A detailed analysis of a PushGP run

Nicholas Freitag McPhee, Mitchell D. Finzel, Maggie M. Casale, Thomas Helmuth and Lee Spector

Abstract In evolutionary computation we potentially have the ability to save and analyze every detail in an run. This data is often thrown away, however, in favor of focusing on the final outcomes, typically captured and presented in the form of summary statistics and performance plots. Here we use graph database tools to store every parent-child relationship in a single genetic programming run, and examine the key ancestries in detail, tracing back from an solution to see how it was evolved over the course of 20 generations. To visualize this genetic programming run, the ancestry graph is extracted, running from the solution(s) in the final generation up to their ancestors in the initial random population. The key instructions in the solution are also identified, and a genetic ancestry graph is constructed, a subgraph of the ancestry graph containing only those individuals that contributed genetic information (or instructions) to the solution. These visualizations and our ability to trace these key instructions throughout the run allow us to identify general inheritance patterns and key evolutionary moments in this run.

Nicholas Freitag McPhee
University of Minnesota, Morris; Morris, MN USA e-mail: mcphee@morris.umn.edu

Mitchell D. Finzel
University of Minnesota, Morris; Morris, MN USA e-mail: finze008@morris.umn.edu

Maggie M. Casale
University of Minnesota, Morris; Morris, MN USA e-mail: casal033@morris.umn.edu

Thomas Helmuth
Washington and Lee University, Lexington, VA USA e-mail: helmutht@wlu.edu

Lee Spector
Hampshire College, Amerherst, MA USA e-mail: lspector@hampshire.edu

1 Introduction

Previous work [10, 5, 2, 1, 3, 4, 9] has illustrated the value of ancestry graphs as a means of analyzing the dynamics of evolutionary computation runs. In [10], for example, we demonstrated the use of graph databases as a tool for collecting and analyzing ancestries in genetic programming runs, identifying several key moments and general patterns in runs using both lexicase and tournament selection.

In this paper we extend that work to provide a more detailed analysis of a single, complete run. We identify *every* ancestor of the evolved solutions, and then reduce that graph (which has 394 individuals) to a graph containing only the individuals that in fact contributed one of the key instructions to the final solutions (73 individuals). We then trace each of these key solution instructions back through the entire lineage, identifying where they were first introduced, and how they were transmitted through the genetic history. This reveals a number of interesting properties of this particular run including, for example, the fact that 4 of the 9 key instructions were introduced via mutation and most crossover events led to changes that could have been brought about by mutation alone.

In Section 2 we review the key components of the system used to generate the run explored here (PushGP, Plush genomes, and the Replace Space With Newline test problem). We then describe and present both the full and genetic ancestry graphs in Section 3, before tracing the evolutionary history of all the key instructions in Section 4. Our discussion in Section 5 builds on the details of these traces and catalogues the kinds of events we see in this run, describing a few in greater detail. We then wrap up with some conclusions and ideas for future work in Section 6.

2 Languages, configuration, tools and setup

The run presented here was generated using a Clojure implementation[1] of the PushGP[2] genetic programming system, which evolves programs in the Push programming language [15, 14]. Push programs use typed stacks to store and manipulate data, taking their arguments from stacks of the appropriate types and leave their results on the appropriate stacks.

Push gains much of its power as an evolutionary language from its ability to manipulate code, including the currently executing code, as a program runs. The running program is stored on the *exec* stack, allowing instructions to change code before it runs. Push programs are hierarchically structured into code blocks delimited by parentheses. Each code block is treated as a single unit when code manipulating instructions act on them.

Unlike previous versions of PushGP, Clojush has recently been changed to not evolve Push programs directly, but to act instead on a new linear genome repre-

[1] Clojush: https://github.com/lspector/Clojush

[2] http://pushlanguage.org/

sentation [8]. Each *Plush (**Linear Push**) genome* consists of a linear sequence of instructions (including literals), and is translated into a Push program prior to execution. Each instruction may have one or more epigenetic markers attached that modify how the genome is translated into a Push program. For more details on the Plush genome representation and operators, see [8].

Most relevant to this study is the *close* epigenetic marker, which affects the hierarchical composition of programs. Since many Push instructions do not act on code blocks from the *exec* stack, it makes sense to limit the appearance of code blocks to follow only instructions that do make use of them. Each instruction that takes one or more argument from the *exec* stack automatically opens one or more code blocks. Then, the integer close marker attached to each instruction tells how many opened code blocks to close after that particular instruction. During translation from Plush genome to Push program, an open parenthesis is placed after each instruction that requires a code block, and a matching closing parenthesis is placed after a later instruction with a non-zero close marker. These code blocks can create hierarchically nested Push programs, allowing, for example, structures such as nested looping and subroutines containing conditional code.

This run used *lexicase selection* [11, 7, 6]. The details of lexicase selection aren't crucial here, but it is important to know that lexicase selection avoids aggregating test case performance (by, for example, computing a single total error as is common when using tournament selection), and instead maintains a vector of distinct errors for each test case. This allows an individual that performs well on a few test cases that the population is generally poor at to be selected, often multiple times, even if it performs very poorly on other test cases.

Our main crossover operator, *alternation*, similar to N-point crossover in genetic algorithms. Alternation traverses two parents in parallel while copying instructions from one parent or the other to the child. While traversing the parents, copying can jump from one parent to the other with probability specified by the *alternation rate* parameter. When alternating between parents, the index at which to continue copying may be offset backward or forward some small random amount.

We also use a *uniform mutation* operator that traverses a parent, replacing each instruction with some small probability. Similarly, a *uniform close mutation* operator can change the close epigenetic marker attached to an instruction by incrementing or decrementing it. Finally, we often apply an alternation operator followed by a uniform mutation of the result, inspired by the ULTRA operator [12].

Clojush also implements a method of automatic simplification, which takes a program and converts it into a smaller, semantically equivalent program. This process uses hill-climbing to remove instructions and code blocks from the program, checking at each step that the resulting program produces the same error vector as the original program [13]. This can dramatically simplify programs, reducing, for example, one program from 194 instructions down to 9 instructions.

Our goal here is to give a deep analysis of a single run of PushGP, exploring and analyzing many of the programs, selections, and variations that make up this run. We chose to analyze a run on the Replace Space With Newline (RSWN) problem, taken from a recent general program synthesis benchmark suite [7]. In this problem,

a program is given a string as input and should perform two tasks: first, it must print the result of replacing each space in the input with a newline character, and second, it must functionally return the number of non-whitespace characters in the input by leaving that value on top of the *integer* stack.

To store and process our ancestry data we used the Titan graph database along with the Gremlin shell and the Apache Tinkerpop query language.[3] This allowed us to store information about nodes (individuals), such as genomes and error vectors, and edges (relations) such as parent-child relationships. The graph database tools then make it easy to trace lineages and extract the subgraphs visualized in the next section. For these visualizations we used the Graphviz `dot` graph layout tool .[4]

3 Ancestry graphs

The run we analyze here used a population size of 1,000. This particular run found a solution after 20 generations, so we stored a total of 21,000 individuals in the graph database for this run. There were thirteen different "winning" individuals in that final generation, each of which had zero error on all of the 200 training cases.

In this section we describe two techniques for extracting and visualizing aspects of the run. The first is the ancestry tree, which contains of every ancestor (e.g., parents, grandparents, etc.) of any individual who found a solution. The second is the genetic ancestry tree, which is the subset of the ancestry tree limited to just those individuals that contributed at least one instruction to a particular successful individual.

3.1 Full ancestry graph

Figure 1 shows the full ancestry tree of the 13 successful individuals in this run. Each individual is represented with a rectangle containing an identifier of the form `X:Y`, where `X` is the generation number, and `Y` is an arbitrary individual number within that generation. Each generation is a row, with the initial random individuals being at the top and the 13 successful individuals at the bottom.

The edges indicate the particular genetic operator used to construct a child:

- Dashed: alternation
- Dotted: uniform mutation
- Thin black lines: uniform close mutation
- Thick black lines: alternation followed by uniform mutation

The graph in Figure 1 includes *every* individual in this run that was an ancestor of one of the winners, i.e., every individual that could possibly have contributed

[3] http://thinkaurelius.github.io/titan/ and https://tinkerpop.apache.org/

[4] http://www.graphviz.org/

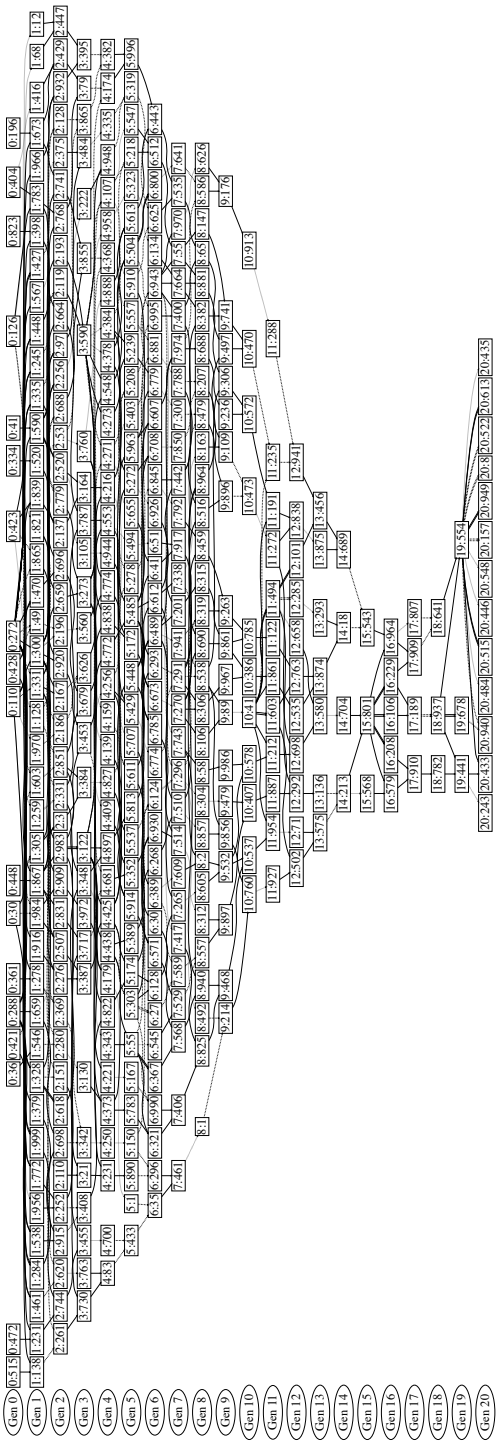

Fig. 1 The full ancestry graph containing all the ancestors of the 13 successful individuals. Individuals are represented with boxes, with the each generation being a row; the top row in the initial random population, and the bottom contains the 13 successful inidividuals. Edges represent parent-child relationships; see the text for descriptions of the meaning of the particular edge decorations.

genetic material to one of the winners. Note, however, that not all these individuals actually contributed to those solutions. There are, for example, cases where one of the parents actually contributed no material in a recombination (alternation) event, and cases where a parent did contribute some genetic material, but that material was later removed or replaced in subsequent mutations or recombinations.

Conversely, while the individuals not represented in this graph are guaranteed to have not contributed to the genetics of the successful individuals, they might have still had some substantial impact on the run's overall dynamics. The presence of those individuals and their error vectors could certainly affect lexicase selection's choice of parents, for example, which could substantially impact the dynamics.

3.2 Genetic ancestry graph

Despite the short length of this run, and the restriction to just displaying ancestors of successful individuals, Figure 1 still contains 394 nodes and 629 edges, making it difficult to analyze in full.

There were 13 successful individuals in this run, most of which had identical simplified programs. To further simplify the graph and the analysis, we picked[5] one of the successful individuals, namely 20:435, which was constructed via a single instruction mutation from individual 19:554. Individual 20:435's genome contained 194 genes, and its program had zero error on both the training and testing cases. The simplified program for 20:435 (which also passes all the tests) contains only 9 instructions:

```
(\space \newline in1 string_replacechar print_string
in1 \space string_removechar string_length)
```

This simplified program is actually quite readable, and has a similar structure to what me might expect from a human solution. The first five instructions (together on the first line) replace all the spaces in the input string with newlines (using the `string_replacechar` instruction) and print the resulting string, thereby solving half the problem. The next four instructions (on the second line) remove all the spaces from a fresh copy of the input string, compute the length and leave that on the `:integer` stack as the "returned" result.

To simplify the graph in Figure 1, we extracted the subgraph containing only those individuals that contributed at least one of these nine *key instructions* to individual 20:435; see Figure 2. Starting from 20:435 we traced backwards through it's ancestors, tracking where the 9 key instructions came from. In doing so we found all of the members in the full ancestry graph that contributed these important instructions. and then extracted the genetic ancestry subgraph containing only these indi-

[5] This choice was somewhat arbitrary, but most of the 13 successful programs simplify down to the same 9 instruction program, so the analysis would have been the same in most cases even if we'd worked back from a different successful individual.

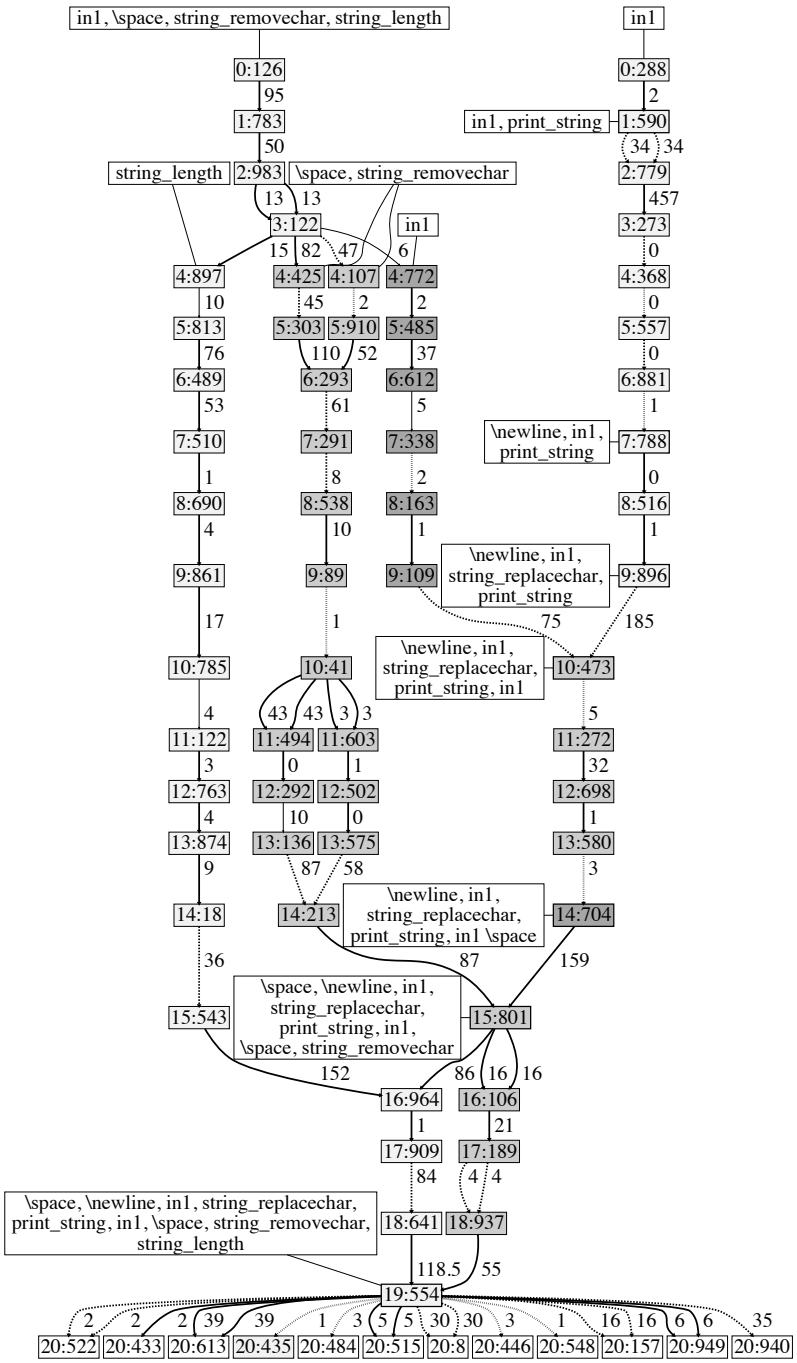

Fig. 2 The genetic ancestry version of the run's full ancestry graph.

viduals. By cutting down on the number of individuals displayed we have a much more readable and focused visualization of this important ancestry information.

The genetic ancestry graph in Figure 2 uses the same basic display of node and edge information as the full ancestry graph. Tthere are, however, several additions that indicate how the 9 key instructions flowed through the ancestry. First, we decorated nodes with boxes showing which of the 9 key instructions were present in that individual. In most cases undecorated nodes contain the same instructions as the most recent labeled ancestor; the exceptions to this are individuals 16:964, 17:909, and 18:641, each of which contribute just the `string_length` instruction inherited from 15:543. Next we added a thicker border to certain nodes, to indicate the introduction or combination of key instructions via either mutation or crossover. Individual 1:590 is highlighted, for example, because the key `print_string` instruction was introduced their via uniform mutation, and individual 10:473 is highlighted because the alternation of 9:109 and 9:896 brought together an `in1` instruction from 9:109 with the four printing instructions from 9:896. Lastly we used a grayscale color gradient to indicate which and how many of the instructions were present in an individual. The earlier of the 9 key instructions are assigned lighter colors in the gradient, and the later instructions are assigned darker gradient colors. So individuals like 7:338 have a fairly "flat" gray because they contributed just a single instruction from near the middle of the program, where 19:554 has a strong gradiant because it contributed all 9 of the instructions.

The other important extension to the graph in Figure 2 is that we labeled each edge with the Damerau-Levenshtein distance (DL-distance) between the genome vectors for each parent-child pair. The genome vectors were generated by concatenating the `:instruction` [6] and `:close` fields from each gene into a single sequence. As an example, the genome of successful individual 20:435 starts

```
{:instruction boolean_and, :close 0}
{:instruction boolean_shove, :close 0}
{:instruction exec_do*count, :close 0}
{:instruction exec_swap, :close 0}
{:instruction integer_empty, :close 0}
...
```

making the associated genome vector

```
boolean_and 0 boolean_shove 0 exec_do*count 0
exec_swap 0 integer_empty 0 ...
```

The Damerau-Levenshtein distance provides a succinct way to see when an individual has received a large amount of genetic material from its parents. It also allows us to easily identify alternation events that have mutation-like behavior, where there is only a small difference between to genome of one of the parents and the child.

[6] Instructions were treated as atomic symbols when computing the Damerau-Levenshtein distances; swapping a `exec_if` with a `print_string` would only add a distance of 1.

4 The (successful) end and how we got there

As discussed earlier, individual 20:435's program simplfies down to just nine instructions:

```
(\space \newline in1 string_replacechar print_string
in1 \space string_removechar string_length)
```

where the first five instructions (the first line) handle the printing part of the Replace Space With Newline problems, and the next four instructions (the second line) handle the the requirement that the program returns the number of non-space characters.

In this section we trace the origin of each of these nine instructions, going back to their introduction either via a mutation or as an element of one of the initial, random programs in the first generation. It's clear that each of these was "necessary" for the construction of this particular solution, so knowing where they all came from and how they came together should give us a valuable sense of the dynamics of this run. It's important to realize, however, that this will never be the whole story. Push instructions and values can play an important role in subtle ways, e.g., as spacers on stacks that when "counting" is implemented with a stack depth command. Removal of instructions can also be important. One key step in this run, for example, is the removal in the construction of 15:801 of an extraneous `print_newline` present in 14:704; the presence of this instruction caused the printed output to always have an error of one, and its removal changed all of the 100 "printing" errors from 1 to 0. All that said, however, we need some way to limit the number of individuals and events to analyze, so here will focus on the how those nine instructions trace through the ancestry.

It's also important to note that we didn't actually collect enough information to always say for *certain* where an instruction came from in a recombination event. There are numerous copies of instructions like `in1` in most of the genomes, for example, and in principle any of them in a parent could be the source of an `in1` in a child. In practice, however, there are constrains of location and order that typically allowed us to identify a single, unique source. There were a few places, however, where judgement calls were made. In future work we're going to explore attaching unique IDs to each gene and track not just parent-child relationships, but also source-destination relationships among genes, as this will give us certainty about the sources of genes, and allow us to automate more of the analysis, all at the expense of larger databases.

Returning to the specific program, it turns out that the evolution of the first five instructions, those handling the printing part of Replace Space With Newline, is largely independent of the evolution of the last four instructions, which handle the return part of the problem. The first five instructions, for example, all appear early in the genome for 20:435, between gene 9 and gene 24, while the last four all appear much later in the genome, between gene 107 and 175. As a consequence we'll trace these two groups one at a time, then discuss the "end game" after those two groups of instructions are brought together in individual 19:554.

4.1 Printing: The first five instructions

The ancestry of the 5 instructions that solve the printing test cases is fairly straightforward, and involves far more mutation than we expected. Unlike the return case instructions described below (Section 4.2), here there is a clear linear path for these instructions. They are introduced over time, and they are never split apart into branches to be recombined later.

Starting at the top-right of Figure 2 with individual 0:288, only one of the key instructions, `in1`, was present in that individual from the initial randomly generated population. The other 4 of the 5 key printing instructions were introduced over time through a series of uniform mutations. The second of these 5 key instructions was introduced in individual 1:590 via a point mutation converting a piece of 0:288's genome into a `print_string` instruction. These two instructions are passed along this branch until they are joined by the next important instruction, `\newline`, introduced in 7:788 again via uniform mutation. After descending another 2 generations these three instructions were joined by `string_replacechar` in 9:896 via yet another uniform mutation. The final of these 5 key instructions, `\space` was added in generation 14 via a uniform mutation of 13:580 into 14:704, thus completing the five key printing instructions. These five instructions were then passed down as a group through 15:801 to the winners.

The impact of these instruction additions can be seen in the individuals' error vectors, as each addition was accompanied by a shift in the printing test cases. Sometimes the effect was minimal, with both small positive and small negative changes in the errors for different test cases, while other times the change led to dramatic improvements. As an example of a dramatic improvement, when `print_string` was introduced into 1:590, for example, the error on nearly all of the 100 printing test improved, with only a few showing an increased error; the total error for 1:590 across all 200 test cases was 492, where it's parents (0:41[7] and 0:288) has total errors of 1,594 and 1,154 respectively. Later, in the creation of 14:704 via mutation from 13:580 all of the printing test scores became 1. This was in general an *vital* step, but did lead to an increased error on a few tests that had passed with no error in 13:580; the total error of 14:704 was 922, where the total error of 13:580 was a slightly worse 1,125.

4.2 Returning: The last four instructions

The last four "returning" instructions were present in the right order in the very first generation, in individual 0:126. The first of these instructions (`in1`) was on gene 75 of the genome, the next two (`\space` and `string_removechar`) were on genes 89 and 94, and then the final instruction (`string_length`) was on line

[7] Individual 0:41 isn't shown in Figure 2 since it didn't contribute any of the 9 key instructions to 1:590 or, ultimately, 20:435.

141 (out of a total of 161 genes in that initial genome). Despite the fact that this individual had "all the right stuff", it's error vector had very few zeros, i.e., it was rarely correct, highlighting the fact that the presence or absence of other instructions can profoundly impact a program's behavior. 0:126 was, however, quite good for a randomly generated program, with all it's errors being under 20, and most being in the single digits. It was selected 45 times to be a parent, making it the seventh most selected parent in the initial generation, and one of only 48 individuals in the initial generation that received any selections. (The most selected parent in that generation, 0:272, was selected 762 times, but ultimately contributed no genes to the winning individual and therefore is not shown in the graph.)

Those four instructions were passed on as a group, with nearly the same relative positions in the genomes, from 0:126 through 1:783 and 2:983 to 3:122 (see Figure 2). 3:122 was the third most selected individual in its generation and had 100 children, several of which went on to carry one or more of these four instructions forward to individual 19:554 when they were finally reunited in the positions that would ultimately lead to success. In particular there were three distinct branches coming from 3:122, each of which will be discussed below.

4.2.1 Branch 4:772 and the carriers of `in1`

Individual 4:772 inherited the copy of the first instruction, `in1`, that would ultimately form part of the solution. This was transmitted down to 9:109 where it was recombined with 9:896 which, as mentioned above in Section 4.1, carried all but one of the first five "printing" instructions.

This recombination led to individual 10:473, which then had 4 of the 5 "printing" instructions, as well as the `in1` that would be the first of the 4 "returning" instructions. These five instructions were then passed down to 14:704, along with the `\space` introduced by mutation in 14:704. 14:704 was one of the parents of 15:801, a recombination which will be described in the discussion of the next branch.

4.2.2 Branches 4:425, 4:107, and multiple blocks

3:122 contained a block of 25 genes that contain the two middle instructions in the "returning" code, `\space` and `string_removechar`. This block was replicated in both 4:425 and 4:107, and then passed, respectively, to 5:303 and 5:910. 5:303 and 5:910 then recombined to create 6:293, which ended up having two complete copies of this block of genes.

These two copies of this block were then copied from 7:291 down through 10:41, to both 13:136 and 13:575. When these recombined to form 14:213 we ended up with *three* near copies of the block. These blocks were no longer identical due to small changes caused by earlier genetic operations, but each block still contained over 20 genes shared, including the two key instructions, `\space` and `string_removechar`, still four instructions apart.

All three of these blocks (and their three copies of these two "final" instructions) were passed on to 15:801 in the recombination of 14:213 and 14:704. 14:704 also bequeathed to 15:801 all of its six "final" instructions, meaning that 15:801 had all but one of those 9 instructions, missing only the final `string_length`.

4.2.3 Branch 4:897 and the carriers of `string_length`

4:897 and its descendants carried the copy of the last instruction, `string_length`, that would ultimately form part of the solution. This was transmitted all the way down to 18:641 without any significant interactions with other "final instructions", as is reflected in the almost entirely linear ancestry from 4:897 to 18:641 in Figure 2.

There was one potentially interesting interaction with the other branches, when 15:543 combined with 15:801 to create 16:964. In this recombination event, however, 15:801 did not contribute any of the 9 key instructions to 16:964. 15:543, on the other hand, transmitted the crucial missing `string_length` gene that had been passed down since our starting random generation, and which went on to be part of the solution in 20:435.

4.3 From 19:554 to the end, and the final adjustments

19:554 was the result of a recombination of 18:641 and 18:937, which finally brought together all nine of the "final" instructions. 18:937 contributed the first 8 instructions, and 18:641 contributed the final `string_length` instruction. Individual 19:554 didn't *quite* solve the problem, however, it did have three "return" test cases with error 1. These three test cases turned out to be the only cases with an input of a single character.

These errors were fairly easy to rectify, however, as evidenced by the fact that 12 of 19:554's 747 offspring (or 1.6%) were indeed successful. Two of these successful children (20:435 and 20:548) were the result of mutating a *single* instruction. The key change brought about by the mutation that led to 20:435 caused an instance of the instruction `string_butlast` to not operate. In 19:554 this `string_butlast` was incorrectly removing the one and only character from the input string when the input consisted of a single character string, so the suppression of that instruction led to a perfect solution.

5 Discussion

The trace in Section 4 provides a sense of where all the key instructions came from, and indicates several of the key moments in the evolutionary process. In this section

Genetic operator	Entire run	Full ancestry graph	Genetic ancestry graph
Alternation + uniform mutation	9,985 (50%)	186 (49%)	39 (54%)
Alternation	4,001 (20%)	67 (18%)	17 (24%)
Uniform mutation	4,026 (20%)	83 (22%)	11 (15%)
Uniform close mutation	1,988 (10%)	40 (11%)	5 (7%)

Table 1 The numbers and proportions of individuals constructed using the different genetic operators. Total percentages might not equal 100% due to rounding.

we'll provide some summary information as well as highlighting both some general patterns and a few important events.

Table 1 enumerates the number and proportions of individuals constructed via the four genetic operators, first across the entire run (so all of the 20,000 individuals generated after the initial random population), then for the ancestry graph in Figure 1 (394 total nodes, 376 constructed after the initial generation), and finally for the genetic ancestry graph in Figure 2 (62 total nodes, 60 constructed after the initial generation). The percentages in the "Entire run" column match the settings in the run configuration, which specified using alternation followed by uniform mutation 50% of the time, alternation alone 20% of the time, uniform mutation 20% of the time, and uniform close mutation the remaining 10% of the time. The other two columns have similar percentages, suggesting that there wasn't a large skew away from those parameter values, and that none of the genetic operators were particularly over- or under-represented in the ancestry graphs.

While there are numerous alternations in the genetic ancestry graph, it's worth noting that many of the DL-distances (the edge labels in Figure 2) are fairly small, even when alternation was involved, as can be seen in Figure 3. Of the 53 alternations in the genetic ancestry graph (ignoring those leading to a successful individual in generation 20), 21 had DL-distances of 10 or less, 6 had DL-distances of just 1, and 5 had DL-distances of 0 (the child was an exact copy of a parent). One might assume that this is partly due to the six self-cross alternations, where the same individual served as both parents, such as individual 16:106 having 15:801 as both of its parents.[8] In fact, however, most of the self-crosses in the genetic ancestry graph had higher than median DL-distances.

These very small DL-distances mean that many of the alternations were effectively acting as mutation-like events. The steps from individual 15:801 to 18:937, for example, are all alternations (possibly followed by mutations), but in fact almost every change in that sequence was due to gene deletions or duplications in those alternation events. There were 3 mutated genes in that sequence of steps, along with 12 deleted genes and the duplication of a block of 7 genes.

Since the genetic ancestry graph (and thus the data in Figure 3) only includes individuals that actually contributed one of the nine key instructions, in many cases the second parent in alternation events isn't included; these DL-distances are in general higher than those listed. This isn't surprising, as a parent with a small DL-distance is very similar to the child, and thus likely to have contributed most of the important

[8] These self-crosses are likely a result of hyperselection events due to lexicase selection [6].

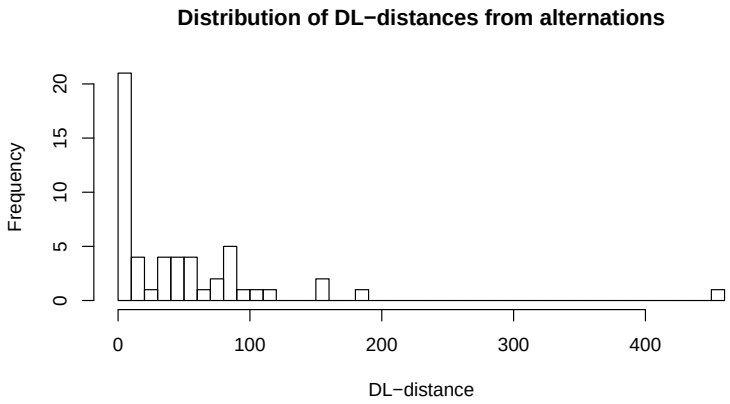

Fig. 3 The distribution of DL-distances for all the alternation events in the genetic ancestry graph (Figure 2) whether or not they were followed by uniform mutation. This does not include the alternations leading to successful individuals in the final generation since those were almost all self-crosses, which skew towards smaller DL-distances.

genetic material. There are, however, a few exceptions to this pattern. Perhaps the most extreme is in the creation of individual 3:273 via alternation between 2:659 and 2:779. Individual 2:659 is not included in the genetic ancestry graph in Figure 2 because it didn't contribute any of the nine key instructions to 3:273, whereas 2:779 contributed two such instructions. However the DL-distance between 3:273 and 2:779 was 457, which was much greater than the distance to 2:659, which was only 50. So despite being much more similar to 2:659 and getting most of its genetic material from that parent, the material that ultimately contributed to the solution all came from the other parent (2:779).

Not all alternation events in Figure 2 could effectively be seen as mutation events, however. The construction of 15:801, for example, was in many ways what we imagine when we think about crossover events, combining significant genetic material and significant functionality from two different parents. It was also a key point in the run, as 15:801 was the first individual to be correct on all of the "printing" test cases, and it was also correct on 26 of the 100 "returning" test cases.

Individual 15:801 was created through the recombination of 14:704 and 14:213, via alternation followed by uniform mutation. Table 2 shows the simplified programs of both parents and the child, aligned to indicate where the various instructions likely came from. The key observation is that 15:801 received most of its initial genetic material from 14:704 (most of genes 1-6), followed by a large section (genes 7-35) taken almost entirely from 14:213's genome. Interestingly, the transition between 14:704 and 15:801 involved a simple but crucial change that fixed all the printing cases. 14:704 had an error of exactly 1 on all the printing cases due to an extra `print_newline` (line 37 in Table 2). In the recombination this gene wasn't passed on to 15:801, which led to a perfect score of 0 on all those test cases. The

performance of 15:801 on the return test cases wasn't quite as strong as that of its other parent, 14:213, but was generally better than 14:704's performance on those test cases. 15:801 went on to receive a large number of selections (595) and being a parent of just over half the next generation (501 individuals).

14:704		**15:801**	**14:213**
	0		(in1
(\space	1	(\space	
\newline	2	\newline	
	3	exec_dup	
in1	4	in1	
string_replacechar	5	string_replacechar	
print_string	6	print_string	print_string
	7	exec_dup	exec_dup
	8		exec_s
	9		(exec_dup
	10		(exec_rot
	11	(string_eq	(string_eq
	12		string_fromboolean)
	13		char_eq
	14		(string_emptystring
	15		boolean_stackdepth
	16		in1
	17		integer_gt)
	18		string_emptystring
	19	\space	\space
	20	string_dup	string_dup
	21	string_removechar	string_removechar
	22	string_rot	
	23		boolean_pop
	24		in1
	25		string_butlast
	26		string_last
	27		string_parse_to_chars
	28		exec_when
	29		string_dup
	30		string_removechar
	31	string_last	string_last
	32	string_parse_to_chars	string_parse_to_chars
	33	string_rot)	string_rot)
	34	in1	in1)
	35	string_stackdepth)	string_stackdepth)
boolean_stackdepth	36		
print_newline)	37		

Table 2 The details of the recombination event (alternation followed by uniform mutation) that created individual 15:801 (center) from parents 14:704 (left) and 14:213 (right) showing the *simplified* programs for those individuals (see Section 2). This shows that individual 15:801 was essentially constructed from a short prefix of 14:704 and a longer suffix of 14:213.

6 Conclusions and future work

Here we traced through the genetic ancestry of a short, successful genetic programming run. While the run was short, it used an "industrial strength" PushGP system on a non-trivial problem that required the manipulation of strings and integers in multiple ways, and a combination of both printing and returning results. We used graph database tools to create ancestry and genetic ancestry graphs, which we were then able to use to visualize and analyze this run. The resulting graphs show the progression of the run and highlight important moments such as key recombination events, gene deletions and duplications, and the introduction of key instructions via mutation. By tracing through the genetic ancestry tree we were able to learn more about how both alternation and mutation played a role in finding a solution.

While we were able to do this for a small run, currently too much of the process is manual for this to scale to larger runs or multiple sets of runs. A key next step in further automating this kind of analysis is automating the process of comparing individuals, especially at the genome level. Tracing each key instruction back through the ancestry graph can be complicated, in part because there are often many different instances of the instruction being traced; individual 19:554, for example, had four instances of `\space`, but only three of those were present in its simplified program, and only two went on to be part of the simplified successful program in 20:435. In this case we were able to deal with these problems by using contextual clues such as order in the genome and surrounding instructions, not unlike how biologists track gene sequences in organisms. To make this process more automatic and exact, however, we'll need to save additional information with the individual genes that allows us to know exactly where they came from.

It would also be valuable to improve our ability to understand and compare program behaviors. We can easily compare genomes and error vectors, and reasonably compare program *texts*, comparing program *behaviors* is much less straightforward. While the simplified program for individual 20:435 is quite short and understandable, the unsimplified program contains 195 instructions, which include a number of complex looping constructs. These are obviously not necessary for the semantics of the program, but they are present in the code that is being tested, and the genes that create those instructions are part of the genome that is being manipulated and inherited. And while those instructions might be removable from 20:435 at the end of the run, it's likely that many of those instructions played some meaningful role in an ancestor that contributed to that ancestor's selection.

Lastly the prevalence of numerous alternation events in the gene ancestry graph that turned out to be just gene deletions or duplications suggests that it might be valuable to include deletion and replication mutations as stand-alone operators, instead of requiring that such events occur via lucky alternations.

Acknowledgements Emma Sax, Laverne Schrock, and Leonid Scott helped with the initial computation and analyses of the differences between the parents and children discussed here. William Tozier provided a host of ideas and feedback all through the process, as did numerous members of the Hampshire College Computational Intelligence lab.

We are very grateful to all the participants in the 2016 Genetic Programming Theory and Practice (GPTP) workshop for their enthusiasm, ideas, and support. In particular we'd like to thank William Tozier, Stephan Winkler, and Wolfgang Banzhaf for their feedback on an earlier draft. Finally, thanks to the GPTP organizers; without their hard work none of those other valuable conversations would have occurred.

This material is based upon work supported by the National Science Foundation under Grants No. 1129139 and 1331283. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Burlacu, B., Affenzeller, M., Kommenda, M.: On the effectiveness of genetic operations in symbolic regression. In: Computer Aided Systems Theory–EUROCAST 2015, pp. 367–374. Springer (2015)
2. Burlacu, B., Affenzeller, M., Kommenda, M., Winkler, S., Kronberger, G.: Visualization of genetic lineages and inheritance information in genetic programming. In: GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion, pp. 1351–1358. ACM, Amsterdam, The Netherlands (2013)
3. Burlacu, B., Affenzeller, M., Winkler, S., Kommenda, M., Kronberger, G.: Methods for genealogy and building block analysis in genetic programming. In: Computational Intelligence and Efficiency in Engineering Systems, *Studies in Computational Intelligence*, vol. 595, pp. 61–74. Springer International Publishing (2015)
4. Burlacu, B., Kommenda, M., Affenzeller, M.: Building blocks identification based on subtree sample counts for genetic programming. In: Computer Aided System Engineering (APCASE), 2015 Asia-Pacific Conference on, pp. 152–157. IEEE (2015)
5. Donatucci, D., Dramdahl, M.K., McPhee, N.F.: Analysis of genetic programming ancestry using a graph database. In: Proceedings of the Midwest Instruction and Computing Symposium (2014). URL http://goo.gl/RZXY2U
6. Helmuth, T., McPhee, N.F., Spector, L.: The impact of hyperselection on lexicase selection. In: GECCO '16: Proceedings of the 2016 Genetic and Evolutionary Computation Conference (2016)
7. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: GECCO '15: Proceedings of the 2015 Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM, Madrid, Spain (2015). DOI doi:10.1145/2739480.2754769. URL http://doi.acm.org/10.1145/2739480.2754769
8. Helmuth, T., Spector, L., McPhee, N.F., Shanabrook, S.: Plush: Linear genomes for pushgp. In: Genetic Programming Theory and Practice XIV, Genetic and Evolutionary Computation. Springer (2016)
9. Kuber, K., Card, S.W., Mehrotra, K.G., Mohan, C.K.: Ancestral networks in evolutionary algorithms. In: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion, pp. 115–116. ACM (2014)
10. McPhee, N.F., Donatucci, D., Helmuth, T.: Using graph databases to explore the dynamics of genetic programming runs. In: Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation. Springer (2015)
11. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report. In: K. McClymont, E. Keedwell (eds.) 1st workshop on Understanding Problems (GECCO-UP), pp. 401–408. ACM, Philadelphia, Pennsylvania, USA (2012). DOI doi:10.1145/2330784.2330846. URL http://hampshire.edu/lspector/pubs/wk09p4-spector.pdf

12. Spector, L., Helmuth, T.: Uniform linear transformation with repair and alternation in genetic programming. In: Genetic Programming Theory and Practice XI, Genetic and Evolutionary Computation, chap. 8, pp. 137–153. Springer, Ann Arbor, USA (2013). DOI doi:10.1007/978-1-4939-0375-7_8
13. Spector, L., Helmuth, T.: Effective simplification of evolved Push programs using a simple, stochastic hill-climber. In: GECCO Companion '14, pp. 147–148. ACM, Vancouver, BC, Canada (2014). DOI doi:10.1145/2598394.2598414. URL http://doi.acm.org/10.1145/2598394.2598414
14. Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, pp. 1689–1696. ACM Press, Washington DC, USA (2005). DOI doi:10.1145/1068009.1068292. URL http://www.cs.bham.ac.uk/ wbl/biblio/gecco2005/docs/p1689.pdf
15. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines **3**(1), 7–40 (2002). DOI doi:10.1023/A:1014538503543. URL http://hampshire.edu/lspector/pubs/push-gpem-final.pdf