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.

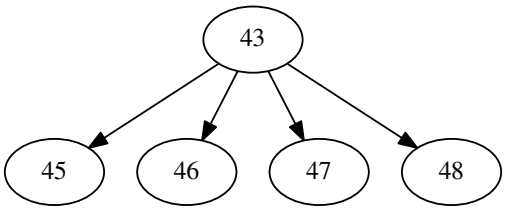

Figure 1: Results of Example Query

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 all of the runs, the configurations remained consistent, with the exception of population size. In the several runs that were performed, the population size was either 1,000, or 10,000. The target function was as follows:

$$\sin(x)$$

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 operations allowed in order to achieve an optimal solution to the target were the binary operations: addition, subtraction, multiplication, and division. Since division is undefined when the denominator is equal to zero, we implemented protected division. In our protected division, if the denominator equals zero, then regardless of the numerator value 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 $x = 0$. Therefore, when evolving x/x in a individual we will continue to obtain the value one.

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 is a way to write a function that places the operator before its arguments. For example, a tree of the function $x + (x * 4)$ would be represented by the following array: [+, x, *, x, 4]. The tree's fitness is the sum of the absolute error between the target function and the tree at all predetermined variable values. The absolute error is the measured value of a quantity x_0 and its actual value x, given by the equation $\Delta x = x_0 - x$.

After the tree's fitness is computed, we add a constant to the fitness as a means to penalize particularly large trees. This encourages trees to not become excessively large, and is commonly referred to in genetic programming as bloat control. We selected this constant to be a hundredth of the tree size. 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.

To create the initial population, we implemented an algorithm called PTC2 [1]. The PTC2 algorithm creates trees by randomly adding operators to an array (leaving blank indices where appropriate for arguments) until a specified length is reached. The blank indices are then filled by leaves (variables and constants). Leaves consist of 63% variables and 37% constants.

In order to selection those individuals which will produce the next generation, we implemented tournament selection. In our tournament, two individuals are selected from the entire generation. The individual with the best fitness of the two selected is chosen to propagate its code in some capacity on to the next generation. This process repeats for the creation of every individual in the next generation.

As discussed previously, individuals may pass on their code by three different means: crossover, mutation, and reproduction. Crossover makes up 90% of all transformations, mutation 1%, and reproduction accounts for the remainder. Reproduction is relatively straightforward. The individual which wins the tournament is simply copied on to the next generation. A variation on reproduction is elitism, where an individual is copied to the next generation by merit of it fitness. In other words, within a generation, 1% of individuals with the best fitness skip tournament and are simply copied over 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 selected from the population to enter the tournament, and the winner is chosen for mutation. However, rather than simply copying this individual on to the next generation, a random index from within the tree is selected. If the index is a operator, the subtree starting at that operator is removed. A subtree is a section of a tree that is itself a valid tree. Otherwise, if the index is a leaf, the index is removed. In both cases, the removed index or indices are 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 in controlling bloat.

Crossover differs from the previous transformations in that it makes two calls to the tournament in order to receive two individuals to produce a single child individual in the next generation. The first tournament results in a root parent. Within this parent, similar to mutation, a random index is chosen. If the index is an operator, the subtree is removed, otherwise only the index is removed. The removed index or indices are then replaced by a subtree randomly selected from the individual that won the second tournament.

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, 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 index at which the root parent was altered by a transformation) is also included as an attribute. These attributes are summarized in Table 1.

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 2.

Individual Attributes								
Transformation	**Tree**	**Fitness**	**Penalized Fitness**	**Generation Number**	**Transformation Type**	**Run ID**	**Unique ID**	**Cut Point**
Elitism	X	X	X	X	X	X	X	
Reproduction	X	X	X	X	X	X	X	
Mutation	X	X	X	X	X	X	X	X
Crossover	X	X	X	X	X	X	X	X

Table 1: Chart summarizing attributes that are recorded for individuals produced by each transformation type.

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

Table 2: 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 relations because two parents were selected in tournament.

5 Results

To obtain the results presented here, we completed six runs at population 1000, and one 10000 run. From these runs, we were able to retrieve a variety of interesting data within a reasonable amount of time. The following are the questions we asked of the database:

- *How many individuals in the initial generation have any root parent descendants in the final generation?*
- *How well do mutations perform against crossovers where the root parent is more fit than the non-root parent, and vice versa?*
- *Do a group of individuals have a common ancestor and in what generation does that ancestor occur?*
- *What does the fitness of the "winning" line look like over time?*

To answer the first question, we implemented the following query:

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

The MATCH statement describes that s and n are individuals in generation one and one hundred respectively using indexing. The relationship between them must be elitism, reproduction, mutation, or root parent crossover and must have a path depth of 99. The RETURN statement returns all individual ids along with its fitness in the initial generation that fit the criteria. This query produces the set of individuals that have descendant in the

Individual	Penalized Fitness
2595	38.9820979981526
3325	40.36373034994929

Table 3: List of individuals in the initial generation of the 10K run which produced root descendants in the final generation.

final generation. These individuals and their penalized fitness are represented in Table 3.

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 relatively small. Furthermore, while neither of these individuals demonstrated the best initial fitness (24.18), both do appear in the top 5% of fit first generation individuals. Whether this high fitness rate is consistent across multiple runs is unclear.

To determine the effectiveness of mutation versus crossover we wrote several queries similar to the following:

```
//Count total number of crossovers

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

//Count total number of crossovers fitting the description

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

These two queries above 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. The other two cases, mutation and non-root crossover (the child was more fit than either parent and non-root parent was more fit than the root parent), represented in Figure 2 had similar structures. The results of this query may be seen in Figure 2.

To confirm the data we found in Figure 2, we used the same queries on three 1K runs to obtain the graph in Figure 3. Again, we obtained similar data. 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 as time progressed dropping

as low as 2% over the span of a five generation section. 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 crossover is a stable transformation over time.

To find a common ancestor of entire last generation, we executed the following query:

```
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);
```

From this query we found several interesting occurrences. An example of a common ancestry graph can be seen in Figure 4 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 chance of survival. In the 10K run, two distinct clades were produced. One clade flourished, accounting for 99.76% of the final population while the second clade consisted of only 24 individuals. These two clades are descended from the individuals presented in Table 3. While there are two individuals that are direct ancestors, descendants of the more fit individual predominated. Given more time, the second clade most likely would have gone extinct.

The fourth and final question asked was answered by the following query:

```
START startNode=node(99000)
MATCH rootPath = (startNode)
   <-[rel:ELITISM|PARENTOF|ROOT_XOOF|MUTANTOF*]-(parentNode)
RETURN DISTINCT rootPath;
```

From this query, we obtained a graph of the entire root line of the "winning" individual and we were able to visualize the differences between the fitness and penalized fitness of each individual in the line. Both the root line and non-root line are displayed. The root line follows the path of individuals along the root parent crossover transformation and all single parent transformations(elitism, reproduction, mutation). Non-root follows the non-root crossover transformation with single parent transformations demonstrated in Figure 5. The resulting information regarding the fitness of both root and non-root ancestors of the most fit final individual over all generations is presented in Figure 6.

As can be seen in Figure 6, 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.

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 ex-

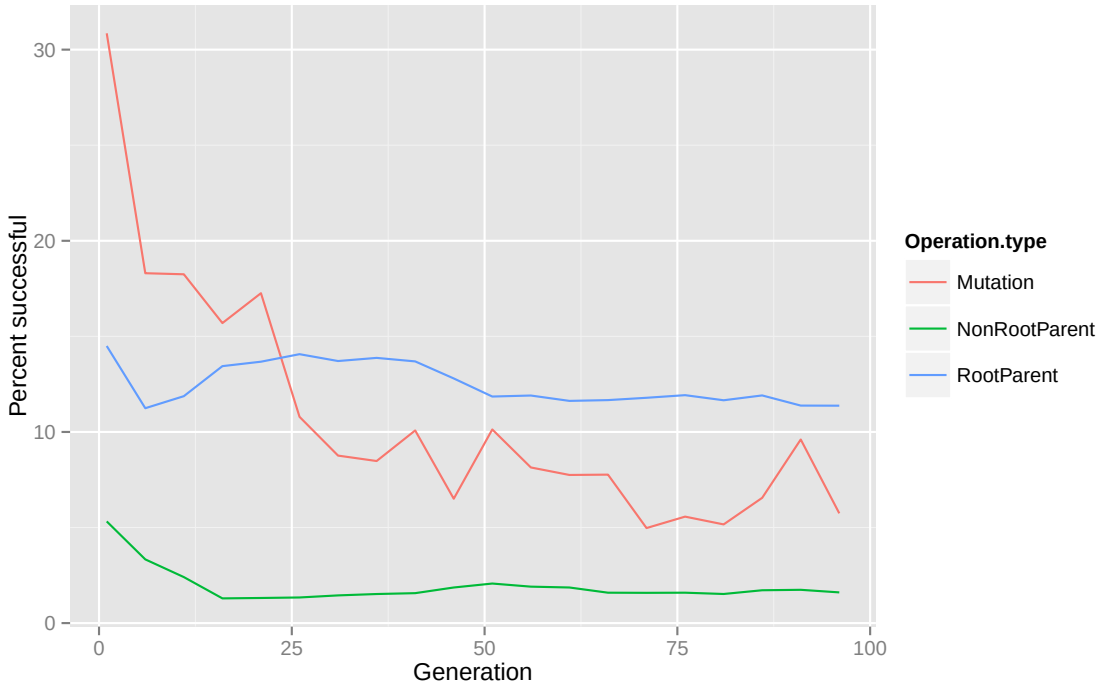

Figure 2: Percentage of Cases Where the Child is Fitter Than the Parent Over Five Generation Blocks in 10K Run

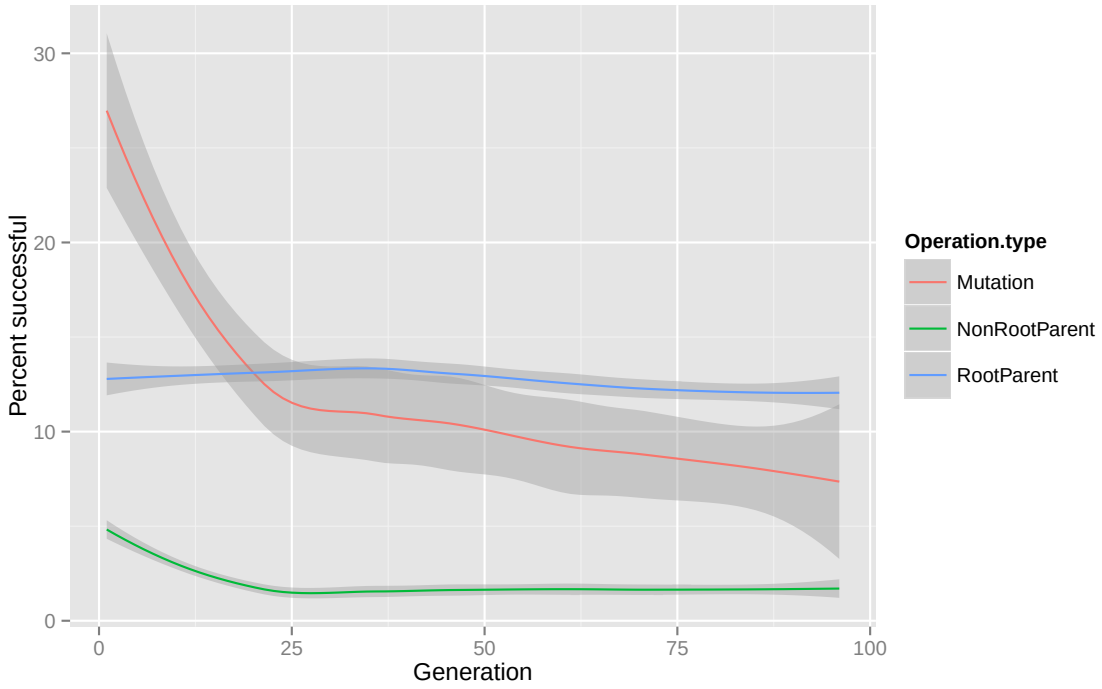

Figure 3: Percentage of Cases Where the Child is Fitter Than the Parent Over Five Generation Blocks in Three 1K Runs. Shadows indicate the variance between each of the three runs.

ploration. Each of the queries and questions we've discussed could be handled by, for example, special purpose code added to the evolutionary system to specifically capture that specific information. Many, if not most, of these 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 specialized 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 myriad of data from a genetic programming run, we are capable of deducing patterns. For future research, we have identified several topics to pursue.

One topic which may benefit from further investigation is the percentage of viable transformations (transformations where the child is more fit than its parent or parents). One method of going about this is to dynamically tune mutation and crossover percentages over time to increase the likelihood of generating viable transformations. Additionally, we could also investigate the effect of forcing the root parent to have a better fitness than the non-root parent due to its record of having a low success rate, as seen in Figure 2 and Figure 3.

Another potential route to explore is 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 identified here for further investigation (although there are many potential others that have been left unmentioned because of the sheer amount of data) is to analyze the fitness over time of the "winning" lineage. In the Figure 6, 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 significant. Investigation into the characteristics of individuals in these spikes could lead to a better understanding of how improvements in fitness occur.

In summary, while this work has demonstrated that Neo4j does in fact provide 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

References

[1] LUKE, S. *Essentials of Metaheuristics*, second ed. Lulu, 2013. Available for free at http://cs.gmu.edu/$\sim$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.

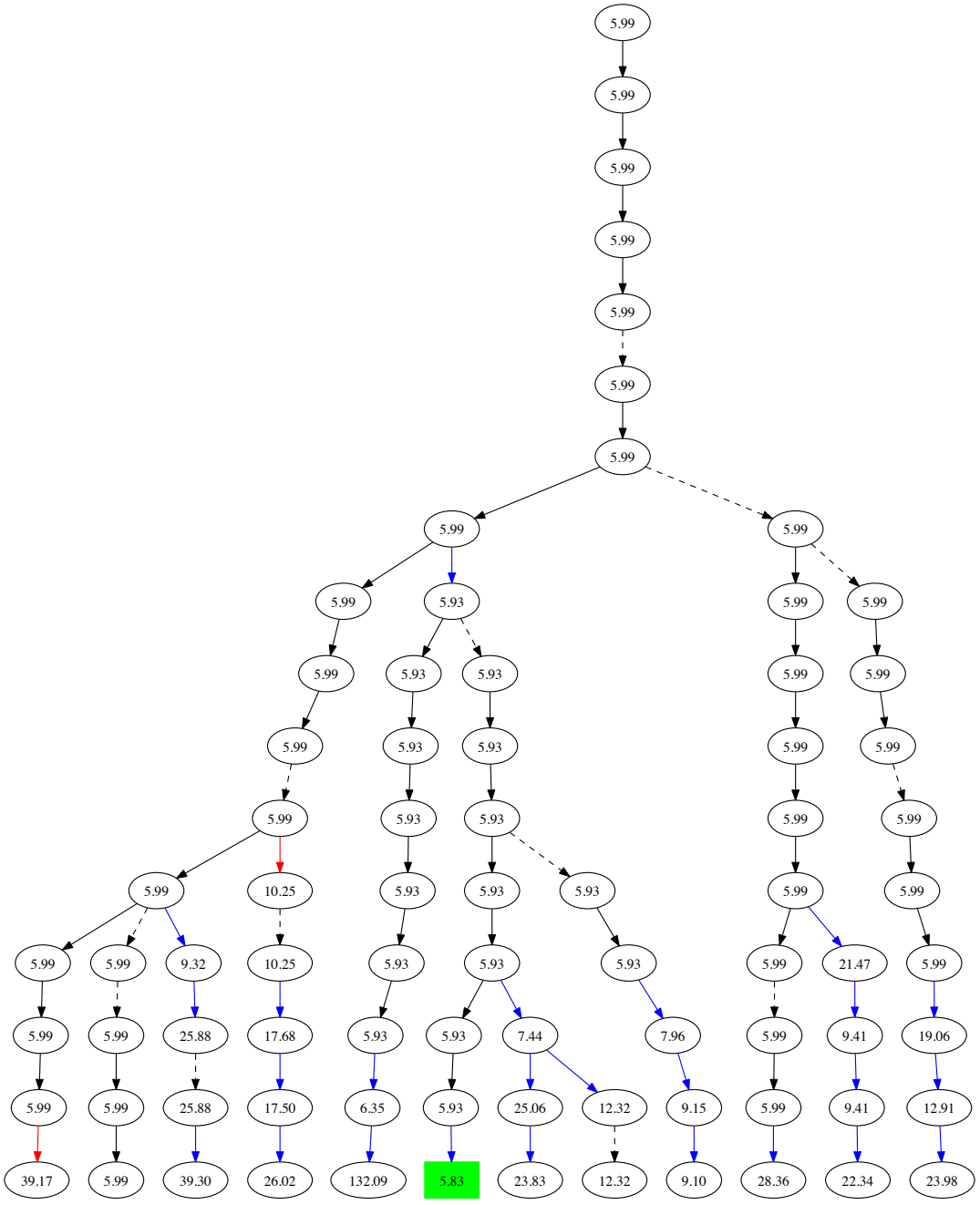

Figure 4: Results of finding the common ancestor between the “winner” highlight by a green box and 11 of its close relatives. Mutations relationships are highlighted by red arrows. Crossovers are blue arrows. Reproduction is a dotted black line and elitism is a solid black line.

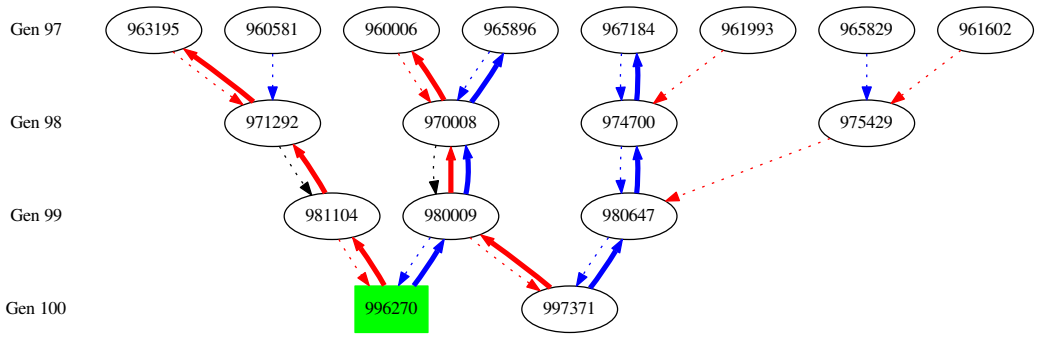

Figure 5: The root and non-root paths of two individuals in the final generation. Dotted arrows represent transformations (blue for root crossover, red for non-root crossover, black for reproduction). Solid arrows represent the root and non-root ancestry paths (blue for root crossover parent path and red for non-root crossover parent path). Solid blue paths traverse dotted blue and black paths, and solid red paths traverse dotted red and black paths. The individual marked by the green box is the most fit individual in the final generation.

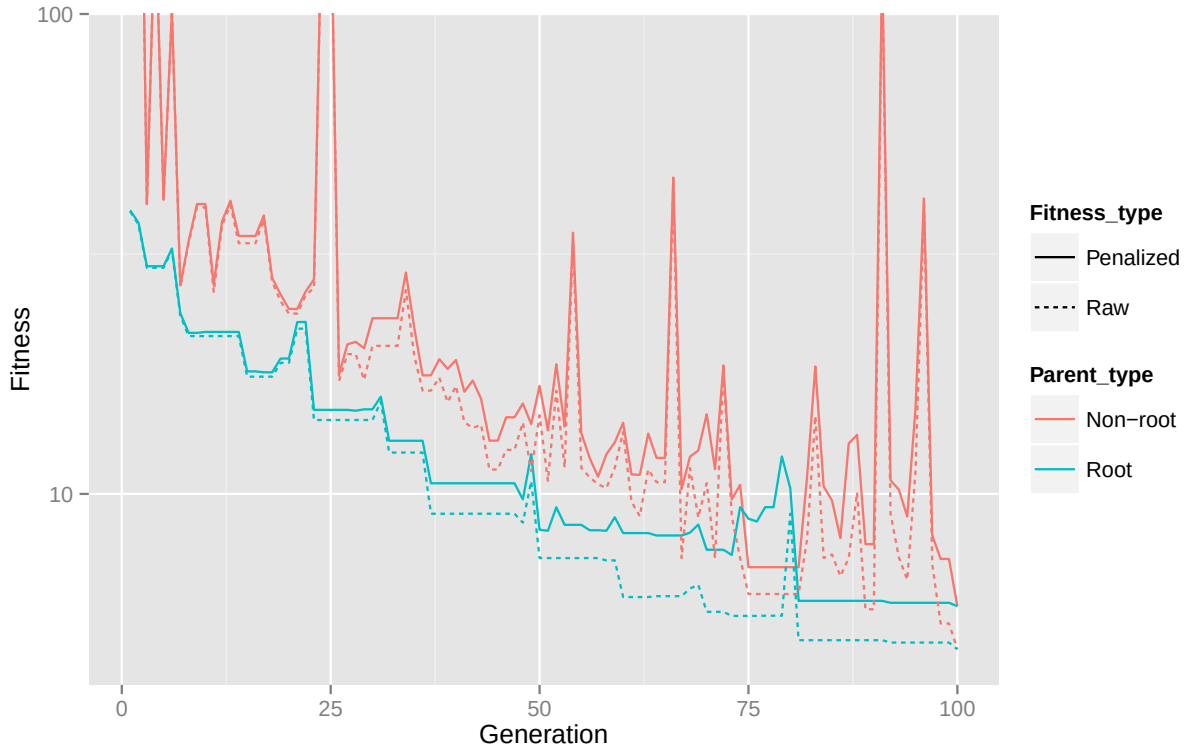

Figure 6: Root Versus Non-Root Fitness in Ancestry of Best Tree from Final Generation of a 10K Run. A logarithmic scale has been applied to the fitness axis.