Programming for Longer Battery Life

Anna Frances Rasburn

**April 2019**

**Dissertation submitted in partial fulfilment for the degree of   
Bachelor of Science with Honours in Computing Sciences**

**Division of Computing Science and Mathematics  
University of Stirling**

Abstract

***Problem*** - This project focuses on using optimisation techniques applied to Java, particularly those which use Genetic Improvement, code refactoring and code changes. This performed by researching and investigating ways in which Java applications can be modified to reduce CPU power and therefore require less battery life. It is hoped that this research can lead to help find out a way to save energy on small devices the same principles can be applied to larger devices such as large-scale data centres which would lead to a big environmental benefit.

***Methodology*** - This project has created a software application which applies edits to source code. The edits included the following operations: moving, deleting, swapping block statements, inserting lines and changing the use of If statements. The application uses artificial intelligence search techniques such as Hill Climbing and Genetic Algorithms to see if it is possible to find an optimised version of the source code by identifying the impact of the edits and iteratively amending the edits.

***Achievements*** – This project inspired a creation an optimisation toolkit that takes in source code and changes said source code so that new source code uses less energy that its previous version. This project inspired further research into effects on loops in sorting algorithms and energy usage.

Attestation

I understand the nature of plagiarism, and I am aware of the University’s policy on this.

I certify that this dissertation reports original work by me during my University project except for the following (*adjust according to the circumstances*):

* The technology review in Section 2.5 was largely taken from [17].
* The code discussed in Section 3.1 was created by Acme Corporation ([*www.acme-corp.com*](http://www.acme-corp.com)*/JavaExpert*) and was used in accordance with the licence supplied.
* The code discussed in Section 3.5 was written by my supervisor.
* The code discussed in Section 4.2 was developed by me during a vacation placement with the collaborating company. In addition, this used ideas I had already developed in my own time.

**Signature** *(you must delete this, then sign and date this page)* **Date**

Acknowledgements

Acknowledge anyone who has helped you in your work such as your supervisor, technical support staff, fellow students or external organisations. Acknowledge the source of any work that is not your own.

Table of Contents

The table of contents below is automatically generated from the paragraphs of style *Heading N* and *Unnumbered N*. To update this after revisions, right-click in the table and choose *Update Field* for the entire table.

Abstract i

Attestation ii

Acknowledgements iii

Table of Contents iv

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Scope and Objectives 1

1.1.1 Scope 1

1.1.2 Users 1

1.1.3 Objectives 1

1.2 Achievements 2

1.3 Overview of Dissertation 2

2 State-of-The-Art 3

2.1 Genetic Improvement and Changing Software 3

2.1.1 Genetic Improvement 3

2.1.2 GIN 4

2.1.3 GI In other Languages 6

2.2 Tools for energy measurement 7

2.2.1 Opacitor 7

2.2.2 Jalen 7

2.2.3 Optimization Framework for Mobile Applications. 9

2.2.4 Code Refactoring’s effects on Energy Usage 10

2.3 Green Software Engineering 11

3 Background 12

3.1 Optimisation regarding Java 12

3.2 Genetic Improvement 12

3.3 Genetic Algorithms 12

3.4 Fitness 14

4 Methodology 15

4.1 Creation of Toolkit 15

4.2 Tools Used 15

4.2.1 Opacitor 15

4.2.2 Test-Runner 15

4.2.3 Fitness 15

4.2.4 Unit tests 16

4.2.5 Maven 16

4.2.6 Spring 16

4.3 Development of Toolkit 17

4.3.1 Gin-Main 17

4.3.1.1 Patch 17

4.3.1.2 Edits 17

4.3.2 Genetic Algorithm 18

4.3.2.1 Initialize Population 18

4.3.2.2 Calculate Fitness 18

4.3.2.3 Selection 19

4.3.2.4 Crossover 19

4.3.2.5 Mutation 19

4.3.3 Hill - Climbing 19

4.3.4 Testing 19

4.3.4.1 Test Driven Development 19

4.4 Structure 20

4.4.1 Modules 20

4.4.2 Overview 20

5 Experiments 21

5.1 Recursive vs Iterative Loops 21

5.1.1 Introduction 21

5.1.2 Method 21

5.1.3 Results 22

5.1.3.1 Arrays in Random Order 22

5.1.3.2 Arrays in Ascending Order 23

5.1.3.3 Arrays in Descending Order 24

5.1.4 Summary 25

*5.1.4.1* *Further Work* 26

6 Use Cases 28

6.1 JCodec 28

6.2 Triangle Program 28

7 Results 29

8 Conclusion 30

8.1 Summary 30

8.2 Evaluation 30

8.3 Future Work 30

Appendix 1 31

Appendix 2 – User guide 32

Appendix 3 – Installation guide 33

Glossary 34

References 35

List of Figures

[**Figure 2.1** Diagram of changing code lifecycle with Genetic Improvement 3](#_Toc702393)

[**Figure 2.2** Example of GIN output 4](#_Toc702394)

[**Figure 2.3** UML diagram of how GIN works 5](#_Toc702395)

[**Figure 2.4** Architecture of Jalen 8](#_Toc702396)

[**Figure 2.5** Ratings for Mobile applications 11](#_Toc702397)

[**Figure 3.1** Flow Diagram of a Genetic Algorithm 13](#_Toc702398)

[**Figure 5.1** Example of a Bean for a removeLineEdit 18](#_Toc702399)

[**Figure 6.1** Graph for relationship between arrays of random order and energy measurement 23](#_Toc702400)

[**Figure 6.2** Graph for relationship between arrays in ascending order and energy measurement 24](#_Toc702401)

[**Figure 6.3** Graph for relationship between array in descending order and energy measurement 25](#_Toc702402)

# List of Tables

[**Table 1** Example to Demonstrate the levels of Fitness. P denotes a pass, F a failure. 14](#_Toc702406)

[**Table 2** Results of Bubble Sort tests with an Array in Random Order 22](#_Toc702407)

[**Table 3** Results of Bubble Sort test with an Array in ascending order 23](#_Toc702408)

[**Table 5** Results of Bubble Sort test with an Array in descending order 24](#_Toc702409)

# Introduction

## [Scope](http://www.cs.stir.ac.uk/~kjt/research/conformed.html) and Objectives

### Scope

The scope of this research is to explore different ways by which developers can change their source code in order to make more-optimised applications that firstly have a shorter execution time but also achieve the same functionality. This research will be followed by exploiting the same principles and seeing whether we can optimise applications to require less energy. Furthermore, it is hoped that this research will lead to the development of a toolkit that uses elements of artificial intelligence methods, such as Genetic Algorithms and Local Search, incorporating a developer’s source code and well-written unit tests to create a more-optimised version of the original source code.

### Users

The users of this research would be Android application developers and Android application users. Many Android mobile users wish to have a device which does not ‘die' as often and therefore users are likely to prefer applications that do not ‘drain’ the battery rapidly. This research can also apply to the work of Java developers more generally as this research should also be applicable to larger applications and devices such as a computer where the lower energy consumption will have a greater benefit for the environment.

### Objectives

The objective for the project is:

1. To research the different ways we can optimise Java, specifically looking at changing operators in the code to see its effect What kind of changes can be made to Java code in general?
2. Applying the research from objective 1 to create some software in a similar vein to the Gin code framework [1] [2] that uses genetic algorithms to change the source code, so it is less execution and energy intensive.
3. To learn about and use the existing Opacitor [3] [4] and Jalen tools [5] to approximate the energy consumption of the changes Java code and prove that the experiments have been successful.
4. Applying GI techniques to software so it has a shorter execution time.
5. Applying GI techniques to software so it requires less energy to run.

## Achievements

Throughout the dissertation, I have learnt and developed a greater understanding of genetic improvement with a particular focus on genetic algorithms. There has also been an increase in knowledge about the subtle ways that developers can optimise their Java code and what changes can be made to source code and their effects on energy measurement.

Based on this knowledge, this lead to a creation of the ‘Anna-Gin’ toolkit. This toolkit is created in Java and Spring with the purpose of to take a source code file and corresponding unit test file and find out that file’s energy usage to run. The toolkit uses a Genetic Algorithm to work out the optimum ways to optimise these source files using measure of fitness of a combination of number of passed unit tests and energy usage.

After researching related work and background as well as the creation of the toolkit led to further experimentation. Such experimentation includes an evaluation of the relationship between recursive loops and energy in comparison to the relationship between iterative loops and energy. The toolkit influenced research in using different programs or use cases and the different optimisation techniques effects on different use cases.

## Overview of Dissertation

We present the related work and background research was used as a starting point for this dissertation, the results of which are seen in *Chapter 2* and *Chapter 3*. A high-level overview of the creation of the Anna-Gin is given in *Chapter 4*, this chapter includes details of the implementation, code examples and UML diagrams. The knowledge gathered from the work in *Chapters 2, 3* and *4* have led to *Chapter 5* which uses this knowledge to conduct experiments on other possibilities for optimising code for energy minimisation, these experiments include the relationship between types of loops and energy consumption. *Chapter 6* examines the programs and use cases that Anna-Gin toolkit has been used to optimise, it focus on the program and their corresponding unit tests. *Chapter 7* presents the results of the Anna-Gin toolkit and the effects it has on the different programs. Finally, in *Chapter 8* we conclude with our final remarks and evaluation on the project as well as some interesting future works.

As non-essential reading, we include *Appendix 1* which gives … In *Appendix 2*, we provide further details regarding how to user the Anna-Gin toolkit so further use could potentially come from the work conducted here. *Appendix 3* offers information on how to install the toolkit onto local machines.

# State-of-The-Art

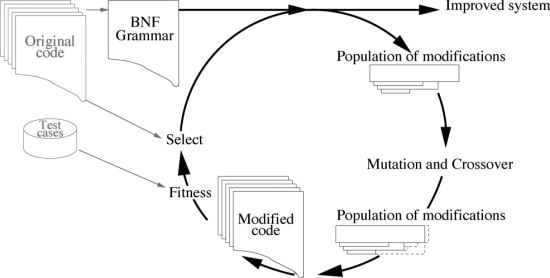
## Genetic Improvement and Changing Software

### Genetic Improvement

Genetic Improvement is used to improve pre-existing software by using an automated search to find improved software. These improvements include improvements in energy consumption, reductions in execution time and reductions in memory consumption. In a survey of 3132 distinct papers that were published between 1995 and 2015 [6], the most popular search technique is evolutionary computing as 96% of these papers referenced or used these techniques, therefore demonstrating the increase in popularity of evolutionary computing.

This survey is very relevant to this study as it is one of the largest and most comprehensive surveys on computer science papers. The authors concluded that there were 40 ‘distinct papers on GI found’ [6].

However, to cover a wide range of topics that fall within the definition of genetic improvement, the number of papers analysed in the survey had to be restricted. Hence, owing to the survey's strict definition, this may have led to the exclusion of many papers which may have had key information about Genetic Improvement.

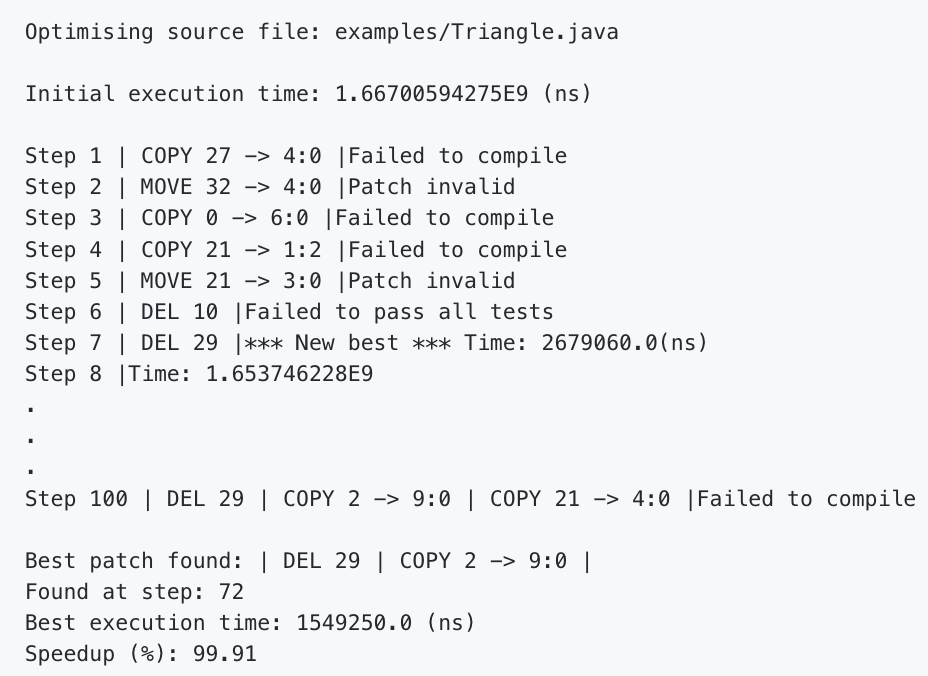


**Figure 2.1** Diagram of changing code lifecycle with Genetic Improvement

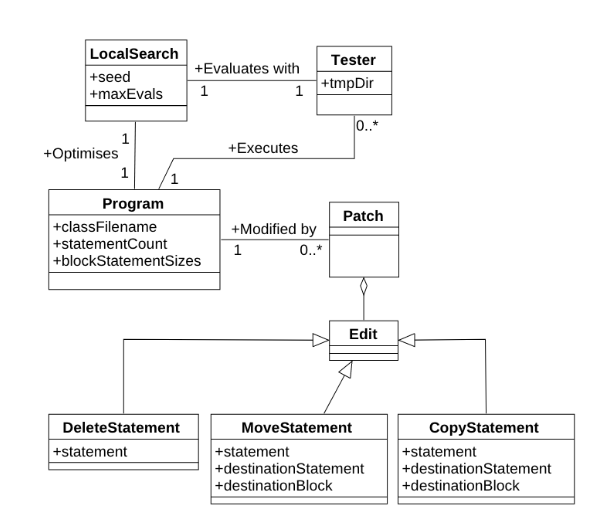
A lifecycle of improving software using genetic improvement can be seen in **Figure 2**. Langdon and Harman [7] [8] [9] state that the ‘original code is a state oracle’, therefore giving a programmer a benchmark for comparison and provide reassurance that it delivers the expected output. Hence, it may be noticed in **Figure 2** that the original code is included in the Select step after the Fitness assessment of the modified code. Genetic Improvement tries to keep the same functionality as the original code to improve the outputs generated as well as optimising the code. However, Langdon and Harman’s work [9] [7] assumes that the original code is correct and is passing unit tests. Therefore, this ignores integration testing and only focuses on genetic improvement for individual sections of code rather than considering a complete system.

### GIN

GIN, Genetic Improvement in no time, is a Java tool that makes it easier to implement Genetic Improvement and has a goal to “to stimulate development in GI tooling” [2] [1]. The Gin application itself optimises a Java class, using a local search to add “Edits” to the source code until a more-optimised version of the code is found. As seen in Figure 2 and 3, GIN manipulates code in three principal ways: deletion, copying and replacement. For instance, particular lines of code or blocks of code could be deleted such as in the ‘Triangle.java’, where the removal of the reference to the ‘delay()’ method creates a version of the code that passes all the unit tests and has a lower execution time. The possible changes are Line Edits, Statement Edits and Constrained Statement Edits. The inspirations for the ‘Delete, Replace, Copy, Swap’ of line edits came from research conducted by Petke and Langdon [8] with their GISMOE tool [9] [7] [1] . The Statement edits changes of ‘Delete, Replace, Copy, Swap’ come from the GenProg [10] automated program repair tool. All Constrained statement edits are Gin’s own creation which presents the innovation of the Gin tool and its exploration of [8] the optimisation field. An example of the Gin code output is seen below.



**Figure 2.2** Example of GIN output



**Figure 2.3** UML diagram of how GIN works

The GIN toolkit is experimental and allows for the development of skills and knowledge about optimised code and GI techniques that could later be integrated into the JVM and compiler.

The GIN toolkit is one of the very few tools looking at integrating GI within a Java project. It applies a variety of edits to source code and concludes in the most optimised version of the source code. The toolkit creates optimised source code with shorter execution time. Its use of edits may seem to be ‘small changes' to the developer but are shown to have a large impact on execution time and optimisation.

The GIN toolkit uses a local search to make decisions on how to apply edits to the source code. The use of a local search affords the ability to find reasonable solutions in continuous state space. In the gin example, there are many neighbouring solutions made quickly that lead to an optimised version of the example source code “Triangle”. Figure 4 presents a UML diagram of the different class that make GIN work where a key component is the Local Search Class. The Local Search class is used to evaluate the tests and to optimise the program within the maximum number of maxEvals.

The fitness function of the GIN toolkit is based on whether the Junit tests are successful as it is a crucial criterion that the edited source code must have the same purpose as the original source code so that the unit tests are still passed. However, this fittest function implies that the developer has created well-written unit tests and source code. Therefore, finding an optimised version depends largely on the developers' initial code.

The GIN toolkit is limited to only optimise Java code. One can imagine many of the principles that are learnt from this software could be applied to other object-oriented languages such as C. Some of the features, such as moving edits, can have a major impact on interpreted languages, such as python, which have every line of the script interpreted individually and where the position of lines are important when optimising source code.

### GI In other Languages

A large proportion of research about Genetic Improvement focuses on object-oriented languages such as Java, C and C++, therefore ignoring dynamically typed languages such as Python. Research conducted by Haraldsson [11] focuses on genetic improvement in Python who showed that it can be used to fix multiple bugs within the Python script at the same time and in a reasonable time frame.

Haraldsson's research works in a similar way to GIN where a series of edits are applied to source code. This research applies experiments to small python scripts to see whether solutions created by genetic improvement have been successful. A random walk analysis is used, where the start state is assuming that the program is correct, and the fitness is monitored as the edit list increases in size one edit at a time. The other stage of the experiment is the exhaustive neighbourhood analysis where the fitness is evaluated for every mutant of the program.

Genetic Improvement in Python has the advantage that the search space is less restricted than the search space for Java programs. This allows for more changes to the source code in a Python Script in comparison to small Java programs. Optimising Python is important because Python is ‘becoming the world’s most popular coding language’ [12] and therefore the increasing number of python developers can code an optimised version of the language with bugs fixed quickly in a reasonable time.

The larger search space of optimising Python requires a longer time to achieve results due to fewer restrictions. The results produced are also difficult to verify. For instance, it can be difficult to prove that the optimised script has the same functionality as the original script as there are no unit tests which are used a measure of fitness in Java and OOP languages. This research in Python programs is instructive as many of the concepts about genetic improvement can be used in my research but Haraldsson's research does not fully apply as it focuses on Python rather than Java.

## Tools for energy measurement

### Opacitor

Opacitor [13] [14] is a tool which measures the energy consumption of JVM programs. It does this by using a bytecode level model of energy cost; counting the bytecodes executed by a running program and matching these to previously measured quantities of energy consumption. Opacitor ‘can detect small changes in execution profile, down to opcode level’ [14] therefore it records the energy used by a program down to its lowest level. The Opacitor measures energy usage in joules and, unlike many similar tools, does not use a CPU as a proxy for energy use. Therefore, using joules and bytecode increases the accuracy of the readings.

Opacitor [3] [4] is one of the first tools of its kind to measure the energy produced by an executing program using Search-Based Software Engineering. Due to the implementation of the Opacitor it can detect the smallest changes in execution, thereby recording all the possible energy usage from the entire execution of the program. The Opacitor is unaffected by the rest of the computational environment so that when it examines the smallest changes these are the only changes in the program and does not include any other environmental factors.

The Opacitor is deterministic, i.e. it will produce the same output from a given input. The Opacitor’s results have helped reduce a program’s energy usage by up to 70% in the best case and 20% in the worst case [3]. The Opacitor is one of the few tools that can detect small changes in the execution and is responsive to metaheuristic searches [3].

The Opacitor's only focus is on the CPU and it does not consider other features of a mobile phone that use a large amount of battery power such as the display, WI-FI and GPS. The Opacitor is exclusive to applications and the CPU does not consider other features or how they work with applications and the effect that they may have on the battery.

### Jalen

Jalen is a ‘runtime measurement software for estimating the energy consumption at code level for Java applications’ [5]. Jalen uses information provided through application-level monitoring tools to calculate the energy used by an application. The energy information is estimations from the levels of threads and methods. The information collected is method durations, CPU time and so on. The information gathered is used to create statistics which is used to calculate the energy consumption as well as using energy models seen in Figure 5. The energy consumption is calculated per method and exposed as a service. Jalen can provide information about Net Energy, Net Library energy and all energy usage which includes energy consumption of all methods running in the JVM.



**Figure 2.4** Architecture of Jalen

Jalen offers a valuable tool that provides an energy count on a code level so that there is an increased understanding at the energy consumption and distribution in software. Jalen is also one of the very few toolkits that detect energy hotspots in code and knows what particular areas of software energy is intensive. Jalen has been tested with experiments on a range of different programs from simple algorithms to real-world scenarios demonstrating its versatility and the potential of energy code profiling. The information gathered from Jalen will help developers find out which methods are energy intensive, so they can change the ways they code to ‘reduce the energy footprint of their application' [5].

Unlike Opacitor, Jalen prefers statistical sampling over bytecode instrumentation. It also presents data as percentages rather than raw Joules values. This is because different hardware changes the determined results which are not a problem that Opacitor finds. Offering the results as a percentage means that results do not change drastically when the hardware changes as they do when the data are presented as a Joules count.

### Optimization Framework for Mobile Applications.

Li et al created a framework which looks at Source-level energy optimization with a particular mobile application [15]. This framework proposed an energy-optimization framework with a source code energy model that allows developers to know about energy usage. This framework changes and improves the code to save CPU energy by analysing energy features of the source code before optimisation. The model maps the energy use to the basic operation within the source code which works in a similar way to Jalen as it focuses on the energy impact on individual operations within the source code.

This framework was a success as it was able to save between 6.4% to 50.2% of the CPU energy in various mobile applications. These toolkits and frameworks highlight to developers which operators are energy intensive, so they may make better choices when developing an application. This framework focuses on mobile applications which are different to Jalen, the significance of which is that the way mobile applications interact with mobile hardware is very different from the way software interacts with other machines.

Unlike the Opacitor, this framework does not look at the bytecode level and therefore its results may be affected by features of a mobile device such as WI-FI and GPS. These features would have a major impact on the results because they are energy intensive and will increase energy usage results markedly. The reliance of highlighting to a developer their energy usage means that for there to be any saving in energy, this is entirely the developer's responsibility and then trusting them to be energy conscious.

### Code Refactoring’s effects on Energy Usage

Research conducted by Sahin et al [16] looked at energy impacts of 197 applications when 6 commonly-used refactoring techniques have been applied to those applications and the change in energy usage those applications have after those techniques applied. The results of this research concluded that there was an impact on energy usage and that a change in refactoring techniques can effectively increase or decrease the amount of energy used by an application. This study conducted 394 experiments on those 197 applications and found that 28% (109 experiments) indicated ‘a statistically significant difference in energy usage between the original and refactored versions.’ [16]. The refactoring technique of ‘Convert Local Variable to Field’ [16] had the largest positive impact on energy out of the six different techniques. This study is notable in that it presents the effects of different refactoring techniques and has a difference in results. However, this study could have been extended to include more applications and refactoring techniques as it makes it harder to generalise. This study uses only two JVMs: 6 and 7, which does not consider whether new versions of Java have integrated optimisation techniques that affect energy usage.

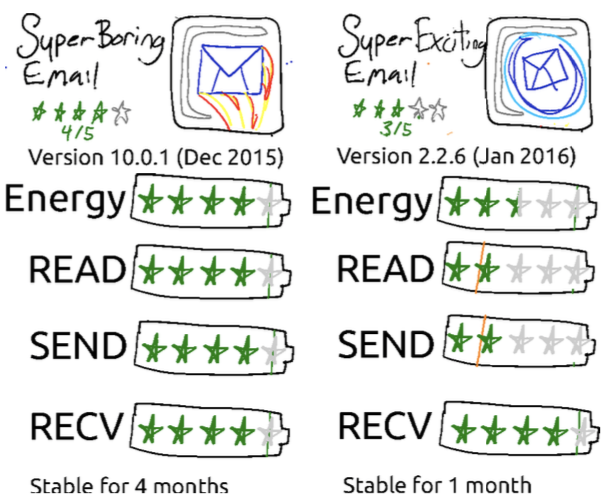
This research supported by research conducted by Morales et al [17] who focuses on the effects of antipatterns in mobile applications and its effect of energy efficiency. This research found that energy consumption of apps with anti-patterns are not statistically different to apps which do not have these anti-patterns. The research did conclude that refactoring anti-patterns can be positive or negative therefore developers need to consider the energy impact of refactoring when creating applications. The research was conducted with 59 android applications and showed that applications containing anti-patterns consume more energy than those which do not have them, therefore refactoring and removing these antipatterns has a major effect on energy usage. Like the Sahin's study, this study does not use many programs and this, therefore, makes it harder to generalise. This study is very focused on Android applications and considers elements of applications that only affect Android.

## Green Software Engineering

Green Software engineering, a term coined by Hindle [18] is a type of software engineering which is more environmentally sustainable, so it takes into consideration the power usage of creating software. There are very few uses of physical measurements in software engineering even though this is critical for measuring energy. This lack of physical measurement creates limitations regarding experimentation and analysing energy usage.

Green Software Engineering is an important yet emerging field. Because this field is at an early stage of development, it has been marked by a low level of resources, a small coherent community and an uncertain methodology. This field needs an increase in support and research which can pose a negative image of making the field an ‘impossible bar' [18]that may not be achieved. An increase in support and research may come with the on climate change [19]. This issue has only amplified the importance and necessity of this research if the desire to keep our current technology-heavy lifestyles as well as to ‘limit climate change'.

Research conducted by Hindle as it focuses on software’s interaction in other contexts other than mobile applications. Hindle researches in mobile applications, Data Centres and embedded sensors. Hindle has a proposal to have an energy rating on mobile applications included with the mobile application other functions; an example of this can be seen in *Figure 2.5*. Therefore, the user has the responsibility of choosing a more sustainable application which offers an interesting proposal as it gives the user more responsibility and can be used an extra selling point for a mobile application.



**Figure 2.5** Ratings for Mobile applications

All the previous research mentioned is useful regarding green software engineering. As the combination of changing source code using Genetic Improvement to reduce energy consumption is the focus and aims of this project.

# Background

A large proportion of this research is building on research previously conducted by Dr Alexander Brownlee [3] and Dr David White [1]which focuses on methods used to optimise source code without losing any functionality through Genetic Improvement. This research also builds upon the research by Bruce et al [20], which considered methods to automatically change source code to reduce energy usage.

## Optimisation regarding Java

Optimisation is ‘the act of making the best or most effective use of a situation or resource'. Optimisation in Java focuses on goals within or related to the source code that we can optimise [21]. This research focuses on changing Java code and Java Projects that can be used to improve an application’s execution time and reduce an application’s requirement for the energy it needs to run.

To find solutions to optimisation problems, the technique chosen for the current work is based on Genetic Algorithms seen in *Section 3.3*, which find solutions using the notion of fitness seen in *Section 3.4*.

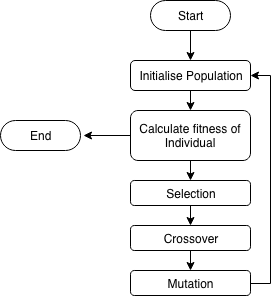
## Genetic Improvement

Genetic Improvement (GI) improves software by using machine learning and optimisation techniques. These techniques include Evolutionary Algorithms of which Genetic Algorithms are a sub-category. Genetic Improvement can be used to improve pre-existing software with improvements such as reducing energy consumption, execution time and memory consumption.

## Genetic Algorithms

Genetic Algorithms (GA) are algorithms based on the biological notion of evolution and natural selection as conceived by Charles Darwin. Genetic algorithms use the principle of survival of the fittest and inheritance. The fittest traits of the parent are passed onto the children who prevail. Throughout the duration of the algorithm, many possible solutions to a problem are found; these solutions are called ‘individuals. Every iteration of the algorithm selects a solution with high fitness from a collection of individuals. Subsequent iterations generate further solutions from these individuals. Some of these solutions are closer at achieving the most optimised solution to the problem and some solutions are not. The selection stage decides what traits are to be kept for the future generations. Traits that do not confer improved fitness are disregarded by the selection in the next iteration.

In standard Genetic Algorithms, there are five main steps that are followed; this is depicted in Figure 1. The steps are: initialising population, fitness, selection, crossover and mutation.



**Figure 3.1** Flow Diagram of a Genetic Algorithm

An “Individual” or “chromosome” is a possible solution to the problem. In a classic GA, a chromosome is a string of bits and each individual bit is a gene. The selection stage ‘selects’ individuals with high fitness that will progress to the next population. In the mutation stage, individual genes of the highly fit parent chromosome are flipped from one to zero or vice versa to create a new mutated chromosome to pass onto further populations. The crossover stage requires taking genes from the two parents and combining them to make a new individual who may be a more optimised solution to the problem than their parents.

A “Population” is a set of individuals. The initialise population stage involves gathering a collection of individuals together to form a population. As individuals increase their fitness throughout the lifetime of the GA, the overall population has a higher fitness. It therefore has a higher possibility of finding a solution to the problem.

In the problems that form the focus of this project, Genetic improvement problems, the individuals can be lists of changes in source code or new versions of the source code. In the GI tool Gin [2] [1], an individual is a list of "Edits". Each "Edit" is a single change to the source code such as moving a line from one place to another. The mutation stage adds or removes edits to source code and in the crossover stage, a combination of edits is combined together into a new version of the source code. An edit is similar to a gene but not a direct parallel because an edit is what has changed in the code. A "Patch" is similar to a chromosome but not by the traditional definition because a "Patch" may comprise of no edits if it happens that in the current version of source code in also the most optimised source code. A patch could also contain a collection of edits to create the optimised source code. In the Gin Code [1] [2], a collection of patches is like a classic Genetic algorithm’s definition of a population.

## Fitness

As mentioned previously, the fitness of an Individual is how well that particular Individual performs the task. In this instance, there are several potential measures of fitness, for example, one can ask: Does the new source code compile? Does the new source code pass the unit tests? What is the energy consumption of the application? This has meant that each element of fitness needs a weight which is how important that element is in the fitness function. For example, whether the source code can compile has a larger weight than the unit tests because if a program does not compile then the unit tests are irrelevant as the program has lost all functionality. Table 1 presents the different elements of the fitness and a way that the different weights can be used together. In the example below, Individual 3 would be the most optimised because it has passed the most weighted elements such as it can compile and energy consumption although it failed two-unit tests.

**Table 1** Example to Demonstrate the levels of Fitness. P denotes a pass, F a failure.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Did the source code compile?** | **Unit tests** | | | **Energy**  **Consumption** | **Use this**  **individual?** |
| **Unit test 1** | **Unit test 2** | **Unit test 3** |
| *Individual 1* | Yes | P | P | P | 3000J | Yes |
| *Individual 2* | No | P | P | F | N/A | No |
| *Individual 3* | Yes | P | F | F | 2100J | Yes |
| *Individual 4* | Yes | F | P | P | 4000J | Yes |

For any optimisation problem, the fitness function needs to be defined before a solution is executed. Changing the fitness function can present new ways to optimise Java. For example, by changing the fitness to focus on execution time or energy consumption, the Java application can be optimised in different ways.

In this research I decided to use a lexicographical fitness. In the research this meant that the first measure of fitness is number of unit tests passed. Once the solution has passed a high number of the unit tests, we use a second measure of fitness which is the energy measurement. The solutions are compared on which has the lowest energy and that solution is seen as the most optimised solution of that generation. This was done because I thought it was more important to prioritise the functionality of the application over the energy because an application that does not work as intended will not be used so the energy measurement would be irrelevant.

# Methodology

## Creation of Toolkit

To begin with, I created a toolkit in a similar style to the GIN toolkit [1] [2]. The toolkit I implemented had a different structure and used different edits in comparison to the GIN toolkit. This was done because I wanted to learn how GIN worked and what techniques they used that I could use or manipulate for my own toolkit. Creating my own toolkit allowed for me to explore different areas that we can optimise code and gave me freedom to try new ideas. I also personally found it easier to create my own toolkit, rather than try to add changes to the GIN tool, which is constantly being changed and refactored by multiple universities and professors with their plethora of ideas.

## Tools Used

### Opacitor

For the Opacitor, I used a jar file provided and previously implemented by Dr Alexander Brownlee. I used the bytecode histogram version of the Opacitor with the garbage collector switched off. This allowed for there to be a more accurate reading as the energy readings come straight from the code and are not affected by external JVM features such as the garbage collector. The Opacitor is written in a builder pattern which allowed to implement the complex object ‘Opacitor’ easily and could be easily integrated into the toolkit and edited so it fit the requirements I needed.

### Test-Runner

For the test runner, I used the code that was previously implemented by Dr Alexander Brownlee. This was because Dr Brownlee had spent a large amount of time creating the test runner and had found some difficulties when implementing it. By using this test runner, it allowed me to focus and spend the time on implementing the toolkit, conducting experiments and obtaining results. I did conduct small changes to the test runner, in order that it be integrated with my toolkit. For instance, changing the ways they find the files by using ‘AnnaPaths’.

### Fitness

I used the ‘death penalty function’, which is when I set the fitness score to be an unrealistic score, which is ‘1234.0’ when the unit tests are failed. This gives that patch an unreasonable fitness score and since the Genetic Algorithm would want a smaller fitness, they would avoid this patch.

### Unit tests

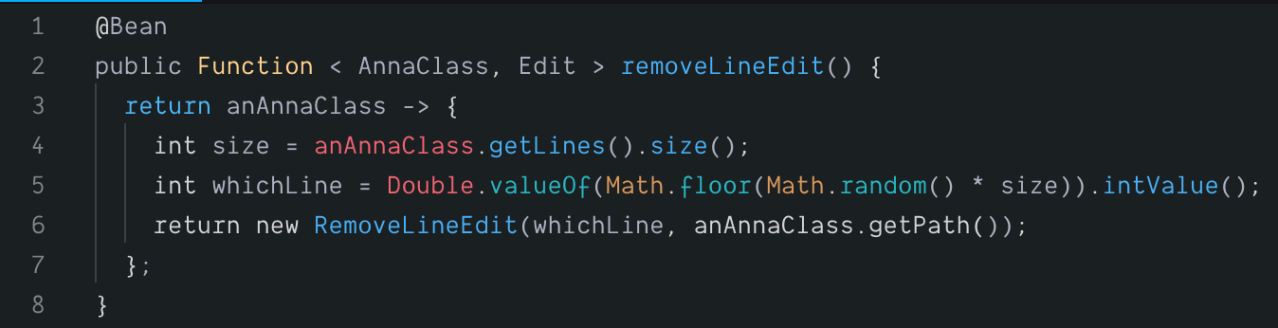
For the toolkit, in order to get an accurate measure of fitness, each project that is being optimised needs well written unit tests. For the toolkit, the unit tests were written using Junit. The toolkit relies on the idea that the developer has written unit tests which can lead to questions on how we know whether a unit test is ‘well-written’ or not. This may have an effect on the toolkit as we are using it as a measure of whether to program is functional or not. If it passes badly written tests, then the toolkit will not be able to effectively optimise the software

### Maven

Maven [22] is a ‘build automation tool’ with a focus for java projects. I used Maven to bring in java libraries and dependencies. Maven also allowed me to structure the project in a way that allowed for different sections of the project to be its own module and I could bring in that module as a dependency and use it as if it were a library. For instance, I had the Opacitor as one module and in the main module (were the main processes begin) would bring in the Opacitor as a dependency. Maven is advantageous because it is standardised and with the maven plugin it is easier to use. The reason I used Maven is because I find it easier to use than other dependency management frameworks such as gradle, this is the xml formatting of maven is easier than gradle’s groovy scripts and it is easier to configure to the project because of its standardisation. Maven is useful as it allows to structure the project, for instance each module has its own ‘pom.xml’ with their dependencies as well as setting which version of Java each module has, for instance the Opacitor needed to be Java 7 or less to run which is different from the main module which I set to be Java 8 so maven allowed different modules to have certain dependencies and Java version.

### Spring

Throughout development of the Anna Gin Toolkit, the use of the spring framework allowed for an ease of inserting particular features into the application. The spring framework is an ‘application framework and inversion of control container for the Java platform’ [23].The use of spring framework was decided because of previous experience with the framework and a personal preference of wishing to avoid direct association with reflection. From a personal experience, the spring framework allowed for easier insertion of different types of edit, an example of an edit bean is seen in Figure 5.1. As you can see in figure 5.1, there is a @Bean annotation to denote the edit and the use of a lambda and function to create the edit. In the project, I created a bean for each of the different types of edit, this allowed for the ApplicationContext to bring all the beans in when ran.



**Figure 4.1** Example of a Bean for a removeLineEdit

This gave for increased ease as if a developer wished to add another type of edit, they would just need to create a new bean in ‘EditConfig’.java file for that edit to be brought into the program.

## Development of Toolkit

The different steps of developing the AnnaGin Toolkit are:

1. Gin-Main
2. Test-Runner
3. Opacitor
4. Genetic Algorithm
5. Hill-Climbing
6. Testing

### Gin-Main

#### Patch

Similarly, to the GIN toolkit, my version of a patch is the code I wish to optimise with the edits applied to it. For example, the original patch such as the ‘Triangle’ program will be 43 lines in length long; I could apply a series of edits to Triangle. Currently there is one edit applied to a patch at any one time with children patches having the possibility for multiple edits. Throughout the creation of the toolkit, I used a seeded random number generator so results from the genetic algorithm can remain consistent

#### Edits

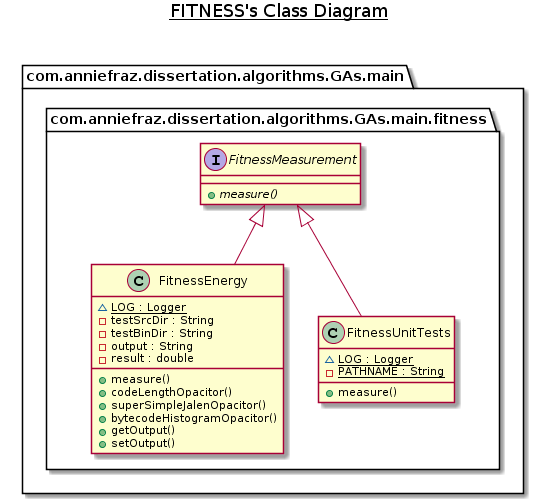
Similarly, to the GIN toolkit, my version of an edit is still a change to a computer program. I implemented 10 different types of edits. There are 3 block type edits which can remove, move and swap blocks of code. There are 6 types of line edits, Insert, Moving, removing, swapping lines as well as insert return and break statements. The final type is operators is changing if statements, so the if statements conditions effects are swapped around to see their details. I used a factory design pattern to implement the edits. I used this design pattern as I found it an easier way to construct different edits as it allows for using the same instance multiple times so an edit object did not need to be newly created each time.

### Genetic Algorithm

#### Initialize Population

This method generates a population of patches. The size of the population is decided randomly, this is because I wanted to experiment with different ways solutions are made and whether a bigger population offer the potential for higher probability of gaining a more optimized problem.

#### Calculate Fitness



**Figure 8** UML Diagram of the Fitness

As seen in Figure 8, fitness is implemented as an interface called ‘FitnessMeasurement’ that returns a double and takes in a patch. For every different type of fitness, there is a class. For instance, there is a class called ‘FitnessEnergy’ which returns the Opacitor measurements for that patch. The other notion of fitness, the unit tests have their class that return a double of 1.0 for all passed unit tests or a 0.0 for if the program fails them; representing the returns as double allowed for an easier calculation of fitness as they are both the same type. This was implemented this way because it led to a level of abstraction and it allows for any future developer to add their own measure of fitness with ease and so it is fully integrated.

#### Selection

This method takes in a list of patches called a population. From this list you randomly sect 2 of the individuals and work out their fitness score. Whichever is has the lowest fitness score which will be decided to be the first parent of which we will later use to produce offspring from. Then the process is done again to obtain the second parent. The two parents are added to a parent list and are then returned.

#### Crossover

In this method, I created three offspring created by three different operators. The first operator took the edits from the first parents and one of the edits from parent two selected randomly. This is conducted in a similar way for the operator two, but with the parents swapped around. The third operator was conducted by combining all the edits from both parents together to create a patch.

#### Mutation

Neighbours are created by cloning the offspring. Then the neighbour is altered to be subtly different from its parent offspring. To calculate whether to add a neighbour to the next population, we work out its random value and then add a random set of edits to that neighbour before being added to the population.

### Hill - Climbing

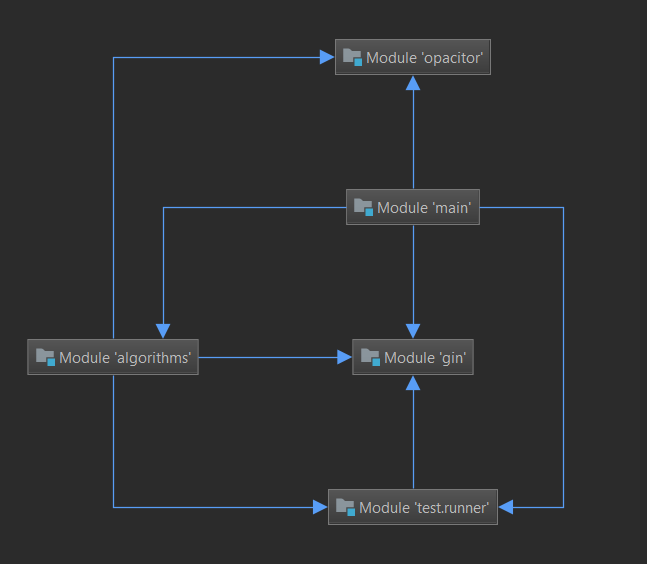
### Testing

#### Test Driven Development

Regarding testing of the project, most of the unit tests were created using Test Driven Development. This is where the developer does not “write a line of new code unless you first have a failing automated test” [24], so I wrote unit tests before implementing the code. This was especially useful when it came to implementing edits as it allowed me to organise in my mind what the edit’s purpose was and think of ways I could implement this. Throughout the project, having pre-existing tests notified of bugs and their whereabouts making it easier to debug and fix.

## Structure

### Modules



**Figure 9** Diagram Demonstrating the interconnecting modules of the Project. Generated by Intellij Diagrams

The structure of the project consisted of several modules which each perform their own section of the task. For example, test-runner had its own module that is separate from the gin-main module which has edit files for example. *Figure 8* shows how the different modules are connected to each other, the elements of one module were brought into another module using maven as a module could be a dependency for another. This allowed for each module to have their own version of Java such as Opacitor was written in Java 7, the module format allowed for this to continue whilst I could use Java 8 and its features like Lambdas when implementing gin-main.

### Overview

INSERT UML DIAGRAM OF THE TOOLKIT HERE

# Experiments

## Recursive vs Iterative Loops

### Introduction

After learning about the Opacitor and thinking of different ways that code can be changed for energy optimisation and a suggestion from another student, I thought of the effects of different loop types on energy. I specifically focused on recursive loop in comparison to iterative loops and whether calling a method repeatedly in a loop influences energy.

I hypothesized that recursive loops take more energy to conduct the same functionality as an iterative loop. I do not expect the energy measurements for recursive loops to be a major increase in comparison to an iterative loop.

I hypothesized that the larger the array, the longer it will take for the sorting algorithm to complete its task regardless of whether it uses an iterative or recursive loop. I do not believe that the size of the values within the array should have a dramatic effect on the energy measurements but potentially some small changes. I expect that the order of the array will have a small effect on energy, so whether this array is in a random, ascending or descending order should influence energy. For example, ascending is the best case so in theory would need less time and energy and the opposite could be said for descending as it is the worst case.

### Method

In this series of experiments, I created an array of a certain length with a certain seed. I created in total 48 (roughly) arrays, with seeds ranging from 10 to 10,000 and array sizes ranging from 10 to 10,000. The 48 arrays are split into 3 subgroups: 16 totally random arrays, 16 ascending arrays and 16 descending arrays. I did conduct some experiments with an array length of 100,000 but conducting these experiments took a long time for up to 4 hours. For each individual array, I ran the array through the complication unit and the Opacitor Bytecode histogram thirty times. Repeating the array this way helped reduce noise in the experimentation.

To get results with as little bias as possible, I did not change the code, with edits mentioned in the Anna-Gin toolkit. Instead I used elements that I learnt from the development of that toolkit to run these tests such as using threads to run multiple tests at once. This multithreading led me to stop keeping records of the time it took the tests to run as I believed that threading might influence time.

### Results

#### Arrays in Random Order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Seed** | **Array Length** | **Iterative Loops** | | **Recursive** | |
| **Average Opacitor Measurement 1 (Joules)** | **Average Opacitor Measurement 2 (Joules)** | **Average Opacitor Measurement 1 (Joules)** | **Average Opacitor Measurement 2 (Joules)** |
| 10 | 10 | 0.618 | 0.618 | 0.687 | 0.687 |
| 10 | 100 | 0.851 | 0.851 | 7.982 | 7.982 |
| 10 | 1000 | 18.781 | 18.781 | 4164.760 | 4164.760 |
| 10 | 10000 | 1744.032 | 1744.032 |  | [[1]](#footnote-1) |
| 100 | 10 | 0.621 | 0.621 | 0.697 | 0.697 |
| 100 | 100 | 0.886 | 0.886 | 8.745 | 8.745 |
| 100 | 1000 | 19.576 | 19.576 | 4236.951 | 4236.951 |
| 100 | 10000 | 1807.108 | 1807.108 |  |  |
| 1000 | 10 | 0.622 | 0.622 | 0.706 | 0.706 |
| 1000 | 100 | 0.904 | 0.904 | 9.490 | 9.490 |
| 1000 | 1000 | 19.790 | 19.790 | 4315.049 | 4315.049 |
| 1000 | 10000 | 1819.797 | 1819.797 |  |  |
| 10000 | 10 | 0.625 | 0.625 | 0.714 | 0.714 |
| 10000 | 100 | 0.935 | 0.935 | 10.283 | 10.283 |
| 10000 | 1000 | 20.191 | 20.191 | 4396.000 | 4396.000 |
| 10000 | 1000 | 1817.000 | 1817.000 |  |  |

**Table 2** Results[[2]](#footnote-2) of Bubble Sort tests with an Array in Random Order

**Figure 5.1** Graph for relationship between arrays of random order and energy measurement

#### Arrays in Ascending Order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Seed** | **Array Length** | **Iterative Loops** | | **Recursive** | |
| **Average Opacitor Measurement 1 (Joules)** | **Average Opacitor Measurement 2 (Joules)** | **Average Opacitor Measurement 1 (Joules)** | **Average Opacitor Measurement 2 (Joules)** |
| 10 | 10 | 0.618 | 0.618 | 0.688 | 0.687 |
| 10 | 100 | 0.801 | 0.801 | 7.932 | 7.932 |
| 10 | 1000 | 12.954 | 12.954 | 4158.933 | 4158.933 |
| 10 | 10000 | 1167.235 | 1167.235 |  |  |
| 100 | 10 | 0.620 | 0.620 | 0.697 | 0.697 |
| 100 | 100 | 0.823 | 0.823 | 8.681 | 8.681 |
| 100 | 1000 | 13.168 | 13.168 | 4230.543 | 4230.543 |
| 100 | 10000 | 1169.390 | 1169.390 |  |  |
| 1000 | 10 | 0.623 | 0.623 | 0.705 | 0.705 |
| 1000 | 100 | 0.844 | 0.844 | 9.431 | 9.431 |
| 1000 | 1000 | 13.402 | 13.402 | 4308.661 | 4308.661 |
| 1000 | 10000 | 1171.745 | 1171.745 |  |  |
| 10000 | 10 | 0.624 | 0.624 | 0.713 | 0.713 |
| 10000 | 100 | 0.868 | 0.868 | 10.216 | 10.216 |
| 10000 | 1000 | 13.643 | 13.643 | 4389.452 | 4389.452 |
| 10000 | 10000 | 1174.123 | 1174.123 |  |  |

**Table 3** Results**[[3]](#footnote-3)** of Bubble Sort test with an Array in ascending order

**Figure 5.2** Graph for relationship between arrays in ascending order and energy measurement

#### Arrays in Descending Order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Seed** | **Array Length** | **Iterative Loops** | | **Recursive** | |
| **Average Opacitor Measurement 1 (Joules)** | **Average Opacitor Measurement 2 (Joules)** | **Average Opacitor Measurement 1 (Joules)** | **Average Opacitor Measurement 2 (Joules)** |
| 10 | 10 | 0.618 | 0.618 | 0.689 | 0.689 |
| 10 | 100 | 0.914 | 0.914 | 8.047 | 8.047 |
| 10 | 1000 | 24.532 | 24.532 | 4170.511 | 4170.511 |
| 10 | 10000 | 2325.573 | 2325.573 |  |  |
| 100 | 10 | 0.621 | 0.621 | 0.698 | 0.698 |
| 100 | 100 | 0.949 | 0.949 | 8.808 | 8.808 |
| 100 | 1000 | 25.900 | 25.900 | 4243.274 | 4243.274 |
| 100 | 10000 | 2443.599 | 2443.599 |  |  |
| 1000 | 10 | 0.624 | 0.624 | 0.707 | 0.707 |
| 1000 | 100 | 0.972 | 0.972 | 9.558 | 9.558 |
| 1000 | 1000 | 26.247 | 26.247 | 4321.507 | 4321.507 |
| 1000 | 10000 | 2457.510 | 2457.510 |  |  |
| 10000 | 10 | 0.626 | 0.626 | 0.715 | 0.715 |
| 10000 | 100 | 0.996 | 0.996 | 10.343 | 10.343 |
| 10000 | 1000 | 26.501 | 26.501 | 4402.310 | 4402.310 |
| 10000 | 1000 | 2461.045 | 2461.045 |  |  |

**Table 5** Results**[[4]](#footnote-4)** of Bubble Sort test with an Array in descending order

**Figure 5.3** Graph for relationship between array in descending order and energy measurement

### Summary

As seen in figures Tables 2, 3 and 4, it is conclusively decided that no matter the size or seed of the array, Iterative loops in an algorithm are consistently using less energy to sort an array than an algorithm that uses a recursive array.

As expected, for both types of loops, the larger the array the more energy needed to sort the array. However, what I did not expect was how large the difference would be between recursive and iterative, for instance with a random array, there is a different (consistently) of over 4,000 Joules, no matter the size of the array’s seed. This is in comparison to iterative loops which consistently remained under 2500.00 joules for all arrays even regarding tests that used larger arrays than the ones used for recursive.

When using an array of size 10 with a seed of 10 in a random order, the iterative loops used 0.618 Joules to sort the array whilst recursive used 0.687 Joules. This is a difference of 0.069 Joules on average. When using an array of size 1000 with a seed of 10 in a random order, the iterative loops used 20.191 Joules to sort the array whilst recursive used 4395.999 Joules. This is a difference of 4375.808 Joules on average.

When using an array of size 10 with a seed of 10 in an ascending order, the iterative loops used 0.617 Joules to sort the array whilst recursive used 0.687 Joules. This is a difference of 0.0699 Joules on average. When using an array of size 100 with a seed of 10 in an ascending order, the iterative loops used 18.781 Joules to sort the array whilst recursive used 4164.759 Joules. This is a difference of 4145.979 Joules on average.

When using an array of size 10 with a seed of 10 in a descending order, the iterative loops used 0.618 Joules to sort the array whilst recursive used 0.689 Joules. This is a difference of 0.071 Joules on average. When using an array of size 100 with a seed of 10 in a descending order, the iterative loops used 24.532 Joules to sort the array whilst recursive used 4170.511 Joules. This is a difference of 4145.979 Joules on average.

There is a difference between whether the arrays are in random, ascending and descending order. There is a small decrease in Opacitor energy measurement if the array is in ascending order for both types of loops in comparison to array in random order and an array in descending array. If the array is in a descending order, there tends to be a smaller increase in energy measurement regardless of whether the algorithm used a recursive or iterative loop or not.

Unfortunately, due to limitations in time and computer power, it was not possible to perform any form of arrays of a size of 10,000 when the algorithm uses a recursive loop. I was able to perform an array of 5,000 to get an idea of what could be but there are not large amounts of this data. For similar reasons, I struggled to perform the algorithms with an array size of 100,000 although I have a few results for 100,000 for an iterative loop but this took a very long time to acquire.

#### *Further Work*

The research gathered in this experiment could be conducted again with a different sort algorithm such as an Insertion Sorts or a Merge sort. This would lead to some potentially interesting results in looking at difference in energy for different sorts. Some sorts may be requiring more energy to sort the algorithm. This could lead to interesting questions about whether an algorithm’s Big O has an impact on energy so for instance does if an algorithm such as Merge sort (which a faster algorithm is than a Bubble Sort), has a smaller energy measurement than slower algorithms like Bubble Sorts.

The experiments could be conducted again with arrays in a variety of different order. For example, it could be possible to have the arrays in a ‘Pipe organ’ format. This is where an array starts in an ascending order till the middle of the array then the rest of the array is descending therefore the highest value of the array is in the centre of the array. I would expect that this would have a small effect on the energy usage as it is not as straightforward solution as just ascending or just descending, as a pipe organ formation is a combination of the best and worst case in an array for a sorting algorithm.

This research can also be implemented into the AnnaGin toolkit. This could be done as implementing an Edit that changes a recursive loop to an iterative loop because largely iterative loops have lower energy measurements so potentially changing the loops can reduce the programs overall energy usage. This could lead to interesting problems because recursive loops are normally implemented for certain reasons so there needs to be verification that the program’s functionality is not compromised by making the program not recursive, this can be conducted by implementing good unit tests that check for compromises in functionality.

# Use Cases

## JCodec

JCodec is ‘a pure implementation of video and audio codecs’ [25]. JCodec is currently being used by the GIN toolkit as a set of programs to improve. There are 203 different programs within JCodec. JCodec is a good library for GIN to optimise because they are written in Java and they contain more than 10-unit tests per program.

The high number of unit tests allow for the functionality not to be compromised significantly so that the programs still perform in the same way after optimisation using GIN. The JCodec implementation is used in many programs and large numbers of developers. This has created much documentation and if GIN finds a way of improving the library, there can be large saving in energy if all those programs use the optimised versions.

JCodec presents its own problems when optimising because many programs use media elements which require much energy to work and cannot have too much content removed otherwise the functionality of the program is adversely affected, causing a wide range of errors. This library of GIN applications to JCodec applications on this library does not work on other multimedia techniques in Java such as the Oracle multimedia API [26] as they work in different ways in comparison to JCodec.

## Triangle Program

Initially, I used a basic ‘Triangle program’ to optimise.

# Results

# Conclusion

## Summary

## Evaluation

## Future Work

Future work can include adapting the toolkit to analyse mobile applications with a focus on the Android Framework such as the Android Pie 9 Operating System [27]. This offers the opportunity to reprogram Java mobile applications, so they require less battery of the mobile device to achieve the same functionality. Implementing this feature would offer an interesting set of problems for instance, most android applications have a ‘AndroidManifest.xml’ file which is ‘describes essential information about your app to the Android build tools, the Android operating system’ [28]. This android manifest file contains many of the application’s components so if the toolkit removes these components, the toolkit needs to have the ability to change this android manifest file, so it adapts to the modified application.

The current toolkit only concentrates on Java programs. It is thought that a lot of the research done about the changing of code and Genetic Improvement could be applied to other languages to optimise them in a similar way. Other languages such as Python are gathering in popularity and the current toolkit would not be able to adapt to these applications. Adapting the toolkit would also be useful for large programs that require a lot of energy to run and optimising these large programs would save last amounts of energy. Many of such large programs use multiple languages that work in tandem, so the toolkit would need to be able to optimise multiple languages as well as ensure that the different areas of the program still work together without compromising the applications functionality.

Appendix 1

You may have one or more appendices containing detail, bulky or reference material that is relevant though supplementary to the main text: perhaps additional specifications, tables or diagrams that would distract the reader is placed in the main part of the dissertation. Make sure that you place appropriate cross-references in the main text to direct the reader to the relevant appendices.

*Note that you must* ***not*** *include your program listings as an appendix or appendices*. You should submit such material to the project *digital repository*.

Appendix 2 – User guide

If you produced software that is intended for others to use, or that others may wish to extend/improve, then it is advisable to include user guide and installation guide appendices.

Appendix 3 – Installation guide

If you produced software that is intended for others to use, or that others may wish to extend/improve, then it is advisable to include user guide and installation guide appendices.

# Glossary

GI – Genetic Improvement

# References

|  |  |
| --- | --- |
| [1] | D. White, “GI in no time,” in *GECCO '17 Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017. |
| [2] | D. White, “gintool/gin,” Github, 2018. [Online]. Available: https://github.com/gintool/gin. [Accessed 07 October 2018]. |
| [3] | N. Burles, J. Swan and A. Brownlee, “Search-Based Energy Optimization of Some Ubiquitous Algorithms,” *IEEE Transactions on Emerging Topics in Computational Intelligence,* vol. 1, no. 3, pp. 188-201, 2017. |
| [4] | S. Black, “Opacitor: Search-Based Energy Optimization of some ubiquitous algorithms,” DAASE Project Blog, 01 June 201. [Online]. Available: https://daaseblog.wordpress.com/2017/06/01/opacitor-search-based-energy-optimization-of-some-ubiquitous-algorithms-sbse/. [Accessed 07 October 2018]. |
| [5] | A. Noureddine, “Towards a Better Understanding of the Energy,” *Université des Sciences et Technologie de Lille,* 2014. |
| [6] | J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White and J. R. Woodward, “Genetic Improvement of Software:,” *IEEE Transactions on Evolutionary Computation,* vol. 22, no. 3, pp. 415 - 432, 2018. |
| [7] | J. Petke, M. Harman, W. B. Langdon and W. Weimer, Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class, Granada, Spain: Springer, 2014. |
| [8] | W. B. Langdon, B. Y. Lam, J. Petke and M. Harman, “Improving CUDA DNA analysis software with genetic programming,” in *Proceedings of the Genetoc and Evolutionary Computation Conference, GECCO 2015*, Madrid, Spain, Jully 11-15, 2015. |
| [9] | J. Petke, W. B. Langdon and M. Harman, “Applying genetic improvement to minisat,” *International Symposium on Seach Based Software Engineering,* pp. 257-262, 2013. |
| [10] | C. Le Goues, T. Nguyen, S. Forrest and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Trans. Software Eng,* vol. 38, no. 1, pp. 54-72, 2012. |
| [11] | S. Haraldsson, “Genetic Improvement of Software: From Program Landscapes to the Automatic Improvement of a Live System,” PhD, University of Stirling, 2019. |
| [12] | The Economist, “Python has brought computer programming to a vast new audience,” The Economist, 19 July 2018. [Online]. Available: https://www.economist.com/science-and-technology/2018/07/19/python-has-brought-computer-programming-to-a-vast-new-audience. [Accessed 02 January 2019]. |
| [13] | N. Burles, E. Bowles, A. Brownlee, Z. Kocsis, J. Swan and N. Veerapen, “Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava,” Search-Based Software Engineering, 2015. |
| [14] | N. Burles, E. Bowles, B. Bruce and K. Skrivisut, “Specialising Guava's Cache to Reduce Energy Consumption,” 2018. |
| [15] | J. Gallagher and X. Li, “A source-Level Energy Optimization Framework for Mobile Applications,” 2019. |
| [16] | C. Sahin, L. Pollock and J. Clause, “How Do Code Refactoring Affect Energy Usage?,” 2019. |
| [17] | R. Morales, R. Saborido, F. Khomh, G. Antoniol and F. Chicano, “Anti-patterns and the energy efficiency of Android applications,” 2019. |
| [18] | A. Hindle, “Green mining: a methodology of relating software change and configuration to power consumption,” *Empirical Software Engineering,* vol. 20, no. 2, pp. 374-409, 2013. |
| [19] | J. Watts, “We have 12 years to limit climate change catastrophe, warns UN,” The Guardian, 08 10 2018. [Online]. Available: https://www.theguardian.com/environment/2018/oct/08/global-warming-must-not-exceed-15c-warns-landmark-un-report. [Accessed 24 01 2019]. |
| [20] | B. Bruce, J. Petke and M. Harman, “Reducing Energy Consumption Using Genetic Improvement,” in *GECCO*, 2015. |
| [21] | J. Gough, B. Evans and C. Newland, Optimizing Java, Sebastopol, United States: O'Reilly Media, 2018. |
| [22] | Apache Software Foundation, “Apache Maven Project,” Apache, 29 01 2019. [Online]. Available: https://maven.apache.org/what-is-maven.html. [Accessed 30 01 2019]. |
| [23] | Spring, “Guides,” Spring, [Online]. Available: https://spring.io/guides. [Accessed 24 01 2019]. |
| [24] | K. Beck, Test-Driven Development: By Example, Addison-Wesley Professional , 2002. |
| [25] | JCodec, “Jcodec.org,” [Online]. Available: http://jcodec.org/. [Accessed 02 January 2019]. |
| [26] | Oracle, “Oracle Multimedia Documentation,” Oracle.com, 2019. [Online]. Available: https://www.oracle.com/technetwork/database/database-technologies/multimedia/documentation/index.html. [Accessed 02 January 2019]. |
| [27] | Android, “Android 9 Pie,” Android, [Online]. Available: https://developer.android.com/about/versions/pie/. [Accessed 24 01 2019]. |
| [28] | Android, “App Manifest Overview,” Android Developers, [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed 24 01 2019]. |
| [29] | B. Evans, J. Gough and C. Newland, Optimizing Java, O'Reilly Media, 2018. |
| [30] | The Apache Software Foundation, “Apache Maven Project,” Apache, 29 01 2019. [Online]. Available: https://maven.apache.org/what-is-maven.html. [Accessed 30 01 2019]. |
| [31] | Oracle, “The Java Tutorials,” Java Documentation, [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. [Accessed 24 01 2019]. |
| [32] | S. Luke, Essentials of Metaheuristics, Lulu, 2015. |
| [33] | J. Petke and W. B. Langdon, “Software is not fragile,” *First Complex Systems Digital Campus World E-Conference 2015,* pp. 203-211, 2017. |
| [34] | M. Negnevitsky, Artificial Intelligence, Harlow: Addison-Wesley/Pearson, 2011. |
| [35] | M. Mitchell, An Introduction to Genetic Algorithms, Cambridge, Mass: MIT Press, 1996. |

1. For Recursive Loops, with an array size of 10,000 in length, it took too long to compile so this data could not be obtained. [↑](#footnote-ref-1)
2. Results have been rounded up to 3 decimal places. There were some variation at 12 decimal place but this could have been from using random. [↑](#footnote-ref-2)
3. Results have been rounded up to 3 decimal places. There were some variation at twelfth decimal place but this could have been from using random. [↑](#footnote-ref-3)
4. Results have been rounded up to 3 decimal places. There were some variation at twelfth decimal place but this could have been from using random. [↑](#footnote-ref-4)