). While this approach is amenable to human writing, it can create significant complexity for machine learning program generation. Furthermore, some instructions within programming languages are potentially harmful and if used in experimentation for ML-based program generation may cause irreversible damage.

A key benefit of using a programming language that has a reduced instruction set is that it allows the state space of any program to be explored more easily by an ML model which has the side-effect of reducing the complexity of generating syntactically legal and functionally correct programs. In other words, the larger the language's unique instruction set, the greater the density of the program's combinatorial search space for a given program. Such density exacerbates the challenge of not only building functionally correct programs, but also, more trivially, syntactically correct ones.

2.2 Program Synthesis at the Instruction Level

After our experiment with BASIC, we shifted away from using a higher-level programming language's instruction set and instead tried an inverted approach starting at the byte-level. In essence, rather than using a pre-existing programming language, we aimed to understand what was required to successfully generate a simplistic program by piecing one byte together at a time. Our working hypothesis was simple: there is a certain representation of bits that when combined together will result in an executable program and correct solution for any given task. The goal then is simply to find the correct corresponding combination of bits.

Because the number of bit combinations can be extremely large, the search space for finding the correct sequence of bits can become a computationally intensive task. To mitigate this, a method was required for narrowing down the search to allow for program generation within a reasonable time constraint. We hypothesized that by using genetic algorithms, we could potentially guide the search in order to locate a working series of bits, resulting in successful program generation.

2.3 A Brief Introduction to Genetic Algorithms

A *genetic algorithm* (GA) is a type of artificial intelligence, modeled after biological evolution, that begins with no knowledge of the subject, aside from an encoding of genes that represent a set of instructions or actions [7]. In the concept of GA-driven computer programming, a series of programming instructions are selected at random to serve as an initial chain of DNA. The complete genome is executed as a program, with the resulting fitness score calculated according to how well the program can solve a given task. This is performed with a sufficiently large population size. Those that have the best fitness are mated together to produce offspring.

Each generation of programs receive extra diversity from evolutionary techniques, including roulette selection, crossover, and mutation [20]. The process is repeated at each epoch, with each child generation hopefully producing more favorable results than its parents' generation until a target solution is found. Through this process, applying GAs to computer programming automation enacts a survival of the fittest model for computer program generation [19]. A deeper examination of these GA principles are provide in Section 4.

2.4 Bit-level Manipulation of an Executable

We had initially considered using GAs to programmatically modify a template executable file, directly altering bits within the executable's code and evaluating the resulting program behavior. However, this was deemed an unreasonable task, as changing random individual bits frequently rendered an executable as corrupt. Further, a corrupt executable has no way of offering granular feedback to indicate how better or worse the changed bits are, leaving us unable to determine the suitability of a resulting sequence of bits for producing a program to solve a given task.

Still, the idea of using GAs to search a space of available instructions seemed a reasonable process, provided we could locate a simplistic programming language instruction set, similar to bit-level instructions, that would more easily allow for feedback during the generation process.

3. Introducing BF-Programmer

Although our early experiments attempting to generate programs using BASIC and bit-level manipulation of executable files were limited in generality or failed, outright, those experiments led us to the hypothesis that a notable limitation in applying ML techniques in automatic program generation was in the size of the vocabulary of a given programming language. In essence, we believed that the smaller the programming language's unique vocabulary, the more likely we would be able to generate meaningful results.

To test this hypothesis, we developed a rudimentary programming language consisting only of add, subtract, loop, and print instructions. We then tried to autonomously generate programs with it using genetic algorithms and neural networks. Although simplistic, this approach was a success. Unfortunately, the generated programs were too simplistic to warrant further exploration. However, this success reinforced our notions needed a programming language with a simplified instruction set, leading us to Brainfuck.

Brainfuck (BF) is an esoteric programming language, which is widely considered a joke [21]. Although BF was not intended for actual programming, we found it to be highly amenable for use in automatic program generation when coupled with genetic algorithms. An example program, generated by our GA program generation system using BF, called BF-Programmer, is shown in Figure 1.

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

Figure 1. BF program that outputs "hello".

BF was created as a joke programming language and is nearly impossible to use meaningfully by humans, yet, it has several distinct advantages when used for automatic computer program generation using ML techniques. We discuss some of these below.

Turing Completeness A *Turing complete* programming language is theoretically capable of completing any (single taped Turing machine) programming task given an unlimited amount of time and memory [30]. In essence, a programming language with this characteristic is capable of implementations of a vast number of programming problems.

The BF programming language is Turing complete. Likewise, programs implemented with it, or in our case, generated with it, are theoretically capable of expressing all tasks that one might want to accomplished with computers.

Simplified Instruction Set The instruction set for BF consists of 8 programming instructions, as shown in Table 1. These instructions manipulate a memory "tape" of byte values, ranging from 0-255. BF works by applying increment and decrement operations to the current memory cell, while shifting the memory cell up and down the tape, as instructed by the program. The values at the current memory pointer can be input from the user or output to the terminal. Primitive instructions for looping are also available, offering the BF programming language a complete instruction set for creating software.

The simplified instruction set reduces the search space in which a target program code can be found. As computational devices improve in speed, larger problem spaces can be searched. However, on less powerful devices, the search space needs to be constrained. By limiting the programming instruction set to 8 unique characters, a genetic algorithm en-

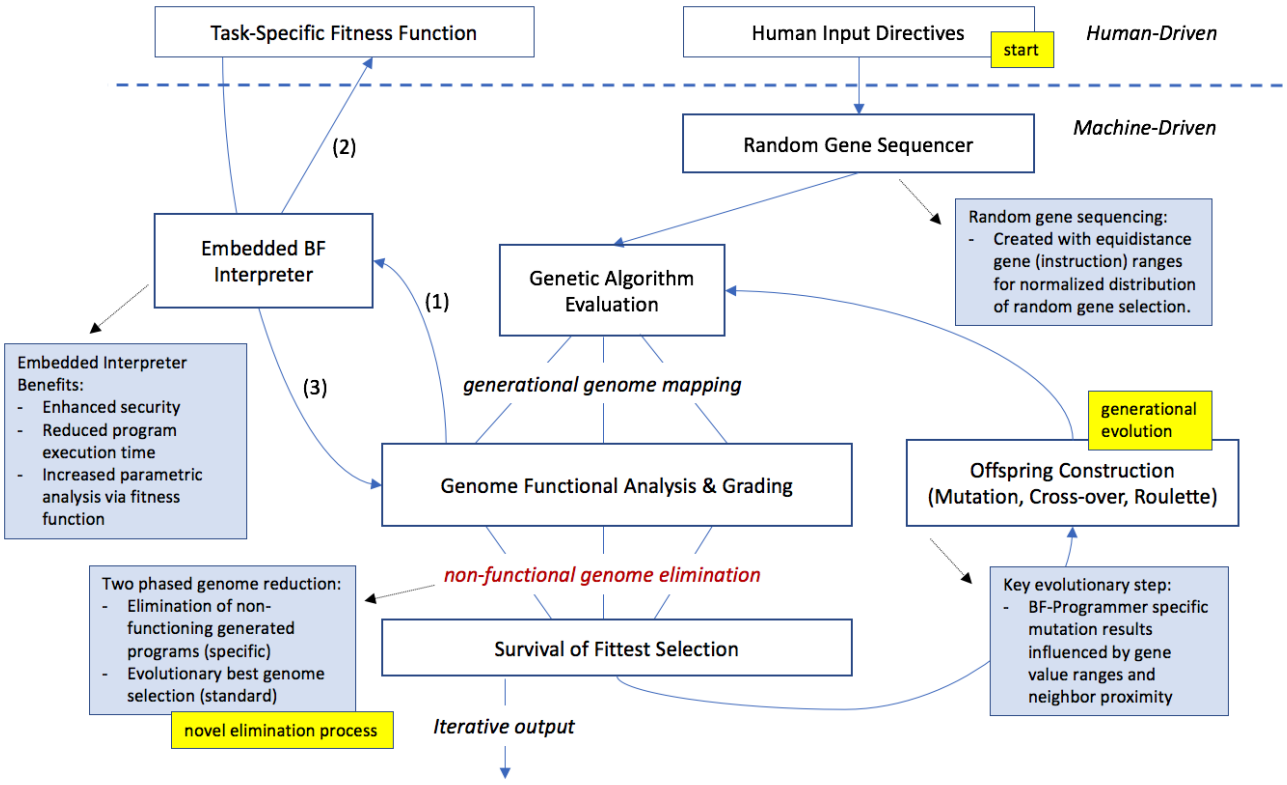

Figure 2. The BF-Programmer Software Architecture.

gine can run significantly faster to obtain an optimal fitness score than using a more verbose instruction set. [1]

3.1 The Benefits of a Self-Developed Interpreter

BF-Programmer must execute the programs that it generates to evaluate them. As such, because of the small size of the BF instruction set, we developed our own BF interpreter, which we then embedded within the GA engine, itself, resulting in a reduced total execution time of the code generation evaluation compared to when the GA system had to make external calls to a BF compiler to execute each child program.

Aside from reduced execution time, our self-developed interpreter provided us with security constraints, as child programs were executed within a controlled environment in the engine. Our fitness scoring also took advantage of the internal components of the interpreter, such as memory, instructions, and output. This was useful in calculating a more granular fitness score, allowing BF-Programmer to incrementally generate better child programs.

BF-Programmer's genetic algorithm engine represents each generated program's instructions as an array of double-precision floating point values, which, when considered as a unit, is its genome. An individual location within a given genome is called a *gene*. Each gene within a program's genome corresponds to a single instruction from the BF programming language as shown in Table 1.

[1] A reduced instruction set can drastically decrease the time it takes to generate a computer program using an ML algorithm. However, due to limited space we have opted to present program generation results instead of ML performance generation results.

Table 1. BF-Programmer Instruction Set and Gene Map

Instr	Gene Range	Operation
>	[0, 0.125]	Increment the pointer
<	(0.125, 0.25]	Decrement the pointer
+	(0.25, 0.375]	Increment the byte at the pointer
-	(0.375, 0.5]	Decrement the byte at the pointer
.	(0.5, 0.625]	Output the byte at the pointer
,	(0.625, 0.75]	Input a byte and store it at the ptr
[	(0.75, 0.875]	Jump to matching] if current 0
]	(0.875, 1.0]	Jump back to matching [unless 0

4. The Design of BF-Programmer

A high-level overview of the BF-Programmer software architecture, which utilizes all elements discussed within this section, is shown in Figure 2.

4.1 Genomes and Generations

To generate a software program using genetic algorithms, one must first create a genome. A *genome* is a set of genes that are grouped together as a single unit. For BF-Programmer, the genome is encoded as an array of floating point values, with fixed value ranges per unique instruction ranging between 0 and 1, as shown in the Gene Range column of Table 1.

Once a genome is created, it is converted to a corresponding program, executed, and the resulting program is assigned a fitness score based on the program's output. The closer a generated program comes to solving the provided task, the greater its fitness score and, the more likely it is to continue to the next evolutionary generation. At each generation epoch, BF-Programmer utilizes roulette selection, along with crossover and mutation, to create child programs that contain slight random perturbations, and potentially better, genomes than their parents for solving the target task.

Constructing a Genome Figure 3 demonstrates an example of constructing a genome from an array of floating point values. Each value range maps to a specific instruction in the BF programming language. Initially, these values are random (see the Random Gene Sequencer in Figure 2), resulting in generated programs that either won't function properly, throw errors, or simply fail [2]. However, one or two are bound to run and execute, at a minimum, some number of valid instructions. The more successful a program is at executing, the more likely it is to continue on and produce offspring with code that achieves more successful results.

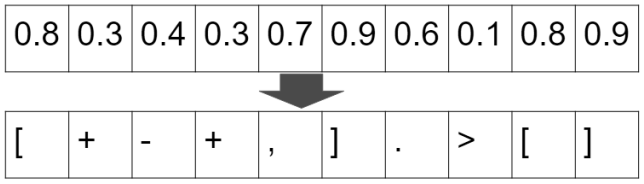

Figure 3. Decoding a genome as a BF program.

Crossover and Mutation To create offspring, a parent genome contributes part of its genes to the child, a process called *crossover*, as shown in Figure 4. In addition to inheriting programming instructions from its parent, each child can also experience *mutation*, which is the process of adding controlled, but random perturbation, to specific genes. This results in modified behavior of the value of a particular gene, resulting in a change to the resulting programming instruction, and thus, the overall program.

Crossover copies forward potentially beneficial parts of the parent, while mutation offers differing behaviors of instruction combinations, which may or may not, end up making the child programs more successful.

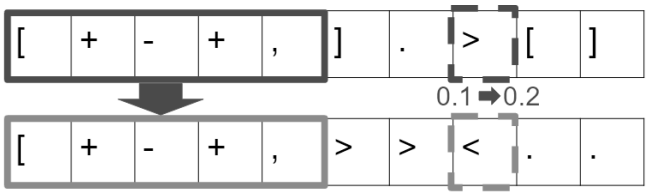

Figure 4. An example of crossover and mutation. The child genome inherits the first 5 instructions from its parent. One instruction is mutated.

Survival of the Fittest Executable programs are ranked according to how well they have performed. As shown in Figure 5, a particular program that has failed is often immediately removed from the pool of genomes. However, programs that succeed are carried forward to produce child programs.[3]

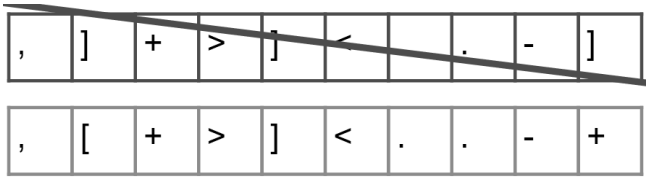

Figure 5. Programs are weighted by fitness, with the most successful used for child program generation.

4.2 The Fitness Function

A GA requires a fitness method to determine how well a generated program performs. This usually involves scoring the byte-level output of the generated program.[4] The score is calculated by analyzing the output of the program (if any) and subtracting its value from the desired result. An example of a byte-by-byte fitness scoring algorithm is shown in Figure 6.

```
fitness += 256 - Math.Abs(console[i] -
           targetString[i]);
```

Figure 6. A simple fitness function for testing the output of a program.

Designing a Fitness Test For any specific GA task, a fitness method is designed by a human, which is then utilized by the GA engine to evaluate all functional, resulting programs. This concept is similar to test-driven development, where unit tests are first created by a developer, prior to writing program methods. When all unit tests pass, a program

[2] Most initial programs in the gene pool fail immediately upon being executed. Others may result in endless loops. It is due to these reasons that exception handling and maximum iteration limits are imposed on the BF interpreter.

[3] In Figure 5, the bottom program is a valid running program that takes one byte for input, increments it, and then displays it twice as output.

[4] Fitness scoring may also include inspecting internal state values of the child program.

may be considered to be functionally correct. Likewise, a fitness test for a GA can be considered as a set of unit tests. In the case of BF-Programmer, a fitness test typically contains a suite of tests for varying scenarios, which guide the genome selection, preserving only programs that evaluate well on the test suite.

Fitness tests can include comparing the output bytes against the target. They can also include examining state properties, such as when input was requested, when output was performed, or even when a program's internal memory state is shifted or changed. This incremental flexibility allows for state-specific programs to be generated, and thus, more complex software to be automatically created.

5. Results

Using BF-Programmer, we were able to generate numerous simplistic programs. A complete listing of these programs, their associated program generation time, and the total number of evolutionary generations required to build them are shown in Table 2. These results were generated on an Intel Quad-Core i7 CPU, 2.7GHz, containing 16GB ram with an x64-based processor [2].

Table 2. BF-Programmer Results

Name	Duration (s)	Generations
hi	52	5,700
Hi!	7,644	1,219,400
hello	1,713	252,000
hello world	7,702	580,900
reddit	1,362	195,000
Keep Calm Keep Coding	944	21,400
I love all humans	36,000	6,057,200
hello {user}	1,793	42,800
Addition	2,698	92,400
Subtraction	4,305	177,900
Multiply x2	6,353	242,000
Multiply x3	5,165	87,200
XOR	2,095	146,400
Fibonacci	21,862	151,900
If/then conditionals	8,313	46,200
cats are evil	10,209	814,400
Bottles of Beer on the Wall	2,957	61,400
Reverse string	49	2,600
CSV parse	173	9,000
Extract in quotes	6,478	212,100
Extract in quotes 2	9,996	188,400
Trim left of quote	9,030	341,700
XML to JSON	6,866	820,900
Warning countdown	48	900

In this section, we highlight the details of some of the programs generated in Table 2 to discuss certain novelty we experienced in generated them.

5.1 Greetings

"Hello World" is usually one of the first programs human programmers create when they begin learning programming. As such, we found it fitting to guide BF-Programmer to learn some basic greetings for its early programs. Rather than starting with "Hello World", we first had BF-Programmer create a more simplistic program that simply output "hi." It was successfully after 5,700 generations and the generated code is shown in Figure 7.

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

Figure 7. Generated program: "hi"

The generated program fulfilled its requirement to output the target text, but interestingly included subsequent random characters, which contained parsing errors, including non-matching brackets. However, BF-Programmer's interpreter computes results until the program fails. In this manner, the syntax error (which is later on in the code, after a solution is reached) does not negatively impact its fitness score, and thus offers a working solution. In fact, the generated code can be executed in almost any BF interpreter as a valid working program (provided, warnings are ignored).

Next, we guided BF-Programmer to generate the famous "hello world" output which was successfully constructed after 580,900 generations and consists of the code shown in Figure 8.

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

Figure 8. Generated program: "hello world"

"I love all humans" As a humorous aside, we asked BF-Programmer to create the program to output "I love all humans," which was successfully generated after 6,057,200 generations. It consists of the code shown in Figure 9. The fitness method for this example includes a check on the output string length to ensure an exact matching output, without extraneous text.

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

Figure 9. Generated program: "I love all humans"

5.2 Input-Output Computations

As with most programs, we next guided BF-Programmer generate programs that performed computations based on user input. In such programs, the user provides some input

and the computer program then generates the appropriate output.

Reversing a String BF-Programmer was able to generate the program to reverse any string after only 2,600 generations. The generated code is shown in Figure 10.

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

Figure 10. Generated program for reversing a string.

When executed, the program prompts the user for input. The user then types one character at a time until a value of "0" is entered. A novelty of this program is that it is required to take variable size input first before performing the majority of its program logic. However, the program's internal memory state must manage the variable input, as the program must read all input first to locate the final character entered, which is the first character in the reversed string. The genetic algorithm was able to produce this logic automatically, based upon the fitness method.

Addition and Subtraction BF-Programmer was able to generate programs for addition after 92,400 generations (Figure 11) and subtraction after 177,900 generations.

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

Figure 11. Generated program for performing addition.

If-Then Conditionals with User Input Generating programs involving more complex programming logic, such as the ability to perform if-then decisions and actions, requires a more advanced type of fitness function. However, as described in Section 3.1, BF-Programmer's embedded interpreter provides significantly more access to program state than just its output, which is essential for generating a large variety of more complex programs.

For example, BF-Programmer was able to produce a program which prompts the user for input (e.g., 1, 2 or 3) and outputs text based on which value was entered, similar to selecting an option from a menu. By entering the value "1", the program would output "hi". Entering "2", resulted in the program output of "z". Entering "3", resulted in the output "bye". The program was generated in 446,200 generations.

The produced code was notably larger than previously generated programs, containing 650 instructions (although not all instructions are needed). The larger code was required, as the conditional branches are contained within individual blocks of the code.

5.3 Complexity in Fitness Functions

As the complexity of the target program grows, so too does the fitness function. After all, the fitness function needs to guide the engine in determining how well a particular child program matches the targeted solution. For conditionals and branching, successful program generation required more advanced techniques within the fitness function.

In particular, a check was needed to examine the interpreter's memory register (i.e., current data pointer via shift operations), where the distinct number memory registers being used by the program was counted, providing a bonus to fitness to favor more memory registers usage over less. This aided in inspiring diversity amongst child programs. Additionally, the instruction pointer used for each print command was recorded and weighed against the fitness score. A penalty was applied for reuse of the same print command. This helped to foster diversity and achieve a successful if-then result.

6. Optimizing Program Generation

When the programs BF-Programmer began generating produced longer output, we noticed the program generation time increased significantly. Furthermore, the need to extend BF-Programmer beyond the classic BF instruction set was deemed a necessity if we were to have it produce programs with more interesting features, such as file I/O and networking capabilities.

As such, we extended BF-Programmer to use an extended BF programming instruction set, which reduced code generation time and improved code compression due to an increased range of instruction specificity (i.e., fewer instructions to achieve the same result). However, a disadvantage of utilizing the extended instruction set is that the generated programs would be difficult to test in standard interpreters. In our case, BF-Programmer's internally developed interpreter was modified to support the extended instruction set, so this was not a practical obstacle.

6.1 BF Extended Type III

Several extensions of BF exist, which are suitable to decrease program generation time. Specifically, the speed-enhancing extension set, BF Extended Type III [8], offers several programming instructions that aid generation. These instructions include the ability to immediately set the value of a particular cell to a multiple of 16, also called "fast cell initializers". This aids in allowing a generated program to quickly reach displayable ASCII range characters for output, thus, decreasing the number of individual increment programming instructions that would normally be required.

In addition to key instructions taken from BF Extended Type III, we added several new instructions to support calling functions from within a BF program, allowing for increasingly complex programs to be generated.

Fibonacci Sequence With these extensions in place, BF-Programmer was able to generate a program to output the Fibonacci sequence up to 233 [5], which was was generated in

[5] 255 is the max value for a byte, with the next Fibonacci sequence value being 377.

```
,>,$[!>--$<<a>>]4]+,,-[-<+>]<+.$@
```

Figure 12. Generated program to output the Fibonacci sequence from two starting input values.

approximately six hours. The program prompts the user for input of the two starting values in the sequence. It then outputs the next digits in the Fibonacci sequence. The generated code for this is shown in Figure 12.

7. Related Work

Genetic algorithm based computer programming has already seen successes in the past, producing software with qualities similar to that of human-produced results. A key limitation to producing more advanced programs has been CPU speed, and thus, generation time. As demonstrated in "Human-competitive results produced by genetic programming" [14], J.R. Koza describes a variety of programs that have been developed by genetic algorithms, largely limited by Moore's law. However, as increasing CPU speed and capabilities advance, the complexity of automatically generated programs are bound to produce more impressive results.

Program synthesis with genetic programming has also provided solutions in hardware-based niche fields. In "Automated synthesis of analog electrical circuits by means of genetic programming" [15], the authors describe an automated process for the creation of analog circuits, using genetically evolved designs with evolutionary computation to produce circuit components that typically require human-level intelligence to design. The genetic programming approach used required the creation of a fitness method to guide the circuit design and resulting behavior. The metrics of the fitness method were tied to measurable outputs in order to granularly guide the resulting circuit design. This is similar to the process used in our research, where fitness methods are given specific human-designed and measurable constraints that guide the genetic algorithm in producing correct and successful programs.

One of the key components of our research is the usage of a minimal and esoteric programming language to limit the complexity of generated programs and foster an optimized generation process. This has been found useful in other areas of genetic programming as well, including the simulation of artificial life, as described in, "An Artificial Life Simulation Library Based on Genetic Algorithm, 3- Character Genetic Code and Biological Hierarchy" [17]. In a simulation library based on genetic algorithms and biological hierarchy, the system "Ragaraja" uses biological concepts to form an esoteric programming language, consisting of a set of 3-character instructions. In this manner, the system is able to simplify the genetic algorithm generation and mutation process by limiting the number of possible instruction combinations. This is similar in nature to our usage of an esoteric programming language, specifically for optimizing the generation time and limiting the complexity of generated programs to a constrained set of instructions.

The potential capabilities of ML-based computer programming has been known, although as of yet, not fully realized. This was due, in part, to a lacking of CPU power, which only recently has grown to the point of demonstrating powerful results, notably in the field of deep learning. Other factors remain a challenge to automated programming, as expressed in, "Open issues in genetic programming" [23], which describes the slow growth of genetic programming, despite the successes that it has achieved in various real-world domains. The BF-Programmer system demonstrates the rapid progress that can be achieved within the area of ML-based programming, provided a suitable programming language is utilized in the process.

8. Conclusion

Human-based computer programming is approaching an obsolete phase. With ever increasingly complex software and hardware integrations, the craft of software development will inevitably surpass the capabilities of humans. As that time approaches, it will become a necessity to have ML-driven systems available to relieve the burden of computer programming from their human counterparts.

The results presented in this paper, provide early notions about the power that ML-based approaches can bring to computer programming and that fully functional programs can indeed be automatically generated, provided they are supplied with formal input parameters and training data. While the initial set of programs generated by BF-Programmer are similar in complexity to a beginner human programmer, the range of generated programs need not be limited by time nor intellect. Rather, they are a function of fitness complexity and computational resources.

In order for ML-based systems to successfully generate programs, they will require specifically crafted programming languages, fostering genetic algorithm based automated programming. The current programming languages we use today, for humans, are insufficiently designed for ML-based program generation. The approach we use for typical program language creation needs to be abandoned and rethought when considering a future of ML-driven program generation. Only once this is done, can we begin to envision a new future of computer software development, driven by artificial intelligence based systems, with human creativity and design guiding the way.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016. URL `http://arxiv.org/abs/1605.08695`.

[2] K. Becker. Bf-programmer results. URL https://github.com/primaryobjects/AI-Programmer/tree/master/Results.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4): 471–523, Dec. 1985. ISSN 0360-0300. URL http://doi.acm.org/10.1145/6041.6042.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[5] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, Nov. 1978. ISSN 0001-0782. URL http://doi.acm.org/10.1145/359642.359655.

[6] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[7] P. Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. 2015.

[8] Esolangs.org. Bf extended type iii. URL https://esolangs.org/wiki/Extended_Brainfuck#Extended_Type_III.

[9] J. E. Gottschlich, M. P. Herlihy, G. A. Pokam, and J. G. Siek. Visualizing transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 159–170, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. URL http://doi.acm.org/10.1145/2370816.2370842.

[10] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982. ISSN 0362-1340. URL http://doi.acm.org/10.1145/872726.806987.

[11] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. ISBN 9780124104143, 9780124104945.

[12] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

[13] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual: For System-on-Chip Design*. Springer Publishing Company, Incorporated, 2007. ISBN 0387718184, 9780387718187.

[14] J. R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284, 2010.

[15] J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane, and F. Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on evolutionary computation*, 1(2):109–128, 1997.

[16] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[17] M. H. Ling. An artificial life simulation library based on genetic algorithm, 3-character genetic code and biological hierarchy. *The Python Papers*, 7:5, 2012.

[18] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pages 10:1–10:9, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4769-3. URL http://doi.acm.org/10.1145/2948618.2954331.

[19] Z. Michalewicz. *Genetic Algorithms Plus Data Structures Equals Evolution Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1994. ISBN 0387580905.

[20] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857.

[21] U. Müller. Brainfuck. URL https://en.wikipedia.org/wiki/Brainfuck.

[22] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428.

[23] M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, 2010.

[24] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. URL http://doi.acm.org/10.1145/1772954.1772958.

[25] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

[26] M. L. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009. ISBN 0123745144, 9780123745149.

[27] J. G. Siek and W. Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

[28] G. P. Stein, G. Hayun, E. Rushinek, and A. Shashua. A computer vision system on a chip: a case study from the automotive domain. *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 00: 130, 2005. ISSN 1063-6919.

[29] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, Apr. 2014. ISSN 1539-9087. URL http://doi.acm.org/10.1145/2584665.

[30] A. Turing. Turing completeness. URL https://en.wikipedia.org/wiki/Turing_completeness.

[31] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing,*

Networking, Storage and Analysis, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. URL http://doi.acm.org/10.1145/2503210.2503232.