Using Graph Databases to Explore Genetic Programming Run Dynamics

Nicholas Freitag McPhee, David Donatucci, and Thomas Helmuth and others(?)

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

1 Introduction

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

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

Nicholas Freitag McPhee
University of Minnesota, Morris

David Donatucci
University of Minnesota, Morris

Thomas Helmuth
University of Massachusetts Amherst

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.

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 this isn'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 this sort of *why* analysis often isn't reported because it isn't even *done*, in no small part because it's hard. As EC researchers we're in the "happy" position of being able to collect anything and everything that happens in a run, but that leaves us with two 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.

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. If, for example, we have a `ParentChild` table, it's easy enough to find Alice's parents, but finding Alice's grandparents, siblings, or cousins requires multiple joins. Moving further out in Alice's tree of relatives requires complex and computationally expensive recursive joins, making this approach increasingly infeasible. In exploring the dynamics of an EC run, for example, we're going to want to be able to make connections across dozens or even hundreds of generations, which simply isn't feasible with a relational database. (See, for example, **?** for more on these feasability/efficiency issues.)

In this chapter we illustrate the use of graph databases as an alternative storage and analysis tool for evolutionary computation runs. In **?** 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 subtree crossover in tree-based GP, **?**. Here we will use the open source Neo4J graph

Fig. 1 Results of Example Query

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 **?**.

2 A little background on problems and tools

2.1 Neo4J and Cypher

Graph databases 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.

Cypherallows 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 query below, the `START` clause indicates that the query should start with node 43, called `parent`, the `MATCH` clause finds all nodes that are children of the starting node, and the `RETURN` clause yields the starting node and all nodes that are children of that node.

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

[1] http://neo4j.com/

Performance is the key advantage in our research of graph databases over relational databases. As the data set grows, recursive queries such as those needed to explore graph relationships become highly inefficient when using relational databases, as numerous joins are needed. In graph databases, however, the portion of the data set that must be searched is limited because the query will only search along an available path connected by relationships, allowing queries to remain efficient

2.2 PushGP and lexicase selection

Say a little about

- Plush genomes
- Push programs
- Alternation
- The two kinds of mutation

Say enough about lexicase so people have some sense of why it might be interesting and different from, e.g., tournament selection.

2.3 Replace-space-with-newline

Say enough about this problem so that people can understand the error vectors.

3 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.

3.1 Hey, we won! But why*?*

It's interesting the lexicase did so well, but that leaves us with the crucial question of *why*? So we chose one successful run to explore in more detail. It's important to note here that we're making no claims that this is a "representative" run (whatever

Query 1 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);
```

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 is at the end when the GP system 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 (Neo4J's query language), we can easily ask for this subset of the population going back to generation 79 using the query in Query 1.[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 2 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, 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] As a happy accident coming from having a population size of 1,000, the first two digits of the Neo4J idea also happens to indicate what generation that individual was from.

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.
- Those 45 winners only had four distinct ancestors in the preceding generation.
- All 45 winners had a single individual (86261, marked with a large shaded diamond near the center bottom) as at least one of their parents, and 42 of them had 86261 as their *only* parent, i.e., they were mutations of 86261, or were the result of self-crosses of 86261. To simplify the graph, we've combined those 42 individuals into a single node labelled "42 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

[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.

[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.

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

population) that are ancestors of winners, and in generation 79 there are still only 14 ancestors of winners. In fact we have to go back 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.

Looking at Figure 2 we can also see that a few individuals have more offspring represented than others. As we've already mentioned, individual 86261 has 45 successful offspring, and both individuals 82447 and 83047 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 home many offspring an individual had that were themselves either a winner or an ancestor of a winner, as no other nodes are displayed. might, however, wonder how many total offspring an individual has regardless of whether they were a winner or not. Query 2 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 86261 was a parent of 934 of the 1,000 individiuals 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 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 86261 did, there are several that had very high reproduction rates, putting them well above what would be possible with something like tournament selection. Individual 82447, 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 82447 through 83124, 84319, 85086 to 86261, each of which had more than 100 offspring. Here the test case results for each of these individuals must be quite "special" in the sense that they are be able to solve an large set of test cases that other individuals simply aren't able to solve.

It would be interesting to explore the details of those error values, but not right now.

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 (mostly) diamond path from 80220 through 82447 to 86261 (displayed in Table 1), the fitness of the first five individuals in the chain aren't too surprising. One (individual 82447) has the best total error in that generation, all the others are in or nearly in the top quarter of the population. 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 85086 has a total error of 100,000, placing it *very near the bottom of the population by total error* (rank 971). Individual 86261, 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.

Individual	Total error	Rank in population
80220	321	147
81691	441	268
82447	107	1
83124	157	85
84319	240	188
85086	100,000	971
86261	4,034	765

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.

Query 2 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;
```

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 85086 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 few key printing test cases before any of the "return" test cases would likely select individual 85086.

Say something about individual 86261.

The dashed lines indicate individuals with lots of different paths to a winner, i.e., they are the ancestor of a winner in several different ways. we still need to write that up.

How did we get the 45 winners?

- 18 uniform-close-mutation alone
- 17 alternation followed by uniform-mutation
- 6 alternation alone
- 4 uniform-mutation alone

Notes

- Individual 81691 is on a critical path from 80220 to 82447, but didn't actually have a ton of children (17 total, only one of which was an ancestor of a winner).
- 82447 has 396 paths to a winner. 83047 only has 69, even though they both have 5 offspring that are ancestors of winners. Maybe that's not a big deal because 82447 is a generation "older" and gets more paths that way? I'm not sure, though

– if there had just been the one path from 82447 to 83047, then their numbers would be the same (e.g., 81691 and 82447).
- There are six distinct paths from 82447 to 86261, more than any other node that isn't an ancestor of 82447.
- Individuals 83124, 83619, and 83047 collectively had 392 offspring of the 1,000 individuals in generation 84.

3.2 Surprising fecundity (especially given that total error)

Lexicase selection (**?**) 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.

4 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations. And please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

5 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

Use the standard `equation` environment to typeset your equations, e.g.

$$a \times b = c\,, \tag{1}$$

however, for multiline equations we recommend to use the `eqnarray` environment.

$$\begin{aligned} a \times b &= c \\ \mathbf{a} \cdot \mathbf{b} &= \mathbf{c} \end{aligned} \tag{2}$$

5.1 Subsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5.

> Please do not use quotation marks when quoting texts! Simply use the `quotation` environment – it will automatically render Springer's preferred layout.

5.1.1 Subsubsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5.1, see also Fig. 3[4]

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

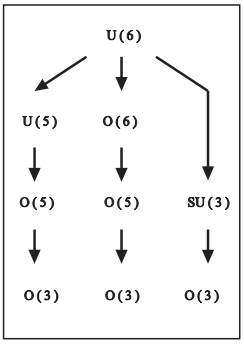

Fig. 3 If the width of the figure is less than 7.8 cm use the `sidecapion` command to flush the caption on the left side of the page. If the figure is positioned at the top of the page, align the sidecaption with the top of the figure – to achieve this you simply need to use the optional argument `[t]` with the `sidecaption` command

[4] Footnotes are easily added with this simple command.

Paragraph Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

For typesetting numbered lists we recommend to use the `enumerate` environment – it will automatically render Springer's preferred layout.

1. Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.
   a. Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.
   b. Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.
2. Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.

Subparagraph Heading

In order to avoid simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5, see also Fig. 4.

For unnumbered list we recommend to use the `itemize` environment – it will automatically render Springer's preferred layout.

- Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development, cf. Table 2.
  – Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.
  – Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.
- Livelihood and survival mobility are oftentimes coutcomes of uneven socioeconomic development.

6 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the

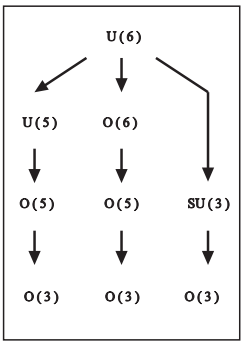

Fig. 4 If the width of the figure is less than 7.8 cm use the `sidecapion` command to flush the caption on the left side of the page. If the figure is positioned at the top of the page, align the sidecaption with the top of the figure – to achieve this you simply need to use the optional argument `[t]` with the `sidecaption` command

Table 2 Please write your table caption here

Classes	Subclass	Length	Action Mechanism
Translation	mRNA[a]	22 (19–25)	Translation repression, mRNA cleavage
Translation	mRNA cleavage	21	mRNA cleavage
Translation	mRNA	21–22	mRNA cleavage
Translation	mRNA	24–26	Histone and DNA Modification

[a] Table foot note (with superscript)

LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

If you want to list definitions or the like we recommend to use the Springer-enhanced `description` environment – it will automatically render Springer's preferred layout.

Type 1 That addresses central themes pertainng to migration, health, and disease. In Sect. 4, Wilson discusses the role of human migration in infectious disease distributions and patterns.

Type 2 That addresses central themes pertainng to migration, health, and disease. In Sect. 5.1, Wilson discusses the role of human migration in infectious disease distributions and patterns.

6.1 Subsection Heading

In order to avoid simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Use the LaTeX automatism for all your cross-references and citations citations as has already been described in Sect. 5.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

If you want to emphasize complete paragraphs of texts we recommend to use the newly defined Springer class option `graybox` and the newly defined environment `svgraybox`. This will produce a 15 percent screened box 'behind' your text.

If you want to emphasize complete paragraphs of texts we recommend to use the newly defined Springer class option and environment `svgraybox`. This will produce a 15 percent screened box 'behind' your text.

6.1.1 Subsubsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

Theorem 0.1. *Theorem text goes here.*

Definition 0.1. Definition text goes here.

Proof. Proof text goes here. □

Paragraph Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Further on please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. 5.

Note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

Theorem 0.2. *Theorem text goes here.*

Definition 0.2. Definition text goes here.

Proof. Proof text goes here. □

Acknowledgements If you want to include acknowledgments of assistance and the like at the end of an individual chapter please use the `acknowledgement` environment – it will automatically render Springer's preferred layout.