Analysis of Genetic Programming Ancestry Using a Graph Database

David Donatucci, M. Kirbie Dramdahl, and Nicholas Freitag McPhee
Division of Science and Mathematics
University of Minnesota, Morris
Morris, MN 56267
donat056@morris.umn.edu
dramd002@morris.umn.edu
mcphee@morris.umn.edu

Abstract

Genetic programming is an artificial intelligence technique that uses concepts from biological evolution such as fitness, mutation, and crossover to manipulate a population of functions, typically represented as trees. Analyzing the complex dynamics of such a system can be challenging. Researchers rarely save or analyze most of the intermediate data from a run, and instead focus on statistical summaries of generations. However, information is lost in this process, precluding potentially important analysis of key events during the run.

Our objective here is to use graph databases to store and analyze the ancestry of individuals. Graph databases are relatively new, and provide features such as queries to obtain data that would be difficult with relational databases. In relational databases, as data sets increase in size, recursive queries become extremely inefficient. By comparison, with a graph database such as Neo4j, the execution time for recursive queries remains relatively constant as the size of data sets grows. Since genetic programming involves a significant number of trees and a multitude of generations, graph databases allow for efficient querying of ancestry that would not be possible with more traditional database systems such as SQL.

Our hope is that by recording and analyzing tree ancestry, we will be able to obtain valuable insight into the evolutionary process of genetic programming. Perhaps most significantly, we hope to discover where trees show significant improvement in fitness and how those improvements are obtained. This will allow for a better understanding of how genetic programming works and provide details for future improvements in the evolutionary computation field.

1 Introduction

Genetic programming (GP) is an artificial intelligence technique that uses concepts from biological evolution such as fitness, mutation, and crossover to discover solutions to user defined problems. Genetic programming manipulates populations of individuals which are evaluated based upon their fitness. Individuals providing the best solution to the target problem will have stronger fitnesses, and will typically produce more offspring than those with weaker fitnesses. Over many generations, descendants generally have stronger fitnesses than their ancestors from previous generations.

Genetic programming systems are great tools for discovering solutions to complex problems, especially those involving many variables that would be difficult to solve by other means. Genetic programming has applications in chemistry, electronic circuit design, economics, and many other areas.

While evolutionary algorithms have clearly been successful in a variety of settings, it is often challenging to determine why this is true. In order to reach a greater understanding of the processes involved in genetic programming, it is necessary to examine the internal interactions of individuals within a run, rather than simply reporting statistical summaries of the final results. Even simple GP runs can generate very large data sets, however, especially if one records all the individuals and relationships from every generation.

Databases are a natural tool for handling such large data sets, but answering important questions and queries for GP work can be onerous when using relational databases. A natural question when analyzing an GP run, for example, would be to find all the ancestors of the "winning" individual. If we used a relational database, we might store the IDs of the parents along with each individual. A single query would then return the parents of an individual, but then additional queries (one per parent) would be needed to get the set of grandparents, and additional queries (one per grandparent) would be needed to get the set of great grandparents, etc. Assuming two parents per individual, the number of queries will then grow as $O(2^n)$ where n is the number of generations we wish to examine, making this approach totally unfeasible for a host of interesting and important questions.

New graph database technologies, however, have the potential to allow us to easily perform these sorts of queries and analyze important dynamic properties of GP runs. In graph databases one stores nodes and relationships, such as individuals and their relationships to their parents, and the query language makes it easy to search for paths through the graph having specified properties. This makes it fairly trivial to ask important ancestry questions about a run; the query `MATCH (a)-[:PARENTOF*]->(d)`, for example, will find all the ancestors a of some individual d. (The details of graph databases and the Cypher query language will be described in more detail in Section 3.)

This paper demonstrates the usefulness of graph databases in recording and analyzing data produced by GP systems. A description of genetic programming is provided in Section 2, and Section 3 discusses graph databases. Section 4 provides details on how we set up our experimental runs. The results of our work are presented in Section 5, and ideas for future implementation and applications of this work are presented in Section 6.

2 Genetic Programming

Genetic programming [3] is based around the interactions of individuals. Individuals are similar to organisms in biological evolution. As in biological evolution, a group of individuals makes up a population. In the process of biological evolution and natural selection, organisms within a population compete in order to survive and reproduce. Those individuals best adapted to their environment have the best chance of fulfilling these objectives. In genetic programming, individuals also compete, but here those individuals that provide better solutions to the user-defined target problem have the best odds. The goal of genetic programming is to produce individuals that provide quality solutions.

GP is commonly applied to symbolic regression problems, where the goal is to evolve a function that passes through a collection of test points, either coming from empirical data or a synthetic test problem. The fitness is then the difference between the target function and the function encoded by the individual. The lower the fitness, the better the solution fits the target problem, and an individual with a solution that perfectly fits the problem would have a fitness of zero. Therefore, at the conclusion of a genetic programming run, it is desirable to have one or more individuals with fitness at or near zero.

At the start of a run, the population is filled with randomly generated individuals. The individuals within this population then compete in order to pass their code on to the next generation, similar to biological evolution. In this work we used tournament selection, where a specified number of individuals are randomly chosen from the population, and those with the best fitness are selected to produce the next generation. These selected individuals can propagate their genetic material to the next generation by one of three transformation methods. The first and most common method is crossover, comparable to sexual reproduction, where two individuals are selected from the current generation, and elements from each selected individual are combined to form a new individual in the next generation. The second method is mutation, in which an individual is selected and randomly altered, much like biological mutation. The third and final method is reproduction, where an individual is copied to the next generation, akin to asexual reproduction. There is also an alternative form of reproduction known as elitism, where the best few individuals are copied to the next generation by merit of their fitness alone. Crossover, mutation, and reproduction are utilized many times, across multiple generations, until an ideal or approximate solution is found or until some sort of resource limit is reached.

3 Graph Databases

Graph databases [4] 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. We used the Neo4j graph database system to collect data generated by GP runs. 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.

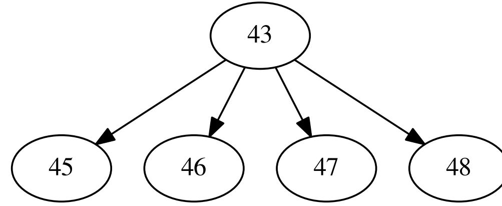

Figure 1: Results of Example Query

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 [4]. In our work, nodes represent individuals, and relationships represent the transformations between individuals. As our GP system generates individuals, new nodes and relationships are added to the database for later analysis.

Cypher [4] 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 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;
```

This query produces the results in Figure 1.

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 [4].

4 Experimental Setup

This section explains the details of the configurations used for this research. Subsection 4.1 covers setup of the genetic programming algorithm, and Subsection 4.2 discusses setup of the graph database Neo4j.

4.1 Genetic Programming Setup

In our system, individuals contain two items: a function called the tree, and the tree's fitness. Trees are represented in prefix notation by arrays containing variables, constants, and operators. Prefix notation places the operator before its arguments. For example, the function $x + (x * 4)$ would be represented by the following array: $[+, x, *, x, 4]$. The tree's fitness is the sum over all the test cases of the absolute error between the target function t and the function f represented by the tree:

$$\text{fitness} = \sum_i |f(x_i) - t(x_i)|$$

After the tree's fitness is computed, we add 1% of the length of the array to the fitness as a means of penalizing particularly large trees. This combats the tendency for trees to become excessively large, and is commonly referred to in genetic programming as bloat control. This implementation of bloat control is relatively weak in the beginning of a run where trees usually have larger fitnesses (therefore not penalizing them unreasonably), but has a significant impact later in the run, where trees should have smaller fitnesses.

In all of the runs, the configurations remained consistent, with the exception of population size, which was either 1,000, or 10,000. The target function was $\sin(x)$, where the value of the variable x ranged from 0 to 6.2, increasing by steps of 0.1. The constants allowed were doubles that ranged between -5 and 5, and x was the sole variable. The function set consisted of the binary operations: addition, subtraction, multiplication, and protected division. In our implementation of protected division, if the denominator equals zero, then regardless of the numerator, the output will be one. The reason we chose the output one for protected division is so there would not be a discontinuity in the function x/x when $x = 0$, thus allowing individuals to use the expression x/x to obtain the value one.

To create the initial population, we used the PTC2 algorithm [1]. This creates trees by randomly adding operators to an array (leaving blank slots where appropriate for arguments) until a specified length is reached. The blank slots are then filled by leaves (variables and constants). In our system, leaves consist of 63% variables and 37% constants, following the proportions used in TinyGP [3].

We used tournament selection to select those individuals which will produce the next generation. In our tournament, two individuals are chosen randomly from the entire generation. The individual with the best fitness of the two is then selected to propagate its code in some capacity to the next generation.

In our system, the top 1% of the current generation is directly copied to the next generation via elitism. The remaining 99% are created via three different means: crossover, mutation, and reproduction. Crossover makes up 90% of all transformations, mutation 1%, and reproduction accounts for the remaining 9%. Reproduction is relatively straightforward, where the individual which wins the tournament is simply copied to the next generation. Mutation and crossover are more complex processes, and will be covered in the following paragraphs.

Mutation begins in a similar manner to reproduction. Two individuals are chosen from the population to enter the tournament, and the winner is selected for mutation. However, rather than simply copying this individual to the next generation, a random position in the tree is selected. The subtree rooted at that position is then removed and replaced by a new subtree generated by PTC2 that is at most half the size of the original tree. This limitation has been put in place to help control bloat.

Crossover differs from the previous transformations in that it makes two calls to tournament selection in order to select two parent individuals to produce a single child individual in the next generation. Within this parent, similar to mutation, a random position is chosen. The subtree rooted at that position is removed and replaced by a subtree randomly selected from the individual that won the second tournament. The first parent, which contributes the root node to the child, is called the *root parent*; the second parent which contributes the subtree is called the *non-root parent*.

4.2 Neo4j Setup

In Neo4j, we set nodes to be individuals and defined their ancestry as relationships. Inside each node, we inserted several attributes belonging to an individual. In addition to the tree and fitness, all nodes also include the penalized fitness, the generation number, the transformation type that generated this individual, the run id (used to differentiate between different runs), and a unique id (used for identifying the specific node). For individuals produced by either crossover or mutation, the “cut point” (the position at which the root parent was altered by a transformation) is also included as an attribute.

Each individual has a relation to its parent (or parents in the case of crossover). To distinguish between each type of transformation, different types of relationships are used. These relationships are demonstrated in Table 1.

5 Results

To obtain the results presented here, we completed three runs using population size 1,000, and one using population size 10,000. The average size of the database for the 1,000 individual runs was 380Mb, and the size of the database for the 10,000 individual run was 3.7Gb.

Relationship Types	
Reproduction	PARENTOF
Elitism	ELITISM
Mutation	MUTANTOF
Crossover Root	ROOT_XOOF
Crossover Non-Root	NONROOT_XOOF

Table 1: On the left are the various transformation types and on the right are the relationship types assigned to each in the Neo4j database. Notice that crossovers have two types of relationships to distinguish between the root parent and the non-root parent.

From these runs, we were able to perform a variety of interesting queries. Most queries completed in a manner of seconds; some of the more complex queries took up to 15 minutes. Computing, for example, the root ancestry path (explained below; see Figure 2) for an individual in the final generation took roughly 1 second for both the 1,000 individual results and the 10,000 results, whereas that sort of recursive query would almost certainly be much slower on the larger database using a relational system such as SQL.

Before describing several questions and queries that we performed using graph databases, we need to define the concept of *the root ancestry line*. The root line of a specified individual n is the path from n to some individual in the initial population, following only root parent crossover transformations and all single parent transformations (elitism, reproduction, and mutation). The non-root line follows the non-root crossover transformation as well as the single parent transformations. Figure 2 illustrates these ideas.

The following, then, are some of the questions we asked of the database:

- *How many individuals in the initial generation have any root parent descendants in the final generation?* (Section 5.1)
- *How often do mutations improve fitness? Also, how often do crossovers improve fitness, where the root parent is more fit than the non-root parent, and vice versa?* (Section 5.2)
- *What does the fitness of the "winning" root parent ancestry line look like over time?* (Section 5.3)
- *Do a group of individuals have a common root parent ancestor and what is the latest generation where such an ancestor occurs?* (Section 5.4)

5.1 Number of Initial Individuals With Final Generation Descendants

To answer the first question, how many individuals from the initial generation have descendants in the final generation, we used the query in Query 1. The `MATCH` statement describes that `startNode` and `endNode` are individuals in generation one and one hundred respectively. The path between them must consist of only elitism, reproduction, mutation, or root

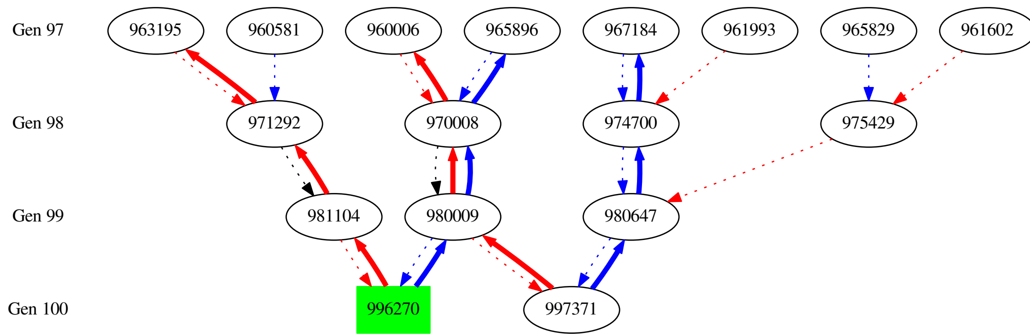

Figure 2: Part of the root and non-root ancestry paths of two individuals in the final generation of a run. Dotted arrows represent transformations: blue for root crossover, red for non-root crossover, black for reproduction. Solid arrows represent the root and non-root ancestry paths, with blue for the root ancestry path and red for non-root ancestry path. Solid blue paths traverse dotted blue and black edges, and solid red paths traverse dotted red and black edges. The individual marked by the green box is the most fit individual in the final generation of this run. Note that the reproduction edge from 970008 to 980009 is part of both the root ancestry path for node 996270 and the non-root ancestry path for node 997371.

Query 1 Cypher query to generate the set of individuals in the initial generation that have root descendants in the final generation.

```
MATCH (startNode:Individual {generation: 1})
   -[:ELITISM|PARENTOF|MUTANTOF|ROOT_XOOF*99]->
   (endNode:Individual {generation:100})
RETURN DISTINCT id(startNode), startNode.penalizedFitness;
```

parent crossover transformations and must have length 99. The `RETURN` statement returns the ID and penalized fitness of every individual in the initial generation that fit the criteria. An example response from this query is presented in Table 2.

This data demonstrates that, at least in this specific instance, all 10,000 individuals in the final generation can be traced back along their root parent line to only two individuals in the initial generation. This, in support of data gathered by McPhee and Hopper [2], indicates that the percentage of initial individuals with direct descendants in the final generation is extremely small. Furthermore, while neither of these individuals had the best initial fitness (24.18), both do appear in the top 5% of first generation individuals. Whether this high fitness rate is consistent across multiple runs is unclear and is worth further exploration.

Individual ID	Penalized Fitness
2595	38.98
3325	40.36

Table 2: List of individuals in the initial generation of the 10K run which produced root descendants in the final generation.

Query 2 Cypher queries to compute the total number of crossover events, and the number where the child's fitness is better than the root parent's fitness, which is in turn better than the non-root parent's fitness.

```
// Count total number of crossovers.

MATCH (rootParent)-[:ROOT_XOOF]->(child)
   <-[:NONROOT_XOOF]-(nonRootParent)
RETURN COUNT(DISTINCT child);

// Count total number of crossovers where the child is more
// fit than the root parent, which is more fit than the
// non-root parent.

MATCH (rootParent)-[:ROOT_XOOF]->(child)
   <-[:NONROOT_XOOF]-(nonRootParent)
WHERE child.penalizedFitness < rootParent.penalizedFitness
   AND rootParent.penalizedFitness
                          < nonRootParent.penalizedFitness
RETURN COUNT(DISTINCT child);
```

5.2 Effectiveness of Mutation and Crossover

To determine the effectiveness of mutation and crossover in producing more fit offspring, we wrote the queries in Query 2. These two queries specifically identified the total number of crossovers and the number of root crossovers where the child was more fit than either parent and the root parent was more fit than the non-root parent. Note that the `MATCH` clause allows relationships in both directions, so the pattern `(r)-->(c)<--(n)` matches cases where there is a relationship from node `r` to node `c`, and a relationship from node `n` to node `c` as well. In the query, we further qualify these two edges, requiring one to be a root parent relationship, and the other a non-root parent relationship. The `WHERE` clause allows us to filter out undesired matches, in this case limiting us to instances where the child's fitness is better than the root parent's, and the root parent's is better than the non-root parent's.

Given the counts from these two queries, we can compute the proportion of crossovers where the child was more fit than the root parent, which was more fit than the non-root parent. The queries for the other two cases, where mutation led to a more fit offspring, and where non-root crossover led to a child which was more fit than either parent and non-root

Query 3 Cypher query to compute the fitnesses along the root ancestry line from the best individual in the final population.

```
START winner=node(996270)
MATCH (winner)
      <-[:ELITISM|PARENTOF|MUTANTOF|ROOT_XOOF*0..]-(parent)
RETURN parent.generation, parent.penalizedFitness,
                                     parent.fitness, "Root";
```

parent was more fit than the root parent, had similar structures. The results of these query may be seen in Figure 3.

The data in Figure 3 is from a single run with 10,000 individuals; we also applied these same queries on three 1,000 individual runs to obtain the very similar graph in Figure 4. In the first generations, mutation produced children that were better than their parents over 30% of the time. As time progressed, the success of mutation decreased dramatically, dropping as low as 2%. On the other hand, the crossover percentages stayed relatively constant with the exception of the first ten generations of non-root crossover. Notice that the crossover variances were very small, meaning that the performance of crossover is stable over time.

5.3 Winning Root Ancestry Line Fitness

We used the query in Query 3 to find the fitnesses along the root ancestry line from the best individual in the final population. A similar query generated fitness information along the non-root parent line; these are graphed in Figure 5. As can be seen, while root parent fitness steadily decreases over time, no such pattern exists for non-root fitness. This implies that the root lineage is far more important in determining the overall success of a specific individual than the non-root lineage.

5.4 Most Recent Common Ancestor

For our fourth and final question, finding a common ancestor of entire last generation, we executed the query in Query 4. An example of a common ancestry graph can be seen in Figure 6. In finding a common ancestor, we can assume that this ancestor carries certain positive traits. This information may be relevant in determining if traits other than fitness give an individual an increased chance of survival. In the 10,000 individual run, two distinct clades were produced. One clade flourished, accounting for 99.76% of the final population (including the individuals in Figure 6), while the second clade consisted of only 24 individuals in the final generation. These two clades are descended from the individuals presented in Table 2. Descendants of the more fit individual profoundly predominated, and given more time, the second clade most likely would have gone extinct.

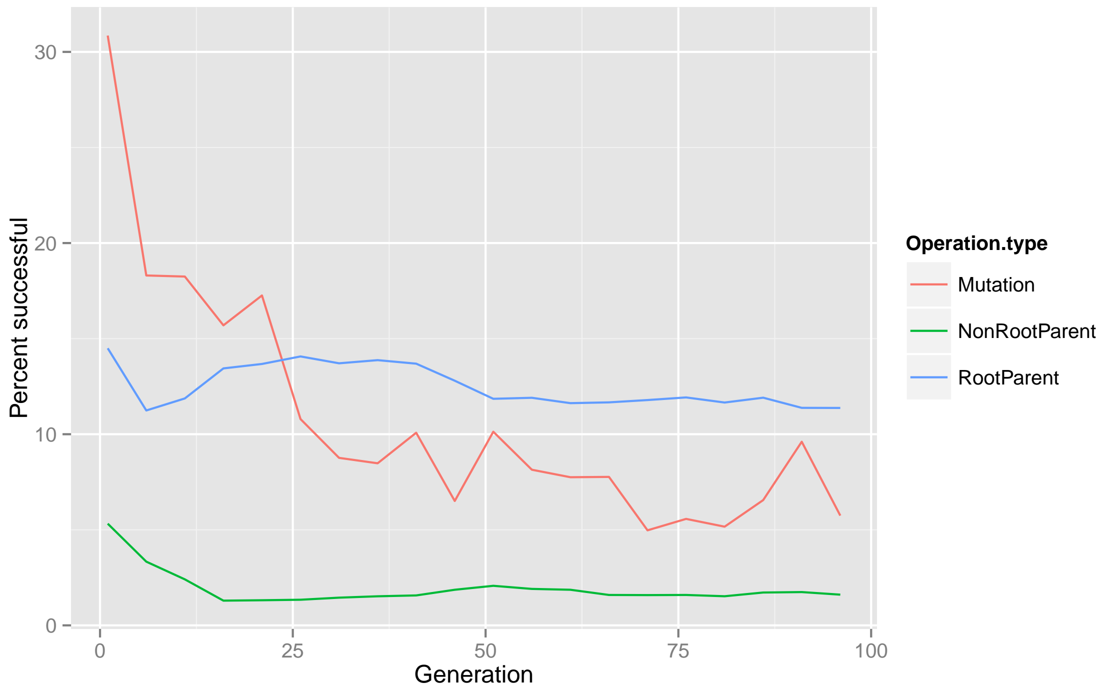

Figure 3: Percentage of cases where the child is fitter than the parent in 10K run.

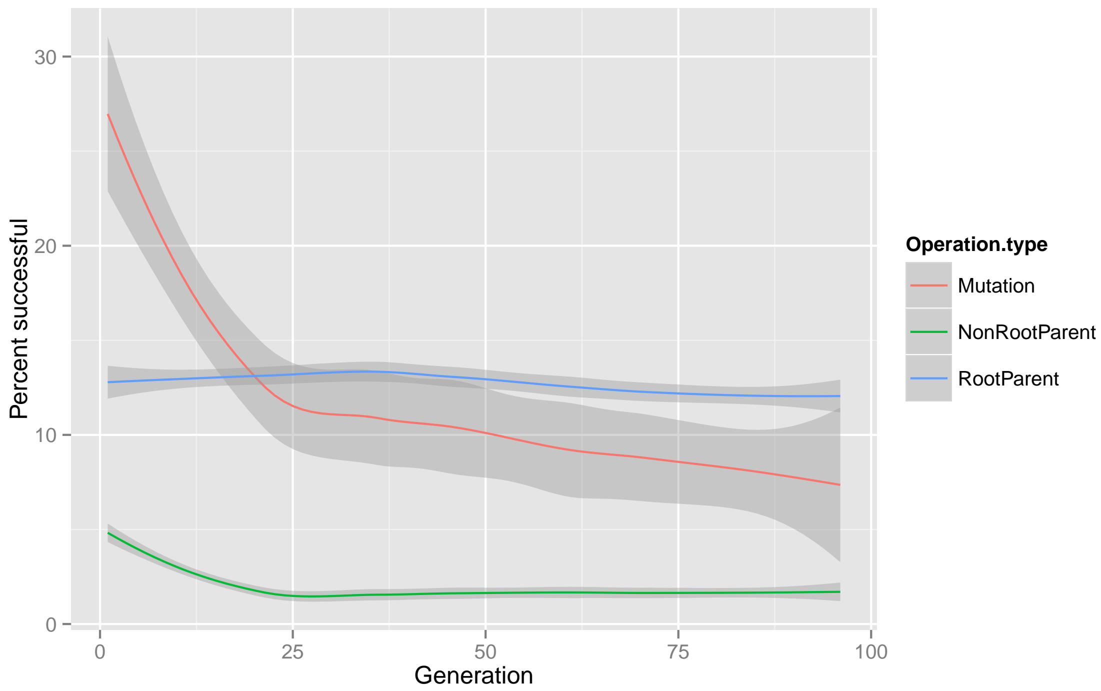

Figure 4: Percentage of cases where the child is fitter than the parent in three 1K runs. Shadows indicate the variance across the three runs.

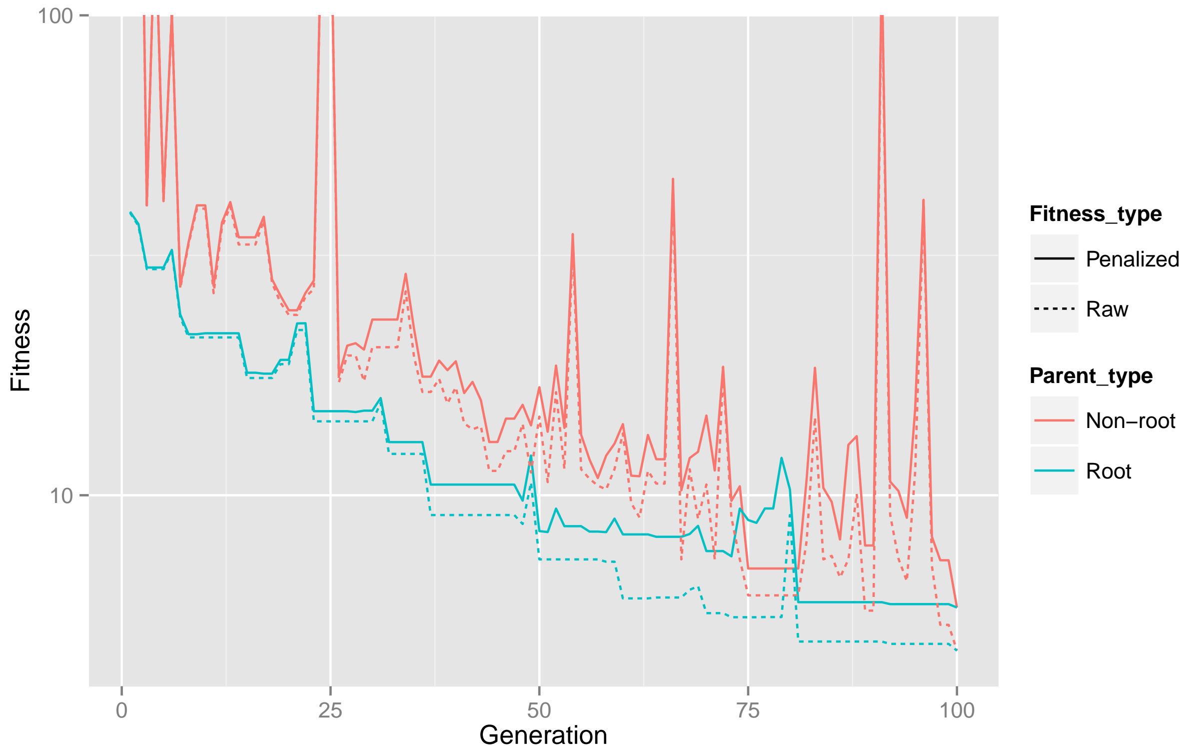

Figure 5: Root versus non-root fitness in ancestry of best tree from final generation of a 10K run. The raw fitness is the fitness before the size penalty is added. A logarithmic scale has been applied to the fitness axis.

6 Conclusions

A critical point is that graph databases like Neo4j don't make anything possible that was once impossible; they instead make these things vastly simpler and allow open-ended exploration. Each of the queries and questions we've discussed could be handled by, for example, special purpose code added to the evolutionary system to capture that specific information. Many, if not most, of these questions have no doubt been addressed in a piecemeal fashion by previous research. Researchers observed [2] nearly 15 years ago that root parent lineages were significant, and that these lineages quickly coalesced into a shared common ancestor, but that was using a custom system to track that data, and there has been limited follow-up by others since then. We suspect a significant reason for the lack of similar work is simply the effort required to collect and analyze the substantial amount of data this entails, and the lack of good tools to simplify that process.

Now that we have access to a plethora of data from GP runs, we are capable of potentially deducing new, important patterns. For future research, we have identified several topics to possibly pursue.

One topic which may benefit from further investigation is the percentage of effective transformations where the child is more fit than its parent or parents. We could, for example,

Query 4 Cypher query to find the set of common root ancestors of all the individuals in the final generation.

```
MATCH (child:Individual {generation: 100})
   <-[:ELITISM|PARENTOF|MUTANTOF|ROOT_XOOF*0..]-(parent)
   <-[rel:ELITISM|PARENTOF|MUTANTOF|ROOT_XOOF]-(grandparent)
RETURN DISTINCT id(parent), type(rel), id(grandparent);
```

dynamically tune mutation and crossover percentages over time to increase the likelihood of generating effective transformations. Additionally, we could also investigate the effect of forcing the root parent to have a better fitness than the non-root parent, since the data in Figures 3 and 4 clearly indicates that offspring from such a combination have a much higher probability of having improved fitness.

Another potential route to explore how often long-term persistent clades exist. It would also be interesting to see if and when clades interbreed, or if they are completely separate species. If there is any interbreeding, this would allow exploration into how interbreeding contributes to the overall fitness of the clade.

A final area for further investigation is to analyze the fitness over time of the "winning" lineage. In Figure 5, we saw several areas were the fitness became worse before becoming better. Although spikes of poor fitness do not happen prior to all improvements, these spikes do happen frequently enough to be potentially significant. Investigation into the characteristics of changes in individuals in these spikes could lead to a better understanding of how improvements in fitness occur.

There are many other questions that could be potentially addressed because of the sheer amount of data Neo4j allows us to collect and process. So far we have done very few runs, and only on a single test problem. Doing more runs, and exploring a variety of problems would help us understand how representative some of these results are.

In summary, while this work has demonstrated that Neo4j is a useful tool in recording and analyzing data collected from genetic programming systems, there is much further work to be done with this information.

Acknowledgements

David's work was supported by the Morris Academic Partners program at the University of Minnesota, Morris. Many thanks to Nicholas Cornhill and Emma Ireland for their early help in connecting evolutionary computation systems to Neo4j.

References

[1] LUKE, S. *Essentials of Metaheuristics*, second ed. Lulu, 2013. Available for free at http://cs.gmu.edu/˜sean/book/metaheuristics/.

[2] MCPHEE, N. F., AND HOPPER, N. J. Analysis of genetic diversity through population history. In *Proceedings of the Genetic and Evolutionary Computation Conference* (1999), vol. 2, Citeseer, pp. 1112–1120.

[3] POLI, R., LANGDON, W. B., AND MCPHEE, N. F. *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[4] ROBINSON, I., WEBBER, J., AND EIFREM, E. *Graph Databases*. O'Reilly Media, Inc., 2013.

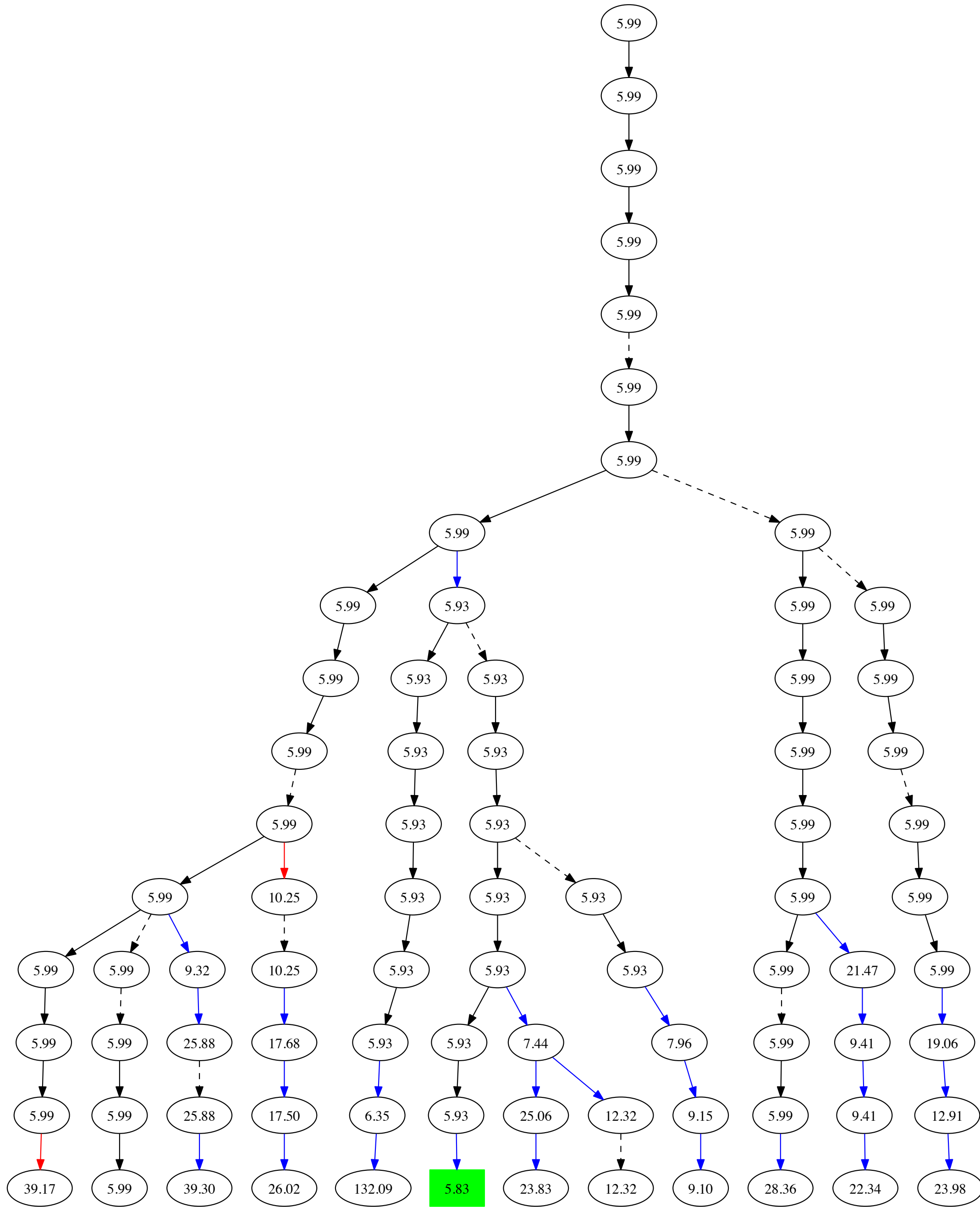

Figure 6: Results of finding the common ancestor between the “winner”, highlighted by a green box, and 11 of its close relatives. Mutation relationships are highlighted by red arrows. Crossovers are blue arrows. Reproduction is a dotted black line and elitism is a solid black line. Note that this is an abbreviated ancestry stopping at generation 84; the common line continues back all the way to generation 1.