Using Phylogenetic Analysis to Enhance Genetic Improvement

Penny Faulkner Rainford
faulknerrainford@gmail.com
School of Computing and Communications
Lancaster University

Barry Porter
b.f.porter@lancaster.ac.uk
School of Computing and Communications
Lancaster University

ABSTRACT

Genetic code improvement systems start from an existing piece of program code and search for alternative versions with better performance according to a metric of interest. The search space of source code is a large, rough fitness landscape which can be extremely difficult to navigate. Most approaches to enhancing search capability in this domain involve either novelty search, where low-fitness areas of the search space are remembered and avoided, or formal analysis which attempts to find high-utility parameterizations for the genetic improvement (GI) process. In this paper we propose the use of phylogenetic analysis over genetic history to understand how different mutations and crossovers affect the fitness of a population over time for a particular problem; we then use the results of that analysis to tune a GI process during its operation to enhance its ability to locate better program candidates. Using phylogenetic analysis on 600 runs of a genetic improver targeting a hash function, we demonstrate how the results of this analysis yield tuned mutation types over the course of a GI process (dynamically and continually set according to individual's ancestors' ranks within the population) to give hash functions with over 20% improved fitness compared to a baseline GI process.

ACM Reference Format:
Penny Faulkner Rainford and Barry Porter. 2022. Using Phylogenetic Analysis to Enhance Genetic Improvement. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528789

1 INTRODUCTION

Genetic algorithms have long been used across many different computing applications, from finding the ideal parameters of systems with large parameter search-spaces [1], to a useful meta-heuristic approach to code repair [5, 6]. In this paper we focus on genetic code improvement (GI), which aims to derive new versions of existing computation logic that are optimised towards a utility function such as calls-per-second. GI has been approached in a wide variety of ways, from modifying bytecode to working on syntax trees or with grammar models of the target programming language [11]. We use an approach based on the compiler-derived syntax tree of a piece of source code, augmented with grammar rules guiding which mutations are valid. We particularly target our GI approach towards emergent software systems, which are composed of many small interchangeable building blocks such as a sorting algorithm or a hash table [13]. At runtime, emergent software systems monitor their environment and learn which variants of each building block are best suited to each set of deployment conditions encountered. Because each building block in these systems is relatively small (100-200 lines of code), our particular GI approach mixes new code synthesis with more traditional mutation types so that sufficient new genetic material is available.

Despite the success of GI processes, finding the ideal parameter setting for genetic algorithms remains a key challenge; at present this tends to be done either through a general search of the parameter space by hand, or through the use of meta-heuristics [10]. Finding the ideal parameter settings for elements such as mutation bias and population size aid in allowing a GI process to reach the highest-utility parts of a search space, and these ideal parameters may be different for different target problems (such as improving a hash function, or improving a cache eviction policy). In this paper we propose the use of *phylogenetic analysis* – i.e., analysis of the way in which particular modifications contributed to utility across successive generations – from existing GI operation on a given target problem to inform the ideal parameter settings for GI against that problem. This approach has the potential to automatically derive the ideal GI configuration, including the relative weightings of each type of mutation, for each target problem as it is encountered.

In this paper we report on an initial study into the analysis of historical GI information in order to find possible improvements to GI parameter settings for a target problem. We consider the independent effects of both crossover and our different mutation types to see exactly what has been most effective in improving the relative performance of individuals. We consider these effects not just at the end of a GI run, but in discrete time quanta during a GI process, to then inform potentially different parameter settings that are introduced in different generations of a new GI process. In this feasibility study we use the results of 600 previous genetic improvers targeting a hash function, the objective of which is to most evenly distribute a set of keys across a set of buckets in order to minimise average lookup time. These historical runs worked across different data treatments and target data classes and variant fitness functions. We use the results of this analysis to define new parameter settings for our GI system on a single setting to examine whether this results in faster training, higher-performing individuals, better generalisation of population members to unseen data, or more reliable results across an entire GI process.

Our results show positive effects in modifying mutation weights based on generation and an individual's lineage to improve performance on unseen data, leading us to conclude that the approach has some significant merit in enhancing GI's ability to find better program variants. We provide a replication package, with detailed

GECCO '22, July 9–13, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9237-2/22/07…$15.00
https://doi.org/10.1145/3512290.3528789

instructions, with which all of our results can be repeated [1]. In the remainder of this paper we first examine related work (Sec. 2), then introduce our GI algorithm (Sec. 3). We then examine our phylogenetic analysis of this algorithm in terms of mutation and crossover (Sec. 4.1-4.2), and present our results from experiments that use GI parameter settings informed by this analysis (5).

2 RELATED WORK

Our general approach aims to improve how a GI process searches a fitness landscape to find higher-value individuals. The main existing approaches to this problem involve either novelty search during execution, or empirical analysis of the problem space.

The most common method used is novelty search [8, 16] in which unique solutions are saved for future reference. In the selection process, the GI system then performs a comparison with each previously saved solution and only uses those individuals in a new population which represent new areas of the search space not present in the stored list. This approach is effective in searching a wider area of the fitness landscape, avoiding repeatedly falling in to the same local minima, and can be applied independently alongside a wide range of other GI process optimisations. It does not by itself, however, guarantee good coverage of a fitness landscape [3]. Novelty search also has an inherent limit to how many individuals can be usefully stored before the cost of enumerating them, when evaluating new individuals, slows down the GI process to a point of being non-viable [14]; because the entire genetic material of each individual must usually be stored, there are memory constraints on the total number of points that can be saved. So while useful novelty search is not a substitute for well-tuned GI parameters which guide a process towards high-value areas of a fitness landscape.

The use of empirical analysis in GI attempts to quantify an overall fitness landscape in order to derive the most appropriate mutations or parameter settings to reach high-utility areas [15]. This is related to our approach of using historical information of the fitness landscape, relative to the modifications that were made in each generation, to inform future decisions on how the GI system proceeds. A complete empirical analysis, however, is only viable for highly constrained fitness landscapes, where success and failure are easily defined. In general the use of synthesised code for insertion-mutations, as employed in our approach to augment small amounts of initial starting genetic material, presents far larger and less well-defined fitness landscapes than GI without synthesis.

Our approach of historical analysis is inspired by the field of phylogenetics, most commonly applied to biological evolution. In biological study, researchers are often curious about when different species appeared or what contributed to the evolution of particular genetic traits. When studying genetic history to answer these questions, researchers are faced with a similar problem of the sheer size of the probability landscape in their attempt to track evolution backwards in time. Even with complete or fragmented DNA available, the probability space for tracking the sexual recombination and mutation of DNA back in time is intractable to formal analysis. Instead a combination of heuristics [4, 7] and phenotype comparisons across generations are used to predict a likely tree of evolution backwards in time which is compared with known

[1]http://www.projectdana.com/research/gecco2022rainford

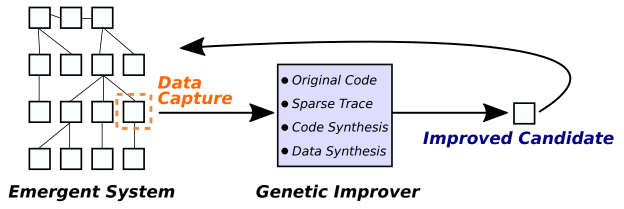

Figure 1: Our overall approach, with data capture from a running emergent software system, requests for improved variations of particular building blocks to a GI system, and a GI process which uses mixed code synthesis to counter a low volume of available genetic material, along with input data synthesis to counter a sparsely sampled input trace towards which we are optimising.

changes to the environment to derive possible drivers of change in evolutionary pressure towards particular adaptions [2, 9].

Unlike the biological analogue, we are able to access complete information about the exact past set of re-combinations and mutations across generations; by analysing these relationships relative to phenotypic improvements (or negative changes) in the code we hypothesise that we should be able to derive the correct pressures to create efficient evolutionary pressure towards our goals. In effect we use a very similar phylogenetic analysis approach to *drive* rather than to *explain* evolution.

While our approach does require the analysis of large bodies of data, from previous uses of a genetic improver, the results of this analysis require minimal storage and can provide useful changes to many future uses of the improver – such that the proportional increase in memory and processing per-run is minimal. Our modified GI process, which is able to support dynamic and individual parameter settings across generations, does very slightly increase the storage size of our individuals but only by the storage of 3 additional floating-point numbers.

3 ALGORITHM

Our overall GI framework is illustrated in Fig. 1. An emergent software system is assembled from a large collection of small building blocks, such as stream processors, memory cache implementations, hash tables, and so on, and is deployed into a real environment. Once that system has learned the best composition of blocks for a given environment, it will select one of those building blocks and capture a short trace of the method calls that are issued to that block within the present environment. This trace is then sent to a GI system, along with the source code of the building block from which it was captured, so that the GI system can attempt to generate an improved variation of that building block which has higher performance for the given input data sample. If the GI system is successful, the improved building block is pushed back to the emergent software system which uses real-time learning to determine if the proposed improvement does yield higher performance for the intended environment conditions in deployment. Our GI implementation is designed to operate on source code written in the Dana programming language, which supports sound hot-swapping of code for emergent software systems [12].

For the purposes of this study we examine only the GI element of this overall concept. We focus on one particular building block throughout our experiments (a hash table), and we assume that short traces of function calls to this block have already been captured by the emergent system. We also assume that the GI process is able to identify the *hash function* of the hash table implementation as the specific area in which to focus when generating improved variations; in practice this focus area could be determined using function call frequency or CPU intensity analysis.

Algorithm 1 Genetic Improvement Algorithm

```
for $i = 0$ to $generations$ do
    if $i == 0$ then
        create initial population of clones
    end if
    mutate a % of the population
    check fitness of all population members
    if $i\%5 == 0$ then
        check performance on unseen data of all population members
    end if
    select new population (roulette wheel)
    crossover a % of the population
end for
```

The core of our system is then based on a typical genetic improvement process, Algorithm 1, using mutation, fitness, selection, and crossover. We include crossover in this work to make use of existing code to expand the code base of members of our population. Because our volume of starting genetic material is very small, we also include *new code synthesis* in our set of available mutations, in combination with more typical mutation types. The crossover and mutations in our system are described in more detail in the relevant sections.

Our selection process, where we choose which individuals to select for copying from one generation for inclusion in the next (with associated crossover/mutation), uses a rank-weighted roulette wheel approach. Each population is ordered by fitness and ranked. This rank is then used such that the fittest individuals have the highest probability of selection for the next generation. Selection is done with replacement, so some individuals will appear multiple times in the following generation, and some will be completely absent, with a (small) possibility of even the worst individual being selected for the next generation.

3.1 Rank vs Fitness

Throughout this paper we use *relative fitness* rather than raw fitness to analyse the results of a GI process. We do this because our raw fitness is drawn from the system clock timing our code running (such as how quickly it takes a candidate hash table to process 1,000 'put' operations). This timing approach includes some degree of noise, both between different machines and, at times, on the same host machine. To account for this noise we measure our original code's raw fitness at the start of each run, with the relative fitness derived as a proportion of that original fitness. Because faster is better in our target problem, this means an individual with a relative fitness of 0.8 is a 20% improvement on the original code.

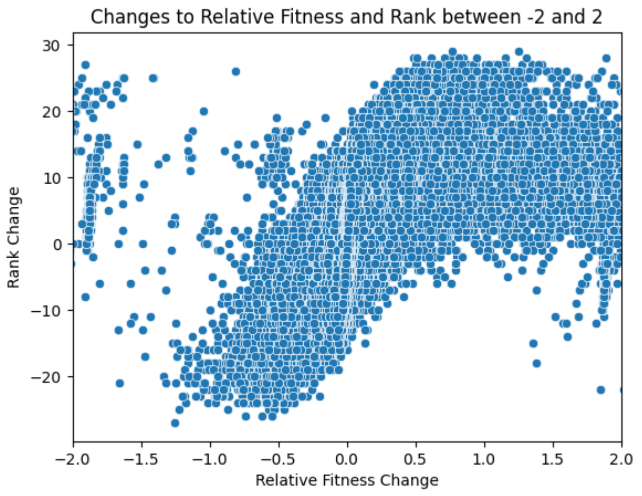

Figure 2: Scatter graph showing the change in relative fitness (scaled to the fitness of the original code) against the change in rank both changes from the individuals immediate ancestor, limited to the rank -2 to 2.

In this research we use the results of many runs, including those that train on different data and with different data treatments. We could examine changes in relative fitness for each individual run in this data, as each run has a common end-point of 200 generations; however, within this dataset some examples have less than 20% improvement by the end while others have almost 40% depending on the training data that was used. Because we want to combine many GI runs for phylogenetic analysis, the difference in improvement between runs and the internal relative fitness noise makes comparison based on relative fitness very difficult. We instead focus on *relative rank* within a population, rather than fitness. Rank is determined by ordering individuals based on their fitness and assigning ranks, e.g. the fittest (fastest) member of the population is assigned rank 1, the second fittest rank 2 and so on with the worst individual given rank 30 (all population sizes in this work are 30).

To verify that improvement in rank does correlate sufficiently with improvement in fitness we compare the two, shown in Figure 2. If this provided a clean correlation, in the form of a strong positive correlation, there would be no need to use rank rather than relative fitness as the two would be equivalent. Conversely if there had been a large number of points in the top left and bottom right quadrants it would suggest that there wasn't a sufficient correlation between the two values. The graph shows that there is not a perfect correlation, but that in most cases individuals with a better fitness do have a better rank, and vice-versa, such that the bottom-left and top-right quandrants feature most of the points. This lends confidence to the use of relative rank throughout our analysis, allowing us to effectively compare the results of our 600 runs even though they have differences in their training data and timings.

4 ANALYSIS

In this section we present our analysis of 600 runs of our GI process on a hash function; these processes are each aiming to derive an improved hash function which most evenly distributes keys across available buckets and therefore minimises average key lookup time. Among these 600 runs we have a large range of different input data sets (lists of keys) for which the GI system is trying to yield an improved hash function variant. We analyse these 600 runs first in terms of mutations and second in crossover.

Our output from these runs is saved to a comma-separated file with groups of values, for each generation, on the following data:

- **ID** - unique identifier of each individual
- **Ancestry** - the ID of the individual's immediate ancestor (recipient code in the case of crossover)
- **Fitness** - raw and relative values for fitness and performance and the difference between them and their ancestor's values
- **Rank** - individual rank, ancestors' rank, change in rank, and average rank of immediate 5 ancestors (based on the experience of performance checks every 5 generations providing an accurate trend of the training of the population)
- **Modification** - boolean flags indicating occurrence of mutation and crossover, ID and rank of crossover donor code, type of mutation including mutation type, type of token affected, and operator affected.

Our analysis programs use this data to construct a map of how modifications, fitness, and rank changes propagate across individuals over time through successive generations of our GI process.

We note that our analysis of existing runs examines only the independent effects of mutation and crossover on individuals in the population so that we can accurately infer the effects of these modification types in isolation, ignoring those individuals that underwent both or neither change. While every one of our 600 studied runs shows continuous or mostly-continuous improvement of the best individual throughout the run, on some of our analysis results it therefore appears that rank changes are only negative; these results are because much of the improvement took place within the unstudied group of combined modification.

4.1 Mutation

Our GI system has three types of mutation available: insertion, modification, and deletion. Each time a mutation is called for, we first select one of these mutation types at random, then choose from the available specific mutations of that type – such as inserting a new variable declaration, or modifying an operator.

Insertions are the most complex mutation in our system as they use newly-synthesised code. When an insertion is selected we first identify a node in the syntax tree at which the insertion will take place. The particular kind of insertion is then selected at random from declarations, operations, and control structures. Declarations are formed using (i) a variable name drawn from a list of ordered strings, based on the 26 letters of the English alphabet, (ii) a random type for that variable, such as int or char[], and (iii) a selection of either an existing variable of that type or a random value of that type for assignment of the starting value of the newly declared variable. These components are assembled as a declaration with a value assignment, and inserted into the syntax tree at the chosen position. The insertion of new operations begins with the selection of an operator followed by the search for matching variables in scope for use as operands; if there are insufficient operands available the process fails. This newly formed operator is then inserted at the chosen position. Finally, the insertion of new control structures are drawn from three types: **if**, **while**, and **for** structures. The **for** loop simply selects a variable name for the iterator variable, in the same way as choosing a declaration variable name, and selects an iterator limit from the available integer variables that are in scope. The **while** loop and **if** statement options select a comparison operator at random, and the matching variables from the scope as operands. Control structures are then inserted including a new scope within them which remains empty to begin with.

Modifications mutations allow for any arithmetic operations and also the conditional statements of **if** and **while** control structure to be changed. In both cases we choose at random between modifying an operator or an operand. For the operator case, we can swap an operator such as '+' for one with the same number and type of operands, and the same return type, such that it can be exchanged for the existing operator with no other adjustments needed. For the operand case, we exchange one operand of a chosen operator for a different variable which is currently in scope and which has identical type (where variables-in-scope includes subfields of objects or array cells indexed by an available integer). If matching operators or operands cannot be found then the modification fails.

Deletions select a token and identify it as either a declaration, operation, or control structure. For operations, which do not create new variables, the token is removed from the syntax tree with no further checks. Declarations, by contrast, create a new variable in the program, meaning we must first check to ensure the variable is not used elsewhere in the program before we consider the declaration for deletion. Finally, control structures are only removed if they are empty, such that the removal of a control structure usually means one or more other deletions must first have taken place within the scope of the control structure to render it empty.

In the data used here, from our 600 GI system runs, all random selections in the mutation process were done based on a uniform probability distribution for the choice of token, operand, insert type, etc. These uniform distributions are one configuration element that could be modified in future GI system runs, based on this analysis, to provide more effective GI optimisation of a target problem. Changes to these distributions could be made at a population or individual level, and could be changed based on generation number, current individual rank, historical rank, and rank trends.

4.1.1 *Average Rank Change for Mutation over Time.*

In our phylogenetic inspired analysis we begin by looking for trends over time in the apparent effects of each of our main three mutation types: insertion (I), modification (M) and deletion (D), shown in Figure 3.

On all of our rank-based graphs lower numbers on the Y-axis are better; on this graph we can see that each class of mutation at some point contributes to lower rank, but that each mutation class viewed individually increases rank (i.e., has an undesired effect) over time. Insertion (blue) has a particularly deleterious effect in early generations but improves over time. Modifications (green) and deletions (orange) both result in steadily worse ranks over time, although the change in rank for deletions is much smaller

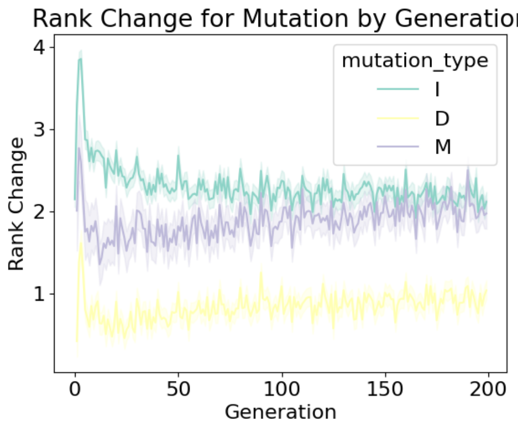

Figure 3: Rank change resulting from mutation of particular type over time. Mutation types: Insertion (I, green), Modification (M, purple) and Deletion (D, yellow).

and suggests that deletions are far more likely to yield neutral drift. Neutral drift can be a useful property in better searching a fitness landscape as it allows us to find new areas to search despite continuous selection pressure to focus on improvements.

By increasing the weightings on modification and deletion in general we may be able to encourage further neutral drift and also reduce the deleterious effects of insertion in the early generations. To acknowledge, however, the importance of insertion to the finding of new solutions, and the improvement in its effect over many generations, we may opt to taper the weights of insertion over the course of the run so that the weightings are again equal by the final generation of a GI process.

4.1.2 Average Rank Change for each Mutation vs Historical Average Rank. We next consider if the history of each individual in our system can provide indicators for which mutations would be most efficient on per-individual basis. To do this we examine the average rank of an individual's last 5 ancestors, which is informative both on the history of an specific individual and on the general survival times of individuals of different ranks. In general, because our selection process prefers lower rank individuals, it should be likely that those with higher rank produce less to no offspring.

In Figure 4 we plot each individual that was subjected to a mutation, from each generation, in each of our 600 GI runs. The X-axis represents the average rank of an individuals five most recent ancestors in its population of 30 (higher ranks are worse), while the Y-axis represents the degree to which that individual's rank has changed relative to the rank of its most recent ancestor. Points that appear in the top-left quadrant of each graph are therefore individuals who are on average from the best half of their population, and whose rank has gotten worse compared to their most recent ancestor. Overall these graphs indicate that our general intuition of which individuals survive is correct: there are very few individuals with an average rank that is currently over 23. In particular, we can see that the only surviving individuals with historical average rank over 25, which appear on the far right of the insertion graph, survive due to an insertion mutation which significantly improves

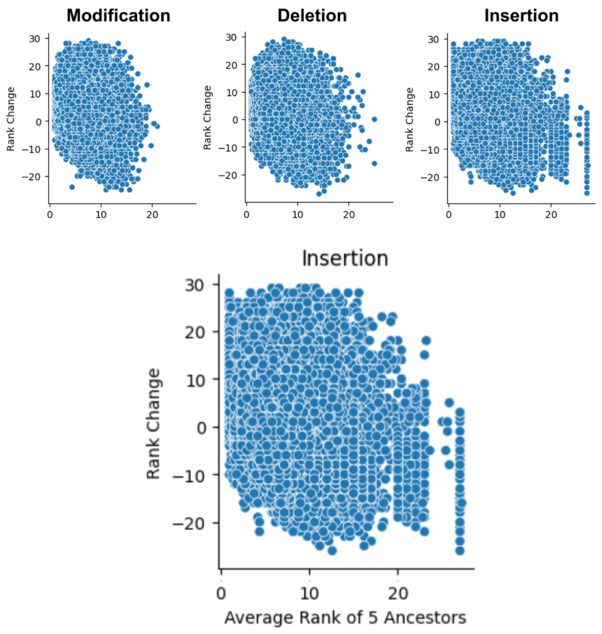

Figure 4: The historic average rank (mean of 5 immediate ancestors ranks) against current rank change (between current individual and immediate ancestor) grouped by type of mutation applied. Mutation types: Modification (top left), Deletion (top middle) and Insertion (top right and main).

their fitness. This is particularly interesting as insertion only produces neutral or better changes to these individuals. We also see, however, that insertions are responsible for taking well-ranked individuals through very negative rank changes: the only way for individuals to appear in the top-right is if they are currently well ranked, but in their last 5 generations have had an insertions which caused their rank to become very poor, before a further mutation brought their rank back to a very good position.

By tracking average historical rank, and limiting those with high average historical rank to insert mutations, we may therefore help to create diversity in the population by increasing the survival chance for these individuals. It may also improve the fitness of the population as a whole by improving lower fitness individuals faster through insertion rather than through other mutations.

4.2 Crossover

We next examine the effects of crossover as evidenced by our analysis – i.e., the process by which a fragment of code is shared between two different individuals in an evolutionary algorithm. Crossover can take many different shapes and forms dependent on the representation of the genotype and the purpose of the algorithm. There are two main factors of difference between types of crossover: directionality and fragment selection.

Directionality refers to whether the crossover process changes one or both individuals. In a bidirectional transfer a fragment is selected from both individuals and swapped into each others places. In single directional transfers a location is chose in the recipient code and a fragment is chosen from the donor code. The fragment from the donor code is copied and inserted into the recipient code at the chosen location.

Fragment selection is either single- or dual-point. A single-point fragment selection is normally used with bidirectional crossover. A single point is selected in both individuals and the code before or after those points are exchanged. Dual-point selection selects the start and end of a fragment. This could be controlled for a particular length of fragment or particular positioning in an area of the code. This can then be used with any directionality for crossover.

The general purpose of crossover is to insert new code into an individual which is known to be valid and with the potential for greater complexity than most synthesised code. In this work we used single directional transfer of code fragments, with dual point selection having a length of 1 (intended to acquire a single line of code, with the exception of control loops which if selected include the contents of the loop).

Possible parameters for control of the crossover in this system that could be changed based on the results of these analysis include: how much of the population we perform crossover on, how we select which member of the population we perform crossover on, how we select our donor code, or how many tokens we select for crossover. Each of these factors could be changed statically or at continuous points in the run time of the algorithm.

We analyse the effects of crossover by considering the relative ranks of the individuals involved in each crossover, each of which is in the range 1 to 30. Crossover takes place immediately after a new generation is formed, transferring genetic material from a donor individual to a recipient individual. The *recipient rank* is the pre-crossover rank from the previous generation, within its population, of the individual selected for insertion of crossover material. The *donor rank* is the rank, from the previous generation, of the corresponding individual from which the crossover material is sourced. The *resultant rank* is the rank given to the recipient, within its population for the current generation, after post-crossover fitness testing. Finally, the *rank change* is the difference between the recipient rank and the resultant rank.

We analyse two properties of crossover in terms of these rank differences: the effect of donor rank on recipient/resulting rank, and general rank change over time as a result of crossover.

4.2.1 *Effect of Donor and Recipient Rank.*

In order to consider the effect of the rank of the donor code on its recipient, we need to compare that rank with the recipient's resultant rank. We note that rank change per individual is generally limited by the starting rank of that individual; poorly-ranked recipients have a relatively high potential for change as their rank can improve more than well-ranked recipients which have little space for improvement but a high potential for decline. Figure 5 uses a heat map to show the mean rank change for each donor-recipient rank pair. The recipient rank is shown in the x-axis, and the donor rank on the y-axis. The top-left heat map square at x(0) and y(0) therefore indicates the resultant rank of a crossover between a donor individual that had the best rank (0) and a recipient individual that had the best rank (0). Light-coloured squares then represent negative rank changes which result in a poorer individual, while dark-coloured squares represent positive rank changes.

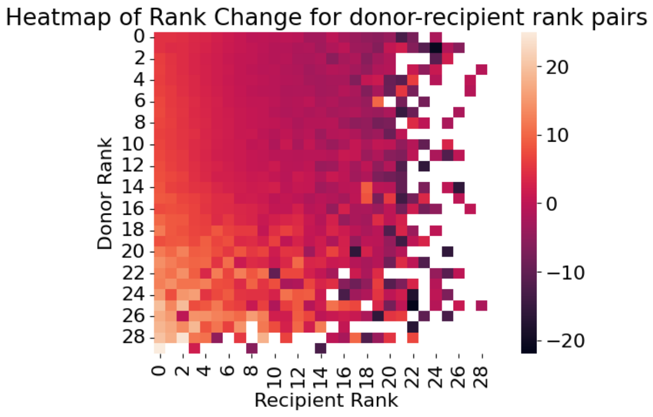

Figure 5: The recipient rank (horizontal) of code before it is changed and donor rank (vertical) of the code from which new material is copied. The heat map shows the average rank change (from recipient to the created code) resulting from the pairing of any two ranks. Pale colours are poor rank change, dark colours are beneficial rank change.

This result shows a clear pattern of crossover efficacy corresponding to the donor individual's rank, but only for better-ranking recipients. For the two vertical lines of heatmap squares intersecting the x-axis at positions 0 and 1, therefore, we see a clear continuous gradient from negative crossover effects originating from poorly-ranked donor individuals through to positive crossover effects originating from well-ranked donor individuals. For worse-ranking recipients, by contrast, crossover generally has a strong positive change regardless of the donor's rank; most of the the right-hand side of the heatmap therefore has a dark colour regardless of the y-axis value of the donor rank. There are also many worse rank recipients for which we have no data, indicated by a lack of heatmap squares on the right side of the graph, as the recipient's resultant rank is limited by the selection process which is biased against selecting worse ranked individuals for the new generation and so they are less likely to undergo crossover.

For better-ranked recipients there is an indication in this data that the improvement potential from crossover is generally reduced for better individuals, and in fact that better individuals may experience a rank decline even when receiving crossover code from good donor individuals. When better recipients receive from poor donors individuals they suffer far more extreme rank changes which are on average very deleterious (a rank degradation of more than 15 places in the population of 30). This suggests an advantage to reducing crossover in general for better-ranked individuals and weighting the selection of donors for them towards better ranked donors.

4.2.2 *Rank Change Over Time.*

To confirm the above effects, and to examine whether there is a temporal effect in crossover characteristics, we split our population into five groups: individuals of rank 1-6, rank 7-12, rank 13-18, rank 19-24, and rank 25-30.

The first two groups, rank 1-6 and rank 7-12, correspond to the better-ranked recipients in our above analysis; their average rank change over time may reveal fine details of the rank change effect.

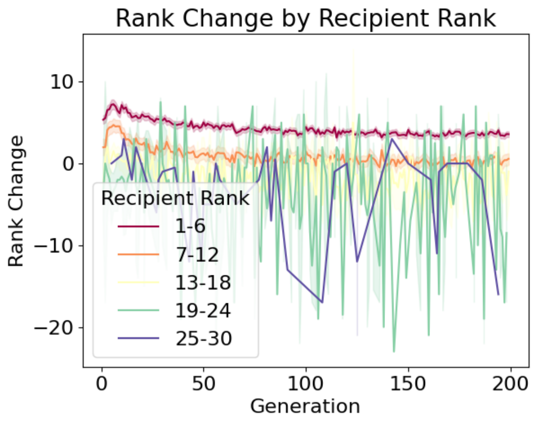

Figure 6: Average rank changes over time grouped based on the rank of the recipient code before crossover

The other groups will show if any recipients undergo changes in their efficacy due to crossover during the optimisation process.

Figure 6 shows the results for each grouping. As would be expected, most improvement occurs on the worse-ranked three groups which have far more room for improvement. The best two rank categories are only made worse by crossover or otherwise remain unchanged (their relative rank changes are always above 0). We stress here that rank change is not the same as fitness change; over the course of these generations fitness scores steadily improve, even if relative rank of individuals in a population does not.

While our better-ranked groups (red and orange in the figure) therefore remain in a near-neutral state as a result of crossover except at the very start of the time line, our worse two rank groups exhibit very different characteristics. The rank 19-24 group (green in the figure) is very erratic in its crossover effects on individual rank changes, but it is also the most common to have larger individual rank improvements due to crossover. We do also see larger improvement in the rank 25-30 group (purple in the figure) but there are far fewer data points in this group due to selection pressure on individuals of this rank. An increased chance of crossover for these poor-rank groups could yield greater improvements on their individuals – while also potentially improving the overall population fitness and diversity by helping to find improvements for poorly-ranked individuals to keep them in the population.

5 EXPERIMENTS

In this section we report on experiments which use new GI processes configured based on our above analysis. This answers the question of whether or not these analysis results can yield improvements in the efficacy of our GI system. We first examine changes to mutation policies and then examine changes to crossover.

Our evaluation metrics examine the *fitness* and *performance* of the best individual after 200 generations. We define fitness as the speed of the best individual when tested on its training data (the set of hash function keys). Performance, by comparison, is the speed of the best individual when tested on *unseen* data drawn from the same distribution as fitness-testing data. Fitness indicates how well we have specialised to training data, while performance indicates how well we have generalised to data of that class.

As well as these evaluation metrics of simple performance against the target problem, we are also interested in the extent to which changes that we make result in expected or predictable outcomes in GI behaviour according to what our analysis results suggested (e.g. in noise reduction, or genetic diversity increase).

Each experiment is repeated 30 times with the average fitness and performance reported. Our results show that our mutation policy changes yield significant improvements in performance of the best individual against unseen data, indicating a better ability to generalise. Our crossover policy changes are less successful, yeilding poorer fitness, but do provide near-equivalent performance for the best individual against unseen data – and notably they greatly reduce the noise level within performance across generations.

5.1 Mutation Changes

We apply two changes to our mutations: the first is to force the use of insertion mutations on any individuals whose 5 most immediate ancestors have an average rank greater than 24. This is referred to as **Lineage Modification**, and is expected to yield more diverse populations which could produce both more noise but also more long-term-effective optimisation by increasing the overall amount of useful genetic material in the system.

The second change, applied separately, is to use a non-uniform probability distribution for the selection of insertion, modification or deletion mutations, using a (0.2, 0.4, 0.4) probability respectively at the start of our GI run which we then gradually adjust over time to finish with a uniform distribution by our terminal generation (200) with increments of (0.001, -0.0005, -0.0005) each generation. We refer to this system as **Generational Modification**, as it is a time-based weighting, and is expected to yield faster optimisation and/or less noise in the population.

When tested, these systems show two distinct results. First, the fitness of the best individual in each generation on training data (the degree to which they specialise) is slightly worse than in our control system which uses uniform random selection throughout (Fig. 7). However, their performance on unseen data (the degree to which they have generalised to the class of data) has greatly improved, yielding individuals that are 20% better than in our control (Fig. 7). The improvement on performance could in both cases come from maintaining higher diversity in the systems encouraging generalisation rather than specialisation to the training data.

Besides these core results, we also see general characteristics which confirm to our expectations of these changes relative to our analysis data. First, we do see higher variation and noise in the **Lineage Modification** system, as expected; this is because more diverse genetic material is kept in the population. Similarly, the **Generational Modification** system does train faster with better ranked individuals focused on incremental changes from modification and deletion than our control, though it does not reach quite the same level of fitness.

These results are encouraging and suggest we can gain meaningful improvement to a GI process based on analysis of its historical behaviour, and that are changes yield predictable characteristics.

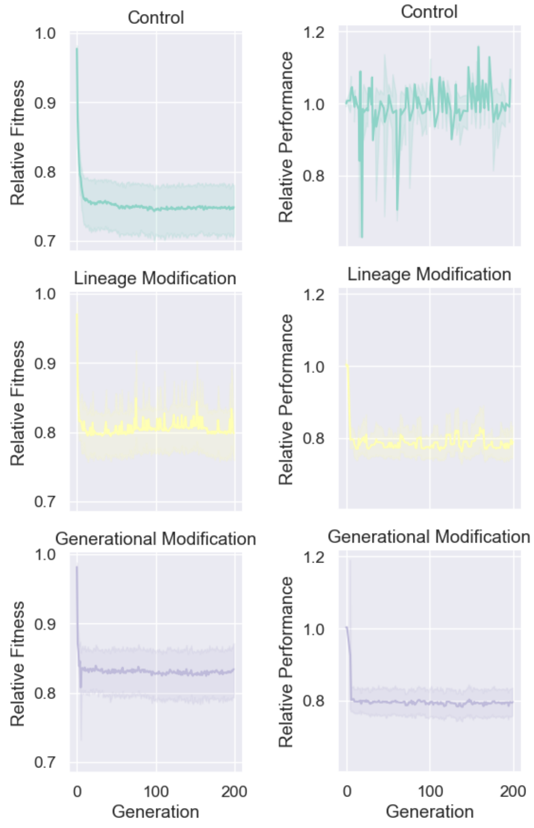

Figure 7: Mean relative (scaled to fitness of original code) fitness (left) and performance (right) of the best individual from 30 runs over 200 generations for the Control, Lineage Modification and Generational Modification systems.

5.2 Crossover Changes

Our crossover procedure was modified by placing weights on the chance that an individual will undergo crossover based on their rank. Instead of all individuals being equal, those in the best 40% (12 best individuals) of a population were given a 10% chance of crossover; those in the middle 20% (6 middle individuals) were given the original 20% chance of crossover; and those in the worst 40% (12 worst individuals) were given a 30% change of crossover. This matches our analysis results that crossover disproportionally benefits poor-ranked individuals and can actively harm better-ranked individuals. These weightings are applied to produce the same average crossover chance across all groups (20%), but distributed different between different rank groups.

If an individual in the best 40% of a population is selected as a crossover recipient we also enforce the selection of the donor code from only the best 40% of individuals. This should encourage incremental improvements in the more optimised individuals while allowing for larger improvements in the less optimised individuals, again following the patterns found in our analysis.

Figure 8 shows the results for fitness (on the left) against training data, and shows that the reduction and restriction in crossover for better ranked individuals has reduced the ability of the system to train as effectively as the control system. This may be a general result, but could also be due to the particular level of reduction and restriction we chose for our particular target problem; further experimentation is needed to fully understand this result, but it may indicate that our current analysis method of crossover lacks sufficient power to yield predictable results.

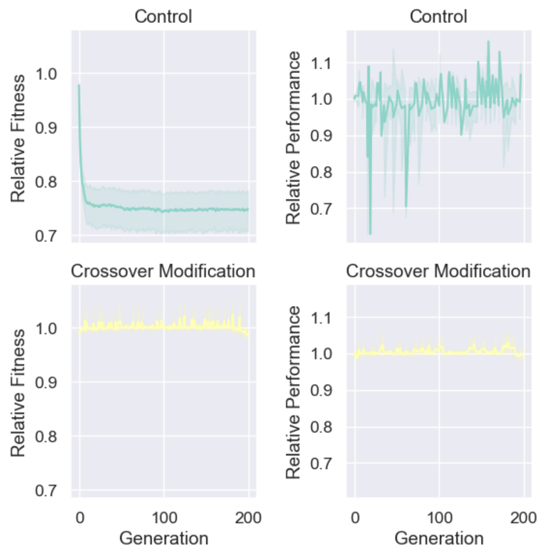

Figure 8: Mean relative (scaled to fitness of original code) fitness (left) and performance (on unseen data, right) of the best individual from each of 30 runs over 200 generations for the Control and Modified Crossover systems.

Despite this, we do see an expected result in our evaluation of the performance of the best individuals against unseen data, shown in Figure 8. Here we see a far less noisy signal across generations and a result which matches the performance of our control.

6 CONCLUSIONS

In this work we have presented an initial exploration of the potential for phylogenetic-inspired analysis on GI characteristics to be used for targeted improvement of parameters on whole-system, temporal, and individual levels.

We find that where the changes have strong enough independent effects we can use these analyses to produce new parameters regimes that can improve the performance of the GI system. Our most positive results relate to changes in mutation regimes, which show improvements in performance and also predictable changes in the characteristics of a GI process; while our changes to crossover regimes do not provide improvement in performance, they do also provide a predictable property of noise reduction.

In future work we will further evaluate our analysis methods, particularly with respect to crossover, and apply our technique to additional target problems to further understand its generality – particular those problems with shared features, as other problems will exist within the same overall fitness landscape.

7 ACKNOWLEDGEMENTS

This work was partly supported by the Leverhulme Trust Research Grant 'The Emergent Data Centre', RPG-2017-166.

REFERENCES

[1] Thomas Bäck and Hans-Paul Schwefel. 1993. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.* 1, 1 (March 1993), 1–23.

[2] Laurence M Cook. 2003. The rise and fall of the Carbonaria form of the peppered moth. *Q. Rev. Biol.* 78, 4 (Dec. 2003), 399–417.

[3] Giuseppe Cuccu and Faustino Gomez. 2011. When Novelty Is Not Enough. In *Applications of Evolutionary Computation*. Springer Berlin Heidelberg, 234–243.

[4] Joseph Felsenstein. 2004. Inferring Phylogenies. https://www.sinauer.com/media/wysiwyg/tocs/InferringPhylogenies.pdf. Accessed: 2022-1-21.

[5] Stephanie Forrest, Thanhvu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (Montreal, Québec, Canada) *(GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 947–954.

[6] Saemundur O Haraldsson, John R Woodward, Alexander E I Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 1513–1520.

[7] Willi Hennig. 1965. Phylogenetic systematics. *Annu. Rev. Entomol.* 10, 1 (Jan. 1965), 97–116.

[8] Víctor R López-López, Leonardo Trujillo, and Pierrick Legrand. 2018. Novelty search for software improvement of a SLAM system. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1598–1605.

[9] M E N Majerus. 1989. Melanic polymorphism in the peppered moth, Biston betularia, and other Lepidoptera. *J. Biol. Educ.* 23, 4 (Dec. 1989), 267–284.

[10] Christoph Neumüller, Stefan Wagner, Gabriel Kronberger, and Michael Affenzeller. 2012. Parameter Meta-optimization of Metaheuristic Optimization Algorithms. In *Computer Aided Systems Theory – EUROCAST 2011*. Springer Berlin Heidelberg, 367–374.

[11] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (June 2018), 415–432.

[12] B. Porter and R. Filho. 2021. A Programming Language for Sound Self-Adaptive Systems. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE Computer Society, Los Alamitos, CA, USA, 145–150. https://doi.org/10.1109/ACSOS52086.2021.00036

[13] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. usenix.org, 333–348.

[14] Anubhav Singh, Nir Lipovetzky, Miquel Ramirez, and Javier Segovia-Aguas. 2021. Approximate Novelty Search. *ICAPS* 31 (May 2021), 349–357.

[15] Marta Smigielska, Aymeric Blot, and Justyna Petke. 2021. Uniform Edit Selection for Genetic Improvement: Empirical Analysis of Mutation Operator Efficacy. In *2021 IEEE/ACM International Workshop on Genetic Improvement (GI)*. ieeexplore.ieee.org, 1–8.

[16] Omar M Villanueva, Leonardo Trujillo, and Daniel E Hernandez. 2020. Novelty search for automatic bug repair. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1021–1028.