Uniform Projection of Program Space Geometry for Genetic Improvement of Software

Benjamin J. Craine
School of Computing and Communications
Lancaster University, UK
b.craine@lancaster.ac.uk

Barry Porter
School of Computing and Communications
Lancaster University, UK
b.f.porter@lancaster.ac.uk

ABSTRACT

Current Genetic Improvement (GI) for software systems use pre-existing program representations, such as abstract syntax trees and bytecode, to apply genetic operations to. These representations, however, were designed for the purpose of translating human readable source code to machine code. When used to underpin GI, these representations have drawbacks, such as the risk of breaking a program when deploying mutations, and the difficulty of expressing search trajectories that are not entirely rooted around the initial individual. We present a novel matrix-based program representation which is specifically designed for the purpose of GI. Our representation (i) makes it *impossible for mutations or crossover to yield an invalid program*, without the need for any syntactic or semantic checks, while still making every valid program reachable by search, and (ii) supports the simple expression of *rich, layered probability distributions* atop the program matrix to guide a GI search process in a wide variety of different ways. We build an end-to-end GI system using this new representation and demonstrate how we can layer a range a probability distributions on top of the representation to gain different effects. We also explore the future research possibilities that this approach to program representation presents.

1 INTRODUCTION

Genetic Improvement (GI) for software has been used extensively in automated bug fixing [15], and more recently in automated performance improvement of software [18]. The state of the art research in GI uses program representations that borrow from existing compilation / interpretation pipelines, such as Abstract Syntax Trees (AST) [2, 12], Bytecode [13], or ASCII source code.

These representations have significant shortcomings when used to underpin GI processes. One is that extensive checks are needed to ensure that mutated programs remain within a syntactically- and semantically- valid envelope. Typically, when considering the deployment of a mutation, a line of code is selected at random, and then the set of all possible mutations is filtered to those that are most-likely-valid at this location; this filtering requires tracking of in-scope variables, their types, and valid operators on those types, etc., and is still subject to mistakes [4, 9, 11, 14]. Another shortcoming is that the trajectories of program search are tightly bound to the locale of the starting individual, since each mutation is an incremental delta from this starting individual; this severely limits the set of search distributions that can be expressed within the full theoretical program space.

We present a novel matrix-based program representation form which, as far as we are aware, is the first to be specifically designed for GI. This representation first allows for the use of any possible mutation and crossover strategy combination, whilst inherently guaranteeing that any resulting program is compile-able, without the need for explicit syntax or semantic checks. This includes the ability to apply multiple mutations to the same program in parallel, which in turns lends itself to deployment on SIMD architectures such as modern GPUs; part of our implemented GI loop is offloaded onto a GPU to demonstrate this. Second, our novel representation allows the layering of arbitrary probability distributions atop the program representation matrix, allowing the expression of rich search trajectories – including those that are bound to the locale of the starting individual with an arbitrary weighting, those that are not bound in any way and can rapidly access the totality of theoretical program search space, or hybrid distributions which keep some parts of individuals in a starting locale and allow others in reach different locales.

We have implemented a mechanism to explore this new space of mutation strategies, which also offers diverse future research directions. Our specific contributions in this paper are:

- We present the details of our novel numeric matrix-based program representation for general-purpose languages (Sections 3 – 4.2)
- We provide a method for realising source code from numerical matrix instances using our novel representation, and show that this is entirely general without the need for validity checks (Sections 4.3 – 4.4)
- We design an end-to-end GI system based on our novel representation, and show how it is suited to highly parallel architectures such as GPUs (Sections 4.2, 4.3 and 5);
- We provide a mechanism for exploring the space of mutation strategies using layered probability distributions, and show how a different set of distributions affects GI performance and outcomes: Sections 5.2 and 6.

We conclude the paper by discussing novel future research directions that are made possible by our approach.

2 BACKGROUND

In this section we first examine why current program representations used in GI may derive programs that are invalid, then show how we may design a novel representation that mitigates this.

We first define two different spaces: *solution space* and *physical space*. We define solution space as the area of syntactically and semantically valid programs – in other words, programs which can be compiled with no errors. We define physical space as the area of all possible program formulations, many of which will be syntactically and/or semantically invalid.

As an example, consider a program represented in an abstract syntax tree. Such a program will have AST nodes representing operators and operands, where some operands may be local variables of a function. If we are to mutate a variable node to be a different local variable, the solution space here is *all other local variables which are in-scope and type-compatible* with the current variable. The physical space, by comparison, is *all other local variables* regardless of whether they are in-scope or type-compatible. The same definition can be applied to operator nodes of the AST, where one operator could be mutated to a set of other operators which are compatible with the existing, or a set which exist in the language but are not compatible with the operands, etc. We can similarly extend this solution space :: physical space definition to other program representations, such as bytecode and machine code.

We illustrate this spatial definition in Fig. 1, where the light-gray area represents solution space and dark-gray represents physical space. The boundaries of solution space are irregular, depending on the direction of travel, which results in the need for extensive correctness checking for mutations that are expressed relative to AST or other classical representational forms.

Our novel representation uses a numeral encoding in a rectangular matrix, such that the problem maps very closely to our illustration in Fig. 1 – in other words, some collections of numerical values for a given matrix instance lie in solution space, and some lie in physical space. Our aim is for a representation which allows mutations over a matrix to do *anything* to the input matrix and still yield a valid program – i.e., a value inside the solution space which is syntactically and semantically correct. Due to physical space being larger than solution space, a corrective approach is needed to move physical space matrix values into solution space.

Our solution to this is to allow multiple numerical points in physical space to represent a single point in solution space, such that solution space is effectively a regular rectangle which encompasses all of physical space. With this, a mutation can do anything to its input matrix values and will always return a value that, under our representation, can be interpreted as a valid program.

In this section we have illustrated the problem and the basic intuition behind our approach. In the following sections we elaborate on how we move from a numerical matrix to program source code, and how this transition ensures that we always stay within solution space. We will also show how we can layer probability distributions atop the numerical program matrix to define a wide variety of search constraints or search directions.

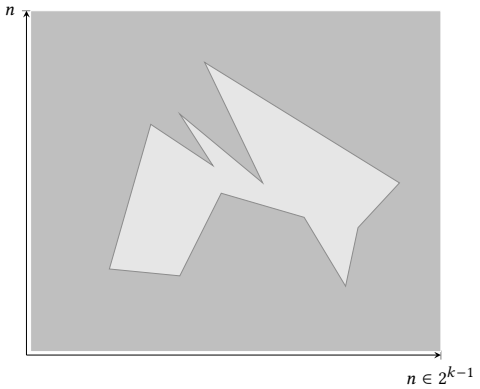

Figure 1: Irregular Search Space Geometries

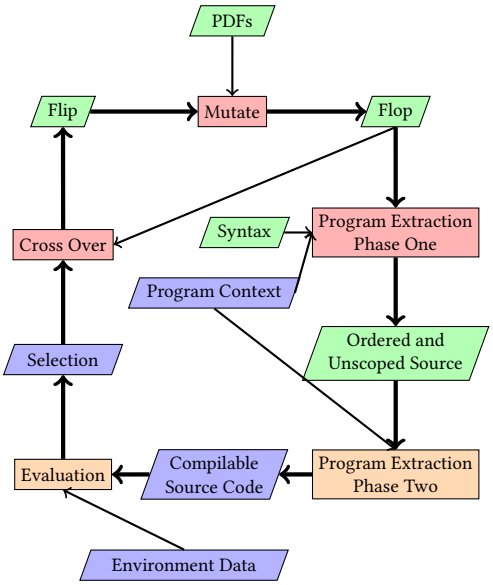

Figure 2: GI System Overview

3 SYSTEM OVERVIEW

In the following sections (4-5) of this paper we describe how we've taken the above principles, and applied them to build an end-to-end GI system that can use any mutation and crossover strategy and still yield output that is guaranteed to be in solution space. An overview of the complete system is shown in Figure 2, and each of the next sections will explain one or more of its components. The system starts from an initial population of compilable source code, shown at the bottom of the loop. This is then evaluated for fitness using environment test data, after which it is converted into a numerical matrix form. This matrix form is used for selection, crossover, and mutation, after which we convert matrices of individuals back to source code for another round of fitness testing.

Elements in the diagram highlighted in green are objects stored in GPU memory, those in blue on the CPU. Elements in red are compute processes that execute on the GPU, with those in beige executing on the CPU. The elements labeled 'flip' and 'flop' are GPU memory areas reserved to store the GI population. Program Extraction phases one and two translate a program from our numerical matrix representation into source code, and the PDFs at the top of the loop are probability density functions which inform the system on how to mutate individuals and navigate search space.

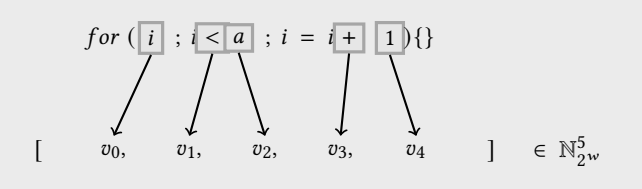

Figure 3: Feature Vector Construction

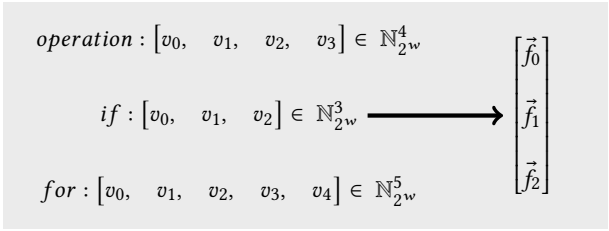

Figure 4: Feature Column Construction

4 OUR REPRESENTATION

4.1 Feature Vectors

Our target programming language is a general-purpose systems language called Dana [17], which has been used in a range of GI research [1, 20] and allows individual parts of programs to be improved and dynamically re-injected into a running system with safety assurance. It has a broadly similar syntax grammar to Java.

Our current representation version focuses on a small subset of the language, but one which enables a broad range of GI optimisation applications. The main constraint is that we currently ignore most typing concerns, assuming that every variable is an unsigned integer or array of unsigned integers. Besides this we support variable assignments, arithmetic operators, and logical comparison operators of equality and greater-then / less-than. We also support the control flow constructs of for-loops, while-loops, and if-statements. Most general purpose languages share these features, making our approach translatable to other languages.

We model each possible kind of source code line, from assignments, to operators, or control-flow constructs, as a *program feature*. Each program feature, such as a for-loop, has its own feature vector which has an associated schema. Figure 3 illustrates how we do this in the case of for-loops. In this example vector, v_0 represents the iterator variable of the for-loop, v_1 represents the termination condition operator, and v_2 represents the variable containing the loop limit value. For any given feature vector and a given schema, we can therefore reconstruct the syntax it represents in source code by reading the current values from the feature vector.

4.2 Function Matrix

Using the above approach to represent individual features of a program, we need a way to package them together in memory to form our search space of all possible combinations of n features that are achievable when producing source code from our encoding.

We do this via a *Function Matrix* which has n columns, where each column has a row for each feature schema. We use the indices of these rows to know which schema to use when translating a program feature from a feature vector into source code. The first row of this matrix may therefore be the row representing assignments, while the second row may represent operators, and the third for-loops, etc. This is illustrated in Figure 4. The column count is then set to be as large as needed to provide sufficient space to allow n feature combinations. Within this representation, feature vectors of a given type (e.g. those for if-statements) are packed into a multiple of n-bits, where n is the default system bit-width. Current GPUs tend to default to 32-bit number representations, so we use 32-bits as our allocation unit; a feature vector needing more than 32-bits bits will span across multiple 32-bit cells.

When reading out an entire function matrix, we read column 0 first, iterating through each of its rows from the top to the bottom. For each row that has a non-zero value, we interpret the schema for that row to translate the numerical value into valid source code. If the 0'th row of column 0 has a non-zero value, therefore, we generate a line of code which is an assignment, using variables according to those indicated by the numerical values of the cell; if the next row of column 0 has a non-zero value, we generate a subsequent line of code which represents an operation, and so on, until we reach the last row of the column. We then move to column 1 to repeat the process, until we reach the end of the columns.

The schema for each row ensures that we derive a syntactically and semantically-valid overall outcome from this process, in combination with additional rules around variable declarations as follows. Variable declarations can be particularly challenging to deal with when applying genetic operators to software representations. It is easy to remove a variable declaration that is referenced later in a program, or reference a variable in a scope where it is not visible. We avoid this complexity simply by deciding at the outset how many variables will be available and declaring all of them preceding any other source code, making them visible to all scopes. Each feature schema is then bounded by the maximum variable name available when decoding the numeric value of a matrix cell into a source code representation.

4.3 Extracting Source Code

Our function matrix, along with information on declared variables, gives us everything we need to construct compiler-safe source code. Having stored all this information in a matrix, we are able to put this data on a GPU; this is because our program representation now appears as a rectangle of numeric values, which is essentially analogous to an image of pixels. We can then highly parallelize many of the operations performed on the matrix itself, as well as parts of the process that decode a matrix instance into source code.

Referring back to our the system overview, an individual in a GI process is represented by one function matrix, with a population being a collection of such matrices. In our case, all individuals have the same dimensions so we can store them contiguously in memory. This population then has two areas of memory allocated for it, labeled as Flip and Flop on the system overview on Fig. 2.

Our source code extraction process converts our numerical matrix into ASCII source code. To do this we first pre-allocate some

memory on the GPU to store the ASCII output, with the knowledge that this memory region is large enough to store any permutation of the entire function. This requires some knowledge of the syntax of the target language and its relation to the length of variable names. In our case the language feature that requires the most ASCII-representation syntax is the for-loop. Assuming that each variable part of the for-statement is filled in with the longest possible value (characters in the name of a variable or digits in a literal value), we can calculate the length of the longest possible for-statement as a whole. We then multiply this by the number of features represented in our function matrix to get a length able to fit any program when converted to ASCII – in this case, the length if every single line of resultant source-code was a for-loop.

We can also calculate all the offsets in the source code array that mark the limits of where each GPU thread should write to, when converting an individual feature vector from one cell of our matrix into ASCII. This pre-allocated and pre-divided array for the source code ASCII is labeled as *Ordered and Un-scoped Source* in the system overview. To extract the features, we assign one GPU thread per cell of the matrix. We pass to each thread its feature vector, its offsets into the above ASCII source code memory region, and its syntax schema. Information on available variables are available as read-only to each thread.

Each thread then executes the procedure in Alg. 1. The function call *mapping* in Alg. 1 is the mechanism by which we take the space of all possible function matrices (of which a lot, if not most, will be unable to produce valid source code), and remap them to values that will produce valid source code. The space of all possible function matrices is effectively the dark gray space represented in Figure. 1. The remapping function takes all space coloured in dark gray to some point inside the light gray shape. Literally, however, this mapping function takes the integer stored as an element in the feature vector and uses some auxiliary information (e.g., the list of available variables) to return another integer with a maximum value of the number of possible values the syntax can take at that point, minus one. For example, it is known that the first integer in a operation feature vector represents the variable that the result of the operation will be assigned to. Therefore we consult the writeable variable array for its length then return a number lower than that value, such that we can then index into the variables array and copy the variable name into our source code array.

Algorithm 1 Program Extraction Phase One

```
unscopedProgram
featureVector
syntaxTemplate
i ← 0
while i < featureVector.length do
    unscopedProgram.add(sytnaxTemplate[i])
    nextString ← mapping(featureVector[i], variableNames)
    unscopedProgram.add(nextString)
end while
l ← syntaxTemplate.length
if syntaxTemplate.length > featureVector.length then
    unscopedProgram.add(syntaxTemplate[i : l − 1])
end if
```

The obvious mathematical implementation method for collapsing this space is performing a modular operation on the integer found in the feature vector, by the number of possible valid syntax options we have. Such a modular operator would, however, result in the property that notionally equivalent source code results (where the same numerical value has the same ASCII output) would not be adjacent in the output space. We therefore use an alternative method to collapse the space, defined as:

$z = \frac{s}{N}$

$mapping(x, N) = n \iff zn < x \leq z(n+1)$

Where s the biggest number for our system width, and N is the number of actual categories in this dimension of the search space. Here x is the numerical value we find in the feature vector cell. We map it to n if and only if the inequality is satisfied. This approach has the property of having all binary strings that will be mapped to the same value are adjacent in the larger output space.

4.4 Scope Handling

Finally, some of our feature types (ifs/whiles/for-loops) introduce scoped segments of code. The start of a scope is part of the syntax associated with such feature types when converting the feature vector to ASCII source code strings. However, as currently described, the function matrix has no way of encoding where such an opened scope should be *closed*. For this, our encoding treats repeated identical feature vectors of the same type, in separate columns of our function matrix, as the start and end of a scope. Where feature vectors inside the function matrix can be processed into strings in parallel, for scoped features, knowing whether this is the only instance of this feature vector inside the matrix requires knowledge of other columns of the matrix. We therefore ignore scoping in the parallel ASCII conversion stage and allow a single thread back on the CPU to convert any necessary strings from full statements to tokens that denote the end of a scope.

5 IMPLEMENTING GI

In this section we describe how a GI process is layered atop our matrix-based representation and its accompanying source code translation approach. Much of this element of our framework looks fairly typical at a high level, however there are key differences due to our numerical matrix-based program representation.

5.1 Mutations and Crossover

Mutations in our approach are realised by simply changing the numerical value of one selected n-bit (e.g. 32-bit) cell in a function matrix, using a source of randomness to derive the mutated numerical value. Because every combinatorial matrix value results in a valid program, the resulting numerical value of a mutation is assured to yield a program in solution space. However, the effect of mutations has subtle differences to those on an AST representation.

The main difference is the lower level of semantic meaning to mutations, in particular that applying the *same* mutation to two different matrices may yield *different* outcomes. This is never the case in AST representations, where inserting a new operator at the same position in two AST individuals will definitely result in both individuals now having that same operator present at this location (assuming it was syntactically and semantically legal for

both individuals). To contrast this with our case, consider two different function matrix instances which currently the same ASCII source code translation. It is possible for this to be true even if the numerical values of the two matrices have minor differences, since multiple numerical values necessarily map to the same point in space in order to convert our solution space to a regular rectangle (as discussed in Sec. 2). In conditions, we can add the value 10 to the same coordinate cell in both matrices, and yield different outcomes in the resultant ASCII. This is because the value of one cell may have been closer to a threshold of a different ASCII effect than the value of the other cell. The inverse effect can also arise, where two *different* mutations on two different individuals can result in those individuals having the same ASCII output where they previously did not. Our results show that GI proceeds in a similar way to AST-based approaches, seemingly impervious to this subtlety, but analysis approaches such as phylogenetics may require additional information to understand mutational causes and effects.

Crossover has a similarly numeric basis; a crossover between two individuals simply copies X cells from one individual's matrix, and Y cells from another, to form a new individual. The number of cells taken from each individual is configurable, as are the coordinates of those cells within their respective donor individuals' matrices.

5.2 Probabilistic Search Space Navigation

As mentioned in Sec. 1, existing GI program representations tend to inherently center their search on their starting individuals. Each mutation represents an incremental step away from this starting individual, but reaching distant parts of a search space requires many sequential mutation steps and must deal with the phenomena of neutral drift [4, 11, 14]; in practice this tends to keep mutated individuals in the close vicinity of the opening progenitor.

In our approach, there is no need to center our search on anything. Indeed, in each generation we could randomly mutate every cell of a function matrix to jump to wildly different points in program search space, knowing that every point yields a valid program. In practice this may be unlikely to yield individuals useful to the problem at hand, but it shows an entirely new degree of freedom.

In our current implementation we use *probability distributions* to allow us to strike a balance between constraining the search to useful areas and enabling wider exploration. Each cell in our function matrix can have a probability distribution associated with it, which is consulted when that cell has been selected for mutation. The purpose of this distribution is to take the entire range of values possible our cell could be mutated to (i.e., any value our system's width can represent) and encourage it to take a value that will be useful for locating an optimized program in our wider GI system.

More specifically, these probability distributions (represented as vectors of real numbers) give the probability that a random variable, between zero and the maximum unsigned integer our systems width allows, is in some range of contiguous values.

Retaining a parent-child relationship between the cell's original value and the mutated value, this random value we have pulled from the distribution is combined in some way with the original value to derive the final muted value. This can be as simple as: $v_1 = x + v_0$
Where v is the cell value and x is a random variable.

This begs the question of whether we can find a set of probability distributions which will produce mutations that facilitate GI in ways that match or exceed current approaches.

We know that in our encoding, representation different function matrices map to the same source code, and all these equivalent permutations cluster together (section 4.3). Therefore a desirable property of our mutations is for them to be "aggressive", i.e., larger jumps in physical numerical space is better up to a point.

We can represent this desire in our probability distribution using an S-curve that gives larger numbers a larger probability of being chosen up to some plateau. The effect this gives is that numerical mutations are more likely to manifest in the source code itself when we convert. Doing these operations on physical space also means that should we meet the case where $x + v_0$ is greater than the maximum value of our system width (e.g. 32-bits), we will roll back over to 0 and still have a successful mutation that has our desired property of being "aggressive". This S-curve mechanic is likely to minimise the phenomena of 'non-deterministic' mutations discussed in the above subsection.

In future work we intend to examine the effect of other curve geometries to describe search distributions, and their impact on mutations, or to make the shapes and degrees of these curves dynamic depending on search results so far. This may include attention-based approaches [25] with distributions that share their state to influence the value of other distributions to encode context about other parts of the program.

6 GI PERFORMANCE TESTING

In this section we experiment with our GI framework to answer two questions. Our first is whether our program representation, and consequent mutation and crossover implementation, does indeed result in a search process that reflects classical GI for source code. Second, we examine a range of different probability distributions to control that search process to show their effects.

Our application domain for GI is performance improvement, inspired by recent work in this area [19]. As in [19], we use a hash table implementation as our target for improvement, specifically targeting the hash function. We supply various input sequences of put/get calls to this hash function, and use execution time of these sequences as our fitness function. Our GI process is generally aiming to identify a hash function which (a) yields a distribution that reflects our training data, and (b) is fast to execute. Achieving both of these effects will minimize the average lookup time for any key stored in the hash table.

6.1 Methodology

We conduct experiments under three different conditions, which loosely envision three different levels of knowledge that an operator of GI might have about the problem domain of a hash function.

Our first condition assumes a user with only general knowledge of GI. This user may control parameters of the GI process, but not constrain the search space itself; in this condition all cells in the function matrix are therefore eligible for mutation, allowing the system to jump arbitrarily to any part of program search space.

Our second experiment condition assumes the operator has some domain-specific knowledge about hash functions, knowing that we

generally need a single for-loop which iterates over each character of the string input key and performs some mathematical operators to yield an integer value which is related to those characters. In this condition some cells will be not be able to be selected for mutation; this will have the effect of fixing the existing of a single for-loop, and disallowing any new scopes to be added. Any arithmetic before, inside or after the for-loop is allowed.

Our third experiment assumes the operator wishes to further constrain a search to allow only the addition of a single line of additional arithmetic within the scope of the for-loop; all other mutation effects across other cells are disallowed.

Across all three conditions we otherwise keep the same overall GI parameters, which are as follows: Population size of 30, generation count of 30, static training set of 1000 keys, fitness test as runtime.

The only difference between each scenario in testing is therefore which cells in the function matrix of an individual will be open for mutation. All mutations follow the S-Curve method discussed in Section 5.2. We repeat each scenario ten times with different random seeds, averaging the results.

6.2 Results

Generation 0 in all three results figures shows the execution time of our original hash function.

We begin with the results from our first condition, where all matrix cells are equally available to mutation, allowing the search to jump to arbitrary points in program space. Figure 5 shows the result. Here we see, on average, only a moderate improvement effect, with extremely wide standard deviation across the 10 experiments. The large deviation has two reasons. Firstly is the introduction of the ability to insert new scoped program features in this experiment condition (i.e., new loops and if-statements). This introduces more instances of undecidability to our pool of candidate programs, increasing the ratio of programs which end in seemingly infinite loops and so are forcibly killed and given a very poor fitness. Secondly is the well-documented propensity for bloat under this condition [6, 23, 26], which means the performance boosts provided by helpful mutations are often negated by large numbers of extra lines of code taking up CPU cycles. It is also interesting, however, that even in this entirely unrestricted condition, we do see some very good individuals emerging in some experiments, indicating that the GI process is still somewhat able to function in gaining improved individuals. This suggests that unrestricted searches of this form may be a good source of genetic diversity, while still remaining in the area of reasonable solutions.

Figure 6 shows the results for our second condition, where we constrain the search by disallowing new scopes / control constructs, therefore forcing a single for-loop to exist, but otherwise allowing new operators either inside or outside of that for-loop, along with general mutations to those lines of code. Here we see an extremely fast average convergence to a very good individual, reaching that point in 3 generations. Further minor fine-tuning happens after this point, and the standard deviation around the average is very low, making this search trajectory very predictable. This shows the value of being able to declare a probability distribution over our program representation, allowing us to target known good areas of the search space.

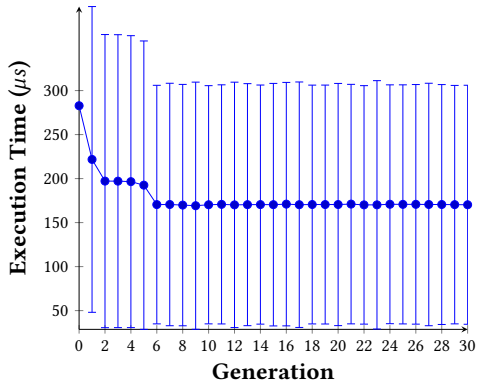

Figure 5: Condition one Zero Performance Test

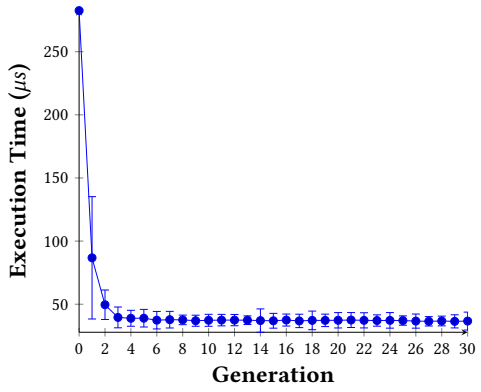

Figure 6: Condition Two Performance Test

Finally, Figure 7 shows the results for our third condition, where we are permitted to only add a single extra operation inside the existing for-loop. The results here are a more mixed picture. They show good convergence on an improved individual, but this takes more generations to achieve, with the earlier generations having a much wider standard deviation than in our second condition. Both effects are somewhat counter-intuitive, since the search space is smaller. Our assumption here is that, aligning with wider research on GI processes [14], we may have a poor ratio between affective and non-affective mutations possible at certain points in the GI process. While condition two has the larger search space, there may therefore be a greater ratio of affective mutations allowing it to traverse neutral space faster than in the smaller space of scenario three.

Overall our results show our ability to exercise clear control over the search trajectory of a GI process, using a unified paradigm of probability distributions over program matrices. The results from each condition are relatively predictable, and may serve as a novel control interface between diversity and convergence of GI.

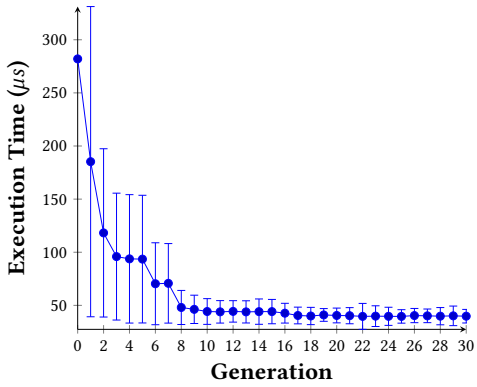

Figure 7: Condition Three Performance Test

7 DISCUSSION

Our results demonstrate that our novel program representation, and its associated GI framework, provide familiar search dynamics compared to classic GI for source code. We are also able to express a range of probability distributions to control the degrees of freedom that a GI search is given. In this section we discuss the qualitative aspects of our approach, and potential future work directions.

7.1 Parallel mutations and Predictive search

An advantage to having unconstrained mutations over numerical representations of programs is the opportunity to explore mutation strategies that may otherwise seem unfeasible. One of these strategies is having a mutation that edits separate parts of a program independently (from a processing perspective). Mutations in our approach are realised by simply changing the numerical value of one selected feature vector cell in a function matrix. Because every collective matrix value results in a valid program, we can freely change multiple cells at the same time; this would be extremely challenging to achieve safely in an AST-based representation. This parallelism extends to the ability to mutate many individuals in a population at the same time. While fitness testing tends to be the main performance bottleneck in GI systems, in future work we intend to examine sub-sampling of fitness with prediction of search direction so that we can rapidly mutate individuals towards predicted high-value areas.

7.2 Predictability of Encoding Cost

An advantage to using matrices, rather than dynamic structures like ASTs, is that we can easily model the memory space requirements for a population. We can do this given a few parameters, as follows:

- w: The width in bits of our system (we have assumed 32-bits in our current implementation).
- f: The number of program features in the source code we're attempting to improve.
- g: The growth potential of our program to encompass a higher feature number (i.e., more lines of code). We will first calculate the minimum size of our function matrix to describe all possible programs with f features, then increase this size by a factor of g.
- l: The length of the longest feature schema, in bits.
- G: The number of unique program feature schemata, indicating the number of rows in our matrix.
- $popSize$: The number of individuals in a population.

Using the above can describe the size of an individual I in our system as follows:

$I = w \times \lceil l \times f \times g \rceil$

This is used to calculate the memory requirement for a population of function matrices. We multiply this value by two, as even though the population can be stored in half the space, the population is transferred from one memory to the other during mutation, and the reverse when crossover is applied:

$P = 2 \times popSize \times I$

An area of memory equal to that required by one mirror of our population is needed to store our mutation matrix:

$M = \frac{P}{2}$

The number of declared variables available to the search is then defined by the user (v). From this we can calculate the space required for variable information. We multiply by two due to having separate arrays for writeable and read-only variables:

$V = 2 \times w \times v \times max(varNameLen)$

Finally we have the memory required to store a program in ASCII for the target language's syntax (S). For this we need to know the string length (s) needed per feature in the function matrix (e):

$s = \lambda(grammar, max(varNameLen))$

$e = f \times G$

$S = w \times popSize \times s \times e$

The total number of bits needed for our search is then given by:

$P + M + V + S$

With this ability of having a known memory size, we have a bounded search space from which to predict required computational memory resources for a GI run of certain parameters, giving human operators an indication as to what degree of freedom to give a potential search. Because we can describe the bounds of search space, it may also be possible to accurately estimate upper and lower bounds on the compute cost of a search process. We intend to examine both dimensions of predictability in future work.

7.3 Human Guidance Mechanisms

An under-explored approach in GI is the deployment of Human Guidance mechanisms to help steer a search through the high-dimensional space of programs [5] – for example to aid in overcoming areas of neutral drift. The goal of such an approach would be to build a toolset which allows a human engineer interact with the evolutionary process in real-time, imparting their intuition for software engineering onto the search.

Our novel program representation, and its probability distribution control interface, opens a wide range of possibilities in this research area. Firstly, matrices are inherently amenable to visual representation; simple form of this could use heatmap-like coloring to show coverage of the space so far. We can also directly translate sketched geometry on a matrix into probability distributions for the ongoing GI process, allowing operators to directly point to specific areas they would like the search to explore.

In future work we will explore a range of guidance approaches and how they interact with the serendipitous nature of GI.

8 RELATED WORK

Genetic Improvement for source code has been explored widely in software engineering over the last two decades, as covered in the survey by Petke et al in 2018 [15]. Our work intersects with that of GI and wider work on general program and learning representations. We also consider existing mechanisms to direct search trajectories, compared to our layered probability distributions.

8.1 GI program representations

In the GI domain, program representations tend to focus on Abstract Syntax Trees (ASTs) [2, 12, 19], bytecode [13], or BNF-inspired forms [16]. We conjecture that the use of these representational forms is largely due to their pre-existence in compilation processes, rather than their inherent suitability to GI.

ASTs are an intermediate compile stage when translating textual source code to executable machine code. GI for source code is often conceptualized as the mutation of operators or operands, or the transplanting of selected code from one individual to another (commonly used in crossover). At first glance, both procedures appear to fit well with an AST representation, where operators and operands are represented by nodes, and the transplanting of code can be achieved by injecting a sub-tree from one AST into another. However, because an AST does not include semantic information for things like data types, and does not embed the grammar of the language, extensive cross-checks are needed to verify whether or not a change is valid. AST representations also create an inherent bias to contain a GI search in the close vicinity of the starting individual, where each mutation is an incremental step away from this. While this is not necessarily a bad thing (it avoids steps into the extremely high-dimensional space of general program search), it tends to preclude more general expressions of search freedom.

GI approaches that operate on bytecode have the advantage of compressing the representation of a program, but again require associated rules on mutations that are viable within the instruction set of the specification, and require augmentation with data type information to avoid invalid operands being used in operations.

BNF-oriented representations are often used to augment the above approaches, and help to enforce syntax rules, lowering the likelihood of invalid programs.

In the context of the above, as far as we are aware our contribution is the first to support context-free mutations with guaranteed validity of the resulting code for a general-purpose programming language, and the first to support a highly configurable set of probability distributions to guide an ongoing search.

8.2 Alternative program representations

Outside of the GI domain, in more general program synthesis research, a set of alternative representations for programs has been examined. This includes graph-based representations of possible paths between input and output examples [8]; vector-based representations of potential operators [3]; and internal representations within neural network models [7].

The FlashFill approach, used within Microsoft Excel, synthesises programs using input/output examples [8]. Using a constrained domain-specific language, it represents potential programs by creating a set of so-called ‘traces’. A trace is a graph that contains all possible paths from on input example to it’s corresponding output example. An inductive algorithm is run to find the intersection of all of these graphs, yielding a final program which converts the given input to the given output example. This approach proves useful for inductive synthesis, but appears less ideal for the mutation and crossover dynamics of GI.

DeepCoder [3] uses a neutral network to make help synthesise programs from input/output examples of the desired function (such as {2, 2} as an input and {4} as an output). Using a domain-specific language comprised of relatively high-level operators, a numerical, vector-based representation of programs is used where each cell is a line of a program. The neural network is trained on various random input/output examples from random arrangements of cells in this vector, and is then able to predict likely values for unseen problems. Our approach is somewhat related in the numeric representation, but encompasses a general-purpose programming language complete with variable declarations and control-flow operators.

The Differential Neural computer [7] is an example of representing programs directly within the trained model of a neural network, again for the purpose of solving problems framed as input/output examples. This encoding allows neural training, similar to the previous approach, but makes it impossible to gain a symbolic representation of a program; it is also not clear that this approach would map to a GI domain.

While more exotic program representations have been explored, therefore, we are not aware of any that are targeted specifically at the GI domain, for general-purpose languages, and with the benefit of a link to symbolic representations of programs.

8.3 Search guidance approaches

Finally, our approach supports a range of search guidance approaches expressed as probability distributions over our matrix representation. Search guidance is a topic that is often visited in GI, most commonly to constrain the dimensionality of the search space to complete GI processes in reasonable time, while addressing the challenges of neutral drift [21, 22].

The main approaches to this are the use of phylogenetics to help analyze useful search trajectories [20], and templating or sketching to aid in isolating the parts of a program that are available for GI operations and those which should remain fixed [10, 24]. Our probability distribution approach supports many of these same constraint techniques, but through a unified paradigm; we therefore see research in this area as highly complementary to our method.

9 CONCLUSION

We have presented a geometric interpretation of program space, method to project irregular program space geometries to a uniform one. Using an associated GI framework, this allows us to arbitrarily mutate and crossover individuals and yield valid programs in solution space; it also supports the use of probability distributions to describe search trajectories. Our system performs comparably to other AST-based GI systems, and demonstrates predictable effects of different probability distributions.

In future work we will further explore the use of this paradigm across parallel architectures, predictability of search space and time, and human guidance approaches.

ACKNOWLEDGMENTS

This work was supported by the Leverhulme Trust Research Grant 'Genetic Improvement for Emergent Software', RPG-2022-109.

REFERENCES

[1] 2024. *Phenotypic Species Definitions for Genetic Improvement of Source Code.* Artificial Life Conference Proceedings, Vol. ALIFE 2024: Proceedings of the 2024 Artificial Life Conference. https://doi.org/10.1162/isal_a_00795

[2] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Applied Soft Computing* 11, 4 (2011), 3494–3514. https://doi.org/10.1016/j.asoc.2011.01.023

[3] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17*. https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[5] Benjamin Craine, Penn Rainford, and Barry Porter. 2024. Human Guidance Approaches for the Genetic Improvement of Software. In *Proceedings of the 13th ACM/IEEE International Workshop on Genetic Improvement* (Lisbon, Portugal) *(GI '24)*. Association for Computing Machinery, New York, NY, USA, 21–22. https://doi.org/10.1145/3643692.3648263

[6] Benjamin Doerr, Timo Kötzing, J. A. Gregor Lagodzinski, and Johannes Lengler. 2017. Bounding bloat in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 921–928. https://doi.org/10.1145/3071178.3071271

[7] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (Oct. 2016), 471–476. http://dx.doi.org/10.1038/nature20101

[8] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330. https://doi.org/10.1145/1925844.1926423

[9] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, Albert V. Smith, and Vilmundur Gudnason. 2017. Genetic improvement of runtime and its fitness landscape in a bioinformatics application. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 1521–1528. https://doi.org/10.1145/3067695.3082526

[10] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1127–1134. https://doi.org/10.1145/3205455.3205603

[11] William B. Langdon and Justyna Petke. 2017. Software is Not Fragile. In *First Complex Systems Digital Campus World E-Conference 2015*, Paul Bourgine, Pierre Collet, and Pierre Parrend (Eds.). Springer International Publishing, Cham, 203–211.

[12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[13] Michael Orlov and Moshe Sipper. 2009. Genetic programming in the wild: evolving unrestricted bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (Montreal, Québec, Canada) *(GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 1043–1050. https://doi.org/10.1145/1569901.1570042

[14] Justyna Petke, Brad Alexander, Earl T. Barr, Alexander E. I. Brownlee, Markus Wagner, and David R. White. 2019. A survey of genetic improvement search spaces. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1715–1721. https://doi.org/10.1145/3319619.3326870

[15] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[16] Justyna Petke, William B. Langdon, and Mark Harman. 2013. Applying Genetic Improvement to MiniSAT. In *Search Based Software Engineering*, Günther Ruhe and Yuanyuan Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–262.

[17] Barry Porter and Roberto Rodrigues Filho. 2021. A Programming Language for Sound Self-Adaptive Systems. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 145–150. https://doi.org/10.1109/ACSOS52086.2021.00036

[18] Penny Faulkner Rainford and Barry Porter. 2022. Code and Data Synthesis for Genetic Improvement in Emergent Software Systems. *ACM Trans. Evol. Learn. Optim.* 2, 2, Article 7 (Aug. 2022), 35 pages. https://doi.org/10.1145/3542823

[19] Penny Faulkner Rainford and Barry Porter. 2022. Code and Data Synthesis for Genetic Improvement in Emergent Software Systems. *ACM Trans. Evol. Learn. Optim.* 2, 2, Article 7 (aug 2022), 35 pages. https://doi.org/10.1145/3542823

[20] Penny Faulkner Rainford and Barry Porter. 2022. Using Phylogenetic Analysis to Enhance Genetic Improvement. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 849–857.

[21] Joseph Renzullo, Westley Weimer, Melanie Moses, and Stephanie Forrest. 2018. Neutrality and epistasis in program space. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop* (Gothenburg, Sweden) *(GI '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3194810.3194812

[22] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (Sept. 2014), 281–312. https://doi.org/10.1007/s10710-013-9195-8

[23] Sara Silva. 2011. Handling bloat in GP. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation* (Dublin, Ireland) *(GECCO '11)*. Association for Computing Machinery, New York, NY, USA, 1481–1508. https://doi.org/10.1145/2001858.2002146

[24] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 4–13.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[26] Peter A. Whigham and Grant Dick. 2010. Implicitly Controlling Bloat in Genetic Programming. *IEEE Transactions on Evolutionary Computation* 14, 2 (2010), 173–190. https://doi.org/10.1109/TEVC.2009.2027314