Exploring Fitness and Edit Distance of Mutated Python Programs

Saemundur O. Haraldsson[1], John R. Woodward[1], Alexander E.I. Brownlee[1], and David Cairns[1]

University of Stirling, Stirling FK9 4LA, Scotland

Abstract. Genetic Improvement (GI) is the process of using computational search techniques to improve existing software e.g. in terms of execution time, power consumption or correctness. As in most heuristic search algorithms, the search is guided by fitness with GI searching the space of program variants of the original software. The relationship between the program space and fitness is seldom simple and often quite difficult to analyse. This paper makes a preliminary analysis of GI's fitness distance measure on program repair with three small Python programs. Each program undergoes incremental mutations while the change in fitness as measured by proportion of tests passed is monitored.

We conclude that the fitnesses of these programs often does not change with single mutations and we also confirm the inherent discreteness of bug fixing fitness functions. Although our findings cannot be assumed to be general for other software they provide us with interesting directions for further investigation.

Keywords: Search Based Software Engineering, Genetic Improvement, Genetic Programming, Automatic Programming, Software Repair.

1 Introduction

In recent years work has been emerging from the Search Based Software engineering (SBSE) community called Genetic Improvement (GI) [18]. GI is where computational search techniques have been applied to already functioning software for the purpose of improvement. The improvement criteria can be various properties of the existing software such as speed, accuracy or energy efficiency. The most commonly used search method for GI is Genetic Programming (GP) although various other techniques like Genetic Algorithms [3] and Grammatical Evolution [33] have also been applied.

The examples of work that can be categorised as GI are widespread. Most of them, with few exceptions, are similar in the sense that they tackle software that can be considered large, with lines of code (LOC) numbering in the thousands. Nearly one third of the GI literature is dominated by examples of automatic bug fixing while approximately another third is concerned with improving non-functional properties [12, 41]. Of those, execution time is perhaps the property

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-55696-3_2

that is easiest to measure and therefore the most commonly researched non-functional property. Bug fixing and execution time make very different fitness functions. The bug fixing fitness function is a *discrete* integer function, counting the number of test cases passed and many non-functional fitness functions, like the execution time and memory consumption, are *continuous* measurements. The main emphasis to date has been on results, rather than an understanding of the GI search space. There is a need for empirical and theoretical analysis of the GI process. Is it easy or difficult to traverse the search space of programs and what can possibly be done to increase the chance that the programs improves? We begin to answer these questions in the space of smaller programs. Using small programs to begin with allows us to easily analyse the full impact of changes introduced to the code. It also keeps uncertainties introduced by the rest of the program to a minimum.

This paper explores the relationship between fitness and number of accumulated incremental changes while using GI to break programs and gain an understanding of the program repair process. We chose three small Python programs to make a preliminary study on empirical properties of GI's fitness and edit lists. The following questions were addressed:

- Is it feasible to apply GI to fix multiple bugs at a time? Will fixing one bug introduce another?
- If a fix needs multiple edits, will GI be able to find it within reasonable time/number of iterations?
- Can we identify any similarities or patterns in fitness distance relationship that might be worth exploring in more detail on larger sets of programs? I.e. is there some rule there that has not been discovered yet?

Although the experiments with only three programs are limiting for generalization of the answers we find, they help us narrow down the most promising route of research for larger and more resource consuming experimentation. Choosing to operate on Python programs also serves two purposes:

- There are very few examples of GI being applied to Python programs [1] and given that it is a very popular language, there is a large gap in the literature.
- Because of dynamic typing of Python programs, the search space is possibly less restricted than for statically typed languages. Therefore changes to the source code might have more possibilities than a static typed language.

The remainder of the paper is structured as follows. Section 2 gives an overview of related work. Section 3 describes the implementation of GI that was used for this paper while Section 4 gives more details on the configuration and data collection. Section 5 summarizes the results with discussion followed by the conclusions in Section 6.

2 Related work

A search through publications in software engineering and computer science suggests that approximately one third of GI related material is on bug fixing.

GenProg [25, 26, 38] is one of the better known automatic bug fixing frameworks and uses GP to evolve patches for a fraction of the price it would cost manually [24]. GenProg itself is derived from earlier work by Weimar *et al.* [11,37,39]. Smith *et al.* is an example of a more recent use of this framework [34]. Although GenProg is perhaps the most commonly known framework, there is also a large body of GI literature dedicated to automatic bug fixing [1, 2, 4, 28, 31] that use alternative approaches.

Automatic program repair can be considered an improvement of a functional property, like improving the quality of hash code functions in Hadoop [15] or grafting new features to existing software [14, 27]. However, optimizing attributes like execution time, memory consumption and power consumption is generally considered an improvement of a non-functional property which spans another big part of the GI literature. Of those attributes, execution time seems to be very popular, with Langdon's work on the 50k line DNA sequencing tool Bowtie [18, 20] possibly the best known. Langdon has also reported 100 fold speed-up of another DNA sequencing tool BarraCUDA [17, 19, 21–23] and the GI improvements have now been included in the official release. Langdon's GI implementation has furthermore been used by others for specializing and optimizing the execution time of MiniSAT [30], a boolean satisfiability solver and for optimizing power consumption of that same solver [5, 6]. Many others have applied or suggested GI for improving non-functional properties such as execution time [9,10,35,42], energy consumption [7,8,13,40,43] and memory usage [32,44].

The literature of GI counts around 100 papers and there is not much work on empirical analysis of the search landscapes of GI like there is for GP [36], nor is there much on Python programs which is the topic of this paper.

3 Our Implementation of Genetic Improvement

There are multiple ways to implement GI. We have implemented a variation of the work of Langdon *et al.* [14,18]. Like their GI, ours operates on the source code with no need to convert the program to a different representation like abstract syntax trees (AST) [1]. Therefore our approach is directly transferable between programming languages with minimal configuration. The source code is read as a text file and stored in a data structure (x) of program lines. For each line we recorded the raw text as it appears in the source file, along with information on: indentation[1]; line type; whether the line can be altered; and any variables or built-in operators that the line includes.

To manipulate the source code we evolve *edit lists* (Figure 1). Each edit consists of: the operation of *Replace*; the source code snippet before and after the edit; and the location of where to apply this edit (line and character number).

The code snippets can be whole lines from the source code, or one of the various single operators or numerical constants listed in Table 1. For other programming languages this table would vary slightly, like the incremental operator ++ which exist in C and Java but not in Python.

[1] Code blocks are defined by indentation in Python and not by {} as in JAVA/C

Table 1: Sets of single operators available to the GI. One member of a given set can be changed to another member of the same set.

		Operations
S1	Numerical constants	Can increment by ± 1
S2	Arithmetic operators	$+, -, *, /, //, \%, **$
S3	Arithmetic assignments	$+=, -=, *=, /=,$
S4	Relational operators	$<, >, <=, >=, ==, !=,$ $is, is\ not, not$
S5	Logical operators	$and,\ or$
S6	Logical constants	$True,\ False$

An individual genome consists of a list of edit operations sampled from the set shown in Table 1 and is applied in sequence from the first edit to the last item in the list.

3.1 Fitness function

In our experiments the fitness function counts the number of test cases for which the program passes. It is inherently discrete and possibly provides no obvious gradient for the search to follow, which poses a difficulty for many search methods, although evolutionary algorithms have been applied successfully [30]. We want to analyse this discrete nature and in addition, explore ways to report on GI problem difficulty. The relationship between the fitness landscape and genotype/phenotype is far from trivial. However some work has been done on introducing a partial evaluation of programs and hence some kind of guidance to the search process in cases where the fitness landscape was largely flat [16].

3.2 Search algorithm

Generally, the GI's search algorithm is guided by a fitness function, applying selection pressure towards better programs. GI methods that evolve edit lists also have to produce a new generation of edit lists from a previous generation. Our implementation uses the base type of mutation illustrated in Figure 1 that is applied to parents to produce offspring: Append randomly generated edits to the parent (*Grow*).

For our experiments, analysing the relationship between edit list size and fitness, we start from an assumed correct implementation of the program and apply a single edit. Then incrementing by using *Grow* with single edit and without any pressure to search, resetting the edit list to a single edit when all conditions are met; maximum size of edit list and minimum fitness.

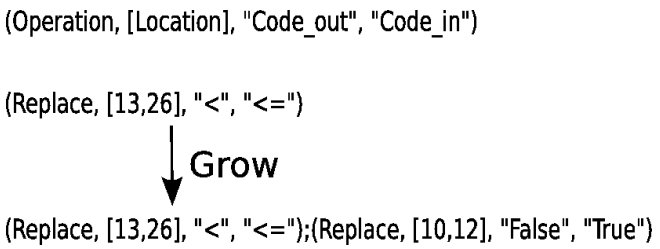


Fig. 1: An example of an edit list and how it can evolve with *Grow*.

4 Experimental setup

Each program source code is subjected to experiments to assess fitness distance which is the change in fitness given a particular number of mutations. The proportion of passed test cases is the chosen measurement for the fitness function.

Procedure 1 A single experimental run

1: $F \leftarrow$ empty array {list of fitnesses}
2: $x \leftarrow [random \quad edit]$ {edit list}
3: **for** $i = 0$ until $i \leq 50$ **do**
4: append $f(x)$ to F
5: **if** $|x| \geq 20$ and $f(x) = 0$ **then**
6: break
7: **end if**
8: append $[random \quad edit]$ to x
9: **end for**

The experiment in procedure 1 is repeated 100 times by initiating an edit list x with a single randomly chosen edit and then appending to the list incrementally. Apart from recording the fitness $f(x)$ for each added edit in every experiment, three variables are recorded for each run. These are the size of the edit list ($|x|$) when the fitness ($0 \leq f(x) \leq 1$) satisfies the following:

Δ decreases for the first time:

$$\text{when} \quad f(x_i) < f(x_0) \quad \text{and} \quad f(x_0) = f(x_j) \qquad \forall j \in (1, ..., i-1) \subset \mathcal{Z}$$

Ω reaches zero:

$$\text{when} \quad f(x_i) = 0 \qquad \text{and} \quad f(x_j) > 0 \qquad \forall j \in (0, ..., i-1) \subset \mathcal{Z}$$

Ψ starts to increase again:

$$\text{when} \quad f(x_i) > f(x_{i-1}) \quad \text{and} \quad f(x_j) \leq f(x_{j-1}) \qquad \forall j \in (1, ..., i-1) \subset \mathcal{Z}$$

This provides us with data to empirically evaluate the nature of the relationship between fitness and the size of the edit list for the three programs described in 4.1. Each program is accompanied by a test suite of different sizes so the fitness is normalized to represent the fraction of test cases passed.

4.1 Description of the programs targeted by GI

The three programs[2] summarised in Table 2 were selected for the experiment. They are all comprised of 3 or fewer functions. They are not a complete Python module but are either part of a module, like **P2** or a standalone function like **P1** and **P3**. However they can be integrated into any Python module and have well defined input, output types.

P1 is a simple text input calculator that reads text from left to right, parses a single character at a time into operator, digits bins and calculates a result using a reverse Polish notation. It is a beginners programming exercise and the only program of the three that is not a part of publicly available software. It branches out for each of the four basic arithmetic operations; addition, subtraction, multiplication and division as well as a special branch for parentheses.

P2 is an initialization function for the K-means clustering method and is a part of the scikit-learn Python toolbox [29]. It determines the initial k centres for the algorithm. **P2** does this with a random factor that can be seeded for consistency purposes.

P3 is a string manipulation function that reads through a text replacing HTML tags with latex equivalent commands. It is a part of a larger software system, *Janus Manager* that is in commercial use by a vocational rehabilitation centre in Iceland.

Table 2 shows basic info about the programs, **P1**, **P2** and **P3**. The numbers in the second column are the number of lines of code and the number of lines that can be changed, i.e. excluding definitions, comments that do not include executable code and functions out of scope. The third and fourth columns are the count of mutable points, the number of instances in the source that fit into any of the sets defined in Table 1 and the count for each set. The fifth column describes the input and output of each program and the last column is a short description of its purpose.

P2 and **P3** are accompanied by test modules which are used to evaluate fitness. However **P2**'s test suite, comprising of approximately 60 test cases, was expanded to 500 by sampling from an estimated distribution of inputs from the original tests and using the **P2** as an oracle. **P3** comes with 124 test cases based on HTML input from users and their verified output, so expanding that test suite is not feasible. The test suite for **P1** is 600 cases made by combining two sets; $A = \{0, 1, 2, 3, 4\}$ and $B = \{+, -, *, /\}$ into:

a) All possible combinations of a single operator from B with two digits from A, an example would be $2 + 2$.

[2] The programs and their test suites are available on `https://github.com/saemundo/Exploring-Fitness-and-Edit-Distance-of-Mutated-Python-Programs.git`

Table 2: Information about the programs that were used in the experiments

Program	LOC (mLOC*)	Mutable points	Types of mutable points	Input ⇓ Output	Description
P1	99 (98)	147	S1: 19 S2: 37 S3: 50 S4-6:41	String ⇓ Float	Simple text input calculator
P2	177 (75)	106	S1: 11 S2: 14 S3: 57 S4-6:24	Cluster data** ⇓ Matrix	Initiation of K-means cluster centers
P3	103 (63)	59	S1: 26 S2: 3 S3: 29 S4-6:1	HTML ⇓ Latex	Html to Latex conversion tool

*Changeble lines of code.
**Data points, number of clusters, initialization method and 3 optional arguments.

b) All combinations of $(Xo_1Y)o_2Z$ where $\{X, Y, Z\} \subseteq A$, $o_1 \in \{+, -\} \subseteq B$ and $o_2 \in \{*, /\} \subseteq B$. An example would be $(4 + 2) * 2$.

P1's test suite was verified with the Python built in *eval* function.

5 Results

We will look at the programs from three different angles; size of edit list versus a specific change in fitness, average fitness as a function of edit list size and, unique and discrete steps in fitness. When statistically comparing the mean of two variables we use Welch's t-test for two independent samples with unequal variance. For testing the likelihood of two variables coming from the same distribution, we compute a two sample Kolmogorov-Smirnoff statistic.

5.1 Change in Fitness

Table 3 lists the basic descriptive statistics for the edit list size ($|x|$) for the three different changes in fitness (see Section 4) and the total number of fitness evaluations for each program. That number varies between programs due to the termination conditions described in Section 4. Firstly we measure the edit distance required for the fitness to decrease for the first time (i.e. $f(x) < 1$).

For **P1** Δ occurs on average when the edit list is 9 edits long, although there is a lot of variations as shown by the standard deviation and the range. We find this a quite surprising result. It should also be noted that in 1 run out of the 100 repeated experiments the fitness did not drop at all, reaching the maximum size of 50 edits. There is a highly significant difference ($p < 0.001$) between **P1**

Table 3: Statistics for the variables defined in section 4, edit list size $|x|$ when changes in fitness $f(x)$ are detected and total number of fitness evaluations during the experiments.

	Variable	Mean (std)	(Min, Max)	Number of occurances	Evaluations
P1	Δ	8.91 (9.83)	(1, 50)	99	2213
	Ω	12.68 (10.55)	(1, 50)	97	
	Ψ	12.0 (7.53)	(3, 25)	12	
P2	Δ	2.56 (3.20)	(1, 19)	100	2267
	Ω	10.28 (8.33)	(1, 42)	100	
	Ψ	7.64 (5.42)	(2, 26)	34	
P3	Δ	2.69 (3.30)	(1, 19)	100	1980
	Ω	3.92 (3.74)	(1, 19)	100	
	Ψ	4.76 (3.44)	(2, 16)	17	

and **P2** for the mean Δ. We can also reject the null hypothesis that the samples come from the same distribution ($p < 0.001$) as is evident when comparing Figures 2a and 2b. However statistically we cannot rule out the possibility that the distribution ($p = 0.99$) and mean ($p = 0.78$) are the same for those measures when comparing **P2** and **P3**. Notice that the bar plots in Figures 2b and 2c are very similar.

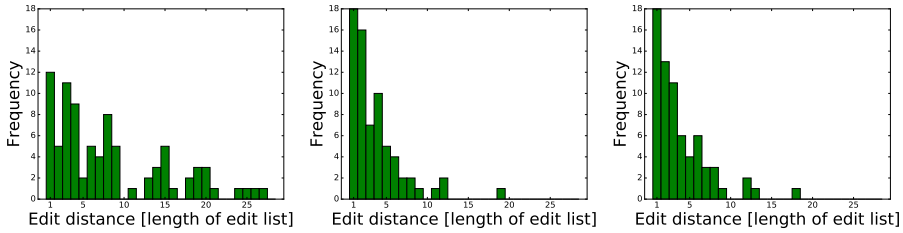
P1 and **P2** are closer together when comparing the number of edits it takes to reach zero fitness and until the fitness might increase again. Testing for the same distribution gives $p = 0.08$ and $p = 0.10$ for Ω and Ψ respectively and we also cannot reject the hypothesis that the means are the same (0.08 and 0.09). Figures 2d and 2e corroborate the observation about zero fitness. However looking at Figures 2g and 2h we are less sure about the number of edits it takes to increase the fitness again and might infer that we do not have enough data to be confident the test results are accurate.

P3 has very different means and distributions than **P1** on all measured variables ($p < 0.01$) which is validated on looking at Figures 2g–2i. The lack of data for Ψ is further verified by the outcome of tests comparing increased fitness between **P3** and **P2**: Rejecting that they have same mean ($p = 0.025$) but failing to reject that they come from the same distribution ($p = 0.09$).

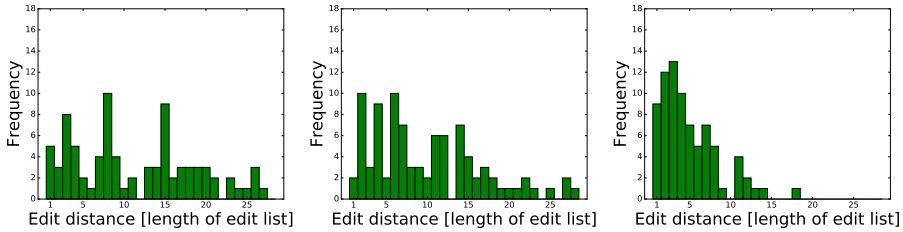
These results indicate that: **P1** is unaffected by many edits and **P2** and **P3** are easily broken and fixed even though they are very different.

5.2 Average Fitness with Respect to Edit List Size

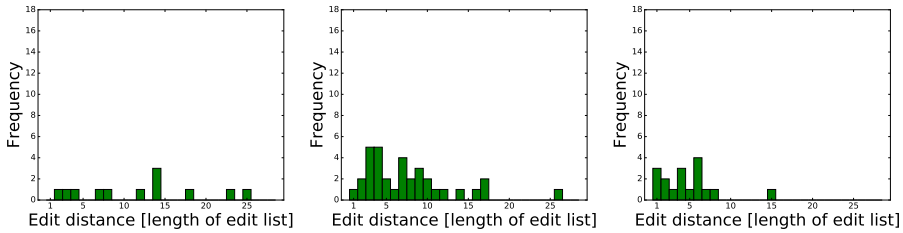
Repeating the experiments 100 times provided us with enough datapoints to construct fitness distance graphs and approximate the distribution of fitness for each increment in edit list size. Looking at the boxplots in Figures 3a–3c we see that overall, the distributions are quite different. **P1**'s first three increments (Figure 3a) have very narrow distributions close to $f(x) = 100\%$ and then the distributions widen considerably until increment 15 where they start narrowing towards the bottom. For **P2** it is a smoother transition (Figure 3b) from top



(a) **P1**. (b) **P2**. (c) **P3**.



(d) **P1**. (e) **P2**. (f) **P3**



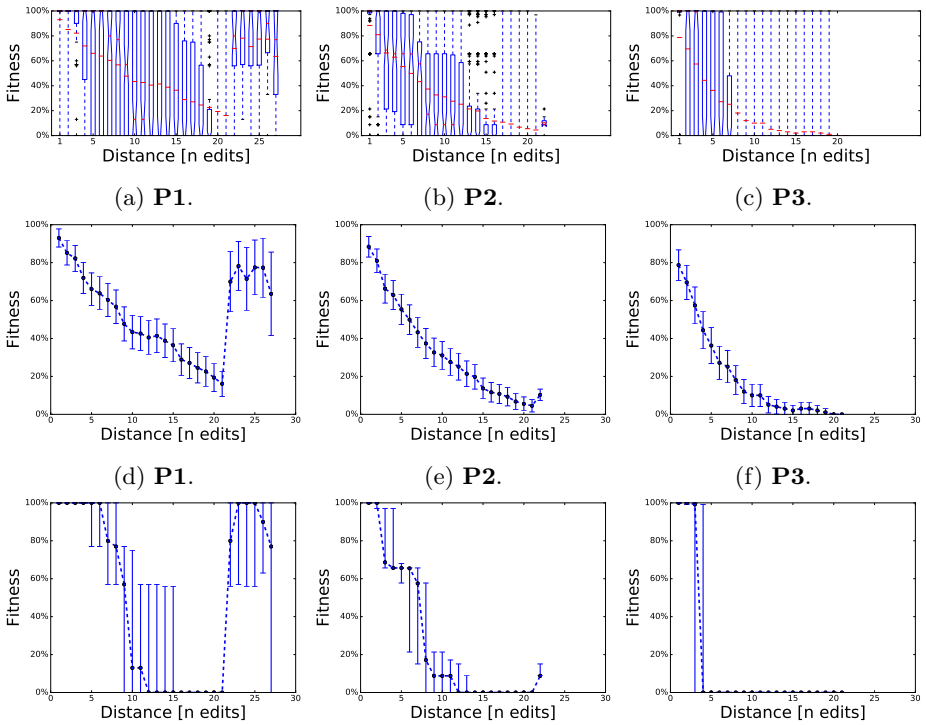
(g) **P1**. (h) **P2**. (i) **P3**.

Fig. 2: Distributions of the three different measurements in Table 3 during the experiments for each program. Top three are Δ, the middle are Ω and the bottom three are Ψ

to bottom, maintaining a similar rate of descent for the mean, maximum and minimum throughout. Then **P3** stands out completely with seemingly only two distributions; covering the entire range and nearly collapsed on either extreme (Figure 3c).

Having a closer look at how the mean fitness changes in Figures 3d–3f we see that these programs are as dissimilar as initially assumed. While both **P2** (Figure 3e) and **P3** (Figure 3f) both follow curves that are concave upwards, **P3** seems to follow a curvature of higher magnitude. Now it is **P1** that is the outlier (Figure 3d) following a noisy line with a negative slope until the edit list reaches size 22 when it jumps up to $f(x) = 80\%$ again. **P2** shows signs of starting to recover in increment 22 as well but on a much slower rate than **P1** and **P3** shows no such signs at all.

The plots for medians in Figures 3g–3i paint a completely different story, displaying no hint of smoothness to the transition from one increment to another.



(a) **P1**. (b) **P2**. (c) **P3**.

(d) **P1**. (e) **P2**. (f) **P3**.

(g) **P1**. (h) **P2**. (i) **P3**.

Fig. 3: Fitness with respect to edit list size for each increment. The top three are boxplots, the three in the middle are the mean fitness with 95% error and at the bottom are the median fitness with 95% error.

However there are obvious steps that highlight the discreteness of each program's fitness function. We see in Figure 3h that **P2** has the most number of steps, while **P1** comes second (Figure 3g) and **P3** last (Figure 3g).

5.3 Discrete steps in fitness

Following the observation of the different steps for each program's median fitness seen in Figures 3g–3i, we counted the unique number of fitness evaluations throughout the entire experiment. As previously inferred, **P2** has by far the largest number of discrete steps, with 46 in total as seen in Figure 4. **P2** had its fitness evaluated 2067 times, as seen in Table 3, and Figure 4 shows how often the fitness changed from one value to another by adding a single edit to the list. The count matrix is very sparse as can be seen by the white squares that denote zero counts, for example the fitness never went from 0.655 to zero with one edit. The diagonal line of shaded boxes from $(0, 0)$ to $(1, 1)$ indicates that the majority of single edits had little to no effect on the fitness, especially when

the fitness is already zero. However there is an abundance of blue squares at the bottom which tells us that a single edit can often lead to complete failure of the program.

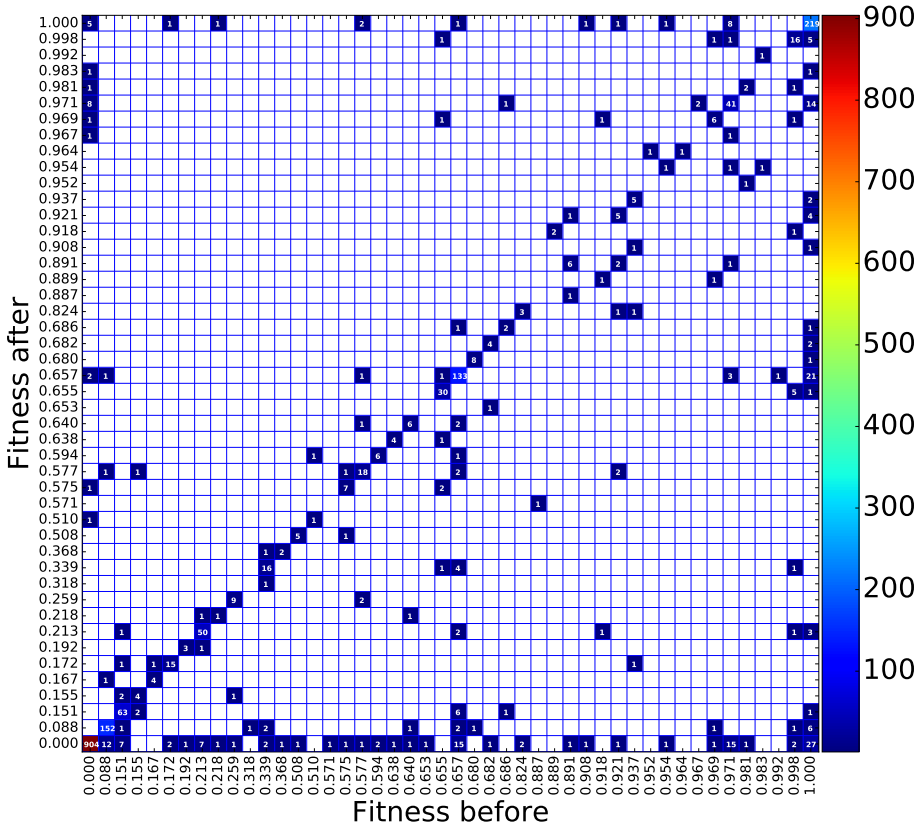


Fig. 4: Frequency chart of fitness changes after a single edit is appended to the edit list for **P2**. Each square is a count of how often the fitness changed from fitness before to fitness after.

P1 has the second most number of discrete steps, 15 as seen in Figure 5, the total number of counts is 2213 (Table 3). This chart looks like a scaled version of **P2**'s Figure 4, displaying the same dominant diagonal line as well as the number of single edits that make the program pass 0% of test cases. We also see here that there are a lot of single edits that decrease the fitness from 1, shown by the large number of non-white squares in the right most column. This is to be expected since every experimental run starts with a correct program, so it visits that state at least once each time while it is not guaranteed to visit other specific steps.

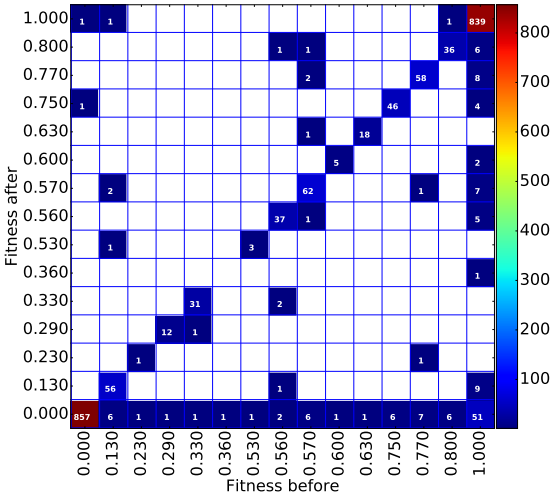


Fig. 5: Frequency chart of fitness changes after a single edit is appended to the edit list for **P1**. Each square is a count of how often the fitness changed from fitness before to fitness after.

Figure 6 is the least sparse of the three, only 5 fitness steps in total. What is surprising however, is that even though there were 1980 fitness evaluations there are still some zero counts. As with **P1** and **P2**, there are more of them above the diagonal line than below, meaning that adding an edit is more likely to decrease fitness than to increase. The highest counts are also on the diagonal line, so the same behaviour can be observed: most likely a single edit will leave the fitness unchanged.

6 Conclusions

The aim of this preliminary exploration of three Python programs' fitness distance is to provide a greater understanding of the search process encountered by GI.

We can conclude that for these three programs and the assumption that the landscape has no inaccessible areas to our GI implementation:

- It is feasible to apply GI to fix multiple bugs simultaneously. Starting from any variation of these programs and applying search pressure can result in a fix.
- If a fix needs multiple edits, GI will be able to find it within reasonable time. If it takes on average 10 edits to completely break these programs (i.e. zero test cases passed), the reverse is likely to be true.
- We have identified a similarity in fitness distance relationship that is worth exploring in more detail on larger set of programs.

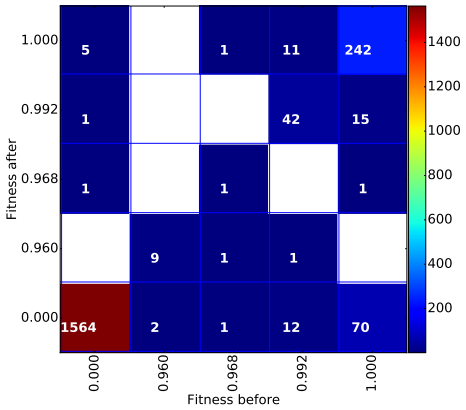


Fig. 6: Frequency chart of fitness changes after a single edit is appended to the edit list for **P3**. Each square is a count of how often the fitness changed from fitness before to fitness after.

Although we cannot conclude general rules or patterns from these preliminary experiments, a common denominator for our programs is that a single edit will likely have no impact on fitness. However, if a change does occur then it can be expected that the change will be large.

It would be interesting to explore the cause of the difference in sparsity of the frequency charts in Figures 4–6. The programs and the test suites were quite different in nature which might explain the stark contrast in discreteness of our programs. While **P1** and **P3** had a single input argument, **P2** had 6, and the test suites reflected this difference. **P2**'s test suite could be divided into multiple categories, testing various aspects and combinations of input arguments. **P1**'s test suite could also be categorised but only in 4-6 groups and it would be a stretch to try and group **P3**'s test suite.

Our next task is to apply the same analysis that we have done here to a larger set of programs with the goal of helping us to form more general and applicable rules for GI fitness distance. With larger sets we can also analyse the fitness distance correlation of GI with variety of edit list mutation operators, such as removing edits or changing individual edits.

7 Acknowledgements

The work presented in this paper is part of the DAASE project which is funded by the EPSRC. The authors would like to thank Janus Rehabilitation Centre for allowing the use of their software in the experiments and consequently making the relevant part of the source code available for others to use in their experiments.

References

1. T. Ackling, B. Alexander, and I. Grunert. Evolving Patches for Software Repair. In *GECCO'11, 13th annual conference on Genetic and evolutionary computation*, pages 1427–1434, Dublin, Ireland, jul 2011. ACM.
2. A. Arcuri. On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 1003–1006, New York, NY, USA, 2008. ACM.
3. A. Arcuri and X. Yao. A Novel Co-Evolutionary Approach to Automatic Software Bug Fixing. In *2008 IEEE World Congress on Computational Intelligence*, pages 162–168. IEEE Computational Intelligence Society, 2008.
4. J. S. Bradbury and K. Jalbert. Automatic Repair of Concurrency Bugs. *Proceedings of the 2nd International Symposium on Search Based Software Engineering*, page 2, 2010.
5. B. R. Bruce. Energy Optimisation via Genetic Improvement A SBSE technique for a new era in Software Development. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 819–820, Madrid, Spain, jul 2015. ACM.
6. B. R. Bruce, J. Petke, and M. Harman. Reducing Energy Consumption Using Genetic Improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, Madrid, Spain, jul 2015. ACM.
7. N. Burles, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, J. Swan, and N. Veerapen. *Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava*, pages 255–261. Springer International Publishing, Cham, 2015.
8. N. Burles, J. Swan, A. E. I. Brownlee, Z. A. Kocsis, and N. Veerapen. Embedded Dynamic Improvement. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion -15, pages 831–832, Madrid, Spain, jul 2015. ACM.
9. B. Cody-Kenny and S. Barrett. The Emergence of Useful Bias in Self-focusing Genetic Programming for Software Optimisation. In G. Ruhe and Y. Zhang, editors, *Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 306–311, Leningrad, aug 2013. Springer.
10. B. Cody-kenny, E. Galván-lópez, and S. Barrett. locoGP : Improving Performance by Genetic Programming Java Source Code. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 811–818, Madrid, Spain, jul 2015. ACM.
11. S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. *Genetic And Evolutionary Computation Conference*, pages 947–954, 2009.
12. S. O. Haraldsson and J. R. Woodward. Automated Design of Algorithms and Genetic Improvement : Contrast and Commonalities. In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1373–1380, Vancouver, Canada, jul 2014. ACM.
13. S. O. Haraldsson and J. R. Woodward. Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate. In W. B. Langdon, J. Petke, and D. R. White, editors, *Proceedings of the 2015 Conference Companion on Genetic and Evolutionary Computation Companion*, pages 831–832, Madrid, 2015. ACM.

14. M. Harman, Y. Jia, and W. B. Langdon. Babel Pidgin : SBSE Can Grow and Graft Entirely New Functionality into a Real World System. In C. Goues and S. Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 247–252, Fortaleza, Brazil, aug 2014. Springer International Publishing.
15. Z. A. Kocsis, G. Neumann, J. Swan, M. G. Epitropakis, A. E. I. Brownlee, S. O. Haraldsson, and E. Bowles. Repairing and Optimizing Hadoop hashCode Implementations. In C. Le Goues and S. Yoo, editors, *6th International Symposium, SSBSE 2014*, volume 8636 of *Lecture Notes in Computer Science*, pages 259–264, Fortaleza, Brazil, aug 2014. Springer Berlin Heidelberg.
16. K. Krawiec and J. Swan. Pattern-Guided Genetic Programming. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 949–956, Amsterdam, The Netherlands, 2013. ACM.
17. W. Langdon and M. Harman. Genetically improved CUDA kernels for Stereo-Camera. Technical report, UCL Department of Computer Science, London, UK, 2014.
18. W. B. Langdon. Genetic Improvement of Programs. *18th International Conference on Soft Computing, MENDEL 2012*, 2012.
19. W. B. Langdon. Improved CUDA 3D Medical Image Registration. In *UK Many-Core Developer Conference 2014 - UKMAC 2014*, page 2014, dec 2014.
20. W. B. Langdon. Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData mining*, 8(1):1, 2015.
21. W. B. Langdon and M. Harman. Genetically Improved CUDA C++ Software. In M. Nicolau, K. Krawiec, and M. Heywood, editors, *Proceedings of the 17th European Conference on Genetic Programming, EuroGP 2014*, Lecture Notes in Computer Science, pages 1–12, Granada, Spain, 2014. Springer Berlin Heidelberg.
22. W. B. Langdon and M. Harman. Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 805–810, Madrid, Spain, jul 2015. ACM.
23. W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1063–1070, Madrid, Spain, jul 2015. ACM.
24. C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Swiss, jun 2012. IEEE.
25. C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.
26. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.
27. A. Marginean, E. T. Barr, M. Harman, and Y. Jia. Automated Transplantation of Call Graph and Layout Features into Kate. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 262–268, Bergamo, Italy, aug 2015. Springer International Publishing.

28. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. *Proceedings - International Conference on Software Engineering*, pages 772–781, 2013.
29. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
30. J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. *17th European Conference on Genetic Programming*, 8599:137–149, 2014.
31. Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In T. Xie, editor, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 24–36, Baltimore, MD, USA, 2015. ACM.
32. J. L. Risco-Martín, J. M. Colmenar, J. I. Hidalgo, J. Lanchares, and J. Díaz. A methodology to automatically optimize dynamic memory managers applying grammatical evolution. *Journal of Systems and Software*, 91:109–123, 2014.
33. C. Ryan, J. J. Collins, and M. O'Neill. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *EuroGP*, pages 83–96, 1998.
34. E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the Cure Worse Than the Disease ? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, Bergamo, Italy, 2015. ACM.
35. J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report CSM-195, Department of Computing Science and Mathematics University of Stirling, Stirling, UK, 2014.
36. M. Tomassini, L. Vanneschi, P. Collard, and M. Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 13(2):213–239, 2005.
37. W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109, 2010.
38. W. Weimer, Z. P. Fry, and S. Forrest. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In E. Denney, T. Bultan, and A. Zeller, editors, *28th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366, Palo Alto, USA, nov 2013.
39. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Vancouver, Canada, 2009. IEEE.
40. D. R. White. Genetic Programming for Low-Resource Systems. (December), 2009.
41. D. R. White. An Unsystematic Review of Genetic Improvement. In *45th CREST Open Workshop on Genetic Improvement*, London, 2016.
42. D. R. White, A. Arcuri, and J. A. Clark. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, aug 2011.
43. D. R. White, J. Clark, J. Jacob, and S. M. Poulding. Searching for resource-efficient programs. *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, (1):1775, 2008.
44. F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep Parameter Optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1375–1382, Madrid, Spain, jul 2015. ACM.