Exploring Genetic Programming Systems with MAP-Elites

Emily Dolson and Alexander Lalejini and Charles Ofria

Abstract MAP-Elites is an evolutionary computation technique that has proven valuable for exploring and illuminating the genotype-phenotype space of a computational problem. In MAP-Elites, a population is structured based on phenotypic traits of prospective solutions; each cell represents a distinct combination of traits and maintains only the most fit organism found with those traits. The resulting map of trait combinations allows the user to develop a better understanding of how each trait relates to fitness and how traits interact. While MAP-Elites has not been demonstrated to be competitive for identifying the optimal Pareto front, the insights it provides do allow users to better understand the underlying problem. In particular, MAP-Elites has provided insight into the underlying structure of problem representations, such as the value of connection cost or modularity to evolving neural networks. Here, we extend the use of MAP-Elites to examine genetic programming representations, using aspects of program architecture as traits to explore. We demonstrate that MAP-Elites can generate programs with a much wider range of architectures than other evolutionary algorithms do (even those that are highly successful at maintaining diversity), which is not surprising as this is the purpose of MAP-Elites. Ultimately, we propose that MAP-Elites is a useful tool for understanding why genetic programming representations succeed or fail and we suggest that it should be used to choose selection techniques and tune parameters.

Emily Dolson
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA e-mail: dolsonem@msu.edu

Alexander Lalejini
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA e-mail: lalejini@msu.edu

Charles Ofria
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Ecology, Evolutionary Biology, and Behavior Program, Michigan State University, East Lansing, MI, USA e-mail: ofria@msu.edu

1 Introduction

When programmers write code, they ideally want to structure it to be fast to implement, easy to extend, and clear for others to understand. Of course, these properties aren't usually compatible: program architectures that are fastest to write are usually not simple to extend or understand. We face a similar problem when we evolve programs with genetic programming (GP); the solutions that evolve most easily are typically a tangled mess. They are not useful as building blocks for solving more complex problems (*i.e.*, they are not evolvable), and they are challenging to tease apart what is going on. Because program architecture is so important for evolvability and decomposability, substantial effort goes into developing genetic programming systems that promote the evolution of programs with more evolvable architectures. For example, modularity is an important principle of software design and is also known to be an important component of evolvability (Kirschner and Gerhart, 1998), which has led many to design genetic programming systems with the goal of promoting the evolution of modular code (Koza, 1994; Walker and Miller, 2008; Spector et al., 2011).

Indeed, within the GP community, we have an abundance of ways to represent programs that we expect will be evolvable or interpretable, each with its own unique set of available programmatic elements and ways of organizing, interpreting, and mutating programs. Given the diversity of GP representations, understanding how to choose the most appropriate representation or configuration of a representation for a particular problem is an open issue in the field (O'Neill et al., 2010). Making headway on this issue requires expanding the existing toolkit of formal analyses for GP representations. While many different high-level properties of code can be considered, for the rest of this chapter we will focus on code evolvability.

In particular, it would be helpful to have a way to get insight into the range of program architectures that a given representation is capable of evolving. Doing so will help us disentangle issues related to the representation itself from issues with the way the rest of an evolutionary algorithm is set up (selection for evolvability is notoriously challenging to facilitate). Moreover, having access to examples of programs with different architectures is critical to setting up experiments that will tell us when these architectures are useful and how they interact with other features of a given representation. Ultimately, a tool for exploring program architectures would aid us in drawing generalizable insights that may be useful for others in the field.

The issue of wanting access to programs with a range of different architectures is an instance of a common problem. Often in evolutionary computation, we would like to evolve good solutions that are diverse with respect to a set of phenotypic traits (Pugh et al., 2015). For example, we might want to provide a variety of options to stakeholders making a decision (Chikumbo et al., 2012). Alternatively, we might want to provide a robot with alternative locomotion strategies to use if it gets damaged (Cully et al., 2015). MAP-Elites has been demonstrated to be an effective technique for evolving a diverse set of solutions to a problem (Mouret and Clune, 2015).

In MAP-Elites, the user selects some number of phenotypic axes that they expect will be relevant to solving the problem but might not be directly correlated with fitness. Each axis is then discretized into some pre-determined number of bins, resulting in a multi-dimensional grid where a location in the grid corresponds to a distinct combination of phenotypic traits. When a new solution is generated, its phenotype is assessed, and it is placed into a bin corresponding to that phenotype. If that bin is empty or occupied by a lower-fitness individual, the new solution replaces it. Otherwise, it is discarded.

Thus far, MAP-Elites had been used to evolve robot arms (Cully and Demiris, 2018), robot gaits (Cully et al., 2015), soft-bodied robots (Mouret and Clune, 2015), and neural networks for a computer vision task (Mouret and Clune, 2015). Interestingly, in this last task, the phenotypic axes (connection cost and modularity) related to the morphology of the neural networks themselves. Using these axes, MAP-Elites not only found a range of good solutions, but provided insight into the topology of the underlying fitness landscape as it relates to these two traits. The heat map produced by MAP-Elites shows which types of networks are capable of succeeding at the task and what constraints network traits place on each other. Here, we attempt to do the same for GP. Are there multiple programmatic paths to solving a problem? Can we identify inherent trade-offs between different traits (*e.g.*, modularity, instruction composition, *etc.*) in evolving programs? We see MAP-Elites as a tool that can be used to answer these questions by illuminating interactions between different aspects of a GP representation when applied to a problem. This increased understanding can help in building an intuition for what types of programs might be most appropriate for a given problem, which can be used to inform representation choice, population initialization, or mutation operators.

In this chapter, we demonstrate the use of MAP-Elites to explore a simple linear GP representation. By selecting phenotypic axes for MAP-Elites that correspond to program architecture and instruction composition, we can show how relevant different features of our GP representation are to the evolutionary process across a variety of problems. We compare the forms of programs evolved under MAP-Elites, lexicase selection, tournament selection, and random drift, demonstrating that MAP-Elites produces more varied solutions. Further, we discuss additional program traits that could be used as phenotypic axes in MAP-Elites that appear promising, but we have not yet explored in this work.

2 Methods

2.1 Computational Substrate

For this study, we evolve linear genetic programs where each program is a linear sequence of instructions, and each instruction has up to three arguments that may modify its behavior. Most notably, our linear genetic programming (LGP) computational substrate supports subroutines, allowing programs with modular architectures to evolve.

Making efficient use of modular subroutines has long been thought to be important to facilitating the evolution of genetic programs that solve complex problems. Indeed, incorporating modules into GP has been extensively explored, and their benefits have been well documented (*e.g.*, (Koza, 1994, 1992; Angeline and Pollack, 1992; Keijzer et al., 2005; Walker and Miller, 2008; Roberts et al., 2001; Spector, 1996; Spector et al., 2011)). We designed our LGP representation to facilitate the evolution of modular and reusable code while minimally affecting the way in which traditional linear genetic programs are organized (*i.e.*, linear sequences of instructions). We include a number of instructions in the language that automatically create programming structures like loops and subroutines, and we enable these features through the concept of "scopes", which are functionally similar to "environments" in Push (Spector, 2001; Spector and McPhee, 2018).

2.1.1 Virtual CPU Hardware

Our linear genetic programs are executed in the context of a virtual CPU with the following components:

- **Instruction Pointer:** A marker to indicate the position in the genome currently being executed. Many instructions will influence how the instruction pointer (IP) moves through the genome.
- **Registers:** Each virtual CPU has 16 registers. Programs can store a single floating-point value in each register. Registers are initialized with numbers corresponding to their ID (*e.g.*, register 0 is initialized to the value 0.0, register 1 is initialized to the value 1.0, and so on).
- **Stacks:** Each virtual CPU has 16 stacks. Programs can push floating point numbers onto these stacks and pop them off later.
- **Inputs:** Each virtual CPU can accept an arbitrary number of input values. These values do not need to be in any particular order, but each value needs to be associated with a unique numerical label that the program can use to access it. For the purposes of this paper, we always use sequential integers starting at 0.
- **Outputs:** Outputs function the same way as inputs. The only difference between inputs and outputs is the way instructions interact with them; whereas instructions can read from inputs but not write to them, instructions can write to outputs but not read from them.
- **Scopes:** Each virtual CPU has 16 scopes (plus the global scope), described in the next section.

2.1.2 Scopes

In software development, the *scope* of a variable specifies the region of code in which that element may be used. In a sense, a scope is like a programmatic membrane, capable of encapsulating all manner of programmatic elements, such as variables, functions, *et cetera*, and allowing regions to be looped through or skipped entirely. Our LGP representation gives evolving programs control over instruction- and memory-scoping, allowing programs to easily manage flow control and variable lifetime.

In our LGP representation, scopes provide the backbone on top of which all of the other modularity-promoting features, such as loops and functions, are built. All instructions in a program exist within a scope, be it the default outermost scope or one of the 16 other available scopes (making 17 possible scopes) that can be accessed via various instructions. These 16 inner scopes have a hierarchy to them, such that higher-numbered scopes are always nested inside lower-numbered scopes.

At the beginning of the program, all instructions before the first change of scope are in the outermost scope. After a scope-changing instruction occurs in the genome, subsequent instructions are added to the new scope until another scope-changing instruction is encountered, and so on. These scopes are ordered numerically. Higher-numbered scopes are always nested inside lower-numbered scopes. Scopes can be exited with the `break` instruction or by any instruction that moves control to a lower-numbered scope.

Scopes are also the foundation of program modules (functions). The `define` instruction allows the program to put instructions into a scope and associate the contents of that scope with one of 16 possible function names. Later, if that function is called (using the `call` instruction), the program enters the scope in which that function was defined and executes the instructions within that scope in sequence, including any internal (nested) scopes.

Similarly, scopes are the foundation of loops. Two kinds of loops exist in the instruction set used here: while loops and countdown loops. Loops of both types have a corresponding scope, which contains the sequence of instructions that make up the body of the loop. Both types of loops repeat their body (*i.e.*, the contents of their associated scope) until the value in an argument-specified register is 0. Countdown loops automatically decrement this register by one on every iteration. When any instruction is encountered that would cause the program to leave the current scope, the current iteration is ended and the next one begins.

2.1.3 Instructions

In this work, evolving programs can contain the following library of 26 different instructions. For each, instruction arguments are limited to 16 values (0 through 15) and are used to specify any of the following: registers, scopes, functions, inputs, or outputs. Arguments for each instruction and their types are shown in the parentheses after each instruction name.

- **`Inc`** *(Register A)*: Increment the value in register A by 1.
- **`Dec`** *(Register A)*: Decrement the value in register A by 1.
- **`Not`** *(Register A)*: If Register A equals 0.0, set to 1.0. Otherwise set to 0.0.
- **`SetReg`** *(Register A, Value B)*: Store numerical value of B into register A.
- **`Add`** *(Register A, Register B, Register C)*: Add the value in register A to the value register B and store the result in register C.
- **`Sub`** *(Register A, Register B, Register C)*: Subtract the value in register B from the value in register A and store the result in register C.
- **`Mult`** *(Register A, Register B, Register C)*: Multiply the value in register A by the value in register B and store the result in register C.
- **`Div`** *(Register A, Register B, Register C)*: Divide the value in register A by the value in register B and store the result in register C.
- **`Mod`** *(Register A, Register B, Register C)*: Calculate the value in register A modded by the value in register B and store the result in register C.
- **`TestEqu`** *(Register A, Register B, Register C)*: Store the value 1.0 in register C if the value in register A is equal to value in register B. Otherwise store the value 0.0 in register C.
- **`TestNEqu`** *(Register A, Register B, Register C)*: Store the value 0.0 in register C if the value in register A is equal to value in register B. Otherwise store the value 1.0 in register C.
- **`TestLess`** *(Register A, Register B, Register C)*: Store the value 1.0 in register C if the value register A is less than the value in register B. Otherwise store the value 0.0 in register C.
- **`If`** *(Register A, Scope B)*: If the value in register A is not 0.0, continue to scope B, otherwise skip scope B.
- **`While`** *(Register A, Scope B)*: Repeat the contents of scope B until the value in register A is equal to 0.
- **`Countdown`** *(Register A, Scope B)*: Repeat the contents of scope B, decrementing the value in register A each time, until the value in register A is 0.
- **`Break`** *(Scope A)*: Break out of scope A.
- **`Scope`** *(Scope A)*: Enter scope A.
- **`Define`** *(Function A, Scope B)*: Define this position as the starting point of function A, with its contents defined by scope B. The function body is skipped after being defined; when called, the function automatically returns when scope B is exited.
- **`Call`** *(Function A)*: Call function A (must have already been defined).
- **`Push`** *(Register A, Stack B)*: Push the value in register A onto stack B.
- **`Pop`** *(Stack A, Register B)*: Pop the top value off of stack A and store it in register B.
- **`Input`** *(Input A, Register B)*: Store the value in input A in register B.
- **`Output`** *(Register A, Output B)*: Write the value in register A to output B.
- **`CopyVal`** *(Register A, Register B)*: Copy the value in register A into register B.

- **`ScopeReg`** *(Register A)*: Backup the value in register A. When the current scope is exited, it will be restored.
- **`Dereference`***(Register A, Register B)*: Store the value of the register specified by the value of register A in register B.

2.2 Evolution

2.2.1 Selection operators

MAP-Elites

The Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm is designed to illuminate search spaces and has been demonstrated as an effective technique for evolving a diverse set of solutions to a problem (Mouret and Clune, 2015). In MAP-Elites, a population is structured based on a set of chosen phenotypic traits. Each chosen trait defines an axis on a grid of cells where each cell represents a distinct combination of the chosen traits. Cells maintains only the most fit (elite) solution discovered with that cell's associated combination of traits. A MAP-Elites grid is initialized by randomly generating solutions and placing them into their appropriate cell in the grid (based on the random solution's traits). After initialization, occupied cells are randomly selected to reproduce. When a solution is selected for reproduction, we generate a mutated offspring and determine where that offspring belongs in the grid. If the cell is unoccupied, the new solution is placed in that cell; otherwise, we compare the new solution's fitness to that of the current occupant, keeping the fitter of the two. Over time, this process produces a grid of solutions that span the range of traits we used to define our grid axes.

Tournament selection

To understand whether it's valuable to use MAP-Elites to explore the range of potential program architectures (rather than simply looking at the architectures evolved under the selection scheme that is already being used), we need to compare it to a more standard approach to selection. Here, tournament selection with a tournament size of two will serve as that control. Any time we need to generate an offspring program using tournament selection, we select two random programs from the population and allow the fitter one to reproduce. We intentionally selected the lowest possible tournament size to minimize the strength of selection, facilitating as much diversification as possible.

Lexicase selection

Tournament selection is known to be bad at maintaining diversity in a population. Lexicase selection (Spector, 2012) is known to maintain phenotypic diversity, although little is known about the diversity of program architectures within these populations. To better understand whether our proposed use of MAP-Elites is more effective at exploring the range of possible program architectures than simply maintaining a diverse population, we compare it to the results of using standard lexicase selection.

In lexicase selection, all of the test cases for a problem are randomly re-ordered each time another program is being selected to reproduce. We then go through the test cases in order and, for each test case, remove all but the top performing programs from the set of candidates for selection. When there is only one program remaining, we allow it to reproduce. In case of a tie we choose randomly.

Random drift

What types of program architectures arise in the absence of selection pressure (*i.e.*, from purely random drift)? We additionally compare the range of evolved program architectures produced by MAP-Elites with those produced under random drift where we select programs to reproduce at random. Although we do not expect any of these programs to actually solve any of our test problems, they will provide insight into large scale statistical trends that we might expect in the absence of selection.

2.2.2 Mutation operators

In this work, we propagated programs asexually and applied consistent rates of mutations to offspring across all treatments. We used four different operators to introduce mutations on reproduction: instruction substitutions, argument substitutions, point insertions, and point deletions (Brameier and Banzhaf, 2007). Instruction substitutions, in which one instruction was randomly replaced with another instruction, had a 0.005 chance of occurring at each site in the genome. Argument substitutions, in which one argument to an instruction was randomly replaced with another, had a 0.005 chance of occurring for each argument in the genome. In point insertions, a random instruction was added after a given site, increasing the length of the genome. Conversely, point deletions removed the instruction at a given site, decreasing the length of the genome. Point insertions and deletions both had a 0.005 chance of happening at every site in the genome.

2.3 Experimental Design

Is MAP-Elites an effective technique for exploring how different aspects of evolving program architectures interact to affect performance in GP? In this work, we evolve linear genetic programs with MAP-Elites to solve four simple programming synthesis problems, selecting phenotypic axes that correspond to program architecture and instruction composition. For each problem, we compare the types of programs evolved with MAP-Elites and with lexicase and tournament selection; additionally, we compare the types of programs evolved with MAP-Elites to programs produced via random drift.

2.3.1 MAP-Elites Phenotype Axes

As a proof-of-concept, we have selected two phenotypic axes that we expected to promote a useful exploration of the features of our linear GP representation. In Section 4, we discuss additional axes that may also prove to be generally useful for exploring GP representations.

Program Composition
A representation's instruction set has a huge impact on what programs are able to do. However, predicting the importance of an instruction *a priori* can be challenging. MAP-Elites can help in understanding instructions' importance in various contexts. Theoretically, a wide variety of phenotypic traits related to instructions could be used. For example, for any individual instruction, the number of times it is used could be an axis.

For the purposes of getting a high-level understanding of the range of programs that can evolve, however, we have chosen to use the overall diversity of instructions in a program as an axis. Here, we quantify the diversity of instructions in a program using Shannon entropy. This measurement provides high-level information about the genotype as whole. Importantly, it cannot be easily altered through small numbers of neutral mutations, meaning it should be informative about practical differences between genomes. We discretize this value into 20 bins between 0 and the maximum possible entropy for a program.

Module Use
Our representation is centered around modules in the form of scopes. As we attempt to understand whether this programming paradigm is useful to evolution, it is critical to understand the extent to which scopes are used. Thus, we chose the number of scopes used by a program as our second phenotypic axis. Importantly, we only counted scopes that were actually used; when a program is run, it must execute at least one instruction in a given scope to get credit for using that scope. This, however, does not guarantee that scopes are used *meaningfully*, only that they are used. Since this measurement is already an integer value, we used 17 bins along this axis so that each bin corresponds to a different possible number of scopes.

2.3.2 Test Problems

All problems except the logic problem were defined by a set of test cases in which programs were given specified input data and were scored on how close their output was to the correct output. For MAP-Elites and tournament selection, we calculated fitness as the sum of scores on these test cases.

- **Logic:** Programs receive two integers in binary form and must output the results of doing bitwise logic operations on them. We reward 10 2-input (with the exception of ECHO, which is 1-input) logic operations: ECHO, NOT, NAND, OR-NOT, AND, OR, AND-NOT, NOR, XOR, and EQUALS. To facilitate the

evolution of these computations, we added a `Nand` instruction to the instruction set, which converts inputs to integers and then performs a bitwise not-and operation, structured in the same way as the `Add` instruction. Every unique logic operation that a program outputs the solution to over the course of its execution increases that program's score by 1. Once a program has solved all of the logic problems, it gets bonus points for the speed with which it solved them. Specifically, the bonus is equal to the total number of allowed instruction executions minus the number of instructions the program actually executed before performing all 10 logic tasks. For lexicase selection, each logic operation was treated as a different test case.
- **Squares:** Programs receive an integer as input and must output its square. Because this problem is known to be easy, we evaluated programs on just 11 test cases.
- **Sum:** Programs receive a list of five integers as input that they must add together and output their total. Programs were evaluated on a set of 200 test cases.
- **Smallest:** Programs receive a list of four integers as input and must output the smallest one (from (Helmuth and Spector, 2015)). Programs were evaluated on a set of 200 test cases.

2.3.3 Experimental Parameters

We ran 30 replicates per condition for 50,000 generations. In conditions where tournament selection, lexicase selection, or random drift is used to determine which programs reproduce, we maintained a population size of 1,000 programs. The maximum population size in MAP-Elites, however, depends on the number and resolution of phenotypic trait axes used to define the MAP-Elites grid. Thus, in conditions that use MAP-Elites, the maximum population size is 340. However, in our MAP-Elites conditions, we define a single generation to be equal to 1,000 reproduction events, which ensures that all conditions experience the same total number of reproduction events.

We initialized the population by generating random programs of random lengths. Programs could not be less than 1 instruction long or greater than 1024 instructions long. Each program was evaluated by executing its instructions in sequence until an upper limit was hit (128 instruction executions for the squares and logic problems; 512 instruction executions for the sum and smallest problems).

2.3.4 Data Analysis

To quantify the different ranges of program architectures explored by our different selection operators we look at the population in the final generations of all of our replicates and filter out all programs that do not fully solve the problem (*i.e.*, those that do not score perfectly on all test cases or, in the context of the logic problem, do not perform all of the logic operations). However, in the random drift condition, we do not filter any programs from the final population, as it is unreasonable to expect that they would have solved the problems. We then look at the distribution of values for each of our phenotypic axes and compare them across selection schemes using a Kolmogorov-Smirnov test to tell us whether MAP-Elites produces a significantly different distribution of program architectures than other selection schemes. To correct for the number of Kolomogorov-Smirnov tests we perform (one for each alternative selection operator that we compare MAP-Elites to, for both scope count and instruction entropy), we use a Bonferonni correction. All data analysis was performed using the R Statistical Computing Platform (R Core Team, 2017), and all data visualization was done using the ggplot2 package (Wickham, 2009). We used the implementation of the Kolmogorov-Smirnov test in the dgof package, as it is able to properly handle discrete variables, such as scope count (Arnold and Emerson, 2011).

2.3.5 Code availability

All code used to generate and analyze the data presented here is open source and publicly available (Lalejini and Dolson, 2018). This code makes heavy use of the Empirical library, which is also open source and publicly available (Ofria et al., 2018)

3 Results and Discussion

The distributions of scope count and instruction entropy values for programs evolved using MAP-Elites were dramatically different from those of programs evolved using other selection schemes (Kolomogorov-Smirnov test, $p < 0.0001$). As is qualitatively evident in Figure 1 and Figure 2, the range of each of the metrics in programs evolved with MAP-Elites is much wider than the range for programs evolved under other selection methods (with the exception in some cases of random drift, which was not subject to the requirement that programs actually solve the problem). While this result is not surprising, it provides confirmation that using MAP-Elites as a tool for exploring GP representations provides information we would not otherwise obtain.

The distribution of metrics evolved under MAP-Elites can suggest the presence of constraints. For example, there generally seems to be some cut-off instruction entropy below which solutions are hard (or impossible) to find. The location of this cut-off varies by problem. This result makes intuitive sense; to achieve the minimum possible instruction entropy, 0, a program would have to consist of a single type of instruction. Any successful solution in this experiment would minimally need to include both the `Input` instruction and the `Output` instructions, as well as some instructions that perform actual calculations. Thus, no successful program could possibly have an instruction entropy of 0. The same logic applies to other low values of instruction entropy. The lack of other empty spots in the distributions of instruction entropy and scope count generated by MAP-Elites indicates that there are not other constraints on these aspects of program architecture; in essence, MAP-Elites has provided an existence proof for programs with these various properties. Note that if we had tried to make the same inference based on one of the other selection operators, we may have been mislead. Of course, even MAP-Elites is not guaranteed to find all possible successful program structures. It just comes closer to doing so.

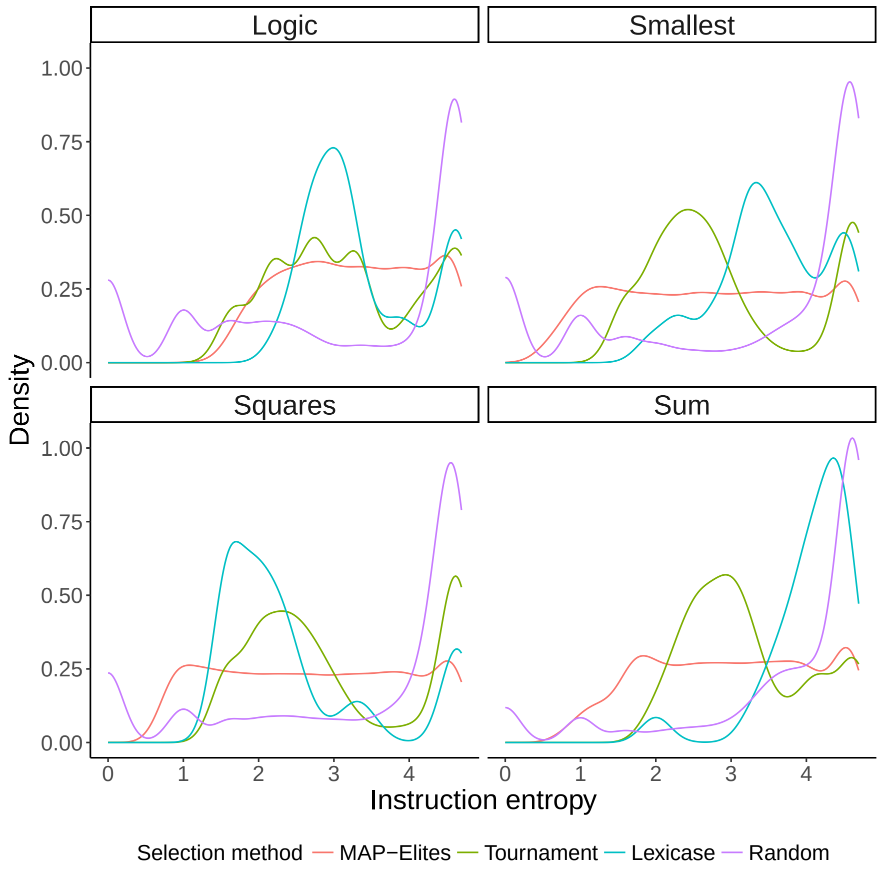

Fig. 1 Distribution (as density plot) of instruction entropy metric across all perfect solutions from the final generation of all replicates for each selection scheme.

Are there interactions between our two phenotypic traits? We can illuminate possible interactions between phenotypic traits by making heat maps showing the number of solutions found with each distinct combination of traits across all replicates of a condition. To compare the interactions discovered by MAP-Elites to the inter-

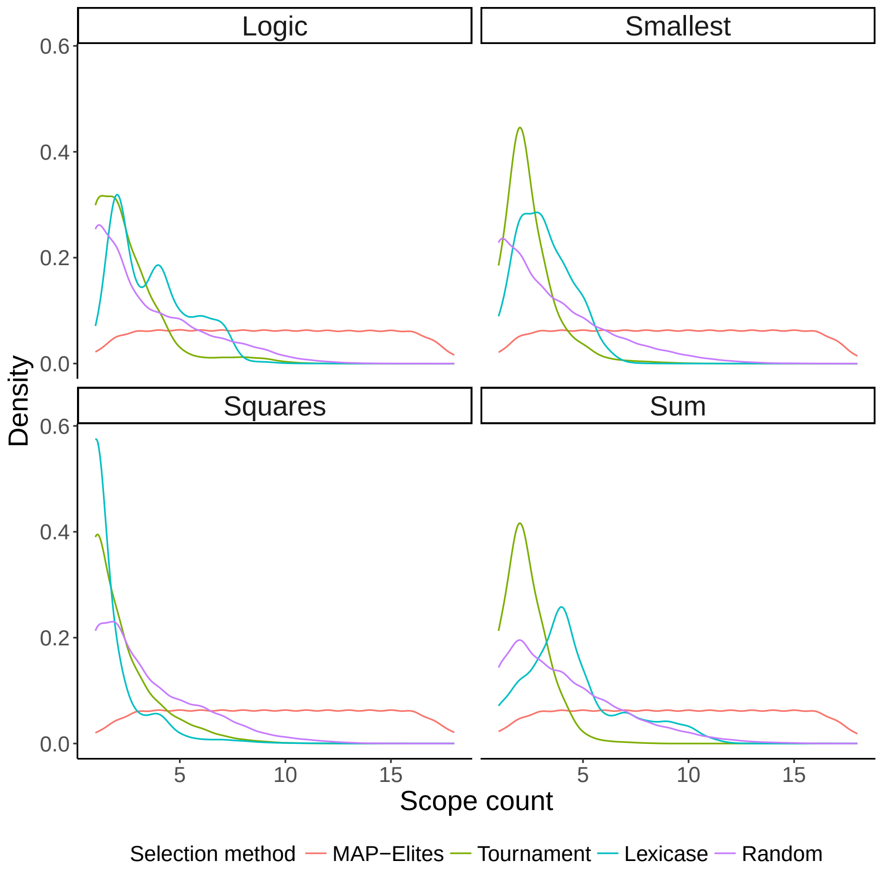

Fig. 2 Distribution (as density plot) of scope count metric across all perfect solutions from the final generation of all replicates for each selection scheme.

actions we might have inferred from looking at the programs generated by other selection schemes, we made a set of heat maps for each selection operator on each problem (see Figure 3). The fact that the heat maps for MAP-Elites are almost completely filled in (with the exception of bins at low instruction entropy) suggests that there are no substantial interactions between instruction entropy and scope count. Note that this is counter to the conclusion we would draw by looking at the results of any of the other selection operators, all of which would seem to suggest that programs with high scope count and low instruction entropy are hard to find. Since random drift displays the same pattern, this is likely some sort of statistical artifact of the types of programs that are most likely to be assembled by chance. Using MAP-Elites to explore the space of program representations allows us to distinguish this artifact from an actual constraint. For an example of using this technique to uncover actual constraints in program architectures, see (Lalejini and Ofria, 2018).

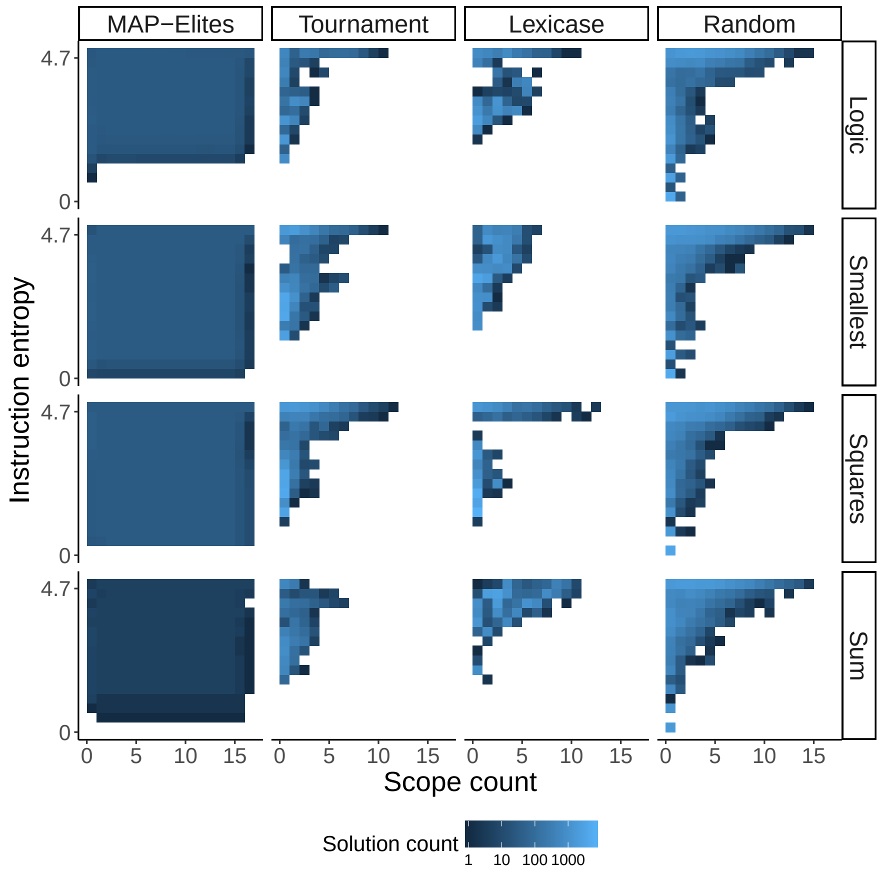

Fig. 3 Heat maps showing the number of perfect solutions from the final generation across all replicates for each problem and selection scheme falling into each phenotypic bin.

4 Conclusion

We have demonstrated the use of MAP-Elites as a tool for exploring simple linear GP representations. By selecting phenotypic axes for MAP-Elites that correspond to aspects of program architecture, we can build an intuition for how relevant different features of a GP representation are to the evolutionary process across a variety of problems. These types of analyses are important as the GP community continues to develop and characterize increasingly expressive representations.

In this work, we limited our selection of MAP-Elites phenotypic axes to instruction entropy and module use; however, there are many possible informative axes. Further, MAP-Elites is not limited to just two axes. We could select any number of traits with which to define axes, allowing us to explore how many different aspects of a GP representation interact in the context of a given problem. There are a wide range of possible metrics that we could use in this context.

In genetic programming, it makes sense to evaluate the composition of instructions (or operations in the context of tree or graph-based GP) in the genome. We can either evaluate the composition of all instructions in the genome, or only those instructions that are actually executed. Such analyses can be performed using the following metrics:

- Total number of instructions in the program (length)
- Total number of unique instruction types in the program
- Entropy of instructions (as used in this paper)
- The number of times a given instruction type was used
- The entropy of numbers of times that instruction types were used
- The average effective dependence distance of instructions (Brameier and Banzhaf, 2007)

We might also care about more abstract attributes of program architecture. For example, there are many quantities related to modularity that it may be informative to measure, particularly in light of the fact that modularity is thought to promote evolvability. The simplest of these is to measure the modularity of the program using metrics such as the physical and functional modularity metrics described in (Misevic et al., 2006). In cases where modules are easy to identify, we can probe further with the following metrics:

- Total number of modules in the program
- Total number of times any module is used
- Total number of unique modules that are used (equivalent to scope count in this paper)
- Entropy of time spent in each module

For representations with a concept of memory positions, it may be useful to measure the way the program makes use of them:

- Number of effective memory locations (Brameier and Banzhaf, 2007)
- Entropy of memory locations used
- The number of times a given individual memory location was referenced/accessed
- Modularity of memory used

There are also a wide range of potentially useful metrics that will depend on the specifics of the genetic programming representation being used. For example, trees and graph-based programs (*e.g.*, Cartesian genetic programming) can be assessed with metrics that describe their topology. Representations that have linear genomes, on the other hand, can likely borrow a variety of useful metrics from biology.

We have demonstrated that using MAP-Elites with phenotypic axes based on program architecture can illuminate constraints on program architecture that we would have been unaware of simply from examining the programs generated by traditional selection operators. Understanding these constraints can help us understand why certain genetic programming representations are more or less successful under certain circumstances, an important goal for the long-term advancement of the field (O'Neill et al., 2010). Thus, we expect that the approach presented here will be a useful addition to the toolkit we use to study genetic programming.

5 Acknowledgements

We thank members of the MSU Digital Evolution Lab for helpful comments and suggestions on this manuscript. This research was supported by the National Science Foundation (NSF) through the BEACON Center (Cooperative Agreement DBI-0939454), Graduate Research Fellowships to ED and AL (Grant No. DGE-1424871), and NSF Grant No. DEB-1655715 to CO. Michigan State University provided computational resources through the Institute for Cyber-Enabled Research and the Digital Scholarship Lab. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or MSU.

References

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241.

Arnold, T. A. and Emerson, J. W. (2011). Nonparametric Goodness-of-Fit Tests for Discrete Null Distributions. *The R Journal*, 3(2):34–39.

Brameier, M. and Banzhaf, W. (2007). *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer US, Boston, MA.

Chikumbo, O., Goodman, E., and Deb, K. (2012). Approximating a multi-dimensional Pareto front for a land use management problem: A modified MOEA with an epigenetic silencing metaphor. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–9.

Cully, A., Clune, J., Tarapore, D., and Mouret, J.-B. (2015). Robots that can adapt like animals. *Nature*, 521(7553):503.

Cully, A. and Demiris, Y. (2018). Quality and Diversity Optimization: A Unifying Modular Framework. *IEEE Transactions on Evolutionary Computation*, 22(2):245–259.

Helmuth, T. and Spector, L. (2015). General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1039–1046, New York, NY, USA. ACM.

Keijzer, M., Ryan, C., Murphy, G., and Cattolico, M. (2005). *Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kirschner, M. and Gerhart, J. (1998). Evolvability. *Proceedings of the National Academy of Sciences*, 95(15):8420–8427.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA.

Lalejini, A. and Dolson, E. (2018). amlalejini/GPTP-2018-Exploring-Genetic-Programming-Systems-with-MAP-Elites: Initial Release. DOI: 10.5281/zenodo.1345799.

Lalejini, A. and Ofria, C. (2018). What else is in an evolved name? Exploring evolvable specificity with SignalGP. *PeerJ Preprints 6:e27122v1*, pages 1–21.

Misevic, D., Ofria, C., and Lenski, R. E. (2006). Sexual reproduction reshapes the genetic architecture of digital organisms. *Proceedings of the Royal Society B: Biological Sciences*, 273(1585):457–464.

Mouret, J.-B. and Clune, J. (2015). Illuminating search spaces by mapping elites. *arXiv:1504.04909 [cs, q-bio]*. arXiv: 1504.04909.

Ofria, C., Dolson, E., Lalejini, A., Fenton, J., Jorgensen, S., Miller, R., Moreno, M., Stredwick, J., Zaman, L., Schossau, J., leg2015, cgnitash, and V, A. (2018). amlalejini/Empirical: GPTP 2018 - Exploring Genetic Programming Systems with MAP-Elites. DOI: 10.5281/zenodo.1346397.

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

Pugh, J. K., Soros, L. B., Szerlip, P. A., and Stanley, K. O. (2015). Confronting the Challenge of Quality Diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 967–974, New York, NY, USA. ACM.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Roberts, S. C., Howard, D., and Koza, J. R. (2001). Evolving modules in Genetic Programming by subtree encapsulation. *Genetic Programming, Proceedings of EuroGP'2001*, 2038:160–175.

Spector, L. (1996). Simultaneous Evolution of Programs and their Control Structures. *Advances in Genetic Programming 2*, pages 137–154.

Spector, L. (2001). Autoconstructive Evolution: Push, PushGP, and Pushpop. *Gecco*, pages 137–146.

Spector, L. (2012). Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In *Proceedings* *of the 14th annual conference companion on Genetic and evolutionary computation*, pages 401–408. ACM.

Spector, L., Martin, B., Harrington, K., and Helmuth, T. (2011). Tag-based modules in genetic programming. *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1419–1426.

Spector, L. and McPhee, N. F. (2018). Expressive genetic programming: concepts and applications. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '18*, pages 977–997, Kyoto, Japan. ACM Press.

Walker, J. A. and Miller, J. F. (2008). The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417.

Wickham, H. (2009). *ggplot2: elegant graphics for data analysis*. Springer New York.