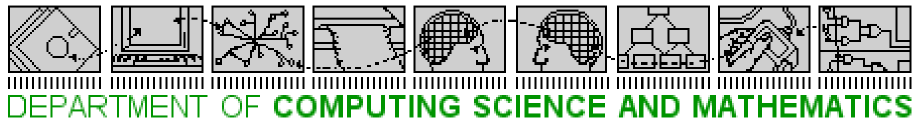
GENETIC IMPROVEMENT OF SOFTWARE

From Program Landscapes to the Automatic Improvement of a Live System

SAEMUNDUR OSKAR HARALDSSON



Doctor of Philosophy

Institute of Computing Science and Mathematics
University of Stirling

May 2017

DECLARATION

I, Saemundur Oskar Haraldsson, hereby declare that all material in this thesis is my own original work except where reference has been given to other authors. The work presented here has not been submitted for the award of any other degree at the University of Stirling nor any other Institute.

Stirling, May 2017

Saemundur Oskar Haraldsson

ABSTRACT

In today's technology driven society, software is becoming increasingly important in more areas of our lives. The domain of software extends beyond the obvious domain of computers, tablets, and mobile phones. Smart devices and the internet-of-things have inspired the integration of digital and computational technology into objects that some of us would never have guessed could be possible or even necessary. Fridges and freezers connected to social media sites, a toaster activated with a mobile phone, physical buttons for shopping, and verbally asking smart speakers to order a meal to be delivered. This is the world we live in and it is an exciting time for software engineers and computer scientists. The sheer volume of code that is currently in use has long since outgrown beyond the point of any hope for proper manual maintenance. The rate of which mobile application stores such as Google's and Apple's have expanded is astounding.

The research presented here aims to shed a light on an emerging field of research, called Genetic Improvement (GI) of software. It is a methodology to change program code to improve existing software. This thesis details a framework for GI that is then applied to explore fitness landscape of bug fixing Python software, reduce execution time in a C++ program, and integrated into a live system.

We show that software is generally not fragile and although fitness landscapes for GI are flat they are not impossible to search in. This conclusion applies equally to bug fixing in small programs as well as execution time improvements. The framework's application is shown to be transportable between programming languages with minimal effort. Additionally, it can be easily integrated into a system that runs a live web service.

The work within this thesis was funded by EPSRC grant EP/J017515/1 through the DAASE project. [1]

1 http://daase.cs.ucl.ac.uk/

To my wife and children

ACKNOWLEDGMENTS

During the four years of my studies at the University of Stirling I have been fortunate to benefit from discussions with a large number of people. I will never be able to include each of them personally in my thanks and so I extend my gratitude to all I have had the pleasure to exchange ideas with.

First I would like to thank my supervisory team for a very fruitful partnership, especially John which I consider my close friend. Sandy has contributed no less to my progress even though he joined the team half-way through. David Cairns gets my thanks for insightful and well thought out observations every time we meet. Edmund Burke was responsible for bringing me to Scotland and for that I am truly thankful. I am also very thankful for having had the opportunity to work with the other original PI's of the DAASE project Mark Harman, John Clark, and Xin Yao. Every discussion (social and technical) with any one of you had an impact on my work and career. Additionally I have benefited for getting to know Earl Barr, the newly appointed PI of the DAASE project. Our discussions when you came to Stirling and while we had our day of adventure in Buenos Aires have helped my research and career directions after the PhD.

Much of the work in this thesis had not been possible without brilliant collaborators. Janus Rehabilitation and the Icelandic Heart Association have provided me with the material to experiment with. Albert V. Smith has my thanks for good ideas and giving me a very short lesson in genetics. I have been extremely lucky by having the full trust of Kristín and Vilmundur. Half of the contents of this thesis would not have been realised if it were not for their complete faith in me.

Within DAASE, I have had the extreme pleasure to work with one of my long time idols, Bill Langdon. Every time we meet I am reminded why I want to stay in academia. Justyna Petke and David White, I thank you for our collaborations, ideas, and most of all friendship. There is never a dull moment when we meet, in conferences and events. Everyone else on DAASE has my thanks as well but there are just to many to list.

I would like to thank the other PhD students, past and present, in particular: Jason, Kevin, Annan, Ken, Paul, and Sarah thanks for sharing coffee, interesting conversations, laughter, and fun whenever there was need.

My parents and brother are admirable for putting up with me before Heiða took me off their hands: I hope I make you proud. My in-laws, Brynjólfur and Kristín, have a special place in my thoughts and I will be forever grateful for their support.

Lastly and most importantly I have everything to thank for my wonderful wife, Heiða, ég elska þig. This thesis along with all the research and work that led to its submission would not have been possible without you. Your patience, support, and encouragements have inspired and helped me immensely. My children, Guðný Birna and Brynjólfur Kristinn, deserve compliments for their endurance in waiting for their dad to finish the thesis.

LIST OF PUBLICATIONS

The following publications present work that was used in this thesis:

- Saemundur O. Haraldsson and John R. Woodward Automated design of algorithms and genetic improvement: contrast and commonalities. In Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation 2014 Jul 12 (pp. 1373-1380). ACM.

  This postition paper served as a basis for the literature review.

- Saemundur O. Haraldsson and John R. Woodward Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate. In Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation 2015 Jul 11 (pp. 821-822). ACM.

  This postition paper is referenced in the literature review and forms a large part of the energy optimisation section.

- Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee and David Cairns Exploring Fitness and Edit Distance of Mutated Python Programs. In European Conference on Genetic Programming 2017 Mar 30 (pp. ?-?). Springer International Publishing.

  This paper is the underlying publication for the work in Chapter 6.

- Saemundur O. Haraldsson, Ragnheidur Dora Brynjolfsdottir, John R. Woodward, Kristin Siggeirsdottir, and Vilmundur Gudnason. The Use of Predictive Models in a Dynamic Planning of Treatment. In Proceedings - IEEE Symposium on Computers and Communications, Heraklion, Greece, 2017. IEEE.

  The work of this paper is used in Chapter 8

- Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee, and Kristin 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.

  This paper forms the work of Chapter 8

- Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee, Albert V Smith, and Vilmundur Gudnason. Genetic Improvement of Runtime and its fitness landscape

in a Bioinformatics Application. In Proceedings of the 2017 Conference Companion on Genetic and Evolutionary Computation Companion, Berlin, Germany, 2017. ACM.

This paper is the basis for Chapter 7

- Saemundur O Haraldsson, John R Woodward, and Alexander 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.

  This paper is the predecessor of the main paper for Chapter 8

The following papers were written while conducting the research for this thesis. They are cited and mentioned but do not contribute to the main narrative.

- Kristin Siggeirsdottir, Ragnheidur Dora Brynjolfsdottir, Saemundur Oskar Haraldsson, Sigurdur Vidar, Emanuel Geir Gudmundsson, Jon Hjalti Brynjolfsson, Helgi Jonsson, Omar Hjaltason, and Vilmundur Gudnason. Determinants of outcome of vocational rehabilitation. Work, 55(3):577–583, nov 2016.
- Kocsis ZA, Neumann G, Swan J, Epitropakis MG, Brownlee AE, Haraldsson SO, Bowles E. Repairing and optimizing Hadoop hashCode implementations. In International Symposium on Search Based Software Engineering 2014 Aug 26 (pp. 259-264). Springer International Publishing.

CONTENTS

i INTRODUCTION 1

1 MOTIVATIONS FOR IMPROVING PROGRAMS AUTOMATICALLY 2

- 1.1 Introduction . . . 2
- 1.2 Genetic Improvement . . . 3
- 1.3 Thesis Structure . . . 4

2 HYPOTHESES 5

- 2.1 Introduction . . . 5
- 2.2 Central Hypothesis . . . 5
  - 2.2.1 Small Programs' Landscapes . . . 5
  - 2.2.2 Execution Time Improvements . . . 6
  - 2.2.3 Dynamic Adaptive Software . . . 7
- 2.3 Summary . . . 7

ii LITERATURE REVIEW 8

3 GENETIC IMPROVEMENT OF SOFTWARE 9

- 3.1 Introduction . . . 9
- 3.2 Representation of Changes to Programs . . . 10
  - 3.2.1 Whole Programs . . . 11
  - 3.2.2 Patches . . . 15
  - 3.2.3 Programming Languages Targeted with GI . . . 20
  - 3.2.4 Non-Evolutionary Representations . . . 24
- 3.3 Search Methodologies used for GI . . . 25
  - 3.3.1 Genetic Programming . . . 25
  - 3.3.2 Genetic Algorithm . . . 27
  - 3.3.3 Other Search Methods . . . 28
- 3.4 Software Properties Improved with GI . . . 29
  - 3.4.1 Functional Properties . . . 29
  - 3.4.2 Non-Functional Properties . . . 32

4 DYNAMIC ADAPTIVE SOFTWARE FOR GENETIC IMPROVEMENT 36

iii CONTRIBUTION 39

5 THE IMPLEMENTATION OF THE FRAMEWORK 40

- 5.1 Introduction . . . 40
- 5.2 Representation . . . 40

5.3 Input and Parameters . . . 43
5.4 Procedure . . . 45
5.5 Summary . . . 46
6 THE FITNESS LANDSCAPE OF MUTATED PYTHON PROGRAMS 48
6.1 Introduction . . . 48
6.2 The Python programming language . . . 49
6.3 The Framework Setup . . . 50
6.3.1 Search algorithm . . . 51
6.4 Experimental setup . . . 52
6.4.1 Random walk analysis . . . 52
6.4.2 Exhaustive Local Neighbourhood Analysis . . . 53
6.4.3 Description of the programs targeted by GI . . . 54
6.5 Results . . . 56
6.5.1 Size Versus Change in Fitness . . . 57
6.5.2 Average Fitness with Respect to Edit List Size . . . 62
6.5.3 Discrete steps in fitness . . . 66
6.5.4 Exhaustive Neighbourhood Evaluation . . . 67
6.6 Summary . . . 71
7 REDUCING EXECUTION TIME WITH GI 74
7.1 Introduction . . . 74
7.2 ProbAbel: A bioinformatics Software Package . . . 75
7.3 Test Data . . . 77
7.4 Experimental Setup . . . 78
7.4.1 GI Parameters . . . 79
7.4.2 Fitness evaluation . . . 81
7.4.3 Exploring the Execution Time Landscape . . . 82
7.5 Results of Execution Time Improvements . . . 82
7.5.1 Random Walk Exploration of Execution Time . . . 85
7.5.2 Local Neighbourhood Exploration of Execution Time . . . 85
7.6 Conclusion . . . 85
8 DEPLOYMENT OF GENETIC IMPROVEMENT IN A LIVE SYSTEM 89
8.1 Introduction . . . 89
8.2 Janus Manager's Daytime activity . . . 91
8.2.1 Usage of Janus Manager . . . 91
8.2.2 Daytime Usage Monitoring . . . 91
8.2.3 Structure of Janus Manager . . . 95
8.3 Janus Manager Nightly activity . . . 95
8.3.1 Log analysis . . . 96

8.3.2 Generating test data . . . 96
8.3.3 Genetic Improvement . . . 98
8.4 Predictions for Dynamic Treatment Planning . . . 100
8.4.1 Evaluation of The Predictor . . . 102
8.5 Performance Review . . . 103
8.5.1 Bug fixing . . . 103
8.5.2 Prediction Improvements . . . 107
8.6 Summary . . . 110

iv CONCLUSIONS 112

9 CONCLUSIONS 113
9.1 Q1: Small Programs' Landscapes . . . 113
9.2 Q2: Execution Time Improvements . . . 114
9.3 Q3: Dynamic Adaptive Software . . . 114
9.4 Central Hypothesis . . . 115
9.5 Future work . . . 115
9.6 Summary of Chapter 9 . . . 116

v APPENDICES 1

A APPENDIX A 2
A.1 Python Source Code . . . 2
A.1.1 Calculator.py . . . 2
A.1.2 _kmeans_.py . . . 5
A.1.3 latex.py . . . 9

B APPENDIX B 11
B.1 Glossary . . . 11

LIST OF FIGURES

Figure 3.1	Traditional Genetic Programming (GP) tree representations. Nodes $*$,$+$ and max are called functions while X, Y, Z and 1 are terminals.	12
Figure 3.2	A simple crossover of two trees. $X \times (Z+Y)$ and $(3+Y)-(Z/X)$ become $(Z+Y)-(Z/X)$. The clouds constitute the two parts that are in the final program.	13
Figure 3.3	Three-stage lookup of a modification table from a genome as represented in Ackling et al. [1]. The image is taken from the same publication.	16
Figure 3.4	An example of a delta debugging process as used in [102]. Process iterates through the edit list and removes a single edit at a time, then evaluates the reduced edit list and if the removal of the edit has no effect on fitness it is not reinserted.	19
Figure 3.5	Sequential representation of a GP program. Each block is an instruction in a sequence but blocks can be repeatedly executed like those inside the *for* loop	26
Figure 4.1	A system diagram of Gen-O-Fix [188]	37
Figure 5.1	An example of an edit list and how it can evolve with *Grow*, *Prune* or *Single edit change*.	40
Figure 5.2	The four types of edits that can be applied to the source code. More detailed descriptions and list of arguments to each can be found in Table 5.1	42
Figure 5.3	The general outline of the search algorithm implemented in the framework.	46
Figure 6.1	Distributions of the edit list size when the fitness decreases for the first time during the experiments for each program (Δ Table 6.5).	59
Figure 6.2	Distributions of the edit list size when fitness reaches zero for first time (Ω Table 6.5).	60
Figure 6.3	Distributions of the edit list size when (if) fitness starts increasing again (Ψ Table 6.5).	61
Figure 6.4	Fitness distribution with respect to edit list size for each increment. The tapering denotes the 95% confidence interval for the median fitness of each edit list size.	63
Figure 6.5	Mean fitness (95% error) with respect to edit list size for each increment.	64
Figure 6.6	Median fitness (95% error) with respect to edit list size for each increment.	65

Figure 6.7 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. 67
Figure 6.8 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. 68
Figure 6.9 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. 68
Figure 6.10 Unique patterns of passed/failed for every program variant (**P1**) and every test case. Green indicates a pass while red is fail. The columns and rows are sorted by the number of occurrences for each unique combination of passed/failed. 69
Figure 6.11 Unique patterns of passed/failed for every program variant (**P1**) and every test case. Green indicates a pass while red is fail. The columns and rows are sorted by fitness (maximum green) for each unique combination of passed/failed. 70
Figure 6.12 Unique patterns of passed/failed for every program variant (**P3**) and every test case. Green indicates a pass while red is fail. 71
Figure 6.13 Unique patterns of passed/failed for every program variant (**P2**) and every test case. Green indicates a pass while red is fail. 72
Figure 7.1 Execution time of the original program with respect to data set size; number of people, and SNPs. The graph shows a surface that is approximately linear with respect to both independent variables. Both variants,1 and 2 produce near identical figures as suggested by the values in Table 7.3 79
Figure 7.2 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. 83
Figure 7.3 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. 84

Figure 7.4 Execution time before and after a single edit is appended to the edit list and evaluated on D3. Each square contains 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. . . . 86
Figure 7.5 *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. . . . 87
Figure 7.6 Distribution of the execution time within a single edit from the original program. . . . 88
Figure 8.1 Janus Manager (JM)'s feature map (part 1) as it developed during the software's first 8 months in use. It shows how rapidly features were added after employees of Janus Rehabilitation Centre (JR) started using the software in March 2016. This rapid development caused buggy and less well-tested code to be released, hence inspiring the integration of GI. 92
Figure 8.2 JM's feature map (part 2) as it developed during the software's first 8 months in use. It shows how rapidly features were added after employees of JR started using the software in March 2016. This rapid development caused buggy and less well-tested code to be released, hence inspiring the integration of GI. . . . 93
Figure 8.3 JM functionality divided into daytime processes and night time processes. 94
Figure 8.4 A flow chart showing the prediction and update process while a single patient receives treatment. The patient attends their treatment schedule and provides data. The specialist records the information, reviews predictions, and plans the treatment jointly with the patient. The predictor processes the data, makes predictions and visualises them. Lastly, the Genetic Improvement updates the predictor when the patient finishes. 101
Figure 8.5 The distribution of treatment length for the 73 patients that finished treatment during the ten month period. . . . 107
Figure 8.6 Precision and accuracy of predictions for dropping out and successful treatment over the trial period. . . . 108
Figure 8.7 Distribution of post hoc evaluation of MAD for every two weeks of updated models for treatment length. Both mean (diamond) and median (triangle) are marked with each box. . . . 109

Figure 8.8 Distribution of post hoc evaluation of MSE for every two weeks of updated models for treatment length. Here, converted to Root Mean Squared Error for scaling on y-axis. Both mean (diamond) and median (triangle) are marked with each box. 110

LIST OF TABLES

Table 3.1 Additional operator classes defined as parameter values for deep parameter tuning representation [210] 18
Table 3.2 The different properties of software that have been improved by GI. . . 30
Table 5.1 The choice of edit operations and their arguments. Granularity can either be 41
Table 5.2 Parameters to the GI framework. 47
Table 6.1 List of defined line types that cannot be altered and therefore not accessible to the GI. 50
Table 6.2 List of defined line types that can be altered and targeted by the GI . . 51
Table 6.3 Sets of single operators and constants available to the GI. One member of a given set can be changed to another member of the same set. . . . 51
Table 6.4 Information about the programs that were used in the experiments . . 55
Table 6.5 Statistics for the variables defined in section 7.4, edit list size $|x|$ when changes in fitness $f(x)$ are detected and total number of fitness evaluations during the experiments. 57
Table 7.1 The 16 targeted files from ProbAbel's source code 76
Table 7.2 Sampled distributions for generating larger data set 77
Table 7.3 Seven different data sets of different sizes (population and SNPs). Execution time is measured in seconds (CPU) and averaged over 20 test runs for each, data set and program variant. For the program variants the p-value of the Student's t-test for two independent variables is also listed. Each variant is tested against the original. 78
Table 7.4 GI parameters. 80
Table 7.5 Sets of single operators available to the GI. One member of a given set can be changed to another member of the same set. 80
Table 7.6 List of defined line types for improving *ProbAbel* 81
Table 8.1 Sets of single operators available to the GI. One member of a given set can be changed to another member of the same set. 98
Table 8.2 List of defined line types that cannot be altered and therefore not accessible to the GI. 99
Table 8.3 List of defined line types that can be altered and targeted by the GI . . 99
Table 8.4 Data types of features in the set, number and examples. 103

Table 8.5 A list of the bugs that were automatically detected in JM and fixed by the GI. They are categorised by the type of exception that they caused and given an identification number in the second column. 105

LIST OF ACRONYMS

AST Abstract Syntax Tree

BNF Backus-Naur Form

GA Genetic Algorithm

GI Genetic Improvement

GP Genetic Programming

EA Evolutionary Algorithm

GWA Genome-Wide Association

LOC Lines of Code

JM Janus Manager

JR Janus Rehabilitation Centre

SBSE Search Based Software Engineering

SBST Search Based Software Testing

JVM Java Virtual Machine

Part I

INTRODUCTION

MOTIVATIONS FOR IMPROVING PROGRAMS AUTOMATICALLY

1.1 INTRODUCTION

In today's technology driven society, software is becoming increasingly important in more areas of our lives. The domain of software extends beyond the obvious domain of computers, tablets, and mobile phones. Smart devices and the internet-of-things [83] have inspired the integration of digital and computational technology into objects that some of us would never have guessed could be possible or even necessary. Fridges and freezers connected to social media sites, a toaster activated with a mobile phone, physical buttons for shopping, and verbally asking smart speakers to order a meal to be delivered. This is the world we live in and it is an exciting time for software engineers, computer scientists, and specifically GI practitioners. The sheer volume of code that is currently in use has long since outgrown beyond the point of any hope for proper manual maintenance. The rate of which mobile application stores such as Google's and Apple's have expanded is astounding. From 2008 to 2015 both stores accumulated over 1 million applications [134] and most of them are regularly updated with new features, fixes, and improvements. Some of these applications are updated more than once a week [135]. In the last decade an academic field has rapidly risen with this technological growth; Search Based Software Engineering (SBSE) and the related field of Search Based Software Testing (SBST) emerged as well. The applicability of this research has not only inspired the next generation of researchers but also grabbed the attention of the software industry. Many of the methods and techniques stemming from SBSE have been consolidated in standard and best practices, like e.g. mutation testing.

Ambitious ideas and industrial interest have since made the realisation of projects like DAASE [1] possible. The ideas and research in this thesis aim to push further the boundaries of what can be done with search-based approach to programming. Ideally we would like to produce autonomous software, that maintains itself and dynamically adapts to its environment but without making it sentient.

The main goals of this work are:

- to provide researchers with empirical evidence for the effectiveness of GI and convince those that are still sceptical of the merits of GI for real-world live systems;

1 http://daase.cs.ucl.ac.uk/

- to give developers tools to better cope with the challenges in today's software environment;
- to increase the understanding of where and when GI is most effective.

1.2 GENETIC IMPROVEMENT

GI [154] is a growing area within SBSE [72, 76] which uses computational search methods to improve existing software. Despite its growth within academic research the practical usage of GI has not yet followed. Like with many SBSE methodologies, the software industry needs an incubation period for new ideas where practitioners come to trust in outcomes and see those ideas as cost effective solutions. GI is in the ideal position to shorten that period for the latter as it presents a considerable cost decrease for the software lifecycle's often most expensive part: maintenance [121, 66]. There are examples of software improved by GI being used and publicly available [113]. This is impressive considering how young GI is as a field. In time it can be anticipated that we will see tools emerging that utilise current advances in GI for various improvements during different stages of development: from early coding, where programmers might want to statically monitor the performance of their work, to maintenance and bug fixing [5]; automatically adding new functionality [132]; and adding new hardware compatibility [105].

Traditionally the application of GI has been done offline where the original program is copied, improved in a lab, and then the researchers have to convince developers to include the improvements in later releases, if the researchers have the patience. However GI's ultimate goal must be self-improving and self-adaptive systems [74, 35, 42]. They have to include minimal manual effort by the developers to truly optimise software maintenance costs. Current research into self-improving systems, consists of early concepts [188, 31], identifying applicable ideas [79], and highly specified but truly dynamic approach [213]. Adaptive systems can be altered with changes in their environment, like turning of the wifi feature on a mobile phone the battery is near empty. Those systems become dynamic adaptive when they do those alterations automatically and as a response to the changes in the environment. The research and methods are slowly approaching dynamic adaptive systems but the biggest obstacle is getting the existing results to market. To achieve that we need to take it in steps, starting by providing developers with tools before advancing to autonomous software. DAASE (Dynamic Adaptive Automated Software Engineering)[2] is a multiple site project that seeks to use computational search methods to advance software engineering towards automation and hence software towards dynamic adaptivity.

2 The author is funded by the EPSRC through the DAASE project, Grant EP/J017515/1

1.3 thesis structure

The thesis is comprised of four parts:

- Part i (this part) details the motivation and explains the hypothesis along with the research questions;
- Part ii gives an overview of current relevant literature;
- Part iii is the main body of work that represents my contribution to the field; and
- Part iv summarizes the results, contributions, flaws, and implications for future work.

2 HYPOTHESES

2.1 INTRODUCTION

This chapter defines the main Hypothesis that is central to the research. It will be separated into three sub hypotheses, each with an accompanying research question that I will approach individually.

The hypothesis is expressed in a specific way that can be demonstrated to likely be true via experimental results that support each claim. However, as with the majority of scientific processes, we cannot fully prove the hypothesis. I will present evidence and examples to support or challenge our supposition. The answers to the three research questions will be collected in Chapter 9 and used to support broader conclusions.

2.2 CENTRAL HYPOTHESIS

The underlying hypothesis of the work presented here is:

Thesis Hypothesis *GI can be used autonomously in software systems to decrease maintenance cost and assist developers to improve various measurable properties of the software.*

This hypothesis will be supported or refuted by addressing three paths of research and their associated research questions.

- For research question 1, I explore the search landscape of bug fixing in small Python programs.
- For research question 2, I explore the search landscape of execution time improvements in larger C/C++ software.
- For research question 3, I integrate a GI component into a live system, written in Python, so to make it semi-dynamically adaptive.

2.2.1 *Small Programs' Landscapes*

Research Question 1 *Can GI fix bugs in small programs, written in a dynamically typed language and how does the GI interact with the search space?*

Smaller programs should present narrower search spaces and therefore, if an improved version of the program exists, it should reside in the neighbourhood of the current one. When the improvement is functional, like fixing a single bug, the *competent programmer hypothesis* [2, 45] argues that the correct implementation is just a few edits away. The hypothesis states that programmers release programs that are, if not correct then nearly correct. To make them fully correct will only require few changes to the code, such as inserting or moving a single statement, or changing a boolean expression from $<$ to $<=$. With this assumption I explore the landscape around a correct implementation and look at how fitness and GI edits are related where the fitness is defined as *number of test cases passed*. This analysis will be conducted on programs written in the Python language, which is dynamically typed so the structure is less rigid than in statically typed languages. That might cause unexpected behaviour during runtime, either beneficial or harmful. This analysis will provide insight into the search process and serve as a step in GI's development.

2.2.2 *Execution Time Improvements*

Research Question 2 *Is it possible to implement GI in such way to be portable between different source languages and different objectives and how does changing these impact on the search?*

GI implementations have traditionally been used to target a specific language syntax and larger programs where Lines of Code (LOC) is generally $> 1K$. To be considered portable the user must be able to apply the same implementation on at least two different programming languages. The user should only have to provide the original source code and an application programming interface to evaluate variations of the software. Inherent in that statement is that the objectives are merely an input to the GI framework.

To answer this question I apply the previously implemented GI framework used in RQ 1 to an existing C/C++ software that is the order of magnitude larger than the targeted programs from earlier. The formerly defined objective to *maximise the number of test cases passed* will be redefined to *minimise execution time*. In addition to actually trying to improve, I will observe the search neighbourhood of all trajectories the GI takes.

Answering this question will strengthen our confidence that GI can be used in an environment of different programming languages. Furthermore, we will be presented with evidence that the implementation will scale up to larger applications as well as if various properties can be improved with GI.

2.2.3 *Dynamic Adaptive Software*

Research Question 3 *Can GI be integrated into an already live system to identify bugs and suggest fixes, thus assisting developers with maintenance?*

Central to the thesis hypothesis is the question if GI can be used autonomously to maintain software. In a sense the GI works autonomously as in, while it is running there is no human in the loop optimisation involved. However, most implementations require the user to set up a testing environment and adjust it for every application, so it is not fully autonomous. I will demonstrate a GI system that works on its own to recommend patches to the developers that can fix bugs that have been discovered with regular usage. Research question 3 asks specifically if a GI framework can be embedded in an already live and running system. To answer this question I identified a system that is in active development while also in use and simply implanted the previously mentioned GI framework. Many programs have periods of high use (e.g. office equipment during daytime) and periods of low use (e.g. office equipment during nighttime). We aim to exploit the low utilisation periods to employ GI to fix bugs found during intervals of high utilisation. This would therefore make full use of the hardware causing little inconvenience to the users. The GI would be activated during periods when the CPU requirements are low if bugs had been detected during higher load periods. After each low load interval that activates the GI the human programmer would be presented with a choice of fixes, if any are found.

2.3 SUMMARY

I have defined the research direction of this thesis with a central hypothesis and briefly explained the three sub hypotheses and questions that will lead the research forward. I will attempt to answer questions 1, 2 and 3 in Chapters 6, 7 and 8 respectively. Chapter 9 will then draw together these answers to provide evidence to support the central hypothesis. First, however, Part ii will review relevant literature about GI and SBSE.

Part II

LITERATURE REVIEW

3 GENETIC IMPROVEMENT OF SOFTWARE

3.1 INTRODUCTION

GI is an emerging research area that has been rapidly expanding in recent years. The first GI survey [154] describes GI as a methodology that uses automated search to improve existing software. It establishes four criteria that define core GI work:

1. The search for improvements uses metaheuristic methods [128];
2. the search can produce non-semantics-preserving software variants;
3. the GI improvement framework reuses existing software;
4. the modified software is an improvement over the original software for a given criteria.

The GI paradigm rests on the shoulders of *The plastic surgery hypothesis*, which is based on the insight that new code added to existing software can often be constructed from fragments that are already in the source [12]. So we do not need to generate new code but can effectively cut and paste code from other parts of the program.

Software in general has always been, and still is persistently being improved [15, 162]. Demands for faster, smaller and more versatile devices, mobile or otherwise are constantly on the rise. Hardware is being developed and improved to keep up with demands and produce profit for the tech industry. The software needs therefore to keep up with the hardware, changing architecture and protocols. If a software company wants to remain competitive it is not sufficient to write a program, release it and forget about it. It has to be continuously maintained and improved in sync with modern technology. Moreover coding a program in today's industry technical environment is not a straight forward venture. For software to survive it needs to be available and perform consistently on multiple platforms. Manually optimising a program for multiple competing objectives such as speed, memory efficiency and bandwidth usage simultaneously, is nigh impossible or would at least be extensively time consuming.

The concept of automatic programming [11], where the computer is used to alleviate workload dates as far back as the 1960s and 70s from two related research fields: program transformation [41, 32] and program synthesis [131, 126]. GI draws from both fields and extends them by applying the methods to extant software and allowing transformations that

do not preserve semantics by mainly using software testing for guidance [173, 2, 33, 4]. GI relies heavily on recent progress in SBST [138], namely automatic test data generation [48, 137, 99, 171] Much of the SBST research has been about the generation of test data such as Beyene et al. [17] where they generated string test data. There are many sophisticated ways of generating new tests. Feldt and Poulding [55, 156] for example use Bolzman samplers [48] to uniformly sample data with specific properties. A random search algorithm can also be replaced with alternatives such as hill-climbing [185], an evolutionary method [99] or optimisation algorithms more generally associated with operation research [56]. Some GI researchers have also opted to restrict the search with more formal methods of testing [81]. The essential objective of most test data generation processes is maximum code coverage which is a measurement of how large proportion of the code is being tested. This measurement can be defined in multiple ways which all have their merits and faults. The common denominator for all the definitions is that they tell what parts of the code the execution visits for each test, whether it is counted in lines, statements, or branches.

GI also borrows from other strands of research, such as program slicing [18, 71, 52, 73, 19, 199] which, in its simplest form, is the act of extracting functionality of interest from a larger piece of software. The idea is to extract a fully functioning program that contains a subset of the functionality of the original. It is analogous to taking a specific slice or part from a cake it is still a cake, you just wanted the frosting. The influence of program slicing is evident in the work by Barr et al. [13] about auto transplantation. While the extracted functionality was perhaps not a self-contained program, it was a specific functionality from another program.

Possibly the most notable feature that defines most work on GI is the search element, which historically has been GP [155] but can be any metaheuristic search [28]. The unifying identity of GI research is being a part of SBSE [72, 76] which also encompasses SBST [138], requirements [216], predictive modelling [3], software design [161] and management [57].

In this chapter I will review current and historical GI literature. I will include some work that does not meet one or two of the criteria here above but either inspired, or was inspired by, core GI work. The review is divided into overview of different types of representations (Section 3.2), description of the search methodologies that are used (Section 3.3), and overview of improved properties (functional in Section 3.4.1, and non-functional in Section 3.4.2).

3.2 REPRESENTATION OF CHANGES TO PROGRAMS

One of the main challenges that GI practitioners face when changing program code is the choice of representation. The representation or the formulation of the problem dictates what options are available for the search and vice versa. This idea is at the centre of the meta-heuristics search [28] where domain knowledge guides the implementations.

GI is closely related to hyper-heuristics [29] where the search space and the solution space are separated. A solution space is where a single point represents a single solution to a problem, e.g. in a routing problem the solution space would be represented by some indication of a single route per solution. Traditionally, the search algorithms will transform one route to another and thus be searching in the plane of solutions. The search space is where the algorithm makes transformations from one point to another and in solution space these points are unique solutions to the problem. By abstracting one level above the solution space, then you are searching for algorithms that represent a method to search in the solution space. This abstraction has separated the search and solution space by distinguishing between where the search makes transformations and the where eventual solutions are found. Hyper-heuristics searches in the space of heuristics that exploit properties of the solution space while GI searches in the space of programs that can be used in multiple different ways. The choice of representations is therefore paramount to be able to effectively map the changes in the search space to changes in the solution space.

Representation implementations for GI are a non-trivial task with many influencing factors and categorising them is difficult. There are numerous different examples in the literature and in the following subsections I list four different perspectives when deciding on representation:

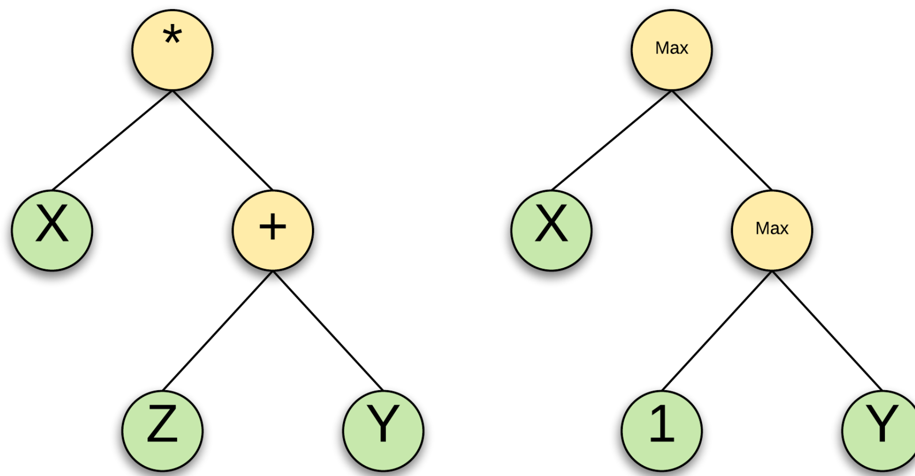
- Whole programs;
- patches or edits to programs;
- programming language differences;
- and everything else that does not fit in with the evolutionary paradigm.

3.2.1 *Whole Programs*

ABSTRACT SYNTAX TREES When manipulating source code it is an intuitive choice to treat the program as a single entity and keep different versions of it. Before the time of version control services, like GitHub [62], the first obvious way for the common programmer to keep track of changes was to rename the current working source code file with enumerating suffix and work with the entire files. Naturally, this perspective influences the choice of evolving whole programs. However there are different ways to represent whole programs.

As GI can arguably be traced back to work on GP [168] the majority of earlier independent implementations have kept the "evolve whole programs" approach. Traditional GP evolves tree structures like in Figure 3.1 which is principally the same as an Abstract Syntax Tree (AST) representation. Each node is either a function if it has children nodes or a terminal when it is at the end of a branch. To manipulate the source code when it has been parsed into these kinds of trees one would change or rearrange the nodes before parsing the trees back to

source code. It is an intuitive method of simplifying the process of changing the code so that the computer can follow the rules and restrictions put in place by the human. Changes to a single node are called mutations in this context and they usually involve changing it for an equivalent node that preserves syntax. Syntactically equivalent nodes accept and return the same data type from children nodes and to their parent nodes. It is called crossover when two (or more) program trees are rearranged together to form new nodes. A simple crossover chooses a node in both trees and swaps the branches from that node (see Figure 3.2).



(a) A simple arithmetic function $X \times (Z + Y)$ (b) A simple conditional function $\max(X, \max(1, Y))$

Figure 3.1: Traditional GP tree representations. Nodes $*$,$+$ and max are called functions while X, Y, Z and 1 are terminals.

Possibly the earliest examples of GI using this AST representation is in the Paragen system [168, 166] and later the updated version, Paragen II [193]. They defined each line in an existing program as a terminal, introduced parallel specific functions, and then searched for a rearrangement to maximise parallelism. These functions controlled if the subsequent branches would be executed in parallel or not. The first version disassembled the target program completely and tried to rebuild it from scratch, the later version started with the original program as a template and was therefore much faster. Additionally Paragen II could perform specific loop transformations. However they both produced the same end results but Paragen II's output could be proved to have the same functionality as the original program. This was possible because the the parallel transformation functions were specifically engineered to preserve data dependencies and semantics.

A bit over a decade later the pioneering GI work of White et al. [204, 201, 202] and Arcuri et al. [6, 7, 9, 8] used this same representation with all the traditional GP mutation and crossover operators. The evolved programs where small in size (maximum 100 LOC) e.g.

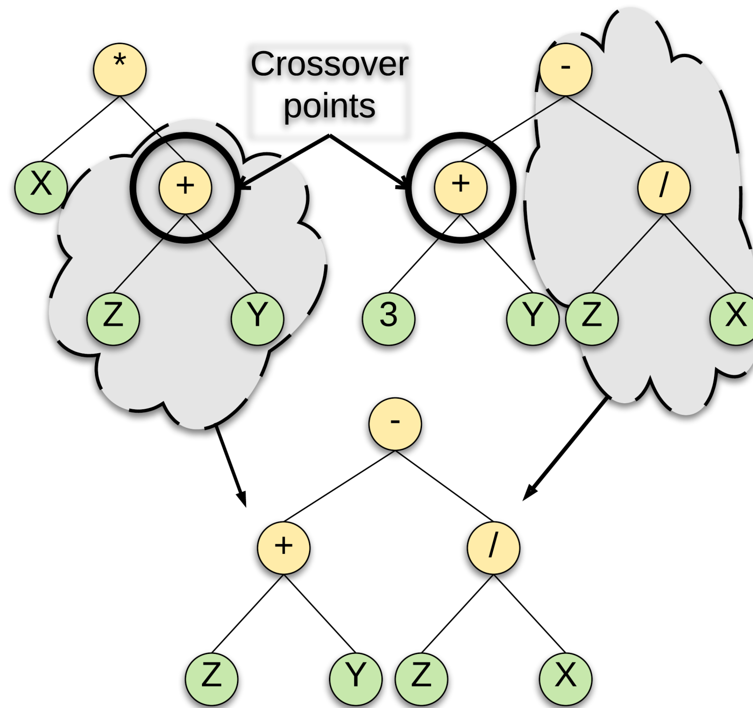


Figure 3.2: A simple crossover of two trees. $X \times (Z+Y)$ and $(3+Y)-(Z/X)$ become $(Z+Y)-(Z/X)$. The clouds constitute the two parts that are in the final program.

pseudo random number generator [201], various mathematical and sort functions [202], and triangle classification [9]. Their work is of particular interest to GI for their use of seeding existing programs to improve non-functional properties [9].

Early versions of the automatic bug fixing framework GenProg [122] adapted the same tree-based representation as did its predecessors [50, 58, 143, 197, 196]. Although the actual search was made with full representations of the source code as trees, the suggested fixes were represented as the programmatic difference between the original program and the resulting fixed variant. To minimise the size of the difference and to increase readability they used delta debugging methods [58] that search sequentially through each change to the tree and remove changes that do not affect the fitness [196]. Since the programs they were targeting were larger than the traditional GP program the trees were very large to begin with. To counter the problems of the huge search space they weighted the trees to focus the search to paths where the bugs had been located [143]. Additionally they improved the efficiency and precision of fitness function by sampling the test suite and pre-processing the instrumented buggy program to determine a set of predicates and associated conditional probabilities. [50]. Successful program manipulations were mainly made with inserting or deleting nodes in the trees [159, 141]

More examples of full GP tree evolution include Cody-Kenny et al.'s execution time performance improvements [37, 38, 39]. They tackle the scaling problem in a similar manner as Le Goues at al. [122], by biasing the search towards sub-trees of interest [37] but with a dynamic weighting scheme. Kocsis et al. [91] also evolved whole HashCode functions and then inserted them to the larger system, Apache Hadoop platform. Their approach did not in fact seed the evolution with an existing program. It evolved a function in isolation with terminals that were programmatically inferred from the larger system and then manually merged with it post-hoc. An equivalent idea was used by Swan et al. [188] in Gen-O-Fix, a framework that can be embedded into software that runs on a Java Virtual Machine to make parts of it (or the entire program) dynamically adaptable. Furthermore Brownlee et al. [21] also adopted the same idea for evolving a median approximation to replace the pivot function in a quicksort implementation. Mrazek et al. [140] also evolved approximation to the median function but for the purpose of using it in low-cost, mass produced microcontrollers. Wilkerson et al. [205, 206] designed their system for program repair, called CASC, to evolve ASTs in XML form. Yet another example of tree based representation is the work of Sitthi-Amorn et al. [183], where they simplified a set of pixel shaders for better performance. The size of the shader programs ranged from 28 to 962 LOC.

GP's tree based representation and visualisation is comprehensible for the human programmer while also being easily converted into executable code, like e.g. the Lisp expressions used by Koza et al. [93, 95, 97, 94, 96]. However GP has historically not been able to evolve large programs and has only been successful with smaller functions [147, 202, 164] up to programs that are about 100 LOC at the most [202] when evolving from scratch. Different representations of whole program evolution have therefore been developed and tested. Later versions of GenProg for an example have moved away from whole program representations [121]. Additionally there are those that discard the need for the programmer to understand in depth what the changes are doing, at least while the changes are being made.

machine code representations Orlov et al. [147, 148] store a population of full programs that are represented directly as Java bytecode, arguing that the purpose of source code is to be readable for the programmer. This readability directly impedes ease of automation of the code manipulation but is reduced after being compiled. The population of the evolving programs are sequences of annotated bytecode which opens up opportunities for more straightforward crossover operators than having to worry about branches as well as types.

Schulte et al. [175, 176] base their work around a similar idea but also included Windows x86 assembly code in their linear representation. They search by permutating, deleting and copying the set of instructions in the compiled code, which each represent a single statement from the source code. The focus of their target programs lies in embedded software [177], where efficiency, both energy and memory, is of the utmost importance. Therefore the changes

cannot add too much to the memory footprint of the improved program and to this end they define an extra objective of minimising the size difference from the original. They also use delta debugging to strip the changes of neutral changes [174]. In addition, this objective has the effect of minimising the risk of introducing regression, new bugs or security vulnerabilities [178]. Landsborough et al. [101] adapted the same notion but with the strict objective of reducing the program to minimal required features.

MISCELLANEOUS REPRESENTATIONS Other whole program representations that are less used in GI include marking all loop definition in the source code while it is being compiled [207]. Each loop is then gets assigned to an element in a sequence of characters that can take values from a finite set. Each character specifies what loop transformation should be implemented instead of the sequential loop in the original program. With this representation the search problem becomes a static-length Genetic Algorithm (GA) genome. Williams [208] has used this to develop a compiler that automatically parallelises loops.

The work of Kocsis et al. [89, 90] treats the program improvement task as a mathematical problem and represents programs as algebraic functions. Providing semantic preserving transformations and using deterministic proof search with finite number of rules to construct code segments. This is very effective but has rather limited applicability because behaviour of software is sometimes not deterministic and developers are humans so their code will probably not always follow strict programming rules. Mechtaev et al. [139] do similar work where the represent the whole program as a trace formula and use satisfiability solvers to repair bugs like in SemiFix by Nguyen et al. [142].

3.2.2 *Patches*

Apart from few exceptions [183] and special cases [101], the complete program representation has not been shown to generally scale to larger programs [167, 201]. However, with GI's speciality of working on existing software we can move up one level and instead of searching in program space we search in the space of changes. Evolving patches to software greatly reduces the memory requirements since we only have to keep a single copy of the original program. This was the main motive for moving GenProg away from full AST representations as mentioned in the previous section. In their effort to identify monetary costs of automatic bug fixing on a cloud out of 105 defects considered 36 populations of 40-80 ASTs were too much for the memory allocated (1.7 GB) to each node in the cloud [121, 123, 120]. When inspecting human written patches for the same defects, half of them were 25 lines or less and therefore two variants might differ by no more than 50 lines while everything else is identical. The patch representation adopted by GenProg is a sequence of edits to the AST, where each edit is a tuple containing an operation and node numbers. An example would be

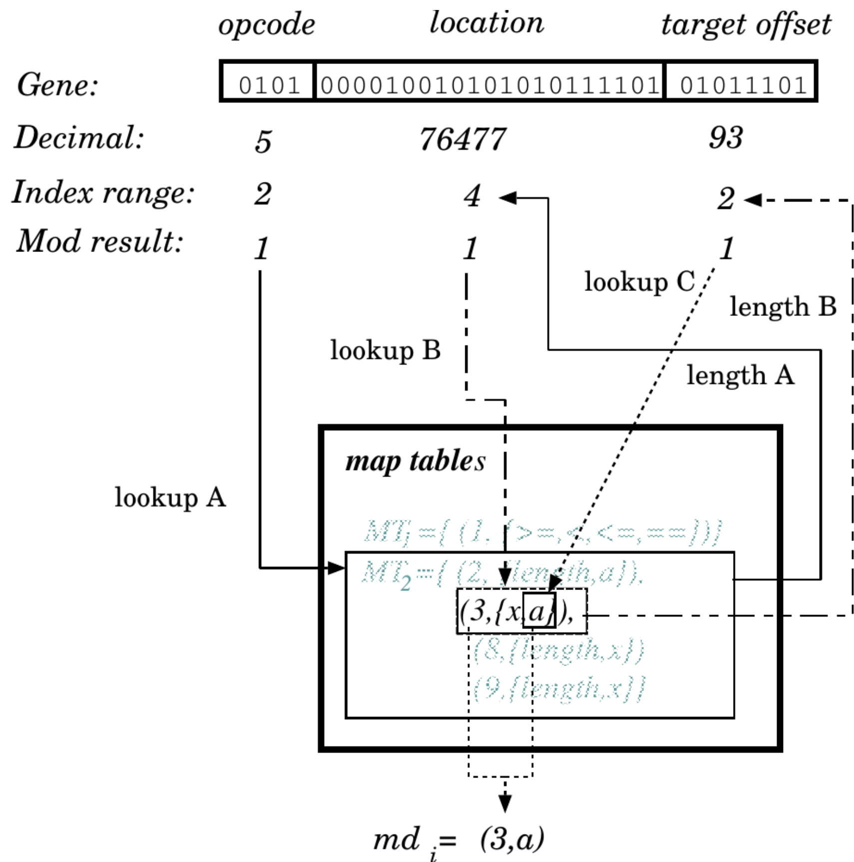


Figure 3.3: Three-stage lookup of a modification table from a genome as represented in Ackling et al. [1]. The image is taken from the same publication.

`Replace(81,41)` that replaces node 81 with node 41. Mutations can change a single edit in a list while crossover concatenates two lists and then randomly removing edits. Weimer et al. [198] use the same representation but do not use any mutations or crossover and instead enumerate all possible single edit changes to the target program. They were able to apply their technique to programs up to 3 million LOC by:

- limiting the search to only single edit changes;
- dynamically prioritising tests and escaping the evaluation when variants failed a test case.

Subsequently, all GenProg derived systems and frameworks [157, 133, 158, 119, 145] do not change the fundamentals of the representation by still evolving patches that change the AST. Qi et al. [157, 158] implement GenProg with random search and test case prioritisation. However, Oliveira et al. extend the representation into finer grained building blocks to allow partial recombinations and improved crossover operator. Martinez et al.[133] package GenProg together with two other automatic repair frameworks (PAR [88] and mutation based approach [43]) in a generic software package called ASTOR.

Ackling et al. [1] also extend the GenProg representation to allow for GA search with variable-length genome. Each edit is a 32-bit binary sequence and is designed to map changes to a predefined lookup table of possible values for each location. An example of such mapping can be seen in Figure 3.3. The gene is described in the top row and it is divided into three parts that correspond with the three stages of the lookup process:

OPCODE is the first four bits that correspond to the type of modification and indicate which modification table should be looked up.

LOCATION a 20-bit sequence that points to where in the AST the change should occur and what row in the modification should be looked up

TARGET OFFSET indicates the target value and the column in the modification table.

Similar methods of lookup and genome representations are used in the work of Risco-Martín et al. [165] and Fatiregun et al. [51]. The main differences are what each gene represents but both alternatives eliminate the need for decoding: in Risco-Martín's work a gene is a single integer that points to a Backus-Naur Form (BNF) grammar lookup table but in Fatiregun's work it points to a list of transformation rules. Additionally, Fatiregun et al.'s work evolve a fixed length sequence, with limited number of possible transformations.

Other work for improving software that use similar genome-based patch representations include the deep parameter tuning paradigm. It is a strand of GI research that has risen from the SBST [138, 80] field of mutation based testing [2, 185]. While automatic parameter tuning searches for the optimal configuration of available parameters for the user, deep parameter tuning accesses configurations that are not originally available. A finite number of constant definitions in the source code are connected to a single gene in the sequence. Each value in the genome then corresponds directly to what value the constant should take, essentially changing the constant into a variable. These constants are usually what programmers call "magic numbers", indicating that these are fixed numerical values that were either arbitrarily decided or with some domain knowledge [203]. However, there is no requirement that these constants need to be primitive data types like e.g. strings, sequences, or numbers [43]. Wu et al. [210, 211] added nine supplementary operator classes (Table 3.1) that represented C and Java fundamental operators. Bruce et al. [25] used that same representation in their work of improving the execution time performance of OpenCV's Viola-Jones face detection algorithm while Sohn et al. [184] parameterised a single variable of five template matching algorithms in OpenCV. Improving imaging software or visual properties other programs' of graphical user interface has made considerable use of this GA related genome representation. Li et al. [125] used it to represent pairs of HTML elements when minimising energy usage of smartphones screens displaying websites. As did Linares-Vásquez et al. [127] but they targeted graphical user interface of Android apps and each genome corresponded to a bag-of-colour-pixels.

Table 3.1: Additional operator classes defined as parameter values for deep parameter tuning representation [210]

Description	Operations
Arithmetic operators	$+, -, *, /, \%$
Arithmetic assignments	$+=, -=, *=, /=,$
Logical constant negation	$expr$, $!expr$
Increment/decrement	$++, --$
Logical operators	&&, $\vert\vert$
Logical negation	x op y, x op $!y$
	$!x$ op y, $!(x$ op $y)$
Relational operators	$<, >, <=, >=, ==$
Bitwise assignments	$\& =, \vert =$
Bitwise operators	$\&, \vert$

A variation of deep parameter tuning was introduced by Burles et al. [30]. They specifically targeted Guava container object creations in Google's Guava library [14] to ensure semantic-preserving transformations for improving the energy consumption. The process iterates through the AST of the code, finds all nodes which created such containers, and records them along with information on possible variations in a sequence object. Orlov et al.'s [146] representation also took the form of a sequence but it contained a subset of Megavac instruction set. Megavac is an 18.000 LOC evolutionary-computation platform that uses steady-state linear genetic programming and it has a large set of parameters.

Grammar has been extensively used in edit list representations by Langdon et al. [78, 84, 104, 102, 103, 105, 106, 107, 109, 113, 113, 112, 111, 117, 115, 116, 108, 110, 114, 24, 153, 152] for various tasks. They abandon any intermediate rendition of the genome and instead evolve the lists as a variable-length sequences of texts which are directly readable for the programmer. The mutations and crossover operators that are mostly used, simply append a single edit or the whole genome of another edit list. This causes bloating which is reduced post-hoc with a delta debugging method as depicted in Figure 3.4.

This text based edit list representation has been adapted by the work presented in this thesis and will be explained in detail in Chapter 5.

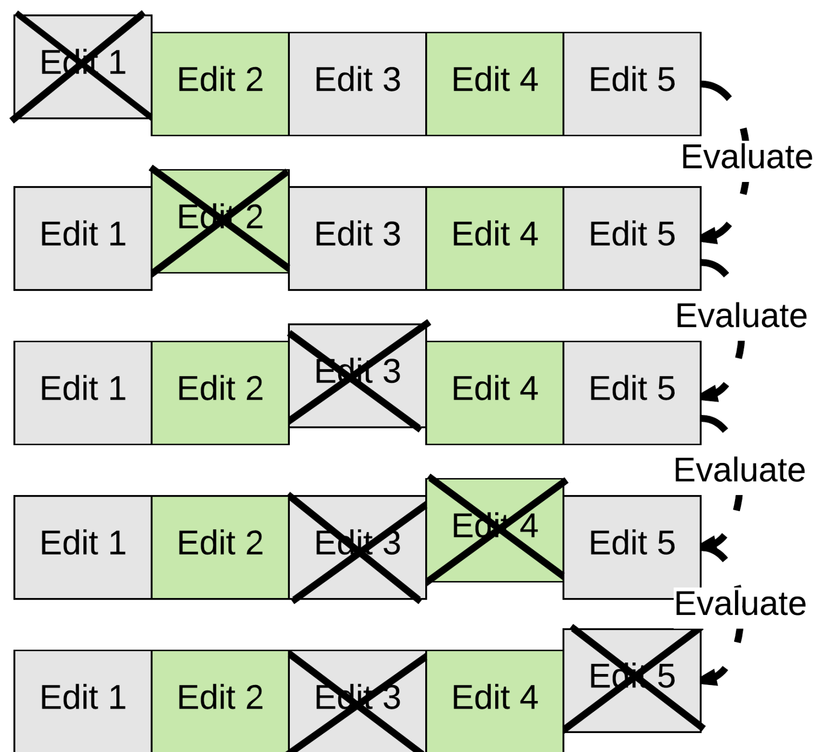


Figure 3.4: An example of a delta debugging process as used in [102]. Process iterates through the edit list and removes a single edit at a time, then evaluates the reduced edit list and if the removal of the edit has no effect on fitness it is not reinserted.

3.2.3 *Programming Languages Targeted with GI*

In today's environment there is a plethora of programming languages available to software developers to choose from. Each language has its specialities, features and properties that usually are designed with a certain programming paradigm in mind. With the different languages come various ways to manipulate the source code, useful techniques, and pitfalls.

The impression got while reading through the GI literature is that the choice of problems that are tackled depend upon each researcher's preference for a specific programming language or type of languages which in turn, can heavily influence the choice of representation.

3.2.3.1 *C Languages*

The majority of GI research target programs written in C or C++ . The two programming languages are quite closely related since C++ is descendent of C, although they are not entirely compatible. It is unclear from the literature why these languages seem to be the choices for most GI researchers. However, the most likely explanation might just be that both languages have been fairly popular for a long time. C++ is standardised by International Organisation of Standardisation (ISO/IEC 14882:2014) while C is standardised by American National Standard Institute (ANSI X3J11) as well as ISO (ISO/IEC JTC1/SC22/WG14). They are both statically typed, can be compiled with the same compilers and, in general, produce relatively efficient program code in terms of execution time compared to other languages. Additionally, a compiled program in either language has small enough memory footprint to be ideally suitable for low-resource devices. Which is possibly the main reason for White et al.'s choice in their early examples of GI [201, 204, 202]. Weimer et al. chose target programs written in C/C++ for GenProg to target, because of the availability, at the time, of legacy C programs with known bugs [197, 196, 58] and existing test suites [143, 50]. Their later work is somewhat influenced by this earlier choice and also targets the same programming language [121, 120, 122] and with the same representation [198, 123]. The work that builds on GenProg's success have not added any language specific insight [119, 145, 157, 158] so it assumed that their choice of target language is based purely on convenience.

Wilkerson et al.'s choice of targeting C/C++ [205] was controlled by their choice of tool for parsing the source code into an XML tree representation. The same tool can also parse Java programs but their definition of node classes were specific to C [206]. Similarly, strategic search definitions affected the choice of Jin et al. when specifically targeting atomicity violations [85] and general concurrency bugs in C programs [86]. Gao et al. [61] combined a static and dynamic buffer overflow analysis with a finite set of predefined C specific fixes that were mined from software development sources which is analogous to what Tan et al. did [189].

Sidiroglou-Douskos et al. implemented their loop perforation tool [179] in the context of a compiler framework [118] that was setup to compile C source code. Their later work

on transferring code segments between programs also targeted C code [180] (this time compiled), however the choice was a consequence of their post-processing tool to translate the changes back into source code. Their code manipulation approach could have been applied to any source language that compiles into binary. The same applies to Schulte et al.'s work [176], where they manipulate x86 code from C compiler but also other types of machine code [177]. The ease of representing assembly instructions as GA genome is most likely what motivated them to target this language [174]. Landsborough et al. [101] were also persuaded by this property to use binary images for removing unused or unnecessary features from programs. The GA sequence representation of a compiled program makes it easy for the search algorithm to turn off or on arbitrarily large segments of code. Moreover, it mitigates various implementations of mutation and crossover operators. Both languages (C and C++) seem to attract work on deep parameter tuning with the GA representation, both offline optimisation [25, 184, 211, 210] and by turning static configuration parameters in to dynamic controls [82]. Whether that is due to any property of the languages or just by convenience is unclear.

The GISMOE framework [75] works by converting C source code into BNF grammar and evolving edit list representations initially used for this task by Langdon et al. [108] when porting a compression algorithm to C++ Cuda code for parallelisation. The vast expansion of this strands of research, and possibly also because Langdon has released the source code of the framework[1], has guided much of the current literature towards targeting legacy C programs [129, 24, 26, 13, 132, 78, 153] and using grammar [165].

3.2.3.2 *Languages Running on Java Virtual Machines*

Java is a static typed language and it is inspired by C/C++ which it shares much syntax with. The most significant factor that contributes to popularity of Java based languages in GI is not any syntax or properties but the Java Virtual Machine (JVM). It is an emulator that interprets Java bytecode and can execute it on any computer architecture. Orlov et al. have made considerable use of this in their evolutionary framework, FINCH [148]. It can evolve unrestricted programs written in any language that compiles to Java bytecode. They exploited the type system of the JVM to perform various checks on the stack, variables, and control flow to ensure compatibility when moving or copying instructions from one place to another [147].

Yeboah-Antwi et al. use the JVM's architecture to inject already running programs with an ability to dynamically adapt to environmental changes at runtime. The framework, called ECSELR, embeds itself into the target program [213]. Although JVM allows programs to run on any operating system ECSELR is limited to a specific architecture as it consist of native system functions and data structures [214]. This makes it possible for it to reside in the same native operating system memory address as the target program.

1 http://www0.cs.ucl.ac.uk/staff/w.langdon/WBL_papers.html#langdon:2010:cigpu

Swan et al. have also made use of the JVM, more specifically various properties of Scala. It is a programming language that was primarily designed to address some shortcomings of Java, such as the expressiveness of the reflection capabilities. They have extensively used it in developing their many frameworks and packages for GI [31]:

- Gen-O-Fix [188], uses reflection to yield source-to-source transformations. Runs in parallel with the targeted program and continuously improves it.
- TEMPLAR [187], a templating framework for hyper-heuristics. It uses the higher-order functions abilities of Scala, where functions can be treated as variables, to simplify rapid construction of algorithms.
- HYLAS [92], a tool that optimises Apache Spark queries. It also uses the Scala reflection properties to manipulate the AST of a running program on a JVM.
- AntBox [31], a construction mechanism for domain specific language. It takes in object's properties as constraints and builds it with Ant Programming [20].
- PolyFunc [90], a Scala specific tool to facilitate semantics-preserving transformations between data types that are of the same class. Scala's the reflection properties make this possible.
- Hyperion^2 [22], a meta- and hyper-heuristic template framework for researchers.

Additional work by Swan et al. have also targeted Java applications, some of which are possibly because of preference [21, 91, 30, 203] and other because of the type system [89]. Much of Swan et al.'s work have used deterministic or guaranteed semantic preserving methods by formulating the transformations as algebraic and constraint functions. DeMarco et al. used a similar tactic when developing their Java code repair system, NOPOL, which targets *if* conditions [44].

JVM languages are currently some of the most popular programming languages, used in numerous client-server web applications and Java is the main language used to develop Android applications for mobile devices. Therefore it has been an object of target when optimising energy efficiency; reducing energy usage by the display by reassigning colours in HTML elements [125] and other applications [127] as well as dynamically controlling functionality on mobile phones [36].

The output of Arcuri et al.'s work on automatic bug fixing [9, 8, 6, 7] repaired Java source code. Convenience is suspected to have motivated the choice, since the evolutionary framework used in their work (ECJ) is written in Java. A common pattern of choosing which programming language to target seems to be the available tools to the researchers. Like Dallmeier et al. [40] who implemented their search algorithm to use ASM [98], a framework for manipulating Java bytecode.

3.2.3.3 *Python*

The Python programming language [192] and was initially intended to serve as a prototyping language. It has since been adopted by many developers and the open source community of users. The language provides interactivity and object-oriented programming with dynamic typing and high level data types. However, despite its popularity it has still not been a major focus of study in GI. Apart from the research presented in this Thesis and related publications [66, 67, 69, 65] there are only two contributions that specifically target Python source code. Ackling et al. adapted GenProg's philosophy and methods into pyEDB [1], a software repair tool that evolves patches. It uses an open source package to construct an AST that it then operates on. A more recent example is presented by Jia et al. [84] where they grow and graft a Python code into the well known and widely used web framework: Django. The process involves generating (*growing*) a function with GP in isolation and then using GI to insert (*graft*) it into the original program.

Hopefully, with the extending popularity of the language more examples of GI for and about Python will emerge. It would be interesting to see what effects the dynamic typing, the flexibility, and the interactivity of the language will have on automatic programming and GI results. The comparison with statically typed languages such as Java and C will be of particular interest.

3.2.3.4 *Miscellaneous Programming Languages*

The amount of different programming languages available is colossal. While C/C++ , Java, Python, and derived languages are possibly the most extensively used in today's technological environment, they are still just a small subset. So there are a few examples of GI that target other languages. Some of these samples date back to the nineties, which is not surprising given that the popularity of the aforementioned favoured languages has been rapidly rising in the last decade. The earliest work is perhaps on automatic parallelisation by Ryan et al. [168, 193, 169, 166, 167] and the integration of search for parallelism into compilers by Williams et al. [207, 208]. Ryan et al. targeted Fortran source code with their Paragen system and a traditional GP tree structure representation. Williams et al. also targeted Fortran but while it was being compiled.

Of popular programming languages that have not been as prevalent as GI targets PHP is probably one of the wider spread as a server side web application. There is only a single example of in the GI literature describing automatic repair of PHP code. Samimi et al. [172] pursued faulty HTML generation statements by expressing the input-output pairs of string literals as satisfiability problem. Regular Expressions are a way to express patterns in sequences of strings. They have a syntax, albeit with variations depending on context, and could theoretically speaking be seen as programming languages of sort since they are essentially

executed when being used. Cody-Kenny et al. utilised this property to improve the efficiency of Perl based-regular expression patterns [39].

With the growing trend of embedding all appliances and electronic devices with digital technology and connectivity, more and more software is deployed as firmware. This provides limited abilities to maintain or repair software in these devices unless the hardware producer directly supplies updates. This has sparked some interesting paths for GI to tackle problems where the human programmer cannot possibly understand the code that is being manipulated. More specifically, targeting binaries with unknown source code like Schulte et al. [178] did when repairing security vulnerabilities in off the shelf router firmware. Mrazek et al. evolved median functions for microcontrollers [140] allowing a compromise in accuracy. Perkins et al. also targeted binaries, although the origins of the source were known and the application domain was not an embedded system [151]. Nonetheless, they developed a system that patches security vulnerabilities in Windows x86 binaries to e.g. counteract ongoing external attacks.

Other languages targeted by GI are quite obscure examples that are either only in use by a small group or specifically devised for the purpose of the problem at hand.

- The Eiffel programming language with a design by contract paradigm as its central philosophy. Wei, Yu et al. [195, 150] developed automatic repair methods for Eiffel
- WSL: wide spectrum language, simultaneously high and low level programming language. Fatiregun et al. evolved transformation sequences specifically aimed at WSL source code.
- Katz and Peled have developed a program synthesis tool that produces code for mutual exclusion algorithm. From their work it is unclear what language the code is in or from what it draws inspirations [87].

3.2.4 *Non-Evolutionary Representations*

Apart from whole program and path representations that are historically related to evolutionary strategies in the context of GI there are others that do not exactly fit in to that binary categorisation. Take for an example BovInspector's representation by Gao et al. [61]. It keeps a single copy of the program under repair and systematically searches for buffer overflow vulnerabilities and reduces their risk with 3 strategies:

1. Adding boundary checks with an if statement in the line above the suspicious one
2. Replacing an API with a predefined known safer API
3. Extending buffers

It can be argued that this is a whole program representation but the program is not being evolved. However, this is clearly an improvement of the program with a computational search so falls under the wider definition of GI. The software repair frameworks, Relifix [189], CFix [86], and PACHIKA [40] have similar setups but look for different types of errors and with different strategies.

3.3 SEARCH METHODOLOGIES USED FOR GI

In the science of search and optimisation there are countless tried and tested researches to proceed from. All have their merits and disadvantages, often connected with the domain in which they are employed. GI is based around the concept of search, where it searches for an alternative version, of an existing program, that possesses some particular properties the developer needs or wants. As one would expect, the search method is strongly codependent on the representation and the search domain.

Domain specific search methods have been referred to as heuristics and they exploit some known properties of the search landscape by defining rules to increment towards better solutions. Metaheuristics have an adaptive nature [128] by re-evaluating the rules at each step. Hyper-heuristics add another layer of search on top to either select or construct metaheuristics. Then the algorithm is said to be searching in the space of metaheuristics. Given the afformentioned definition, GI can be regarded as a hyper-heuristic since it is searching in the space of programs. The programs can be considered the algorithms that provide solutions and GI is searching or constructing better programs but not the actual solutions. GI researchers have mostly utilised meta- or hyper-heuristics as the search layer, where GP and GA have been recognisably dominant. Nonetheless, there are examples of traditional optimisation methods which are commonly thought of as a deterministic and mathematical way to find the optimal solution to a problem. There is even an example of work where the primary search method is simple random search [158]. In this section we will be inspecting the top layer, exploring the literature and asking the question: *What method is at the top?* While disregarding intermediate (and sometimes convoluted) search layers.

3.3.1 *Genetic Programming*

GP [94, 155] can be considered to be a hyper-heuristic [29] search methodology. It searches in the space of programs and has commonly been used to construct functions, models, or procedures to solve particular problem that depends on some input. For historic reasons it has been the prevalent GI search method and is from where the "genetic" part of the name is drawn from. The traditional way of evolving programs with GP is by constructing trees (see Figure 3.1) and evaluating them with input-output testing methods. A large portion of the

GI literature adopts this method with minimum adjustments in their implementations, only varying some parameters of the search. More specifically:

- GenProg and related implementations [122]
- White et al. low-resource system improvements [202, 201, 204]
- The Apache Hadoop HashCode repairs [91]
- Arcuri et al.'s pioneering program repairs [9, 8]
- Gen-O-Fix, the embeddable framework for JVM programs [31, 188]
- Sitthi-Amorn et al.'s shader simplification work [183]
- Robert Feldt work on generating numerous equivalent programs that would complement each others' faults [53, 54] Closely related to the field of N-version programming [126]
- LocoGP by Cody-Kenny [37, 39, 38]

However, there are other methods of implementing GP as there are many ways of representing an executable program. One such technique, commonly referred to as linear GP, represents the program as a sequence of instructions and manipulates that sequence (see Figure 3.5). Research with a top layer of linear GP include some of the assembly-/binary-based work of e.g. Schulte et al. [174, 178, 176, 175] and Orlov et al. [148, 147]. Additionally, the work of Ryan et al.[169, 166, 167, 168, 193] and Williams et al. [207, 208] utilise a so-called grammatical evolution where the elements in the sequence describe the transformation grammatically.

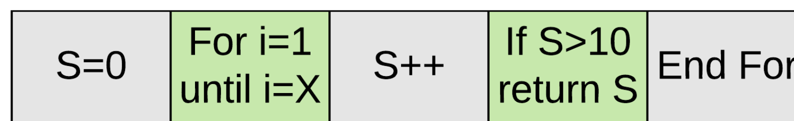


Figure 3.5: Sequential representation of a GP program. Each block is an instruction in a sequence but blocks can be repeatedly executed like those inside the *for* loop

A notable portion of GI literature that manipulates sequences of grammar instructions stems from the work of Langdon et al. [107]:

- MiniSAT efficiency improvements by Petke et al. [153, 152]
- *Grow and Graft*; Cuda pknotsRG for RNA analysis [110], interpreter for Babel Pidgin [78], citation service for Jango [84], call graph and layout feature for Kate [132], and the realisation of the tool μScalpel [13].
- Efficiency improvements of the DNA analysis tools BarraCUDA (sequencing) [115] and Bowetie2 (alignment) [109].

- Imaging software improvements of the execution time of a 3D medical image registration [112] and a stereo camera program from nVidia [106].
- Energy optimisation in mobile devices by Bruce et al. [23, 24, 26].
- Transform a version of gzip to parallel Cuda source code [102].

The general unifying property of GI-GP work is the use of tests to evaluate fitness and faithfulness to properties that are to be preserved. Still, Katz and Peled added an element of a formal verification method alongside testing [87]. The testing mechanism came into effect when the model checking failed to deliver proof.

3.3.2 *Genetic Algorithm*

GA is a closely related methodology to GP. It has traditionally been used to directly search for solutions as in the metaheuristic concept but the generality of the method has promoted it for the top layer for hyper-heuristics. As such, it is ideally adaptable to be used when searching in the space of programs. One could argue that it is more constricted than GP but that debate is beyond the scope of this thesis. The traditional representation for GA is a sequence of values or genome of genes respectively. The genes map to a choice in the solution and can be a binary (on/off) mapping, a real value number ($\mathbb{R}$), or any choice from a finite set. Usually, the genome is of a fixed length but not always. Although this seems indistinguishable for linear GP, the difference lies in what they represent. While the GP sequence is executed in the order it is presented, the GA's genome is not necessarily in the same order as in the executable final result. What that means is that the first gene in a GA's genome might affect line 30 in a target program and the last gene could at the same time affect line 2. Additionally, the genome does not always represent the whole program but only choice points in the program.

A program repair system by Wilkerson et al. called CASC [206, 205] mapped each gene to a fixed node in an AST. The choices of each gene depended on the node class such that the content could only be swapped with syntactically equivalent statement. A few year later Wu et al. adapted the concept to directly map to the source code instead of an AST and called it deep parameter tuning [211, 210]. Each gene could contain a finite set of values from C operator classes. White et al. did the same thing with three algorithms in the Akka toolkit [203] but they limited the gene values to integer constants in the source code. Additionally they used covarience matrix adaptation to approximate a gradient for the selection mechanism. Burles et al. defined each gene as a node in the AST of Google Guava that created a Guava container object they could then search the choices of possible semantically correct substitutions for the whole node [30]. Orlov et al. used the GA to search for combinations of the instruction set of a evolutionary system [146], so not directly in the source code but they were transferable parameters between solution spaces.

A more traditional GA, in the sense that the genome was a binary sequence, was used by Cito et al. [36] to dynamically determine which features in a mobile device should be turned off. Linares-Vásquez et al. [127] had a similar concept in mind for choosing colour maps in a mobile applications on Android as did Yeboah-Antwi et al. in the slimming procedure with ECSELR [214, 213].

These are examples of where the genes represented a fine-grained choice in the source but Ramirez et al. used a GA search to make design choices of a software's architecture [163]. Thus each choice had much more impact on the whole program.

3.3.3 *Other Search Methods*

Search or optimisation methods used in GI research and do not fit into the GP/GA paradigm are various:

- Exhaustive search of a limited search space, either by natural constraints or design.
- Constraint solving by means of proof search or algebraic methods.
- A stepwise hill climbing algorithm that is either greedy or not. The difference being in the acceptance rate of improving moves.
- Gradient descent/ascent with estimated or exact measure of gradient.

As well as a single example of simulated annealing to search through colour conflict graphs to minimise energy usage of displaying a website on mobile devices [125]. Simulated annealing is a metaheuristic akin to hill climbing but with an adaptable step size parameter, called temperature.

EXHAUSTIVE SEARCH METHODS The exhaustive search of the entire search space is seldom possible in the context of program manipulations because of the sheer volume of the space. However, artificial constraints can be made to reduce it to a manageable size. Many automatic software repair methods constrain themselves to searching a limited set of strategies [86, 61, 85] or patterns [189, 88]. Sometimes the factor that limits the number of solutions available for the search is a design choice e.g. the framework only targets for loops [179] or has a low finite number of resources [180, 40]. Debroy et al. for an example enumerated all first order mutants of a faulty program and then iterated through them to find a repair. They implemented a clever selection mechanism based on fault localisation algorithm to determine the order of the iteration and the algorithm escaped the search if it found a fix. [43] Kocsis et al. had a single transformation strategy and walked it along the AST, applying it everywhere there was a possibility that the program could create redundant data

structures [92]. In another work, they use proof search which iterates through a finite number of rules to construct code segments that could be grafted into existing software [90].

mathematical solvers When problems can be modelled as an arithmetic, logical, or algebraic function it seems like an obvious choice to solve it, deterministically, for the optimal solution as opposed to searching stochastically. As calculus predates any thoughts about computers and software, there are well established methods of solving a number of deterministic problems. Problems that GI has been used to target have been modelled as satifiability constraints for input-output pairs of string literals [172], and for a execution trace in the DirectFix system [139]. Satisfiability solvers have also been used to fix conditional bugs based on preconditions [44], and faulty data structures [49]. Similarly Nguyen et al. employ symbolic execution with input-output oracle to formulate a set of equations that can be solved to synthesis a repair [142].

hill climbing Hill climbing is an exemplar heuristic, i.e.: at each step search neighbouring solutions and move in a direction that leads you to a fitter state. Greedy hill climbing algorithms will accept the first improving move they encounter otherwise they accept the next move with a fixed probability. It is a search on the path of least resistance. Two GI examples utilise this method for automatic fixing by considering a single variant at a time, estimating its neighbourhood of programs, and moving towards higher fitness [150, 195].

3.4 software properties improved with gi

As one would expect a crucial part of the GI paradigm is that there is some property that is being improved. Programs have a huge number of different properties that they can be categorised by so the taxonomy of software by some or all of those properties could fill a thesis. However here we define properties of software as improvable and therefore measurable characteristics that can be affected by manipulating the code in some way. A list of the properties that have been the focus of current GI studies can be found in Table 3.2. We have divided those properties into two groups: *Functional (or physical)* and *Non-functional (or logical)* properties. Which we will address separately in the following subsections.

3.4.1 *Functional Properties*

Functional properties refer to the purpose of the program or what it was designed to do. They are often difficult to measure as sometimes they depend on subjective view of the user, like the optimisation of colours in a graphical user interface [127]. However such opinion-based measurements are most often a secondary fitness unless it can be measured with input-output

Table 3.2: The different properties of software that have been improved by GI.

Property	Fitness measurement	Objective	Number of GI examples	References
Correctness	Number of test cases passed	Maximise	48	[53, 7, 6, 8, 197, 196] [58, 143, 198, 122, 121, 123, 120] [119, 145, 158, 133, 157, 50] [85, 43, 40, 176, 175, 205] [189, 86, 44, 91, 180, 88] [1, 206, 144, 195, 150, 178] [111, 117, 66, 67, 69]
Correctness	Constraint equations	–	4	[61, 139, 142, 172]
Correctness	Inferred vulnerability	–	1	[151]
Add feature	Number of test cases passed	Maximise	6	[78, 84, 132, 13, 153, 148]
Remove feature	Compiles	–	3	[101, 214, 213]
Execution time	Time	Minimise	41	[193, 168, 169, 167, 166, 207] [208, 102, 104, 105, 106, 107] [109, 113, 115, 116, 108, 114] [110, 103, 112, 153, 152, 179] [183, 211, 210, 25, 202, 204] [203, 37, 39, 38, 184, 92]
Energy usage	Joules or Watts per time	Minimise	11	[201, 204, 202, 24, 127, 125] [36, 82, 21, 140, 187]
Memory usage	KB	Minimise	6	[165, 210, 211, 213, 214, 51]

pairs. The functional properties that are addressed in current GI literature are either added functionality or correctness, otherwise known as fixing.

3.4.1.1 *Program repair*

The largest body of GI work regarding functionality is bug fixing, most of which is purely search based and dependent on testing with test cases. However there are examples of other

measurements of correctness. Sometimes it is possible to verify the semantics with formal methods [61] so the fitness is either correct or not [139]. Bugs can also present themselves as security vulnerabilities and those can be difficult to measure. Perkins et al. use an inference method, in their ClearView system to estimate the fitness of a running program to know where and what it should patch [151].

Dijkstra advocated proof over testing for software correctness measure [46] as he was of the opinion that programs had to be verified and it should not be done approximately. Nonetheless, the fitness function when doing bug fixing with GI is most often the number of test cases passed or some derivative. Although Feldt's work [53] predated Arcuri et al. by a decade the paved the way for test based program repair with co-evolving small programs and test cases [7, 6, 8] Le Goues and Weimer et al. followed shortly [197, 196, 58, 143, 198] with their GenProg framework [122, 121, 123, 120] which is a generate-validate framework. It has since been adapted by many GI practitioners [119, 145, 158, 133, 157].

Fast and Le Goues et al. [50] also tried combining test driven repair and dynamic predicates. The overhead computational cost of calculating the predicate was not justified by the small increase in precision. However their use of test case selection allowed considerable resource savings when repairing software with large test suites. There are various implementations of test strategies in the literature but the majority of GI test-based bug fixing work still measures correctness with number or proportion of test cases passed. In current literature the following 17 papers, besides those cited here above, only use testing to measure fitness [85, 43, 40, 176, 175, 205, 189, 86, 44, 91, 180, 88, 1, 206, 144, 195, 150] In cases where test suites are small there is no need for test strategies, e.g. Schulte et al. only have a single test case [178] in their repair of the router firmware.

Test case fitness can also serve as an analysis of the correctness property of programs. Langdon's and Petke's analysis on the fragility of software was made by measuring correctness with test cases [111] as did Langdon et al.'s work on visualising the landscape [117]. Additionally, the use of test cases, even if they are many, does not explicitly mean that the proportion of the pass test cases will be used. Nguyen et al. construct a set of equations [142] with pass-all-tests as a variable, thus the correctness depends on the tests to correctly reflect expected behaviour but the fitness function essentially returns a binary true or false value. Samimi et al. had the same two states in their fitness function as they also integrated the testing as a constraint in a satisfiability formulation [172].

The contributions of this thesis in Chapters 6 and 8 are bug fixing improvements that also rely entirely on testing [66, 67, 69].

3.4.1.2 *Other Functionality Improvements*

Adding functionality is a relatively new application of GI but has a lot of potential and draws attention from the computer science community. From the point of view of the artificial intelli-

gence community, ideally one would like to be able to give vague and high level description of a desired functionality and get it as soon as possible. However, current techniques should not make many programmers unemployed and it does not reflect the goal of GI accurately. Rather it is to facilitate the process of improving software with tools and techniques.

There are multiple ways of adding new features to a program. A developer might decide to add specific functionality for the end user, e.g. a translation mechanism in a messaging application. Harman et al.'s [78] method of *Grow and Graft* was used to embed the messaging software Babel Pidgin with such functionality. Other end user specific features that were grown in isolation and then grafted into existing programs include: a citation service integrated into a website that uses the python based web service framework Django [84], and a call graph and layout feature into the open source text editor Kate [132]. Grow and graft is directly related to the automated transplantation framework of Barr et al. [13]. The framework omits the grow and substitutes with finding the object to graft either elsewhere in the same source code, like Petke et al. did with MiniSAT [153], or from another program. These functionality improvements are all test-driven search processes. However under specific circumstances and with particular data types the grow process can be done with deterministic proof search [90].

The added feature need not always be connected to the user interface. Orlov et al. developed FINCH and used it to improve a number of programs, including trail navigation and image classification [148]. Gen-O-Fix [188] has been used to improve accuracy of a regression model and the framework presented in this thesis has been used to improve and maintain accuracy of a prediction mechanism [65]. While Gen-O-Fix was used to predict stock prices this thesis' framework was used to predict outcome of rehabilitation for clients in an Icelandic rehabilitation centre.

Although slimming might be considered more of a memory optimisation problem it is also a process of removing unwanted or unnecessary features. Landsborough et al. [101] use a GA to remove code segments from a binary image of compiled programs as do Yeboah-Antwi et al. [214, 213]. While Landsborough et al. removed features post-compilation, they did so with dynamic analysis before retesting the program. However, Yeboah-Antwi et al. removed features at runtime, essentially making the program truly dynamically adaptive.

3.4.2 *Non-Functional Properties*

Non-functional properties refers to features of the program that do not dictate what it does. These are measurable characteristics that could directly influence the users perspective but has no semantic effects on the processes of the software. The only three non-functional properties that GI researchers have attacked are *execution time*, *energy consumption*, and *memory efficiency*. In that order of decreasing popularity.

3.4.2.1 *Execution Time*

Execution time is the most explored non-functional property [64]. This might possibly be because implementing a time measurement is relatively easy compared to other non-functional properties such as memory use or energy consumption [63].

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 [153]. 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.

Chapter 7 of this thesis adds to an already extensive list of GI work on optimising execution time. Successes range from subtle 1% gain by mostly removing assertions [152] and moderate 17% reduction [153] to extraordinary 70 and 100 fold speed up [109, 113]. Nearly all of Langdon's and Harman's et al. work aims to make less time consuming computations whether it is via parallelisation of medical applications [103, 112] or various bioinformatics software [102, 104, 105, 106, 107, 109, 113, 115, 116, 108, 114]. One of which used the grow and graft method to evolve improved pknotsRG for the RNA folding software [110]. Much of it based on their GISMOE concept [75], trading accuracy for efficiency. Sidiroglou-Douskos et al. [179] followed that idea by perforating loops to shorten execution time. Sitthi-Amorn et al. [183] worked on the trade-off for graphical shaders, sacrificing some accuracy for increased rendering speed, which has further imbued other researchers with the idea to seek execution time improvements [211, 210, 25].

Some of the work in execution time improvements include automatic parallelisation [193, 168, 169, 167, 166, 207, 208], a multi-objective optimisation of embedded systems [202, 204, 203] and arbitrary small functions [37, 39, 38]. Kocsis et al. [92] reported 10,000 fold improvement on Apache Spark query optimisation by eliminating unnecessary data structures. Sohn et al. [184] used GI to tune OpenCV's group size setting and achieved significant results.

3.4.2.2 *Energy*

Since the 1980s awareness of climate change and the importance of conserving energy has not been lost on the computer science community. Efforts have mainly been focused on large scale

computing and hardware in data centres [70]. Energy consumption of hardware can however only be optimised to a certain degree. Energy consumption of deleting a single bit has been shown to have a lower bound [16]. Therefore, hardware and consequently general computer optimisation pertaining to energy will also have a lower bound. No matter how energy efficient the hardware we produce, if we do not develop software to follow suit, we can never hope to get close to reaching the limit of Landauer's principle [100]. It is derived from the second law of thermodynamics and states that the energy consumed by computations has a theoretical lower limit. The principle was introduced in the early 1960s but was experimentally verified in 2012 by Berut et al. [16]. Like a car is only as energy efficient as the person driving it, the same applies to computers; the hardware can only be as efficient as the software running on it.

One could argue that there are four levels to improving energy usage in computing.

1. Hardware optimisation, designing and producing hardware that conserves energy.
2. Optimizing the OS or kernel.
3. Minimizing the amount of computing used for a particular task.
4. End user specific energy conservations like turning off features that are not in use at all times such as network devices and positioning system.

In GI we have mostly been concerned with the second and third levels where we can directly apply it to improve certain parts of the OS or specific software. Although we can specialise software to be energy efficient in isolation, we cannot prevent other applications from interacting with it. However, GI gives developers the flexibility to specialise software between platforms, computers and hardware but caution should be taken when measuring the energy used. Schulte et al. have presented an approach which can potentially be used to reduce energy usage of compiled x86 programs [177]. Which could serve as a specialisation tool for platforms.

Smaller scale examples of GI, include energy usage reduction in embedded, low resource systems [201, 204, 202]. Bruce et al. [24] have made a version of MiniSAT that draws less energy as measured on the CPU, disregarding memory devices and peripherals. Li et al. [125] and Linares-Vasquez [127] target colour compositions in mobile devices. Hoffman et al. [82] and Cito et al. [36] implemented two separate frameworks, also for mobile devices. However both implementations control which features are turned on and off while the device is running on battery. Two examples of evolved median functions for conserving energy exist in the GI literature. Mrazek et al. [140] made one specifically for micro-controllers while Brownlee et al. [21] tested an estimated median function as the pivot function for quicksort, which has also been the target of energy optimisation [187]. The latter example could serve as a specialisation to conserve energy in devices that have to identify denial-of-service attacks. Additionally

Brownlee et al. developed OPACITOR [30], a tool to measure energy consumption of programs on JVM.

Until now the only reliable way to measure how much energy a software consumes has been through highly specialized or customized system. Surrogate objectives are necessary to achieve dynamically adaptive "green" software. A single point of measurement as a substitute such as CPU cycles is not enough since they provide no consistency across programs or platforms. We therefore can not rely on them to make scientific conclusions nor use them in practice. We would want alternative measurements such as simulations or modelling from multiple programmatical sources so that we can reliably use them in science or at the very least make energy optimisation practical. Recent effort has been made by Bruce et al.[26] to estimate energy measurements and effects on the GI process. They used MAGEEC energy measurement boards that attach to Raspberry Pi and found that although precise, they lack accuracy. However the proportional energy decrease translated between devices.

3.4.2.3 *Memory*

Other non-functional properties that have been explored with GI include a limited work on memory optimisation. Wu et al. [210, 211] have applied their deep parameter tuning method to control and minimise memory usage of C programs while Risco-Martín et al. [165] targeted dynamic memory managers. These two examples of work used a trace and instrumentation to measure in KB how much memory was in use during execution. Fatiregun et al. [51] took another path by minimising the size of the program itself, which is the same objective as Yeboah-Antwi et al. had in their first experiments with the ECSELR framework [213, 214]

4 DYNAMIC ADAPTIVE SOFTWARE FOR GENETIC IMPROVEMENT

The software engineering community has been researching adaptive software implementations [60] and architecture [136] for a long time [35]. In the increasing uncertainty that today's software deals with it is imperative that at least large systems are equipped with some self-adaptability. The rapid expansion of code that is currently in use has long since outgrown beyond the point where maintaining it all manually is nigh impossible. Software engineers have come to realise this and have increasingly started integrating dynamic adaptive capabilities into software before deployment [27].

Many methodologies and programming paradigms are available for the developers to consider [215, 170, 42], such as feedback loops [27] and reflection [188]. The goal is to apply changes or adapt online and without disturbing normal activity. Carzaniga et al. [34] for an example suggested a technique to avoid runtime failures in Java programs. It works by exploiting redundancy in reusable components but before the program is started the source code needs to be processed. The pre-process identifies these reusable units and instruments them so the program can produce execution paths variations. This makes the program resilient to failure and does not interrupt execution flow when it has to use the backup plan. Runtime insurance plans are effective but not guaranteed [59].

GI has most commonly been used offline to improve various software properties [154, 64]. As GI is a fairly young field the work on self-improving software with GI is not extensive, although SBSE literature has been considering self-adaptive systems for some time [74, 35, 42]. A few examples of adaptive or dynamic GI include a framework (Gen-O-Fix [188]) to continuously improve software on Java Virtual Machines in parallel with the targeted program, Burles at al.'s list of suggestions for embedded improvement methods [31], and an approach (ECSELR [213]) that injects dynamic adaptability in an already running target software. However, JM's approach (Chapter 8) is inspired by Harman et al.'s suggestion for *Dreaming Devices* [79], exploiting the fact that the majority of software is not in continuous use. Even if that were the case, the usage load will usually be periodic and during lower load times the device should be able to afford some capacity for improvements.

The common theme is that GI can and should be used to make dynamic adaptive software [74, 77] but the differences are the implementations and applications [213]. The debate is between what is truly dynamic adaptive and semi-dynamic adaptive. Dynamic adaptive connotes a reaction to changes in environment (outside source), the debate seems to be

focused on the time frame of those reactions. Do they have to be instantaneous or can they happen whenever? What defines dynamic adaptive software? Surely it is that it reacts on itself, without the human middleman but are there other factors? Dynamic literally means that there is a constant change and adaptive describes an ability to adapt to different circumstances. So Dynamic Adaptive Software is a software that can constantly adapt to changing environment, whether the changes are in usage, hardware, or other software. The time frame of the adaption is irrelevant as long as the software takes care of the adaptation.

The few examples of adaptive software in the GI literature include the framework Gen-O-Fix [188], written in Scala. It was one of the first implementations of dynamic adaptive software using GI. The framework makes use of the following features that Scala offers:

- The ability to treat both code and data as first class variables.
- Declarative reflection, which provides complete information about classes, variables, functions and methods at runtime.
- Powerful pattern matching for object structures
- Implicit conversion, allowing the user to define implicitly what data types are equivalent

The user is only required to provide the initial source code of the application that is to be improved and a function that measure fitness. The framework instruments the source code and then runs in parallel updating the software every time it finds a better version (Figure 4.1) Another framework from Swan et al. named TEMPLAR [187] exploits much the same capabilities of Scala but to provide researchers with a tool to rapidly prototype Hyper Heuristics.

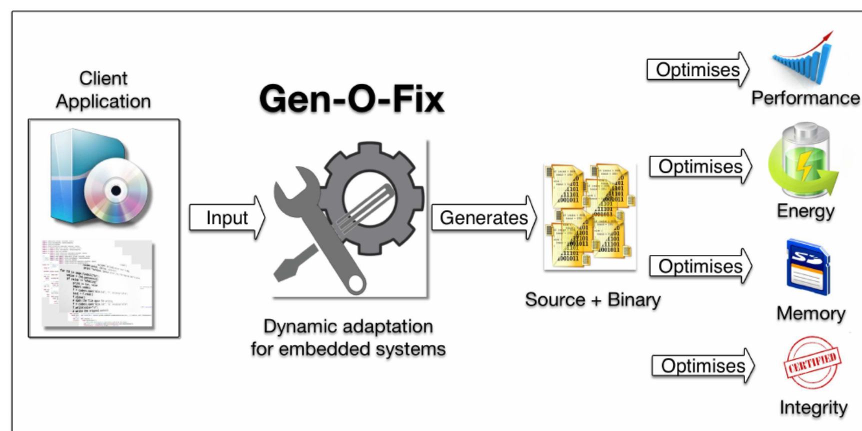


Figure 4.1: A system diagram of Gen-O-Fix [188]

Surprisingly in today's environment with a excessive amount of mobile devices and their rapidly expanding use, there is only one application of GI that specifically targets software for that domain and two that can be applied there. Cito et al. implemented an adaptive

binary modification system that identifies and prevents recurrent background requests in Android devices when battery charge gets low [36]. The system must be applied to individual applications but their initial evaluations shows up to 5.8% decrease in battery usage. Although not specifically aimed at mobile devices, Mrazek et al.'s work for microcontrollers [140] is at least applicable for devices that use them, e.g. routers and hand-held calculators. Their approach is based on approximation algorithms and manage to save space (at most 50% of the available capacity), energy (up to 75% reduction) and time (6 fold speedup) in their initial evaluation. Hoffman et al. [82] have published PowerDial, a system that turns static configuration parameters into dynamic variables that can be adjusted as battery levels change. It is possibly more suited for larger systems but the concept is definitively transferable to mobile devices or embedded systems. PowerDial performs a pre-processing analysis of the parameters before the application is started to identify and calibrate the parameters with sensitivity analysis.

Yeboah-Antwi et al. have developed the ECSELR framework [213, 214] that targets programs running on a JVM. The framework defines a number of agents that hook into an already running program with operation system specific methods. It is an event based observations system that evolves the target program in the same system memory but still separately. The system is ambitious but still in its infancy so it is still experiencing problems that can be connected with new development. The brute force tactic that is used to make the evolutionary environment as similar to what the actual program is running in is quite computationally expensive. Other issues are regarding unexpected side effects of the evolutionary process. At current time it is suitable for software that resides within large systems with plenty of resources.

There are two examples of adaptive GI applications that are integrated parts of larger systems. In those programs, the concept of the *Dreaming Device* suggested by Harman et al. with Arcuri et al's co-evolutionary bug fixing [6, 7, 9, 8] has been realised. They implement a self healing property [67] and accuracy improvement of a predictive model [65] in a live system. Both are presented and described in Chapter 8.

Part III

CONTRIBUTION

THE IMPLEMENTATION OF THE FRAMEWORK 5

5.1 INTRODUCTION

In this chapter I will detail the implementation of the GI framework used in this thesis. The implementation of the GI is inspired by the work of Langdon *et al.* [102, 78]. Like their framework, this one operates on the source code with no need to convert the program to a different representation like AST [1]. Therefore this approach is directly transferable between programming languages with minimal configuration as is shown in this thesis. In Chapters 6 and 8 it is applied to Python programs while in Chapter 7 it is being applied to C/C++ software.

This chapter is structured as follows: Section 5.2 explains the representation the GI framework uses. Section 5.3 gives an overview of inputs and parameters that can be passed to the framework. Section 5.4 walks through the general procedure that is implemented and Section 5.5 summarises the framework's details.

5.2 REPRESENTATION

The framework evolves lists of edits (patches) as seen in Figure 5.1. A single patch consists of a list of edit operations sampled from the set shown in Table 5.1 and is applied in sequence from the first edit to the last item in the list. Each edit details how the source code should be manipulated with one of the operations in Figure 5.2

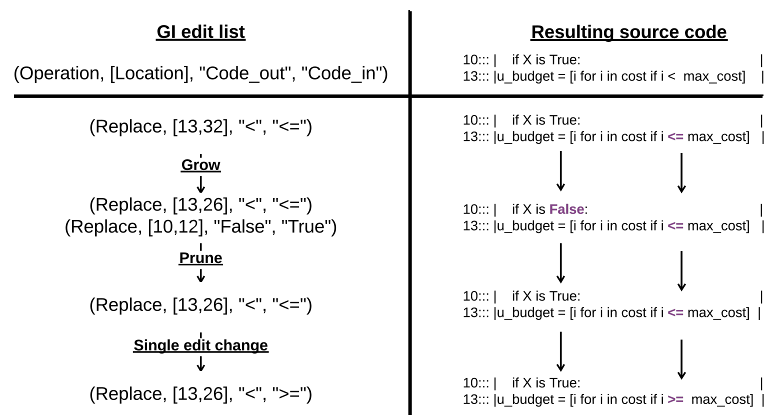


Figure 5.1: An example of an edit list and how it can evolve with *Grow*, *Prune* or *Single edit change*.

The information contained in a single edit is controlled by which of the operations is selected but generally it is a location and the fragment(s) of code involved. *Delete* and *Swap* only manipulate whole lines but *Copy* and *Replace* can be used on parts of lines. The reason for this is that deleting a part of a line will likely cause syntax errors and *Swap* is a combination of *Copy* and *Delete*. Table 5.1 lists the different ways they can be used and what arguments should be passed with the operation. Location is denoted with a line number L_i and column number C_i, which is character number C counted from left in line L_i

Table 5.1: The choice of edit operations and their arguments. Granularity can either be

Operation	**Granularity**	**Arguments**		
		Location	**Code before/after**	**Description**
Copy	Whole line	$[L_1, L_2]$	Just the line being copied	Copy line number L_1 above line number L_2
	Part of line	$[L_1, C_1]$, $[L_2, C_2]$	Just the snippet being copied	Copy code snippet in line L_1 starting from column C_1 in front of column C_2 in line L_2
Delete	Whole line	$[L_1]$	Line being deleted	Add the appropriate *comment* character at start of line L_1
Swap	Whole line	$[L_1, L_2]$	Both lines	Swap the contents of lines L_1 and L_2
Replace	Whole line	$[L_1, L_2]$	Both lines	Copy line L_1 and replace line L_2 with the copy
	Part of line	$[L_1, C_1]$	Code snippet before and after	Replace the *before* code snippet found in line L_1 starting from column C_1 with the *after* code snippet

The granularity of an edit is determined by the code snippets it manipulates. They can be whole lines from the source code, or one of various single operators or numerical constants that are defined separately for each programming language. A table of Python specific operators can be found in Section 6.3 and for C++ in Section 7.4.1. As seen in column four in Table 5.1 the code before/after argument can be either one or two pieces of source code.

The edits are constructed from the source code which is read as a text file and stored in a data structure of program lines. Each line's data structure holds the following information:

- The raw text as it appears in the source file.

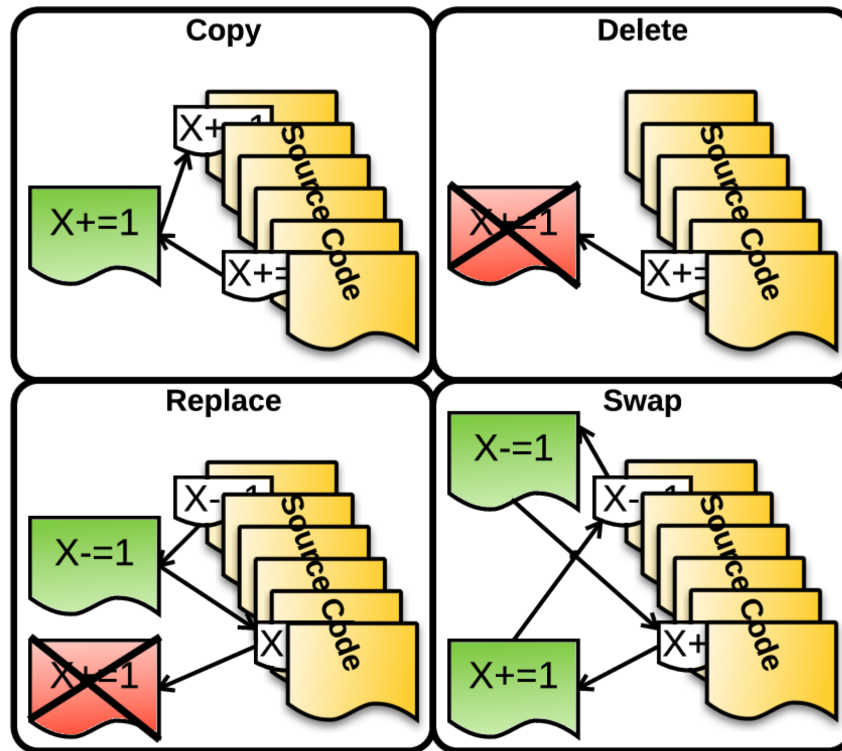


Figure 5.2: The four types of edits that can be applied to the source code. More detailed descriptions and list of arguments to each can be found in Table 5.1

- Line types as defined by the GI practitioner (see Section 5.3).
- Indentation as the number of space characters[1].
- If the lines immediately following should be indented or not. This is only relevant when the target source code is in a language like Python where otherwise it has no effect.
- Whether the line can be altered or not. This can be varied by including or excluding certain line types in a list of untouchable line types. Empty lines, function and class definitions, imports, multiple line comments, and a few others are included by default (see Section 5.3).
- A list of variables, and operators that the line contains, along with their location on the line. Regular expression patterns are used to identify and locate the operators. The patterns are kept simple to minimise constrictions to the search so the identification is vulnerable to false positives and might make changes available to the GI that do not always make sense. For an example the GI might find the operator < in a string constant which has no effect on any semantics nor syntax.

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

The GI framework requires five inputs to work with:

TARGET SOURCE FILES: A list of paths to files it has access to and can edit.

EVALUATION FUNCTION: A callable script or a command that returns a comparable value to assign fitness to program variants. This can measure correctness by running software tests, performance by recording execution time and memory use, or energy efficiency by inspecting the energy use when executing the target program.

LINE TYPES: A list of different line types as defined by the GI practitioner. Each element in the list contains: a name for the type, regular expression pattern to identify the lines, an indicator if it can be modified, and another if it starts a block like a *for* loop. An example of defined line type list can be found in Tables 6.1 and 6.2 in Section 6.3.

OPERATORS: A list of sets that are interchangeable as defined by the GI practitioner. An example of such list can be seen in Table 6.3 in Section 6.3, a single set might contain arithmetical operators ($+ \quad - \quad \times \quad /$).

COMMENT CHARACTER: A symbol that the programming language defines as the start of an in-line commented text. In Python it is e.g. # while in Java one can use //. It is used for the edit operation: *Delete*.

There are three compulsory inputs, the two first are like in Harman et al. [76] the target source files and evaluation function, and the third is the comment character. The other two can be omitted if the GI practitioner wants to put less restrictions on the search space. The line type list defines what lines can be modified and restricts line swaps and replacements between lines of same type. If omitted, then every line, that is not a *comment*, is tagged as *generic*, modifiable, and replaceable with any other line. If the operator sets are not provided the framework uses regular expressions to try to identify all operators used in the target files by searching for variable names and connecting symbols. It searches for whole words and any sequence of non-alphabetical characters or digits are determined to be operators that can be modified. It then puts all the symbols it finds in the same operator set and so they all become interchangeable.

In addition to inputs the framework has a number of parameters to control the algorithm that have default values but can be controlled by the GI practitioner. Table 5.2 lists these parameters, expected type, what values can be chosen, and the default value if the practitioner does not provide one. It includes the traditional Evolutionary Algorithm (EA) parameters, such as:

NUMBER OF GENERATIONS is the number of iterations of generate, evaluate, and select the algorithm goes through;

POPULATION SIZE defines how many edit lists should be generated for each iteration;

NUMBER OF PARENTS counts how many are selected to be mutated in each iteration. This number can be anywhere between 0 and population size, if it is less than 1 then it is interpreted as a proportion of the population.

ELITISM FACTOR sets the number of edit lists that are copied without change from one iteration to another.

There are other less traditional parameters like *penalty* that is added to or subtracted from the fitness for non-compilable or non-testable variants, *line range* and *granularity*. The *line range* parameter specifies what lines in the target files are open for modifications and can be used to focus the GI on specific methods, functions or classes. For each target file a pair of integers denote the beginning and end of the line range, if omitted the whole file will be modified. *Granularity* and is defined as the "size" of the code snippets that are modified and the framework distinguishes between two levels:

MACRO Moving or modifying whole lines.

MICRO Moving or modifying variables, operators and parts of lines.

The framework uses both types as a default. Initial edit list size is equivalent to GP's initial tree size and defines the maximum number of edits to generate when constructing a new list. An additional parameter, called *Termination time limit* is the maximum execution time an evaluation has before the variation is forced to terminate and determined to be non-halting.

The parameters for the *Breeding* and *Selection* functions in Table 5.2 are limited to the options that have been implemented, so only require the GI practitioner to choose. The choices for a breeding function are displayed in Figure 5.1 and are as follows:

grow is where a randomly generated edit is appended to the edit list. The edit is generated by a stochastic selection from all possible mutable locations in the source with a equal probability. The location is a line and a column (if micro) or just a line (if macro). If the case is a line, then another uniform random selection is made from possible replacements for the content of the location (see Table 7.5). 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. The reason for removing all and not just one is that the following edits might depend on changes of previous edits.

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.

all gives the three above equal chance of being selected for each time an edit list is modified.

The selection function has three options: weighted selection, truncated selection, or random selection. *Weighted* selection gives each edit list i a probability (p) of being selected, which is its fitness f in proportion to the population's overall fitness (eq. 5.1).

$$p_i = f_i \Big/ \sum_{j=1}^{N} f_j \qquad \text{for edit list} \quad i \tag{5.1}$$

Truncated selection sorts the edit lists by fitness in descending order and selects the upper part of the population. How many are selected is controlled by the number of selected parents parameter. It is a number anywhere between 0 and the population size. If set between 0 and 1 then it is interpreted as a proportion and the default value is half the population, otherwise it denotes the actual number of parents.

5.4 PROCEDURE

When the framework is first initiated it parses the targeted files into the line data structures (see Section 5.2) and tags the line type by matching them with regular expression patterns. If the line does not match up with any of the predefined patterns it is assigned the default *Generic* line type. *Comment* is the only line type that possibly can change types as the GI has the option of deleting the comment characters from the start of the line but only if that line was previously a target of a delete edit. If that happens the GI searches through the patterns again to redefine the line type.

The search method of the framework is a simple population based evolution as seen in Figure 5.3. Every stage of the algorithm is customisable with the parameters detailed in Section 5.3. The general outline of the procedure is to start by initiating a population of randomly generated edits and evaluate them with the given evaluation (fitness) function. If the termination criteria is not satisfied, then select parents from the population to base the next generation on. Together the breeding and selection functions initiate each subsequent generation and the available remaining spaces are filled up with new edit lists.

The evaluation involves applying the edits to the source code in sequence and then measure the specific property the intention is to improve. The GI terminates programs that do not halt after a specified execution time and assigns them the appropriate penalty value (Table 5.2). The default penalty is zero when the objective is to maximise but ∞ when it is to minimise. The resulting outcome from the procedure is a list of edits that can either be manually inspected by the GI practitioner or directly applied as a patch to the original program.

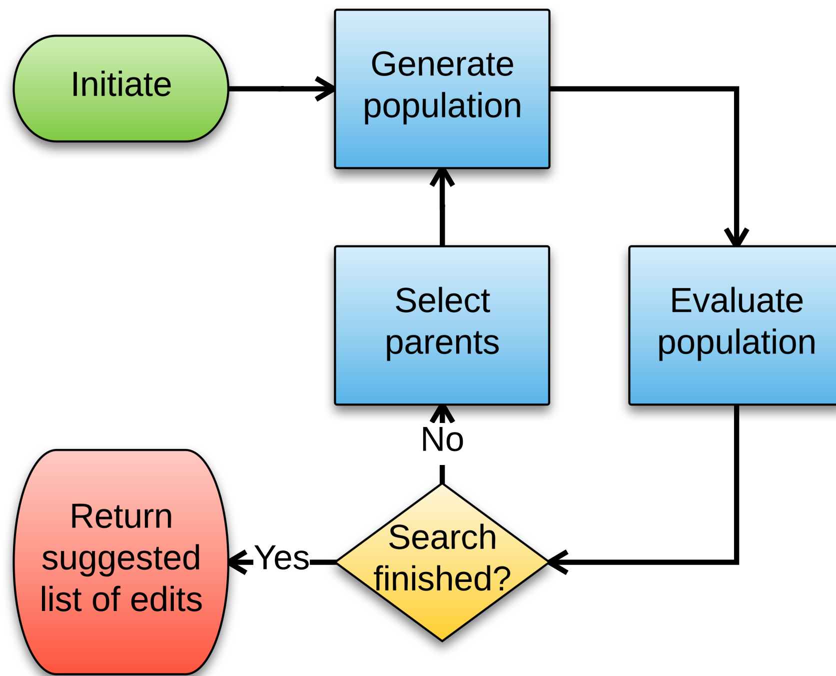


Figure 5.3: The general outline of the search algorithm implemented in the framework.

5.5 SUMMARY

I have described the general structure of the framework used in the experiments for this thesis. It uses an EA to generate lists of edits that can be used to change the source code directly. It parses relevant information about the source code into data structures to restrict the changes to correct syntax. The source code is treated as a string type variable when edits are applied. The framework has a number of parameters that control the search, most of them are optional arguments but can be set to specific values, depending on the purpose of the search. The edit list representation of this framework was inspired by Langdon et al.'s framework introduced in Part ii. The difference is that theirs is a series of BNF grammar rules while this one produces human readable instructions. This framework is also only as restricted as is needed each time. It can be provided with as many or few rules to follow while modifying the source like any string without structure. Other GI implementations usually enforce some kind of structure by using tree based structures with predefined nodes, types, and blocks. All subsequent chapters in this thesis use this overall framework. Each chapter uses it in different ways showing the framework's versatility and will contain an explanation of the specialised setup for its experiments.

Table 5.2: Parameters to the GI framework.

Parameter	Type	Possible options	Default value
Line range	$[x_i, y_i] \in \mathbb{Z}^2$ for each file i	$0 < x_i \leqslant y_i \leqslant n_i$ where n_i is number of lines in file i	$[0, n_i]$
Breeding function	String	*"grow"*, *"prune"*, *"single"*, *"all"* (see Figure 5.1)	*"all"*
Selection function	String	*"weighted"*, *"truncated"*, *"random"*	*"weighted"*
Objective	String	*"minimize"*, *"maximize"*	*"minimize"*
Granularity	String	*"macro"*, *"micro"*, *"both"*	*"both"*
Penalty	$x \in \mathbb{R}$	$0 < x < \infty$	0 or ∞ Depending on objective
Number of generations N	$N \in \mathbb{Z}$	$0 < N < \infty$	50
Population size P	$P \in \mathbb{Z}$	$0 < P < \infty$	40
Number of parents	$x \in \mathbb{R}$	$0 < x < P$	0.5
Elitism factor	$x \in \mathbb{R}$	$0 < x < N$	0
Initial edit list size	$x \in \mathbb{Z}$	$0 < x < \infty$	1
Termination time limit	$x \in \mathbb{R}$ seconds	$0 < x < \infty$	60

6 THE FITNESS LANDSCAPE OF MUTATED PYTHON PROGRAMS

6.1 INTRODUCTION

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 [64, 200]. Whether GI is used to fix faulty programs or improve some non-functional aspects it always searches in the space of program variants of the existing software. Like in most search algorithms the search is guided by fitness and in the case of bug fixing it is commonly defined as number of test cases passed.

The relationship between the program space and fitness is seldom simple and often quite difficult to analyse. Yet, this relationship, often called the "the fitness landscape", has a direct bearing on the efficiency of the search [194]. When investigating it we need to consider multiple factors both related to the search space and the evaluation of the fitness. On the search space side there are factors such as step size and ruggedness. For the evaluation side we might consider code coverage, size, and overlap of the test suites.

The main emphasis in the literature to date has been on research of applying GI to determine the improvement in performance, 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. What can possibly be done to increase the chance that the programs improve? We begin to answer this question in the space of smaller programs. Understanding the landscape of small programs can lead to insight to landscapes of larger programs. The landscape of smaller programs also allows us to analyse the full impact of changes introduced to the code and keeps uncertainties introduced by the rest of the program to a minimum.

This chapter aims to answer RQ 1:

Research Question *Can GI fix bugs in small programs, written in a dynamically typed language and how does the GI interact with the search space?*

We do that by exploring the relationship between fitness and number of accumulated incremental changes. We use GI to introduce bugs into correct programs and gain an understanding of the program repair process in a controlled setting. 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 in conjunction with the research question:

- 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 generalisation 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:

- Most examples of GI are applications on C/C++ and Java but very few applied to Python programs [1] and given that it is a very popular language, there is a corresponding 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 this chapter is organised as follows: Section 6.2 gives a short overview of the Python programming language in the relevant context of this chapter. Section 6.3 details how the GI framework from Chapter 5 is specifically set up for this chapter's experiments. Section 6.4 explains the experiment conducted to answer the above questions. Section 6.5 examines the results and Section 6.6 gathers the findings into a cohesive summary of the chapter. Part of this chapter's work has been published in Haraldsson *et al.* [66].

6.2 THE PYTHON PROGRAMMING LANGUAGE

The Python programming language was first developed in 1989-1990 by Guido van Rossum [192] and was intended as an interpreted prototyping language. It has since become widely popular and much more than just a prototyping language due to, in large, the open source community of users and developers. The programming language offers interactivity and object-oriented programming with dynamic typing and high level data types.

Although the dynamic typing offers a lot of flexibility, it also poses a problem for debugging. The dynamic nature can introduce well hidden bugs by subtly changing a type of a variable or assigning value to an already defined variable name. If the programmers are not careful they can easily code in such a way that a variable assignment in one function has side effects on

output from another function at runtime. It can then take an experienced Python programmer considerable time to locate and fix even a single such bug.

The dynamic typing of Python could also have an impact on the GI's performance, for better or worse on various levels. The search process can be impacted in the sense that the search space is potentially larger than for static typed languages and so the search will take longer. Less restrictions and a finer grained search space might also result in clever or unconventional solutions that the developers might not have identified themselves. That however might in turn cause them to distrust the outcome because they are unable to verify its results.

6.3 THE FRAMEWORK SETUP

Chapter 5 describes the framework which was used for these experiments. Each of the three programs is contained in its own file which are the targeted files. The files are read into memory and each line gets tagged as one of the types in tables 6.1 and 6.2 by matching them with regular expression patterns. If the line does not match up with any of the predefined patterns it is assigned the default *Generic* line type. The framework also searches each line for numerical constants and the operators in Table 6.3. The fitness landscape that we explore in this chapter is limited to *micro* mutations that only do replacements of the sets in Table 6.3.

The evaluation functions are basic scripts that run unit tests and count the number of passed and failed test cases.

Table 6.1: List of defined line types that cannot be altered and therefore not accessible to the GI.

<table>
<tr><th>Line type name</th><th>Indent following line/s</th><th>Regex pattern</th></tr>
<tr><td>Empty</td><td>False</td><td><code>^$</code></td></tr>
<tr><td>Import</td><td>False</td><td><code>^(import|from).*$</code></td></tr>
<tr><td>Multiple line comment</td><td>False</td><td><code>^\s*""".*$</code></td></tr>
<tr><td>class definition</td><td>True</td><td><code>^class .*:.*$</code></td></tr>
<tr><td>Function definition</td><td>True</td><td><code>^\s*def .*:.*$</code></td></tr>
<tr><td>try statement</td><td>True</td><td><code>^\s*try\s*:.*$</code></td></tr>
<tr><td>finally statement</td><td>True</td><td><code>^\s*finally.*:.*$</code></td></tr>
<tr><td>assert statement</td><td>True</td><td><code>^\s*assert .*:.*$</code></td></tr>
<tr><td>with statement</td><td>True</td><td><code>^\s*with .*:.*$</code></td></tr>
</table>

Table 6.2: List of defined line types that can be altered and targeted by the GI

Line type name	**Indent following line/s**	**Regex pattern**
Generic	False	`^.*$`
Return	False	`^\s*return\s*.*$`
Comment	False	`^\s*#$`
If statement	True	`^\s*if .*:.*$`
else statement	True	`^else\s*:.*$`
elif statement	True	`^\s*elif .*:.*$`
for loop	True	`^\s*for .*:.*$`
while loop	True	`^\s*while .*:.*$`
except statement	True	`^\s*except .*:.*$`

Table 6.3: Sets of single operators and constants available to the GI. One member of a given set can be changed to another member of the same set.

Description	Set of operators and constants
Numerical constants	Can increment by ± 1
Arithmetic operators	$+, -, *, /, //, \%, **$
Arithmetic assignments	$+=, -=, *=, /=,$
Relational operators	$<, >, <=, >=, ==, !=,$ is, is not, not
Logical operators	and, or
Logical constants	True, False

6.3.1 *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. For the experiment in this chapter the implementation uses only the first type of mutation illustrated in Figure 5.1 that is applied to parents to produce offspring: Append a generated edit 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. We do this for every possible single-edit change to the original to exhaustively record the neighbourhood fitness. From there on a random walk is implemented by incrementally adding a single edit to

the edit list by using *Grow*. There is no selective pressure to search and the edit list is reset to a single random edit when these two conditions are met; maximum size of edit list and minimum fitness. In general terms such a random walk is not new and has frequently been used in analyses of various problems in physics and computer science [186].

A program's minimum fitness is zero, which can be obtained by failing to produce correct output on any input, throwing an exception, or not halting for every test case. The GI terminates programs that do not halt after a specified execution time and assumes they have failed that particular test case. Defining test suite $\hat{T}$ and T_i as test case i and n is the total number of test cases, then $T_i = 1$ if the variant passes that test otherwise it is zero. Given those definitions the fitness function ($0 \leqslant f(x) \leqslant 1$) for variant x can be formalised like in equation 6.1

$$f(x) = \begin{cases} 0 & \text{if not halting or compiling,} \\ \dfrac{1}{n}\displaystyle\sum_{i=1}^{n} T_i & \text{else} \end{cases} \tag{6.1}$$

6.4 EXPERIMENTAL SETUP

The experiments are divided into two parts:

RANDOM WALK ANALYSIS where starting from the assumed correct program and the fitness is monitored as the edit list incrementally grows by one edit at a time.

EXHAUSTIVE NEIGHBOURHOOD ANALYSIS where we evaluate the fitness of every first order mutant of the original program. A single edit change to a program is called its first order mutant.

The former part explores the fitness landscape globally, while the latter will indicate the local gradient the GI has to work with.

6.4.1 *Random walk analysis*

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 while the distance is measured in the number of edits applied to the original to obtain the variant.

The experiment is repeated 100 times by initiating an edit list x with a single randomly chosen edit and then appending it to the list incrementally. The pseudo code for the experimental process can be seen in procedure 1. Apart from recording the fitness $f(x)$ for each added edit in every experiment, three metrics (Δ, Ω and Ψ) are recorded for each run. These are the size of the edit list ($|x|$) when the fitness:

Procedure 1 A single experimental run for a random walk

```
1: MinX ← 20        {Minimum size of edit list}
2: MaxX ← 50        {Maximum size of edit list}
3: F ← empty array        {list of fitnesses}
4: x ← [random   edit]        {edit list}
5: for i = 0 until i ⩽ 50 do
6:    append f(x) to F
7:    if |x| ⩾ 20 and f(x) = 0 then
8:       break
9:    end if
10:   append [random   edit] to x
11: end for
```

- decreases for the first time Δ:

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

- reaches zero Ω:

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

- starts to increase again Ψ:

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

These measurements provide us with data to empirically explore the nature of the relationship between fitness and the size of the edit list for the three programs described in Section 6.4.3. Each program is accompanied by a test suite of different sizes so the fitness is normalised to represent the fraction of test cases passed.

6.4.2 *Exhaustive Local Neighbourhood Analysis*

For the second series of experiments each program is systematically mutated. The complete neighbourhood of the original program at distance one (first order mutants) is generated. This comprises of a variant for every possible replacement of every identified operator. Then each test case is individually recorded passed or failed for those variants. The evaluation of a mutated program returns a list containing ones and zeros of (passed, failed) for each test case. The same index in any two lists refers to the same test case so it is possible to compare the entire neighbourhood on a case by case basis. Fitness is a vector as opposed to a single measurement.

6.4.3 *Description of the programs targeted by* GI

The three programs summarised in Table 6.4 were selected for the experiment, their source code can be found in Listings A.1– A.3. They are all comprised of 3 or fewer python functions. They are not complete Python modules but are either part of a module, like **P2** or a standalone functions 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, digit) bins and calculates a result using a reverse Polish notation. It is a beginner's programming exercise and has proved difficult for automatic methods to evolve from scratch [212]. It is the only program of the three that is made specifically for this chapter's experiments. It branches out for each of the four basic arithmetic operations; addition, subtraction, multiplication and division as well as a special branch for parentheses. This adds redundancy to the source code but allows us to observe partial failures and a partition of the test suite.

P2 is an initialisation function for the K-means clustering method and is a part of the Scikit-learn Python toolbox [149]. It determines the initial k centres for the algorithm. **P2** does this with a random number generator that can be seeded for consistency.

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. It implements a purely string manipulation method unlike **P1** and **P2** that are numerical and mathematical in nature.

Table 6.4 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 6.3 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.

Table 6.4: Information about the programs that were used in the experiments

Program	Total lines of code (*)	Total number of mutable points	Mutable points by type (Table 6.3)	Size of distance -1 neighbourhood	Input ⇓ Output	Description
P1	143 (98)	147	S1: 19 S2: 37 S3: 50 S4-6:41	376	String ⇓ Float	Simple text input calculator
P2	177 (75)	106	S1: 11 S2: 14 S3: 57 S4-6:24	317	Cluster data** ⇓ Matrix	Initiation of K-means cluster centers
P3	66 (63)	59	S1: 26 S2: 3 S3: 29 S4-6:1	401	HTML ⇓ Latex	Html to Latex conversion tool

*Lines of code that the GI was allowed to modify.

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

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 with replacement from the inputs of the original tests and using **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 $B = \{+, -, *, /\}$. The set of numbers was chosen because we can use it to test boundary conditions with an *additive identity* (0), a *multiplicative identity* (1), an odd number (3), and an even number (4). Adding more numbers would increase the evaluation time but would be unlikely to increase the information provided with larger test suite. Two categories of test cases were made:

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

b) All combinations of $(X o_1 Y) o_2 Z$ 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.

The time it takes any of the original programs to execute its full test suite is on average under a second so the limit for execution time is set to 3 seconds. This was found to be enough to account for any delays due to other processes on the testing machine.

6.5 RESULTS

We consider four points for analysis in the following subsections:

1. Size of edit list versus a specific change in fitness
2. Average fitness as a function of edit list size
3. Unique and discrete steps in fitness
4. Every possible combination of pass and fail for the test suites and single-edit program variant

When comparing the mean of two metrics we use Welch's t-test for statistical significance between two independent samples with unequal variance.

HYPOTHESIS FOR WELCH'S T-TEST The means (μ) of a given metric for program i and program j are equal.

$$H_0 : \mu_i = \mu_j$$

$$H_1 : \mu_i \neq \mu_j$$

For testing the likelihood of two metrics coming from the same distribution, we compute a two sample Kolmogorov-Smirnoff statistic.

HYPOTHESIS FOR KOLMOGOROV-SMIRNOFF Metric X_i from program i and X_j from program j are drawn from the same distribution D.

$$H_0 : X_i \sim D \text{ and } X_j \sim D$$
$$H_1 : X_i \sim D \text{ and } X_j \sim G \text{ and } D \neq G$$

H_0 states that the metric for both program i and j are drawn from distribution D. H_1 states that the metric for i is drawn from distribution D, the same metric for j is drawn from G, and G and D are not the same distribution.

6.5.1 *Size Versus Change in Fitness*

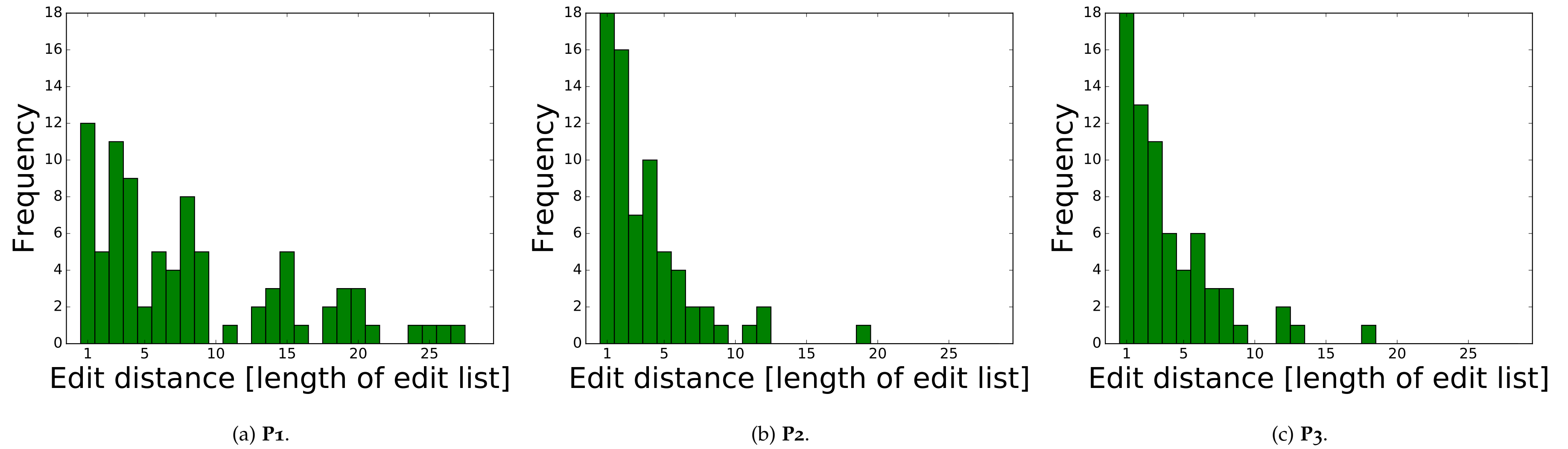
Table 6.5 lists the basic descriptive statistics for the edit list size ($|x|$) for the three different changes in fitness (see Section 6.4) and the total number of fitness evaluations for each program. The number of evaluations varies between programs due to the termination conditions which are to continue adding an edit to the list until either it is of size 50 or if it has fitness 0 after it grows to size 20 as described in Section 7.4. Firstly we measure the edit distance required for the fitness to decrease for the first time (i.e. $f(x) < 1$).

Table 6.5: Statistics for the variables defined in section 7.4, 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	

For **P1**, Δ occurs on average when the edit list is 9 edits long, although there is a lot of variation as shown by the standard deviation and the range $(1, 50)$. As seen in Figure 6.1 the mean is a poor measure of the average. Although programs are fragile to multiple changes,

there are a significant number of neutral paths that random walks can follow, even when the size of the programs is 100 LOC or less. 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** 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 6.1a and 6.1b. 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 6.1b and 6.1c are very similar. Both resemble the probability mass function of a geometric distribution where the probability decreases inversely by an exponential factor with respect to edit list size.

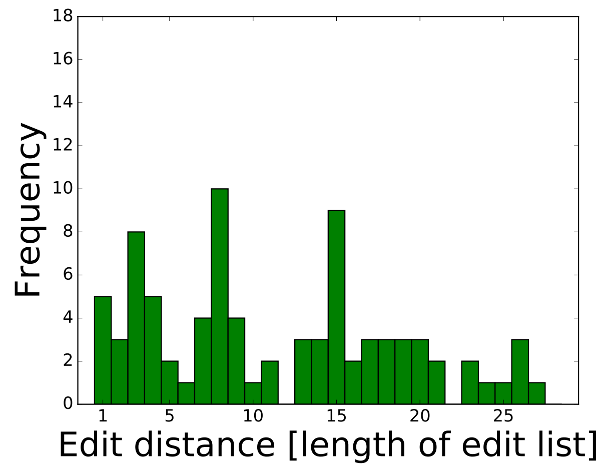


(a) **P1**.

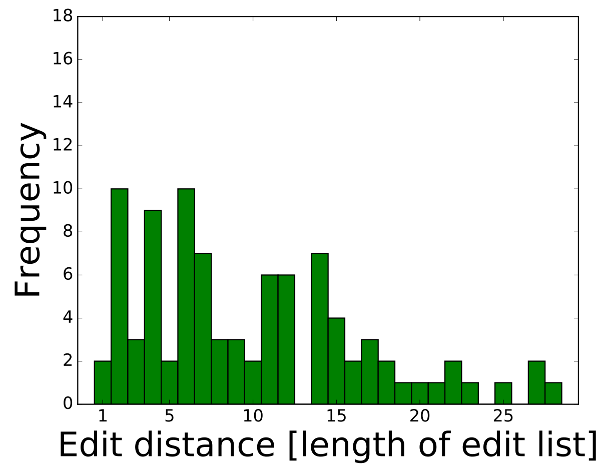
(b) **P2**.

(c) **P3**.

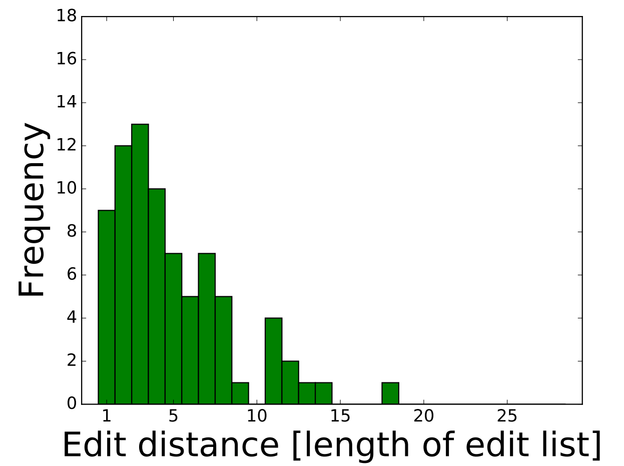
Figure 6.1: Distributions of the edit list size when the fitness decreases for the first time during the experiments for each program (Δ Table 6.5).



(a) **P1**.



(b) **P2**.



(c) **P3**

Figure 6.2: Distributions of the edit list size when fitness reaches zero for first time (Ω Table 6.5).

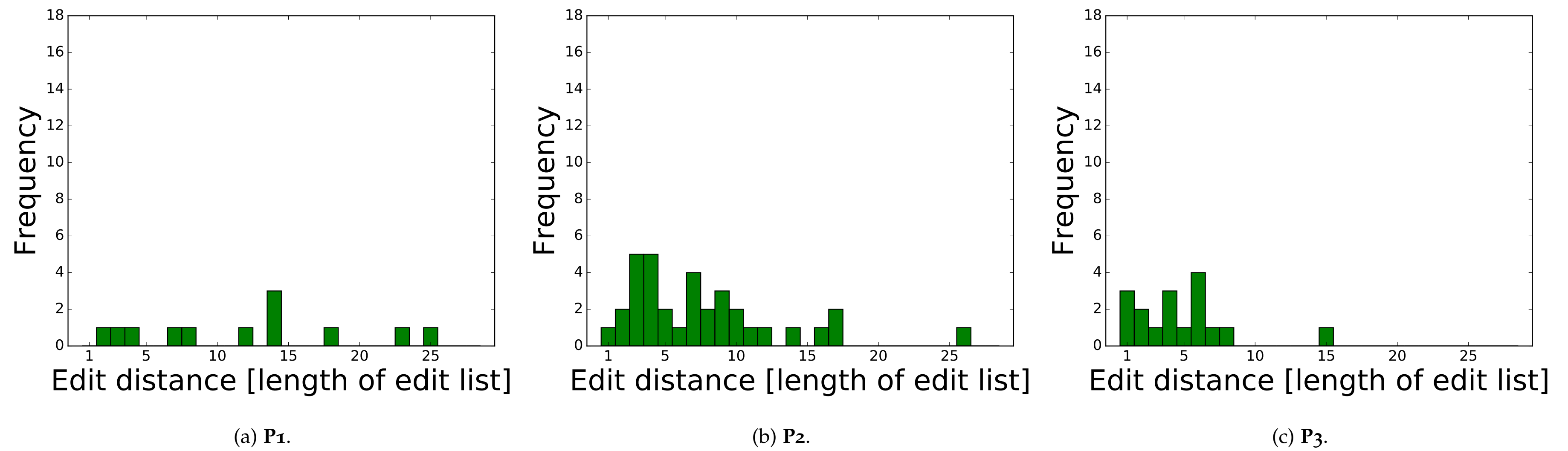


Figure 6.3: Distributions of the edit list size when (if) fitness starts increasing again (Ψ Table 6.5).

[18th October 2017 at 10:37]

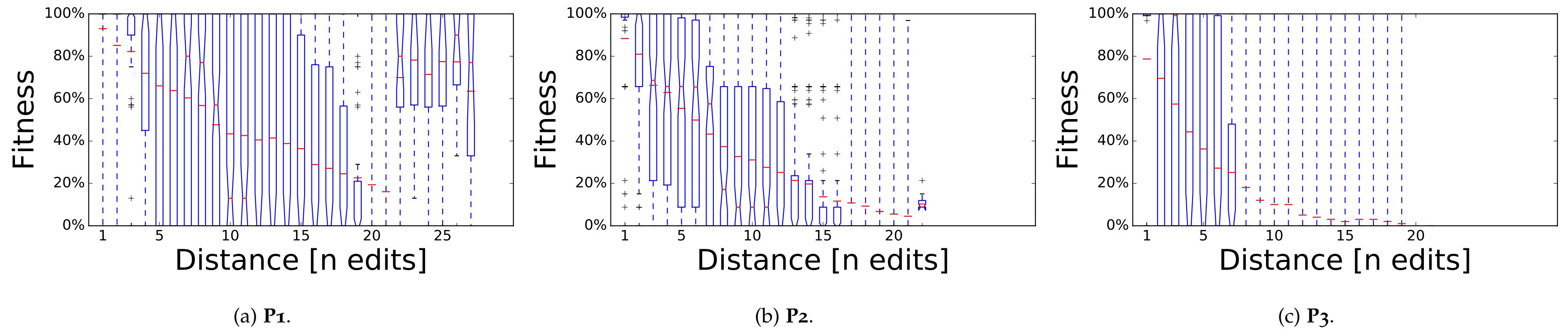
P1 and **P2** are however closer together when comparing the number of edits it takes to reach zero fitness and until the fitness might increase again. Testing for the same distribution gives $p = 0.08$ and $p = 0.10$ for Ω and Ψ respectively and we also cannot reject the hypothesis that the means are the same (0.08 and 0.09). Figures 6.2a and 6.2b corroborate the observation about zero fitness. However looking at Figures 6.3a and 6.3b 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 6.3a–6.3c. 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.

6.5.2 *Average Fitness with Respect to Edit List Size*

The experiments were repeated 100 times, from which fitness distance graphs were generated. These allow us to approximate the distribution of fitness for each increment in edit list size. Looking at the boxplots in Figures 6.4a–6.4c we see that overall, the distributions are quite different. **P1**'s first three increments (Figure 6.4a) 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 6.4b) from top 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 6.4c).

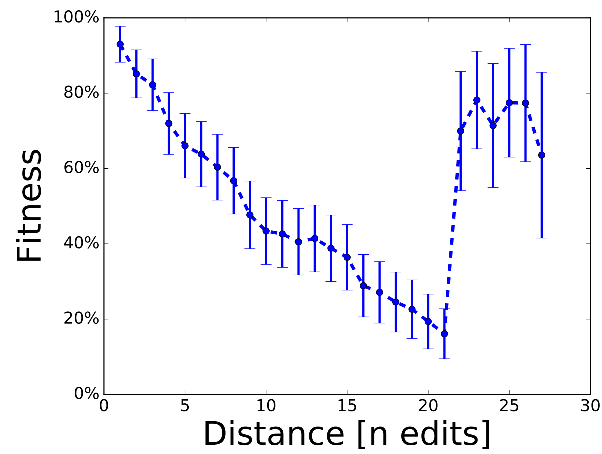


(a) **P1**.

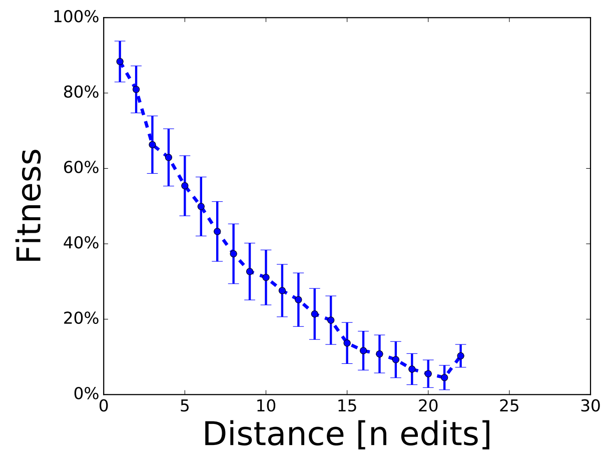
(b) **P2**.

(c) **P3**.

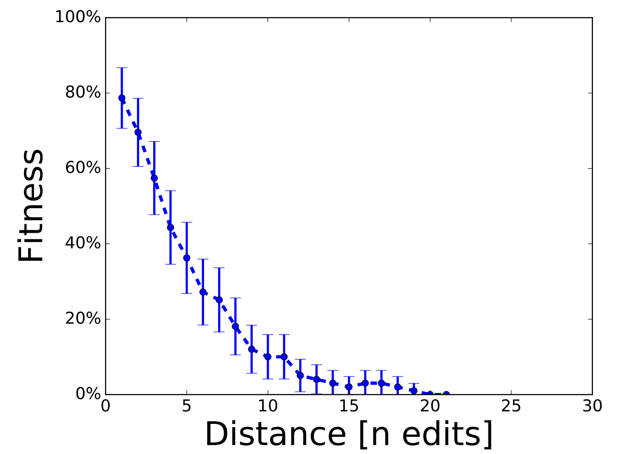
Figure 6.4: Fitness distribution with respect to edit list size for each increment. The tapering denotes the 95% confidence interval for the median fitness of each edit list size.



(a) **P1**.

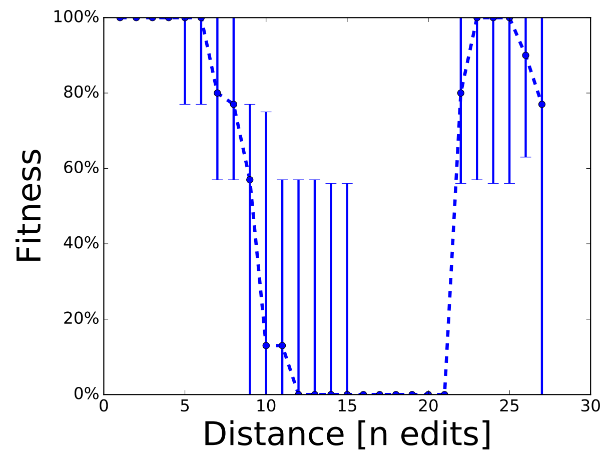


(b) **P2**.

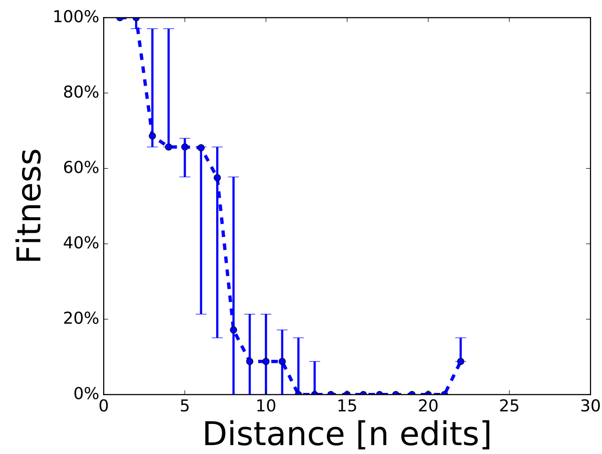


(c) **P3**.

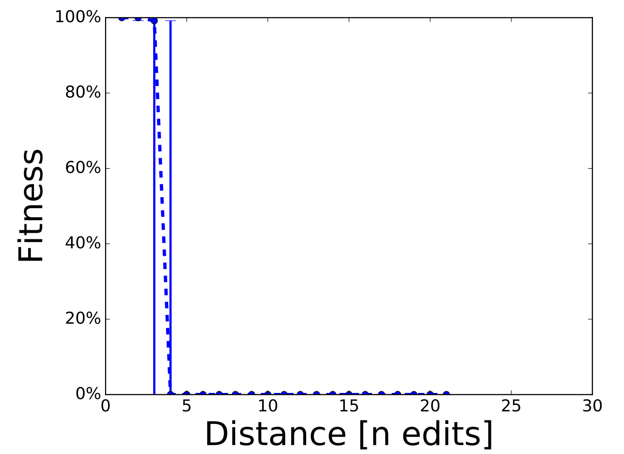
Figure 6.5: Mean fitness (95% error) with respect to edit list size for each increment.



(a) **P1**.



(b) **P2**.



(c) **P3**.

Figure 6.6: Median fitness (95% error) with respect to edit list size for each increment.

Having a closer look at how the mean fitness changes in Figures 6.5a–6.5c we see that these programs are as dissimilar as initially assumed. While both **P2** (Figure 6.5b) and **P3** (Figure 6.5c) both follow curves that are concave upwards, **P3** follows a sharper curve. Now **P1** is the outlier (Figure 6.5a) following a noisy line with a negative slope until the edit list reaches size 22 when it jumps up to $f(x) = 80\%$ again. This seems very unexpected as the subsequent edits start to repair the program but given the size of the program and the number of edits this is quite possible. **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 6.6a–6.6c paint a completely different picture, displaying no hint of smoothness to the transition from one increment to another. This, again, verifies that the mean is not a good indicator for the average of these measurements. However there are obvious steps that highlight the discreteness of each program's fitness function. We see in Figure 6.6b that **P2** has the most number of steps, while **P1** comes second (Figure 6.6a) and **P3** last (Figure 6.6a).

6.5.3 *Discrete steps in fitness*

Following the observation of the different steps for each program's median fitness seen in Figures 6.6a–6.6c, 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 6.7. **P2** had its fitness evaluated 2067 times, as seen in Table 6.5, and Figure 6.7 shows how often the fitness changed from one value to another by adding a single edit to the list. The histogram 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.

P1 has the second largest number of discrete steps, 15 as seen in Figure 6.8, the total number of counts is 2213 (Table 6.5). This chart looks like a scaled version of **P2**'s (Figure 6.7), 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.

Figure 6.9 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,

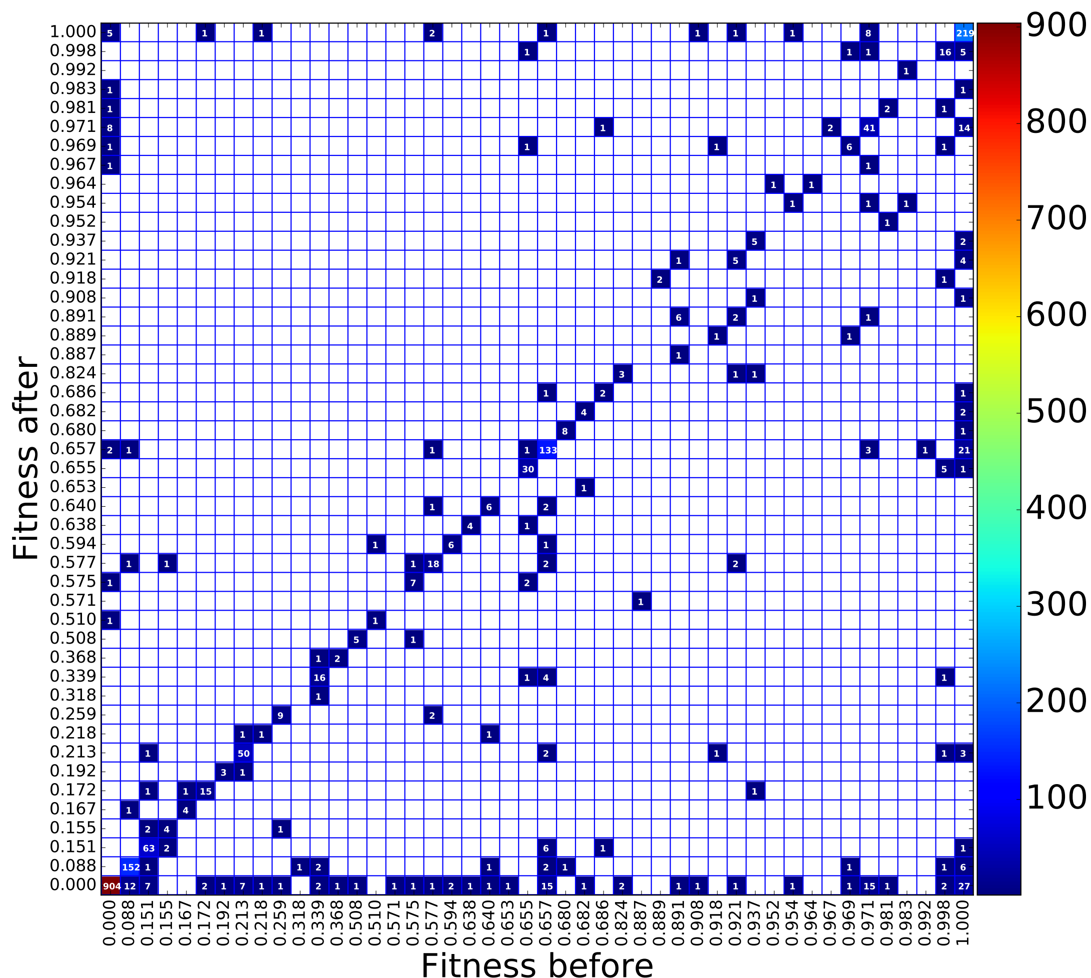


Figure 6.7: Frequency chart of fitness changes after a single edit is appended to the edit list for **P2**. Each square is a count of how often the fitness changed from fitness before to fitness after.

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.

6.5.4 *Exhaustive Neighbourhood Evaluation*

The exploration of the nearest neighbours to the original program revealed interesting information. Figures 6.10–6.13b are obtained by gathering the pass/fail lists for all the program variants in a matrix so that each row represents a program variant and each column represents a single test case. We then collapse the matrix so that each row and column are unique and keep count of how many instances each represents. The collapse is done by grouping together identical rows and columns, count the members of each group, and show only a single pattern but record the counts. Those counts are on the y- and x-axes, indicating the number for each pattern of program variants and test cases respectively.

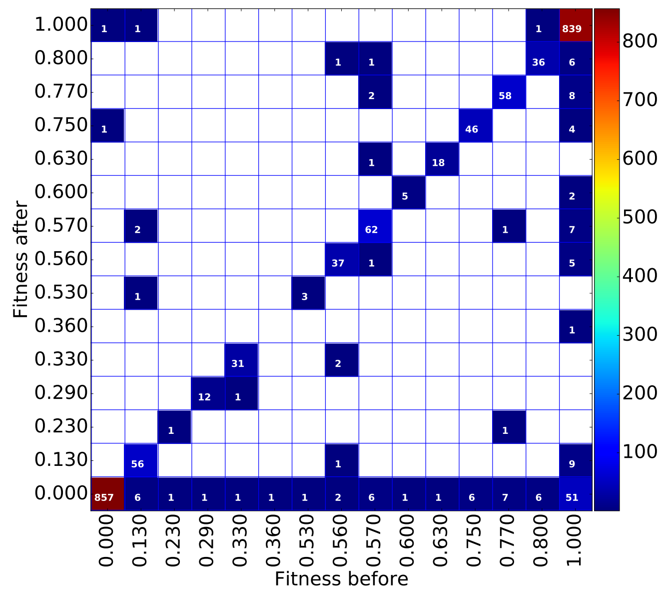


Figure 6.8: Frequency chart of fitness changes after a single edit is appended to the edit list for **P1**. Each square is a count of how often the fitness changed from fitness before to fitness after.

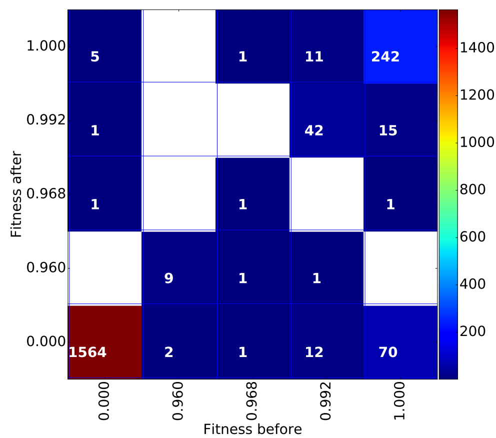


Figure 6.9: Frequency chart of fitness changes after a single edit is appended to the edit list for **P3**. Each square is a count of how often the fitness changed from fitness before to fitness after.

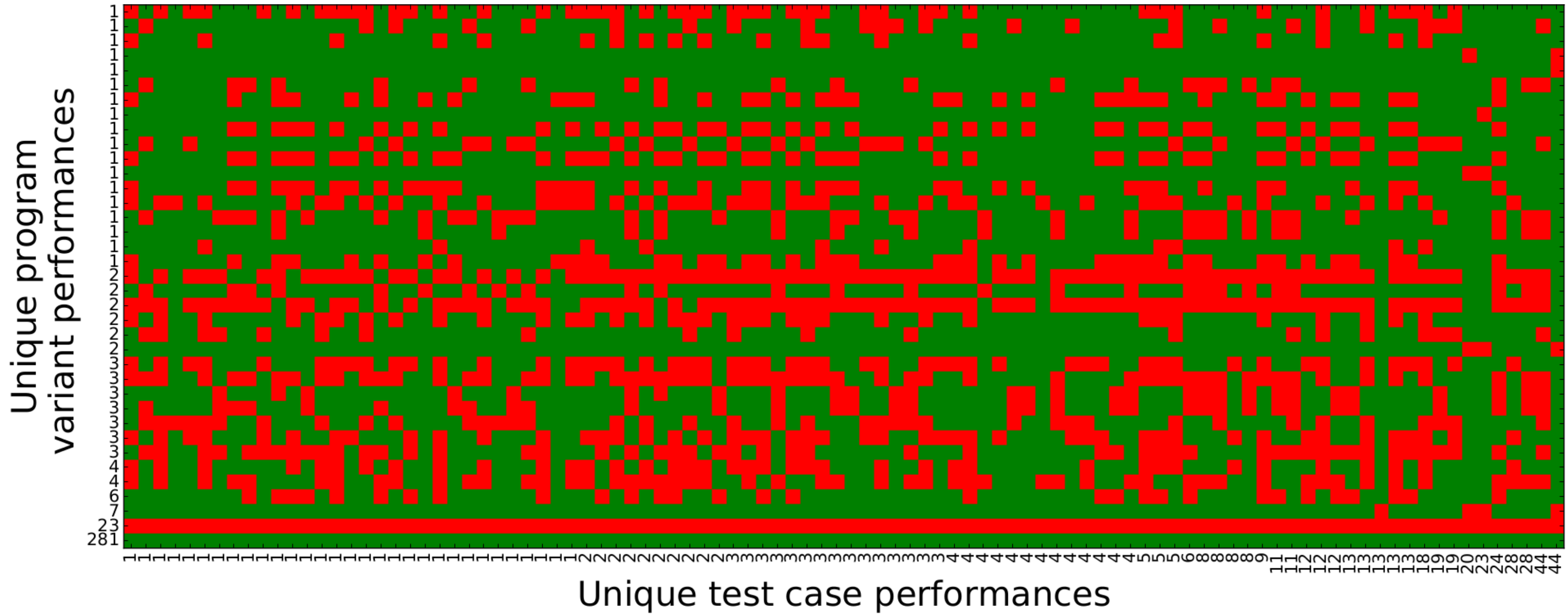


Figure 6.10: Unique patterns of passed/failed for every program variant (**P1**) and every test case. Green indicates a pass while red is fail. The columns and rows are sorted by the number of occurrences for each unique combination of passed/failed.

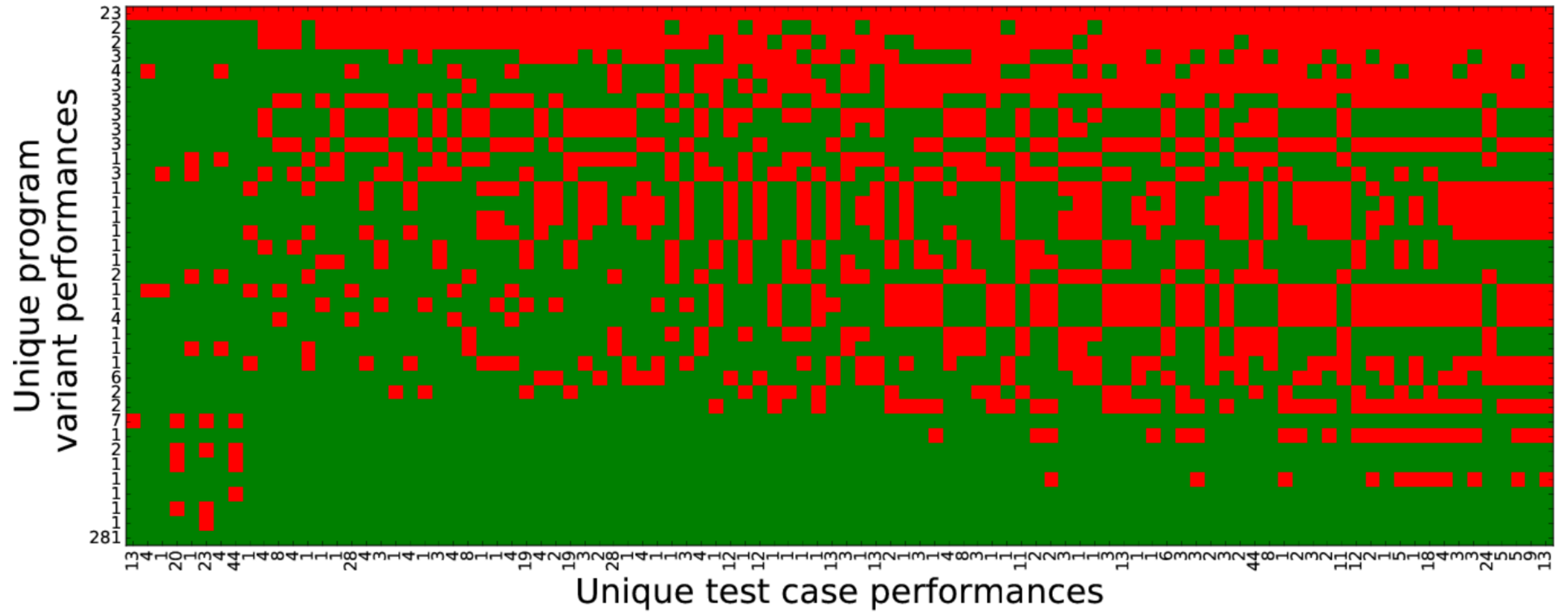
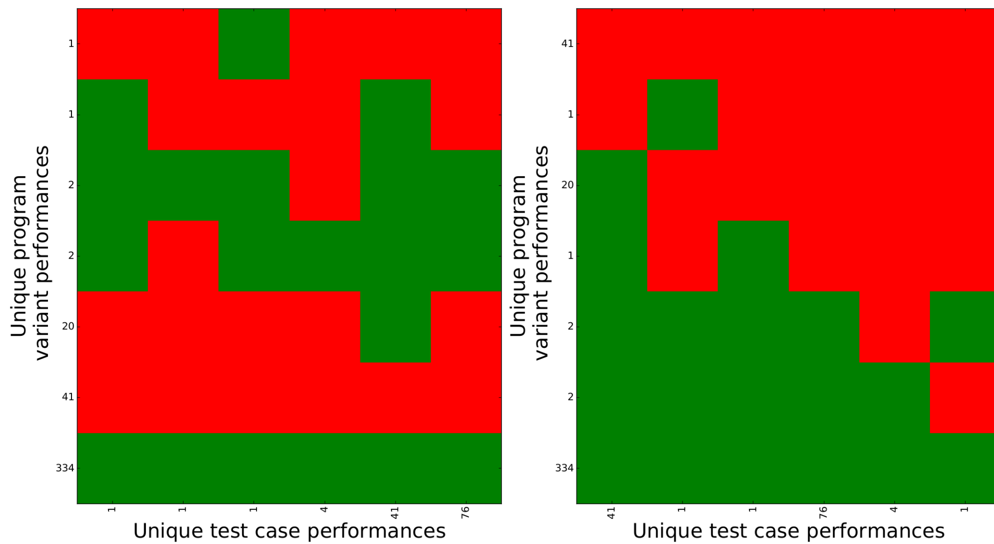


Figure 6.11: Unique patterns of passed/failed for every program variant (**P1**) and every test case. Green indicates a pass while red is fail. The columns and rows are sorted by fitness (maximum green) for each unique combination of passed/failed.



(a) The columns and rows are sorted by the number of occurrences for each unique combination of passed/failed.

(b) The columns and rows are sorted by fitness (maximum green) for each unique combination of passed/failed.

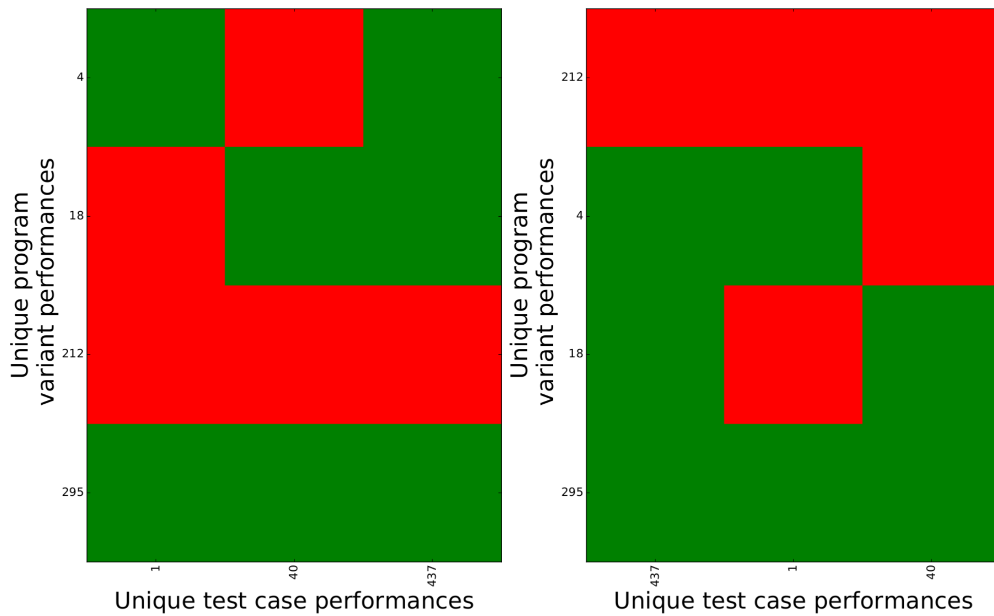
Figure 6.12: Unique patterns of passed/failed for every program variant (**P3**) and every test case. Green indicates a pass while red is fail.

Of the three programs **P1** has the most variation in its single-mutation programs (Figures 6.10 and 6.11), with 37 unique combinations. However the number of unique fitness values is 14, one less than from Figure 6.8 meaning that one of the fitness steps needs at least 2 or more edits to be reached. **P2** has 4 unique combinations (Figure 6.13) and only 2 fitness variations, which is far less than the 46 in Figure 6.7. **P3** has 7 combinations (Figure 6.12 and 5 fitness variations.

Random mutations to **P1** and **P3** will most likely have no effect but **P2** is equally likely to fail all test cases. The immediate practicality of the results presented in Figures 6.10–6.13b is not apparent. They represent a condensed information about groups of test cases as well as program variants. The most obvious usage of such data is in test case selection problems. In situations where one needs to save resources they could largely decrease the number of test cases by selecting one from each group. Take for an example **P2**. The expanded test suite counted 500 test cases but according to Figure 6.13 only 3 would have sufficed, one from each group on the horizontal axis.

6.6 SUMMARY

This exploration of three Python programs' fitness distance provides us with a greater understanding of the search process encountered by GI. The experiments allow us to come to a conclusion for these three programs. In general we are unable to state the conclusions as



(a) Unique patterns of passed/failed for every program variant and every test case. Green indicates a pass while red is fail. The columns and rows are sorted by the number of occurrences for each unique combination of passed/failed.

(b) The columns and rows are sorted by fitness (maximum green) for each unique combination of passed/failed.

Figure 6.13: Unique patterns of passed/failed for every program variant (**P2**) and every test case. Green indicates a pass while red is fail.

affirmative because we do not have enough data. Additionally, we assume that the landscape has no inaccessible areas to our GI implementation. Since we are using the GI to break correct programs and concluding about the opposite search direction, we assume that bugs will be a variation of something that the GI can modify.

Revisiting the four questions asked at the beginning of this chapter and given our assumptions:

- We observe that it is feasible to apply GI to fix multiple bugs simultaneously. The search space from the original program to a variation is most likely contained within 10 edits. Starting from any variation of these programs and applying search pressure can result in a fix. However since the space has large areas that are flat and provide no gradient for the search to follow the search method has to contain a powerful escape mechanism. The GI framework in this thesis has essentially a repeated soft restart built-in by filling half of every generation with randomly generated new edit lists.

- If a fix needs multiple edits, GI will be able to find it. The same reasoning applies for this statement as the last. If it takes on average 10 edits to completely break these programs

(i.e. zero test cases passed), then if we are starting from a nearly correct program it is likely to be found within 10 edit distance.

- We have identified a similarity in fitness distance relationship that is worth exploring in more detail on larger set of programs. Programs are robust with respect to small changes in the source code. That is, most small changes will have no impact on the number of passed test cases. However, those changes that do affect it could possibly cause failure of all test cases.

Although we cannot conclude general rules or patterns from these experiments, the difference of the programs that were tested gives us hope that these patterns can be found in a larger set of programs.

It would be interesting to explore the cause of the difference in sparsity of the frequency matrices in Figures 6.7–6.9. 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.

7 REDUCING EXECUTION TIME WITH GI

7.1 INTRODUCTION

This chapter analyses the GI's capabilities to improve a non-functional property of ProbAbel [10], a bioinformatics program written in C. Non-functional properties have been of particular interest [200], mainly due to the popularity in mobile computations and the need to save resources [63, 23, 24].

Execution time has been a popular GI target, specifically for computationally expensive programs for bioinformatics. Target software has included Bowtie 2 [109] which is used to align genome sequences and the sequence mapping software BarraCUDA [113]. The traditional program targeted by GI is relatively large (>10K lines of code), with a few exceptions [201, 153].

We consider how well the GI is able to target execution time. We also briefly analyse the landscape of the non-functional property improvement which is measured with a continuous variable from $\mathbb{R}$. Our intentions are to informally compare this landscape with that of bug fixing which is represented with a discrete variable from $\mathbb{N}$. 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 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 the overall gain is accumulated for every execution (eq. 7.1).

$$\Delta = Y + \sum_{i=1}^{n} (X - X') \tag{7.1}$$

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 remainder of this chapter is as follows: Section 7.2 gives a short explanation of ProbAbel's functionality. Section 7.3 describes the data set that was used for the experimentation and how it was generated. Section 7.4 lays out the experimental procedure. Section 7.5 details our results. Lastly, Section 7.6 summarizes and concludes the chapter.

Part of this chapter's work has been published in Haraldsson *et al.* [68]

7.2 PROBABEL: A BIOINFORMATICS SOFTWARE PACKAGE

ProbAbel is a specific piece of software widely used in bioinformatics for Genome-Wide Association (GWA) studies. It is a collection of programs for regression models; linear, logistic and Cox proportional hazard. A GWA study is the analysis of genetic variants in groups of people. The purpose is to identify which variations, if any, are associated with a certain trait or disease. An examples of such studies include identifying correlations between genome locations and known risk factors for coronary artery disease [190] and cardiovascular disease [124]. 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 [130, 191] but the details of the process is beyond the scope of this thesis. 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 variation 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 [124]. 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.

ProbAbel is versatile and relatively fast because it uses estimations and floating point data 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 increased risk of many diseases or conditions. Typically the data consists of approximately 30 million SNPs from 10-20 thousand people. Each run of the software can take up to 12 hours as reported by researchers in the Icelandic Heart Association.

ProbAbel is written in C and C++ and utilises the R project's [160] *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.

Table 7.1: 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

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 7.1 lists these files, their sizes and the number of operators that were marked as being changeable given the mutation operators in Table 7.5. The program can be changed in multiple ways but assuming we only consider the sets of operators from Table 7.5 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 lines within the source.

Table 7.2: Sampled distributions for 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	$\mu = 55$ $\sigma = 15$ $a = 10, b = 99$

7.3 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 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 confounding variables such as these. 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 less than a second, most of which is 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. Each trait (height, age and sex) was sampled independently to avoid having identical samples in the generated data set and to emulate a sample of real life population. This sampling scheme has more possibilities for the trait combinations than either sampling from a conditional probability distribution or bootstrapping from the example data. Sampling by bootstrapping would generate a population that has multiple people with identical traits. Furthermore, conditional probability distribution biases towards homogeneous population, especially when constructing the distributions from such a small group. Nonetheless, sampling the traits independently ensures that the underlying distribution for each trait is the same as that from the example data. Table 7.2 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 the nucleobases: adenine (A), cytosine (C), guanine (G), and thymine (T). 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. The detailed description of these variables is not within the scope of this thesis and is not necessary for applying GI to ProbAbel. There is generally no need to be a domain expert or have intricate knowledge of the end-user purpose of the software to be able to utilise GI.

Table 7.3 lists the 7 generated data sets; their size and average execution time for the original program and two best GI variants as trained on data set **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 7.1 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. Both program variants exhibit the same behaviour.

Table 7.3: Seven different data sets of different sizes (population and SNPs). Execution time is measured in seconds (CPU) and averaged over 20 test runs for each, data set and program variant. For the program variants the p-value of the Student's t-test for two independent variables is also listed. Each variant is tested against the original.

Data set	Number of People	SNPs	Original program	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)

7.4 EXPERIMENTAL SETUP

The experiments were conducted to answer the two questions from Section 7.1:

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

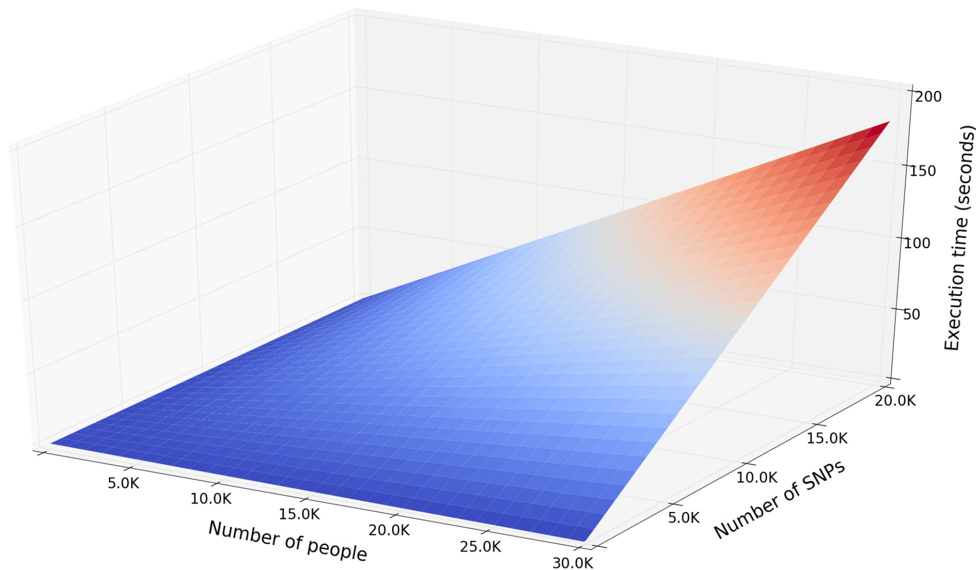


Figure 7.1: Execution time of the original program with respect to data set size; number of people, and SNPs. The graph shows a surface that is approximately linear with respect to both independent variables. Both variants,1 and 2 produce near identical figures as suggested by the values in Table 7.3

ensuring that we only had to compile part of the software and decreasing the overhead time for every evaluation considerably, from approximately 50 seconds for all models 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 7.3).

Statistical tests were run to determine whether the variants performed better than the original. For each set of results we give the outcome of a two-tailed Student t-test where:

H_0: The means are equal.

H_1: The means are different.

The significance level is predetermined and set at 5%, so we reject H_0 if $p < 0.05$.

7.4.1 *GI Parameters*

The first part of the experiment, unlike the setup for Chapter 6, uses the framework (Chapter 5) to its full extent and searches with a population based evolutionary algorithm and parameters as listed in Table 7.4. No attempt was made to optimise the parameters since one of the goals was to apply the framework on a different programming language with minimal effort. The improvement process iterates for 50 generations with a population size 40. The entities being

evolved are, like before, edit lists as described in Chapter 5. We use both types of edits: *Macro* (moving whole lines) and *micro* (changing a sub-string of a line). We provide the framework with the sets of C syntax operators in Table 7.5 for *micro* edits and the line types in Table 7.6 for *macro* edits.

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, making half of the generation. 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. However it might also promote homogeneity if the newly generated edit lists all have lower fitness than the lists that are there from before.

Table 7.4: GI parameters.

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

Table 7.5: Sets of single operators available to the GI. One member of a given set can be changed to another member of the same set.

Description	Operations
Numerical constants	Can increment by ± 1
Arithmetic operators	$+, -, *, /, \%$
Arithmetic assignments	$+=, -=, *=, /=,$
Incremental operators	$++, --$
Relational operators	$<, >, <=, >=, ==, !=$
Bit assignments	$\&=, \mid=$
Bit operators	$\&, \mid$
Logical constants	True, False

Table 7.6: List of defined line types for improving *ProbAbel*

Line type name	Can be altered or moved
Empty	False
Include	False
Function or class definition	False
Compiler macro instructions	False
Log statement	False
Comment	False
Generic	True
If statement	True
else statement	True
else if statement	True
for loop	True
while loop	True
Return	True

7.4.2 *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 background CPU 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 contains 52 tests and for each failed test case a proportion of the original execution time is added to the fitness evaluation. So for each failed test case $1/52$ of the original execution time is added to the 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 occupies the CPU. 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 7.2.

7.4.3 *Exploring the Execution Time Landscape*

There is essentially no limit to the number of combinations a variable length list of lines can represent but there are a finite number of mutable points in the program. Therefore, to explore the landscape of ProbAbel's mutants, we consider only *micro* edits, as adding the *macro* edits would expand the search space considerably. The second part of the experiment is a random walk starting with the original. This process 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 Chapter 6. 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.

7.5 RESULTS OF EXECUTION TIME IMPROVEMENTS

When the GI finished, it had evaluated in total 2000 edit lists, 240 of which were duplicates (two or more identical) and 1760 were unique. 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

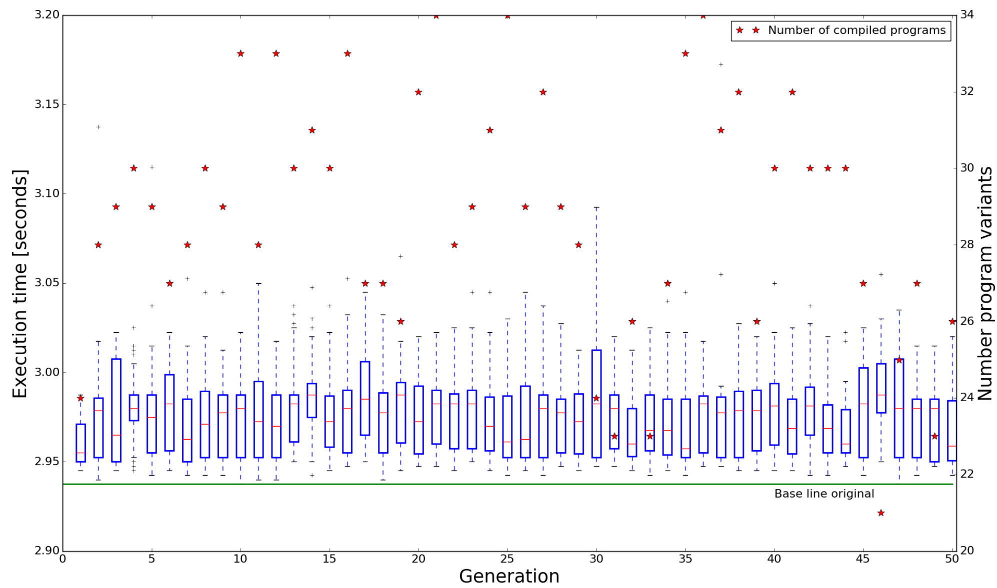


Figure 7.2: Distribution of ProbAbel's fitness and the number of compiled variants for each generation. *Left axis*: The execution time and the boxes are the distributions of mean execution times for each generation. *Right axis*: Number of program variants and the stars are the number of compiled variants.

execution time was **2.957 (SD 0.012)** seconds (*Variant 1*) on data set D3 and the next best executed on average in **2.959 (SD 0.010)** 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 line performs arithmetic operations on a variable (*chi2*) that contains a matrix. 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;">>>
```

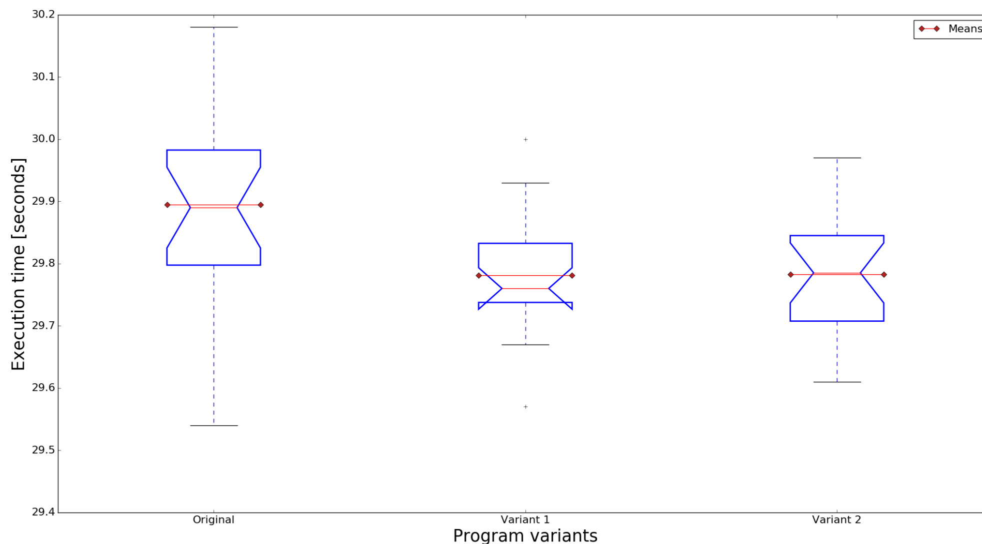


Figure 7.3: Distribution comparison of the execution time of the original and the two best variants on D5. Each box is constructed from 20 runs. The notches in the boxes are the 95% confidence interval for the median of each.

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. These two lines are never executed when *ProbAbel* runs a linear regression and are already excluded from compilation with C macro instructions. The two micro edits change 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 7.3 neither of these variants' mean execution time is significantly different from the original on **D3**. They were significantly better on **D4** and **D5** only and the difference is quite small, less than 0.5%. As we further confirm in Figure 7.3 where we see that the difference in means is small but the confidence intervals of the medians do not overlap. Looking at Figure 7.2 we see that there is minuscule variation in the 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).

On two occasions the GI found *loopholes*, e.g. changed an `if` statement such that the program read in much less data than was given to it. This yields a significant reduction in execution time, approximately 97%, but the resulting coefficients of the linear models differed from the original's results by approximately a factor of eight. The way in which the GI was instructed to read from the test log files allowed for that loophole. The penalty scheme, which assumed that there were 52 tests, only registered 4 failed test cases and ignored the other 48 that failed

as well. These two "cheats" the GI found were not included in any figures or tables for two reasons:

- They would have skewed the scales and obscured the improvements of the two "good" variants
- They were a result of a poorly implemented test evaluation by the GI researcher

However, the fact that the GI found these "cheats" stresses the importance of proper implementation of constraints and being aware of any contingencies.

7.5.1 *Random Walk Exploration of Execution Time*

Figure 7.4 shows the 15 levels of execution time that the generated ProbAbel variants exhibited. The graph demonstrates frequency of transitions from one execution time performance to another when 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 compiled variant. As seen in Figure 7.4, most often a single edit caused a compilation error. Otherwise the execution times of the variants are evenly spread over the range of possible times. 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% of the 900, not counting the first order mutants.

Figure 7.5 shows that the execution time does not get worse as we travel further away from the original. However, the proportion of program variants that do compile without errors decreases rapidly until it reaches zero after 9 steps.

7.5.2 *Local Neighbourhood Exploration of Execution Time*

An overnight run generating first order mutants produced 2400, of which 2265 were unique. 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 7.6. It is interesting to see that, although the execution time does not improve with first order mutants, it does not increase considerably. The implications are that execution time reductions can probably not be achieved with a single edit. This contrasts with the effects of single edits when fixing bugs.

7.6 CONCLUSION

In this chapter 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

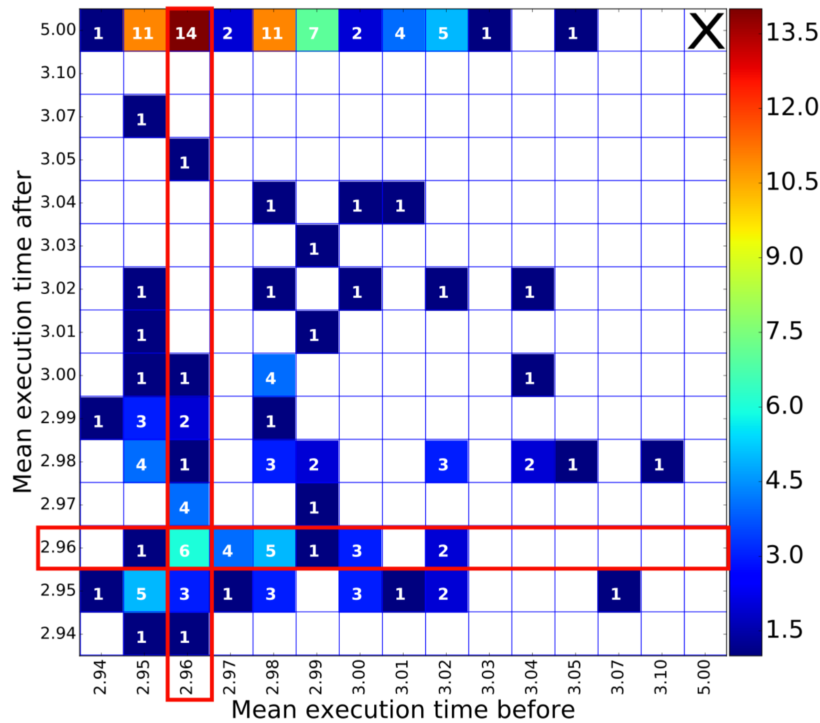


Figure 7.4: Execution time before and after a single edit is appended to the edit list and evaluated on D3. Each square contains a count of how often the execution time changed from before to after. Row and column marked with red is equal to the original execution time. Execution time of 5 seconds denotes an uncompilable program variant and X is the omitted count of 748.

to operate on C included only small changes to the class of operators in Table 7.5. Therefore we can apply this same approach to many other programming languages.

Our intentions with this chapter were to answer the two questions in Section 7.1. The answers we have arrived at are:

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 (Figure 7.2) but largely flat with the occasional drops and peaks. Our findings are complementary to the statement that "Software is Not Fragile" [111]. The majority of the first order mutants (1622) compiled and executed without an error.

The GI framework introduced in Chapter 5 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 7.4. However, the 0.5% execution time decrease can translate into hours of saved time in the long term. The application of 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.

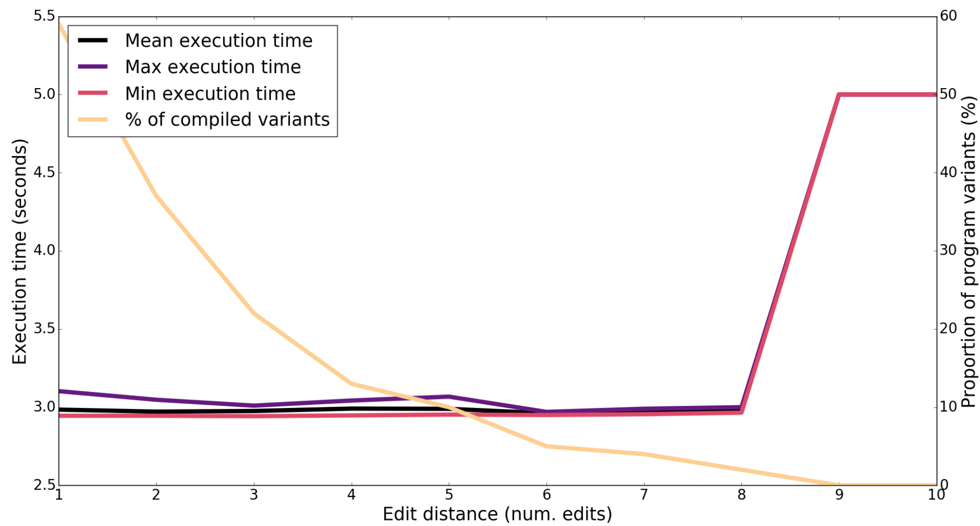


Figure 7.5: *Left axis*: The change in execution time as the program variants move away from the original. Mean, maximum and minimum execution time for 100 traces. *Right axis*: Proportion of program variants that compiled without errors.

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 7.5 exceeds 10^{2683}. In our experiment we explored so little of the search space 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? We cannot be entirely sure. However, it can be argued that the explored landscape is representative of a trajectory from the original towards an improved version. Therefore 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.

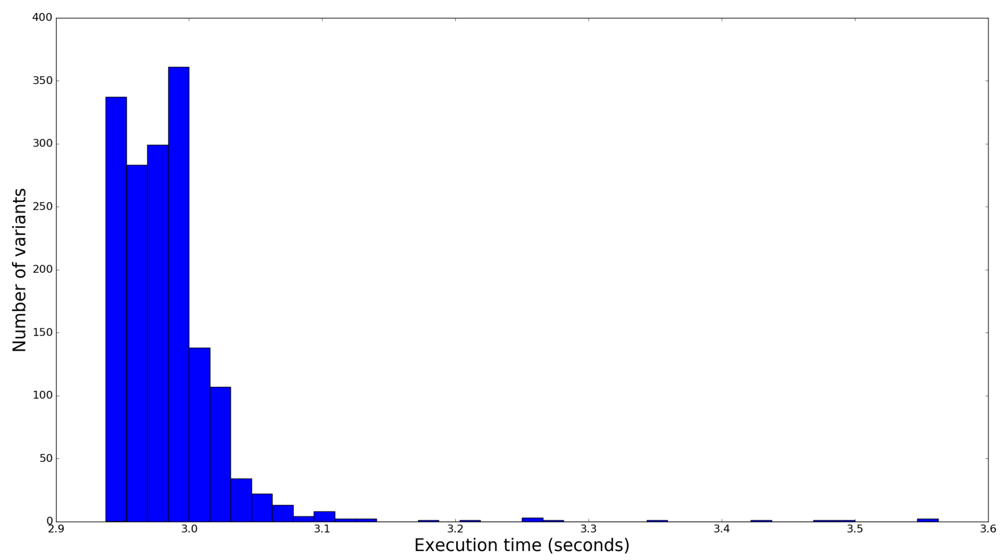


Figure 7.6: Distribution of the execution time within a single edit from the original program.

8 DEPLOYMENT OF GENETIC IMPROVEMENT IN A LIVE SYSTEM

8.1 INTRODUCTION

Chapter 7 expands upon Chapter's 6 groundwork for fundamental analysis into GI and the search space of bug fixing, by applying the same methods on a C source code to reduce execution time. We saw that landscapes, on both occasions, are predominantly flat and therefore might be difficult to search. Furthermore, the likelihood of finding an improvement with evolutionary search methods increases considerably if given enough time. Harman et al.'s suggestion for *Dreaming Devices* [79], allows room to provide GI process with adequate time, exploiting idle time of software that is not in continuous use. Even if that were the case, the usage load will usually be periodic and during lower load times the device should be able to afford some capacity for improvements.

Opportunities for using GI to make dynamic adaptive software are copious, there are various methods to implement and there is a vast number of available software applications. Despite its growth [154] within academic research the practical usage of GI has not yet followed. Like with many SBSE applications, the software industry needs an incubation period for new ideas where they come to trust in outcomes and see those ideas as cost effective solutions. GI is in the ideal position to shorten that period for the latter as it presents a considerable cost decrease for the software life cycle's often most expensive part: maintenance [121, 66]. There are examples of software improved by GI being used and publicly available [113] which is impressive considering how young GI is as a field. In time it can be anticipated that we will see tools emerging that utilise current advances in GI for various improvements during different stages of development; from early coding where programmers might want to statically monitor the performance of their work to maintenance on bug fixing [5], automatically adding new functionality [132], and adding new hardware compatibility [105].

Traditionally GI has been used offline. In the lab the target program is copied, improved, and then the researchers have to convince developers to include the improvements in later releases. This is often difficult because researchers are impatient to move on to the next project. However GI's ultimate goal must be self-adaptive systems [35, 42, 74] with minimal manual effort by the developers to truly minimise software maintenance costs. Current research into self-improving systems, consists of early concepts [188, 31], identifying applicable ideas [79], and highly specified but truly dynamic approach [213]. The research and methods are slowly

approaching dynamic adaptive systems but a big obstacle is getting the results to market. To achieve that we need to take it in steps, first we provide developers with tools before we provide autonomous software.

In this Chapter we describe a live system, JM, that takes that first step. JM is a bespoke program for a vocational rehabilitation centre, developed and maintained by Janus Rehabilitation Centre (JR) in Reykjavik, Iceland. A small part of JM was used for Chapter's 6 experiments and here we describe how GI was integrated into the system. JM utilises the GI framework described in Chapter 5 in two ways:

- BUG FIXING JM monitors itself during daytime use and collects data whenever user input raises an exception. Overnight it uses data collected during the day to test itself and searches for adaptions that do not raise exceptions when similar data is submitted later. At the end of the nightly self-improvement process it presents the developers with a list of suggested bug fixes to fix the perceived fault.
- PREDICTION MODEL IMPROVEMENTS JM makes prediction about rehabilitation length and outcome to periodically provide the care givers with objective view of each client's current status. Whenever a client finishes treatment the system uses their anonymised data and the GI framework to update and improve prediction models for future clients.

Using the GI to assist with bug fixing significantly decreases the cost of maintenance after the initial release. It also allows developers to ultimately have the control and provide a sanity check before patches are issued to the live software. The prediction model improvements have provided the rehabilitation specialists with consistently accurate view of their clients' circumstances and progress. JM marries the concept of the *Dreaming Device* with Arcuri's co-evolutionary bug fixing [5, 8] and further adds usage evolution. The *Dreaming Device* has two separate phases: normal use when it keeps its resources available for the user, and dreaming state when it dedicates its resources for introspection and self-improvements. The co-evolutionary bug fixing evolves both program variants and test suites at the same time. It essentially maintains two populations of two different entities that challenge each other to improve. The addition of usage evolution, where the users' behaviour and interaction with the system changes over time, adds challenges made by a human in the loop effects.

The remainder of the Chapter is structured as follows. Section 8.2 details what the system does during business hours and how it keeps records for later improvements. Section 8.3 explains how the daily data is used to improve the software system including generating test cases and the GI stage. Section 8.4 describes the implementation of the prediction mechanism and how the GI is used to improve it. Section 8.5 reviews the performance of both the bug fixing and prediction enhancements since the launch of JM. Section 8.6 summarises the chapter.

Part of this chapter's work has been published in Haraldsson *et al.* [69], Haraldsson *et al.* [67] and Haraldsson *et al.* [65]

JM is a software system that supports the vocational rehabilitation process and internal communications. Its creation was initially motivated by JR's need for specialised data management and statistical analysis for a rehabilitation service. The functionality was simple to begin with (Figure 8.1a) but the lack of specialised software for complete management of rehabilitation services further drove the development (Figures 8.1b – 8.2a) to produce today's version as seen in Figure 8.2b.

Moreover, JM is a tool for the managers to be able to continually improve the rehabilitation process with statistical analysis of client data and performance of methods and approaches. It has to manage multiple connections between users, specialists and clients.

8.2.1 *Usage of Janus Manager*

The left side of Figure 8.3 displays the daytime normal usage of JM. The users are all employees of JR, over 40 in total, including both specialists and administrators. They interact with JM by either requesting or providing data which is then processed and saved. Example request are:

- Internal communications between the interdisciplinary team of specialists about clients.
- A journal record from a meeting, or an update to some information regarding the client.
- Logging of attendance to scheduled courses, interviews, or events related to the rehabilitation.

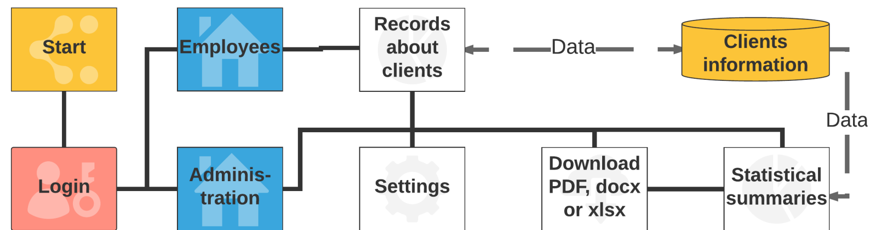
The system can also produce reports and bills in PDF format or rich text files (see Figure 8.2).

The clients have access to specialised and standardised questionnaires that measure various aspects of the clients' welfare and progress. The specialists then use the results from those questionnaires to plan a treatment or therapy. Additionally it uses the complete database of information to identify risk factors of unsatisfactory treatment results, and predict the possible outcomes of treatment and its length [65].

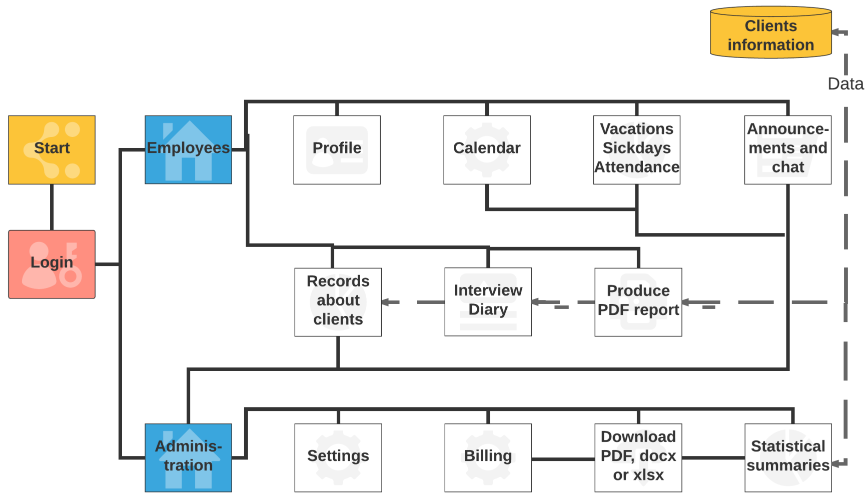
8.2.2 *Daytime Usage Monitoring*

Every time an input data causes an unintentional exception to be thrown, JM logs the trace, input data and the type of exception in a daily log file shown in the middle of Figure 8.3.

A typical log would look similar to that in Listing 8.1. Starting with the exact time the exception occurred and the error message. Followed by the exception type and the trace, which is a list of file names and paths, line numbers, function names, and the source code in

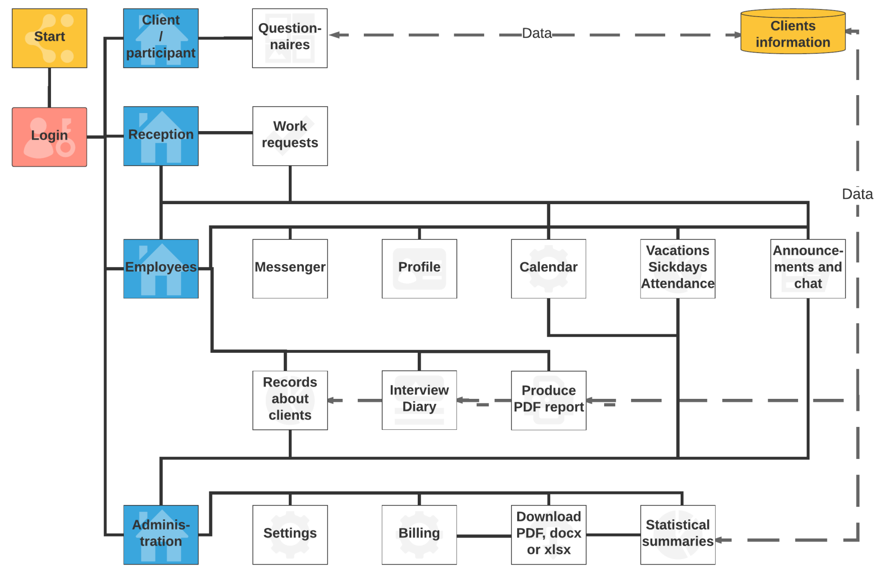


(a) From March 2016

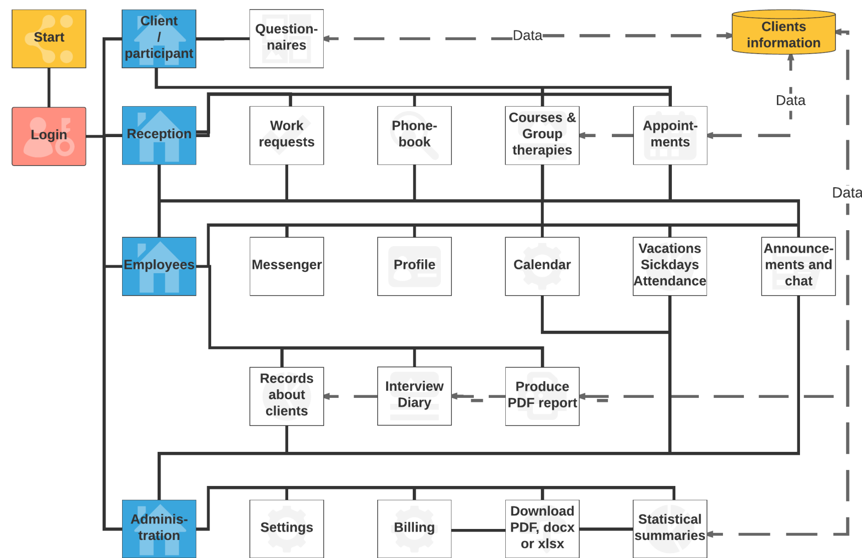


(b) From April 2016

Figure 8.1: JM's feature map (part 1) as it developed during the software's first 8 months in use. It shows how rapidly features were added after employees of JR started using the software in March 2016. This rapid development caused buggy and less well-tested code to be released, hence inspiring the integration of GI.



(a) From July 2016



(b) From September 2016. Self-improving capabilities are released.

Figure 8.2: JM's feature map (part 2) as it developed during the software's first 8 months in use. It shows how rapidly features were added after employees of JR started using the software in March 2016. This rapid development caused buggy and less well-tested code to be released, hence inspiring the integration of GI.

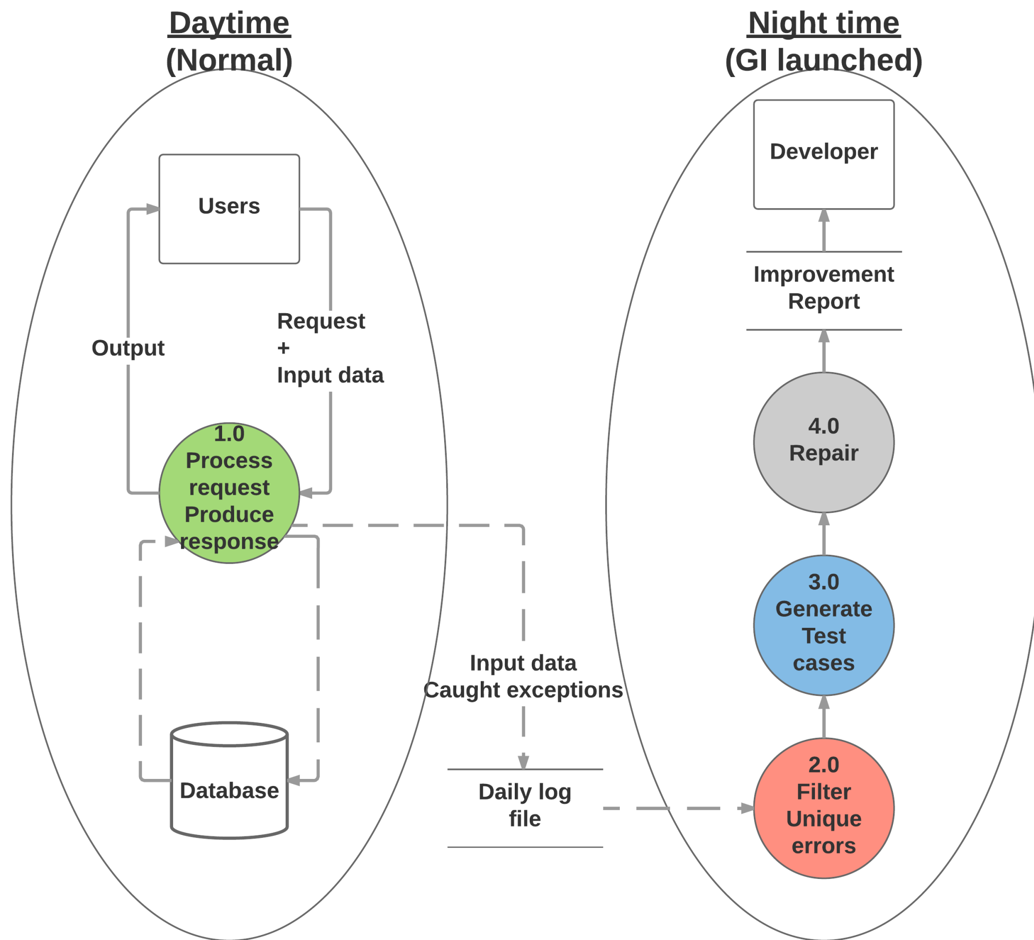


Figure 8.3: JM functionality divided into daytime processes and night time processes.

that line. The log also records where the call originated from (the referrer) and the arguments passed with the request as a Python dictionary object.

Listing 8.1: Typical log for a single error that JM records

```
2016-10-15 10:15:19.642972:
invalid literal for float(): 450 kr
<type 'exceptions.ValueError'>
Trace: [
('/var/www/JanusManager.local/public_html/JanusManager.py', 109, 'decorated_function'
    , 'return f(*args,**kwargs)'),
('/var/www/JanusManager.local/public_html/JanusManager.py', 7030, '
    participant_fjarhagskonnun', 'fjarh = Fjarhagskonnun.create(**answers)')]
Referrer: http://XX.XXX.168.10/participant_fjarhagskonnun/
Form data: {'participant':1242,'date':'2017-03-21','field_1':'450 kr.'}
```

8.2.3 *Structure of Janus Manager*

JR provides a personal vocational rehabilitation plan [181, 182] and as such users of JM regularly encounter unique use cases. JM is in active development while being in use as seen in Figures 8.1 and 8.2. Features are continually added, based on user experience, feedback and convenience as the usage evolves with changing group of clients and requirements. The average growth of the system, in the first months following its initial release, was approximately 20 lines of code (LOC) per week. Six months later, in the weeks preceding the integration of the GI, the growth had decreased to 15 LOC per week and has been relatively stable since.

Currently JM is over 25K lines of Python 2.7 (300 classes and 600 functions). Functions are on average 26 LOC and classes 36, ranging from 2 to 251 and 918 respectively. Each function and class has their own test suite, with a few exceptions[1]. The test suites are also of various sizes, depending on the functionality of the entity under testing.

JM runs as a web service on an Apache server running on a 64 bit Ubuntu with 48GiB RAM and two 6 core Intel processors. The GUI is an HTML web page that JM builds from pre-defined templates.

JM's structure made the integration of its nighttime activity as simple as wrapping the whole system in a single *try* and *except* statement that catches all exceptions that are not expected (see Procedure 2). The modularity gave the GI easy access to test and improve each component in isolation.

Procedure 2 A pseudo code of the wrapper function that logs unexpected failures during daytime use of JM.

```
1: while Application running do
2:   if Exception then
3:     logfile ← trace+input
4:     reset session
5:   end if
6: end while
```

8.3 Janus Manager Nightly Activity

After the last user logs off in the evening the nightly routine initiates (Figure 8.3, right side). The process runs until all reported bugs are fixed or until the next morning. During the night JM analyses the logs, generates new test data and uses GI to fix bugs that have been encountered during the day. The identified bugs are not necessarily faults in the program

1 Some base classes are never used directly so they do not have any tests

itself, rather a result of the developer's inability to account for all possible use cases of the system. On all occasions when the nightly activity was invoked and had bugs to fix it was because JR's employees had used JM in a way that had not been foreseen.

8.3.1 *Log analysis*

Going through the daily logs involves filtering the exceptions to obtain a set of unique errors in terms of exception type and location in the source code. The input is defined as the argument list at every function call on the trace route from the users' request to the location that caused the exception. The type of the exception can be any subclass of *Exception* in Python, either built in or locally defined.

The exceptions are sorted in decreasing order of importance, giving higher significance to errors that occurred more often, arbitrarily choosing between draws. This measure of importance assumes that these are use case scenarios that happen often and are experienced by multiple users and not a single user who repeatedly submits the same request.

8.3.2 *Generating test data*

The test data is generated by a simple random search of the neighbourhood of the users' input data retrieved from the log entry. The input is represented by a Python *dictionary* object, where elements are (key, value) pairs and the values can be of any type or class. However, most types are strings, dates, times, integers or floating point numbers. The objective of the search is to find as many versions of the input data as possible that trigger the same exception. Procedure 3 details the search for new test data.

Starting with the original input θ we make 100 instances of $\theta^{mutated}$ where a single value has been randomly changed. For each instance the value to be mutated is randomly selected while all other values are kept fixed. Every $\theta^{mutated}$ that causes the same exception as the original is kept in Θ, others are discarded. Different exceptions are not considered since the setup looks for the specific exception from the log rather than any general exception. This is then repeated by randomly sampling from the latest batch of $\theta^{mutated}$, Θ^{latest} (see line 10) until either no new instances are kept or the maximum of 1000 instances have been evaluated (line 5).

The mutation mechanism in line 11 first chooses key, value pairs in θ^{r} at random, only considering pairs where values are of type string, date, time, integer or float. Then depending on the type, the possible mutations are the following:

string mutations randomly add strings from a predefined dictionary with white space and special characters, keeping the original as a sub-string

Procedure 3 Test data search

1: $\Theta \leftarrow [\theta]$ {Start with the original input}
2: $n \leftarrow 0$
3: $\Theta^{new} \leftarrow [\theta]$
4: $\Theta^{latest} \leftarrow []$
5: **while** $(n < 1000)$ AND $(|\Theta^{new}|! = 0)$ **do**
6: extend Θ with Θ^{latest}
7: $\Theta^{latest} \leftarrow \Theta^{new}$
8: $\Theta^{new} \leftarrow [\ ]$
9: **for** $i = 1$ until $i == 100$ **do**
10: $\theta^{r} \leftarrow$ random choice Θ^{latest}
11: $\theta^{mutated} \leftarrow$ mutate θ^{r}
12: **if** $\theta^{mutated} \rightarrow$ causes exception **then**
13: append $\theta^{mutated}$ to Θ^{new}
14: **end if**
15: $n+ = 1$
16: **end for**
17: **end while**

date mutations can change the format (e.g. 2017-01-27 becomes 27-01-17), the separator, or randomly pick a date within a year from the original

time mutations can change the format (e.g. 7:00 PM becomes 19:00), the separator or randomly pick a time within 24 hours from the original

integer mutations add or subtract 1, 2 or 3 from the original. Maximum of ± 3 variation was arbitrarily chosen as a starting point for integer mutations because we assume integer inputs will not deviate much more from what is being observed from the user.

float mutations change the original with a random sample from the standard normal distribution $N(0, 1)$

All of the instances in Θ along with the original θ are then the inputs of the new unit tests. The assertion for each instance will check that the response is of the specific exception type and the tests will fail if the input triggers that exception. The new unit tests are then added to the existing test suite, automatically expanding the library of test cases.

There are mainly two problems with this approach; *a)* it does not check whether new test cases are complementary or not, i.e. if the two or more test cases are validating the same part of the code, and *b)* it assumes that if the exception is not raised then the output, if any is expected, is correct. The first problem is trivial when computing power is not an issue or if testing is not impeding development. The second problem is more serious because we cannot

Table 8.1: Sets of single operators available to the GI. One member of a given set can be changed to another member of the same set.

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

guarantee that the assumption holds and it might give false confidence to developers and wrong fitness evaluation to the GI. However the implementation of the whole system should at least catch any mistake the GI could introduce with these tests by passing the responsibility for sanity checks to the developers.

8.3.3 *Genetic Improvement*

The GI part of the overnight process relies on the new test cases in conjunction with a previously available test suite. The assumption is that, given the test suites, the program is functioning correctly if it passes all test cases and so is awarded highest fitness. Otherwise fitness is proportional to the number of test cases the program passes of the whole suite.

The target files and functions of JM is dynamically determined for each time by the locations specified in the log files. Only functions on the execution path of each bug are subject to modifications by the GI. Both *micro* and *macro* edits are used (see Chapter 5) with the operator sets in Table 8.1 and the line types in Tables 8.2-8.3 respectively. A single mutation of an edit list can be made with any of the three edit list mutations (*Grow*, *Prune*, and *Single edit change*) detailed in Section 7.4.1.

The evolution is population based with 50 edit lists (solutions) in each generation. Each generation is evaluated in parallel to minimise GI's execution time and to utilise the full power of the server. "Fitness Proportional Selection" is used to select parents for the next generation, i.e. each lists' weight determines how likely it is to be selected. The weight is determined by the edit list's proportional fitness with respect to the generation's total fitness. Only half of the population gets selected and they undergo mutation to start the next generation. Crossover is not used in the current implementation as well as elitism. The other half of the subsequent generation are randomly generated new edit lists. This selection method in this context should

Table 8.2: List of defined line types that cannot be altered and therefore not accessible to the GI.

<table>
<tr><th>Line type name</th><th>Indent following line/s</th><th>Regex pattern</th></tr>
<tr><td>Empty</td><td>False</td><td>^$</td></tr>
<tr><td>Import</td><td>False</td><td>^(import|from).*$</td></tr>
<tr><td>Multiple line comment</td><td>False</td><td>^\s*""".*$</td></tr>
<tr><td>class definition</td><td>True</td><td>^class .*:.*$</td></tr>
<tr><td>Function definition</td><td>True</td><td>^\s*def .*:.*$</td></tr>
<tr><td>try statement</td><td>True</td><td>^\s*try\s*:.*$</td></tr>
<tr><td>finally statement</td><td>True</td><td>^\s*finally.*:.*$</td></tr>
<tr><td>assert statement</td><td>True</td><td>^\s*assert .*:.*$</td></tr>
<tr><td>with statement</td><td>True</td><td>^\s*with .*:.*$</td></tr>
</table>

Table 8.3: List of defined line types that can be altered and targeted by the GI

Line type name	Indent following line/s	Regex pattern
Generic	False	`^.*$`
Return	False	`^\s*return\s*.*$`
Comment	False	`^\s*#$`
If statement	True	`^\s*if .*:.*$`
else statement	True	`^else\s*:.*$`
elif statement	True	`^\s*elif .*:.*$`
for loop	True	`^\s*for .*:.*$`
while loop	True	`^\s*while .*:.*$`
except statement	True	`^\s*except .*:.*$`

deter homogeneity in the population and early convergence to a local optimum. There is also the possibility that it might prevent the GI in finding solutions that require more than a few edits where the path to the correct program leads through space of poorer fitness. In most search problems this will probably promote homogeneity if the newly generated edit lists all have lower fitness than those that were generated with mutation. However, the assumption here is that a fix to a bug is only a few edits away from the original program.

The GI only stops if it has found a program variant that passes all tests or just before the users are expected to arrive to work. It then produces an HTML report detailing the night's process for the developers. The report lists all exceptions encountered, new test cases and a list

of possible fixes, recommending the fittest. If more than a single fix is found, then the report recommends the shortest in terms of number of edits. However it is always the developer's choice to implement the changes as they are suggested, build on them or discard them.

8.4 PREDICTIONS FOR DYNAMIC TREATMENT PLANNING

JR has a large database of earlier patients that has gradually been building up. In less than a year nearly 400 new instances from 73 patients have been added to the database which already contained over 4300 instances at the start. Each event in a patient's prediction history counts as one instance. Each instance currently has 180 features of 4 data types as listed in Table 8.4. The data processing is generic enough to allow JR to add features whenever they decide to collect new information about patients. The new features are then added to subsequent models.

Since June 2016 JR has used the predictor and successfully confirmed it as a viable tool. It differs from the traditionally off-line predictors by updating its rules on-line whenever new data is recorded. JR has developed a predictive model which the rehabilitation specialists use to inform decisions at every stage of the rehabilitation process. The predictor identifies possible risk factors, making treatment more efficient because the specialist can intervene quickly and knows where to act.

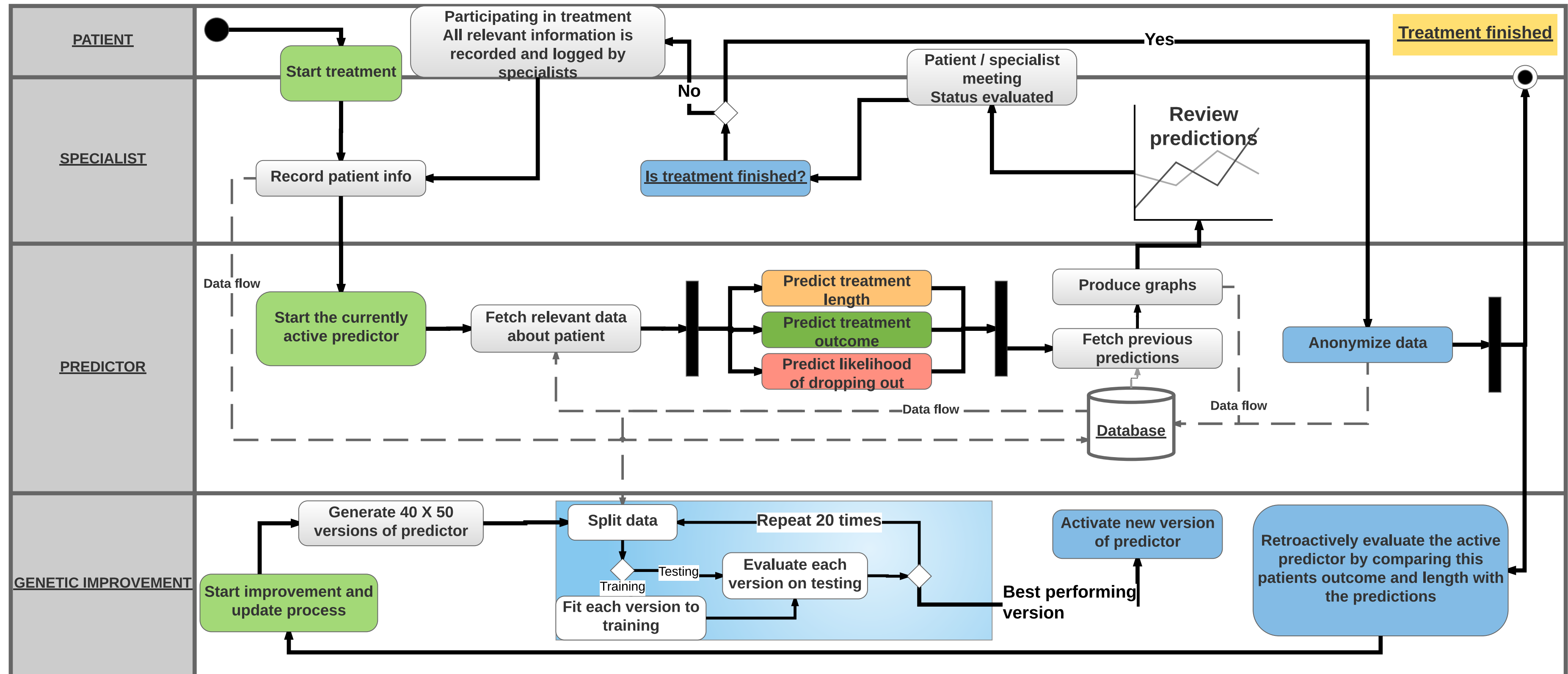


Figure 8.4: A flow chart showing the prediction and update process while a single patient receives treatment. The patient attends their treatment schedule and provides data. The specialist records the information, reviews predictions, and plans the treatment jointly with the patient. The predictor processes the data, makes predictions and visualises them. Lastly, the Genetic Improvement updates the predictor when the patient finishes.

Figure 8.4 shows a flow chart of a consultation between a specialist and patient while the predictor operates and is updated in the background. When the patient enters the treatment, the specialist records the information to the database which initiates the predictor. Three predictions are made and stored, based on the entered data:

- Likelihood of successful rehabilitation
- Drop out probability
- Treatment length, in months

The first two are binary classification problems where the outcome can be interpreted as true or false, while the last is a prediction of a continuous variable and therefore made with regression. The specialist can then review the patient's status with this new perspective and plan the next steps. Additionally, the predictor lists the ten most influential features for each prediction to help identify risk factors that affect the outcome and length.

This cycle is repeated, every time new information is recorded and as long as the treatment lasts. When patients finish treatment, all the collected data regarding them is anonymized and added to the database. The current version of the predictor is then evaluated by comparing all its previously stored predictions about them with the actual outcome.

The bottom layer of Figure 8.4 is the GI procedure which updates the predictor with the new data. The targeted part of JM is a Python script that pre-processes the data, and selects a prediction algorithm from scikit learn [149] and tunes its parameters. The objective of the improvement process is to minimise the mean squared error of regression models and maximise accuracy of classification models. The GI uses Monte Carlo cross-validation [47], with 20 repetitions, to evaluate fitness. The dataset is randomly divided into training and testing sets of equal sizes. The fitness of each edit list is then the average performance over 20 splits. Other configuration parameters are identical to the bug fixing setup for JM.

When the GI has finished, the best performing variation, out of 2000 tested, replaces the current predictor instance which fits three models, one for each of the three predicted variables. They are then used for all predictions until the next person leaves treatment and the updating process starts again.

8.4.1 *Evaluation of The Predictor*

The predictor was added to JM in June 2016. For JR, the most important evaluation of the predictor is how its specialists experience it in practice.

However, we also verify the predictor objectively by evaluating its performance on those patients that have completed their treatment after it was implemented. The procedure involves iterating over 73 versions of the predictor from June 2016 until March 2017 and compare

Table 8.4: Data types of features in the set, number and examples.

Data type	Number	Examples
Float	120	Age, Length of unemployment, Quality of Life measurement Current treatment duration
Integer	18	Number of children, Number of medical diagnoses
Boolean	37	Bullied, Dyslexic, Been JR patient before
Categorical	5	Education, Income, Gender, Housing, Relationship status

each version's predictions with actual outcome. For the classification problems, we measure accuracy ($\frac{c}{n}$) and precision ($\frac{p}{n}$), where n is the number of predictions, c is the number of correctly labelled predictions, and p is the number of correctly labelled predictions of the positive class. Accuracy is the proportion of correct labels while precision is the proportion of correct positive labels. For the regression problems, predictions (denoted with $\hat{Y}$), and true labels (Y), we measure *mean squared error* (MSE) (8.1)

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{8.1}$$

and *median absolute deviation* (MAD) (8.2)

$$\mathrm{MAD} = \mathrm{med}(|Y_i - \mathrm{med}(Y)|) \qquad \forall i \in [1, 2, ..., n] \tag{8.2}$$

The MAD is a robust variation measurement while the MSE is a well know measure of spread. Additionally, to evaluate how the GI is affecting performance of the predictor we compare these values before and after each improvement process.

8.5 Performance review

8.5.1 *Bug fixing*

Development on JM started as a small in-house data management project by JR in March 2016. However, as JR's employees started using the software they identified multiple ways to enhance JM to improve productivity and efficiency in the rehabilitation process. JM's development has

since been user-driven and quite rapid, with a new feature being added weekly in spring and summer 2016. As JR is not a software developer, its core development team is minimal. This, combined with JM's rate of expansion caused poorly tested code to be repeatedly released and subsequently occupying the development team with bug fixing instead of further enhancing JM in a meaningful way. Therefore JM has been running its self-healing processes every night with exceptional results since September 2016.

In the four months succeeding the launch of GI within JM, 22 bugs have been identified and fixed. Table 8.5 lists the bugs that have been encountered, categorised by the exception type and order in descending order of how often each bug was invoked by users (Column 3). The table also lists how many new test cases, without duplicates, were added to the test suite (Column 4). In total 29 unique test cases have been added to JM's test suites but initial number of automatically generated test cases was 408. The developers could easily and swiftly discard duplicate test cases by hand.

Table 8.5: A list of the bugs that were automatically detected in JM and fixed by the GI. They are categorised by the type of exception that they caused and given an identification number in the second column.

Exception type	Id	Invocations	New test cases	Input type that caused exception	Suggested fix size (Accepted fix size)	Mean edit list size	Number of edit lists considered
IndexError	E1	6	1	Date tuple	2 (2)	2.53	412
	E2	3	1	Integer	3 (3)	2.48	356
	E3	3	2	Integer	2 (2)	2.38	367
	E4	1	1	Integer	4 (4)	2.62	437
TypeError	E5	36	1	Integer	3 (3)	2.23	426
	E6	6	3	String	4 (3)	2.41	465
	E7	1	1	Integer	2 (2)	2.67	457
	E8	2	1	(String, Integer)	1 (1)	2.50	442
	E9	2	1	String	5 (3)	2.50	413
	E10	1	1	String	2 (2)	2.47	412
UnicodeDecode Error	E11	4	1	String	2 (2)	2.48	424
	E12	3	2	String	3 (3)	2.48	404
	E13	2	2	String	2 (2)	2.48	465
ValueError	E14	4	2	Date and time	1 (1)	2.54	435
	E15	4	1	Date and time	1 (1)	2.57	388
	E16	3	1	String	3 (3)	2.44	428
	E17	3	1	Integer	5 (4)	2.49	353
	E18	2	1	Date and time	3 (3)	2.49	467
	E19	2	2	Date and time	3 (3)	2.40	405
	E20	1	1	Time	4 (3)	2.39	477
	E21	1	1	String	2 (2)	2.47	371
	E22	1	1	String	1 (1)	2.56	478

The fifth column of Table 8.5 describes the input types that caused the exceptions. In most cases the types listed there are a single variable from an array of inputs but only the variables that were directly involved in throwing the exception are mentioned. Column six lists how many edits each suggested and accepted fix contained. In majority of cases the suggested fix was accepted as it was but on three occasions a single neutral edit was removed before accepting the fix (*E6*, *E17*, *E20*) and two edits from *E9*. All removed neutral edits were either duplicating a line or a variable, without it having effect on output. In four instances, either a single edit was slightly altered or a small manual edit to the source code was applied post-hoc (*E1*, *E15*, *E16*, *E22*). The last two columns contain the average size of all edit lists that were evaluated for each bug and how many were evaluated, respectively.

If we look at the seventh column, we see that the average edit list size is nearly the same for all fixes ([2.23, 2.67]), which is to be expected since the same search parameters were used in every case. On closer inspection, one way ANOVA reveals that the mean size of all evolutionary runs is most likely equal ($p > 0.9$). Furthermore, we see that the evolution never exceeds 10 generations (500 evaluated edit lists) and consequently the maximum limit of edit list size was 10 edits. Given that the average size of each function being fixed is 26 LOC the search space is relatively small so if a fix exists we expect to find it rather quickly.

A typical fix replaced a single line with a similar line from elsewhere in JM, like *E15* replaced:[2]

```
dum.occurance = \
   datetime.datetime.combine(dum.expected_occurance,\
   datetime.datetime.strptime(form['occurance'],\
   '%H:%M').time())
```

with:

```
dum.occurance = \
   datetime.datetime.combine(dum.expected_occurance,
   datetime.datetime.strptime(form['occurance'],\
   '%H:%M:%S').time())
```

The only difference is that the latter expects seconds to be included in the time format. The human programmer recognises it as a single edit of adding `:%S` but the GI replaced the whole line.

Another example is *E20*, in a function that checks for reoccurringly available meeting spaces on given weekdays. The bug was that the user sometimes omitted the time of day to be checked. The accepted fix wrapped the line obtaining the time argument in a *try* clause so this:

2 The character \ denotes line continuation and is only used here for aesthetic purposes

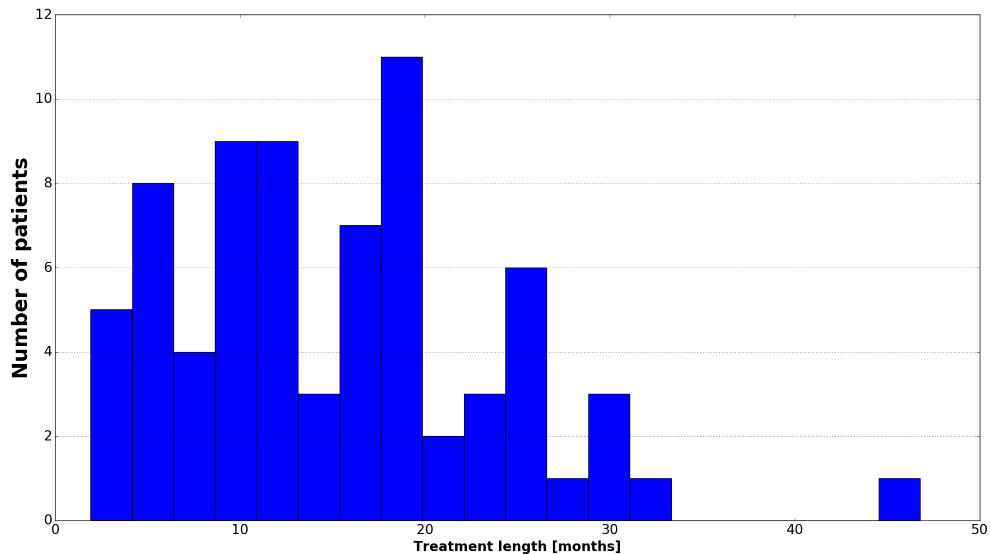


Figure 8.5: The distribution of treatment length for the 73 patients that finished treatment during the ten month period.

```
the_time = datetime.datetime.strptime(\
request.args.get('the_time','08:00'),\
'%H:%M').time()
```

became this:

```
try:
 the_time = datetime.datetime.strptime(\
 request.args.get('the_time','08:00'),\
 '%H:%M').time()
except ValueError:
 the_time = datetime.time(8,0)
```

Three edits of "copy line x above line y" were needed to accomplish this fix, the edit that was removed, duplicated the last line.

8.5.2 *Prediction Improvements*

8.5.2.1 *The Practical use of the predictor*

JR's specialists have expressed that being able to identify important factors of the patient's current status is particularly helpful, along with the graph of previous predictions. It has been used as a visual aid by demonstrating an increased likelihood of a positive outcome, and also to encourage the patient when they cannot perceive progression themselves. Some specialists have also used it to expedite appointments with their patient when the predictor

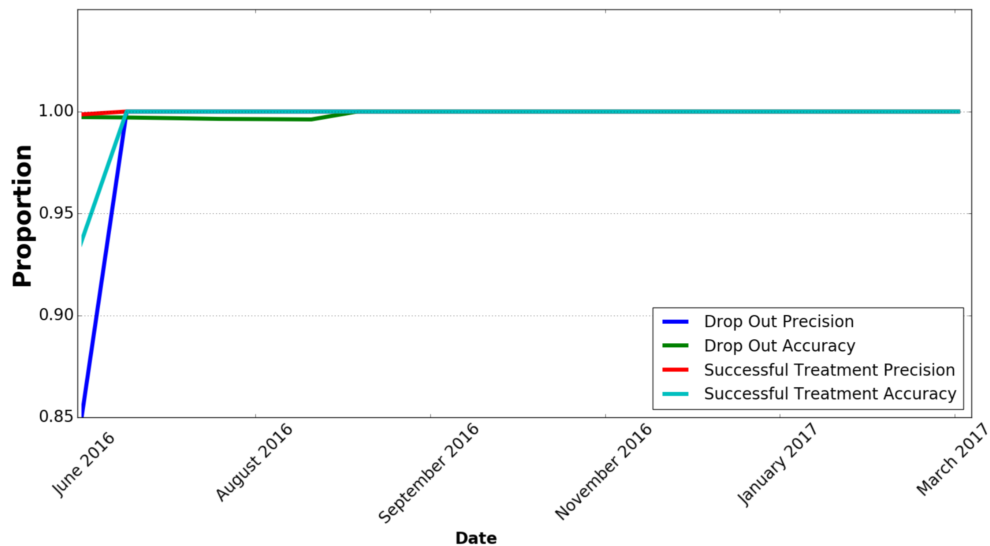


Figure 8.6: Precision and accuracy of predictions for dropping out and successful treatment over the trial period.

shows increased drop out probability. However, to be able to verify if the use of the predictor has decreased the number of drop outs we need to collect data over a longer period. There are a number of seasonal variables that might have confounding effects, such as seasonal affective disorder.

8.5.2.2 *Results for classification problems*

The predictor did well on two classification tasks; if treatment will be successful, and if a patient will drop out. The models were being used in real-time while patients were being treated and up-to ten months before the actual outcome was know. Figure 8.6 shows the precision and accuracy of predictions that were made for patients while they were in treatment and then evaluated after they finished and the outcome was known.. The predictions for dropping out had over 99% accuracy from the start, and after August 2016, the accuracy was 100%. Its precision started at 85% but increased to 100% within 2 weeks (see). Similarly, predicting a successful treatment was also at 100% accuracy and precision in week two after the release of the first version of the predictor.

8.5.2.3 *Results for regression problems*

The regression models were able to predict treatment length within three months from the actual duration. A three month difference is an acceptable estimation because in practice this is a lead-in time. Two predictor versions out of total 73 had an error of up to five months. Those versions were updated within a week of being activated. Figures 8.7 and 8.8 show the performance, as measured post hoc, of number of different models for every two weeks over the period, June 2016–March 2017. The boxes in the figures are the first and third quartiles,

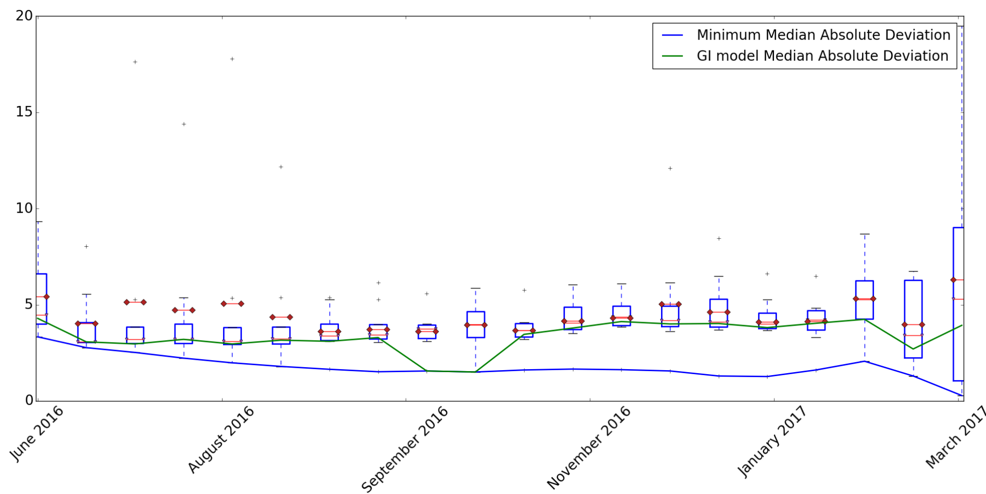


Figure 8.7: Distribution of post hoc evaluation of MAD for every two weeks of updated models for treatment length. Both mean (diamond) and median (triangle) are marked with each box.

the blue line are the best performing models, and the green line is the performance of the models that were in use each time. Note that the performance of the model in use was always better than the mean and median performing model. This indicates that the fitness evaluation on the training data was sufficient to guide the selection of a better than average performing model. Furthermore, the variation in the performance of the models gets increasingly larger when adding more data to the training set. The GI is producing a more diverse populations of regression models. The most likely explanation for this phenomenon is that the data is undergoing some changes and there are at least two possible reasons

- The specialists are gathering data differently, i.e. asking questions in a different manner or even collecting data more frequently.
- The variation of the patient base is changing. New people with different challenges enter rehabilitation every week.
- The patient behaviour is changing. The evaluation covered a 10 month period and it is quite possible that seasonal variables are affecting behaviour.

It is also possibly linked to the variation of treatment length as seen in Figure 8.5.

8.5.2.4 *Effects of Genetic Improvement*

The GI was used to improve the selection and tuning of both classification and regression models. The GI could not improve beyond maximum regarding the classification accuracy as seen in Figure 8.6 and the variation of the accuracy between different versions of the predictor was less than 1×10^{-5}.

However, the regression models were quite different as mentioned in Section 8.5.2.3. In Figure 8.7 and 8.8 we can see performance spread of the top performing version of each

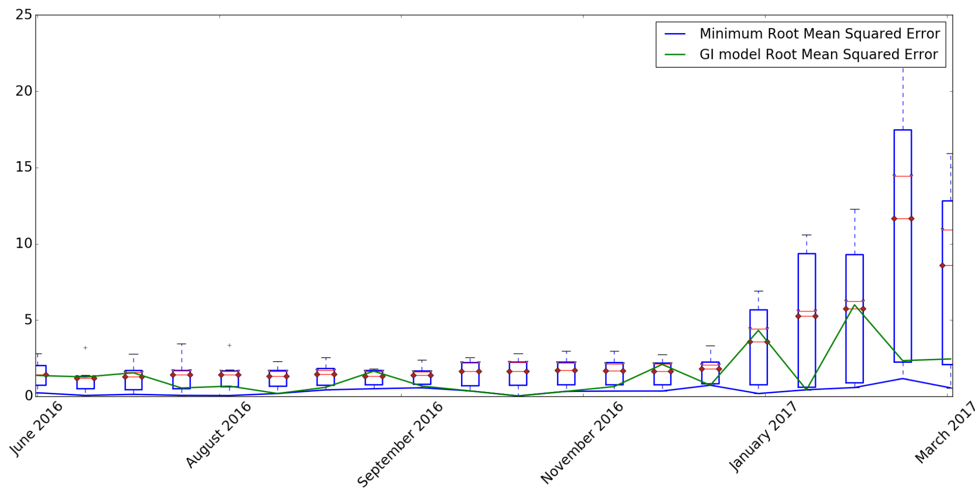


Figure 8.8: Distribution of post hoc evaluation of MSE for every two weeks of updated models for treatment length. Here, converted to Root Mean Squared Error for scaling on y-axis. Both mean (diamond) and median (triangle) are marked with each box.

generation from the GI. The green lines indicate the performance of the best version on the test set after correct results were known. The GI managed to keep predictions mostly within three months from the actual length, causing the overall performance of the predictor to be more than adequate.

8.6 SUMMARY

In this chapter we have described two applications of GI within the same live system, JM. The GI serves two purposes:

- Bug fixing assistance
- Prediction improvements

The same GI framework is used for both tasks, only difference is the targeted files and objectives. The former task seeks to maximise the number of test cases passed, while the latter tries to minimise prediction error. Both implementations show that GI can, with a tiny effort, be used in various ways to make software dynamically adaptive.

JM, as introduced in this chapter is fully implemented and live, although it has been so for a few months, it still needs to be tested further. Over a 4 month period, from September 2016 to January 2017, the bug fixing implementation found and fixed 22 bugs. It is still in active use and has contributed to a decrease in JR's maintenance cost of JM.

The performance of the predictor was outstanding, with almost 100% accuracy and precision in classification predictions over a 10 month period. Additionally, it has consistently predicted treatment length within satisfactory margin for a vocational rehabilitation.

The predictor was developed to meet the demand for an objective view of each patient's status while receiving treatment. To the authors' best knowledge, JR is the first facility to use a predictor, designed for that purpose in practice. JR's specialists have integrated the use of the predictor into their daily routine to get a clear view of the progress of each patient, at every stage of their treatment. With the increasing number of patients in treatment, the predictor helps by identifying possible risk factors. It assists the specialist to know where and when to intervene, possibly shortening the treatment time. They have used the predictor in various ways, discussing progress or the lack there of with the patient, to encourage or recognise what might be interfering with the treatment. Some specialist have used the predictor at the start of the rehabilitation to focus efforts on specific areas in the patient's circumstances.

The predictor's first 10 months in use have been evaluated by comparing the initial predictions for patients that were receiving treatment with actual outcomes and treatment lengths after they finished. The results for classifying a patient's treatment as drop out, unsuccessful or successful were more than satisfactory according to experts in rehabilitation, with near 100% accuracy.

The graph in Figure 8.5 shows us that the treatment time can vary from 2 months to 47 and that is a possible culprit for the large variation in treatment time predictions for JR's patients. However, the GI was consistently able to find versions of the predictor that had decent performance. Overall the predictions for treatment length where within 2 to 3 months from what actually occurred. The combination of GI and prediction models has proven to be beneficial for the vocational rehabilitation treatment.

The predictor is still in use and continually evolving with the expanding dataset, providing dynamic predictions for the specialists. With current progress in software and hardware development it is well worth exploring automatic adjustments of predictive models. Automatic algorithm design [209] and GI are two of many methodologies to make portable CI tools for healthcare, rather than depending on predefined models that might work well for the general population but not in specific treatments or facilities. In other words, GI can adapt the predictor to the specific data and patients at a given facility. Therefore the predictor was able to perform so well for JR, it was specialised to their database, which contains a narrow population. The predictor is a valuable asset for specialists, patients, and the facility as a whole. The predictor needs to be adapt to each treatment facility and database, and this can be achieved with GI. This predictor can reduce cost and identify possible risk factors, helping specialists to intervene earlier.

Part IV

CONCLUSIONS

9 CONCLUSIONS

In this thesis we set out to investigate if GI can be used as a dynamic component in a software system. This, I did by exploring three complementing research questions detailed in Chapter 2. Part iii of the thesis presented three strands of research, each addressing a single research question. The conclusions for Chapters 6–8 with the framework from Chapter 5 serve as supporting evidence to the following claims:

- GI can fix bugs in small programs. Although the fitness landscape for GI when measured with number of test cases passed is generally flat, the framework can find a fix. The soft restart mechanism seems to be escaping plateaus and sub-optimal valleys in our experiments.
- GI is portable between source languages and objectives to optimise. The framework modifies source code as a string and is only aware of syntax that the practitioner specifically defines to constrain the search. The syntax constrictions are simple lists of interchangeable tokens (e.g. <, ==) and categorisations of different types of lines (e.g. *if* declarations `if X<=1:`).
- GI is well suited to adapt software systems to be dynamically adaptive. As long as the system is non-critical, not like e.g. air traffic control so the users can wait overnight for a fix. That is assuming the bug only affects parts of the software's usage.

In this chapter I summarise the thesis' contributions in Sections 9.1–9.3, by connecting each claim above with the work in the corresponding chapter. Which will, when collected into Section 9.4, form a convincing case for the thesis hypothesis and conclude this research.

9.1 Q1: SMALL PROGRAMS' LANDSCAPES

Research Question 1 *Can GI fix bugs in small programs, written in a dynamically typed language and how does the GI interact with the search space?*

In Chapter 6 I investigated how three small (~ 100 LOC) Python programs' landscapes look from the GI's perspective. Although the landscapes were mostly flat, combining large plateaus of correct programs and flat valleys of programs that fail every test case, the transition between the two can be achieved with as little as a single edit. The dynamically typed nature of Python

did not contribute either way to support or disprove the answer to the research question. However, the smaller programs allow the GI to search a large portion of the feasible landscape with relative ease and if an improved version exists in the space, then the GI will likely find it given long enough time.

9.2 Q2: Execution Time Improvements

Research Question 2 *Is it possible to implement GI in such way to be portable between different source languages and different objectives and how does changing these impact on the search?*

Chapter 7 detailed an experiment on improving the execution time of a bioinformatics program (ProbAbel). It is written in C/C++ and considerably larger (~ 5K LOC) than the ~ 100 LOC programs written Python (Section 6). Therefore we can definitively answer the first part of the question with a yes. The execution time experiments were conducted with the same GI implementation as was used for bug fixing explorations. It targeted the majority of ProbAbel's source code and was able to find a version that executed faster. However, the improvement was not substantial (0.5% decrease) but demonstrated that the implementation was portable. Regarding the second part of the question, the impact on the search is minimal. The fitness landscape proved to be largely flat and single edit changes to the program seemed to have little to no effect on execution time.

9.3 Q3: Dynamic Adaptive Software

Research Question 3 *Can GI be integrated into an already live system to identify bugs and suggest fixes, thus assisting developers with maintenance?*

JM was introduced in Chapter 8, the bespoke software system for vocational rehabilitation management. The system has been live and in full use since March 2016 by a rehabilitation centre in Reykjavik, Iceland[1]. Due to a combination of rapid, user-driven development and a workload that exceeded the capacity of the small team of programmers to both implement and fully test, the early versions of the system were buggy and not well tested. This provided the opportunity to test if the GI framework used in the work for previous Chapters was suitable to be integrated in a dynamic setting. It was called on to fix bugs that were detected after the software had been released. As demonstrated the integration was successful and JM has, since September 2016, had an autonomous GI run every night. The favourable results have already saved on maintenance cost for JR by identifying and suggesting fixes for 22 bugs from September 2016 to January 2017. The developers have accepted 20 fixes without amendments and 2 with minor adjustments. This shows that the integrated GI framework

1 https://janus.is/

can be a useful tool for developers. Even when the suggested fixes are not accepted verbatim, they guide the programmers towards an implementation (changes and location) that will fix the bug. An additional experiment was conducted within the same system but with the objective of improving predictions for the outcome of rehabilitation processes. The GI framework was able to help the prediction models to repeatedly make predictions with 100% accuracy and precision within 3 weeks of being deployed. The successful result is evidence of the GI framework's versatility.

9.4 CENTRAL HYPOTHESIS

Thesis Hypothesis *GI can be used autonomously in software systems to decrease maintenance cost and assist developers to improve various properties of the software.*

The research questions are all yes or no questions and as demonstrated this thesis has presented evidence to support a positive answer for each. Combined, these results encourage confidence in the hypothesis. I argue that Chapter 8 shows that GI can be used autonomously within running software services to decrease maintenance cost. Chapter 7 gives evidence for the ability to improve other properties and for the portability of the GI. Additionally Chapter 6 provided insight that was useful for the final integration of GI into JM.

9.5 FUTURE WORK

JM, as introduced in this thesis is fully implemented and live, although it has been so for a few months, it can still be tested further. Finding and fixing 22 bugs is impressive but we would like to do more. Our current task list includes but is not limited to:

- Integrate the GI in another software system. The GI is a standalone piece of software that is easily integrated in most software systems so a natural next step would be to identify services and systems where it could be of use.
- Improve the developer's interface with the GI. Current implementation reports only fixes that pass all tests but we might want to consider lesser variations and get a Pareto front of possible improvements.
- Improve the search ability. Truncated selection might inhibit larger edit lists to be evolved and possible multi-edit solutions are therefore lost. Parameter tuning is our initial step forward in this task.
- As the current implementation of JM's bug fixing mechanism only reports suggestions that pass all test cases there might be some performance gain in lesser variations that

could be considered. An idea is to use lexicase selection to promote diversity amongst test cases and build a Pareto front of possible improvements.

- Improve the test data generation mechanism. Ideally we want predict expected inputs to the system and possibly generate test data that imitates unseen future inputs. Also, we would like to improve the search process for new test data by implementing something other than random search. That includes changing the fitness function (currently binary), adding more objectives and improving the sampling methods.

- Investigate the fitness landscape of larger modifications to the source code. In this work we have explored the fitness landscape of a few *micro* edits at a time. It would be interesting to see what effects *macro* edits have on the search capabilities of the GI framework.

- Explore a larger portion of each variant's local neighbourhood. This research has exhaustively inspected the local neighbourhood of small Python programs (correctly functioning) and examined a large portion of the same neighbourhood for a larger C++ program. Additionally we looked at a number of variable sized sequences of edits, single edit at a time. Exploring the local neighbourhood of each of these single edits in the sequences could be an engaging idea for further understanding the search abilities of the GI framework.

- Regarding JM's bug fixing it would be interesting to analyse the edit distance between partially accepted fix suggestions and what was eventually implemented. This could provide an insight into how much the suggestions are helping even if they are not accepted fully.

However a continuous task will be to monitor the JM system while it is being developed further and gather data on the bugs that are caught and fixed.

9.6 SUMMARY OF CHAPTER 9

This chapter has summarised the results of each chapter in Part iii which led to the conclusion that the thesis hypothesis holds. Moreover, it has been demonstrated that the hypothesis remains valid in a practical setting where the GI continues to be used every night. The GI framework is effectively doing a night shift while rehabilitation staff are working the day shift. This situation is a realisation of one of the DAASE project's [74][2] objectives: a Dynamic Adaptive SBSE. The work in this thesis was funded by the EPSRC grant EP/J017515/1.

2 http://daase.cs.ucl.ac.uk/

BIBLIOGRAPHY

[1] Thomas Ackling, Brad Alexander, and Ian 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. ISBN 9781450305570. doi: 10.1145/2001576.2001768.

[2] Allen Troy Acree Jr. *On Mutation*. Phd, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[3] Wasif Afzal and Richard Torkar. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems with Applications*, 38(9):11984–11997, sep 2011. ISSN 09574174. doi: 10.1016/j.eswa.2011.03.041.

[4] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6): 957–976, 2009. ISSN 09505849. doi: 10.1016/j.infsof.2008.12.005.

[5] Andrea Arcuri. On the Automation of Fixing Software Bugs. In *ICSE Companion '08 Companion of the 30th international conference on Software engineering*, pages 1003–1006, Leipzig, Germany, 2008. ACM. ISBN 9781605580791. doi: 10.1145/1370175.1370223.

[6] Andrea Arcuri. *Automatic Software Generation and Improvement Through Search Based Techniques*. Phd, School of Computer Science The University of Birmingham, 2009. URL `http://etheses.bham.ac.uk/400/`.

[7] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4): 3494–3514, jun 2011. ISSN 15684946. doi: 10.1016/j.asoc.2011.01.023.

[8] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168. IEEE Computational Intelligence Society, IEEE, jun 2008. ISBN 978-1-4244-1822-0. doi: 10.1109/CEC.2008.4630793.

[9] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In Xiaodong Li, Michael Kirley, Mengjie Zhang, et al. editors, *7th International Conference, SEAL 2008*, Lecture Notes in Computer Science, pages 61–70, Melbourne, Australia, dec 2008. Springer Berlin Heidelberg. ISBN 0302-9743. doi: doi:10.1007/978-3-540-89694-4_7.

[10] Yurii S Aulchenko, Maksim V Struchalin, and Cornelia M van Duijn. ProbABEL package for genome-wide association analysis of imputed data. *BMC bioinformatics*, 11:134, 2010. ISSN 1471-2105. doi: 10.1186/1471-2105-11-134.

[11] Robert Balzer. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, nov 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.231877.

[12] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 306–317, New York, New York, USA, nov 2014. ACM Press. ISBN 9781450330565. doi: 10.1145/2635868.2635898.

[13] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, ISSTA 2015, pages 257–269, New York, New York, USA, 2015. ACM Press. ISBN 9781450336208. doi: 10.1145/2771783.2771796.

[14] Bill Bejeck. *Getting Started with Google Guava*. Packt Publishing, 2013. ISBN 1783280158, 9781783280155.

[15] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, volume 225, pages 73 – 87, Limerick, Ireland, 2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336534.

[16] Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider, and Eric Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012. ISSN 0028-0836. doi: 10.1038/nature10872.

[17] Michael Beyene and James H Andrews. Generating String Test Data for Code Coverage. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 270–279. IEEE, 2012. ISBN 9780769546704. doi: 10.1109/ICST.2012.107.

[18] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science*, 360(1-3):23–41, aug 2006. ISSN 03043975. doi: 10.1016/j.tcs.2006.01.012.

[19] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 109–120, New York, New York, USA, nov 2014. ACM Press. ISBN 9781450330565. doi: 10.1145/2635868.2635893.

[20] Mauro Birattari, Gianni Di Caro, and Marco Dorigo. Toward the Formal Foundation of Ant Programming. In Marco Dorigo, Gianni Di Caro, and Michael Sampels, editors, *Ant Algorithms: Third International Workshop, ANTS 2002 Brussels, Belgium, September 12–14, 2002 Proceedings*, pages 188–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45724-4. doi: 10.1007/3-540-45724-0_16.

[21] Alexander E I Brownlee, Nathan Burles, and Jerry Swan. Search-based energy optimization of some ubiquitous algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence*, Forthcoming:1–13, 2017.

[22] Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J Parkes. Hyperion2: A toolkit for {meta-, hyper-} heuristic research. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion - GECCO Comp '14*, pages 1133–1140, New York, New York, USA, 2014. ACM Press. ISBN 9781450328814. doi: 10.1145/2598394.2605687.

[23] Bobby R Bruce. Energy Optimisation via Genetic Improvement. In William B. Langdon, Justyna Petke, and David R. White, editors, *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, GECCO Companion '15, pages 819–820, New York, New York, USA, jul 2015. ACM Press. ISBN 9781450334884. doi: 10.1145/2739482.2768420.

[24] Bobby R Bruce, Justyna Petke, and Mark Harman. Reducing Energy Consumption Using Genetic Improvement. In Sara Silva, Anna I Esparcia-Alcazar, Manuel Lopez-Ibanez, et al. editors, *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, Madrid, Spain, jul 2015. ACM, ACM. ISBN 9781450334723. doi: 10.1145/2739480.2754752.

[25] Bobby R Bruce, Jonathan M Aitken, and Justyna Petke. Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering. SSBSE 2016*, volume 9962 of *Lecture Notes in Computer Science*, pages 238–243. Springer International Publishing, Cham, 2016. ISBN 978-3-319-47106-8. doi: 10.1007/978-3-319-47106-8_18.

[26] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. Research Note RN/17/01 Approximate Oracles and Synergy in Software Energy Search Spaces. Technical report, UCL Department of Computer Science, London, UK, 2017.

[27] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525 LNCS of *Lecture Notes in Computer Science*, pages 48–70. Springer Berlin Heidelberg, 2009. ISBN 3642021603. doi: 10.1007/978-3-642-02161-9_3.

[28] Edmund K Burke and Graham Kendall. *Search Methodologies*. Springer US, Boston, MA, 2nd edition, 2014. ISBN 978-1-4614-6939-1. doi: 10.1007/978-1-4614-6940-7.

[29] Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 449–468. Springer US, 2010. ISBN 978-1-4419-1665-5. doi: 10.1007/978-1-4419-1665-5_15.

[30] Nathan Burles, Edward Bowles, Alexander E I Brownlee, Zoltan A Kocsis, Jerry Swan, and Nadarajen Veerapen. Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering. SSBSE 2015*, volume 9275 of *Lecture Notes in Computer Science*, pages 255–261. Springer, Cham, 2015. doi: 10.1007/978-3-319-22183-0_20.

[31] Nathan Burles, Jerry Swan, Edward Bowles, Alexander E.I. Brownlee, Zoltan A. Kocsis, and Nadarajen Veerapen. Embedded Dynamic Improvement. In William B. Langdon, Justyna Petke, and David R. White, editors, *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. ISBN 9781450334884. doi: 10.1145/2739482.2768423.

[32] Rod M Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, jan 1977. ISSN 00045411. doi: 10.1145/321992.321996.

[33] Ugo Buy and Alessandro Orso. Automated Testing of Classes. *SIGSOFT Software Engineering Notes*, 25(5):39–48, 2000. doi: 10.1145/347636.348870.

[34] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic Recovery from Runtime Failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 782–791, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

[35] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for SelfAdaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 9783642021602. doi: 10.1007/978-3-642-02161-9_1.

[36] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin Rinard. Battery-aware transformations in mobile applications. In *Proceedings of the 31st IEEE/ACM International*

Conference on Automated Software Engineering - ASE 2016, pages 702–707, New York, New York, USA, 2016. ACM Press. ISBN 9781450338455. doi: 10.1145/2970276.2970324.

[37] Brendan Cody-Kenny and Stephen Barrett. The Emergence of Useful Bias in Self-focusing Genetic Programming for Software Optimisation. In Guenther Ruhe and Yuanyuan Zhang, editors, *Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 306–311, St. Petersburg, Russia, aug 2013. Springer. doi: 10.1007/978-3-642-39742-4_29.

[38] Brendan Cody-kenny, Edgar Galván-lópez, and Stephen Barrett. locoGP : Improving Performance by Genetic Programming Java Source Code. In William B. Langdon, Justyna Petke, and David R. White, editors, *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. ISBN 9781450334884. doi: 10.1145/2739482.2768419.

[39] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A Search for Improved Performance in Regular Expressions. pages 1–17, 2017. URL `http://arxiv.org/abs/1704.04119`.

[40] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating Fixes from Object Behavior Anomalies. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554. IEEE, nov 2009. ISBN 978-1-4244-5259-0. doi: 10.1109/ASE.2009.15.

[41] John Darlington and R.M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976. ISSN 0001-5903. doi: 10.1007/BF00263742.

[42] Rogério de Lemos, Holger Giese, Hausi A. Müller, et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 9783642358128. doi: 10.1007/978-3-642-35813-5_1.

[43] Vidroha Debroy and W Eric Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010. ISBN 978-1-4244-6435-7. doi: 10.1109/ICST.2010.66.

[44] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis - CSTVA*

2014, pages 30–39, New York, New York, USA, 2014. ACM Press. ISBN 9781450328470. doi: 10.1145/2593735.2593740.

[45] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, apr 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136.

[46] Edsger W Dijkstra. Structured programming. 1969. URL `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF`.

[47] Werner Dubitzky, Martin Granzow, and Daniel Berrar. *Fundamentals of data mining in genomics and proteomics*. Springer, 2007. ISBN 9780387475097. doi: 10.1007/978-0-387-47509-7.

[48] Philippe Duchon and Guy Louchard. Boltzmann Samplers For The Random Generation Of Combinatorial Structures. *Combinatorics Probability and Computing*, 13(4-5):577–625, 2004. doi: 10.1017/S0963548304006315.

[49] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S McKinley. Starc: Static Analysis for Efficient Repair of Complex Data. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, page 387, New York, New York, USA, 2007. ACM Press. ISBN 9781595937865. doi: 10.1145/1297027.1297056.

[50] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing Better Fitness Functions for Automated Program Repair. *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 965–972, 2010. doi: doi:10.1145/1830483.1830654.

[51] Deji Fatiregun, Mark Harman, and Robert M. Hierons. Evolving Transformation Sequences Using Genetic Algorithms. *4th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2004.*, pages 65–74, 2004. doi: 10.1109/SCAM.2004.11.

[52] Deji Fatiregun, Mark Harman, and R.M. Hierons. Search-Based Amorphous Slicing. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 3–12, Carnegie Mellon University, Pittsburgh, Pennsylvania, {USA}, nov 2005. IEEE. ISBN 0-7695-2474-5. doi: 10.1109/WCRE.2005.28.

[53] Robert Feldt. Generating diverse software versions with genetic programming: an experimental study. *IEE Proceedings - Software*, 145(6):228, 1998. ISSN 14625970. doi: 10.1049/ip-sen:19982444.

[54] Robert Feldt. Genetic Programming as an Explorative Tool in Early Software Development Phases. In Conor Ryan and Jim Buckley, editors, *1st International Workshop on Soft*

Computing Applied to Software Engineering, pages 11–20, University of Limerick, Ireland, 1999. Limerick University Press. ISBN 1-874653-52-6.

[55] Robert Feldt and Simon Poulding. Finding Test Data with Specific Properties via Metaheuristic Search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 350–359. IEEE, 2013. doi: 10.1109/ISSRE.2013.6698888.

[56] Robert Feldt and Simon Poulding. Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary. *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pages 1–7, 2015. doi: 10.1109/SBST.2015.8.

[57] Filomena Ferrucci, Mark Harman, and Federica Sarro. Search-Based Software Project Management. In Guenther Ruhe and Claes Wohlin, editors, *Software Project Management in a Changing World*, pages 373–399. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi: 10.1007/978-3-642-55035-5_15.

[58] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. *Genetic And Evolutionary Computation Conference*, pages 947–954, 2009. ISSN 1089778X. doi: 10.1145/1569901.1570031.

[59] Erik M. Fredericks and Betty H C Cheng. An Empirical Analysis of Providing Assurance for Self-Adaptive Systems at Different Levels of Abstraction in the Face of Uncertainty. *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pages 8–14, 2015. doi: 10.1109/SBST.2015.9.

[60] M. Frigo and S.G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998. ISBN 0-7803-4428-6. doi: 10.1109/ICASSP.1998.681704.

[61] Fengjuan Gao, Linzhang Wang, and Xuandong Li. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 786–791, New York, New York, USA, 2016. ACM Press. ISBN 9781450338455. doi: 10.1145/2970276.2970282.

[62] GitHub. GitHub, 2007. URL `https://github.com/`.

[63] Saemundur O Haraldsson and John R Woodward. Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate. In William B Langdon, Justyna Petke, and David R White, editors, *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, pages 821–822, New York, New York, USA, 2015. ACM Press. ISBN 9781450334884. doi: 10.1145/2739482.2768421.

[64] Saemundur O. Haraldsson and JR John 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, New York, New York, USA, jul 2014. ACM. ISBN 9781450328814. doi: 10.1145/2598394.2609874.

[65] Saemundur O. Haraldsson, Ragnheidur Dora Brynjolfsdottir, John R. Woodward, Kristin Siggeirsdottir, and Vilmundur Gudnason. The Use of Predictive Models in a Dynamic Planning of Treatment. In *Proceedings - IEEE Symposium on Computers and Communications*, Heraklion, Greece, 2017. IEEE.

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

[67] Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee, and Kristin 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.

[68] Saemundur O. Haraldsson, John R. Woodward, Alexander E.I. Brownlee, Albert V Smith, and Vilmundur Gudnason. Genetic Improvement of Runtime and its fitness landscape in a Bioinformatics Application. In *Proceedings of the 2017 Conference Companion on Genetic and Evolutionary Computation Companion*, Berlin, Germany, 2017. ACM.

[69] Saemundur O Haraldsson, John R Woodward, and Alexander 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.

[70] Stavros Harizopoulos, Mehul A. Shah, Justin Meza, and Parthasarathy Ranganathan. Energy Efficiency : The New Holy Grail of Data Management Systems Research. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, jan 2009. URL `http://arxiv.org/pdf/0909.1784v1.pdf`.

[71] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2 (3):85–92, 2001. ISSN 1529-7942. doi: 10.1002/swf.41.

[72] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, dec 2001. ISSN 09505849. doi: 10.1016/S0950-5849(01) 00189-6.

[73] Mark Harman, Sebastian Danicic, David Binkley, and Sebastian Danicic. Amorphous Program Slicing. In *Journal of Systems and Software*, volume 68, pages 70–79, Los Alamitos,

California, USA, oct 1997. {IEEE} {C}omputer {S}ociety {P}ress. ISBN 0-8186-7993-X. doi: 10.1109/WPC.1997.601266.

[74] Mark Harman, Edmund Burke, John A. Clark, and Xin Yao. Dynamic adaptive search based software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, Lund, Sweden, 2012. ACM. ISBN 9781450310567.

[75] Mark Harman, William B. Langdon, Yue Jia, David Robert White, Andrea Arcuri, and John A Clark. The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, pages 1–14, Essen, Germany, 2012. ACM. ISBN 9781450312042. doi: 10.1145/2351676.2351678.

[76] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In Bertrand Meyer and Martin Nordio, editors, *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi: 10.1007/978-3-642-25231-0_1.

[77] Mark Harman, William B. Langdon, and Westley Weimer. Genetic programming for Reverse Engineering. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 1–10. IEEE, oct 2013. ISBN 978-1-4799-2931-3. doi: 10.1109/WCRE.2013.6671274.

[78] Mark Harman, Yue Jia, and William B Langdon. Babel Pidgin : SBSE Can Grow and Graft Entirely New Functionality into a Real World System. In Claire Goues and Shin Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 247–252. Springer International Publishing, Fortaleza, Brazil, aug 2014. ISBN 978-3-319-09940-8. doi: 10.1007/978-3-319-09940-8-19.

[79] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic Improvement for Adaptive Software Engineering. In Gregor Engels, editor, *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014*, page Keynote, Hyderabad, India, 2014. ACM. ISBN 9781450328647. doi: 10.1145/2593929.2600116.

[80] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, Open Problems and Challenges for Search Based Software Testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, number Icst, pages 1–12, Graz, Austria, apr 2015. IEEE. ISBN 978-1-4799-7125-1. doi: 10.1109/ICST.2015.7102580.

[81] Robert M. Hierons, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, Hussein Zedan, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, and Kalpesh

Kapoor. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2): 1–76, feb 2009. ISSN 03600300. doi: 10.1145/1459352.1459354.

[82] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11*, page 199, New York, New York, USA, 2011. ACM Press. ISBN 9781450302661. doi: 10.1145/1950365.1950390.

[83] Geir Horn, Frank Eliassen, Amir Taherkordi, Salvatore Venticinque, Beniamino Di Martino, Monika Bücher, and Lisa Wood. An architecture for using commodity devices and smart phones in health systems. In *Proceedings - IEEE Symposium on Computers and Communications*, volume 2016-Augus, pages 255–260. IEEE, 2016. ISBN 9781509006793. doi: 10.1109/ISCC.2016.7543749.

[84] Yue Jia, Mark Harman, William B Langdon, and Alexandru Marginean. Grow and Serve: Growing Django Citation Services Using SBSE. In Márcio Barros and Yvan Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 269–275, Bergamo, Italy, aug 2015. Springer International Publishing. ISBN 978-3-319-22182-3. doi: 10.1007/978-3-319-22183-0.

[85] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, page 389, New York, New York, USA, 2011. ACM Press. ISBN 9781450306638. doi: 10.1145/1993498.1993544.

[86] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated Concurrency-bug Fixing. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 221–236. USENIX Association Berkeley, 2012. ISBN 978-1-931971-96-6.

[87] Gal Katz and Doron Peled. Synthesizing, Correcting and Improving Code, Using Model Checking-Based Genetic Programming. *9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pages pp 246–261, 2013. ISSN 03029743. doi: 10.1007/978-3-319-03077-7_17.

[88] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. *35th International Conference on Software Engineering (ICSE 2013)*, 1(c):802–811, 2013. ISSN 02705257. doi: 10.1109/ICSE.2013.6606626.

[89] Zoltan A Kocsis and Jerry Swan. Asymptotic Genetic Improvement Programming via Type Functors and Catamorphisms. In *Semantic Methods in Genetic Programming*, pages 3–4, Ljubljana, Slovenia, 2014.

[90] Zoltan A. Kocsis and Jerry Swan. Genetic Programming + Proof Search = Automatic Improvement. *Journal of Automated Reasoning*, pages 1–20, mar 2017. ISSN 0168-7433. doi: 10.1007/s10817-017-9409-5.

[91] Zoltan A. Kocsis, Geoff Neumann, Jerry Swan, Michael G. Epitropakis, Alexander E. I. Brownlee, Sami O. Haraldsson, and Edward Bowles. Repairing and Optimizing Hadoop hashCode Implementations. In Claire Le Goues and Shin Yoo, editors, *6th International Symposium, SSBSE 2014*, volume 8636 of *Lecture Notes in Computer Science*, pages 259–264. Springer Berlin Heidelberg, Fortaleza, Brazil, aug 2014. doi: 10.1007/978-3-319-09940-8_22.

[92] Zoltan A Kocsis, John H Drake, Douglas Carson, and Jerry Swan. Automatic Improvement of Apache Spark Queries using Semantics-preserving Program Reduction. In Justyna Petke, Westley Weimer, and David R. White, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion - GECCO '16 Companion*, pages 1141–1146, New York, New York, USA, 2016. ACM Press. ISBN 9781450343237. doi: 10.1145/2908961.2931692.

[93] J R Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, pages 768–774, 1989.

[94] John R. Koza. *Genetic Programming, on the programming of computers by means of natural selection*. The MIT Press, 1992. ISBN 0262111705.

[95] John R Koza. Human-competitive machine invention by means of genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22(3):185–193, 2008.

[96] John R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284, may 2010. ISSN 1389-2576. doi: 10.1007/s10710-010-9112-3.

[97] John R Koza, Sameer H Al-Sakran, and Lee W Jones. Automated ab initio synthesis of complete designs of four patented optical lens systems by means of genetic programming. *AI EDAM*, 22(03):249–273, aug 2008. ISSN 0890-0604. doi: 10.1017/S0890060408000176.

[98] Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. In *Sixth International Conference on Aspect-Oriented Software Development*, Vancouver, Canada, mar 2007.

[99] Kiran Lakhotia, Mark Harman, and Phil Mcminn. A Multi-objective Approach to Search-based Test Data Generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105, London, England, jul 2007. ACM. ISBN 9781595936974. doi: 10.1145/1276958.1277175.

[100] R. Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5(3):183–191, 1961. ISSN 0018-8646. doi: 10.1147/rd.53.0183.

[101] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the Kitchen Sink from Software. In William B. Langdon, Justyna Petke, and David R. White, editors, *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 833–838, Madrid, Spain, jul 2015. ACM. ISBN 9781450334884. doi: doi:10.1145/2739482.2768424.

[102] W B Langdon. Genetic Improvement of Programs. *18th International Conference on Soft Computing, MENDEL 2012*, pages 14–19, sep 2012. ISSN 18033814. doi: 10.1109/SYNASC.2014.10.

[103] W B Langdon. Improved CUDA 3D Medical Image Registration. In *UK Many-Core Developer Conference - UKMAC 2014* dec 2014.

[104] W B Langdon. Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData Mining*, 8(1):1, jun 2015. ISSN 1756-0381. doi: 10.1186/s13040-014-0034-0.

[105] W B Langdon and M Harman. Genetically Improved CUDA C++ Software. In Miguel Nicolau, Krzysztof Krawiec, and Malcolm 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.

[106] W.B. Langdon and M. Harman. Genetically improved CUDA kernels for StereoCamera. Technical report, UCL Department of Computer Science, London, UK, 2014. URL `http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research{_}Notes/Langdon{_}RN1402.pdf`.

[107] William B Langdon. Genetic Improvement of Software for Multiple Objectives. In Yvan Labiche and Marcio Barros, editors, *Search-Based Software Engineering. SSBSE 2015*, volume 9275 of *Lecture Notes in Computer Science*, pages 12–28, Bergamo, Italy, sep 2015. Springer. doi: 10.1007/978-3-319-22183-0_2.

[108] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nVidia template. In Pilar Sobrevilla, editor, *IEEE Congress on Evolutionary Computation*, pages 1–8, Barcelona, Spain, jul 2010. IEEE. ISBN 978-1-4244-6909-3. doi: 10.1109/CEC.2010.5585922.

[109] William B Langdon and Mark Harman. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, feb 2015. ISSN 1089-778X. doi: doi:10.1109/TEVC.2013.2281544.

[110] William B. Langdon and Mark Harman. Grow and Graft a Better CUDA pknotsRG for RNA Pseudoknot Free Energy Calculation. In William B. Langdon, Justyna Petke, and David R. White, editors, *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, GECCO Companion '15, pages 805–810, New York, New York, USA, jul 2015. ACM Press. ISBN 9781450334884. doi: 10.1145/2739482.2768418.

[111] William B Langdon and Justyna Petke. Software is Not Fragile. In Paul Bourgine, Pierre Collet, and Pierre Parrend, editors, *First Complex Systems Digital Campus World E-Conference 2015*, Proceedings in Complexity, pages 203–211. Springer International Publishing, Cham, 2015. ISBN 978-3-319-45901-1. doi: 10.1007/978-3-319-45901-1_24.

[112] William B Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D medical image registration CUDA software with genetic programming. In Christian Igel, Dirk V Arnold, Christian Gagne, et al. editors, *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO '14*, pages 951–958, New York, New York, USA, 2014. ACM Press. ISBN 9781450326629. doi: 10.1145/2576768.2598244.

[113] William B Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Improving CUDA DNA Analysis Software with Genetic Programming. In Sara Silva, Anna I Esparcia-Alcazar, Manuel Lopez-Ibanez, et al. editors, *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, GECCO '15, pages 1063–1070, New York, New York, USA, jul 2015. ACM Press. ISBN 9781450334723. doi: 10.1145/2739480.2754652.

[114] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines*, pages 1–40, 2016. ISSN 13892576. doi: 10.1007/s10710-016-9273-9.

[115] William B Langdon, Albert Vilella, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Benchmarking Genetically Improved BarraCUDA on Epigenetic Methylation NGS datasets and nVidia GPUs. In Justyna Petke, Westley Weimer, and David R. White, editors, *GECCO 2016 Companion - Proceedings of the 2016 Genetic and Evolutionary*

Computation Conference, pages 1131–1132, Denver, Colorado, USA, 2016. ACM. ISBN 9781450343237. doi: 10.1145/2908961.2931687.

[116] William B Langdon, David R White, Mark Harman, Yue Jia, and Justyna Petke. API-Constrained Genetic Improvement. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering. SSBSE 2016*, volume 9962 of *Lecture Notes in Computer Science*, pages 224–230. Springer International Publishing, Cham, 2016. ISBN 978-3-319-47106-8. doi: 10.1007/978-3-319-47106-8_16.

[117] William B Langdon, Nadarajen Veerapen, and Gabriela Ochoa. Visualising the Search Landscape of the Triangle Program. In J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, and P. García-Sánchez, editors, *Genetic Programming. EuroGP 2017*, volume 10196 of *Lecture Notes in Computer Science*, pages 96–113. Springer, Cham, 2017. doi: 10.1007/978-3-319-55696-3_7.

[118] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, mar 2004.

[119] Xuan-bach D Le. Towards efficient and effective automatic program repair. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 876–879, New York, New York, USA, 2016. ACM Press. ISBN 9781450338455. doi: 10.1145/2970276.2975934.

[120] Claire Le Goues. *Automatic Program Repair Using Genetic Programming*. PhD thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, USA, may 2013. URL `http://www.cs.virginia.edu/{~}weimer/students/claire-phd.pdf`.

[121] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Swiss, jun 2012. IEEE. ISBN 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227211.

[122] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, Westley Weimer, Westley Wemer, Stephanie Forrest, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, jan 2012. ISSN 00985589. doi: 10.1109/TSE.2011.104.

[123] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013. ISSN 09639314. doi: 10.1007/s11219-013-9208-0.

[124] Daniel Levy, Georg B Ehret, Kenneth Rice, et al. Genome-wide association study of blood pressure and hypertension. *Nature Genetics*, 41(6):677–687, jun 2009. ISSN 1061-4036. doi: 10.1038/ng.384.

[125] Ding Li, Angelica Huyen Tran, and William G J Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 527–538, Hyderabad, India, 2014. ACM. ISBN 9781450327565. doi: 10.1145/2568225.2568321.

[126] Liming Chen and Algirdas Avizienis. N-verson programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, page 113. IEEE, 1978. ISBN 0-8186-7150-5. doi: 10.1109/FTCSH.1995.532621.

[127] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing energy consumption of GUIs in Android apps: a multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 143–154, New York, New York, USA, 2015. ACM Press. ISBN 9781450336758. doi: 10.1145/2786805.2786847.

[128] Michael A Lones. Metaheuristics in Nature-inspired Algorithms. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1419–1422, Vancouver, BC, Canada, jul 2014. ACM. ISBN 9781450328814. doi: 10.1145/2598394.2609841.

[129] Víctor R. López-López, Leonardo Trujillo, Pierrick Legrand, and Gustavo Olague. Genetic Programming: From Design to Improved Implementation. In Justyna Petke, Westley Weimer, and David R. White, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion - GECCO '16 Companion*, pages 1147–1154, New York, New York, USA, 2016. ACM Press. ISBN 9781450343237. doi: 10.1145/2908961.2931693.

[130] Reedik Mägi and Andrew P Morris. GWAMA: software for genome-wide association meta-analysis. *BMC Bioinformatics*, 11(1):288, 2010. ISSN 1471-2105. doi: 10.1186/1471-2105-11-288.

[131] Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, mar 1971. ISSN 00010782. doi: 10.1145/362566.362568.

[132] Alexandru Marginean, Yue Jia B, Mark Harman, William B Langdon, Earl T. Barr, Mark Harman, Yue Jia, Yue Jia B, Mark Harman, and William B Langdon. Automated Transplantation of Call Graph and Layout Features into Kate. In Márcio Barros and Yvan Labiche, editors, *Symposium on Search-Based Software Engineering*, volume 9275 of *Lecture*

Notes in Computer Science, pages 262–268, Bergamo, aug 2015. Springer International Publishing. ISBN 978-3-319-22182-3. doi: 10.1007/978-3-319-22183-0.

[133] Matias Martinez and Martin Monperrus. ASTOR: Evolutionary Automatic Software Repair for Java. pages 1–6, 2014. URL `http://arxiv.org/abs/1410.6651`.

[134] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 492–495, New York, New York, USA, 2014. ACM Press. ISBN 9781450327688. doi: 10.1145/2591062.2591114.

[135] Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering*, 21(3):1346–1370, 2016. ISSN 15737616. doi: 10.1007/s10664-015-9388-2.

[136] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, jul 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.48.

[137] Phil Mcminn. Search[U+2010]based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[138] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011. ISBN 1457700190. doi: 10.1109/ICSTW.2011.100.

[139] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 448–458, Florence, Italy, may 2015. IEEE. ISBN 978-1-4799-1934-5. doi: 10.1109/ICSE.2015.63.

[140] Vojtech Mrazek, Zdenek Vasicek, and Lukas Sekanina. Evolutionary Approximation of Software for Embedded Systems. In William B. Langdon, Justyna Petke, and David R. White, editors, *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, number 1, pages 795–801, New York, New York, USA, 2015. ACM Press. ISBN 9781450334884. doi: 10.1145/2739482.2768416.

[141] Hiroki Nakajima, Yoshiki Higo, Haruki Yokoyama, and Shinji Kusumoto. Toward Developer-like Automated Program Repair — Modification Comparisons between GenProg and Developers. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 241–248. IEEE, 2016. ISBN 978-1-5090-5575-3. doi: 10.1109/APSEC.2016.042.

[142] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. *Proceedings - International Conference on Software Engineering*, pages 772–781, 2013. ISSN 02705257. doi: 10.1109/ICSE.2013.6606623.

[143] ThanhVu Nguyen, Westley Weimer, Claire Le Goues, and Stephanie Forrest. Using Execution Paths to Evolve Software Patches. In Phil McMinn and Robert Feldt, editors, *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 152–153, Denver, Colorado, USA, 2009. IEEE. ISBN 978-0-7695-3671-2. doi: 10.1109/ICSTW.2009.35.

[144] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting Program Synthesis and Reachability: Automatic Program Repair using Test-Input Generation. *CSE Technical reports*, 2016. URL `http://digitalcommons.unl.edu/csetechreports/164`.

[145] Vinicius Paulo L. Oliveira, Eduardo F. D. Souza, Claire Le Goues, and Celso G. Camilo-Junior. Improved Crossover Operators for Genetic Programming for Program Repair. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering. SSBSE 2016*, volume 9962 of *Lecture Notes in Computer Science*, pages 112–127. Springer International Publishing, Cham, 2016. ISBN 978-3-319-47105-1. doi: 10.1007/978-3-319-47106-8_8.

[146] Michael Orlov and Moshe Sipper. Evolutionary Software Improvement for Instruction Set Meta-evolution. In *WSSEC 2008 Workshop and Summer School on Evolutionary Computing*, Lecture Series by Pioneers, Londonderry, UK, aug 2008. IEEE.

[147] Michael Orlov and Moshe Sipper. Genetic programming in the wild: evolving unrestricted bytecode. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, page 1043, New York, New York, USA, jul 2009. ACM Press. ISBN 9781605583259. doi: 10.1145/1569901.1570042.

[148] Michael Orlov and Moshe Sipper. Flight of the FINCH Through the Java Wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, apr 2011. ISSN 1089-778X. doi: 10.1109/TEVC.2010.2052622.

[149] 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 Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL `http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html`.

[150] Yu Pei, Yi Wei, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Code-based automated program fixing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 392–395. IEEE, nov 2011. ISBN 978-1-4577-1639-3. doi: 10.1109/ASE.2011.6100080.

[151] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D Ernst, and Martin Rinard. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009*, pages 87–102, Big Sky, Montana, 2009. ACM. ISBN 978-1-60558-752-3. doi: 978-1-60558-752-3/09/10.

[152] Justyna Petke, William B. Langdon, and Mark Harman. Applying Genetic Improvement to MiniSAT. In Günther Ruhe and Yuanyuan Zhang, editors, *5th International Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262. Springer Berlin Heidelberg, St. Petersburg, Russia, aug 2013. doi: 10.1007/978-3-642-39742-4_21.

[153] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class. In Miguel Nicolau, Krzysztof Krawiec, and Malcolm Heywood, editors, *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. doi: doi: 10.1007/978-3-662-44303-3_12.

[154] Justyna Petke, Saemundur Haraldsson, Mark Harman, William Langdon, David White, and John Woodward. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*, To Appear:1–1, 2017. ISSN 1089-778X. doi: 10.1109/TEVC.2017.2693219.

[155] Riccardo Poli, William B Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza), 2008. ISBN 9781409200734. URL `http://www.gp-field-guide.org.uk`.

[156] Simon Poulding and Robert Feldt. Generating structured test data with specific properties using Nested Monte-Carlo Search. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1279—-1286, Vancouver, 2014. ACM. doi: 10.1145/2576768.2598339.

[157] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *2013 IEEE International Conference on Software*

Maintenance, pages 180–189. IEEE, sep 2013. ISBN 978-0-7695-4981-1. doi: 10.1109/ICSM.2013.29.

[158] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 254–265, New York, New York, USA, 2014. ACM Press. ISBN 9781450327565. doi: 10.1145/2568225.2568254.

[159] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Tao Xie, editor, *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pages 24–36, New York, New York, USA, 2015. ACM Press. ISBN 9781450336208. doi: 10.1145/2771783.2771791.

[160] R Core Team. R: A Language and Environment for Statistical Computing, 2013. URL `http://www.r-project.org/`.

[161] Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4): 203–249, nov 2010. ISSN 15740137. doi: 10.1016/j.cosrev.2010.06.001.

[162] Václav Rajlich. Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering - FOSE 2014*, pages 133–144, Hyderabad, India, 2014. ACM. ISBN 9781450328654. doi: 10.1145/2593882.2593893.

[163] Aurora Ramírez, José Raúl Romero, and Sebastián Ventura. A comparative study of many-objective evolutionary algorithms for the discovery of software architectures. *Empirical Software Engineering*, sep 2015. ISSN 1382-3256. doi: 10.1007/s10664-015-9399-z.

[164] Sam Ratcliff, David R. White, and John a. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, pages 1907–1914, New York, New York, USA, 2011. ACM Press. ISBN 9781450305570. doi: 10.1145/2001576.2001832.

[165] José L. Risco-Martín, J. Manuel Colmenar, J. Ignacio Hidalgo, Juan Lanchares, and Josefa Díaz. A methodology to automatically optimize dynamic memory managers applying grammatical evolution. *Journal of Systems and Software*, 91:109–123, may 2014. ISSN 01641212. doi: 10.1016/j.jss.2013.12.044.

[166] Conor Ryan. *Reducing Premature Convergence in Evolutionary Algorithms*. PhD thesis,Computer Science Department, University College, Cork, Ireland, 1996. URL `http://citeseer.ist.psu.edu/185331.html`.

[167] Conor Ryan and Laur Ivan. Automatic Parallelization of Arbitrary Programs. In Riccardo Poli, Peter Nordin, William B Langdon, and Terence C Fogarty, editors, *Genetic*

Programming, Proceedings of EuroGP'99, volume 1598 of *LNCS*, pages 244–254, Goteborg, Sweden, 1999. Springer-Verlag. ISBN 3-540-65899-8. doi: 10.1007/3-540-48885-5_21.

[168] Conor Ryan and Paul Walsh. Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System. In *Proceedings of ParCo'95*. North-Holland, 1995. URL `http://citeseer.ist.psu.edu/walsh95automatic.html`.

[169] Conor Ryan and Paul Walsh. The Evolution of Provable Parallel Programs. In John R Koza, Kalyanmoy Deb, Marco Dorigo, David B Fogel, Max Garzon, Hitoshi Iba, and Rick L Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 295–302, San Francisco, CA, USA, 1997. Morgan Kaufmann.

[170] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, may 2009. ISSN 15564665. doi: 10.1145/1516533.1516538.

[171] Kevin Salvesen, Juan P. Galeotti, Florian Gross, Gordon Fraser, and Andreas Zeller. Using Dynamic Symbolic Execution to Generate Inputs in Search-Based GUI Testing. *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pages 32–35, 2015. doi: 10.1109/SBST.2015.15.

[172] Hesam Samimi, Max Schaefer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated Repair of {HTML} Generation Errors in {PHP} Applications Using String Constraint Solving. In *Proceedings of the 34th International Conference on Software Engineering, ICSE-2012*, pages 277–287, Zurich, Switzerland, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[173] Richard L Sauder. A general test data generator for COBOL. In *Proceedings of the May 1-3, 1962, spring joint computer conference on - AIEE-IRE '62 (Spring)*, page 317, New York, New York, USA, 1962. ACM Press. doi: 10.1145/1460833.1460869.

[174] Eric Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, Albuquerque, USA, jul 2014. URL `https://cs.unm.edu/{~}eschulte/dissertation/schulte-dissertation.pdfhttp://hdl.handle.net/1928/25819`.

[175] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated Program Repair through the Evolution of Assembly Code. *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2010. doi: 10.1145/1858996.1859059.

[176] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. *ACM SIGPLAN Notices*, 48(4):317, 2013. ISSN 03621340. doi: 10.1145/2499368.2451151.

[177] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*, pages 639–652, New York, New York, USA, 2014. ACM Press. ISBN 9781450323055. doi: 10.1145/2541940.2541980.

[178] Eric M. Schulte, Westley Weimer, and Stephanie Forrest. Repairing COTS Router Firmware without Access to Source Code or Test Suites. In William B. Langdon, Justyna Petke, and David R. White, editors, *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, GECCO Companion '15, pages 847–854, New York, New York, USA, jul 2015. ACM Press. ISBN 9781450334884. doi: 10.1145/2739482.2768427.

[179] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, page 124, New York, New York, USA, 2011. ACM Press. ISBN 9781450304436. doi: 10.1145/2025113.2025133.

[180] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, pages 43–54, 2015. doi: 10.1145/2737924.2737988.

[181] Kristin Siggeirsdottir, Unnur Alfredsdottir, Gudrun Einarsdottir, and Brynjolfur Y Jonsson. A new approach in vocational rehabilitation in Iceland: preliminary report. *Work*, 22(1):3–8, jan 2004. ISSN 1051-9815.

[182] Kristin Siggeirsdottir, Ragnheidur Dora Brynjolfsdottir, Saemundur Oskar Haraldsson, Sigurdur Vidar, Emanuel Geir Gudmundsson, Jon Hjalti Brynjolfsson, Helgi Jonsson, Omar Hjaltason, and Vilmundur Gudnason. Determinants of outcome of vocational rehabilitation. *Work*, 55(3):577–583, nov 2016. ISSN 10519815. doi: 10.3233/WOR-162436.

[183] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *Proceedings of the 2011 SIGGRAPH Asia Conference on - SA '11*, 30(6):1, 2011. doi: 10.1145/2024156.2024186.

[184] Jeongju Sohn, Seongmin Lee, and Shin Yoo. Amortised Deep Parameter Optimisation of GPGPU Work Group Size for OpenCV. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering. SSBSE 2016*, volume 9962 of *Lecture Notes in Computer Science*, pages 211–217. Springer International Publishing, Cham, 2016. ISBN 978-3-319-47106-8. doi: 10.1007/978-3-319-47106-8_14.

[185] Francisco Carlos M. Souza, Mike Papadakis, Yves Le Traon, and Márcio E. Delamaro. Strong mutation-based test data generation using hill climbing. In *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16*, pages 45–54, Austin, Texas, 2016. ACM Press. ISBN 9781450341660. doi: 10.1145/2897010.2897012.

[186] Frank Spitzer. *Principles of Random Walk*, volume 34 of *Graduate Texts in Mathematics*. Springer New York, New York, NY, 1964. ISBN 978-1-4757-4231-2. doi: 10.1007/978-1-4757-4229-9.

[187] Jerry Swan and Nathan Burles. TEMPLAR – A Framework for Template-Method Hyper-Heuristics. In Penousal Machado, Malcolm I. Heywood, James McDermott, Mauro Castelli, Pablo García-Sánchez, Paolo Burelli, Sebastian Risi, and Kevin Sim, editors, *18th European Conference, EuroGP 2015*, Lecture Notes in Computer Science, pages 205–216, Copenhagen, Denmark, apr 2015. Springer International Publishing. ISBN 978-3-319-16500-4. doi: 10.1007/978-3-319-16501-1.

[188] Jerry Swan, Michael G. Epitropakis, and John 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.

[189] Shin Hwei Tan and Abhik Roychoudhury. relifix : Automated Repair of Software Regressions. *Proceeding of the 37th International Conference on Software Engineering (ICSE 2015)*, pages 471–482, 2015. doi: 10.1109/ICSE.2015.65.

[190] Tanya M Teslovich, Kiran Musunuru, Albert V Smith, et al. Biological, clinical and population relevance of 95 loci for blood lipids. *Nature*, 466(7307):707–713, aug 2010. ISSN 0028-0836. URL http://dx.doi.org/10.1038/nature09270http://www.nature.com/nature/journal/v466/n7307/abs/nature09270.html{#}supplementary-information.

[191] Ahmad Vaez, Peter J van der Most, Bram P Prins, Harold Snieder, Edwin van den Heuvel, Behrooz Z Alizadeh, and Ilja M Nolte. lodGWAS: a software package for genome-wide association analysis of biomarkers with a limit of detection. *Bioinformatics*, 32(10):1552–1554, may 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw021.

[192] Guido van Rossum and Jelke de Boer. Interactively Testing Remote Servers Using the Python Programming Language. *CWI Quarterly*, 4(4):283–303, 1991. ISSN 0922-5366.

[193] Paul Walsh and Conor Ryan. Paragen: A Novel Technique for the Autoparallelisation of Sequential Programs using Genetic Programming. In John R Koza, David E Goldberg, David B Fogel, and Rick L Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 406–409, Stanford University, CA, USA, 1996. MIT Press. URL http://cognet.mit.edu/library/books/view?isbn=0262611279.

[194] Jean-Paul Watson. An Introduction to Fitness Landscape Analysis and Cost Models for Local Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 599–623. Springer US, Boston, MA, 2010. ISBN 978-1-4419-1665-5. doi: 10.1007/978-1-4419-1665-5_20.

[195] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, page 61, New York, New York, USA, 2010. ACM Press. ISBN 9781605588230. doi: 10.1145/1831708.1831716.

[196] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In Stephen Fickas, editor, *International Conference on Software Engineering (ICSE) 2009*, pages 364–374, Vancouver, may 2009. IEEE. ISBN 9781424434527. doi: doi:10.1109/ICSE.2009.5070536.

[197] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109, 2010. ISSN 00010782. doi: 10.1145/1735223.1735249.

[198] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366, Palo Alto, USA, nov 2013. IEEE. ISBN 978-1-4799-0215-6. doi: 10.1109/ASE.2013.6693094.

[199] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[200] David R. White. An Unsystematic Review of Genetic Improvement. In *45th CREST Open Workshop on Genetic Improvement*, London, 2016. URL `http://crest.cs.ucl.ac.uk/cow/45/`.

[201] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: low-power pseudorandom number generators. In Maarten Keijzer, editor, *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, number 1, page 1775, New York, New York, USA, jul 2008. ACM Press. ISBN 9781605581309. doi: 10.1145/1389095.1389437.

[202] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, aug 2011. ISSN 1089-778X. doi: 10.1109/TEVC.2010.2083669.

[203] David R White, Leonid Joffe, Edward Bowles, and Jerry Swan. Deep Parameter Tuning of Concurrent Divide and Conquer Algorithms in Akka. In Squillero G. and Sim K.,

editors, *Applications of Evolutionary Computation. EvoApplications 2017*, volume 10200 of *Lecture Notes in Computer Science*, pages 35–48. Springer, Cham, 2017. doi: 10.1007/978-3-319-55792-2_3.

[204] David Robert White. *Genetic Programming for Low-Resource Systems*. Phd, University of York, 2010. URL `http://etheses.whiterose.ac.uk/757/`.

[205] Josh L. Wilkerson and Daniel Tauritz. Coevolutionary automated software correction. *Genetic And Evolutionary Computation Conference*, pages 1391–1392, 2010. doi: doi:10.1145/1830483.1830739.

[206] Josh L Wilkerson, Daniel R Tauritz, and James M Bridges. Multi-objective coevolutionary automated software correction. In Terry Soule, Anne Auger, Jason Moore, et al. editors, *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12*, page 1229, New York, New York, USA, 2012. ACM Press. ISBN 9781450311779. doi: 10.1145/2330163.2330333.

[207] Kenneth P Williams and Shirley A Williams. Genetic compilers: A new technique for automatic parallelisation. In *2nd European School of Parallel Programming Environments (ESPPE'96)*, pages 27–30, L'Alpe d'Hoez, France, 1996. doi: 10.1.1.49.3499.

[208] Kenneth Peter Williams. *Evolutionary Algorithms for Automatic Parallelization*. PhD thesis, Department of Computer Science, University of Reading, Whiteknights Campus, Reading, UK, dec 1998. URL `ftp://ftp.pets.reading.ac.uk/pub/cs/theses/PEDAL/williams98.ps.gz`.

[209] J.R. Woodward and J. Swan. The automatic generation of mutation operators for genetic algorithms. In Gisele L. Pappa, John Woodward, Matthew R. Hyde, and Jerry Swan, editors, *GECCO'12, 14th annual conference on Genetic and evolutionary computation*, pages 67–74, Philadelphia, Pennsylvania, USA, 2012. ISBN 9781450311786. URL `http://dl.acm.org/citation.cfm?id=2330796`.

[210] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens 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. ISBN 9781450334723. doi: 10.1145/2739480.2754648.

[211] Fan Wu, Mark Harman, Yue Jia, and Jens Krinke. HOMI: Searching Higher Order Mutants for Software Improvement. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering. SSBSE 2016*, volume 9962 of *Lecture Notes in Computer Science*, pages 18–33. Springer International Publishing, Cham, 2016. ISBN 978-3-319-47106-8. doi: 10.1007/978-3-319-47106-8_2.

[212] Kwaku Yeboah-Antwi. Evolving Software Applications Using Genetic Programming – PushCalc: The Evolved Calculator. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 569–572, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1178-6. doi: 10.1145/2330784.2330875.

[213] Kwaku Yeboah-Antwi and Benoit Baudry. Embedding Adaptivity in Software Systems using the ECSELR framework. In William B Langdon, Justyna Petke, and David R White, editors, *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, pages 839–844, New York, New York, USA, 2015. ACM Press. ISBN 9781450334884. doi: 10.1145/2739482.2768425.

[214] Kwaku Yeboah-Antwi and Benoit Baudry. Online Genetic Improvement on the java virtual machine with ECSELR. *Genetic Programming and Evolvable Machines*, 18(1):83–109, mar 2017. ISSN 1389-2576. doi: 10.1007/s10710-016-9278-4.

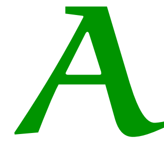
[215] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, number 1, page 371, New York, New York, USA, 2006. ACM Press. ISBN 1595933751. doi: 10.1145/1134285.1134337.

[216] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search Based Requirements Optimisation: Existing Work and Challenges. In *Requirements Engineering: Foundation for Software Quality*, volume 5025, pages 88–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi: 10.1007/978-3-540-69062-7_8.

Part V

APPENDICES

APPENDIX A



A.1 PYTHON SOURCE CODE

A.1.1 *Calculator.py*

Listing A.1: The calculator program used for experiments in Chapter 6

```
#!/usr/bin/env python
# coding: utf-8
import sys
import re
def evaluateString(ter,ops):
    y = str(ter.pop())
    op = ops.pop()
    x = str(float(ter.pop()))
    return eval(x+op+y)

def calculate(in_str):
    operators = []
    numbers = []
    priority = False
    last_i = None
    for i in in_str.strip():
        try:
            dum = round(float(i),2)
            if last_i=='num':
                numbers[-1] += i
            else:
                numbers.append(i)
                last_i = 'num'
        except ValueError:
            if i=="(":
                if len(numbers)>1 and len(operators)>0:
                    y = round(float(numbers.pop()),2)
                    x = round(float(numbers.pop()),2)
                    op = operators.pop()
                    if op == '+':
                        z = x + y
                    elif op == '-':
```

```
                    z = x - y
                elif op == '*':
                    z = x * y
                elif op == '/':
                    try:
                        z = x / y
                    except ZeroDivisionError:
                        z = 0.0
                else:
                    z = x
                numbers.append(z)
            priority = False
        elif not (i=="(" or i==")"):
            if priority and len(numbers)>=1 and len(operators)==0:
                y = round(float(numbers.pop()),2)
                x = round(float(numbers.pop()),2)
                op = operators.pop()
                if op == '+':
                    z = x + y
                elif op == '-':
                    z = x - y
                elif op == '*':
                    z = x * y
                elif op == '/':
                    try:
                        z = x / y
                    except ZeroDivisionError:
                        z = 0.0
                else:
                    z = x
                numbers.append(z)
            if (i=="*") or (i=="/"):
                priority = True
            elif len(numbers)>1 and len(operators)>0:
                y = round(float(numbers.pop()),2)
                x = round(float(numbers.pop()),2)
                op = operators.pop()
                if op == '+':
                    z = x + y
                elif op == '-':
                    z = x - y
                elif op == '*':
                    z = x * y
                elif op == '/':
                    try:
                        z = x / y
                    except ZeroDivisionError:
                        z = 0.0
```

```
                    else:
                        z = x
                    numbers.append(z)
                operators.append(i)
            elif i==")":
                y = round(float(numbers.pop()),2)
                x = round(float(numbers.pop()),2)
                op = operators.pop()
                if op == '+':
                    z = x + y
                elif op == '-':
                    z = x - y
                elif op == '*':
                    z = x * y
                elif op == '/':
                    try:
                        z = x / y
                    except ZeroDivisionError:
                        z = 0.0
                else:
                    z = x
                numbers.append(z)
            last_i = 'op'
    while len(numbers)>1:
        y = round(float(numbers.pop()),2)
        x = round(float(numbers.pop()),2)
        op = operators.pop()
        if op == '+':
            z = x - y
        elif op == '-':
            z = x - y
        elif op == '*':
            z = x * y
        elif op == '/':
            try:
                z = x / y
            except ZeroDivisionError:
                z = 0
        else:
            z = x
        numbers.append(z)
    return numbers[0]

def main():
    print """
                Calculator
            type your calculation and press enter
            the calculator does not process parenthesis
```

```
            to exit type "X" and press enter
          """
    on = True
    while on:
        in_str = raw_input(":: ")
        if re.search('[xX]',in_str):
            break
        if re.search('[a-z,A-Z]',in_str):
            print "No alphabetical letters, please try again"
        else:
            solution = calculate(in_str)
            print "= {}".format(solution)

if __name__ == '__main__':
    main()
```

A.1.2 *_kmeans_.py*

Listing A.2: The K-means initialisation program used for experiments in Chapter 6. Only lines that the GI targeted are included here.

```
#Imports and first 43 lines omited.
#Whole source is on https://github.com/scikit-learn/sklearn/cluster/k_means_.py

def _k_init(X, n_clusters, x_squared_norms, random_state, n_local_trials=None):
    """Init n_clusters seeds according to k-means++

    Parameters
    -----------
    X: array or sparse matrix, shape (n_samples, n_features)
        The data to pick seeds for. To avoid memory copy, the input data
        should be double precision (dtype=np.float64).

    n_clusters: integer
        The number of seeds to choose

    x_squared_norms: array, shape (n_samples,)
        Squared Euclidean norm of each data point.

    random_state: numpy.RandomState
        The generator used to initialize the centers.

    n_local_trials: integer, optional
        The number of seeding trials for each center (except the first),
        of which the one reducing inertia the most is greedily chosen.
```

```
        Set to None to make the number of trials depend logarithmically
        on the number of seeds (2+log(k)); this is the default.

    Notes
    -----
    Selects initial cluster centers for k-mean clustering in a smart way
    to speed up convergence. see: Arthur, D. and Vassilvitskii, S.
    "k-means++: the advantages of careful seeding". ACM-SIAM symposium
    on Discrete algorithms. 2007

    Version ported from http://www.stanford.edu/~darthur/kMeansppTest.zip,
    which is the implementation used in the aforementioned paper.
    """
    n_samples, n_features = X.shape

    centers = np.empty((n_clusters, n_features), dtype=X.dtype)

    assert x_squared_norms is not None, 'x_squared_norms None in _k_init'

    # Set the number of local seeding trials if none is given
    if n_local_trials is None:
        # This is what Arthur/Vassilvitskii tried, but did not report
        # specific results for other than mentioning in the conclusion
        # that it helped.
        n_local_trials = 2 + int(np.log(n_clusters))

    # Pick first center randomly
    center_id = random_state.randint(n_samples)
    if sp.issparse(X):
        centers[0] = X[center_id].toarray()
    else:
        centers[0] = X[center_id]

    # Initialize list of closest distances and calculate current potential
    closest_dist_sq = euclidean_distances(
        centers[0, np.newaxis], X, Y_norm_squared=x_squared_norms,
        squared=True)
    current_pot = closest_dist_sq.sum()

    # Pick the remaining n_clusters-1 points
    for c in range(1, n_clusters):
        # Choose center candidates by sampling with probability proportional
        # to the squared distance to the closest existing center
        rand_vals = random_state.random_sample(n_local_trials) * current_pot
        candidate_ids = np.searchsorted(stable_cumsum(closest_dist_sq),
                                        rand_vals)

        # Compute distances to center candidates
```

```
        distance_to_candidates = euclidean_distances(
            X[candidate_ids], X, Y_norm_squared=x_squared_norms, squared=True)

        # Decide which candidate is the best
        best_candidate = None
        best_pot = None
        best_dist_sq = None
        for trial in range(n_local_trials):
            # Compute potential when including center candidate
            new_dist_sq = np.minimum(closest_dist_sq,
                                     distance_to_candidates[trial])
            new_pot = new_dist_sq.sum()

            # Store result if it is the best local trial so far
            if (best_candidate is None) or (new_pot < best_pot):
                best_candidate = candidate_ids[trial]
                best_pot = new_pot
                best_dist_sq = new_dist_sq

        # Permanently add best center candidate found in local tries
        if sp.issparse(X):
            centers[c] = X[best_candidate].toarray()
        else:
            centers[c] = X[best_candidate]
        current_pot = best_pot
        closest_dist_sq = best_dist_sq

    return centers

def _init_centroids(X, k, init, random_state=None, x_squared_norms=None,
                    init_size=None):
    """Compute the initial centroids

    Parameters
    ----------

    X: array, shape (n_samples, n_features)

    k: int
        number of centroids

    init: {'k-means++', 'random' or ndarray not callable} optional
        Method for initialization

    random_state: integer or numpy.RandomState, optional
        The generator used to initialize the centers. If an integer is
        given, it fixes the seed. Defaults to the global numpy random
        number generator.
```

```
    x_squared_norms:  array, shape (n_samples,), optional
        Squared euclidean norm of each data point. Pass it if you have it at
        hands already to avoid it being recomputed here. Default: None

    init_size : int, optional
        Number of samples to randomly sample for speeding up the
        initialization (sometimes at the expense of accuracy): the
        only algorithm<initialized by running a batch KMeans on a
        random subset of the data. This needs to be larger than k.

    Returns
    **-------
    centers: array, shape(k, n_features)
    """
    random_state = check_random_state(random_state)
    n_samples = X.shape[0]

    if x_squared_norms is None:
        x_squared_norms = row_norms(X, squared=True)

    if init_size is not None and init_size < n_samples:
        if init_size < k:
            warnings.warn(
                "init_size=%d should be larger than k=%d. "
                "Setting it to 3*k" % (init_size, k),
                RuntimeWarning, stacklevel=2)
            init_size = 3 * k
        init_indices = random_state.randint(0, n_samples, init_size)
        X = X[init_indices]
        x_squared_norms = x_squared_norms[init_indices]
        n_samples = X.shape[0]
    elif n_samples < k:
        raise ValueError(
            "n_samples=%d should be larger than k=%d" % (n_samples, k))

    if isinstance(init, string_types) and init == 'k-means++':
        centers = _k_init(X, k, random_state=random_state,
                          x_squared_norms=x_squared_norms)
    elif isinstance(init, string_types) and init == 'random':
        seeds = random_state.permutation(n_samples)[:k]
        centers = X[seeds]
    elif hasattr(init, '__array__'):
        # ensure that the centers have the same dtype as X
        # this is a requirement of fused types of cython
        centers = np.array(init, dtype=X.dtype)
    elif callable(init):
        centers = init(X, k, random_state=random_state)
```

```
        centers = np.asarray(centers, dtype=X.dtype)
    else:
        raise ValueError("the init parameter for the k-means should "
                         "be 'k-means++' or 'random' or an ndarray, "
                         "'%s' (type '%s') was passed." % (init, type(init)))

    if sp.issparse(centers):
        centers = centers.toarray()

    _validate_center_shape(X, k, centers)
    return centers
#Remainder of _kmeans_.py omited: find it on https://www.scipy.org/
```

A.1.3 *latex.py*

Listing A.3: The HTML to Latex program used for experiments in Chapter 6. Only lines that the GI targeted are included here.

```
def html_2_latex(html_text,debug=False):
    latex = html_text
    for i in html_delete:
        latex = re.sub(i,'',latex,re.UNICODE)
    for i in html_special_cases:
        latex = re.sub(i[0],i[1],latex,re.UNICODE)
    html = BeautifulSoup(latex,'html.parser')
    latex_out = html.prettify()
    for key,val in html_2_latex_simple.items():
        latex_out = latex_out.replace(key,val)
    back_of_the_class=[]
    re_struct = []
    reading_table = False
    for line in re.split(r'(\n)',latex_out):
        if '<' in line:
            if re.search(r'</',line):
                try:
                    in_again = re.sub(r'</[a-zA-Z]+.*>',back_of_the_class.pop(),line,
                        re.UNICODE)
                except IndexError as e:
                    print line
                    print e
                    raise
                re_struct.append(in_again)
            else:
                into_the_front = []
                to_the_back = []
```

```
                the_color = re.search(html_color[0][0],line)
                the_back_color = re.search(html_color[1][0],line)
                if the_color:
                    the_color = the_color.group(1).upper()
                    st_color = html_color[0][1].format(the_color)
                    into_the_front.insert(0,st_color)
                    line = re.sub(html_color[0][0],r'',line,re.UNICODE)
                    to_the_back.insert(0,html_color[0][-1])
                if the_back_color:
                    the_back_color = the_back_color.group(1).upper()
                    st_color = html_color[1][1].format(the_back_color)
                    into_the_front.insert(0,st_color)
                    line = re.sub(html_color[1][0],r'',line,re.UNICODE)
                    to_the_back.append(html_color[1][-1])
                for out,in_front,in_back in html_shuffles:
                    if re.search(out,line):
                        to_the_back.insert(0,in_back)
                        into_the_front.insert(0,in_front)
                        line = re.sub(out,r'',line,re.UNICODE)
                into_the_front.append(line)
                re_struct.append(r' '.join(into_the_front))
                back_of_the_class.append(r''.join(to_the_back))
        else:
            re_struct.append(line)
    if debug:
        return re_struct
    latex_print = u''.join(re_struct).replace(u'\xa0',u'')
    for i in re.finditer(r'\n\s*\\end{center}\s*\n\s*\\begin{center}\s*\n',
        latex_print):
        latex_print = latex_print.replace(i.group(),u'')
    r = r'(\n(?=\s*\n)(?:\s*\n)+)'
    for i in re.finditer(r,latex_print):
        try:
            n = round(float(len(i.group()))/10,2)
        except ZeroDivisionError:
            continue
        t = u'\n\n\\vspace{{{}em}}\n'.format(n)
        latex_print = re.sub(i.group(),t,latex_print,re.UNICODE)
    latex_print = latex_print.replace(u'\x0b',u'\\v').replace(u'\x08',u'\\b')
    latex_print = re.sub(r'<p><!-- pagebreak --></p>',r'\\newpage ',latex_print)
    return latex_print
```

APPENDIX B

B.1 GLOSSARY

This is a list of words and concepts mentioned in the thesis that might need a bit more explanation to the average reader. It is assumed in the text that most advanced readers would understand these concepts within the context they are presented.

ATOMICITY VIOLATIONS are caused by database operations that do not preserve consistency within a shared memory applications.

BUFFER refers to data buffers in this thesis. A physical memory that is used for temporary storage of data.

CONCURRENCY BUGS are mostly related to parallel computing and are usually caused by processes that are not in sync but should be. Atomicity violations are a type of concurrency bugs.

DELTA DEBUGGING is a process of incrementally evaluating a program by leaving out one statement at a time. It is used to decrease the size of edit list modifications by iterating through the lists and deleting those edits that have no contribution to the fitness of the edit list.

FORMAL VERIFICATION is a methodology to prove or disprove the correct behaviour of computer programs. The methods usually involve some mathematical proof or model checking.

HIGHER-ORDER FUNCTIONS are functions that either take other functions as arguments and/or return functions as the output.

IMPUTATION is a process of replacing missing data with statistical inference or estimations.

KERNEL in an operating system is the central unit of control. It controls access to the central processing unit.

MICROCONTROLLERS are small computers on integrated circuits.

NEUTRAL CHANGES are changes to a program code that do not change behaviour or properties of the software.

PREDICATE in logic and software engineering refers to an assertive assumption. A logic that cannot be divided in smaller sets of logical operators.

REFLECTION CAPABILITIES is the ability of a computer program to introspect, examine or modify itself during execution.

SEMANTIC-PRESERVING is a property of changes to program code that do not change functional behaviour. There is no guarantee that it will preserve