Using Graph Databases to Explore the Dynamics of Genetic Programming Runs

Nicholas Freitag McPhee, David Donatucci, and Thomas Helmuth

Abstract For both practical reasons and those of habit, most evolutionary computation research is presented in highly summary form. These summaries, however, often obscure or completely mask the profusion of specific selections, crossovers, and mutations that are ultimately responsible for the aggregate behaviors we're interested in. In this chapter we take a different approach and use the Neo4j graph database system to record and analyze the entire genealogical history of a set of genetic programming runs. We then explore a few of these runs in detail, discovering important properties of lexicase selection; these may in turn help us better understand the dynamics of lexicase selection, and the ways in which it differs from tournament selection. More broadly, we illustrate the value of recording and analyzing this level of detail, both as a means of understanding the dynamics of particular runs, and as a way of generating questions and ideas for subsequent, broader study.

Key words: graph database, Neo4j, ancestry, genealogy, lexicase selection, tournament selection

1 Introduction

It is common practice in empirical evolutionary computation (EC) research to perform a substantial number of runs, and then report a handful of aggregate statistics that summarize and (hopefully) represent the complex dynamics of those many runs. Tables present values such as mean or median best fitnesses at the end of runs,

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

collapsing the complexities of dozens or hundreds of runs into a single number, possibly with a standard deviation or a confidence interval to give a sense of the distribution. Plots can often be more informative, showing how these numbers change over time during the runs, possibly giving a sense of the system dynamics and the range of behaviors. These plots, however, are typically still aggregate representations that obscure or completely hide 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 low-level details of these runs. Databases provide a natural tool for storing and accessing large data sets, but traditional relational databases are poorly suited for many of the queries that are important for genealogical analysis. In this chapter, we illustrate the use of graph databases as an alternative storage and analysis tool for evolutionary computation runs. We have previously demonstrated that graph databases can be an effective tool for analyzing complex genetic programming (GP) dynamics (Donatucci et al, 2014), 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 (Helmuth et al, 2015) on several problems drawn from a benchmark collection of introductory programming problems (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 not presenting a tidy, sterile summary of our adventures, but the messier (but we think more informative in this context) journal of what Pickering might call our "mangle of practice" (Smith et al, 2008; Pickering, 1993).

Here we explore the impact of lexicase (Spector, 2012) and tournament selection on the dynamics of runs whose aim is to solve a basic software synthesis problem. In the process we are able to discover surprising and likely important properties of lexicase that suggest areas of additional exploration and indicate reasons for the substantially better performance seen when using lexicase on a variety of software synthesis problems (Helmuth and Spector, 2015).

We're not the first people to recognize the potential value of exploring lineages and ancestry graphs. The HeuristicLab team has been working for several years on a set of tools to analyze at least small genealogical run histories (Burlacu et al, 2013, 2015); hopefully these exciting features will be in an upcoming release. (Burlacu et al, 2013) also has an excellent survey of a variety of work that uses genealogical information in EC work; none of this, however, appears to save and analyze full genealogical histories, but instead tends to use local ancestry information for purposes

---

[1] http://neo4j.com/

such as diversity promotion. Recent work (Kuber et al, 2014) applies network theory to ancestry graphs, looking for things like cliques as a way of better understanding EC dynamics; that work is similar in spirit to this chapter, but differs in the kinds of graphs that are built and the tools used to analyze them.

Because we're going to focus on the use of graph databases, there will on occasion be avenues of exploration that we won't pursue because they would properly involve different tools. This exploration, for example, raises important questions about the relationships between parent and child genomes. These could be addressed using, e.g., difference-merge tools from software engineering, or sequence alignment tools from genomics; see, e.g., (Burlacu et al, 2013) for an excellent example of this kind of analysis. We will, however, consider that beyond the scope of this chapter. A key value of our graph database results will be in providing focus for our use of those other tools, identifying key moments and individuals in the course of a run that deserve additional attention. There are thousands of potential genome comparisons to make in a single run, for example, but our graph databases analysis helps identify some of the critical individuals, crossovers, and mutations in the run, allowing us to concentrate on the steps that are likely to have mattered most.

We'll provide expanded motivation for this work in Section 2, and background on relevant tools and concepts in Section 3. In Section 4 we explore in some detail a successful lexicase selection run, identifying several properties of lexicase selection that distinguish it from other, more traditional selection methods. We then explore a successful tournament selection run in Section 5, comparing those results to the earlier lexicase results. In Section 6 we step back a little and look at the results of expanding some of our queries across hundreds of runs, and then wrap up with some conclusions in Section 7.

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 a 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 non-trivial.

In evolutionary computation, however, we have access to *everything*, at least in principle. We could gather every selection, every mutation, and every crossover as they play out in our systems. 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.

While this sort of aggregate reporting is often valuable, allowing for important comparative analysis, 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 end result is ultimately the intricate combination of thousands or millions of 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 in a way that improved performance.

Unfortunately, published research 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. We suspect, however, 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, but one still needs good tools to process and explore what could quickly run into terabytes of data.

Assuming one has access to the necessary storage, databases are the obvious tool for the collection of the data. Most common database tools, however, don't lend themselves to the kinds of analysis that we need in evolutionary computation work. Most relational and document-based databases, for example, require complex and expensive recursive joins to trace significant hereditary lines. In exploring the dynamics of an EC run, it may be necessary to make connections across dozens or even hundreds of generations, which simply isn't plausible with a relational database (Robinson et al, 2013). While we use Neo4j as our graph database in this work, there are numerous other graph databases that could potentially be effective tools (Wikipedia, 2015a). We make no claims to have exhaustively explored the range of possible database tools for this sort of work.

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; and the replace-space-with-newline test problem.

3.1 Neo4j and Cypher

Graph databases (Robinson et al, 2013) are a relatively new database tool, 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. In our work, nodes typically represent individuals, and `:PARENT_OF` relationships capture the central genealogical connections. We store important data such as the total error as properties of individual nodes, and genetic operators as properties on `:PARENT_OF` edges.

The Neo4j query language, Cypher, allows patterns in this data to be readily extracted. A detailed description of Cypher is beyond the scope of this chapter, but Cypher's central feature is the ability to describe sub-graph patterns. The Neo4j engine can then search for subgraphs matching these patterns. Cypher also provides the ability to filter results based on properties in a manner quite similar to more traditional SQL queries.

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, but it is useful to know a few things:

- PushGP uses a linear genome, which is then converted into a program.
- PushGP supports a variety of *typed* stacks, with corresponding typed instructions. 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 an `exec` 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 `exec` causes those instructions to be executed next.

While traditionally PushGP has evolved Push programs themselves, the most recent version of PushGP instead evolves linear *Plush genomes* consisting of instructions paired with *close counts*. The Plush genomes are manipulated by genetic operators, but are translated into Push programs prior to execution. During translation, any instruction that uses code from the `exec` stack implicitly opens a code block; the close counts are natural numbers indicating how many open code blocks should be closed after a given instruction.

In the runs explored here, there are three genetic operations: Alternation, uniform-mutation, and uniform-close-mutation. Alternation is based on the earlier ULTRA operator (Spector and Helmuth, 2013), and is similar to an N-point crossover in genetic algorithms. The two parent genomes are traversed from left to right, copying instructions from the source parent to the child. There's a small probability at each instruction of an alternation event, which switches which parent is being used as the instruction source. For every alternation event there's a small chance of slightly

Algorithm 1 Pseudocode for lexicase selection, in the context of error minimization. Here the function $\text{perf}(i, p)$ computes the performance of program p on test case i.

```
candidates := the entire population
cases := list of all the test cases in a random order
while |candidates| > 1 and |cases| > 0 do
    current, cases := first(cases), rest(cases)
    best_performance := min{perf(i, current) | i ∈ candidates}
    candidates := {i | i ∈ candidates ∧ perf(i, current) = best_performance}
end while
return random individual from candidates
```

shifting the instruction location in the source parent; 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. For additional details and the particular parameters used in these runs see (Helmuth and Spector, 2015).

3.3 Lexicase selection

Lexicase selection is a recently developed selection method for evolutionary computation in which individuals are selected by filtering the population according to performance on individual fitness cases, considered in random order (Spector, 2012). Lexicase selection, when used as the parent selection method in genetic programming, has been shown to provide significant improvements in terms of problem-solving power (Helmuth et al, 2014; Helmuth and Spector, 2015).

For each parent selection event, lexicase selection (Algorithm 1) randomly orders the test cases and then removes any individuals that do not have the best performance on the first case. If more than one individual remains, then those that do not have the best performance *among those that remain* on the second case are also removed. This continues until only one individual remains and is selected, or until all cases have been used, in which case a random member of the set of remaining individuals is selected. Key properties of lexicase selection are (a) it avoids combining all errors into a single value, (b) because of the random ordering of test cases, every test case will be most important (first to be considered) at least occasionally, and (c) similarly, each pair of test cases, and each triple, etc., will be most important now and then.

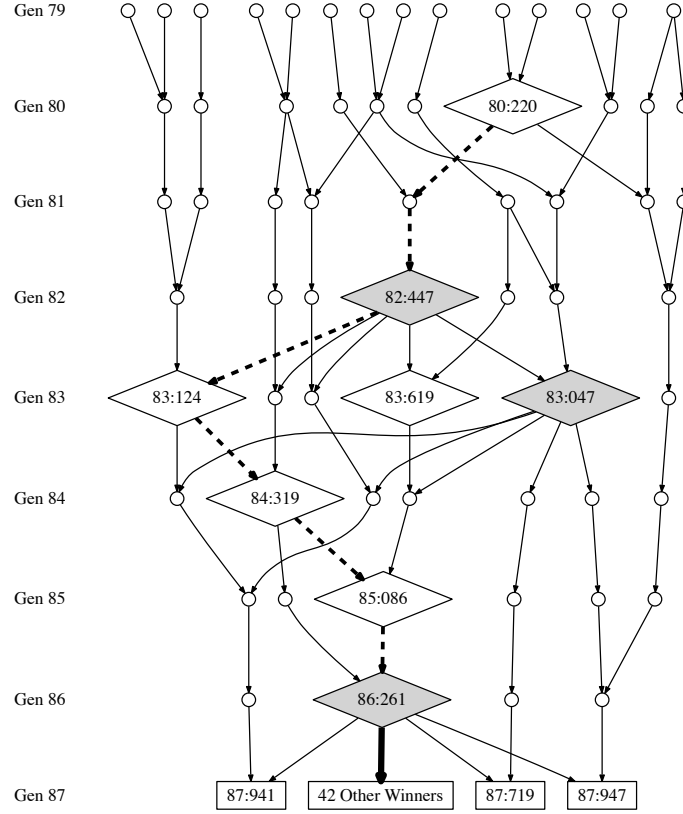### *3.4 Replace-space-with-newline*

The replace-space-with-newline problem is an introductory programming benchmark problem taken from (Helmuth and Spector, 2015). Here 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 error values: (a) the Levenshtein distance between the printed output and the target print string, and (b) 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 selection the errors were kept separate in an error vector of 200 values; this, as we shall see, frequently allowed individuals to be selected who did well on some test cases, but very poorly on others.

3.5 Our data

In this chapter we explore a subset of the data collected for (Helmuth et al, 2015). In particular we have the full genealogical records for 100 runs of replace-space-with-newline using lexicase selection, and 100 runs using tournament selection with tournament size 7. In those runs, 57 of the 100 lexicase runs succeeded, i.e., an individual was discovered that had zero error on all 200 of the training cases. Tournament selection only had 13 successes out of 100 runs, so lexicase selection provides a significant advantage on this problem. Similar results in (Helmuth and Spector, 2015) indicate that lexicase is in fact generally much more successful than tournament selection across a broad range of software synthesis problems.

4 Lexicase, meet Replace-space-with-newline

It's one thing to know that lexicase succeeds 57 out of 100 times on the replace-space-with-newline problem, but that leaves us with the crucial question of *why*? In order to study this question, we chose one successful run to explore in more detail. 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. Looking at this run in some detail certainly unearthed several surprising results, and in Section 6 we'll expand our view by looking at some cumulative results across all 100 lexicase runs.

Fig. 1 Ancestry of the 45 "winners" from a successful run of replace-space-with-newline using lexicase. Diamond-shaped nodes had an unusually large number of offspring (over 100 each). Shaded nodes had at least five offspring that were ancestors of winners.

4.1 Working backwards

A natural place to start our analysis is at the end of the run, when the GP system created one or more individuals that solved the problem. So we used Neo4j to find all the ancestors of any "winning" individual, i.e., an individual with a total error of zero on all 200 test cases. Figure 1 shows the ancestry of all of the winners from generation 87 (when we first found a winner in this run) back to generation 79.[2] Each node in the graph represents an individual, and each directed edge indicates

---

[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.

a parent-child relationship, 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. 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 go all the way to generation 63 before over 100 individuals were ancestors of a winning individual.

4.2 Surprising fecundity (especially given that total error)

Looking at Figure 1 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 1 with a shaded diamond.

Figure 1, 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 wonder how many total offspring an individual has regardless of whether they led to a winner. Using a Cypher query to identify the most fecund ancestors of winners in these last nine generations reveals several things that were quite surprising. The most remarkable of these was 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 to 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 level of selection focus would simply be impossible using almost any other common type of selection such as tournament

---

We went back to generation 79 because that was the most recent generation that had more than 10 distinct ancestors of a winning individual.

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 1.

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

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 1 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 1. In fact there were eight individuals in Figure 1 that have more than 100 offspring; each of these is indicated with a diamond shape. 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 1. 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 that part of the evolutionary process

If we look at the total error in of the individuals in Figure 1, we again find some 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 total errors of the first five individuals in the chain are reasonably low. 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.

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 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 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 has error zero on 194 of the 200 test cases. On 4 of the remaining 6 test cases 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 1) 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 could 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 earlier, a full analysis of the genomes and behaviors of the the individuals involved is beyond the scope of this chapter. Such an analysis is possible, however, and our graph database work has clearly identified individuals whose genomes and programs deserve additional study.

Based on this exploration, we can also propose a hypothesis for further exploration. 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 with our earlier lexicase results. As noted in Section 4, lexicase produced at least one individual with an error of zero in 57 of 100 runs while tournament selection only produced 13 of 100 successful runs. In this section we'll explore one of these 13 successful tournament runs in a little more detail.

An immediate difference between the lexicase and tournament runs is that there was only one solution discovered in the tournament selection run, in contrast to the 45 different individuals that solved the problem in the lexicase run.

Figure 2 shows the ancestry of the winning individual from generation 150 (when the winner was discovered) back to generation 145. It's clear that the branching factor in this ancestry is much higher than with lexicase in Figure 1. Table 2(a) shows the number of ancestors n generations back that contributed to the winning individual, and we can see that the number of ancestors increases much more quickly for tournament than lexicase; at 10 generations back, there were approximately three times the number of contributing parents in tournament as in lexicase. This is likely partially due to the fact that in lexicase some parents produced a surprisingly large number of children. Another possible contribution to this asymmetry is a difference in the role of mutations under lexicase, but we haven't yet explored that in any detail.

Another major difference was the selection pressure exerted by the two selection mechanisms. As we saw earlier, in lexicase selection one parent can dominate the selection if it performs well for a significant number of test cases. However, tournament selection can never impose such a strong selection pressure. Throughout the entire run, the most a single parent in the tournament selection run ever produced was 24 children (see Table 2(b)), and all of the 18 most prolific parents produced between 17 and 24 offspring. Compare this to lexicase selection, where all of the 18 top parents produced over 200 offspring. This extreme difference in selection pressure may also help explain the differences in the branching factor of the two ancestry trees.

We also noticed another crucial difference between the types of individuals selected for reproduction. With tournament selection, the primary bias is towards individuals that have the lowest total error. However, this is not the case in lexicase where, as long as an individual performs extremely well for enough cases, it is still possible to be selected for reproduction, even if it has substantial errors on other test cases. In this tournament run, for example, every ancestor of the winner in the last six generations has a total error of either 83 or 132, which is in marked contrast to the diversity of total errors in the lexicase run (see Table 1). Additionally, across all individuals chosen as parents in the last 20 generations of the tournament run (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 kept mutating and recombining a small set of behaviors until it managed, essentially by accident, to produce an improved child. Lexicase, on the
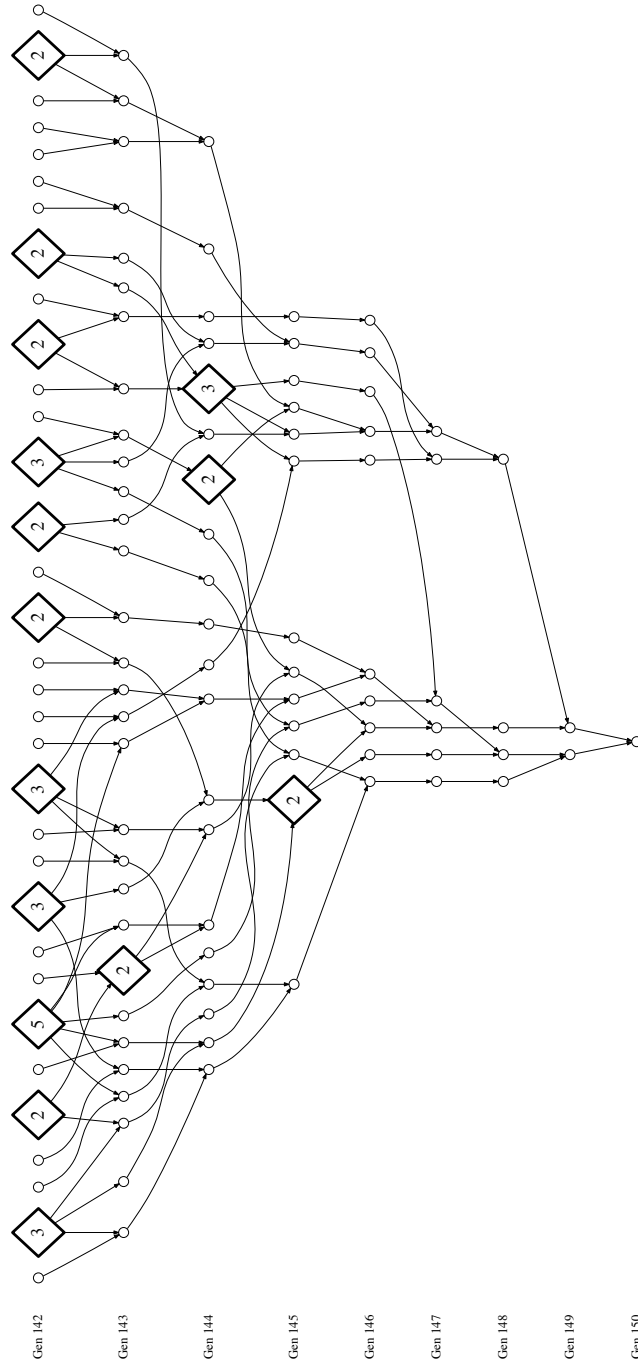
Fig. 2 Ancestry of the sole "winner" from run 74 of tournament selection, replace-space-with-newline. The few nodes with more than one offspring that is an ancestor of the winner are marked with diamonds containing the number of children (in this graph) for that node. Most of those nodes had additional children, not pictured in the figure, that are not ancestors of the winning individual.

Table 2 Two examples of the impact of selection on evolutionary dynamics in the two explored runs. Table (a) lists the number of parents contributing to a winning individual n generations away for both the lexicase and tournament runs explored in this chapter. The top row, for example, indicates that in the lexicase run there were 58 distinct ancestors of a winning individual 18 generations before the discovery of a winner, and in the tournament run there were 297 distinct ancestors 18 generations before the discovery of a winner. Table (b) lists the 18 most fecund individuals across the entirety of each of the lexicase and tournament selection runs.

	(a)			(b)	
	Number of ancestors			Number of children	
n	Lexicase	Tournament	Rank in run	Lexicase	Tournament
18	58	297	1	934	24
17	52	236	2	657	23
16	46	180	3	594	23
15	49	152	4	590	21
14	45	209	5	433	20
13	46	212	6	326	20
12	41	146	7	297	19
11	29	97	8	294	19
10	22	63	9	285	19
9	14	42	10	283	18
8	14	33	11	279	18
7	10	30	12	271	18
6	9	20	13	234	18
5	7	13	14	220	18
4	6	10	15	212	18
3	7	6	16	205	18
2	6	4	17	203	18
1	4	2	18	202	17

other hand, maintained a much more diverse population and appeared to somehow leverage that diversity to continue to discover improvements.

6 A few cumulative results

The bulk of this chapter has focused on exploring two specific successful runs on the replace-space-with-newline problem, one using lexicase selection, and one using tournament selection. To better understand how well this application of graph databases scales, we also created two larger cumulative databases (one for lexicase selection and one for tournament selection), each containing the complete genealogical record for all 100 runs on replace-space-with-newline. Given these cumulative databases, we were then able to do broad queries against those collections of runs. These were typically inspired by observations from the explorations of individual runs, with the broader queries helping us understand to what degree an observation in an individual run was representative or an outlier.

An obvious question, for example, is how unusual is the individual we discovered in Section 4.2 that had 934 offspring? Was that an aberration, or are these kinds of hyper-selected and hyper-fecund individuals a regular occurrence when using lexicase selection? Querying the combined database revealed that there were 71 individuals in the 100 lexicase runs that were selected more than 900 times, where the average number of selections in a given generation was 1,700. So each of these 71 individuals received over half the total selections in its generation, and consequently had numerous offspring; all had over 700 offspring out of the 1,000 created for the next generation. 22 of those 71 individuals had over 900 offspring, with the biggest winners being two individuals that had 990 and 991 offspring, respectively, after being selected over 1,600 times each.

These 71 individuals clearly represent a very small fraction of the over 18 million nodes encapsulated in our 100 lexicase runs. 50 of the 100 runs, however, had at least one individual with over 900 selections, so this kind of hyper-selection is clearly common in the dynamics of these lexicase runs. This sort of hyper-selection has a profound impact on the dynamics of a run, as almost every individual in the subsequent generation is a child of the hyper-selected individual, and due to self-crosses and mutations that individual is often the *only* parent of those children. Thus the genetics of that individual are likely to have an enormous influence on the make-up of the next generation, creating a substantial population bottleneck. So while those 71 individuals only represent a tiny proportion of the cumulative population, they're likely to have a tremendous impact on the run dynamics; thus the ability to identify and examine these individuals is potentially very informative.

One of the other surprises from our earlier exploration is how "unfit" some of those highly selected individuals were when viewed through the lens of total error. Turning now to these cumulative results, we find that 15 of these 71 hyper-selected individuals had total error at or below 10, and so would likely be selected by tournament selection (although never more than a few dozen times). On the other end of the spectrum, however, 7 of these 71 hyper-selected individuals had total error over 3,000 and would have been *extremely* unlikely to ever be chosen using tournament selection. So here again we see a substantial difference between the dynamics of lexicase and tournament selection, especially given the impact these hyper-selected individuals have on their runs.

Finally, looking at all 200 runs makes it clear that lexicase and tournament selection differ considerably in the likelihood of discovering multiple "winning" individuals in the same generation. Over the 100 runs of the replace-space-with-newline with tournament selection, only 13 runs found a solution with zero total error, and only one of those runs had more than one solution in the final generation (there were two). Of the 57 successful lexicase runs, however, 30 (so just over half) had multiple solutions. Many were only a few (6 runs just had 2 solutions), but 6 runs had over 30 solutions, including runs with 69 and 74 solutions. This strongly suggests that when tournament discovered a winning individual, that discovery was fairly random and therefore had a low probability. The prevalence of multiple solutions in the lexicase runs, however, indicates that the discovery of those solutions had a much higher probability. What's less clear is whether that increased probability

was driven by lexicase's hyper-selection in the last generation, or whether lexicase selection throughout the run had led to Push program structures that were easier to combine/mutate into winning individuals.

7 So what did we learn in all this?

In this chapter we've illustrated the potential of graph databases such as Neo4j as tools for exploring and analyzing some of the rich matrix of low-level events that ultimately make up any evolutionary computation run. Here we've used graph databases as a new kind of "lab bench" instrument, allowing us to zoom in and capture the details of potentially crucial moments in our GP runs.

This has exposed surprising and potentially important properties of lexicase selection. These properties will hopefully help us better understand dynamics of runs using lexicase, and why those runs are frequently more successful than runs using tournament selection. A key goal in the original design of lexicase selection (Helmuth et al, 2014) was to increase and maintain diversity in GP runs which, as (Helmuth et al, 2015) shows, does appear to be the case, with lexicase generally leading to more diversity than either tournament selection or implicit fitness sharing. One would, however, typically consider hyper-selection and the consequent population bottlenecks to be the enemy of diversity, so our discovery of a pattern of individuals being selected hundreds, and sometimes even *thousands*, of times was at the very least striking and worthy of further study.

A key risk in this sort of narrow examination is that one might mistake an observation for a pattern. Thus there will always be a need for broader summary statistical analysis. The kind of exploration we've demonstrated here will ultimately have to be used in conjunction with those statistical tools; discoveries in "digs" such as these can raise questions and suggest hypotheses that can then be supported or refuted through the use of more "traditional" studies.

One other concern is how to scale our use of graph databases. Databases for individual runs are quite tractable, and can be populated and explored using basic off-the-shelf computers. Combing multiple runs, however, presents a number of challenges. The database containing the 100 lexicase runs, for example, contained over 18 million nodes and over 25 million edges, with the full Neo4j database weighing in at 18GB. The database combining the 100 tournament runs was even larger because most of the runs went the full 300 generations: over 28 million nodes, over 48 million edges, and a database of over 31GB. That said, these databases runs reasonably on stock desktop hardware, which is how all the results presented here were generated. To scale up further, however, we'd quickly start to need more specialized infrastructure to, for example, combine the lexicase and tournament selection runs into a single database containing the history of those 200 runs, or combining the results of runs on multiple problems. We have data from thousands of runs (Helmuth et al, 2015), but bringing that together in a single database is clearly infeasible using these tools. An alternative to this approach would be to have a host of database

engines on a cluster, each serving a subset of related data, and then providing tools that would allow queries to be run across all these endpoints, aggregating those results into a single response. Version 1.1 of the SPARQL query language (Wikipedia, 2015b), for example, includes support for queries across multiple endpoints and could potentially be used to run queries across large distributed datasets.

References

Burlacu B, Affenzeller M, Kommenda M, Winkler S, Kronberger G (2013) 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, ACM, Amsterdam, The Netherlands, pp 1351–1358

Burlacu B, Affenzeller M, Winkler S, Kommenda M, Kronberger G (2015) Methods for genealogy and building block analysis in genetic programming. In: Computational Intelligence and Efficiency in Engineering Systems, Studies in Computational Intelligence, vol 595, Springer International Publishing, pp 61–74

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: GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation

Helmuth T, Spector L, Matheson J (2014) Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation DOI doi:10.1109/TEVC.2014.2362729, accepted for future publication

Helmuth T, McPhee NF, Spector L (2015) Lexicase selection for program synthesis: a diversity analysis. In: Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation, Springer

Kuber K, Card SW, Mehrotra KG, Mohan CK (2014) Ancestral networks in evolutionary algorithms. In: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion, ACM, pp 115–116

McPhee NF, Dramdahl MK, Donatucci D (2015) Impact of crossover bias in genetic programming. In: GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation

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 (2012) Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report. In: McClymont K, Keedwell E (eds) 1st workshop on Understanding Problems (GECCO-UP), ACM, Philadelphia, Pennsylvania, USA, pp 401–408, DOI doi:10.1145/2330784.2330846, URL http://hampshire.edu/lspector/pubs/wk09p4-spector.pdf

Spector L, Helmuth T (2013) Uniform linear transformation with repair and alternation in genetic programming. In: Riolo R, Moore JH, Kotanchek M (eds) Genetic Programming Theory and Practice XI, Genetic and Evolutionary Computation, Springer, Ann Arbor, USA, chap 8, pp 137–153

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: GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 2, pp 1689–1696

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

Wikipedia (2015b) SPARQL — Wikipedia, The Free Encyclopedia. URL http://en.wikipedia.org/w/index.php?title=SPARQL&oldid=661970711, [Online; accessed 8-June-2015]