Program Synthesis through Evolution of Augmenting Topologies

Alexander Ng[1] and **Stefan Stealey-Euchner**[2]

Advised by Dr. **Stephanie Forrest**[3] and Dr. **Cole Mathis**[4]

A thesis presented for the completion of Honors requirements
for the degree of Bachelor of Science

ASU Biodesign Institute
Center for Biocomputing, Security and Society
Barrett, the Honors College at Arizona State University
Presented May 8, 2025

[1]Undergraduate, School of Computing and Augmented Intelligence, School of Mathematical and Statistical Sciences, Arizona State University; an@asu.edu

[2]Undergraduate, School of Computing and Augmented Intelligence, School of Mathematical and Statistical Sciences, Arizona State University; stefanse@asu.edu

[3]Center Director and Professor, Biodesign Center for Biocomputing, Security and Society, Arizona State University; steph@asu.edu

[4]Assistant Professor, School of Complex Adaptive Systems, Biodesign Center for Biocomputing, Security and Society, Arizona State University; cole.mathis@asu.edu

Program Synthesis through Evolution of Augmenting Topologies

Alexander Ng and **Stefan Stealey-Euchner**

Abstract

Despite the impressive results demonstrated by modern neural network-based artificial intelligence, it often acts as a black box, with this low interpretability restricting its use in sensitive governmental and cybersecurity applications. We seek to combine these strong developments with more human-readable programs synthesized via genetic programming to create a generic model which can create highly interpretable solutions to a wide range of problems.

We explore the effectiveness of two techniques in evolutionary program synthesis through a novel WebAssembly bytecode-based format: (1) applying historical markings to construct intuitive crossover and speciation and (2) providing evolutionary scaffolding in the fitness function. The former builds upon the NeuroEvolution of Augmenting Topologies (NEAT) framework, which evolves artificial neural networks for reinforcement learning tasks, and the latter is inspired by scaffolding in natural evolution; we develop the analogous idea of partial test cases to smooth out the fitness function by providing intermediary goals (partial solutions) for individuals to reach.

These techniques allow us to use augmentation to build up minimal programs from scratch, eliminating extraneous genetic information. We present PEAS (Program Evolution through Augmenting Synthesis), a linear genetic algorithm that implements these techniques, and show that they provide significant benefits in synthesizing solutions as demonstrated on a simple programming task. In particular, we find that speciation is necessary for successful evolution, as the model rarely finds solutions with unspeciated crossover alone. We explore the potential, problems, and limitations of this type of model in extension to more general tasks.

Acknowledgments

We would like to thank Dr. Forrest, for giving us the targeted and pivotal advice in all areas of our project to which we owe much of its functionality; Dr. Mathis, for his field-linking suggestions which most drastically influenced our project's direction; Devansh Patel, for his keen observations and unforgiving analysis of our framework; and Dr. Pemma Reiter, for sharing with us her expertise in data and control flow graphs. We would also like to thank everyone at the Biodesign Center for Biocomputing, Security and Society for their guidance and hospitality, and the Fulton Undergraduate Research Initiative (FURI) program at ASU for financially supporting this research.

Contents

List of Figures

Chapter 1

Introduction

How can complex, intelligent behavior be created in computer programs without the direct, tailored intervention of human intelligence? This question has motivated machine learning research over the past half-century, as researchers boldly aim to create models that can solve problems in an automated manner, independently of human ideation, breaking free of the traditional paradigm in which human developers must create and implement a unique solution for each unique problem encountered.

Recent approaches have utilized artificial neural networks (ANNs) with great success, though not without drawbacks. The best-performing ANN models have billions of parameters, all of whose operations depend on the others' [1]. This leads directly to low interpretability (the ability of an outside observer to understand the inner workings of a model), which limits the use of ANNs in critical applications (governmental, medical, etc.) and in cybersecurity; as the models are functionally unverifiable, one cannot be sure that their behavior will remain predictable and correct in all situations [2].

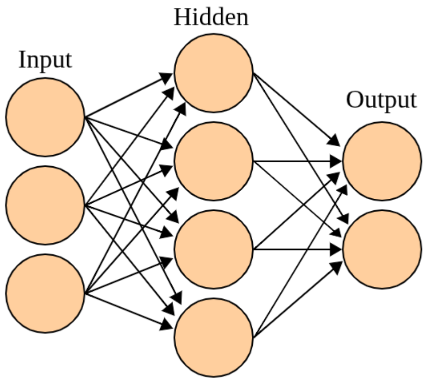


Figure 1.1: An artificial neural network with one hidden layer [3]. Modern ANNs may have hundreds of hidden layers and millions or billions of parameters.

An alternate approach to algorithmic problem-solving is through Evolutionary Algorithms (EAs), a class of algorithms which simulate aspects of biological evolution (typically mutation, reproduction, genetic recombination, and selection, among others) in order to approach solutions “naturally,” and without necessitating the intercession of an intelligent designer. A germane type of EA is genetic programming (GP), in which each individual in the population represents an executable computer program. GP is applicable in scenarios where the “problem” to be solved is to create a computer program that meets certain specifications; GP will be run on the problem constraints, and, once an individual in the population is deemed satisfactory, that individual will be returned as the solution. In order to measure the suitability of individuals as solutions, as well as to compare individuals to each other, each GP problem (and, in general, each EA problem) is endowed with a fitness function, which takes a potential solution as its argument and returns a score representing the suitability of its argument as a solution. Because the final output of a GP run is a computer program, it can be far more interpretable to the human eye than a trained ANN.

Though these approaches seem to approach problem-solving in highly distinct manners, the application of techniques from EAs, and GP specifically, to ANNs (called “neuroevolution” or NE) has proven successful in some applications. Such neuroevolution was typically performed on the weights of the ANN, in order to “tune” them to their optimal values, and preliminary research had been done throughout the 1990s on NE of both the weights and structure of the network itself. In a landmark paper, Stanley and Miikkulainen developed NeuroEvolution of Augmenting Topologies (NEAT), which extended viable neuroevolution to such an algorithm which evolved both the parameters of the network and the network topology itself, allowing for a much richer search space of potential solutions than traditional NE. The authors showed that NEAT could far outperform traditional fixed-topology networks (with or without traditional NE) at challenging reinforcement learning tasks [4].

In this paper, we seek to return the favor by showing that techniques from NE, specifically from NEAT, can be incorporated with traditional GP to rapidly achieve highly accurate solutions that maintain interpretability. We present a generic model, “PEAS” (Program Evolution through Augmenting Synthesis), leveraging a novel WebAssembly bytecode format to synthesize programs with a focus on augmenting mutations, and its results when applied to some simple GP problems. We hope that this model (and future extensions) can serve as a way to evolve programs which can be used in a wider array of applications than traditional ANNs, especially in cybersecurity applications.

Chapter 2

Background

2.1 Genetic Programming for Program Synthesis

The archetypal GP implements at least the following operations on individuals in its population: selection, genetic recombination (often called crossover), and mutation [5]. Many also include an elitism factor, applied before selection. In such an implementation, the population is quantized into discrete generations, with each individual's life cycle occurring for the same length (exactly one generation) [6]. A single GP generation follows, in order: elitism (if present), selection, crossover, then mutation. At the end of this process, all previous individuals (except for elites) are culled, and the new, post-mutation individuals (along with elites) make up the next generation.

In order to promote genetic diversity and prevent stagnation of the population, a high rate of genetic turnover/augmentation is generally considered desirable. However, this can lead to accidental destruction of quality genetic material if negative mutations happen to affect high-fitness individuals. Additionally, GP is generally used in scenarios where the ultimate goal is to evolve a single individual with high enough fitness; therefore, it is imperative that direct routes toward high fitness be maintained, rather than risking damaging the best genomes by placing too high a priority on genetic turnover. This can be viewed as an exploration/exploitation tradeoff, where exploration is prioritized to aggressively search for new high fitness innovations, while the best few individuals are exploited to lock in beneficial innovations. Experimental results have shown that a small elitism percentage (5%) contributes to lower mean program size in common GP systems, meaning that models with elitism can achieve similar results to those without, while using less total genetic material [7].

Tournament selection is used nearly ubiquitously in selection of EAs for its simplicity and ease of implementation, while retaining good results compared to other selection schemes [8]. In the standard tournament selection, as many "tournaments" are run as there are desired individuals to be chosen for crossover. Within each tournament, a random subset of the population is chosen, and the winner (most fit individual) is selected for crossover. The primary control parameter is the size of the tournament (larger relative to the population means stronger selection, but is also more computationally expensive) [8].

Crossover is, in most GP implementations, the sole method of genetic transfer between individuals, and can be thought of as akin to sexual reproduction in biology (where a set of two or more parents' genomes are combined to create a single child genome). An important consideration while crossing over two genomes is whether or not they are "compatible"—that is, whether or not their offspring will be valid executable programs. If each genome is represented as a Lisp-like symbolic expression (S-expression), this consideration is relatively minimal, as this formulation can be directly represented as a tree. This allows subtrees in each parent with the same type of root node (instruction or variable) to simply be swapped with each other, and be guaranteed of validity since both subtrees will evaluate to the same type of node [6]. Despite the convenience of this type of genetic representation for compatibility checking, other genetic representations with different advantages have been developed, which we also take inspiration from (see Sections 2.4 and 4.1 for further discussion).

Implementing mutation involves similar considerations as crossover, though only constraints on the single genome must be considered. In the canonical GP, a node can only be mutated to one of the same operation type as it (i.e. addition to subtraction) [5]. Even a single mutation can influence the behavior of the resulting program massively, and generally in a negative way, so regular mutation of genomes is likely to decrease average fitness [9]. This belies mutation's impressive power to increase the fitness of the occasional successful genome, giving further evidence for the importance of elitism in removing the fittest individuals from the filter before they have a chance to be irreparably damaged. It is unclear to what degree mutation influences overall gene distribution in genetic algorithms, but there is evidence that the specific problem in question has a significant influence on which genetic material generation method (crossover vs. mutation) is more effective [10].

2.2 Scaffolding in Evolution by Natural Selection

Traditional Darwinian explanations of evolution by natural selection suffer from a "circularity problem": one must explain the rise of the mechanism of natural selection without utilizing the idea itself [11]. Any sufficiently large evolutionary change can be viewed as an extension of this problem, as such a large change may seem unreachable from a population's current state. Evolutionary scaffolding provides an explanation for how evolution can circumvent some such problems; rather than jumping straight to the full development, intermediate levels of the development are achieved, one at a time, like stepping stones, until the full innovation is attained.

We believe that this idea is highly applicable in GP, since many problems' fitness functions are based on the proportion of test cases passed. These test cases are assumedly based on the full desired functionality of the solution, which, like the evolutionary innovation, may be functionally unachievable via small mutations, perhaps due to intermediate steps having lower fitness than both the final solution and initial position. Artificially boosting the fitness of "intermediate stages" of the solution could promote development of these "partial solutions," which can then be enhanced to attain the full solution via small alterations. This approach has recently been attempted in GP with promising results, which we aim to extend (see Section 3.2 for further discussion) [12].

2.3 NEAT and Its Derivatives

Stanley and Miikkulainen's NEAT framework made several major contributions to the field of neuroevolution. In particular, the authors found a solution to the competing conventions problem, a method for protecting innovation through speciation, and a strategy to reliably minimize search space dimension [4].

"Competing conventions" refers to a scenario where two neural networks can produce the same output for any input combination yet have distinct enough representations that a crossover of the two networks is likely to damage the functionality of the child [13]. In other words, the networks are functionality equivalent, but produce offspring missing key information to that functionality due to differences in the networks' phenotypic representations [4].

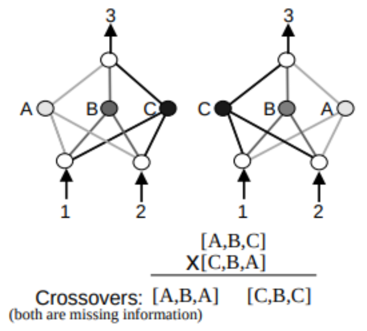


Figure 2.1: Both networks compute the same function, but either possible single-point genetic recombination produces a genome lacking its parents' full functionality [4].

Due to this problem, crossover algorithms for ANNs had to implement expensive and convoluted topological analysis to ensure the viability of the child, essentially by determining which genes from each parent "match up" with those of the other. NEAT solves this problem via the use of "historical markings" to chronologically track all of the unique genes in a population. To accomplish this, NEAT maintains a global "innovation number"; whenever a new unique gene is created, the innovation number is incremented, and the gene marked with that value. Now, finding comparable genes between two networks is nearly trivial, as their genomes can simply be ordered by their genes' innovation numbers, with corresponding markings matched, and the genetic information in between matching genes copied into the child's genome. This eliminates the need for costly topological analysis, while ensuring that the output of the crossover will be a valid child highly likely to retain the relevant functionality of its parents.

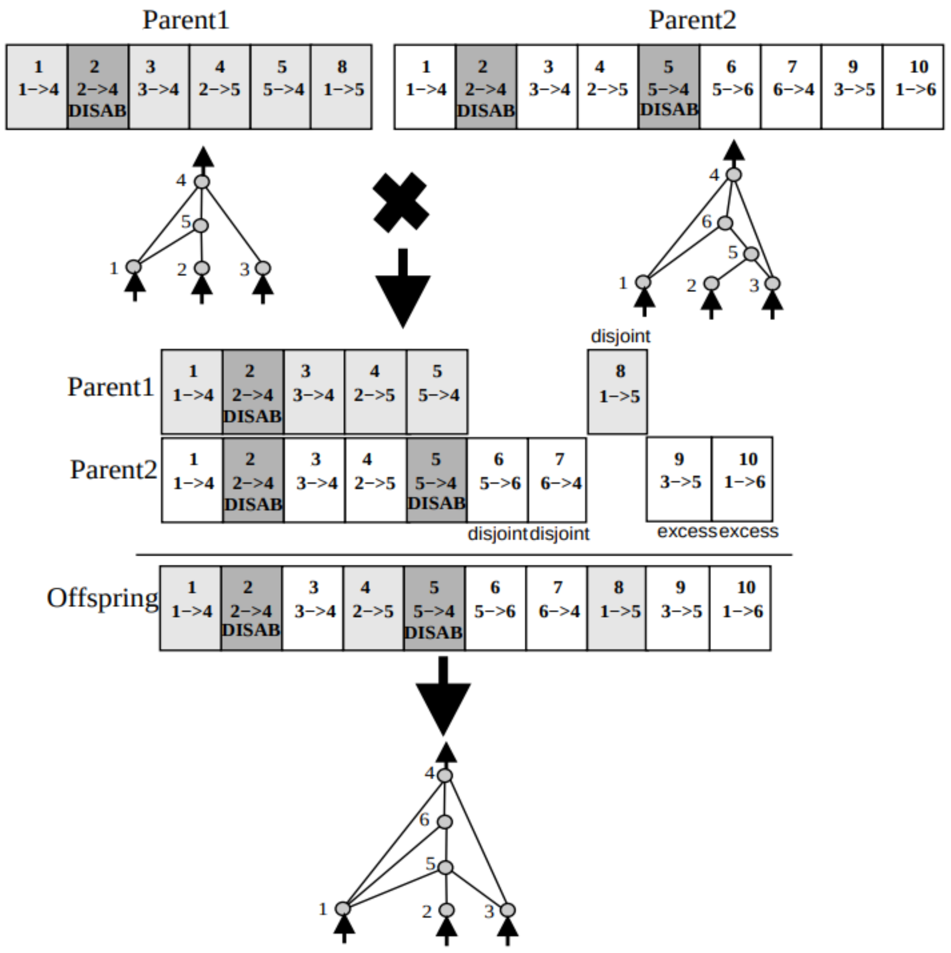


Figure 2.2: Though the crossover of the two individuals can be viewed in terms of their phenotypes (graphs), the operation itself can be performed with only the genomes by matching the innovation numbers [4].

Adding additional structure to an ANN typically reduces fitness at first, which can be problematic since it creates a negative feedback loop discouraging radical and exploratory structural innovations. This in turn could leave a genome stuck in a local optimum without a way to break through to a higher fitness level [14]. In the mid 1990s, Potter and De Jong developed speciation in ANNs as a technique for accelerating the development of decomposable problems by first developing solutions for subproblems, then combining them. NEAT brought this technique to NE with the purpose of protecting novel innovations [15]. Through speciation, innovations with no immediate fitness gain, which may otherwise be quickly discarded, can be allowed to develop in a niche and grow mature structure before needing to compete with individuals from the wider population.

Most pre-NEAT neuroevolutionary models, because they do not utilize speciation or other structural innovation protecting methods, require their solutions to be initialized with large random topologies, so as to endow each population with some baseline diversity and established structure to build upon. However, since these initial topologies are random, they are highly likely to include large amounts of extraneous genetic information that will never become useful for the actual solution [4]. At a fundamental level, this random structure is detrimental because it has, by definition, been subjected to no evaluation or even heuristic for applicability to the problem at hand—the NEAT paradigm functions under the idea that truly useful structure in an evolutionary system should arise naturally in response to innovations borne of positive or negative feedback from the environment. However, since NEAT's speciation allows it to be confident that innovative structures can arise of their own accord, NEAT can start with an absolutely minimal structure, meaning only useful additional structure will develop. This not only ensures dimension reduction relative to random-origin topology models, and thus a simpler overall topology on average, but also that only useful innovations exist in the population, naturally begetting further innovations.

NEAT has been extended to the use of heterogeneous activation functions—that is, the activation function of each neuron is included within the genome. This shifts some of the importance from the edges to the nodes. The creators of heterogeneous activation NEAT (HA-NEAT) simply add the activation function as an additional aspect to the genetic representation of each node [16]. They vary from a select few typical activation functions such as sigmoid and ReLU, and are changed through a mutation. To use the historical markings introduced by NEAT, HA-NEAT introduces it as a new node, disabling the old one, and re-wiring all the current edges as new edges. This mutation has a wide variability in effect, so the large change in genome is justified. It can be destructive, but it can also unlock innovations that allow for higher fitness—the authors demonstrated significant reductions in network size when utilizing HA-NEAT compared to base NEAT, as the network can choose well-performing activation functions post facto [16].

Research on Weight Agnostic Neural Networks (architectures without explicit weight training) demonstrates the possibility of having an effective neural net despite all the weights being the same [17]. This approach, based on NEAT, allows the neural net to evolve its topology, while maintaining a constant weight across all edges. This emphasizes the structure and functions of the neural net, rather than the values of the weights. This technique has been used to evolve smaller and simpler neural nets [17].

The graphs representing these neural nets can be seen as a data flow graph of a program. While neural networks are typically visualized as graphs, in reality they are run as programs. The topology of the graph acts as the structure for the program: how the data flows between activation functions, manipulated by weights. We wondered: could we apply the above techniques to a more general program as opposed to one constrained by the operations of the neural net?

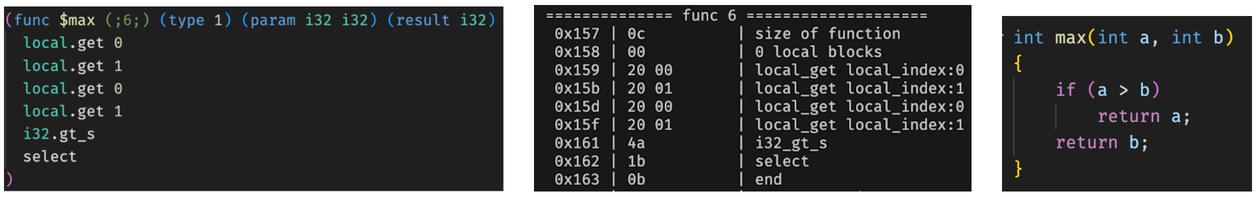
2.4 Linear and Stack-based Genetic Programming

While traditional GP inherits many aspects of the functional programming paradigm (à la Lisp or Haskell), naturally represented in a graph format, Linear GP represents an evolutionary program (individual) as a sequence of instructions executed in order, usually from an imperative programming language or assembly language. All operations use ”registers” (constants and memory variables), accepting them as parameters and assigning the result to another register [18]. Advantages of Linear GP include its tendencies to be more human-readable and compact than traditional GP representations.

Push is a “stack-based” programming language created by Lee Spector in the early 2000s, specifically for use in GP. In this implementation, this means that each data type in the language (e.g. int, float, etc.) has its own stack, and the program code itself is treated as data, making stack manipulation part of the language itself, a process specified as “autoconstructive evolution” by Spector [19]. Traditional GP often struggles with syntactic correctness during mutation, crossover, and other genetic operations, since the traditional genomes are typically quite complicated, representing complex graphs. PushGP’s simple structure allows it to bypass many of these concerns, as ensuring a stack operation is valid (will not result in any syntax errors) is much easier than doing the same for a corresponding graph operation [19]. PushGP then implements standard GP techniques in this stack-based format, while not compromising its ability to actually solve problems (Push is Turing-complete, and also supports several high-level constructs such as recursion and named subroutiunes).

2.5 WebAssembly

WebAssembly (Wasm) was designed in the late 2010s as a portable interface format for websites, à la the Java bytecode format but for primary use within web browsers (though with portability to many other environments). Wasm is a bytecode format for sandboxed programs, running on a stack machine: instructions take inputs from the stack, and push their outputs onto it [20]. In addition to its native bytecode, WebAssembly also provides a text format, WebAssembly text (Wat), which decodes the bytecode into a highly readable Lisp-like chain of S-expressions. Using readily available (de)compilers, Wasm can easily be converted to and from high-level languages such as C and Rust [21].



```
(func $max (;6;) (type 1) (param i32 i32) (result i32)
  local.get 0
  local.get 1
  local.get 0
  local.get 1
  i32.gt_s
  select
)
```

(a) Wat S-expression

```
============== func 6 ====================
 0x157 | 0c          | size of function
 0x158 | 00          | 0 local blocks
 0x159 | 20 00       | local_get local_index:0
 0x15b | 20 01       | local_get local_index:1
 0x15d | 20 00       | local_get local_index:0
 0x15f | 20 01       | local_get local_index:1
 0x161 | 4a          | i32_gt_s
 0x162 | 1b          | select
 0x163 | 0b          | end
```

(b) Annotated memory

```
int max(int a, int b)
{
    if (a > b)
        return a;
    return b;
}
```

(c) Decompiled C

Figure 2.3: Three representations of the same simple Wasm program

Advantages of Wasm’s stack machine include its relatively “safe” control flow, with no goto and deterministic linear execution, as well as its sandboxed execution as a disjoint module from its surrounding environment. Furthermore, Wasm’s naturally constrained instruction set, stack-based execution, and simple type system make it highly amenable to the implementation of techniques from linear GP, especially PushGP, given its construction using a stack based language. Wasm's linear execution and small instruction set provide an intrinsically simplistic base from which a GP algorithm can be created, while its powerful instructions can still provide a rich evolutionary landscape with the theoretical capacity for high correspondence with human-coded programs in imperative or functional languages.

Another of Wasm's advantages is the large number of robust Rust libraries built to interact with it. Especially relevant to our project are "wasm-encoder", "wasmparser", and "wasmtime" (technically a Wasm runtime implemented in Rust). Wasm-encoder provides a way to build a from-scratch Wasm binary entirely from Rust code, without necessitating any other intra-Wasm interaction [22]. This allows PEAS to operate entirely in Rust, despite creating and executing agents made up of Wasm bytecode. The wasmparser crate provides the inverse functionality, taking in a raw Wasm binary and converting it to an internal Rust representation which we can specify [23]. This completes PEAS's toolbox of Wasm interactions, allowing it to create and parse Wasm binaries without needing to utilize Wat or any other intermediaries. Finally, in order to compile and execute these created Wasm modules, PEAS utilizes the wasmtime runtime, a standalone runtime which enables modules to be sandboxed outside of any particular context, such as a web browser [24]. Additionally, wasmtime implements the WebAssembly System Interface, which gives PEAS the ability to control access to system resources and host functions (i.e. I/O, files, provided functions) through a secure API.

Chapter 3

Motivation

3.1 Essential Features

Inspired by the interpretability problem in modern machine learning and our goal of synthesizing complex behavior "from scratch," we concluded upon reviewing the literature that we wanted to create a model with the following aspects.

It should be generic, able to solve a wide range of problems with minimal problem-specific hard-coding. All that should be necessary to solve a problem is a description of that problem's input/output format and a fitness function. In this aspect, we were inspired by prior research in automated code repair, such as the GenProg method, which can detect and fix bugs in developed code without the need for any metadata like specifications or comments [25].

It should create "minimal" solutions; rather than solutions being based on some arbitrary topology or predefined level of complexity, they should be decided by the fitness landscape generated from the affiliated fitness function. In this aspect, we draw heavily from NEAT, which broke ground on the regular creation of optimal topologies.

It should have diverse evolutionary capabilities. In other words, it should have as few evolutionary constraints as possible while simultaneously ensuring all genomes are valid programs. This encourages genetic diversity while reducing the likelihood of becoming stuck in suboptimal local maxima.

It should be interpretable; solutions should be readable and understandable by a human programmer. This is where we make use of Wasm's unique bytecode format and its translatability to a text-based format.

It should be small and fast, not needing large networks or graphs, and maximizing efficiency, so as to be portable and widely applicable. Here we also make use of Wasm's compact, stack-based format, as well as the concept of historical markings from NEAT. These allow us to endow each genome with information that otherwise would have been included in a graph or network structure within the structure of the linear stack itself, saving on space and metadata while increasing speed.

With these goals in mind, especially trying to create genetically diverse and powerful solutions, and since we wanted to use a true linear genome, rather than a graph or network, we needed to consider how we would perform the genetic operations. Here, we applied the key idea of historical markings from NEAT, though modified for our linear framework. In NEAT, one of the purposes of the historical markings is giving the neural network graph an ordering, so edges can be matched. That genome consists of an unordered list of edges. That is, two genomes will be functionally identical if the ordering of genes is shuffled, since each gene represents the presence/activation of an edge in the graph (see Section 4.4.2 for further discussion). We also considered which mutations would be most advantageous to endow our model with, and which could potentially lead to invalid programs if handled incorrectly, something we considered in tandem with the overall format of our genomes (see Section 4.4.3 for further discussion).

3.2 Testing Protocols

Next, we needed to formulate a suitable test problem for our model. It needed to be simple, so we could manually review progress and inspect solutions, but complex enough that our model would need to exhibit some level of "intelligence" to solve it. With the guidance of Helmuth and Spector's program synthesis benchmark suite, we devised the problem "sum3": given an input of three numbers, output one number such that the output is the sum of the inputs [26]. We chose the fitness function to be the proportion of test cases passed for a large number of randomly generated test cases. In order to solve this problem, a model must make use of multiple variables and manipulate them using arithmetic, each of which involves multiple mutations. Furthermore, there are risks of becoming caught in local extrema, such as if a nonzero constant is introduced as an addend or if a more complicated function of the inputs is introduced that happens to solve a nonnegligible subset of the test cases. However, the problem does not require memory access or complex functions to solve.

We also wished to test how our model behaved in different "environments," which we accomplished through "degenerate" test cases. Preliminary results of running our model on sum3 with randomly generated test cases revealed a common pattern: a trial would have large clusters of low fitness individuals for many generations, then suddenly generate a full solution (or, in some runs, a failure to ever reach a solution). We noticed that, often, the few test cases such solutions originally solved were those where one of the three inputs happened to be equal to 0. We thus saw an opportunity to test if our model could take advantage of a smoothing out of the fitness function, via the deliberate introduction of some of these "degenerate" test cases amongst the completely randomly generated ones.

A solution that successfully utilizes this smoothing can be viewed as the culmination of partial solutions in the following way. If an individual is able to pass a single randomly generated test case with all nonzero inputs, it is likely able to pass any such test case. However, an individual could certainly pass a degenerate test case, such as one where two inputs are equal to zero, and only the third randomly generated, and fail at every fully random test case. Notably, though, we could consider the second individual as a partial solution of the problem, and we would also expect it to be much simpler than a full solution, and thus easier to evolve.

Thus, by the introduction of several levels of degenerate test cases (some test cases with zero as one or two of the three inputs) amongst fully randomly generated ones, we create a hierarchy of problems and solutions to them: the lowest level can only solve the most degenerate test cases; the next level, those and more; and the final level, the full solution, can solve all test cases. We can view these levels of degeneracy as discerning between partial solutions. Since the lower levels of the hierarchy should be easier to evolve, our model can "build up" a solution to the full problem by first implementing solutions to all lower levels of partial problem, rather than jumping straight to the final conclusion. We hypothesized that this would help to smooth out the clustering effect we initially observed, where solutions were "all or nothing".

Chapter 4

Implementation

4.1 Genetic Representation

We combine the aforementioned ideas to formulate a linear genome of Wasm instructions. Each gene represents an instruction in the program, with the full genome representing its code. The evaluation process simply translates these instructions into bytecode. Like in NEAT, genes are marked with unique innovation numbers as historical markings.

One consequence of this simple linear genome is that gene behavior is highly dependent on locality, with the positioning of the gene in the genome encoding much meaning in and of itself. Wasm's arithmetic instructions are stack-based binary operations, popping two arguments and pushing the result. Consequentially, our linear code is highly dependent on ordering. Two individuals could have the same genes and both be valid, but have different permutations of those genes, leading to different behavior. This presents limitations on how we can perform crossover and the types of mutations our genomes can undergo (see Sections 4.4.2-3).

Despite being linear, we can visualize the genome as a data flow graph (DFG), with sources being constants and input arguments, and the single sink being the return value of the function. Intermediary nodes are operations, where incoming edges are data from the stack at that point in the program execution and the outgoing edge is the value returned to the stack. However, because Wasm instructions are low-level operations, we cannot actually use a graph format directly as if it were a neural net, whose nodes can take any number of inputs.

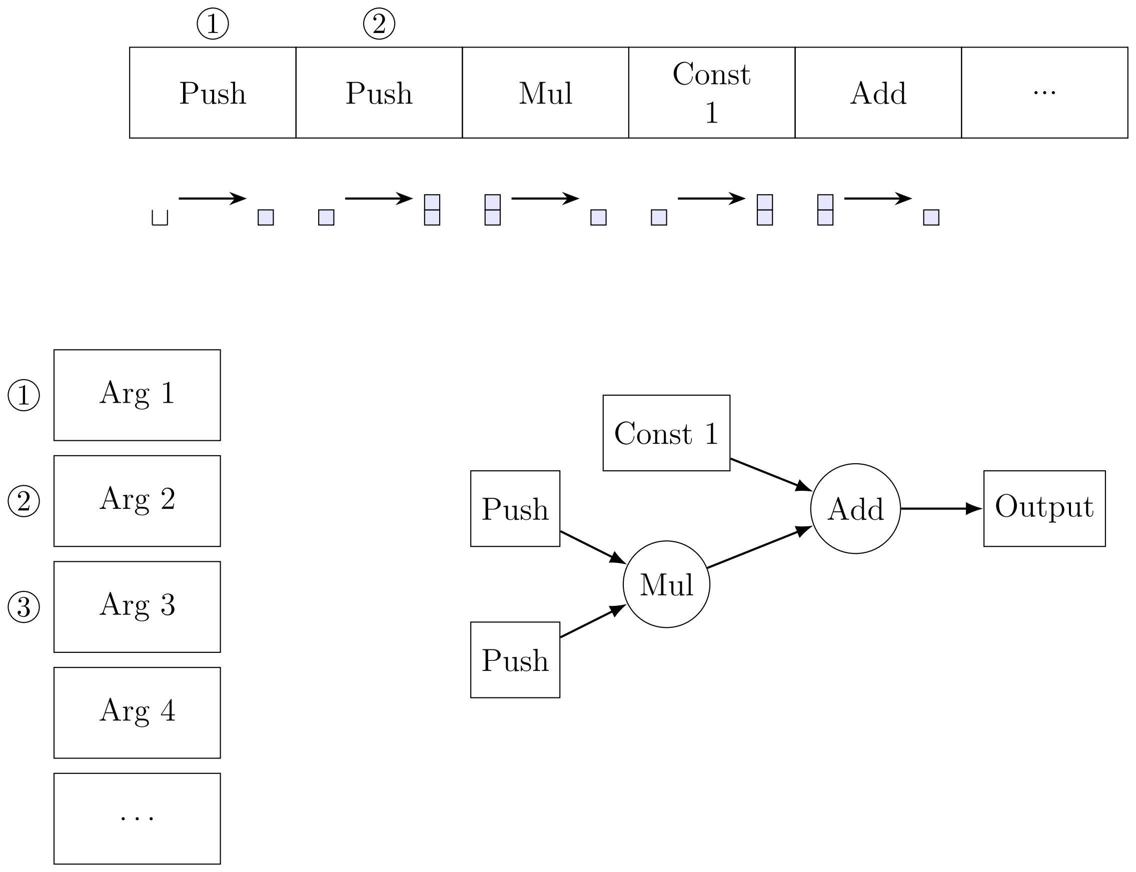


Figure 4.1: Counterclockwise from top: An example linear PEAS genome and the height of the stack after each operation; the corresponding local Wasm stack; and a DFG representation of the genome

4.2 Using Historical Markings in a Linear Genome

NEAT's historical markings provide powerful capability for defining crossover and speciation for neural nets. However, rather than a graph, we have linear genomes of varying sizes. We can't naïvely merge the two, since the stack height must be more carefully managed.

To maintain the chronologicity of the historical markings during crossover, we must maintain the relative order of genes. If two genomes contain the same gene (same innovation number), this implies that they have a common ancestor which introduced it. Ergo, we must ensure that, in our implementation of crossover, no two genes swap places in the child. That is, if both genomes have genes marked with innovation numbers 3 and 5, then if 3 is before 5 in one genome, then it must also be before 5 in the other. If we don't use mutations that swap the order of genes, we can be certain that this property is maintained. Crossover of a sequence of genes must also be done in such a way that the sequence remains "valid" in its operation of the stack. If a sequence of instructions is to be transplanted from one parent to another, it must net the same stack operation as the sequence it replaced. A sequence which pops two values and pushes one can only be replaced by another which does the same. Thus, a typical 1-point crossover would require statically analyzing the stack height for a match, which would be difficult as the genomes may be different sizes.

Historical markings also allow a compatibility distance function to be defined, since they give us an ordering for the genes, which allows for speciation. The function maps a pair of genomes to a float, where larger outputs indicate a greater difference between the two genomes. This “difference” is computed as a linear combination of the disjoint and excess genes of the two genomes, normalized for genome length. We define our function δ similarly to NEAT’s, with coefficients c_1 and c_2, and E and D being the number of excess and disjoint genes, respectively:

$$\delta = \frac{c_1 E + c_2 D}{N}$$

However, this metric may not be as granular as NEAT’s since we can’t incorporate difference in edge weights as a factor.

4.3 Partial Test Cases as Evolutionary Scaffolding

In certain problems, inputs can be used to reduce the problem to an easier sub-problem. In sum3 and sum4, this is accomplished by setting some inputs to zero. Suppose, when solving sum3, an individual adds the first two numbers but ignores the third. This individual would fail the full test case, but would pass a partial test case where the third input is fixed at zero. With this modified fitness function, the individuals that obtain partial solutions will be promoted, establishing the evolutionary scaffolding. We use the proportions of different types of partial test case as a parameter to the algorithm (see Section 3.2 for extended discussion).

4.4 The Parts of a PEAS Epoch

As in most traditional GP algorithms, PEAS quantizes populations into discrete generations (aka epochs), each with a defined set of steps used to generate the next generation from the current population of individuals.

4.4.1 Elitism and Selection

At the start of each epoch, for each species, PEAS calculates the number of elites which will come from that species, “elitism count”, as the product of the number of individuals in that species and the elitism rate, fixed in PEAS at 0.05 (see Section 2.1 for further discussion). The individuals in the species are then sorted in decreasing order of fitness, and the best “elitism count” number of individuals are reserved to be appended to the population of the current generation after mutation of the current generation is complete. If speciation is disabled, elites are chosen in the same manner, but from the whole population of the previous generation without any stratification.

Like elitism, selection is performed separately for each species. For each species, the number of selected individuals (survivors) is computed as the product of the number of individuals in that species and the selection rate, fixed in PEAS at 0.6 (see Chapters 2 and 5 for further discussion). A standard tournament selection with replacement is then run with tournament size 2 and choice probability 0.9 (see Chapters 2 and 5 for discussion on hyperparameter fine-tuning) until the decided number of survivors have been selected. The survivors are then passed on to crossover as the parents of the next generation, while the remaining individuals are discarded. As in elitism, if speciation is disabled, selection is simply run on the whole population, and the survivors are passed on to crossover without any stratification.

4.4.2 Crossover

If speciation is enabled, the number of children to be produced is computed as the product of the number of remaining spots in the species (the number of individuals allotted to the species at the end of the last epoch minus the number of elites chosen in the current epoch) and the crossover rate, fixed in PEAS at 0.8 (see Chapters 2 and 5 for further discussion). Any remaining spots allotted to the species not filled by elites or children will be filled with randomly chosen clones of parents, while all parents will be culled after crossover is complete. This serves to both maintain genetic diversity (by cloning parents of the past generation rather than the children themselves, which are already part of the new generation) while also maintaining high fitness and limiting overexploration, since the parents by definition have already passed the new generation's selection filter. The group of children (and potentially cloned parents) are then passed on to mutation.

Pairs of parents are chosen at random with replacement (meaning a child can have identical parents) from the survivors of selection, with each pair crossed over to make a single child, until the allotted child count is reached. Because any pair of survivors can be parents, it is crucial that the crossover algorithm always produces a viable child from any two genomes.

As with elitism and selection, if speciation is disabled, the same crossover process will be run on the survivors (which have come from the whole population), with the children passed on to mutation and the parents discarded.

4.4.3 Mutation and Evaluation

Each genome passed to mutation undergoes a single mutation chosen at random from the possible mutations. PEAS implements two types of mutations: "AddOperation" and "ReplaceSource". Once mutated, if speciation is enabled, all mutated individuals from each species are combined along with their elites to form the next generation's population. These are then passed to speciation to be divided into species. If speciation is disabled, the mutants immediately become the population of the next generation.

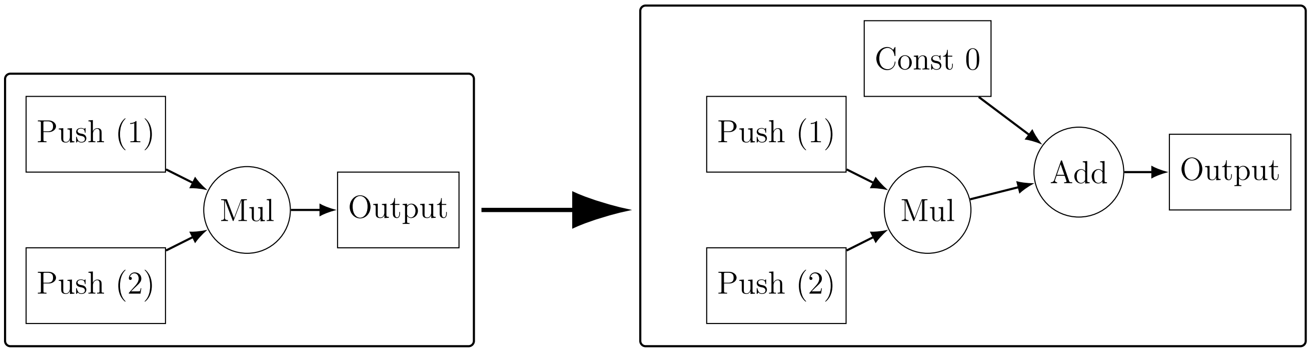


Figure 4.2: A graphical example of a neutral AddOperation mutation

Because the operations are stack based, basic instructions can be roughly categorized into "sources"/"roots" and regular operations. Sources add something to the stack, like pushing a local variable or a constant. Regular operations take their arguments by popping from the stack and then pushing the result.

In order to add a binary operation, we must also push an extra source in order to maintain stack height. In this way, we can add an operation at any point. We decided to make this mutation, "AddOperation", neutral by making the extra source pushed an identity with respect to the operation. This way, AddOperation has no immediate effect, but increases genetic potential to be exploited by crossover or other mutations later.

In line with NEAT's design of augmentation only with simple mutations, we only use simple additive mutations as well. NEAT has "add connection" and "add node". While we don't have an analogue for "add connection", we can imagine our "AddOperation" acting similarly to "add node". It inserts an operation in the data flow, initially passing the data through transparently. Later ReplaceSource mutations will activate them by diverting it from the identity to some variable.

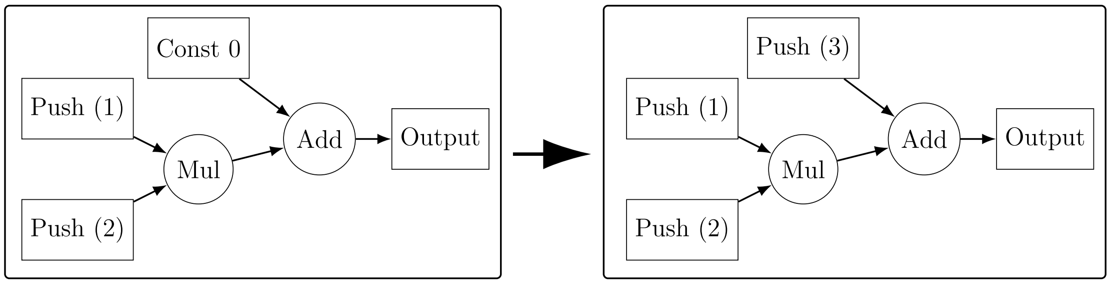


Figure 4.3: A graphical example of a ReplaceSource mutation

After mutation and speciation complete, the population of the new generation is evaluated. At this step, each individual is run as a solution to the problem at hand, and its fitness is computed and recorded. If any individual achieves the desired fitness, or the maximum number of generations has been reached, PEAS will terminate. Otherwise, the epoch cycle will begin anew.

4.4.4 Speciation

For each species in the previous generation PEAS chooses an individual at random to serve as that species's representative. The species are then emptied of members. Each individual in the new generation is compared to species representatives using PEAS's genetic compatibility function (see Section 4.2), and, once a sufficiently similar representative is found, the individual will be added to that species for the next generation. If no sufficiently similar representative exists, the individual becomes the representative and member of a new species.

After all individuals have been assigned to species, extinct species with no members are removed and the remaining species have their fitness adjusted inversely to species size (this discourages bloated species and gives smaller species the chance to develop). Finally, each species is assigned its allotted capacity in the next generation proportional to the adjusted fitness of its members. This ensures that the best performing species persist and have a proportionally greater impact on the next generation, while also nurturing innovation by not stifling the development of currently less fit but promising (genetically diverse) species.

Chapter 5

Results

5.1 Experiment Design

We ran several experiments, first to find the optimal hyperparameter values for our model, and eventually to answer the question: do the aforementioned techniques improve our model beyond a base genetic algorithm?

An experiment consists of running multiple trials across different configurations while varying some parameter. We run these trials in parallel for efficient computation. The following figure displays the results from a typical trial run of the sum3 example on our full model with speciation disabled. This trial terminated successfully after 33 generations (at a fitness of 1). We plot the max fitness across all individual genomes in the population after each generation as well as their average fitness against the generation number. As expected, the best individual(s) trend upwards in fitness by roughly discrete jumps of the size of our stratified test cases, while average fitness stays very low (since most individuals in the population solve few test cases).

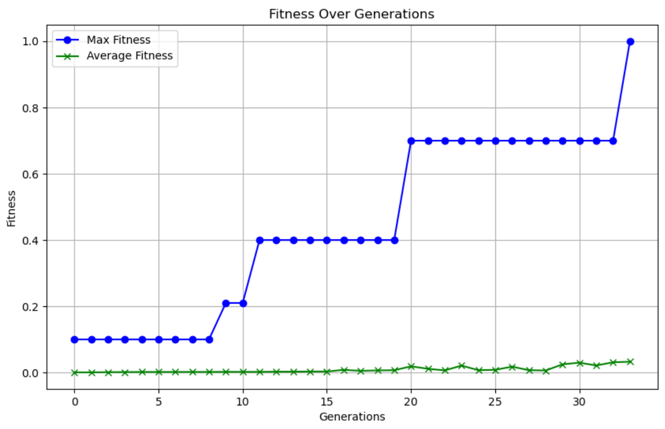


Figure 5.1: Maximum and average fitness of individuals in the population against generation number for a sum3 trial

For sum3 and sum4, we fix a cutoff of 300 generations, treating a trial as a success only if it finds a solution prior to hitting that maximum. As will be seen in our main experiments, the success rate doesn't vary much after a certain length of max generations, indicating that it is best to adopt a "fail early" approach, cutting the algorithm off if it doesn't have anything and starting anew rather than waiting indefinitely. Running the model twice with a cutoff of 300 generations will yield a solution much faster on average than running it once with a cutoff of 600 generations.

5.2 Optimizing Compatibility Threshold

The compatibility threshold is the constant that determines whether two genomes are similar enough to be considered in the same species and eligible for reproduction (measured against the output of the δ function defined in Section 4.2). Because this number is arbitrary, and we found little discussion of it in the literature, we proceeded experimentally with finding the optimal (range of) value(s). Preliminary testing showed that if the threshold was too large, all individuals would be assigned to the same species, defeating the purpose of the speciation. Thresholds that were extremely small had the opposite issue, with most individuals being placed into their own species, without enough interaction with other individuals to meaningfully exchange genetic material. Based on these results, we narrowed down the range between 1.5 and 2.5 for further testing and found that a threshold of 2.0 yielded the best success rate and average generations to finish.

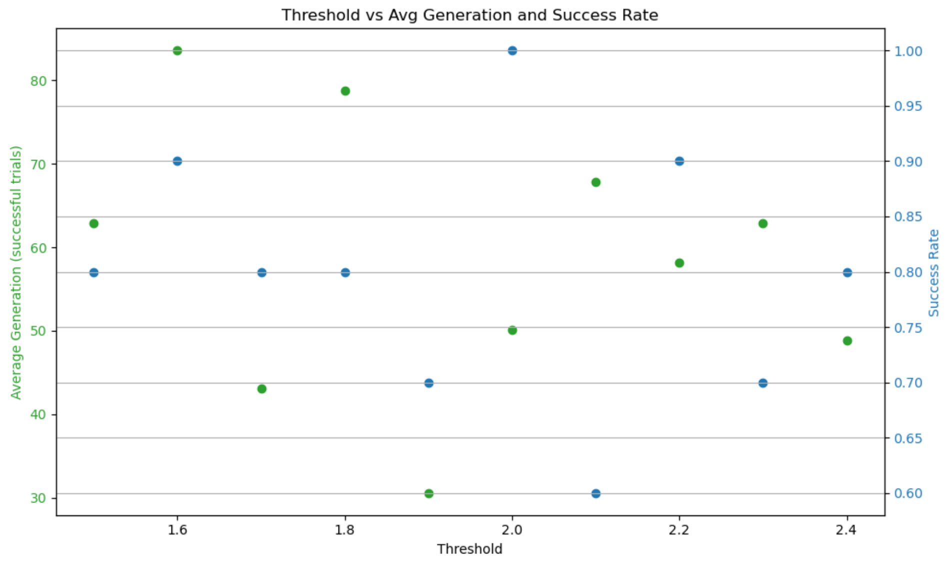


Figure 5.2: The success rate and average generations to completion from 100 PEAS trials on sum4 are plotted on a dual y-axis against the speciation threshold value.

5.3 Optimizing Degenerate Test Case Distribution

We tested whether smoothing out the fitness function by using degenerate test cases was effective in reducing the time needed to find a solution. We compared different proportions of the different partial test cases in a large experiment on sum3. For sum3, we manipulated both the rate of partial test cases where only one value is randomized and two are fixed at zero and the rate of partial test cases where two values are randomized and one is zero. We found the following using the full PEAS model:

Test Case Regime	Number of Trials	Success Rate	Average generations
0.0-0.2	5000	0.720	138.1
0.2-0.1	5000	0.708	137.5
0.1-0.2	5000	0.723	133.9
0.1-0.0	5000	0.702	141.2
0.0-0.0 (control)	5000	0.748	131.2

The value of these parameters is arbitrary, but their value relative to each other may matter. The partial test case regime that had the best results aside from the control was the 0.1-0.2 (meaning 10% of test cases have only one degree of freedom, and 20% have only two degrees of freedom), so this is what we'll incorporate into future experiments. This raises the question of how appreciable the differences that the partial test cases make are, one which we will scrutinize further.

We performed a similar comparison on different partial-test-case distributions on sum4 (defined analogously to sum3) using the full PEAS model, and collected our results in the following table:

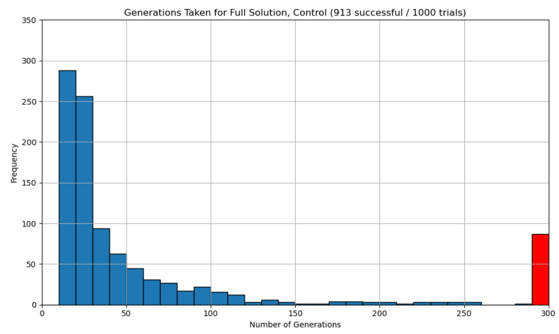
Test Case Regime	Number of Trials	Success Rate	Average generations
0.04-0.04-0.00	1000	0.384	368.8
0.00-0.00-0.00 (control)	1000	0.384	365.8
0.08-0.04-0.02	1000	0.371	371.6
0.10-0.00-0.00	1000	0.396	362.1
0.02-0.04-0.08	1000	0.366	368.1
0.00-0.04-0.04	1000	0.358	375.7
0.00-0.10-0.00	1000	0.356	374.0
0.04-0.00-0.04	1000	0.386	364.5
0.00-0.00-0.10	1000	0.361	374.3

Because of the remarkable similarity of the results, we chose the proportion 0.02-0.04-0.08 for future tests for its easy scalability.

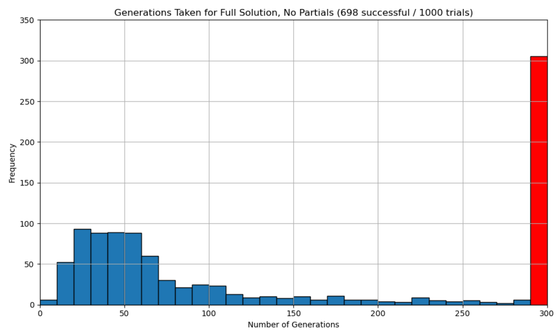
5.4 Determining Feature Significance with Ablation

In order to determine whether each added GP feature actually improves our model as hypothesized, we performed an ablation experiment, testing combinations of removing crossover, speciation, elitism, and partial test cases against our full PEAS model, on both the sum3 and sum4 problems.

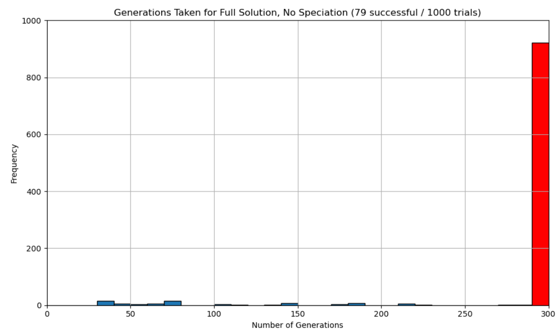
Beginning with sum3, we compared the full PEAS model to itself sans each of the four aforementioned functionalities individually, as well as the combinations of no crossover/no partial test cases and no speciation/no partial test cases. We performed 1000 trials of sum3 on each version of the model, computed how many were successful (found a perfect solution in less than 300 generations) and the average number of generations it took to find such a solution (300 if no solution was found) and compiled the results in the following table and histograms (speciation and partial test case related models displayed).



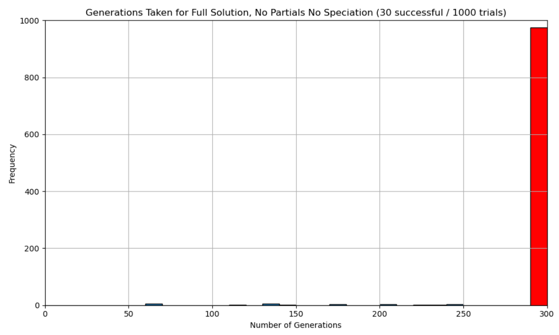
(a) 1000 sum3 trials on PEAS with all operations



(b) 1000 sum3 trials on PEAS without partial test cases



(c) 1000 sum3 trials on PEAS without speciation

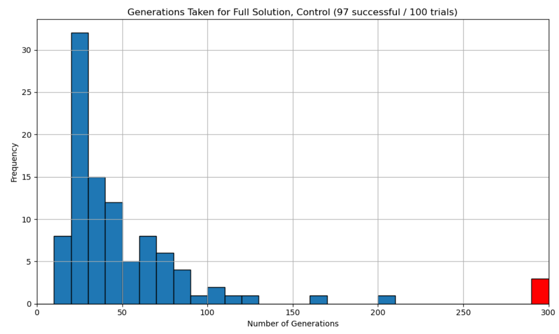


(d) 1000 sum3 trials on PEAS without partial test cases and without speciation

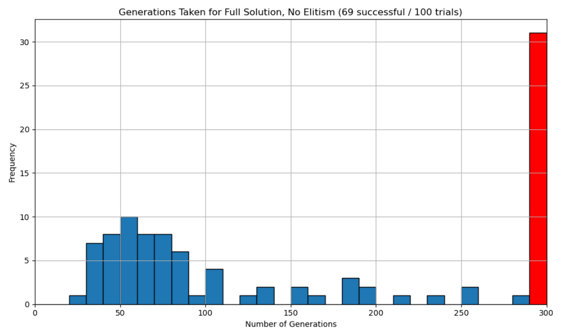
Figure 5.3: Ablation test comparing the model with and without partial test cases and speciation. Blue bars represent successful trials; red bars represent trials that reached our generation cutoff before achieving success.

Model Version	Number of Trials	Success Rate	AvgGens
no elitism	1000	0.76	123.8
no crossover	1000	0.00	N/A
no speciation	1000	0.08	284.6
no partials	1000	0.70	140.8
no crossover/partials	1000	0.00	N/A
no speciation/partials	1000	0.03	296.3
full model (control)	1000	0.91	63.5

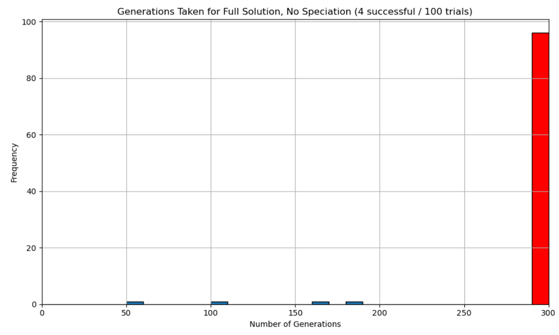
Continuing with sum4, we ran the same experiment with the same models, though with 100 trials each, and compiled the results in the following table and histograms (top 4 models by success rate selected for the histograms).



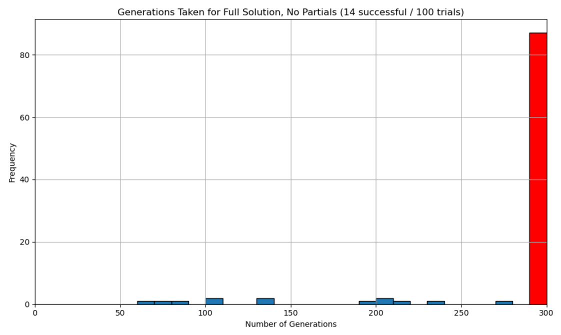
(a) 100 sum4 trials on PEAS with all operations



(b) 100 sum4 trials on PEAS without elitism



(c) 100 sum4 trials on PEAS without speciation



(d) 100 sum4 trials on PEAS without partial test cases

Figure 5.4: The best 4 performing ablated PEAS models on sum4. Blue bars represent successful trials; red bars represent trials that reached our generation cutoff before achieving success.

Model Version	Number of Trials	Success Rate	AvgGens
no elitism	100	0.69	156.6
no crossover	100	0.00	N/A
no speciation	100	0.04	293.0
no partials	100	0.14	281.3
no crossover/partials	100	0.00	N/A
no speciation/partials	100	0.00	N/A
full model (control)	100	0.97	53.6

Chapter 6

Conclusion

6.1 Analysis of Ablation Experiments

We've shown both that the application of NEAT-inspired historical markings for genetic operations and speciation as well as using partial test cases as evolutionary scaffolding are effective in a linear GP model using a novel WebAssembly-based genome. Furthermore, using those techniques together provides benefits beyond an additive effect. We conclude that they have a synergistic effect, with speciation being positively associated with the benefits of the partial test cases.

The ablation experiment allows us to decompose which functionalities are actually useful to the model. First, we find crossover to be an absolutely necessary step in sum4, even without the use of partial test cases. The mutation-only model had zero successes over 100 trials, while the unspeciated crossover was slightly more successful.

Furthermore, we found that using speciation yields a significant benefit over unspeciated crossover, even when partial test cases are disabled. Notably, in both the sum3 and sum4 results, when speciation is disabled, using partial test cases provides an appreciable, but small benefit. This indicates that speciation is a necessary component, but may have a more complex relationship with the partial test cases than we first realized. An intuitive idea is that as the problem gets more complicated (sum3 $\rightarrow$ sum4), the partial test cases make a more significant difference, as the scaffolding effect is proportionally more useful, although the differences between sum3 and sum4 as problems may be small enough to make the magnitude of this discrepancy small as well. Either way, observing Figures 5.3-4, it is apparent that the use of partial test cases provides a significant increase in successful trials and lowers the overall distribution of the number of generations.

As with NEAT, which improved upon previous neuroevolutionary techniques by introducing historical markings and enabling crossover and speciation, we show in our model that speciation is necessary for this implementation of linear genetic programming. It is by using that concept of historical markings that we can define our compatibility function and enable speciation. We were able to successfully use this idea in our model, marking code instructions rather than edges of a neural network.

The partial test cases provide evolutionary scaffolding by establishing partial solutions as intermediary goals in the fitness function. In our model, the strategy was to fix an input such that the expected behavior was some simpler function. This idea can be applied to other genetic programming methods which use test cases in the fitness function. For instance, if the desired behavior is complicated and contains edge cases, then one could define a “partial credit” where the program could be tested on a simpler test case for less fitness if it fails the more complicated test case. Furthermore, as discussed in Sections 2.1-2, the feasibility of this idea is predicated on the problem’s ability to be split into modular subproblems, which cannot be assumed ante facto. This requires domain knowledge of the problem itself and potentially manual decomposition into such partial solutions.

Speciation, in crossover between similar parents, shortens the distance between them in the solution space. Therefore, the smoother the fitness function (i.e. the less variance in a local area/less choppy), the better its effect. Partial test cases smooth out the fitness function by providing more granular feedback. Thus, when using partial test cases, speciation becomes more effective. However, it’s not obvious that having a more detailed fitness function alone is able make it more locally stable.

We view the core intuition behind this synergistic effect as speciation acting to preserve innovations, while partial test cases reflect the value of those innovations in the fitness function. Without speciation, those innovations would not be preserved across generations; and without partial test cases, the innovations being preserved would not be valued by the genetic algorithm. Speciation allows useful building blocks to be kept, and partial test cases uncover the value of these small building blocks. The larger the jumps in the fitness function, the larger those building blocks need to be in order to be recognized as worthy to keep. The genetic algorithm without partial test cases may have overlooked many of these small building blocks—code expressions—since they didn’t provide value in the fitness function, thus leaving that species to die out.

The idea behind speciation is that, when crossing over, two similar parents will have less likelihood of losing fitness, thus preserving existing innovations. Success from speciation implies that inter-species reproduction yields higher average child fitness than the global average. For the inclusion of partial test cases in the fitness function to increase the effectiveness of speciation, we would speculate that it must be common that a child evaluated against the modified fitness function with partial tests does much better than when evaluated by the original fitness function without. This may seem unsurprising, since it is the case in general that the modified fitness scores are higher than the original fitness scores, however it is yet to be seen if this rate is actually heightened for children with similar parents. Further research is necessary to confirm this.

Speciation inherently increases exploitation at the cost of exploration. By limiting the search for a successful pairing to similar parents, we attempt to build upon their shared, existing innovations, instead of searching blindly for combinations where each parent has components that would combine well. Thus, potential gains from combining varying components are not explored by speciation, since those variations are constrained by the compatibility threshold. We speculate that the nature of linear genetic programming is such that with simple crossover, there's a low probability that potentially useful segments of code in the parents will recombine in a beneficial way. That is, even if two parents have the potential to combine their innovations fittingly, with crossover points being random, the reality is that most of the time the child may end up losing those innovations. One area of interest may be in more intelligent recombination methods, or genetic representations which yield efficient recombination.

6.2 Future Work

While we only demonstrate PEAS on simple arithmetic problems, Wasm is a Turing-complete language and is used in real-world applications. Due to the linear genome representing the actual instructions executed, adding new operations would just be a matter of adding them to the mutation. It would be possible to call externally provided functions from within the module, providing the individuals a library of higher-level actions they can perform. However, more complicated structures would require more effort.

6.2.1 Control Flow

Currently our allowed operations limit us to straight-line programs without control flow. Implementing this is an important step in being able to synthesize complete programs, and would require further (re)design of the genetic representation and operators. This is primarily because it is difficult to model both data flow and control flow together in our model [18]. This would require some sort of code property graph.

Conditionals in Wasm consist of an “if” instruction which contains two sections of instructions, the “then” block and the optional “else” block. They act by popping the condition off of the stack and continuing to the “then” block if the value is nonzero.

In WebAssembly, blocks of instructions are sectioned with a “block” instruction and a corresponding “end” instruction, and can be exited using a “br” break instruction. Loops are similarly organized as blocks of code which have a break. This introduces complications when viewing the genome as a linear list of instructions. One must ensure that each block has a corresponding end, and that it has a proper break. Moreover, loops or conditionals which change the stack height can cause it to be unpredictable for later instructions. This means they would have to store the result in a variable or memory location instead of leaving it on the stack for the next instruction. After a certain length of genome, we start reaching individuals which are doing things too complicated for the problem. Another solution might be to add a well-designed delete mutation.

6.3 Lessons Learned

We learned not to be too focused on our preconceived ideas, and to be flexible to trying different approaches. An area we could have benefitted from exploring more earlier on was higher level representations of the code. We spent a fair amount of time trying to make a graph representation of the genomes work, but encountered the stubborn problem of how to translate such a graph into low-level WebAssembly code and how this might extend to control flow. While it seems beneficial to have a simplified instruction set to work from, it also limits the possible genetic operators since these instructions are so fixed in format. They take a fixed number of inputs, so operators must be careful in how they are manipulated and added.

While previous work using NEAT on an Abstract Syntax Tree (AST) graph has shown some success [27], we wanted to try a linear approach. We found some success for these basic problems, but it's obvious that for more complex tasks and programming primitives, we would need a more structured genome. The main issue when designing additional operators for our genome is crossover; if we continue using the same method of crossover, it becomes easy to generate invalid code—for instance, taking the beginning of a block or loop, but not the end.

Appendix A

Source Code Repository

All source code for PEAS was written by Alexander Ng and Stefan Stealey-Euchner, except for noted imported libraries, and can be accessed at
`https://github.com/alexandermng/peas/`.

References

[1] Y.-c. Wu and J.-w. Feng, “Development and application of artificial neural network,” *Wireless Pers Commun*, 2017.

[2] A. Erasmus, T. D. Brunet, and E. Fisher, “What is interpretability?” *Philos. Technol*, 2020.

[3] C. M. Burnett. “Artificial neural network.” Accessed: 05-07-25. (2006), [Online]. Available: `https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg`.

[4] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary Computation*, 2002.

[5] W. B. Langdon and R. Poli, *Foundations of Genetic Programming*. Springer Berlin, 2002.

[6] J. R. Koza, *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*. Stanford University, 1990.

[7] R. Poli, N. F. McPhee, and L. Vanneschi, “Elitism reduces bloat in genetic programming,” *GECCO*, 2008.

[8] Y. Fang and J. Li, “A review of tournament selection in genetic programming,” *Advances in Computation and Intelligence*, 2010.

[9] J. Page, R. Poli, and W. Langdon, “Mutation in genetic programming: A preliminary study,” *EuroGP’99*, 1999.

[10] S. Luke and L. Spector, “A comparison of crossover and mutation in genetic programming,” *Proceedings of the Second Annual Conference on Genetic Programming (GP-97)*, 1997.

[11] W. Veit, “Scaffolding natural selection,” *Biol Theory*, 2021.

[12] H. J. Goldsby, R. L. Young, J. Schossau, H. A. Hofmann, and A. Hintze, “Serendipitous scaffolding to improve a genetic algorithm’s speed and quality,” *Association for Computing Machinery*, 2018.

[13] J. D. Schaffer, D. Whitley, and L. J. Eshelman, “Combinations of genetic algorithms and neural networks: A survey of the state of the art,” *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1992.

[14] S. W. Mahfoud, "Niching methods for genetic algorithms," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1995.

[15] M. A. Potter and K. A. D. Jong, "Evolving neural networks with collaborative species," *[Proceedings] 1995 Summer Computer Simulation Conference*, 1997.

[16] A. Hagg, M. Mensing, and A. Asteroth, "Evolving parsimonious networks by mixing activation functions," *[Proceedings] GECCO'17*, 2017.

[17] A. Gaier and D. Ha, "Weight agnostic neural networks," *Conference on Neural Information Processing Systems 32*, 2019.

[18] M. F. Brameier and W. Banzhaf, *Linear Genetic Programming*. Springer New York, 2007.

[19] L. Spector, "Autoconstructive evolution: Push, pushgp, and pushpop," *[Proceedings] GECCO-2001*, 2001.

[20] A. Rossberg. "Webassembly specification." Accessed: 05-06-2025. (2025), [Online]. Available: `https://webassembly.github.io/spec/core/`.

[21] Emscripten Contributors. "Emscripten documentation." Accessed: 05-06-2025. (2025), [Online]. Available: `https://emscripten.org/docs/index.html`.

[22] A. Gutev. "Wasm-encoder." Accessed: 05-06-2025. (2025), [Online]. Available: `https://docs.rs/wasm-encoder/latest/wasm_encoder/`.

[23] The Bytecode Alliance. "Wasm-tools." Accessed: 05-06-25. (2025), [Online]. Available: `https://github.com/bytecodealliance/wasm-tools`.

[24] The Bytecode Alliance. "Wasmtime." Accessed: 05-06-25. (2025), [Online]. Available: `https://docs.wasmtime.dev/`.

[25] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, 2012.

[26] T. Helmuth and L. Spector, "General program synthesis benchmark suite," *[Proceedings] GECCO '15'*, 2015.

[27] L. Trujillo, L. Muñoz, E. Galván-López, and S. Silva, "Neat genetic programming: Controlling bloat naturally," *Information Sciences*, 2016.