Using Graph Databases to Explore Genetic Programming Run Dynamics

Nicholas Freitag McPhee, David Donatucci, and Thomas Helmuth

Abstract Each chapter should be preceded by an abstract (10–15 lines long) that summarizes the content. The abstract will appear *online* at www.SpringerLink.com and be available with unrestricted access. This allows unregistered users to read the abstract as a teaser for the complete chapter. As a general rule the abstracts will not appear in the printed version of your book unless it is the style of your particular book or that of the series to which your book belongs.

Key words: keywords to your chapter, these words should also be indexed

Notes to GPTP readers: *This is still a work in progress. There is just SO much data available, and we're still sifting, picking, and choosing. There are occassional questions and comments in the margin that are in some sense to addressed to ourselves, but feedback from you on any of these would be greatly appreciated. There are two other "big" questions we'd love feedback on:*

- How might we better name/refer to the individuals in the discussion in places like Section 4? Names like "86:261" just don't trip off the tongue and are pretty hard to keep straight in the narrative. Would things like "86:A", "86:B", etc., be better? Introducing "real" names, like "86:Alice" or "Alice:86" be better? Some other terribly clever idea we didn't think of?
- Which of these stories are more or less interesting/useful? Given how much material it there that we *could* discuss, having some guidance on what you personally were excited by would certainly be appreciated.

Many thanks for your time and feedback!

Nicholas Freitag McPhee
Division of Science and Mathematics, University of Minnesota, Morris, MN USA

David Donatucci
Division of Science and Mathematics, University of Minnesota, Morris, MN USA

Thomas Helmuth
Computer Science, University of Massachusetts, Amherst, MA USA

1 Introduction

It would be nice to scrape, say, the GECCO 2014 proceedings and get some numbers to back this up.

It is common practice in most empirical evolutionary computation (EC) research to perform numerous (possibly hundreds) of runs, and then simply report a handful of aggregate statistics at the end that are expected to summarize and represent the (hopefully) complex interactions and dynamics of those many runs. Tables present values such as mean or median best fitnesses at the end of runs, collapsing the complexities of dozens or hundreds of runs into a single number, possibly with a standard deviation or (even better) a confidence interval to give a sense of the distribution. Often more informative are plots, which can, for example, show how these numbers change over time during the runs, possibly giving a sense of the system dynamics. These plots, however, often aggregate runs in a way that obscure important moments that, if explored, might reveal valuable insight into the evolutionary dynamics being reported.

An alternative would be to collect, store, and analyze at least some of the rich panoply of evolutionary and genealogical events that make up the vital low-details details of these runs. Databases provide a natural tool for storing and accessing the data, but traditional relational databases are poorly suited for a variety of queries that are important for the genealogical analysis we need for exploring the evolutionary dynamics of our EC runs. In this chapter, we illustrate the use of graph databases as an alternative storage and analysis tool for evolutionary computation runs. In Donatucci et al (2014) we have demonstrated that graph databases can be an effective tool for analyzing complex genetic programming (GP) dynamics, which led directly to a proposed change to standard sub-tree crossover in tree-based GP, McPhee et al (2015). Here we will use the open source Neo4J graph database tool[1] to explore data from a collection of PushGP runs on several problems drawn from a benchmark collection of introductory programming problems taken from Helmuth and Spector (2015).

Note that this is *not* going to be a presentation of "traditional hypothesis-driven research". It will be based on an *assumption*, namely that something interesting happens in these runs, and that we can learn useful things by exploring them in more detail, but the presentation will be fairly discursive, reflecting our back-and-forth experience of wrestling with the data. Our initial queries start from fairly obvious questions (e.g., "Why did we succeed here?"), but from there we engage in a dialog with data, letting the answers to early questions shape and guide our subsequent exploration. We are presenting not the tidy, sterile summary of our adventures, but the messier (but we think more informative in this context) journal of what Pickering would call our "mangle of practice" (Smith et al (2008); Pickering (1993)).

Because we're going to be focusing on the use of a graph database to explore this data, there will on occasion be avenues of exploration that we won't pursue here because they would properly involve different tools. This exploration raises obvious and important questions about the relationships between, for example, parent and child genomes. These would be best addressed using things like difference-merge

[1] http://neo4j.com/

tools from software engineering, or sequence alignment tools from genomics. Entire papers could be written on the use of those tools in this context, but we'll consider that beyond the scope of this paper. Our queries in the graph database will, however, provide a much more focused collection of comparisons to make using those other tools, allowing us to concentrate on the steps that are likely to have mattered most in the success of a evolutionary run.

More stuff would have to be added here, including a roadmap.

2 Motivation

Consider the job of a paleontologist, who regularly reconstructs not just individuals but also species and entire phylogenetic trees on the basis of handful of teeth and bones, or even just impressions left in prehistoric mud. They rarely have DNA, so any evolutionary relationship is inherently speculative, subject to constant debate and revision. Even with detailed DNA sequences, the construction of phylogenetic trees for existing species is a challenge.

A reference for all of this phylogenetic reconstruction stuff would be useful.

In evolutionary computation, however, we have access to *everything*, at least in principle. Every selection, every mating, every mutation, and every crossover happens in our code and on our watch. Yet we typically throw almost all that data away, reporting just aggregate statistics and summary plots, completely failing to take advantage of our privileged position, a position most paleontologists would presumably eye with considerable envy. Not only does this seem an inherent waste, these aggregations typically obscure critical moments in the dynamics of runs which might speak volumes if explored.

Perhaps include a sample plot and show how it hides things? One of Tom's diversity or cluster plots? A synthetic plot? Maybe that's just not necessary?

While this sort of aggregate reporting is often valuable, allowing for important comparative analysis of, for example, the impact of different genetic operators, it typically fails to provide any sense of the *why*. Yes, Treatment A led to better aggregate performance than Treatment B – but what happened in the runs that led to that result? Any success at the end of a run is ultimately the intricate combination of hundreds or thousands of selections, recombinations and mutations, and if Treatment A is in some sense "better" than Treatment B, it must ultimately be because it affected all those genealogical and genetic events in some significant way, biasing them towards events that made success more likely.

Unfortunately, published research very rarely includes information that might shed light on these *why* events. We rarely see evolved programs, for example, or any kind of post-run analysis of those programs, and there is almost never any data or discussion of the genealogical history that might help us understand how a successful program actually came to be. Sometimes these events and details aren't included for reasons of space and time; evolved programs, for example, are often extremely large and complex, and a meaningful presentation and discussion of such a program could easily take up more space than authors have available given the typical space limitations in published work. Our suspicion, however, is that another reason this sort of *why* analysis often isn't reported is because it isn't done, in no small part because it's hard. As EC researchers we're in the "privileged" position of being able to collect anything and everything that happens in a run, but that's a potentially huge amount of data, and leaves us with two substantial problems: How to *store* the data, and how to *analyze* the data after it's stored. Decreasing data storage costs have done much to mitigate the first problem. If, however, one collects a very rich data set it's still easy to quickly generate terabytes of data, and even if one has a place to put the data, one still needs reasonable tools to analyze the data.

Assuming one has access to the necessary gigabytes or terabytes of storage, databases are the obvious tool for the collection of the data. Most common database structures and tools, however, don't lend themselves to the kinds of analysis that we want and need in evolutionary computation work. Most relational and document-based databases, for example require things like complex and expensive recursive joins to trace significant hereditary lines, making this approach increasingly unfeasible. In exploring the dynamics of an EC run, we're going to want to be able to make connections across dozens or even hundreds of generations, which simply isn't plausible with a relational database. (See Robinson et al (2013) for more on these feasability/efficiency issues.) While we use Neo4j as our graph database in this work, there are numerous other graph databases that would be an effective tool to use as well; see, for example, Wikipedia (2015).

3 A little background on tools and problems

This section provides some background on some of the key subjects of this work:

- The Neo4J graph database and its query language Cypher
- The PushGP system
- Lexicase selection
- The replace-space-with-newline test problem

3.1 Neo4J and Cypher

Graph databases Robinson et al (2013) are a relatively new approach, where data is stored as a collection of nodes and relationships in a graph, with a specialized query language that makes it easy to ask questions about complex relationships. This section further describes Neo4j, its query language Cypher, and the various advantages they hold over relational databases in recording and accessing information that relies heavily on recursion.

Neo4j is a form of data management system based upon a graph. Information is stored by means of vertices and edges, commonly referred to as nodes and relationships, respectively. In our work, nodes represent individuals, and relationships represent the transformations between individuals. As a GP system generates individuals, new nodes and relationships can be added to the database for later analysis.

Query 1 An example Cypher query that finds all children of the node with ID 43.

```
START parent=node(43)
MATCH (parent)-[:PARENTOF]->(child)
RETURN parent, child;
```

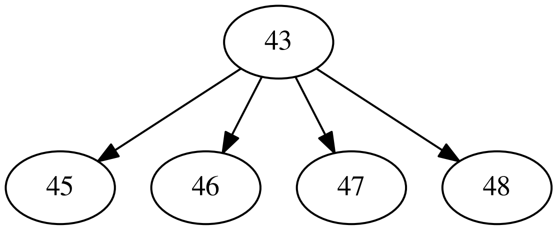

Fig. 1 Graphical representation of the results of a sample Cypher query in Query 1.

Cypher allows this data to be readily extracted from the Neo4j database. There are three fundamental elements to queries in Cypher. The `START` clause specifies a starting location in the database, indicating the node or nodes where the query will begin. The `RETURN` clause specifies which nodes, relationships, or properties should be returned to the user. The `MATCH` clause is the main section of a query, specifying what patterns in the graph the query will discover. To write the `MATCH` clause, nodes and relationships are drawn with ASCII characters. A node is indicated by parenthesis `( )`, directed relationships are indicated with `-->` or `<--` depending on the direction of the relationship, and undirected relationships are indicated with `--`. Brackets `[ ]` between the dashes can be used to specify relationship names prefixed by a colon. In the example in Query 1, the `START` clause indicates that the query should start with node 43, which we call `parent` so we can refer to it elsewhere in the query. The `MATCH` clause finds all nodes that are children of the starting node `parent`, and the `RETURN` clause returns `parent` paired with all its child nodes. This is graphically represented in Figure 1; the actual return is a text table with columns for all the different values in the `RETURN` clause.

3.2 PushGP

PushGP (Spector and Robinson, 2002; Spector et al, 2005) is a stack-based genetic programming system. The details of PushGP aren't crucial for this analysis as we're going to focus on the behavioral properties of individuals in this work, but it useful to know a few things:

- PushGP uses a linear genome which is converted into a linear program via as described below.
- PushGP supports a variety of *typed* stacked, with corresponding typed instructions. The `string-pop` instruction, for example, pops the top of the `string` stack, and the `integer-add` instruction takes the top two items from the `integer` stack, adds them, and pushes the result back onto the `integer` stack.
- There is a `code` stack which can hold blocks of instructions. This is what allows PushGP programs to loop or recurse, as pushing a block of instructions onto the `code` causes those instructions to be executed next.

The PushGP system used here uses *Plush genomes*, which are linear genomes consisting of instructions paired with *closed counts*. The close counts are natural numbers indicating how many open code blocks should be closed after this instruction. Several instructions that are typically used with blocks of code in human programming, such as conditionals and loops, have an implicit "open block" that is translated into an explicit "open block" when the genome is converted to a Push program. The close counts, then, are necessary to allow the PushGP system to evolve the desired "end block"s.

In the runs explored here, there are three genetic operations:

- alternation
- uniform-mutation
- uniform-close-mutation

Alternation is similar to an N-point crossover in genetic algorithms. The two parent genomes are traversed from left to right taking instructions from one or the other for the child, with a small probability at each instruction of switching which parent is being used as the instruction source. When an alternation event happens, there's a small amount of gaussian noise added to the instruction location; how much deviation is possible is controlled by an *alignment deviation* parameter.

Uniform-mutation simply replaces each instruction with a randomly chosen instruction with some small probability. Uniform-close-mutation modifies each close count value with some small probability. The runs discussed here allowed for *pipelining* of genetic operators, so we might have combinations like alternation followed by uniform-mutation.

Mention timing out and errors penalties.

For additional details and the particular parameters used in these runs see Helmuth and Spector (2015).

3.3 Lexicase selection

Pseudocode for the lexicase selection algorithm is outlined in Algorithm 1. In each parent selection event, the lexicase selection algorithm first randomly orders the test

Algorithm 1 Psuedocode for the lexicase selection algorithm. The use of min when computing `best_performance` assumes that the goal is to minimize on each test case, which is true in the work presented here, where the goal for all test cases is to minimize error. This can be easily generalized to other settings.

```
candidates := the entire population
cases := list of all the test cases in a random order
while $|\texttt{candidates}| > 1$ and $|\texttt{cases}| > 0$ do
    current, cases := first(cases), rest(cases)
    best_performance := $\min\{\text{perf}(i, \texttt{current}) \mid i \in \texttt{candidates}\}$
    candidates := $\{i \mid i \in \texttt{candidates} \wedge \text{perf}(i, \texttt{current}) = \texttt{best\_performance}\}$
end while
return random individual from candidates
```

cases. It then eliminates any individuals in the population that do not have the best performance on the first test case. Assuming that more than one individual remains, it then loops, eliminating any individuals from the remaining candidates that do not have the best performance on the second test case. This process continues until only one individual remains and is selected, or until all test cases have been used, in which case it randomly selects one of the remaining individuals.

The central properties of lexicase selection are that (a) it doesn't combine all the errors into a single fitness value and (b) because of the random ordering of test cases, every test case is likely to be most important (first to be considered) at least occasionally. With a population of 1,000 individuals and a problem with 200 test cases, for example, we would expect each test case to be first several times in each generation. The hope, then, is that this ensures some diversity in the population, with different (groups of) individuals being rewarded for their ability to perform well on different test cases.

3.4 *Replace-space-with-newline*

The replace-space-with-newline problem is an introductory programming benchmark problem taken from **?**. In this problem the program is given an input string and required to both (a) print the string with all the spaces replaced by newlines and (b) return an integer that is the number of non-space characters in the input string. There are 100 different training instances for this problem, each of which generates two test cases: One is the Levenshtein distances between the printed output and the target print string, and the other is the absolute difference between whatever value is on the top of the `integer` stack and the expected return value. A penalty value of 1,000 is assigned for test cases that were expecting a return value but found the `integer` stack empty.

For tournament selection runs, all 200 of these error values were added together to form the total error, which was used as the fitness for the individuals. For lexicase

Query 2 Cypher query to find all the ancestors of "winners" in the last 9 generations of a run. The pattern `(w {total_error: 0})` matches nodes with total error 0, i.e., "winners". The pattern `(c)-[*0..7]->(w)` matches any path from some node `c` to a winning node `w` that has between 0 and 7 edges.

```
MATCH (p)-->(c)-[*0..7]->(w {total_error: 0})
RETURN DISTINCT id(p), id(c);
```

selection the errors were kept separate which, as we shall see, frequently allowed individuals to be selected who did well on some test cases, but very poorly on others.

4 Lexicase, meet Replace-space-with-newline

We did one hundred runs of the replace-space-with-newline problem using lexicase selection, and found that 55 of these succeeded in the sense that an individual was discovered that had zero error on all 200 of the training cases. Tournament selection with tournament size 7 only had 13 successes out of 100 runs, and IFS only had 17 successes out of 100 runs, so it seems that lexicase selection provides a significant advantage here.

4.1 Hey, we won! But how did we get there?

Interestingly enough, lexicase performed very well, but this leaves us with the crucial question of *why*? In order to answer this question, we chose one successful run to explore in more detail. However, it's important to note here that we're making no claims that this is a "representative" run (whatever that would even mean); it's an *interesting* run, though, and our hope is that by understanding its dynamics better we can learn useful things about both the problem and the tools we're applying.

An obvious place to start analyzing is at the end when the GP system created individuals that solved the problem. So we used Neo4J to find all the ancestors of any "winning" individual, i.e., individual with a total error of on all 200 test cases. Using Cypher, we can easily ask for this subset of the population going back to generation 79 using the query in Query 2.[2]

Should all the query details go in an appendix that eventually becomes a tech report or some such? I'm not sure if they aren't just a distraction here.
We need the crazy `p->c->w` thing in Query 3 to extract single edges, but I don't know if we want to talk about that.

Figure 2 shows the ancestry of all of the winners from generation 87 (when we first found a winner in this run) back to generation 79. Each node in the graph represents an individual, and each directed edge indicates a parent-child relationship,

[2] We could certainly have gone back farther in time, but the graph would have become impossible to read as the number of nodes would have ballooned from a few dozen to hundreds or thousands. We went back to generation 79 because that was the most recent generation that had more than 10 distinct ancestors of a winning individual.

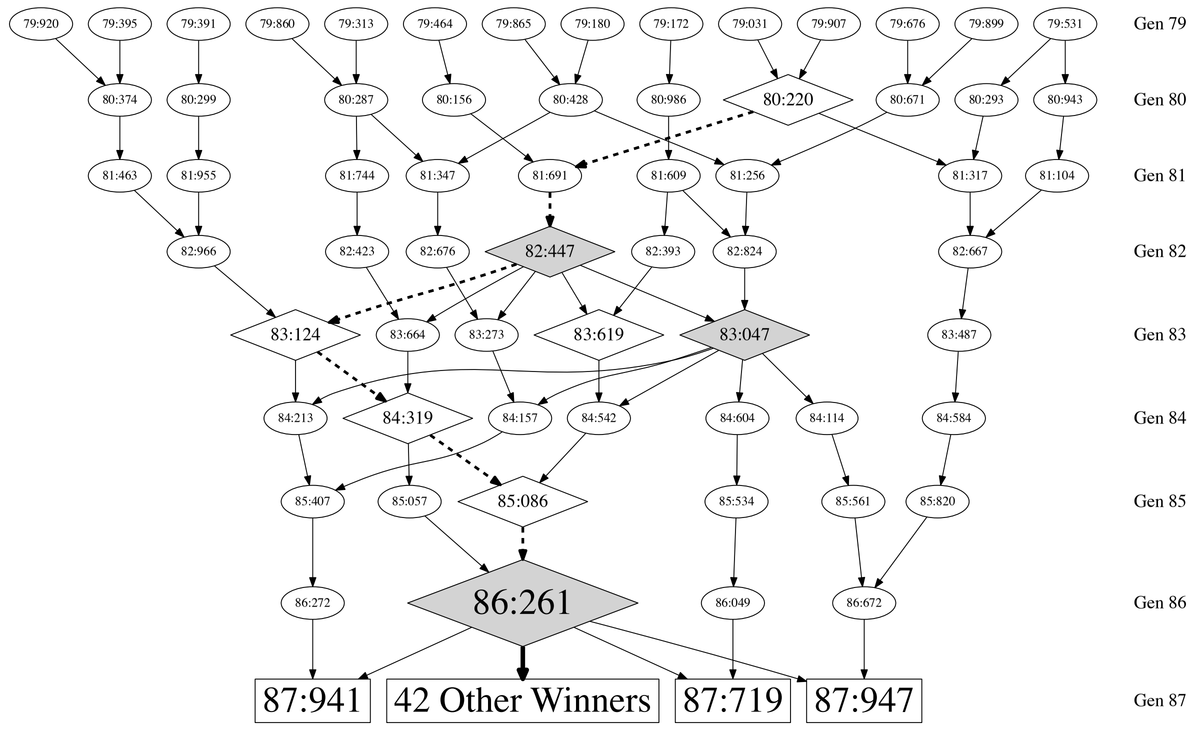

Fig. 2 Ancestry of the 45 “winners” from run 6 of lexicase, replace-space-with-newline. Nodes with diamonds instead of ellipses had an unusually large number of offspring. Shaded nodes had an unusual number of offspring that were ancestors of winners. The dashed lines highlight individuals that had an unusually large number of distinct ancestry paths down to a winner. See the text for more details

with the edge going from the parent to the child. The numbers inside the nodes are Neo4J internal IDs; we'll use these as "names" for the individuals as we tell the stories we uncover.[3] Each ID has two parts: the part before the colon is that individual's generation, and the part after is effectively just a random three digit identifier.

Ignoring for the moment the adornments (shape, shading, etc.), there are several things that we can observe right away:

- There are 45 distinct winners in the final generation, or 4.5% of the population of 1,000 individual. This tells us that constructing a winner from the individuals in generation 86 wasn't entirely trivial, but it also wasn't a huge challenge and happened multiple times.
- The 45 winners only had four distinct ancestors in the preceding generation.
- All 45 winners had a single individual (86:261, marked with a large shaded diamond near the center bottom) as at least one of their parents, and 42 of them had 86:261 as their *only* parent, i.e., they were mutations of 86:261, or were the result of self-crosses of 86:261. To simplify the graph, we've combined those 42 individuals into a single node labeled "42 Other Winners".
- The number of ancestors of winners doesn't grow quickly as we move back in time. We have to go back to generation 80 to find 10 individuals (or 1% of the population) that are ancestors of winners, and in generation 79 there are still only 14 ancestors of winners. In fact, we have to go trace the ancestry all the way to generation 63 to find a time where over 100 individuals (or over 10% of the population) were ancestors of a winning individual.

4.2 Surprising fecundity (especially given that total error)

Looking at Figure 2 we can see that a few individuals have more offspring represented than others. As we've already mentioned, individual 86:261 has 45 successful offspring, and both individuals 82:447 and 83:047 have five offspring in the graph, i.e., five offspring that were ancestors of a winning individual in generation 87. Each of these is marked in Figure 2 with a shaded diamond.

Figure 2, however, only tells us how many offspring an individual had that were themselves either a winner or an ancestor of a winner, as no other nodes are displayed. One might, however, wonder how many total offspring an individual has regardless of whether they were a winner or not. Query 3 identifies the most fecund ancestors of winners in these last nine generations. That reveals several results that were quite surprising to at least some of the authors, the most remarkable being that individual 86:261 was a parent of 934 of the 1,000 individuals in generation 87! Given that lexicase selection was designed in significant part of spread selection events out across the population, this makes it clear that there are times when lexicase does the opposite, and instead puts nearly all its eggs in a single basket. This

[3] We actually assign each individual a UUID so we can combine multiple runs, but the Neo4J IDs are shorter and easier to use in our story telling.

Query 3 Cypher query to find, for each ancestor `p` of a winner, how many distinct offspring `n` that ancestor `p`, regardless of whether `n` is itself an ancestor of a winner. The query then sorts by that count, and returns the 20 highest results.

```
MATCH (p)-->(c)-[*0..7]->(w {total_error: 0})
MATCH (p)-->(n)
RETURN DISTINCT id(p), count(DISTINCT n)
ORDER BY count(distinct n) DESC
LIMIT 20;
```

level of selection focus would simply be impossible using almost any other common type of selection such as tournament selection; in most uses of tournament selection, for example, no individual can be in more than a relative handful of tournaments, and thus can't be a parent terribly often no matter how fit they are.

While no other node in Figure 2 has nearly as many children as 86:261 did, there are several that also had very high reproduction rates, putting them well above what would be possible with something like tournament selection. Individual 82:447, for example, had 443 offspring, including the 5 illustrated in Figure 2. In fact there were eight individuals in Figure 2 that have more than 100 offspring; each of these is indicated with a diamond shape instead of the standard ellipse. This highlights a particularly interesting ancestry chain from 80:220 through 81:691, 82:447, 83:124, 84:319, 85:086 to 86:261, marked with dashed edges in Figure 2. With the exception of 81:691, which "only" had 17 offspring, each of these seven individuals had more than 100 offspring, and thus had a fairly dominate role in shaping the generation that followed them.

If we look at the total error in of the individuals in Figure 2, we again find some significant surprises that tell us quite a lot about lexicase selection. In particular, if we look at the total error for each individual along the dashed path from 80:220 through 82:447 to 86:261, the fitness of the first five individuals in the chain aren't too surprising. One (individual 82:447) has the best total error in that generation and all but 81:691 (the individual with only 17 offspring) are in the top fifth of the population when ranked by total fitness. The fitnesses of the last two (the grandparent and parent of *every* one of the 45 solutions), however, came as quite a shock. In particular, individual 85:086 has a total error of 100,000, placing it *very near the bottom of the population by total error* (rank 971). Individual 86:261, which was the parent of 924 of the 1,000 individuals in the next generation, has a total error of 4,034, placing it below $3/4$ of the population in its generation by that aggregate measure.

So how could individuals with such terrible total fitness end up being selected so often as parents? Exploring the specific test case errors reveals that individual 85:086 is perfect on half of the test cases (all those that involve printing), but gets a penalty error of 1,000 on the other half, presumably because it never actually returns a value. Every one of its ancestors in Table 1, however, has at least a few non-zero errors on the printing test cases, meaning that any lexicase ordering that places a

Table 1 The total error and rank (by total error) in the population in that individual's generation for the sequence of "diamond" individuals from in Figure 2.

Individual	Total error	Rank in population
80:220	321	147
81:691	441	268
82:447	107	1
83:124	157	85
84:319	240	188
85:086	100,000	971
86:261	4,034	765

few key printing test cases before any of the "return" test cases would likely select individual 85:086.

What about individual 86:261, with it's 934 offspring? It is perfect (has error zero) on 194 of the 200 test cases, with it's total error of 4,034 coming from the remaining 6 test cases. On four of these it, like individual 85:086, fails to return a value and gets the penalty of 1,000; it has an error of 17 on the other two. Thus it gets 97% of the test cases correct, but happens to be *heavily* penalized for its behavior on 4 of the 6 it gets wrong. In a system that aggregates the errors, its rank of 765 out of 1,000 would mean that it would probably have no offspring. With lexicase selection, however, it's success on the 194 test cases means that it is selected (from this population) almost every time. In fact only 152 of the 1,000 individuals in the final generation had a parent who *wasn't* 86:261, and only 116 other individuals in generation 86 had an offspring in the next generation. While four of those had 10 or more offspring in the last generation, none of those four actually gave rise to a winner. The three parents of winners other than 86:261 (individuals 86:272, 86:049, and 86:672 in Figure 2) had very few offspring (1, 2, and 2 respectively), suggesting that they may not have contributed much (or anything) to their successful progeny, and the success of their offspring was due more to the good fortune of mating with 86:261 than anything else.

4.3 How exactly did we get here?

Now that we know quite a lot about who gave rise to those 45 winners, what genetic operations brought them about? The largest group was 18 of the 45 which came about through uniform-close-mutation alone, *all* of which were mutations of individual 86:261. This indicates that success can be achieved via a fairly simple modification to 86:261's genome that only modifies where some code blocks end.

The other large group was 17 winners that arose via alternation followed by uniform-mutation. 14 of these were the result of a self-cross of 86:261 and itself, with the other three being crosses between 86:261 and the other three parents of winners (86:272, 86:049, and 86:672). There were also two smaller groups of winners, 6 which were the result of alternation alone (all self-crosses of 86:261), and 4 from uniform-mutation alone applied to 86:261.

An obvious question then is what changed in moving from 86:261 to the final solutions. The genomes and programs involved are fairly complex (over 200 instructions), and as mentioned in the introduction, a full analysis of the genomes and behaviors of the the individuals involved is beyond the scope of this paper. It is possible, however, and our graph database analysis has clearly identified individuals whose genomes and programs deserve additional study.

Based on our graph database work, we can also propose a hypothesis that this additional study could explore. 86:261's total error of 4,034 comes in large part from failing to return a value on four test cases. A distinct possibility is that 86:261 simply times out on those four test cases. The efficacy of uniform-close-mutation suggests that there might be some sequence of instructions that are being executed repeatedly via a loop or recursion, and there are uniform-close-mutations that shorten that block in ways that allow it to complete all the test cases within the time limit without changing the value returned.

5 How is tournament selection different?

In addition to studying lexicase selection, we wanted to collect data from replace-space-with-new-line with tournament selection in order to compare lexicase versus tournament. As previously noted in Section 4, lexicase produced at least one individual with an error of zero in 55 of 100 runs while tournament selection only produced 13 of 100 successful runs. Of these 13 successful tournament runs, we selected one run to study.

An immediate difference that we notice between the lexicase and tournament runs was the number of individuals that solved the problem in the final generation. In lexicase, there were 45 different individuals that solved the problem while there was only a single individual in the tournament run that had an total error of zero across all the test cases.

Figure 3 shows the ancestry of the winning individual from generation 150 back to generation 145. It was not possible to show individuals farther back than generation 145 because of an interesting property of tournaments. Table 2 shows the number of parents n generations away that contributed to the development of the winning individuals. The further away from the winning individual, the number of parents that contributed to the winner in tournament selection increased rapidly. In contrast, lexicase grew much more slowly, presumably due to the fact that parents produce a surprisingly amount of children. This means that the ancestry of tournament selection becomes incredibility more complex and branches out extensively. At 10 generations back, there were approximately three times the number of contributing parents in tournament as lexicase.

Another major difference was the selection pressure of the two selection mechanisms. In lexicase selection, one parent can dominate the selection if it performs

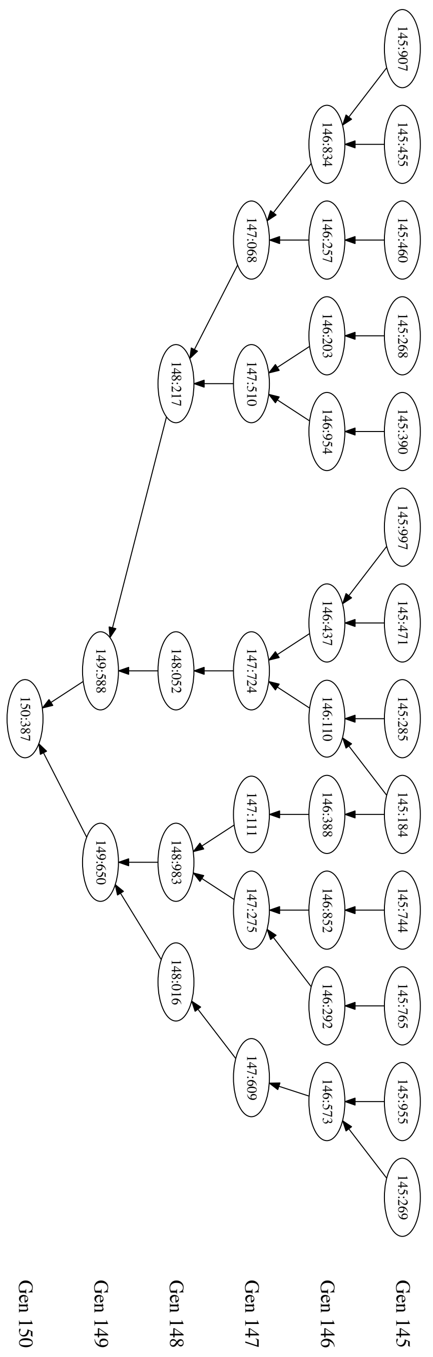

Fig. 3 Ancestry of the sole "winners" from run 74 of tournament selection, replace-space-with-newline.

Table 2 The number of parents contributing to a winning individual *n* generations away.

Gens Away From Winner	# of Tournament Parents	# of Lexicase Parents
18	297	58
17	236	52
16	180	46
15	152	49
14	209	45
13	212	46
12	146	41
11	97	29
10	63	22
9	42	14
8	33	14
7	30	10
6	20	9
5	13	7
4	10	6
3	6	7
2	4	6
1	2	4

well for a majority of the test cases as we saw in section 4.2 where individual 86:261 was a parent for all but 76 individuals in the final generation. However, tournament selection never can have this level of selection pressure. Throughout the entire run, the most a single parent in the tournament selection run ever produced was 24 children (see Table 3), and all of the 20 most prolific parents produced between 17 and 24 offspring. Compare this to lexicase selection, where 19 of the 20 top parents produced over 200 or more children. This is an extreme difference in selection pressure between the two types of selections. This also helps explain the observation that the ancestry tree in the tournament selection run branched out more quickly, leading to a larger number of ancestor of the winning individual.

Beside selection pressure, we noticed another crucial difference between the types of individuals selected for reproduction. With tournament selection, there is a bias towards individuals that have the lowest total error. However, this is not the case in lexicase. As long as an individual performs extremely well for many cases but not all, it is still possible to be selected for reproduction. In Table 1, individual 85:086 has an error of 100,000. This late into the run, it would be very unlikely that tournament would select this individual to reproduce. In fact, we see this in the tournament run, where every ancestor of the winner in the last six generations of the has a total error of either 83 or 132. Additionally, across all individuals chosen as parents in the last 20 generations (regardless of whether they were an ancestor of the winner), there were as few as one and at most five distinct total errors within each generation. This suggests that tournament selection keeps slamming the best individuals together until it miraculously produces an improved child better, whereas lexicase identifies qualities that solve different test cases and utilizes those to con-

awkward wording... table?
Yeah, I think that would be good.

Table 3 The top twenty children producing parents in tournament and lexicase throughout the entire run.

Top 20 Parents	Tournament (Children Produced)	Lexicase (Children Produced)
1	24	934
2	23	657
3	23	594
4	21	590
5	20	433
6	20	326
7	19	297
8	19	294
9	19	285
10	18	283
11	18	279
12	18	271
13	18	234
14	18	220
15	18	212
16	18	205
17	18	203
18	17	202
19	17	200
20	17	189

struct improved offspring. This might contribute to the fact that lexicase had many more successes than tournament selection.

- Tournament winner was produced with alternation and uniform-mutation (mutation could have possibly caused the problem to be solved)
- Many alternations and mutations in the last generations along the winning ancestry produced children that had 0 error difference from the better parent.

6 So what did we learn in all this?

Connect the following back to Section 4.2.

Lexicase selection (Helmuth et al, ????) was designed in significant part with the intent of increasing and maintaining diversity. The key assumption was that it would distribute the selection events across a variety of groups of individuals, as the population separates into sections focusing on different subsets of the test cases. As **?** shows, this is to a significant degree a “true” (or at least reasonable) story, with lexicase generally leading to more diversity than either tournament selection or implicit fitness sharing.

A flip side of that assumption was that individuals probably didn’t have disproportionately large numbers of offspring, as the selections are being spread out across these different groups of individuals. In exploring one lexicase run on the Replace Space With Newlines problem, however, we discovered that while in general this story held true, there were moments in the course of the run where the reality was *wildly* different.

Acknowledgements Thanks to all the cool people!

We need to make this real.

References

Donatucci D, Dramdahl MK, McPhee NF (2014) Analysis of genetic programming ancestry using a graph database. In: Proceedings of the Midwest Instruction and Computing Symposium, URL http://goo.gl/RZXY2U

Helmuth T, Spector L (2015) General program synthesis benchmark suite. In: People M (ed) GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation

Helmuth T, Spector L, Matheson J (????) Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation DOI doi:10.1109/TEVC.2014.2362729, accepted for future publication

McPhee NF, Dramdahl MK, Donatucci D (2015) Impact of crossover bias in genetic programming. In: tons (ed) GECCO proceedings, 2015

Pickering A (1993) The mangle of practice: Agency and emergence in the sociology of science. American Journal of Sociology 99(3):pp. 559–589, URL http://www.jstor.org/stable/2781283

Robinson I, Webber J, Eifrem E (2013) Graph Databases. O'Reilly, URL http://info.neotechnology.com/rs/neotechnology/images/GraphDatabases.pdf

Smith BH, Weintraub ER, Franklin A, Pickering A, Guzik K (2008) The mangle in practice: Science, society, and becoming. Duke University Press

Spector L, Robinson A (2002) Genetic programming and autoconstructive evolution with the push programming language. Genetic Programming and Evolvable Machines 3(1):7–40, DOI doi:10.1023/A:1014538503543, URL http://hampshire.edu/lspector/pubs/push-gpem-final.pdf

Spector L, Klein J, Keijzer M (2005) The push3 execution stack and the evolution of control. In: Beyer HG, O'Reilly UM, Arnold DV, Banzhaf W, Blum C, Bonabeau EW, Cantu-Paz E, Dasgupta D, Deb K, Foster JA, de Jong ED, Lipson H, Llora X, Mancoridis S, Pelikan M, Raidl GR, Soule T, Tyrrell AM, Watson JP, Zitzler E (eds) GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 2, pp 1689–1696, DOI doi:10.1145/1068009.1068292, URL http://www.cs.bham.ac.uk/ wbl/biblio/gecco2005/docs/p1689.pdf

Wikipedia (2015) Graph database — wikipedia, the free encyclopedia. URL http://en.wikipedia.org/w/index.php?title=Graph_database&oldid=653752823, [Online; accessed 28-March-2015]