Genetic Improvement of Runtime and its Fitness Landscape in a Bioinformatics Application

Saemundur O. Haraldsson*
University of Stirling
Stirling, United Kingdom FK9 4LA
soh@cs.stir.ac.uk

John R. Woodward
University of Stirling
Stirling, United Kingdom FK9 4LA
jrw@cs.stir.ac.uk

Alexander E.I. Brownlee
University of Stirling
Stirling, United Kingdom FK9 4LA
sbr@cs.stir.ac.uk

Albert V. Smith
The Icelandic Heart Association
Kopavogur, Iceland
albert@hjarta.is

Vilmundur Gudnason
The Icelandic Heart Association
Kopavogur, Iceland
v.gudnason@hjarta.is

ABSTRACT

We present a Genetic Improvement (GI) experiment on ProbAbel, a piece of bioinformatics software for Genome Wide Association (GWA) studies. The GI framework used here has previously been successfully used on Python programs and can, with minimal adaptation, be used on source code written in other languages. We achieve improvements in execution time without the loss of accuracy in output while also exploring the vast fitness landscape that the GI framework has to search. The runtime improvements achieved on smaller data set scale up for larger data sets. Our findings are that for ProbAbel, the GI's execution time landscape is noisy but flat. We also confirm that human written code is robust with respect to small edits to the source code.

CCS CONCEPTS

•**Software and its engineering** → **Genetic programming;** *Software performance; Search-based software engineering;*

KEYWORDS

Genetic Improvement, Execution Time, Landscape, Bioinformatics

ACM Reference format:
Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee, Albert V. Smith, and Vilmundur Gudnason. 2017. Genetic Improvement of Runtime and its Fitness Landscape in a Bioinformatics Application. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082526

1 INTRODUCTION

In recent years GI has been gaining attention [5, 20]. With every GI publication we are presented with successful applications of search techniques on various programs for functional or non-functional improvements. Non-functional properties have been of particular interest [30], mainly due to the popularity in mobile computations and the need to save resources [2, 3, 6].

Execution time has been a popular GI target, specifically for computationally expensive programs for bioinformatics. Target software have included Bowtie 2 [12] which is used to align genome sequences and the sequence mapping software BarraCUDA [14]. The traditional programs targeted by GI are relatively large (>10K lines of code), with a few exceptions [21, 29].

In our previous work we presented a GI framework that targeted Python programs. We successfully integrated it into a live system [7, 9] and gave examples of the fitness landscape for three Python programs [8]. The landscape with "number of test cases passed" as fitness was shown to be largely flat and often dropping from passing all tests to zero with a single edit. In this paper we analyse our GI's capabilities to improve the execution time of ProbAbel [1], a bioinformatics program written in C[1]. We briefly analyse the landscape of the non-functional property improvement which is measured with a continuous variable from $\mathbb{R}$. Out intentions are to informally compare it with the fitness function landscape of bug fixing which is represented with a discreet variable from $\mathbb{Z}$.

GI work on landscape analysis in general is sparse and even more so when considering non-functional properties. Specifically, we want to see if a GI framework that has initially been applied to Python code can also operate on C code and answer two questions:

(1) Can GI, within reasonable time, find improvements to ProbAbel that decrease its execution time?
(2) What does the landscape for the execution time improvements look like?

The term: "reasonable time" as stated in question 1, is subjective and therefore we have to define it in this context. Let us start by imagining a program that executes in X time units. The GI's improvements decrease the execution to X' time units and it took the GI, Y time units to find them. If said program is only supposed to be used once, then the overall gain would be $\Delta = X - X' + Y$. However if the program is going to be used on n occasions, then

*Corresponding author

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082526

[1]ProbAbel's source can be found on http://www.genabel.org/packages/ProbABEL

the overall gain is accumulated for every execution (eq. 1).

$$\Delta = Y + \sum_{i=1}^{n} (X - X') \quad (1)$$

So we define the limit for the GI's reasonable time to be when $\Delta < 0$. More informally: the time it takes GI to decrease execution time *is reasonable* if the improved version can be found in less time than the accumulated saved time. The overhead of running GI will pay off even if the improvement is small if the resulting program is executed many times. In other words, the trade-off between investing in GI and running the improved version of the code multiple times is less than running the original code multiple times.

The name, GI, has a historical origin in the field of "Genetic Programming", a machine learning technique. The GI technique should not be confused with genetics in the biological field of genome analysis. The remainder of this paper is as follows: Section 2 lists some related work and inspirations. Section 3 gives a short explanation of ProbAbel's functionality. Section 4 describes the data set that was used for the experimentation and how it was generated. Section 5 lays out the experimental procedure. Section 6 details our results. Lastly, Section 7 summarizes and concludes the paper.

2 RELATED WORK

GI has been used to optimise various properties of software [4, 5, 11, 13], both non-functional and functional. Of the functional properties, the most studied improvement is bug fixing [8, 17, 27, 28]. Execution time is however the most explored non-functional property [5]. We believe this might be because implementing a time measurement is relatively easy compared to other non-functional properties such as memory use or energy consumption [6].

Some of the metrics used in the literature to quantify execution time include the elapsed time from invocation until termination, the CPU time, or number of lines of code executed. The elapsed time from program invocation to termination is dependent on multiple outside factors and is therefore an inaccurate measurement, unless it is made in the exact, unchanging environment the program will always be running in. That is however nearly never the case and the CPU time is a much more accurate and reliable measurement of the program's performance. The CPU time only counts the time it takes the computer to process the program in question so environmental variable interference is kept lower than with elapsed time but not entirely eliminated. Counting the lines of code that are executed has been argued to be the least biased with respect to the execution environment [21]. Nevertheless, the implementation would rely on instrumentation of the code or some kind of profiling tool and therefore is not as portable between programming languages.

This paper adds to an already extensive list of GI work on optimising execution time. Successes range from subtle [22] and moderate gain [21] to extraordinary [12, 14]. Much of it based on Harman's and Langdon's work [10] which also inspired the work presented in this paper.

Other examples of execution time improvements also include the early example of automatic parallelisation [26] and a multi-objective optimisation of embedded systems [29, 31].

Table 1: The 16 targeted files from ProbAbel's source code

File name	Size (LOC)	Number of mutable points
reg1.cpp	879	1236
main.cpp	619	284
coxph_data.cpp	556	201
coxfit2.c	465	696
main_functions_dump.cpp	448	159
eigen_mematrix.cpp	433	348
gendata.cpp	276	218
phedata.cpp	275	217
data.cpp	273	152
regdata.cpp	270	261
cholesky.cpp	154	216
maskedmatrix.cpp	154	105
chinv2.c	64	71
cholesky2.c	60	68
chsolve2.c	46	43
dmatrix.c	19	22
Total	4991	4297

Although GI has been used to improve execution time there is limited number of publication on the analysis of the search landscape. Our previous work examined the landscape for three small Python programs [9], while other recent papers look at the robustness of software [15] and bug fixing landscape of the *triangle* program [16].

3 GENOME-WIDE ASSOCIATION

A genome-wide association (GWA) study is the analysis of genetic variants in groups of people to identify which variations, if any, are associated with a certain trait or disease [24]. Generalised Linear Models (GLM) are typically used to approximate the effect size of a genetic variation by calculating the *odds ratio* (logistic models only) and a significance (*p-value*). There are multiple different methods and programs available to perform the computation of a GLM [19, 25]. The data input to a GWA study is a record of multiple single-nucleotide polymorphisms (SNPs) for a population of people and the trait or disease of interest. An SNP is a variations of a nucleotide in a specific location of a genome. Most GWA studies gather DNA samples from multiple people and consider millions of SNPs from each person [18]. Given that the studies are analysing correlation between millions of data points and perhaps multiple traits in hundreds or thousands of people the computation is often expensive.

3.1 ProbAbel

ProbAbel is widely used in bioinformatics for GWA studies. It is a collection of programs for regression models; linear, logistic and Cox proportional hazard. The software is versatile and relatively fast because it uses estimations and floating point types instead of double precision numbers. The Icelandic Heart Association is one of ProbAbel's users. They conduct GWA studies on a regular basis, trying to identify an underlying genome variation associated with

Table 2: Sampled distributions for the generating larger data set

Trait	Data type	Distribution	Parameters
Sex	Categorical	Discrete Uniform	$\{0, 1\}$
Height	Continuous	Truncated Normal	$\mu = 172$ $\sigma = 8$ $a = 150, b = 200$
Age	Continuous	Truncated Normal	$\mu = 55$ $\sigma = 15$ $a = 10, b = 99$

increased risk of many diseases or conditions. Typically the data consists of approximately 30 million SNPs from 10-20 thousand people and each run of the software can take up to 12 hours.

ProbAbel is written in C and C++ and utilises the R project's [23] *GLM* functionality for the bulk of the computations. The source code is approximately 8k lines of code, including comments, divided between 31 source files in total.

ProbAbel was profiled before any modifications were made to it. The profiling revealed that the majority of the execution time was spent in 16 files which the GI was set to target. Table 1 lists these files their sizes and the number of operators that were marked as being changeable given the mutation operators in Table 5. The program can be changed in multiple ways but assuming we only consider the sets of operators from Table 5 there are 18993 first order mutants and over 360 million second order mutants for the total 4297 locations of mutable points. The GI's search space for *ProbAbel* is vast since the number of all possible variations of the program is over 2.5×10^{2683}, without counting variations that can be made by also moving around lines.

4 TEST DATA

ProbAbel ships with a small set of example data for testing purposes and to enable potential users to become familiar with its use. The example data only has 5 SNPs for 200 people and the recorded trait is height in centimetres. It also has a record of age and sex of each person because GWA studies often have to account for such confounding variables. The data has intentionally missing values and marks them as *NA* value to test the imputation ability of *ProbAbel*. Running the program with the example data takes a fraction of a second which is largely due to overhead like initiation and reading the data into memory. To successfully measure the impact of the GI's improvements on execution time we generated a larger set of data from the example set.

Statistical sampling was used to increase the data set's size, both generating samples of more people and SNPs. To avoid too much homogeneity in the generated data set each trait (height, age and sex) was sampled independently. Table 2 lists the estimated distributions from the example data set used to generate each trait. Gender has equal likelihood of being male or female, height is drawn from a truncated normal distribution, as is the age.

The SNP data was expanded using bootstrapping with replacement for allele, frequency and dosage. An allele is a variation of the gene expression and can be multiple combinations of GACT. Frequency is the frequency of each allele and is a real number in the open interval $(0, 1)$ and dosage is the number of copies of the SNP, measured as a real number. For more detailed description of these variables we suggest any book on genetic biology.

Table 3 lists 7 generated data sets; their size and average execution time for the original program and two best GI variants as trained on D3. We assume that the random sampling of the traits from continuous distributions ensures that training on D3 will not overfit for the other datasets. As seen in Figure 1 the relationship between both the number of people and SNPs, and execution time is linear. Additionally the computational cost of the GWA is more affected by the number of SNPs than the number of people as demonstrated by the much higher gradient on the axis with the number of SNPs.

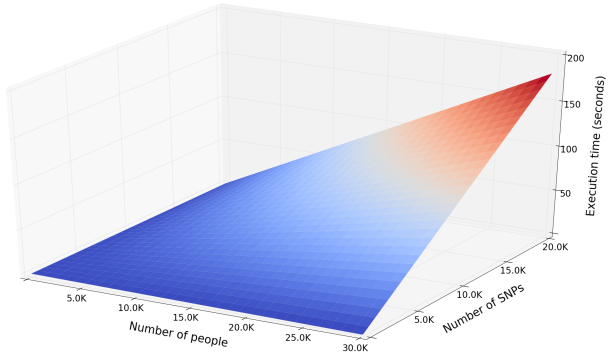

Figure 1: Execution time of the original program with respect to data set size; number of people, and SNPs.

5 EXPERIMENTAL SETUP

The experiments were conducted to answer the two questions from Section 1:

(1) Can GI, within reasonable time, find improvements to ProbAbel that decrease its execution time?
(2) What does the landscape for the execution time improvements look like?

To answer both we focused on the execution time of linear modelling with *ProbAbel*. We focused only on one model in order to reduce the amount of code modified and thereby ensuring that we only had to compile part of the software and decreasing the overhead time for every evaluation considerably, from approximately 50 seconds down to 12. For both questions we utilise data set **D3** because it is the smallest of the datasets such that the measurements' variations are guaranteed to be less than the average execution time. The timing mechanism we used measures in milliseconds and executing *ProbAbel* on **D3** takes more than a second. However, we do test the original and two of the best performing variants on all seven sets (see Table 3).

Statistical tests were run to determine whether the variants performed better than the original. We pairwise test the two programs against the original with a two-sided Student t-test where the null hypothesis is H_0: The means are equal.

Table 3: Seven different data sets of different sizes; population and SNPs. Execution time is measured in seconds and averaged over 20 test runs for each, data set and program variant. For the program variants the p-value of the Student t-test for two independent variables is also listed. Each variant is tested against the original.

Data set	People	SNPs	Original	Variant 1 (p-value)	Variant 2 (p-value)
D1	200	5	0.0050	0.005 (0.81)	0.005 (0.70)
D2	5,000	100	0.1600	0.158 (0.25)	0.158 (0.10)
D3	10,000	1000	2.9625	2.957 (0.36)	2.959 (0.46)
D4	20,000	1000	6.0020	5.985 (0.42)	5.998 (0.84)
D5	20,000	5000	29.895	29.781 (< 0.01)	29.782 (0.01)
D6	20,000	10000	60.020	59.722 (< 0.001)	59.708 (< 0.001)
D7	30,000	20000	182.000	182.110 (0.84)	181.920 (0.46)

5.1 GI Parameters

The first part of the experiment is set up as a population based evolutionary algorithm with the search parameters listed in Table 4. The improvement process iterates over 50 generations with population size 40. The entities being evolved are edit lists as seen in Figure 2. Edits are of two types: *Macro* (moving whole lines) and *micro* (changing a sub-string of a line). *Macro* edits consist of an operation (*delete*, *replace*, *copy* or *swap*) and line numbers, one or two depending on the operation. The *micro* edits have a location (line number and column number), the sub-string to replace, and the replacement. The sub-strings can be any defined token, such as variable names but in this work only operators from Table 5 are considered.

The first generation is a set of edit lists of length one. There is no elitism and half of each generation is selected as parents to the next generation. Selection is made by weighing each program with normalised fitness, so even those with poor performance have a chance of being picked. Every parent undergoes a single mutation to make a single child for the next generation. That makes half of the generation and the remainder of the 40 edit lists are randomly generated with single-edits. So the search effectively has a soft restart implementation which should prevent early convergence or too much homogeneity in the population.

A single mutation to an edit list can be made with any of the examples in Figure 2. The options for mutations are:

Grow is where a randomly generated edit is appended to the edit list. The edit is generated by a stochastic selection from all possible locations in the source with a equal probability. The location can be a line and a column or just a line. If the case is the former, then another uniform random selection is made from possible sub-string replacements (see Table 5) or another location is selected to which the content at the first location is to be copied. For the latter case, either one or two more selections are made. First a selection of what operation will be applied to the selected line; *delete*, *replace*, *copy* or *swap*. The random edit build stops here if *delete* is selected. For replace and swap another line of same type is randomly selected with equal probability to be the replacement or the line that swaps places. For copy a random line number in the source is selected to be the location above which the selected line is copied to.

Prune is when an edit in the list is selected with uniform random distribution and every subsequent edit in the list is removed.

Single edit change is perhaps the least disruptive mutation. A single edit is selected and one of its features is randomly changed, such as the replacement code is re-selected or the copy location is altered.

Every time a selection has to be made, while either constructing or modifying an edit, the probabilities are always uniform.

Table 4: GI parameters.

Number of generations	50
Population size	40
Initial edit list size	1
Survival rate	0

Table 5: Sets of single operators available to the GI. Any member of a given set can be changed to another member of the same set, ensuring syntactically valid modification.

Description	Operations
Numerical constants	Can increment by ±1
Arithmetic operators	+, −, ∗, /, %
Arithmetic assignments	+ =, − =, ∗ =, / =,
Incremental operators	++, −−
Relational operators	<, >, <=, >=, ==, ! =
Bit assignments	& =, | =
Bit operators	&, |

5.2 Fitness evaluation

Fitness is the accumulated time in seconds the CPU is occupied by the program. The objective is to minimise this value. Each variant's fitness was the mean execution time for 20 runs to even out the effect of outside processes. Additionally, every program variant is first tested by compiling, then running the test suite to confirm if the output is as required. A part of the test suite compares the actual output values with known correct values. If a program does not compile it is not completely discarded but given a fitness of approximately twice the original execution time. The test suite counts 52 tests and for each failed test case a proportion of the original execution time is added to the fitness evaluation. So for 1 failed test case 1/52 of the original execution time is added to the

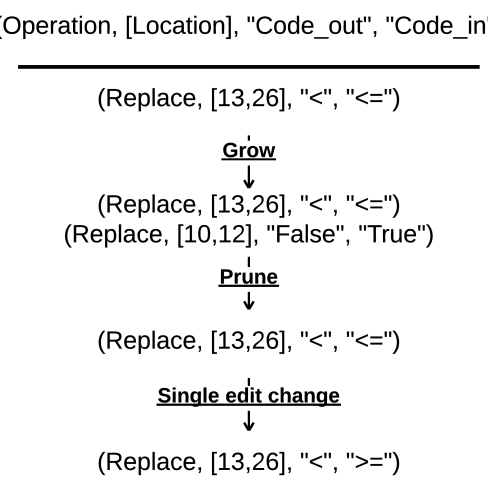

Figure 2: An example of an edit list and how it can evolve with *Grow*, *Prune* or *Single edit change*.

measured cpu time. This translates to roughly twice the original execution time for most variants that compile but fail all test cases. The penalty scheme ranks the following sub-performing programs in this order of preference:

(1) Variants that compile, fail all test cases but run faster than the original
(2) Uncompilable variants
(3) Compilable, fail all test cases, and run slower than the original.

This penalty scheme encourages shorter execution time.

The experiments were conducted on Ubuntu 14.04, with Intel i7-3820 and 16GiB RAM. Each program's execution time evaluation was measured with the Linux command *time* that returns the total number of seconds a process spends in *user mode*. To decrease the likelihood of dynamic overclocking from the base frequency of 3.60 GHz to 3.80 GHz, and to address the inherent noisy environment of a running machine we have done three things: 1) Ensured a single run of *ProbAbel* at a time, with no other intensive tasks running, 2) used the same specific machine for all experiments, and 3) we show the variation as box plots in Figure 3.

5.3 Exploring the landscape

To explore the landscape of ProbAbel's mutants we consider only *micro* edits, as adding the *macro* edits would expand the search space too much. The second part of the experiment is a random walk away from the original, that is repeated 100 times. For every walk we modify the original program in ten steps, with a single randomly generated edit in each step and measure the execution time. Effectively taking ten random steps into the landscape from the original. In addition, we evaluate a sample of first order mutants of the program by sampling the neighbourhood with uniform random selection with replacement from all 18993 possible first order mutants. We opt to have replacement to minimise memory usage that would have been used to keep record of previously evaluated program variants. The sample size is limited to what can be run in under ten hours which is approximately 2400 programs.

The setup for the second part of the experiment is similar to that in our previous work [9]. However, as the evaluation of ProbAbel's execution time is computationally more expensive than any fitness evaluation of a simple calculator or *K-means* initiation it is not feasible to do as thorough an analysis here. We nevertheless explore the landscape in the same manner, only with a few limitations. Apart from the fitness measurements the analyses differ in two ways. In our previous work the maximum edit list size we considered was 50 edits and we exhaustively searched the neighbourhood.

6 RESULTS

When the GI finished, it had evaluated 1760 unique edit lists, 240 were duplicates. The overall runtime of the GI was eight hours and fifteen minutes. The CPU time was consistently stable with maximum variation from the mean less than 25% for all data sets and each program variant. The overall best mean execution time was **2.957** seconds (*Variant 1*) on data set D3 and the next best executed on average in **2.959** seconds (*Variant 2*). Variant 1 was only a single edit:

```
<<< MacroEdit: Delete,[reg1.cpp, 321],
    chi2 = chi2 * (1. / sigma2_internal);
    //chi2 = chi2 * (1. / sigma2_internal); >>>
```

and was found in generation 10, after approximately an hour and a half. It deletes line 321 in `reg1.cpp` which has some effect on execution time but not on the output of the linear model of ProbAbel. The second best was found in generation 5, after 45 minutes, and was 4 edits long:

```
<<< MicroEdit: Copy,[reg1.cpp:153,40->153,33]
    "col_new++;", "++col_new;">>>
<<< MacroEdit: Delete, [main.cpp,169],
    coxph_reg nrd = coxph_reg(nrgd);
    //coxph_reg nrd = coxph_reg(nrgd);>>>
<<< MacroEdit: Delete, [main\_functions\_dump.cpp,73],
    std::cout.flush();
    //std::cout.flush();>>>
<<< MicroEdit: Copy,[reg1.cpp:791,14->791,9]
    "niter++;", "++niter;">>>
```

Manual inspection revealed that both macro edits delete lines that have no effect on the linear modelling functionality of ProbAbel and do not contribute to improved execution time. The two micro edits improve by changing a post-increment to pre-increment by copying the ++ in front of the variable. There is a slight performance improvement in using pre-increment in C because post-increment stores the old value after the increment. For a single execution of the statement, the difference is minimal but it accumulates when it is revisited for every column and row in a $10,000 \times 1,000$ matrix.

However, as seen in Table 3 neither of these variants' mean execution time is significantly different from the original on **D3**. They did significantly better on **D4** and **D5** only and the difference is quite small, less than 0.5%. As we further confirm in Figure 4 where we see that the difference is small but the confidence intervals of the medians do not overlap. Looking at Figure 3 we see that there is minuscule variation in the distribution of mean execution time over the whole evolution and we can also note that the number of variants that compiled without errors ranges from 23 (generation 46) to 34 (generations 21, 25 and 36).

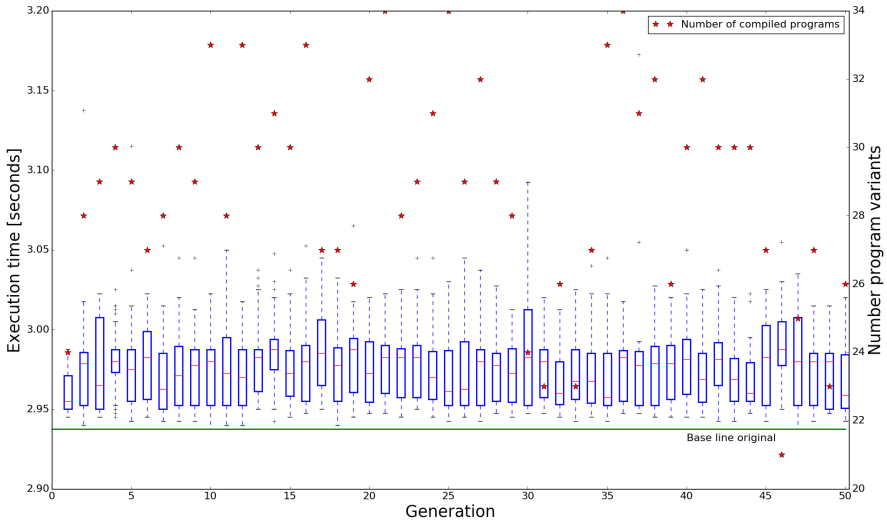

Figure 3: Distribution of ProbAbel's fitness and the number of compiled variants for each generation. *Left axis*: The execution time and the boxes are the distributions of mean execution times for each generation. *Right axis*: Number of program variants and the stars are the number of compiled variants.

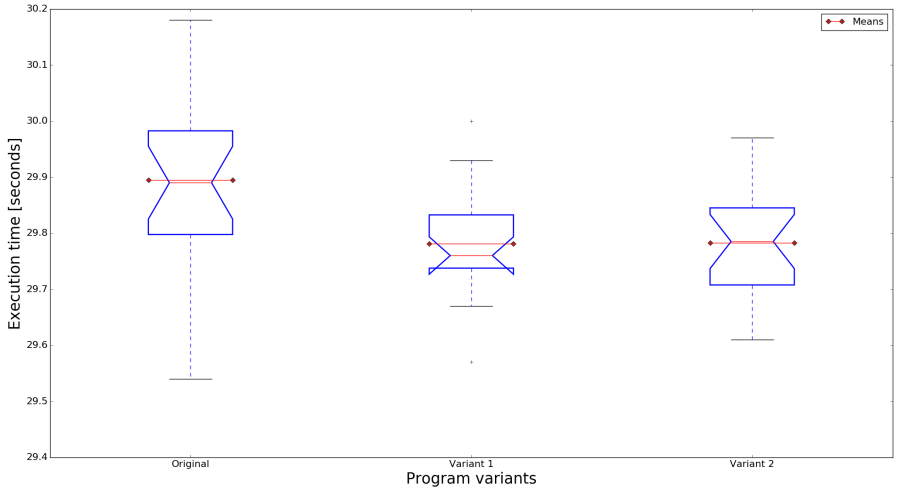

Figure 4: Distribution comparison of the execution time of the original and the two best variants on D5. Each box is constructed from 20 runs. The notches in the boxes are the 95% confidence interval for the median of each.

On two occasions the GI found *loopholes*, e.g. change an if statement such that the program read in much less data than was given to it. This results in a significant reduction in execution time, approximately 97%, but the results for the linear model were completely wrong. These two "cheats" the GI found were not included in any figures or tables because they would have skewed the scales and obscured the improvements of the two "good" variants.

6.1 Tracing from the original

Figure 5 shows the 15 levels of execution time ProbAbel variants exhibited. The graph demonstrates frequency of transitions from one execution time performance to another by adding a single random edit to an edit list. The execution time of 5 seconds denotes that a program variant was unable to compile and is an arbitrary number that is at least higher than the worst execution time of a

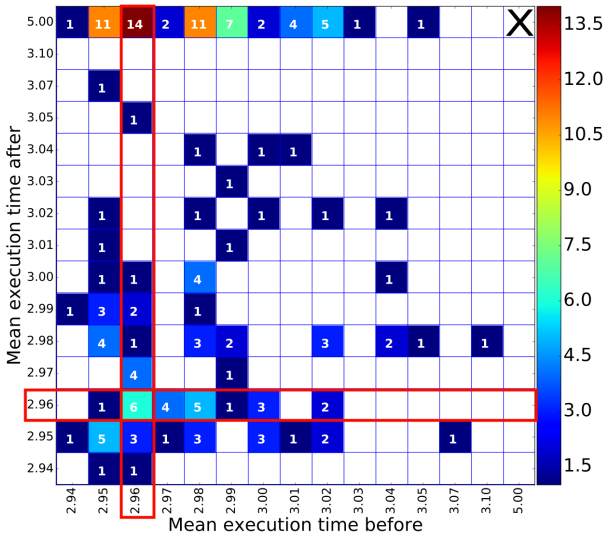

Figure 5: Frequency chart of execution time changes after a single edit is appended to the edit list and evaluated on D3. Each square is a count of how often the execution time changed from before to after. Row and column marked with red is equal to the original execution time. Execution time of 5 seconds denotes an uncompilable program variant and X is the omitted count of 748.

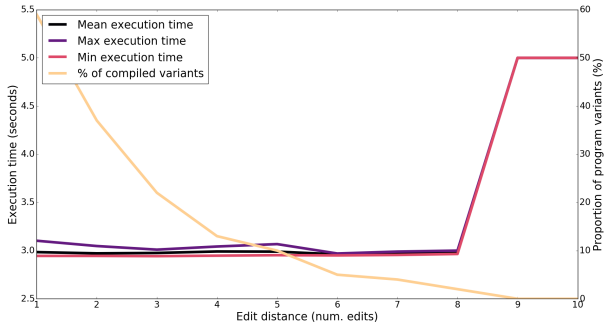

Figure 6: *Left axis*: The change in execution time as the program variants move away from the original. Mean, maximum and minimum execution time for 100 traces. *Right axis*: Proportion of program variants that compiled without errors.

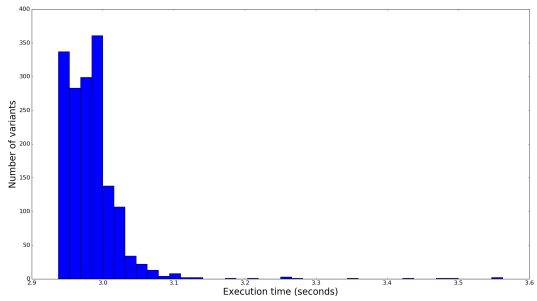

Figure 7: Distribution of the execution time within a single edit from the original program.

compiled variant. As seen in the Figure 5 most often a single edit caused a compilation error but otherwise the spread is relatively uniform from the original (bottom left). Furthermore we omitted the most frequent transition (marked with an X), which was essentially a non-transition; a neutral edit to an already uncompilable program variant. The number of such transitions was 748 or about 83% out of the 900, not counting the first order mutants.

By inspecting Figure 6 we see that the execution time itself does not get worse as we travel further away from the original. However, the proportion of program variants that do compile without errors decreases significantly until it reaches zero after 9 steps.

6.2 Neighbourhood exploration

After an overnight sampling of first order mutants we ended with 2265 unique variants out of 2400. Of those, 1622 compiled without errors but 643 failed to compile. The distribution of the execution time for those that compiled can be seen in Figure 7. It is interesting to see that, although the execution time does not improve with first order mutants, it does not increase considerably.

7 CONCLUSION

In this paper we describe a successful application of a GI framework to improving C/C++ source code, following previous success with improving Python source code. The adaptation to operate on C included only small changes to the class of operators in Table 5. Therefore, in theory, we can apply this same approach on many other programming language.

Our intentions with this paper were to answer the two questions in Section 1.

(1) GI can find improvements to ProbAbel that decrease its execution time. However, we have yet to confirm if the improvements were found within reasonable time.
(2) The execution time landscape is much like the bug fixing landscape. It is noisy as can be seen in Figure 3 but largely flat with the occasional drops and peaks. Additionally, our findings are complementary to the statement that "Software is Not Fragile" [15]. The majority of the first order mutants (1622) compiled and executed without an error.

The GI framework introduced in our previous work [8, 9] was able to improve the execution time of a C program. We were only able to find marginally better variants of ProbAbel as seen in Section 5. However, the 0.5% execution time decrease can translate into hours of saved time in the long term, which is a consideration for future work. GI is a one off, up front cost and considering that the improved version did better on a larger data set than it was trained on means that this cost does not need to be large.

Given the size of the search space, we also find it impressive that the GI found improvements at all. The size of the search space limited to only changing numbers and operators from Table 5

exceeds 10^{2683}. In our experiment we explored so little of it that the percentage is close to zero, about 4×10^{-2680}.

This presents us with one of the threats to the validity of our experiments: How can we be sure that what we explored is representative of the majority of the landscape? The short answer is that we cannot be sure. However, it can be argued that the explored landscape is representative of a trajectory from the original towards an improved version. This is, at least, an area of the landscape that is useful to explore.

A consideration for future work is to evaluate the penalty for failed test cases. Was the proportional increase to the execution time evaluation too harsh? That might have been restricting the search by not giving enough access to solutions that need to break the program before they improve it. Both variants that were considered overall best contained only edits that could be considered beneficial or neutral on their own and no edit that made ProbABEL uncompilable. Another item on the agenda for future work is to investigate the fitness landscape of larger modifications to the source code. Will it be similar to the one presented here with smaller changes?

8 ACKNOWLEDGMENTS

The work presented in this paper is part of the DAASE project which is funded by the EPSRC Grant EP/J017515/1.

REFERENCES

[1] Y. S. Aulchenko, M. V. Struchalin, and C. M. van Duijn. ProbABEL package for genome-wide association analysis of imputed data. *BMC bioinformatics*, 11:134, 2010.

[2] 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.

[3] 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.

[4] S. O. Haraldsson, R. D. Brynjolfsdottir, J. R. Woodward, K. Siggeirsdottir, and V. Gudnason. The Use of Predictive Models in a Dynamic Planning of Treatment. In *Proceedings - IEEE Symposium on Computers and Communications*, Heraklion, Greece, 2017. IEEE.

[5] 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.

[6] S. O. Haraldsson and J. R. Woodward. Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate. In *Proceedings of the 2015 Conference Companion on Genetic and Evolutionary Computation Companion*, pages 831–832, Madrid, 2015. ACM.

[7] S. O. Haraldsson, J. R. Woodward, A. E. Brownlee, and K. Siggeirsdottir. Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success. In *Proceedings of the 2017 Conference Companion on Genetic and Evolutionary Computation Companion*, Berlin, Germany, 2017. ACM.

[8] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and D. Cairns. Exploring Fitness and Edit Distance of Mutated Python Programs. In *Proceedings of the 17th European Conference on Genetic Programming, EuroGP*, Amsterdam, The Netherlands, 2017. Springer Berlin Heidelberg.

[9] S. O. Haraldsson, J. R. Woodward, and A. I. E. Brownlee. The Use of Automatic Test Data Generation for Genetic Improvement in a Live System. In *8th International Workshop on Search-Based Software Testing*, Buones Aires, 2017. ACM.

[10] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 1–14, 2012.

[11] 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 *6th International Symposium, SSBSE 2014*, volume 8636 of *Lecture Notes in Computer Science*, pages 259–264. Springer Berlin Heidelberg, Fortaleza, Brazil, aug 2014.

[12] W. B. Langdon and M. Harman. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, feb 2015.

[13] W. B. Langdon, B. Y. H. Lam, M. Modat, J. Petke, and M. Harman. Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines*, pages 1–40, 2016.

[14] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO '15, pages 1063–1070, Madrid, jul 2015. ACM.

[15] W. B. Langdon and J. Petke. Software is Not Fragile. In *First Complex Systems Digital Campus World E-Conference 2015*, Proceedings in Complexity, pages 203–211. Springer International Publishing, Cham, 2015.

[16] W. B. Langdon, N. Veerapen, and G. Ochoa. Visualising the Search Landscape of the Triangle Program. In *EuroGP 2017*, pages 19–21, 2017.

[17] 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.

[18] D. Levy, et al. Genome-wide association study of blood pressure and hypertension. *Nature Genetics*, 41(6):677–687, jun 2009.

[19] R. Mägi and A. P. Morris. GWAMA: software for genome-wide association meta-analysis. *BMC Bioinformatics*, 11(1):288, 2010.

[20] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*, In press, 2017.

[21] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class. In *17th European Conference on Genetic Programming, EuroGP 2014*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149, Granada, Spain, 2014. Springer Berlin Heidelberg.

[22] J. Petke, W. B. Langdon, and M. Harman. Applying Genetic Improvement to MiniSAT. In *5th International Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262, St. Petersburg, Russia, aug 2013. Springer Berlin Heidelberg.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[24] T. M. Teslovich, et al. Biological, clinical and population relevance of 95 loci for blood lipids. *Nature*, 466(7307):707–713, aug 2010.

[25] A. Vaez, P. J. van der Most, B. P. Prins, H. Snieder, E. van den Heuvel, B. Z. Alizadeh, and I. M. Nolte. lodGWAS: a software package for genome-wide association analysis of biomarkers with a limit of detection. *Bioinformatics*, 32(10):1552, 2016.

[26] P. Walsh and C. Ryan. Paragen: A Novel Technique for the Autoparallelisation of Sequential Programs using Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 406–409, Stanford University, CA, USA, 1996. MIT Press.

[27] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109, 2010.

[28] 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.

[29] D. R. White. *Genetic Programming for Low-Resource Systems*. Phd, University of York, 2009.

[30] D. R. White. An Unsystematic Review of Genetic Improvement. In *45th CREST Open Workshop on Genetic Improvement*, London, 2016.

[31] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, aug 2011.