Graduate School of Natural Sciences

Computing Science

Fitness Landscape Analysis applied to functional Genetic Improvement

Author
D. van Laar

First supervisor
Dr. ir. D. Thierens

Second supervisor
Dr. S.W.B. Prasetya

November 15, 2021

Abstract

Genetic Improvement is the concept of a computer improving human-written code. This improves either the functional or the non-functional properties of the program. Genetic Improvement uses mutations to find improved versions of the original program. This makes the search space for Genetic Improvement very large. Furthermore for functional improvement, the fitness landscape forms large plateaus. In this thesis, we will attempt to analyse the search space of Genetic Improvement using Fitness Landscape Analysis techniques to achieve a better understanding of the search space.

To achieve this, we have edited the PyGGI framework to perform a random walk, and to analyse how large plateaus are. The PyGGI framework has been edited in such a way that it suits our needs and has such a performance that the experiments can be concluded in a reasonable amount of time. We perform the Genetic Improvement process on programs selected from the Bears benchmark, which contains many programs with bugs and test suites.

The results of this thesis conclude that while the plateaus are near-infinitely big, a random walk over the plateau often finds the global optimum. The only cases where the global optimum could not be found are the experiments which could not be improved with the used set of mutations. These results are in line with similar results in researches in this area.

Contents

1 Introduction and Research Question

While developing any type of software, it is always possible to run into bugs. It is very common for developers to make little errors. Most of them are noticed during development, from a peer-review, or from continuous integration. However, some bugs always manage to slip through the cracks and make their way into published software. Therefore it is common for developers to remain active on finished projects, because they require bug fixing.
Most bugs can be fixed by reusing the source code from their own repository, also called the plastic surgery hypothesis [7]. This means that it would be advantageous to have some sort of automated technique for reusing the source code from a project to fix a given bug. This is one of the problems that is solved by Genetic Improvement (GI). GI is the field of automated methods that improve an aspect a given piece of software, while maintaining the remaining aspects. GI can be very useful in reducing the amount of effort is put into bug fixing by a developer, so that they can focus on developing new software.
A lot is still unknown about GI, as it is a relatively new field. In this thesis, we will attempt to learn more about GI by approaching it with a fitness landscape analysis. We will analyse the search behaviour of GI in the landscape of programs and attempt to improve its capabilities. This leads to the following research question:

What insight can be gained from applying landscape analysis to functional improvement using Genetic Improvement on programs derived from the Bears benchmark?

To answer the research question above, a GI algorithm has been developed which analyzes the search space, instead of searching for improvements per se. In particular, it performs a random walk and explores the plateaus on which the fitness landscape is presumably built.
The GI algorithm has been implemented using the PyGGI framework with a custom connection to the Bears Benchmark [3]. The Bears benchmark is a collection of 251 bugged Java program with full test suites, ready to be compiled and tested automatically. These bugged programs have been collected from 72 different repositories with different code/test sizes, and error sizes. On these programs, a number of experiments will be applied in which the fitness landscape will be explored.

In this thesis, a detailed overview of the performed research and results is provided. In Section 2, a detailed summary of the required background information will be provided, while in Section 3 a breakdown of the relevant research will be given. In Section 4, we will break down how and why the experiments were performed, of which the results will be provided in Section 5. Then in Section 6, we will tie conclusions to these results and discuss these in Section 7. Finally, in Section 8, we will provide suggestions for future work to be completed regarding this area of research.

2 Background Information

Before delving into the details of the research performed in this thesis, some background information will be provided of the techniques and the tools used. In this section, the definitions and explanations of these techniques will be provided. The most important techniques are GI and fitness landscape analysis which will be explained first, and then some information and background will be provided on the tools used.

2.1 Genetic Improvement

GI is the a part of genetic algorithms and evolutionary algorithms and is a way of automatically improving human-written software. There are a lot of variants of GI, but all variants improve some aspect of a starting program. These improvements can be broken down into two categories: functional improvement and non-functional improvement. If GI is used to improve functional properties of a program, it means that the algorithms attempts to find a version of the program that performs functionally better than the original program. Non-functional improvement, however, is more concerned with improving properties like energy consumption for mobile devices, memory usage, and CPU usage, while maintaining the same functionality of the original program. GI typically requires 4 parts to function: a property to be improved; a formalization of the functional properties; a search heuristic; and a search algorithm. These parts will be explained in this subsection in more detail.

2.1.1 Improvement Property

First of all, there is the improvement property. This is the property that GI is attempting to improve upon, while maintaining some sort of functional property, which was also previously. The improvement property can be divided into two categories: functional and non-functional. We will start by looking at the functional properties.

First of all, there's the functional improvement property, in which a program is improved to expand its functionality. In practice, this usually means that a bug is present in a program and was not picked up upon by the original unit test suite. Then a new test is added to the test suite that is capable of unearthing the bug. On this program and test suite, GI can be applied to attempt to automatically fix the bug without introducing new bugs. This means that the old functionality of the program is retained, while more functionality (a bug fix) is introduced.
Let's take a look at an example of a bug in a program which can be solved with GI. For this, we are examining a small program that takes two integers a and b, and if a is greater than b, their values get swapped, after which a gets returned. However, the developer of this piece of code unfortunately made a small error while assigning a value to the temporary variable, resulting in an unfavourable result, as can be seen below:

```
1 if (a>b)
2 {
3     int tmp = b;
4     a = b;
5     b = tmp;
6 }
7
8 return a;
```

Listing 1: Example of a piece of code which returns the smallest of two integers, and can be optimized in terms of functionality.

This bug can easily be identified using a unit test, where a is greater than b and the result would be that their values get swapped. Using the above code, both values would come out to be equal to b's, which is not what we want. GI is able to solve such scenario's and edit the third line to instantiate the temporary variable to the right value. Assuming this is the correct behaviour for this part of the

code, no tests should be failing after the described edit and the program should be improved without having a developer fix the bug itself. Only a new test was required to fix the bug using GI, which is what makes GI a very useful and powerful tool.

The other type of improvement is the non-functional improvement. These are improvements which don't necessarily change the behaviour of the program, but result in some other improvements. The most common improvements are energy consumption, CPU usage and memory usage. In these cases, the test suite is still very important, as the optimizations made can not interfere with the original behaviour of the program. If this were to happen, the easiest optimization technique would be destroying the entire program, which can not be called an improvement. This type of improvement is very useful to apply to programs which are heavy resource users, where the optimizations can achieve the best improvements.
Let's take a look at another example. For this, we will look at the same small piece of code from the previous example, but we will assume that while developing, the developer of this code has added some debug code and forgot to remove it, resulting in a slower computation time than required for the functionality.

```
1  if (a>b)
2  {
3      int tmp = a;
4      a = b;
5      b = tmp;
6
7      Console.WriteLine(a);
8      Sleep(100);
9      Console.WriteLine(b);
10 }
11
12 return a;
```

Listing 2: Example of a piece of code which returns the smallest of two integers, and can be optimized in terms of computational time.

Of course, this example is quite exaggerated, but the point is clear. GI would be able to remove the lines 7-9 which would improve this piece of code in computational time. Note that this piece of code could also be optimized in terms of memory usage, by replacing the int type by a smaller type. Usually, a given GI algorithm is only attempting to optimize one property at a time, but there have been examples of multi-objective GI algorithms. Also note that while choosing a memorywise smaller data type in the above code might reduce the memory used while still passing the tests, it might not be intended behaviour to limit the size of tmp. Thus, a robust and complete test suite is required for GI to do what we want it to do.

Depending on the situation, a different property is chosen for GI to improve and will exhibit different behaviour based on which property is being improved. The difference between functional and non-functional property improvement has major impact on the different other parts of GI, which will be explained next. Especially the search heuristic is very different between functional and non-functional improvement, as the measurement of how much a program has improved is vastly different.

2.1.2 Formalization of Functional Properties

As also noted in the previous section, the improvement of a property requires that the functional properties of a program are well defined so that GI can search within those boundaries. We don't want to interfere with the original purpose of a program, so we need to find some way to formalize what requirements a program should be able to fulfill. This is almost always achieved using a test suite. A test suite is a collection of unit tests which assert that each part of a program behaves like it is expected to. An example of a test for the smallest integer code is given below.

```
1 int a = 5;
2 int b = 2;
3
4 int result = Smallest(a, b);
5 Assert.AreEqual(result, 2);
```

Listing 3: Example of a test for the smallest integer code.

This test would be part of a greater test suite, where more cases are tested of this method. For example, a case where a and b are equal or where a is smaller than b initially. These tests together make it that the behaviour of a method can be formalized, and when this is done for all methods, GI can be applied. This does mean that a thorough test suite is required for an efficient GI process. There are several external tools that can be used to achieve such a thorough test suite, ranging from Intellisense to Evosuite.

The test suite is required for GI in both the functional and the non-functional cases. The difference is that in the functional improvement case, not all tests start as passing. Those test cases are what is being improved upon and, if all goes well, will end up passing in the end. For example, the test from Listing 3 does not pass with the code from Listing 1, but once the right program variant has been found, the test will pass.

Conclusively, the functionality of a program is formalized by using a rigorous test suite. This test suite is required for both functional and non-functional improvement, and can either be manually created or automatically created with external tools. The quality of the test set has direct influence on the quality of the GI process.

2.1.3 Search Heuristic

The third part required for GI is a search heuristic. GI is a technique that searches for variant of an original program that performs better than the original program. Therefore, we need to introduce some kind of measurement of how well an individual program performs. This measurement, the fitness function, will be a function that the algorithm attempts to maximize. This means that the higher the fitness value is, the better a program is. For this we will once again make distinction between the functional and the non-functional cases of GI. In both cases, we will be defining a fitness function to determine the fitness of any given program, which will be used in the search algorithm.

Firstly we will look at the functional case. In this case, the fitness of a program is reliant on how much of the wanted functionality is present in the program. We will define the amount of passing tests of a program p with regards to test suite t as $P_t(p)$, and the amount of failing tests as $F_t(p)$. The most simple fitness function can then be defined as follows.

$$f_t(p) = P_t(p). \tag{1}$$

In this case, the fitness function is just equal to the amount of passing tests. However, this is not the most readable fitness function, as the fitness can not be directly translated to a sense of well fit. For example, if a program has a fitness of 500, is that good? It all depends on the total amount of tests. Furthermore, it is hard to determine when a global optimum has been reached. In the case of functional improvement, we know what the global optimum is, namely all tests passing. It would be beneficial to easily see when this global optimum is reached, as that is a program that contains all the properties that we want it to have. The following fitness function takes that into consideration.

$$f_t(p) = \frac{P_t(p)}{P_t(p) + F_t(p)}. \tag{2}$$

The fitness function above reaches its maximum value when $F_t(p) = 0$, where $f_t(p) = 1$, and its minimum value when $P_t(p) = 0$, making $f_t(p) = 0$. In other words, if no tests are failing, then the fitness value is 1, but if all tests are failing, the fitness value is 0. So the fitness function is translated

to the interval $[0, 1]$, making it more readable. A fitness of 500 does not express a lot, but a fitness of 0.75 does.
Finally, there are many variants possible on the fitness function. For example, GenProg, a popular tool for applying GI, has a fitness function that distinguishes between different types of tests [38]. Tests that passed on the original program are treated differently then tests that failed on the original program. Let p be the original program and p' be the program of which we are determining the fitness. Then P_i is the set of tests that pass given program i, and F_i is the set of tests that fail given program i. Then the fitness function of GenProg is defined as follows:

$$f_t(p') = W_+ \cdot |P_p \cap P_{p'}| + W_- \cdot |F_p \cap P_{p'}| \,. \tag{3}$$

In this equation, W_+ and W_- are static weights which determine how important it is that already passing tests keep passing, and how important it is that previously failing tests now pass. The default values make the second category twice as important as the first. This makes it possible to value newly passing tests higher than previously passing tests, as the latter is explored territory and should be easier to return to. Do note however that this fitness function suffers from the same problems as function (1). In all cases, the fitness function is heavily reliant on the amount of passing tests, but many different variants exist.

The fitness function for non-functional improvement is different from the functional improvement. If a program fails any test from the test suite, that program should be considered not to have any fitness value, as it does not have the same functionality as the original program and is thus unfit. These programs get skipped during the iterative phase of the GI algorithm. The programs that do have the same functionality as the original program do have a fitness, which is dependent on which non-functional property gets improved. In the most cases, these properties can easily be measured. There exist a lot of tools that can help track CPU and memory usage and energy consumption for a given program. An easier property to measure is computational time, but this is often not a favourable measurement. Computational time is very dependent on other factors such as other processes on the computer at the same time.
Now let's look at an example of a fitness function for non-functional improvement. We will name $R(i)$ the measurement for resource usage of a program i. Now the fitness function of a program p' with respect to original program p is given as follows:

$$f(p') = R(p) - R(p'). \tag{4}$$

Note that in this equation, we measure the improvement that some program p' made over the original program. If p' uses more resources that the original, the fitness value of this program would become negative. This is why we can't say that a program that does not pass all the test would have a fitness value of 0, as that would imply that it is equally good as the original program, which is blatantly not the case. This fitness function does not have a maximum or a minimum value. This problem is harder to avoid compared to the functional case, as there is no sense of what the maximum improvement will be. The improvement can be translated to a fractional improvement instead, which will look somewhat like this:

$$f(p') = \frac{R(p) - R(p')}{R(p)}. \tag{5}$$

While the property that gets improved differs between different implementations of GI, the fitness functions are kept the same across different implementations. This simple function captures what we want to measure, and will thus suffice for GI.

Conclusively, the fitness functions for GI are relatively simple, which is a good thing. GI is a complicated process that take a lot of time to run, so a simple fitness function helps condense different programs into a numerical value while being easy to calculate. In every instance, there are some variations possible to the likings of the developer, or to suit the situation. No single fitness function is strictly better than another, so the choice should be based on the context of the GI implementation.

2.1.4 Search Algorithm

In this section we will further elaborate on the search algorithm of GI itself. This explains how the algorithm takes the original program and finds variations on this program. There are three parts to this; the edits applied to the original program, the representation of variant programs, and the order in which programs are evaluated.

Firstly, there are the different edits, also called mutations, that can be applied to a program. There are three operations that are commonly applied to a program: the delete edit, the replace edit, and the insert edit. These edits can be applied to a program on different levels, where the abstract syntax tree (AST) and the line level are the most common. We will first explain what the edits are on the line level, and then expand to the AST level.

Firstly, we will examine the delete edit. This is a mutation where a piece of the code gets removed from the program. Let us take a look at an example of the delete program by examining the code in Listing (2). Let's say we perform a delete operation on line 8, the resulting code would look like this:

```
1  if (a>b)
2  {
3      int tmp = a;
4      a = b;
5      b = tmp;
6
7      Console.WriteLine(a);
8      Console.WriteLine(b);
9  }
10
11 return a;
```

Listing 4: The code of Listing 2, after the delete(8) mutation has been applied.

In some variation of the delete mutation, the line is not deleted as a whole but rather replaced by an empty line. Effectively, both these mutations provide identical programs. There is a subtle difference though in the further process of GI. Leaving an empty line means there is a possibility for it to be mutated on again, whereas removing the line entirely does not allow this. The implications of this are very subtle, so it is up to the developer to choose which behaviour the delete mutation has.
If any given program has n lines, there are $O(n)$ possibilities to apply a single delete mutation. Since we are talking about the line mutation variant, the n in this case is equal to the amount of lines within a program. One can imagine that in larger program this number can grow quite high. To ensure that GI is useful, there are some ways to reduce the amount of places where the mutation can be applied. More on this will be elaborated on later.

The second mutation is the replace mutation. This means that a line of code is replaced by another line of code. The line that gets replaced is called the target, and the line of code that replaces it is called the ingredient. Let's once again look at the code from Listing (2) and apply the replace edit with target line 7 and ingredient line 4:

```
1  if (a>b)
2  {
3      int tmp = a;
4      a = b;
5      b = tmp;
6
7      a = b;
8      Sleep(100);
9      Console.WriteLine(b);
10 }
11
12 return a;
```

Listing 5: The code of Listing 2, after the replace(7, 4) mutation has been applied.

As can be seen, the code at line 7 is replaced by a copy of line 4. If there are n lines of code in a program, then there are n possibilities for the target line and $n-1$ possibilities for the ingredient. It would not make sense to choose the same line for both the target and the ingredient, as that would result in the exact same program as before the mutation. That means that there are $O(n^2)$ possibilities to apply the replace mutation to a program. Even more than the delete mutation, the amount of possible variant programs grows quickly as programs get larger.

The last common mutation is the insert mutation. This edit is very similar to the replace edit. The difference is that the insert mutation takes an ingredient and inserts it before or after a given target line. Whether this is before or after can either be chosen statically, be determined randomly, or any other way to determine it. Below is an example of the insert mutation with target line 7, ingredient line 4 and the insertion happens before the line:

```
1  if (a>b)
2  {
3      int tmp = a;
4      a = b;
5      b = tmp;
6
7      a = b;
8      Console.WriteLine(a);
9      Sleep(100);
10     Console.WriteLine(b);
11 }
12
13 return a;
```

Listing 6: The code of Listing 2, after the insert(7, 4) mutation has been applied.

Note that the code in Listing 6 is the result of insert(7, 4) with the insertion happening before the line, but it could also be the result of insert(6, 4) with the insertion happening after the line. This is why the determination of before or after does not matter much, as long as the insertion can take place before line 1 and after line n. This leaves us with n possibilities for the target line of the insert mutation, and $n+1$ possibilities for the ingredient line. This means that in total, the insert mutation gives us $O(n^2)$ possibilities, just like the replace mutation.

One important thing to note about the mutations is that these mutations can be constructed by applying each other. For example, a delete edit is basically a replace edit using an empty line as the ingredient. Or the replace edit is applying a delete edit followed by an insert edit. Some times, a fourth edit is used as a base as well, namely the move edit. This removes code from the ingredient location and moves it to the target location. This mutation can also be built from the mutations we already know, by applying an insert followed by a delete. In most researches, the delete, insert and replace edits are seen as the base edits, and other edits are compound mutations from these. Also note

that some variant programs can be found by applying different mutations, meaning that there is some overlap in the search space for GI.
Another important note is that not every mutation leads to a compilable program. For example, a line with only a curly bracket could be inserted into the code somewhere, leading to parse errors when compiling. While these programs do not need to run every test on them, they do require time to find the compilation error. Therefore it is still important to optimize the mutations to reduce the amount of programs with a parse error in them.

One way to reduce the amount of parse erroring programs is by mutating the abstract syntax tree, instead of mutating a program on the line level. For this, let us first examine what the abstract syntax tree is and how a program gets broken down into such a tree.
The abstract syntax tree is a representation of a program. This representation is language-independent as far as imperative languages go, and any program can be translated into a tree. For the purpose of this thesis, we will consider the abstract syntax tree as an XML file. Let us consider the if-statement from Listing 1, and transform it into an abstract syntax tree. The result is shown below.

```
<if>if
    <condition>(
    <expr><name>a</name>
    <operator>></operator> <name>b</name></expr>
    )</condition>
    <then>

    <block>{
            <decl_stmt><decl>
            <type><name>int</name></type>
            <name>tmp</name>
            <init>=
            <expr><name>a</name></expr></init>
            </decl>;</decl_stmt>

            <expr_stmt><expr>
            <name>a</name>
            <operator>=</operator>
            <name>b</name>
            </expr>;</expr_stmt>

            <expr_stmt><expr>
            <name>b</name>
            <operator>=</operator>
            <name>tmp</name>
            </expr>;</expr_stmt>
    }</block>
    </then>
</if>
```

Listing 7: The if-statement from Listing 1 transformed into AST in XML format.

As one can see, even a small piece of 6 lines of code creates quite a large XML file. In this file, every aspect of different statements and different blocks are dictated by XML tags. This makes it possible to only exchange expressions which are of equal type to one another. It would not make sense to exchange a variable for an operator for example, or inserting a declaration statement into an if-header. By breaking down a program into these tags, we can make mutations on the statement level, which will lead to fewer parse errors. Furthermore, it allows us to make more precise mutations instead of editing entire lines at a time. Of course, it is still possible to mutate on entire blocks at a time, which means that less precise mutations are possible. Using the abstract syntax tree in GI is known as using tree mode, and allows for generalizing GI algorithms to be language independent.

Now that we know how programs get edited to form new programs, we will examine in which way these new programs are saved. One can imagine that it is highly infeasible to save all new programs. Instead, only the edits to a program are saved. This means that the search space is actually just a series of edit sequences. By examining the programs in this way, it makes it much easier for genetic techniques to be applied, like crossover.

The final piece of the puzzle that is GI is the way the search space gets explored. Many different techniques exist, but we will explain random walks, local search, and genetic programming. These are all ways in which the search space is travelled upon to find an improved version of the original program.

The first search strategy is the random walk. This strategy is the simplest and purely exploratory. In each iteration of this strategy, an edit sequence of up to m mutations is generated and applied to the original program. The newly acquired program is then tested and the best program is kept track of. Each iteration starts with the original program, so there is no sense of continuation from a previous iteration. After a set amount of iterations, the best found program is returned.
This strategy is quite a simple one, and does not always provide good results. Because of its simplicity, it does not explore much of the available space. However, as there is no one good solution in terms of programming, it is very much possible that this strategy will come up with improved programs, albeit slightly.
The strategy has some upsides and some downsides. As the search space is visited very randomly, a good sample of the search space is taken implicitly. However, there is no sense of how good a single mutation is. It is very much possible that the best found edit sequence contains edits that do not contribute to the improvement, or even negatively interact with the fitness.

The second search strategy is the local search strategy. This strategy is very similar to the random walk strategy, but in this strategy not every iteration starts with the original program. If a better program is found at any point, it gets chosen as the new base program from which to mutate. In this search strategy, each iteration an edit gets added to, or removed from a base edit sequence. This makes it possible to remove any edits that can have a negative effect on the overall fitness, which was not possible in the random walk.
There are a few variants within local search that are used. This has mostly to do with edge cases where other programs are accepted as better or not. Most commonly, there is the first improvement local search and the best improvement local search. First improvement accepts any solution that is found that is equally good or better than the base program, where best improvement only accepts better programs. There is a case to be made for both strategies. First improvement explores a larger portion of the search space, where best improvement attempts to converge faster to the best solution. Especially in the functional case where there are no slight changes in fitness value the choice of improvement is an important one.
Furthermore, there is a variant in how neighbours are being searched for. It is possible to enumerate all possible mutations and evaluate them one by one, or by selecting random mutations. Enumeration could work well for the best improvement local search, where random mutations are better for first improvement local search.

Finally, there is also the concept of genetic programming. Genetic programming is quite different to local search and random walks. Genetic programming starts with a population of k program variants, each consisting of an edit sequence containing one edit. In every generation, the best $k/2$ edit sequences get selected and each parent produces two offspring. One is generated using mutation and one is generated using crossover. These offspring are filtered on validity and then evaluated and sorted based on fitness. The k best programs continue to the next generation.
The process of mutation is similar to local search and random walks, where a random mutation is added to the parent's edit sequence. It is also possible that a random mutation from the parent's edit sequence is removed. If a mutation is added to the edit sequence, then it is appended at the end of the sequence. The removal of a mutation can happen at any point in the sequence.
Crossover is a concept that is only found in genetic programming. Each parent selects a random other

edit sequence from the population and together they produce one offspring. There are generally four different types of crossover. The first type is concatenation, where the edit sequences of the parents are simply concatenated one after another. The order in which the sequences are mostly irrelevant, as most mutations are independent of each other. The second type of crossover is single-point crossover. In this type of crossover, both edit sequences are split into two subsequences at a random location in the sequence. Then, the first subsequence of one parent is concatenated to the second subsequence of the other parent, producing a new offspring. The third type of crossover is uniform concatenation. This is the same as regular concatenation, except that each individual edit has a 50% chance of being dropped from the final sequence, resulting in a sequence that is the average length of its parents. The final type of crossover is uniform interleaved crossover. For this, from each parent half of the edits get randomly selected, and these edits retain their relative position. This process can also be seen in Figure 1.

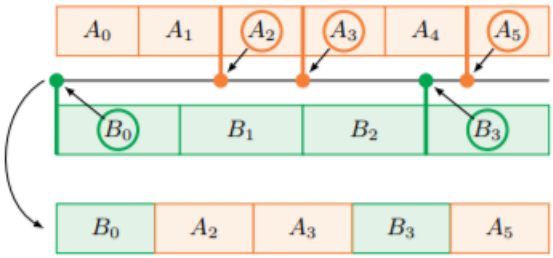

Figure 1: An example of how uniform interleaved crossover is performed on two edit sequences of different lengths. Extracted from [8].

Each search algorithm has their own set of upsides and downsides and the implementation that is chosen should be dependent on the situation in which GI is being used. For the purpose of this thesis, the main objective is learing about the search space. Therefore, the techniques that will be used in this thesis have been chosen to be exploratory, and not efficient per se. For more information on the chosen techniques and why these were chosen, see Section 4.

2.2 Fitness Landscape Analysis

The other technique that is used in this thesis is fitness landscape analysis. This is the study of researching a fitness landscape that is generating from some fitness function, and determining interesting characteristics about the problem. For example, problem hardness is a characteristic that has been researched thoroughly in the field. Other characteristics are, for example, more basic descriptors of the landscape such as local optima density and basins of attraction.

In this section we will explore how a fitness landscape analysis looks like and define some properties that will be helpful in this thesis. In this research, we will use fitness landscape analysis to determine characteristics of the fitness landscape of GI algorithms. This may help by improving the GI algorithms or by determining in what area more efficiency can be gained. For this reason, we will focus this section on the parts of fitness landscape analysis that are useful when looking at GI.

To achieve this, we will first lay a foundation on the concepts of fitness landscape analysis. Then we will look at one of the biggest hurdles of GI, namely plateaus. Finally we will examine what the latest advances in fitness landscape analysis are and whether they can be applied to GI.

2.2.1 Basic Characteristics

First of all, let us examine the very foundation of fitness landscape analysis by defining what a fitness landscape is. We start by defining the space of program variants as S, and the fitness function as

some function $f : S \to \mathbb{R}$. Furthermore, we define a function $d : S \times S \to \mathbb{N}$, which measures the distance between two given program variants. This function usually forms a metric. Now we can define the fitness landscape itself as a combination of the search space, the fitness function and the distance function.

$$\mathcal{F} = (S, f, d). \tag{6}$$

Now we will also define a set which contains the neighbourhood for any given program x:

$$N(x) = \{n \in S : n \neq x, d(x, n) = 1\}. \tag{7}$$

So, the neighbourhood of some give program is all other programs in the search space which are not x, and the distance to x is one. This is the distance we define neighbouring programs to have, meaning that one mutation can bring you from a program to its neighbour. A program itself is not contained within its own neighborhood. Now that we have these definitions, we can formalize some common concepts, such as a local optimum and the global optimum.

Definition 1. *A program x is a local optimum if and only if $\forall y \in N(x)$ it holds that $f(x) > f(y)$.*

Definition 2. *A program x is a global optimum if and only if $\forall y \in S$ it holds that $f(x) \geq f(y)$.*

Now that the very basic definitions for the fitness landscape have been provided, let's take a look at how some basic descriptors of fitness landscapes are. For this we first introduce the concept of a connected set C. A connected set is a set of programs where every program can be reached from every other program without traveling through programs outside of the set. Like the definition for the neighbourhood of a single program, we can define the neighbourhood of a connected set as follows.

$$N(C) = \{n \in S : n \notin C, \exists x \in C, d(x, n) = 1\}. \tag{8}$$

A connected set C has equal fitness if $\forall x, y \in C, f(x) = f(y)$. Connected sets of equal fitness are either a plateau, also called a generalised local optimum, which means that all neighbours of the set having a worse fitness; or it is a ridge, where there is at least one neighbour which has a better fitness then the connected set. We say that a connected set of equal fitness C has fitness $f(C)$. This leads us to the definition of a generalized local optimum.

Definition 3. *A connected set of equal fitness C is a generalised local optimum if $\forall x \in N(C)$ it holds that $f(x) < f(C)$.*

These are the definitions of the fitness landscape that will be used during this thesis. While there are a lot more definitions in the area, we will not expand upon these in this work. For an overview of these definitions, we advise to take a look at the survey by Pitzer and Affenzeller [58].

2.2.2 Plateaus

The main problem with the local fitness landscape of GI with functional improvement is that the connected sets of equal fitness are very large in size. There are a lot of edits that can be applied to a single program without changing the outcome of any of the tests. These ridges and plateaus do not seem to have any distinguishing factors to search within, so this is something that will be researched in this thesis. It is currently unknow how large the plateaus actually are, and whether or not there are plateaus which are not the global optimum. Do note that in the functional improvement variant of GI we do know what the global optimum is. The global optimum is any program that passes all tests in the test suite.

2.2.3 Latest Advances

Most of the latest advances in fitness landscape analysis require large amounts of computational time and are not very suitable in terms of the GI fitness landscape. For example, there is a trend in using network analysis techniques in combination with fitness landscape analysis to gain new understandings.

These techniques can be applied to GI algorithms on small programs, which has been done in a research by Veerapen, Daolio and Ochoa, but on medium to large sized program this is not feasible anymore [77]. This also has to do with the evergrowing size of plateaus as programs gets larger. Therefore we will stick to more basic metrics and characteristics which can easily be applied to fitness landscapes where the neighbourhood is very large.

2.3 PyGGI

One of the main tools that was used during the course of this thesis is the PyGGI framework [3]. This framework provides tooling to easily implement a GI algorithm in python, which is then capable of adapting and evolving programs from all languages. The framework itself comes with few simple implementations. In these implementations, it is shown that the framework is capable of improving both python and Java programs, and that it is able to work towards different goals. In this section, we will provide a more in-depth analysis of the PyGGI framework. We will talk about the different possibilities that are delivered with the framework and how they can be used to create custom GI algorithms.

The PyGGI framework is written in python and comes with different algorithms and example programs to show its functionalities. These demo programs show that PyGGI can work with both functional and non-functional improvement, and that the framework works across languages. In particular, it shows that a python and a java program can be improved. Furthermore, PyGGI shows that it is able to work in tree mode, by having XML files available as well. PyGGI is not capable of translating source code into the AST, but it does advise using srcml for this purpose. This has also been applied in the context of this thesis.

PyGGI is built using an abundance of abstract classes, with a few implementations of these to provide an example. This makes it very easy for a developer to create custom implementations of these classes to suit their particular needs. For example, new mutations can be added, as long as they have a create method and an apply method. The create method is used to save edits in the edit sequence, whereas the apply method is only used during validation of a certain edit sequence.

Some other variables that can be customized by the developer are the amount of iterations, the amount of episodes, and the amount of warmup reps. The amount of iterations is how many programs get validated if the stopping condition is not met. The stopping condition is something that can be defined by the developer and is an optional way the GI process can be stopped preemptively. This can be used when GI is applied with a certain goal in mind, such as passing all test cases or achieving a certain computation time. The process of taking the base program and mutating it until the stopping criterion is met or all iterations have passed is repeated as many times as the amount of episodes. The amount of warmup reps is only relevant when performing non-functional GI, where a solid baseline is needed in terms of fitness of the original program. It is possible that if the original program is only evaluated once that the measurement is not representative of its actual fitness, because the computer was simultaneously performing some other task, for example. The warmup reps are the amount of times the original program is validated to gain a representative baseline. These are not applied to the variant programs, that would take too long computationally wise, and is not as bad as having a bad baseline.

Next up, we will take a closer look at the workings of the basic PyGGI supplied algorithms. Firstly, there is the basic local search that is supplied. This local search algorithm is a first improvement local search as explain in Section 2.1.4, and the code for it can be seen below. The odds of adding a mutation to the current edit sequence are as large as the odds of removing a mutation from the edit sequence. This makes it so that the general edit sequence size is small, but that there is room for exploration. If the odds of adding a mutation would be greater, you would expect the average edit sequence size to grow over time. If the odds of deleting a mutation would be greater, it would reduce the probability of evaluating an edit sequence with multiple edits, and thus remove some of the exploratory characteristics of the search algorithm.

The edits that are added to an edit sequence are chosen randomly, and the target and ingredient domains can be defined in the configuration of the algorithm. By reducing the domains, the amount of possible mutations can be reduced by a lot. Reducing the ingredient domain is something that is possible, but the effect is not as clear as reducing the target domain. The ingredient to fix a bug in a certain location could come from anywhere in the code. In the case of functional improvement, the target file can easily be reduced, as the bug can be located when examining which test does not pass.

```
class MyLocalSearch(LocalSearch):
    def get_neighbour(self, patch):
        if len(patch) > 0 and random.random() < 0.5:
            patch.remove(random.randrange(0, len(patch)))
        else:
            edit_operator = random.choice([StmtReplacement, StmtInsertion,
                StmtDeletion])
            patch.add(edit_operator.create(program))
        return patch

    def stopping_criterion(self, iter, fitness):
        return fitness < 100
```

Listing 8: The standard local search algorithm that PyGGI uses to improve a program in tree mode.

The last thing to talk about is how the PyGGI framework handles the tree mode improvement. A custom set of mutations has been developed for the tree mode which essentially do the same thing as the mutations for line mode, but are adapted to work with the tags rather than with lines. An interesting decision made by PyGGI is to rewrite some tags to a single mutatable tags. These tags are "if", "decl_stmt", "expr_stmt", "return", and "try". These all get rewritten as "stmt" and those are the only mutable tags in the xml file. This means that any loop headers can be mutated, but no loops can be added or removed, and some information is lost on which tags are of equal type. However, it does make it easier to have different tags mutate with each other in situations where this results in a valid program.

2.4 Bears Benchmark

The Bears Benchmark[1] is a collection of 251 bugged programs from 72 different repositories [42]. The bugs in these programs can be verified because of the existence of a test case which tests for the bug. These programs contain the bugged version of the particular program, and also a human-written patch for the bug. In most occasions, the original repository did not contain a test for the bug and the test is added in a prior commit. In these cases, the commit in which the test is added is also collected. All these programs and patches are obtainable through the Bears Benchmark repository on GitHub, where each bugged program is represented by a branch in the repository. The main branch of the Bears Benchmarks also contains a number of scripts that help with checking out a particular bug by number (for instance, Bears-123), compiling it and testing it. All programs in the benchmark have been written in Java and are compiled under Maven. As the benchmark was created a few years ago, the most recent versions of both Java and Maven cannot be used, but legacy versions are available.
Furthermore, the Bears Benchmark contains an XML file for each bugged program containing all sorts of metrics of the project itself, the exact build that caused the problem and the build that fixes the problem. It is also detailed how many tests pass and how many tests fail, what the assertion failures are, and which classes contain the failures. This makes it very easy to collect information from the entire benchmark on metrics like the amount of tests run and the amount of tests failing. Other information that could potentially be used, but was not used for this thesis, is for example the amount of time between the buggy build and the patch build.

[1] https://bears-bugs.github.io/bears-benchmark/!/

3 Literature Research

In this section, a scientific background of the relevant fields will be given. GI is a field which has mostly started to gain traction in the last decade, so most scientific background is from around this era. There is also some background which laid the foundation for GI, like program synthesis. This is why the scientific background of GI has been split into two parts. This section starts by exploring the foundational background on GI (Section 3.1), starting with program synthesis. Then, the GI papers from the last decade will be discussed (Section 3.2). Then we jump into the field of Fitness Landscape Analysis (Section 3).

3.1 Background of GI

There are three fields that form the basis for GI as we know today. Two of these fields, program synthesis and program transformation, were already researched in the 70s. The third field, automated test suite generation, was a more recent development in the field. This addition led to all the pieces being available for GI to gain traction. These three fields thus lay the foundation for GI and will be analysed in this section. [55]

Program synthesis is the study of forming a program from a set of corresponding inputs and outputs. Generally, program synthesis is performed as an extension on theorem-proving programs, where the relation between the input and the output is formulated as a theorem. The way this theorem is then proven is used as a basis to synthesize the program from. Manna and Waldinger [44] managed to improve the then current program synthesis research field by exploring how loops could be integrated into the target program. [44]
The concept of introducing loops into programs created by program synthesis proves to be a worthy challenge. This challenge is also addressed in a paper by Manna and Waldinger, which also addresses conditional tests and instructions with side effects [43]. In this specific paper, the bridge to GI has been mentioned, as the authors claim that starting with an existing program to solve a slightly different problem is a powerful approach. While this is not exactly what GI is, it is remarkable given the publication date.
A more recent paper on program synthesis is a paper which explores the different dimensions of program synthesis [22]. This demonstrates that the field of program synthesis is still an active one and quite some advancements have been made in the past decades. As this paper presents, currently the focus of program synthesis is not so much on how to make it work, but what to make it work on. The current application of program synthesis includes finding new algorithms, easing repetitive programming and understanding programs.
Program synthesis forms the basis for GI in the sense that it seeks to create a correct program for a given problem. The most notable difference between the two fields is that program synthesis starts from scratch, while GI improves something that already exists. However, there is also a difference in perspective on correctness of programs. Where program synthesis mostly relies on correctness by definition, GI relies on correctness by testing, and agrees that this may not always assure correctness [55].

The next field that forms the foundation for GI is program transformation. The field is mostly concerned with transforming programs to minimize either space or time constraints. Therefore the field is interested in formulating equivalence classes for programs, such that programs within one class constitute the same behaviour. Impressively, this concept has first been mentioned in 1843 [41]. Since then, a lot of effort has been put into understanding when a program can be transformed into another program while preserving the semantics [74] [49]. This would make it possible to provide a solid mathematical background on computing science theory and practice.
Another research more focused on the equivalence classes by determining that some general transformations do exists [65]. Furthermore, there are also specific transformations for specific program classes. This means that certain transformations within a program class stays within the class, while this may not be the case for the same transformation in other classes.

More recently, the research focus has shifted within the program transformation field. For example, a language has been created which enables source to source transformations [15]. It allows the adaptation of language features themselves to obtain equivalent programs. Furthermore, experimentation with different parsing methods can be experimented with, and one can find out whether different parsing methods result in the same compiler.
A similar tool to TXL is the Stratego/XT framework [78]. This framework provides facilities for the infrastructures of parsing and pretty-printing as well as addressing the entire process of a transformation.
Program transformation is useful for GI in the obvious sense that both fields are interested in changing something to a program resulting in another program. For program transformation, a program is attempted to be changed into a program with the same semantics, while in GI the semantics are attempted to be changed as well. Yet, the advances in program transformation have been very useful for GI as well. For example, the development of the XML language helped both fields, as it became easier to target specific parts of a program for it to be transformed. For this reason, program transformation makes an excellent foundation for GI.

The last field that forms the foundation for GI is that of automated test generation. Automated test generation is a field that has been researched very early on in the history of computing science. There has been some discussion among researchers about the usefulness of testing in the first place. This doubt is because testing can be used to find bugs in programs, but it can not be used to show an absence of bugs, as Dijkstra formulated back in 1969 [18]. However, more recent systems have made it possible to show the absence of bugs under certain assumptions. While testing may not be perfect, its application in GI suffices.
The first known entry of automated test generation is from 1962 [62]. In this research, Sauder described a system for generating test data to conform with the description given in the data division of a COBOL program. While the notion of test data sounds primitive, the relations between the data were quite advanced already. Furthermore, they feel familiar to the assert statements we know today. Since then, a lot of new innovations have been found, mostly in the period until the 1990s. These include a system which tests and debugs programs using symbolic execution [11], automating test generation using algebraic constraints [17], and automated test data generation using dynamic data flows [34]. However, the most significant advancements have been made in the last decade. These advancements can roughly be divided into three subareas: dynamic symbolic execution, search-based software testing, and mutation testing [55].
Dynamic symbolic execution is a technique that can be used to find deep-rooted bugs in complex software architectures. The key idea behind dynamic symbolic execution is not to generate test data, but to analyse the possible execution paths through a given program. For each of the possible execution paths, a single test data point is then generated to test this particular path. Recently, symbolic execution has been mixed with assigning concrete values to execution paths [64]. This has been done because a software system may be reliant on external systems with unknown execution paths, which is a weakness of traditional symbolic execution methods. Another weakness of symbolic execution testing is the vast amount of execution paths in large programs [14]. Generally, this amount grows exponentially. A solution to this is to use heuristics to extrapolate important execution paths and to provide tests for these within a certain time constraint [14]. [33]
Search-based software testing tackles the problem of test generation by using computational search techniques, as the search domain is typically vast and complex. Search-based software testing is a subdomain of search-based software engineering [50]. In fact, about half of the search-based software engineering research is concerned with search-based software testing. Search-based software testing is very useful for the field for GI, because it is able to search for test cases which exhibit erroneous behaviour in programs. The step from this to finding versions of programs that do behave correctly for these tests. The same can be said the other way around, making GI and search-based software testing fields with a very symbiotic relation. [24]
Mutation testing is a bit different, as it is not a technique of generating tests for software. It is, instead, a test of the test suite itself. In mutation testing, small common errors are applied to a piece of code after which the test suite is run. The test suite should be able to detect such common errors that

programmers often make [28]. Mutation testing dates back to 1971 after which it was picked up in the late 1970s [40]. This technique is commonly applied in many different languages. For example, the research by Kim et al. tested multiple mutation methods for Java and compared effectiveness [31]. It shows that the different methods do not differ much from each other, but there is space for improvement, especially compared to procedural languages. Mutation testing is not only useful at the unit level, but also at the design level, where specifications or models of an entire project can be tested. For example, this research by Le Traon et al. attempts to create a policy for testing security policies, also using mutation testing [39]. This specific research concludes that these types of tests are required to retain a reasonable amount of confidence in security mechanisms.
Of course, next to these fields, there are other examples of test suite generation. One such an example is Evosuite, which is a very recent addition in the automatic test suite generation [21]. It uses a genetic algorithm to find suitable tests for a given Java class, and can be extended to find suitable tests for entire pieces of software.
In conclusion, the field of automated test suite generation is not a straightforward one. However, due to recent advances, the field has evolved far enough to allow GI to begin its uprising. Especially the search-based software testing synergizes well with GI.

As mentioned above, each of the three fields have their reasons why they contribute to the origination of GI. Without any of these fields, GI may not found its origin, or at least not in the way it has now. Therefore, these fields are considered the background of GI.

3.2 Genetic Improvement

In GI, there are always one or more properties that are preserved. If this were not the case, and all properties of software would be altered, it could not be called improvement anymore. The preserved property is not always the same. GI can be used to make a program exhibit the same behaviour, while using less resources (time, space, or energy typically), or it can be used to fix non-functional parts of software, while preserving functional parts.
One way of preserving behaviour of software is by only using semantics-preserving transformation [65]. However, in practice this does not guarantee correct program behaviour [52]. In the case of Orlov and Sipper, they prevented verification errors in Java bytecode, but still ran into run-errors. Another property of using semantics-preserving transformations is that the search space is vastly reduced. This can be viewed both as an advantage and a disadvantage. It is an advantage because GI typically requires some solution to deal with a vast search space. It is also a disadvantage however, as there is no guarantee on the quality of the remaining search space; it might be possible that all good solutions require transformations that do not preserve semantics.
Another possibility is that functionality is only partly preserved as a trade off for efficiency. The overall goal of the program then stays the same, but a level of inaccuracy is accepted in order to get a faster program. This technique can be seen in the paper by Sitthi-Amorri et al. where a shaders in procedural languages can be improved in terms of efficienc, while maintaining an acceptable level of accuracy [67]. The challenge in using this technique is determining the right balance between accuracy and efficiency.
A third alternative to preserve functional properties is by taking the desired functionality and evolving it in a vacuum, apart from the rest of the program. The name of this technique is software transplantation. This approach, also known as the grow and graft approach, grows a certain function using functional demands for this function, and is afterwards grafted into the original program. [36]
In all cases, the way that the properties that need to be preserved are expressed in test cases satisfying that property. In the most favourable case, the available tests are equal to all tests required to test a certain property. However, in practice this means infinite, or near-infinite test suites. This is not feasible because all tests need to be performed on every candidate solution, so this computation time must be kept small. It can be reduced with prioritization techniques as proposed by Fast et al. and Mao et al. [19] [45]. The fact that the choice of test suite is very important for the function of GI is shown by Smith et al. [68]. While it is not entirely clear what the relation between the quality of GI and the choice of test suite is, Smith et al. have shown that a higher test coverage leads to better patches. Thus, the test suite is vital for preserving some property when using GI.

As discussed before, GI does not start from scratch, like program synthesis does. Instead, all known applications of GI take some sort of starting program which is improved upon [83]. The way this source material is processed generally falls into one of two category. Either the source itself is reused to from patches, or code transplants are used.
The plastic surgery hypothesis assumes that a code patch can be reconstructed from some code that already exists somewhere [7]. Typically, there are three possible sources to look for this already existing code. The first is the easiest, namely the original program itself. It is also possible to find a code fragment in another program of the same language, or even another program in a different language. Most often, the code is attempted to be reconstructed from the original program itself. In fact, 43% of all Java bugfixes could have been reconstructed from the original source [7]. This makes the original source a valid and easy candidate for extracting necessary code fragments.
The idea of code transplantation in terms of GI is similar using code from another program to improve one's own code. The key difference is that in code transplantation the feature that is being transplanted is often missing from the target program. As adding a feature to a program like that can cause some issues, the refinements are then completed using GI [47]. Although it is uncommon, it is also possible to grow a transplant from scratch after which it is grafted into the target program. For example, Jia et al. grew a system for citations in a web development framework from scratch [29].

In GI, the choice of fitness function is an important one. Not only does it influence the search behaviour, but also which properties are preserved and which are improved. The properties to improve are either functional or non-functional. The functional properties in GI are bug fixes and feature implementations. The non-functional properties are typically run-time or energy consumption. The latter is especially popular in recent years, mostly in terms of mobile applications. The most researched property to improve is software repair, however. Depending on the aim of the GI software, different fitness functions are implemented. For functional properties, tests are used as a fitness function, while non-functional properties often employ a different function. It is uncommon, but possible for a single GI system to improve both functional and non-functional properties, which require multiobjective fitness functions. [55]
It has often been demonstrated that a GI can be successful with passing test suites in terms of program repair [63] [4] [6]. In particular, a lot of work has been done on the GenProg tool [38]. These researches on program repair using tests as fitness function typically just use the amount of passing tests as fitness measure. Sometimes a distinction is made between which tests which have always failed, and tests which fail only in the modified version. This is to remove any chance of malicious side effects occurring in the GI process [63]. While a simple approach, it has proven to be successful on numerous occasions.
On the other hand are the fitness functions which lead to non-functional properties to be improved. These are typically either memory usage [84], time usage [56], or energy usage [13] of the program. Memory usage is a property that can be measured from the compiler and thus does not require much overhead. The optimization of memory usage may also lead to an optimization of time usage [84]. For energy optimisation, it is especially tough to come up with a working fitness function. Energy consumption is quite a volatile property and the measurement of it is not always possible during GI iterations. As a result, a system has been made that can create fitness functions for GI by using GI[25]. When improving a certain property, it might mean a degradation to another property [82]. For example, it is possible to improve the runtime of some algorithm by using more memory, but this may not always be the sought after result. The improvement of one property can influence other functional or nonfunctional properties [84].

The power of GI compared to human improvement lies in the ability to create and evaluate a large search space. The trick is creating such a search space that it is still possible to evaluate the programs in such a way that improvement is actually achieved. The way that a GI algorithm searches through the available space is generally dependent on two factors: the available search operators, and the internal representation of the program.
The search operator can be chosen freely, mostly dependent on the result to be achieved. Among the

possibilities are deletion on the binary level [35], making modifications in the abstract syntax tree [20], and mutation testing [84].
The program on which GI is performed can be stored internally in three different ways typically. These are in an abstract syntax tree [5], as bytecode [20], or as plain text itself [3]. Depending on the complexity of a bug or the required search space, a choice of method can be made, as these naturally generate different search spaces together with the choice for search operators.

3.3 Fitness Landscape Analysis

When we say we are analysing a landscape, we mean that we look at certain (numerical) features of a landscape, such that it can be compared to other landscapes or to gain understanding of valid search behaviours within the landscape. There are numerous features to be found, and a few will be discussed now.
An easy starting point for a feature of a landscape is the notion of local optima. A local optimum is a point in the landscape, for which all neighbours have a smaller fitness value then the local optimum itself. However, it is also possible for plateaus or ridges to occur. This is when there is a connected set of points with the same fitness value, while all neighbours of all points have lower fitness value. If this isn't the case, we speak of a saddle point. The amount of local optima forms the first measure. The frequency of local optima in the size of the search space is called the modality, and can be used instead of the amount of local optima. Furthermore, it is possible to analyse the distribution of the local optima. Furthermore, the distribution of optima can be compared to their fitness value, for example to see if optima near the global optimum are fitter than those further away. [75]
The next measure that can be used are the so called basins of attraction. Basically, when considering a minimization problem, it is the area from which the optimum lies downhill. This is not as obvious at it sounds. From any given point, there may be many ways to move downhill. Therefore, we split the basins into basins of strong attraction, and basins of weak attraction. In the strong attraction, any form of downhill leads to a single optimum, while in the weak attraction multiple optima can be reached. From the basins, we can extract a number of measures again. For a single basin, we can determine the fraction of the search space contained within it. We can also calculate the convergence volume, which is the sum of probabilities to reach a certain optimum from each point in the space. In general, we can also look at the minimum, maximum and average diameter of basins, and we can look at the relation between basin size and basin depth. [59]
Another measure is the notion of fitness barriers. This terminology stems from physics, where an energy barrier needs to be crossed to reach a metastable state from another one. For fitness landscape analysis, a fitness barrier is the minimum value necessary when traversing from one optimum to another. The fitness barrier is thus defined between a pair of optima. This does not take into account how far optima are separated from each other or the fitness difference between the two optima. An advantageous property of fitness barriers is that they adhere to the triangle inequality, as well as forming a metric [66]. If it is possible to compute all fitness barriers, a barrier tree can be constructed which depicts the hierarchy between the optima. [73]
A very important class of tools for fitness landscape analysis is the notion of walks. Walks are continuous movements from a starting point to another point. Depending on which neighbour is chosen to walk to at any given point in the walk alters what kind of walk it is. A non-exhaustive list of walks is given below:

- A random walk, where each neighbour is chosen randomly.
- An adaptive walk, where each neighbour has to be at least as fit as the current node. Depending on how the choice for a fitter neighbour is made, we can distinguish any ascent, steepest ascent, and minimum ascent walks.
- A reverse adaptive walk, where each neighbour is less fit than the current node, very much alike to the adaptive walks.
- An uphill-downhill walk, where we start with an adaptive walk until an optimum is achieved, followed by a reverse adaptive walk from this optimum. These can be chained together.

- A neutral walk, where every neighbour has equal fitness and it is attempted to increase the distance from the starting point.

As each walk has very different properties, it is generally a good idea to use multiple walks to paint a picture of a landscape. No walk is superior over another walk, as each explores a section of the search space which is interesting for some reason. If the search space is too great for an exhaustive detailing, sampling techniques are often used. For unknown problems, this can be dangerous, but there is not always a choice. In such cases, a population of individuals is formed and each individual iteratively walks along the landscape. In each iteration, the average progression characteristics can be computed. It is also possible to use a genetic algorithm tuned towards exploration to explore the fitter parts of a search space while avoiding the unfit areas. [60] [26] [48]
Then there is the notion of ruggedness, which was one of the first measures of problem difficulty in landscape analysis. In its most basic form, it is the frequency of changes in slope from rising to descending. Related to ruggedness are autocorrelation and correlation length. Autocorrelation is the correlation of neighbouring fitness values along a random walk on the landscape. It has been shown that this can be used as a measure of problem difficulty for certain classes of problems under certain assumptions. Correlation length is the average distance between points until they become uncorrelated. There is no one definition when points become uncorrelated that is maintained; multiple definitions have shown to be valid. A common definition is to take the inverse of the autocorrelation. There is also the Box-Jenkins model, which uses an autoregresssive moving average process to average over a white noise series. This results in a model of the landscape, instead of a single number, which provides more depth to the analysis. These measures are measured over random walks, though. When using an adaptive walk instead, the correlation levels will show a more correlated and more smooth landscape. [79] [10]
Similar to this notion of ruggedness is the notion of information analysis. The idea is to measure the amount of information required to describe a random walk. If more information is required to describe it, the landscape must be quite difficult. Instead of using the fitness values themselves, the information is compressed into positive slopes (1), negative slopes (-1) or almost flat slopes (0). A random walk can thus be described as a series of symbols. This makes it possible for a few analysis techniques to be applied. Information content is the number and the frequency of different combinations of consecutive symbols. Partial information content is the number of changes in slope, mostly measured relative to the length of the random walk. Information stability is the largest difference in fitness value along the random walk. Density-basin information measures how smooth the landscape is by measuring a series of consecutive symbols. [76] [9]
Ruggedness, local optima analysis and information analyses are techniques which provide a local understanding of the fitness landscape, but not a global understanding. One technique which solves this issue is the fitness distance correlation. A prerequisite of this technique is that the global optimum must be known. Then, the correlation between fitness values and distances to the global optimum from a representative sample of the landscape is computed. Due to the simplicity of this measure, it has been widely used. Whilst boiling down the landscape to a single correlation number bears with it a lot of information loss, it proves to be a good indicator for problem difficulty. In cases where the number does not give ample insight into the problem difficulty, it is better to look at the distribution of fitness versus distance. [30] [51]
Some measures require a different perspective on the fitness landscape as a whole. An example of this is the popular perspective of perceiving the landscape as a graph and performing Fourier Analysis. In spectral graph theory, the landscape is transformed using the Laplacian. For each vertex, the eigenvectors are calculated and can be used as weights for the fitness values. As the adjacency matrix is symmetrical, the eigenvectors form an orthogonal base. The transformed landscape is called the elementary landscape, and has the property that the fitness value of each vertex can be calculated as a weighted average of its neighbours. The elementary landscapes are easier to analyse than the original landscape. Spectral landscape analysis forms a powerful basis for other analysis techniques, but it is quite complex in nature, which makes it hard to automate. [80] [71] [72].
A term often used when considering evolutionary algorithms is evolvability. This is the chance for a population to produce offspring that is more fit. Many algorithms use techniques to ensure evolvability is high, such as random restarts and temporarily allowing worse solutions. While this term is mostly

used for search heuristics, it can also be measured for landscapes in two ways. The first is called evolvability portraits. This technique measures the evolvability for each candidate solution. Using this, metrics can be formed to depict the average expected offspring fitness, or the top or bottom percentiles. When comparing the evolvability portrait of a single solution to other solutions with the same fitness, an important perspective can be captured. The other technique is the fitness cloud. In this technique, each solution parent candidate chooses a single offspring. Their respective fitnesses are then plotted in a scatterplot and forms the fitness cloud. From this cloud, meaningful measures can be extracted. It is also possible to experiment with different ways of selecting the offspring for each candidate, like best neighbour, worst neighbour, or most average neighbour. These decisions provide meaningful new perspectives on the landscape. [2] [69] [57]
The next measure is concerned with the neutrality of the landscape. With neutrality, we mean connected areas of the landscape with the (roughly) the same fitness. These areas makes it possible to achieve a much wider range of candidate solution during an uphill walk. Neutrality plays an important role in natural evolution, and is thus also considered for evolutionary computing. For each neutral area, features can be extracted (for example, size) and over all neutral areas, distributions can be taken. These distributions form the measure of neutrality of a landscape. [32]
The final measure that is often used is epistasis. Epistasis is the impact that genes have on each other. This definition is ambiguous, but the concrete measures are defined unambiguously. Epistasis can be quantified with epistasis variance. This technique collects a sample of solution candidates and attempts to explain their fitness values with a liner model. Then, an analysis between the model and the actual fitness function reveals how much variance is explained by the linear model. The rest of the variance is assumed to be caused by epistasis. This measure works in some cases, but fails in other cases. This method is critiqued, partly because it does not distinct different orders of non-linearity. This is solved in the graded epistasis method, which models some levels of non-linearity as well. These methods all don't directly measure epistasis, but measure the absence of epistasis and calculate the impact of epistasis. This measure provides insight in deceptiveness of problems. [16] [61]

In recent years, not much has changed in the field of landscape analysis. The most exploration has been done in applying landscape analysis on different problems. For example, Yafrani et al. applied fitness landscape analysis to the traveling thief problem, with two different hill climbing algorithms [85]. Some of the measures explained above have been improved upon, like the multiobjective differential evolutionary algorithm for ruggedness as proposed by Huang et al. [27].

4 Research Methodology

To provide an answer to the research question, we will perform multiple experiments. These experiments are aimed at exploring the fitness landscape generated by the PyGGI GI algorithm on the Bears benchmark programs. There are two kinds of experiments. In the first kind, a random walk will be performed to explore the fitness landscape as a whole. In the second kind, more focus will be drawn to the large plateaus that functional GI has to deal with. For this reason, both experiments use a functional improvement GI algorithm. In this section, we will explain how the algorithms were implemented into PyGGI and which Bears programs have been selected to perform GI on. At the end, we will highlight some common problems with GI algorithms and how we managed to overcome these.

4.1 PyGGI algorithms

We will firstly look at the GI algorithms that have been implemented into PyGGI and any other changes that have been made to the PyGGI framework to suit the needs of the experiments. We will first examine the parts of the program that have been altered for the experiments but are the same between the two algorithms. Then we will examine the GI search algorithms in more detail, starting with the random walk algorithm.

4.1.1 Custom Alterations

The PyGGI framework has undergone some alterations to suit the needs for this thesis, and thus be able to run the experiments we want to run. We will explain how and why these edits were made and how it affects the framework as a whole.

First of all, and most importantly, a connection between the PyGGI framework and the Bears benchmark had to be created. This connection allows us to retrieve the Bears program on command so it can be edited. Luckily for us, the Bears benchmark comes with three python scripts: A script for checking out a program, a script for compiling a certain program, and a script for running the test of a certain program. Given that all programs in the Bears benchmark are compiled and tested using Maven, these commands could be generalised into a python script. The script that checks out a program just uses some git command to check out a specific branch containing the program and copies it to a certain file location.
The scripts, while very helpful, were not suited for Windows with the command line commands used. Therefore, these scripts had to be edited to achieve the same effect. The maven and the git commands are the same between Windows and Linux, but there were also commands that interacted with the file system. For example, the check out script copies the program to a given location. Furthermore, there were differences with how multiple commands are executed right after another. The check out script was edited to be called from the PyGGI program, while the compile and test scripts only had their command incorporated into the PyGGI framework. As for these scripts, only the actual Maven commands were necessary for our purposes, which is why these are directly called from the framework itself. This has been integrated in the PyGGI config, by editing the "test command" variable.
By using the Bears scripts, we can check out and test any Bears program that we desire. These programs do need to be stored at some location. This location can be added to the PyGGI framework when it is called from the command line.
Lastly, we edit the computation of the fitness score. The demo PyGGI program is not compatible with the format in which Maven exports the results of the test. For this, we first detect whether the compilation of the program was succesful. if not, we do not calculate a fitness value and return while setting the result status to `'ERROR'`. If the compilation was succesful, we observe the amount of tests that result in success, the amount that result in failure, and the amount that result in error. As Maven also reports the amount of tests that have been run and use that metric as well. The fitness value is computed as detailed in Equation 2. This results in the following code, and is the final part of the connection between PyGGI and Bears.

```
def compute_fitness(self, result, return_code, stdout, stderr, elapsed_time):
    try:
        tmp = re.findall("Tests run: ([0-9]+),", stdout)
        run    = int(tmp[len(tmp) - 1])

        tmp = re.findall("Failures: ([0-9]+),", stdout)
        failed = int(tmp[len(tmp) - 1])

        tmp = re.findall("Errors: ([0-9]+),", stdout)
        error  = int(tmp[len(tmp) - 1])

        fitness = (run - failed - error) / run
        run = 0

        result.fitness = fitness
    except:
        result.status = 'ERROR'
```

Listing 9: The method that calculates the fitness value for a given program, given that it compiles.

The next alteration to the original PyGGI framework is the translation from the java files to XML files. This is done using srcml, which takes any given source file and translates it into the XML format we require for our statement mutations. This requires us to find all the source files in a program. To achieve this, we created a recursive function that takes a base folder and collects all .java files from that folder, and then recursively calls the function on all subfolders. This function can be found in Listing 10.

```
def get_files_rec(path):
    result = []
    currentdir = os.listdir(path)
    for f in currentdir:
        if isfile(path + "\\" + f):
            if f[-4:] == "java":
                result.append(path + "\\" + f)
        else:
            result = result + get_files_rec(path + "\\" + f)

    return result
```

Listing 10: The function used to recursively collect all Java files.

Now that we have collected all .java files, we only need to convert these to a .xml file. Like mentioned before, this is done using srcml, which can be called from the command line. Therefore, for each file we have found, we execute a certain command: `os.system("srcml %s -o %s" % (f, new))`. In this command, `f` is the original .java file and `new` is the name of the new .xml file. The `-o` tag tells srcml that we want to output from a source file into a .xml file.

The next thing that needs to be done is to tell PyGGI what the target files are. This is also something that can be specified in the PyGGI config. Here we make another edit to the original framework, by not only specifying the target files, but also the ingredient files. We have added this to the config of PyGGI. For the ingredient files, we have already kept track of all the source files in the program and can supply these to the framework easily. As for the target file, this is a manual process. In each Bears program, there is an extra file named bears.json, which contains some meta data about the program. This includes the amount of failing tests, the test files in which a failure occurs, and the type of failure that occurs. However, it does not provide a location in the original code where the error has taken place. This means that we cannot automatically find where the target file is. Thus, we need to do this manually. Because we do know which tests fail, it is not much work at all to find the file in which a bug is present. Furthermore, this allows us to be even more precise in where the algorithm should try to fix the error, and pinpoint it to a single or multiple target methods. These methods get added to

the config, just like the ingredient files are.
PyGGI itself does not provide a way to add the ingredient files to the config, so this functionality had to be built into PyGGI as well. To do this, we follow the same logic as the target files follow, but ensure that the ingredient can be chosen from the ingredient files, instead of the target files. Thanks to the structured way PyGGI is built, this was a relatively easy addition to its functionality.

Next to these major functionality additions, there were some small adjustments made to the base code as well. First of all, there was a small edit to the PyGGI base code to make the command line commands return properly, by setting the `shell` variable to `True`. Furthermore, there were some quality of life updates that have been performed to ease running multiple experiments. To ensure that every time the experiment is run, a fresh copy of the original program is downloaded, the old version gets deleted. Because PyGGI always creates a log of the experiments, the actual results of the previous experiments are saved and only the program itself gets replaced with a fresh copy. All these edit to the PyGGI framework ensure that we can run the experiments we wanted.

4.1.2 Random Walk Algorithm

Now we come to the first search algorithm that we have implemented into the PyGGI framework. In PyGGI we need to define three methods to actually implement a search algorithm: A method that returns a neighbour from a certain patch, a method that determines if the search algorithm can be stopped, and a method that determines when a patch is better than another patch. The last method is useful in determining how ties are broken and, in our case, help with achieving a random walk.
The method that returns a new neighbour from a certain patch is a very simple method, which just selects a random edit and adds it to the patch. It is not possible to remove any edits from the patch in the random walk.
The method that determines the stopping criterion always returns `False`, as we do not want the GI process to terminate early. This criterion is not related to the amount of iterations we can give to PyGGI. This would only be an alternative stopping condition, which is not what we want.
The last method determines when a patch is considered to be the new base patch to mutate from. As we are performing a random walk we want to accept all valid patches. The invalid patches never make it to this method, as they are immediately rerolled to a patch that is valid. This means that we can simply return `True` to accept all valid patches.

4.1.3 Plateau Algorithm

The second algorithm is the algorithm that navigates the plateau on which the original program lies. This means that we want to explore the neighbours of the starting solution and examine those that have equal fitness. Ideally, we want to find the edges of the plateau, so that we can find a sense of how large the plateaus are. We have found an edge if we obtain a solution of which all neighbors either have different fitness as the solution, or have already been visited before. To achieve this, we have chosen to take a sample of the neighbourhood for each solution, as it is not feasible to search the entire neighbourhood. We do keep track of the patches that have been evaluated, and hope to find the edges of the plateau.

Because this algorithm is fundamentally different to the first algorithm, we need to make changes to the run method of PyGGI as well. In particular, we add a queue which starts with the original program in it. In each iteration, one program is dequeued and is called the current program of the iteration. From this program, ten valid neighbours are randomly generated and evaluated. Those programs that have a different fitness value than the current program are discarded, as they are not part of the plateau. The other programs are added to the queue, if their patch is not equal to one that has already been observed. This process ends if the queue ever ends up empty, or after eight hours.

The methods that define this algorithm are edited in the following way. The method that gets a neighbour is equal to the original PyGGI framework, as we do want removals of edits to be possible. These may result in unvisited patches.

The stopping criterion is different from the one in the first algorithm. Here we return `queue.Empty()`, because we want to stop the algorithm when the queue is empty. The stopping criterion of eight hours is one that is enforced in the run method, where the criterion of iterations has been replaced. Instead, the time is kept track off and the algorithm stops once the eight hour mark has been reached.
Finally, the method that determines if the current patch is the best patch is removed in this algorithm. This method has been replaced with the queue, which determines which patch is the current patch in an iteration. Therefore, this method has been removed, which concludes this algorithm.

4.2 Bears programs

The Bears Benchmark contains 251 different bugged programs. While this provides great variety in possible experiments, it is infeasible to perform an experiment on all these programs. This is why a selection has been made from the benchmark. The chosen programs are as follows:

- Bears-188 with the least amount of tests (7).
- Bears-237 with the greatest amount of tests (8606).
- Bears-44 with the average amount of tests (1029).
- Bears-27 with the median amount of tests (968).
- Bears-204 with the greatest amount of failing tests (10).
- Bears-121 with the greatest amount of tests in error (145).
- Bears-225 with both failing tests and tests in error.

The first four programs were chosen to analyse if the relative difference in fitness makes any difference in the search space. A difference of 1 test failing makes a greater relative difference than the same situation in Bears-237. While this itself should not make a difference for the algorithm it is reasonable to assume that the tests of Bears-237 would start failing after a smaller change than the tests of Bears-188.

Bears-204 and Bears-121 were selected to observe a program where a multi step solution to the global optimum is more probable, as the tests can be solved with different mutations. This might also make it more probable for local optima to occur, which is analysed by the second algorithm.

The final program, Bears-225, is selected for its uniqueness in having both failing tests and tests in error. This is the only program in the Bears Benchmark to exhibit this behaviour and might make for interesting results because of this.

Now that the programs have been chosen, we will disclose some interesting characteristics from these programs. First of all, while explaining how big the search space is, we analysed it by using a variable n. We will now show how big the search space actually gets by filling in the n for the chosen Bears programs. Note that for the count of the line mode, we only counted lines with content. Any whitespace has been ignored. Furthermore, for tree mode we counted all individual tags as a point on which a mutation is possible. This results in the following counts.
The table also shows how many tags get generated for each line of code. From this we can see that going from line mode to tree mode, the amount of mutable components grows by about a factor of 10. This does not mean that the search space itself also grows by a factor of 10, because each mutation only affects similarly tagged statements.

Next to the size of the search space, there are some more interesting characteristics. For instances, the structure of Bears-237 is different from other programs that were viewed thus far. This is a repository that consists of different individual programs that work together. Each program has its own set of tests that can be run individually. This means that while the repository does indeed have the greatest amount of tests, only the ones for the bugged program would realistically have to be run. This finding

Bears Program	Line mode	Tree mode	Tags per line
Bears-188	687	7783	8.98
Bears-237	125112	1138081	9.10
Bears-44	56131	750468	13.37
Bears-27	54375	734919	13.52
Bears-204	1919	24750	12.90
Bears-121	46973	780620	16.62
Bears-225	83214	904098	10.86

Table 1: A table showing how large programs can get, and how problematic that can be for GI.

skews the greatest amount of tests, but also the median and mean amount of tests, as the tests are counted per repository instead of per program. We continued with the chosen programs either way, as the programs are still quite different from each other and still provide meaningful insights into the fitness landscape for different programs.

4.3 Reducing Search Space

The search space for any given program is interesting to analyse as one can expect it to be very large. This is in fact one of the major problems of GI. Let us consider the search space of the GI algorithms used in this research. As both algorithms use the same mutations, the actual search space itself is consistent between them. The only difference is the choice of which program variant to use in the next iteration. For each mutation, we will analyse how many possibilities there are of applying it to a random program consisting of n code statements.

The first mutation is the replace mutation. This mutation takes a random statement and replaces it with another randomly chosen statement. As these statements are chosen independently of each other, this results in a search space of size $O(n^2)$.

The insert mutations works similarly to the replace mutations, but it inserts the new statement before or after the chosen statement instead of replacing it. There are $n + 1$ locations to choose to insert the new statement, of which there are n, which means that the insert mutations also creates a search space of size $O(n^2)$.

The delete mutation selects a random statement and deletes it. There are exactly n statements to select, so the delete mutation creates a search space of $O(n)$.

Considering all the search space sizes mentioned above, any random program has $O(n^2)$ neighbors. This is fine for very small programs, but one could imagine this being a problem for larger programs. This is why we have chosen to optimize the search space in a number of ways.

Firstly, the information files from the Bears Benchmark have been used to determine which parts of the code contain the bug. Usually each Java code file contain a single class, and for each class a dedicated test class has been created bearing the same name. In these cases the name of the test class is automatically converted into the name of the Java file with the bug and only this file would be inputted into the GI program. This raises the question whether only the target should come from the specific file, or if the ingredient should come from this file as well. On one hand, it could be argued that it is more likely for a correct ingredient to be contained in the same code file as the target, but this might not necessarily be the case if the code is broken down into many files. For the purpose of this thesis, we have chosen to consider all files for the ingredient, as this limits the chances of not being able to find the global optimum at all.

Another optimization in terms of search space is by limiting the tags which can be used for mutations. For example, it is improbable that a using statement is the source of a bug or the solution to

a bug. Furthermore, we assume that headers for loops are correct. Manual inspection of the chosen programs proves this assumption to be correct for the current experiments. Still, in general this assumption can be made, as loop headers tend not to contain bugs. The statements that are available to be selected for mutation are: if statements; declaration statements; expression statements; return statements; and try statements.

4.4 Improving Performance

One limiting factor of GI is the performance. Due to the great search space available, it would be beneficial to limit the computation time necessary per iteration. This has been attempted in a few different ways.

Firstly, it was attempted to reduce the actual time required per iteration. The compilation is optimized as it keeps track of which files have been changed and only compiles those files. The test time would have been possible to optimize if an improved program is what we were after. In this case, only the previously failing tests had the be run in the first case. Only if these tests passed, would it be necessary to run all other tests to ensure those still pass as well. In repositories with hundreds or thousands of tests and only a few failing tests, the speedup would probably be significant. However, as we are analysing the fitness landscape, it is important to perform all tests at all points to obtain a valid impression of the search space. Thus, unfortunately, the actual computation time per iteration could not be optimized.

Another approach to optimise the computation time is with concurrent calculations. The first attempt at this was by using Google Colab to create multiple notebooks of the same project and running these simultaneously in the cloud. The problem with this is that the PyGGI framework relies heavily on being able to use the file system to create temporary variants of the initial program. This was impossible to recreate with Google Colab on short notice, which made this approach infeasible at the time.

A method that was also considered was to use concurrent methods in python to speed up the entire process. This was also not implemented as it was considered easier to run the experiments overnight while working on other aspects of this thesis, like the analysis of experiments already performed. The speedup achieved by using concurrent methods would not be worth it when considering the time required to research and implement concurrent methods.

Finally, there is the possibility to simply run multiple instances of the GI program simultaneously. This optimisation has been applied by running up to eight instances of the GI algorithm overnight, significantly reducing the length of time required to finish all experiments. The individual experiments probably took longer to finish, but that is negligible when regarding the time gained.

Conclusively, while there were theoretical optimisations to be done, in the end it was chosen only to run the experiments overnight, simultaneously. This approach proved itself to be quick enough for the purpose of this thesis, and thus did not require other speedups.

5 Results

In this section we will provide the results of the experiments that we have run according to Section 4. We will first look at the experiments which performed a random walk, and discuss the results in general, and when comparing individual experiments. Then we will look at the experiments which travel over the starting plateau, and discuss those results.

5.1 Random walk

For the experiments where a random walk has been performed, we will look at general and individual results. We will first look at the overall results of the experiments, after which we will discuss the individual experiments which were chosen to compare with each other according to Section 4.2.

The experiments that were performed by doing a random walk from the starting program could be split into two categories. In the first categories, no improvement could be found in any of the experiments. To be precise, the fitness valued stayed at the original fitness value. These experiments were Bears-237, Bears-44, Bears-204, and Bears-225. The fitness value of these programs were always equal to the fitness value of the original program, even after a random walk of 1000 iterations. This also points out the problem with the fitness landscape, which is caused by the plateau's that exist there. On further inspection of the human patch of these programs, it was clear to see why no improvement could be found. In all these programs, an entire code block had been added, sometimes even in files which were not the target file. For example, in Bears-225 an Equals override had been implemented into multiple classes. Adding an override method is not something we allowed our GI algorithm to do, so logically no improvement could be found. Furthermore, no degradation could be found because of the limitations to the target domain. No other tests could be invalidated because they do not test the code which we allowed our algorithm to edit.

The other programs, Bears-188, Bears-27, and Bears-121, show improvement in their fitness scores once. These programs start with a fitness of 0.857; 0.999; and 0.451 respectively. All programs improved to a fitness value of 1 afterwards. Because of the limitations to the target domain, no degradation to the original fitness value was found. More interestingly, there was a consistency in finding a mutation that improved the fitness score. In all fifteen experiments the improvement had been found, and in 11 experiments this was found within the first 250 experiments. This shows that even a random walk can find a solution if the solution is feasible in a relatively short amount of time. In the table below, we have detailed the amount of iterations it took during each of the five runs to find an improved solution. Observe that the edit distance is the same as the first iteration that found an improved solution, as we only add edits to the sequence.

Experiment	Bears-188	Bears-27	Bears-121
1	96	179	259
2	113	162	253
3	125	244	178
4	108	179	247
5	78	178	219

Table 2: The first time the improved program has been found after a random walk in each of the five experiments.

Now we look at the individual results of the experiments. We have chosen Bears-188, Bears-237, Bears-44, and Bears-27 because of a varied amount of tests, to determine what the effect of the amount of tests is on the runtime and efficiency of the GI algorithm is. Below is a table depicting the runtimes of these programs for all five experiments in seconds.

Run	Bears-188	Bears-237	Bears-27	Bears-121
1	39891	35974	39030	27440
2	26878	37018	30880	30490
3	31765	35828	38632	29626
4	28935	30832	36964	37159
5	37596	27437	39279	29789

Table 3: The runtimes of the experiments measured in seconds on the programs with varying amounts of tests

These times look pretty similar to one another. While there are experiments that generally took longer than other experiments, we can not judge by looking at it if there is a significant difference in the runtime of the different experiments. For this reason, we have deployed a student t-test to determine whether there is a significant difference between any of the Bears programs in terms of run time. The 95% confidence intervals are shown in Table 4, which shows us that there are no significant differences in the runtimes between the different programs. As these experiments have a vastly different amount of test cases we would expect the test times to differ, but there is not effect on the final runtime.

	Bears-188	Bears-237	Bears-27	Bears-121
Bears-188	X			
Bears-237	[-7547.79, 6738.19]	X		
Bears-27	[-10737.16, 2849.16]	[-9121.10, 2042.70]	X	
Bears-121	[-4773.24, 8997.64]	[-3176.85, 8210.85]	[-2427.34, 10539.74]	X

Table 4: The results of performing the student t-test on the results of Table 3

The next set of experiments, Bears-204 and Bears-121, were chosen because they had multiple tests failing and erroring respectively. However, as we have already stated, this did not seem to have an effect on the way the fitness behaves during the random walk. For this, we need to look at why multiple tests were failing and erroring.
In the case of Bears-204, the errors that were occurring were all assertion errors, where a certain response message was not the response message that was expected. Given that the response messages were of the correct type, and that the human patch overrides the Equals method for all of these messages, we can assume that there was a problem with when a message was considered equal to another message. The solution to this was not found by our GI algorithm, as it was unable to add an Equal overrider for any of the message types.
On the other hand, for Bears-121 all errors were caused by a NullPointerException. As we can see in the human patch, an extra requirement was added to an if statement which could solve all the errors at once. This improvement was also found by the GI algorithm, meaning that all tests were succeeding. The fitness value thus went from 0.45 to 1 in a single iteration, instead of the step by step improvement we were aiming for.

Finally, we chose Bears-225, as it had both tests resulting in errors and tests resulting in failure. There were two Assertion errors and one NullPointerException in this test suite. Unfortunately, the GI algorithm was not able to solve either of these issues. Looking at the human patch, it seems there was an if-statement added to prevent the NullPointerException, and part of a while-loop header was removed. While the GI algorithm is not to add an entire if-statement, it is capable of editing a while loop. However, this improvement was either not found, or was not seen as an improvement by an increase in fitness.

5.2 Plateau

The second set of experiments explore the plateau on which the starting program exists. The algorithm searches for a subset of its neighbours for programs with equal fitness. In all seven of the experiments,

the eight hour limit of searching was reached before the edges of the plateau could be found. The amount of iterations that have been visited is shown in this table.

Bears Program	Correct programs	Parse errors
Bears-188	903	2270
Bears-237	1375	3417
Bears-44	1131	2804
Bears-27	1067	2614
Bears-204	974	2435
Bears-121	994	2480
Bears-225	1093	2719

Table 5: The amount of correct programs and programs with parse errors found after 8 hours of running

Observe that the amount of iterations is not as high as one would perhaps expect. However, when comparing these results with the first set of experiments, the amount of iterations seems to be in line with the amount of time spent in those experiments of 1000 iterations. In both sets of experiments, programs that resulted in a parse error were not counted towards the total, while taking up a significant amount of time. Among all experiments, there was an average of 2.5 programs with a parse error for every program without a parse error, and this average seems to be very consistent among all experiments. The reported amount of programs with a parse error in these experiments is also shown in Table 5.

Unfortunately, in no experiment was the edge of the plateau found. This means that we can only say for sure that there are at least a thousand different iterations of a starting program that gives the same results on all tests. Note that while these programs are all unique in the sense that they have a different edit sequence, it is entirely possible that multiple edit sequences lead to syntactically the same program. Also, different programs can have the same behaviour while not being identical.

6 Conclusion & Discussion

In this section we will review the results we have found and see what conclusions can be drawn from this. Furthermore, we will evaluate how our results compare to other results in the field and discuss the implications our results have on the field. Finally, we will provide an answer to our research question. In this section, we will look at the two sets of experiments and compare the individual results to the field, after which we will reflect on the research question.

6.1 Random walks

In the first set of experiments, there were two results observed. Either the fitness value stayed the same from the original value, or the fitness value increased once. We will examine these experiments separately, starting with the experiments where the fitness value did improve once.

In these experiments, the fitness value only takes one of two fitness values: one value where the targeted test passes and one where that test fails. This can be explained by the performance improvements implemented, where only the code that contains a certain bug is targeted, as explained in Section 4.4. Even the experiments where multiple tests were failing, only two fitness values occur in the results. As we reported in the results, this is because the multiple test failures were caused by the same bug and could therefore be solved by the same mutation. This behaviour of only having two fitness values can be explained by our approach to the GI algorithm.
In our approach, we have chosen to only target the method in which the failure of a test is caused. In this method, the bug is presumably contained. When we look to other researches in the GI field, we see that this is not uncommon to optimize on. For example, in the GIN framework (GI in no time [81]), the target method is found in a pre-processing stage, and then applies edits only in this target method [12]. A case study with the Gin tool proves that the tool is suited for finding improvements in programs in an acceptable amount of time, as long as the test suite offers an accurate representation of the actual software behaviour [54]. Reflecting on our optimization, we conclude that this optimization is valid and does not limit the exploration of the fitness landscape. However, it may be possible that the test suites for the Bears programs do not provide an accurate representation of the software behaviour. This results in the found improved patches may not actually retain the original behaviour of the program.
The question still remains whether our results tell us something about the fitness landscape that can be used in practice or in future research. We have found that when an optimization can be found using GI, it is often found after a limited amount of iterations. This is in line with what other researches have found recently [55] [8] [54]. When we look at how functional improvement is implemented in practice, we see that the proposed patches from GI are often small and found without using sophisticated search algorithms [46] [23]. Perhaps this is all that is possible regarding functional improvement, as even the most recent GI research reports that limited success can be achieved using basic algorithms [8]. As long as there are no successful ways to find larger patches to known bugs, GI will not be able to solve larger bugs and this remains to be done by developers.

We think that the fact that GI is able to find the fixes for small bugs is a logical result if we compare the GI process to a chance game. We assume that the iterations that precede a successful iteration do not impact the successful iteration. So, the GI process is a series of independent experiments where a certain mutation is being looked for. We will analyse the GI process this way by using Bears-188 as an example. The mutation we are looking for is the deletion of one tag among a total of 122 tags. The chance that the deletion mutation is chosen is 1/3, because the mutation is chosen first with equal chances. It is also possible for the replace mutation to achieve what we want, which also has a chance of 1/3 to get chosen. Assuming any other tag replacing this particular tag leads to an increase in fitness, that leaves us with a chance of $\frac{2}{3} \cdot \frac{1}{122} = \frac{1}{183}$. This means that we expect the mutation to be found after 92 iterations. In practice, the improved version was found after on average after 104 iterations, which is slightly more than what we calculated. This is probably due to the assumptions that we have made that the iterations are independent, but we can conclude that this analogy works.

From this analogy we can conclude that the GI process of functional improvement can be compared to a series of independent chance games. To further solidify this analogy, we propose that the individual patches of the GI process leading to an improved version of the program are examined in-depth. Furthermore, a GI algorithm with a random walk should be compared to a GI algorithm with a best improvement local search algorithm, and see if there is a significant difference between the amount of iterations it takes to find the improved program. If there is no significant difference, this tells us that the analogy is valid.

Next up, we will evaluate the random walk experiments where no improvement was found at all. This can be explained by examining the human patch that fixed the bug in the actual repository. In all of the experiments, a different mutation was required to achieve the required patch. Furthermore, the human patches were also quite large. These have to do with each other, because the GI algorithm was not able to create new override methods, nor create entirely new code blocks. These things are both quite large in execution, and also require some change to the mutations to be able to be found at all. The addition of such mutations is probably not wise, as the patch required still is a large one which makes it improbable to find either way.

The first type of experiments are programs that can not be improved using the current GI framework, because the corresponding human patch is too large. These patches are too large to be found without having some sort of intermediate result first. These programs require a different approach to be able to improve, which is not in the scope of this thesis. It is important to realize that GI isn't a failsafe method that can be used to improve any program. A certain set of requirements have to be met for GI to work, but not all of these requirements are plain to see. In this case, the patch size needs to be small enough for GI to find an improved patch in a reasonable amount of time. An example of such a human patch is given below.

```
190      @Override
191      public void removeStatement(CtStatement statement) {
192          if (this.statements != CtElementImpl.<CtStatement>emptyList()) {
193  +
194  +           boolean hasBeenRemoved = false;
195  +           // we cannot use a remove(statement) as it uses the equals
196  +           // and a block can have twice exactly the same statement
197  +           for (int i = 0; i < this.statement.size(); i++) {
198  +               if (this.statements.get(i) == statement) {
199  +                   this.statements.remove(i);
200  +                   hasBeenRemoved = true;
201  +                   break;
202  +               }
203  +           }
204  +
205  +           // in case we use it with a statement manually built
206  +           if (!hasBeenRemoved) {
207  -           this.statements.remove(statement);
208  +               this.statements.remove(statement);
209  +           }
210  +
211               if (isImplicit() && statements.size() == 0) {
212                   setImplicit(false);
213               }
```

Listing 11: The human written patch corresponding to Bears-44, which could not be found with GI due to the patch size

The second type of experiments are programs that require other edit parameters to achieve the corresponding human patch. These patches clearly can not be found using the current GI setup, but requires the implementation of additional edits. Implementing additional edits may result in more programs being improvable with GI, but each additional edits also vastly increases the search space. This in turn

reduces the efficiency of the GI process itself. This is a trade-off that has to be made by the developer of a GI algorithm, and often no further edits are implemented. An example of a patch that could not be found with the current edits is given below.

```
27          this.succes = success;
28          this.nickname = nickname;
29      }
30  +
31  +   @Override
32  +   public boolean equals(Object o){
33  +       if (o == null || !(o instanceof LoginResponseMessage)) {
34  +           return false
35  +       }
36  +       LoginResponseMessage other = (LoginResponseMessage) o;
37  +       return success == other.success && nickname.equals(other.nickname);
38  +   }
39   }
```

Listing 12: The human written path corresponding to Bears-204, which could not be found with GI due to missing edits

In this case, the override, and the addition of a method in and of itself, is something that the GI algorithm is not able to reproduce. As all test cases failed on the default implementation of the equals method, the only way to improve the existing code is by implementing this equals method.

In the current applications of GI, the amount of bugs that could not be found with GI is not really reported on. In the application of SapFix, we do see that the patches from SapFix are the correct fix in 48% of the cases [46]. We can conclude from this that in the other cases, SapFix was unable to come up with the correct fix for a bug. This means that our results are similar to the results from GI in practice, in the sense that we do not always find the correct fix.
We have also compared our results to the theoretical research in the area. For example, in a research by Ackling, Alexander and Grunert, a different approach was taken to GI with an abstract syntax tree. The way the code is broken down into the abstract syntax tree is different from our approach with srcml, but the main difference comes into play when we consider the mutations. There is a bigger emphasis on creating mutations that are syntactically valid as opposed to the research in this thesis. However, the results are quite similar to our thesis, where small bugs get fixed in a reasonable amount of time. The actual time spent is longer in this thesis, which can be explained by the time it takes to test and the less effective search algorithm. The result from Ackling et al. could also not be used on larger bugs. [1]

The next result we will reflect on is the runtime of the different experiments. As we can see, there is no significant difference between the runtimes of the Bears programs, even though they have very different amounts of tests. We can assume from this that the amount of tests is not actually a significant part of the GI process. The running of the tests takes about 10 seconds, which is comparable by the research by Petke and Brownlee [54]. There is some difference in runtime of the tests, but not a significant amount according to our results. We can conclude from this that an optimization to the runtime of GI should be either focussed on limiting the runtime of the tests entirely, by not using Maven for example, or by optimizing in other aspects of the GI algorithm. If we assume that each test batch takes 10 seconds to complete, and the test batch is performed 1000 times, it will take almost 3 hours to complete in total. This accumulates to about a third of the total runtime of GI. Finding further runtime improvements is something that should be researched in the future.

When reflecting on the experiments as a whole, it would have made more sense to not use the Bears benchmark, or use different programs from the benchmark. While it provided for easy automation of the GI process and accessible programs, the programs that were selected did not always behave as we expected it to, or would make for fair comparisons. It would probably be better to create programs that do behave like we want. Another option would be to take a working program, and to manually

introduce bugs in different locations that can be solved by GI guaranteed. This also makes it possible to introduce multiple bugs in different locations and see what the behaviour is like when the bugs are partially solved.

6.2 Plateaus

Regarding the experiments aimed at exploring the plateaus, a few conclusions can be drawn. First of all, it is clear that the plateaus have not found a boundary in the running time of the experiments.

The runtime of the GI algorithm is not quick enough to explore an entire fitness plateau. Similar to the chance game analysis used with the other experiments, there are a number of mutations that do not change the syntax of the program. If this analysis turns out to be correct, any combination of these mutations result in a program with identical syntax and thus are part of the plateau of the original solution. The amount of possible combinations grows rapidly as the amount of mutations grows.

What this conclusion tells us, is that it is important to find mutations that do have syntax altering effects, or find a way to avoid these mutations. Without such precautions, a lot of the search space is equivalent to the starting solution. One possible way to achieve this is by logging the results of the test cases and ensuring that at least one test case provides a different result than the previous iteration. With such a method, one must be careful to prevent overfitting by adding too many test cases, and with extending the running time of a single iteration. Another way of achieving this is by using program transformation techniques to construct all equivalent programs to the initial program and check that a mutation does not belong to this set. This method does consume a large amount of memory and should probably be tried on smaller programs before trying to extend it to larger programs, to have a sense of achievability.

The result that the plateaus are large in size is in the same line as what is expected in other researchees. In the survey by Petke et al. it is often suspected that the plateaus with equal fitness in functional GI are very vast, and this is something we could confirm in our research. In the research by Veerapen et al. it was already concluded that local optimum plateaus exist by using a local optima network [53]. This research confirms this with a enumerative approach, and elaborates why these plateaus exist.

6.3 Research Question

Now that we have drawn our conclusions, let us once again look at the research question, which is "What insight can be gained from applying landscape analysis to functional improvement using Genetic Improvement on programs derived from the Bears benchmark?". The insight that we have gained can be summarized in the following points.

- Small bugs can be solved in a reasonable amount of time, even the random search algorithm, provided that the GI algorithm possesses the mutations required to fix the bug.
- There currently is no known way to solve large bugs using GI.
- The plateaus which make up the fitness landscape for functional GI are too large to traverse without using Program Transformation techniques.

When discussing these results, we showed how our results compare to those in the field of GI. Overall, our results are similar to results performed in different researches. One final observation to make is that this research focuses on the functional improvement aspect of GI, and not on the non-functional improvement. The non-functional fitness landscape is very different to the functional fitness landscape, where plateaus are not prevalent at all. Every mutation can and probably will have some impact on the non-functional property, whether it is memory, runtime or energy consumption. Within this research field, there are a lot more search heuristics possible which can not be applied to functional improvement. For example, a hill climbing algorithm is much more powerful in non-functional improvement, as shown

by Blot and Petke [8]. Furthermore, Langdon and Harman have shown that their GISMOE approach is very capable of searching through the non-functional fitness landscape and provide good results [37].

7 Future Work

After this thesis, there still is future work to be completed regarding fitness landscape analysis and GI. These suggestions for future work will be expanded upon in this section.

One possible alteration to the current research is the choice of the domain for the ingredient in the mutations. In this thesis, we have chosen to use all available code as the ingredients, but it might suffice to limit the ingredient to a certain file or a base set of ingredients. It is also possible within PyGGI to add weights to certain files or lines. This feature can be used in a number of different weight. It could be possible to statically add weights to certain files or methods. It is also possible to add weights dynamically, adding a penalty for ingredients that cause compile errors more often and adding a bonus to ingredients that lead to better solutions.

Another simple suggestion for future work is switching over to non-functional improvement of GI. We have chosen to examine the functional improvement of GI in this thesis because of the ease of use of the Bears benchmark and the interesting prospect of automated bug fixing. However, the non-functional improvement of GI looks to have a more varied fitness landscape where more search heuristics can be used. It has already been proven that an A* implementation exists in the non-functional improvement area, as well as the usage of multiple search heuristics [8] [12].

Another thing that might be worth looking into in the future is the PyGGI assumption that it is correct to change tags of the mutable statements to a single identifier. As information is lost in this step, it might make sense to handle different tags differently. This requires the rewriting of the available mutations to work with different tags. It also might open up some possibilities to allow certain tags to mutate with certain other tags. This opens up search space that was previously closed. The question remains whether it is worth opening up this search space, or that it will lead to relatively worse results.

During this thesis we have already mentioned some parts that might be changed for future research. For example, we suggested that program transformation techniques could be applied to small programs to significantly reduce the search space. This might make it possible to find the actual edges of the local fitness landscape if the memory usage is not too great. Research on this topic is already taking place within Utrecht University. For example, the thesis by Sprokholt indeed uses Program Transformation techniques to programs to speed them up, while being semantically equivalent. However, on larger programs the optimization still takes months to complete. [70]
We also suggested to take a single working program, and introduce bugs by hand in this program. In this way, we can use a controlled environment to examine how environments influence the GI process. We can be sure that a solution can be found with the GI algorithm and we can examine what happens when multiple bugs are present in different parts of the program. We can also tweak the test suite to see what influence that has on the GI process.

References

[1] Thomas Ackling, Brad Alexander, and Ian Grunert. Evolving Patches for Software Repair. 2011.

[2] Lee Altenberg. The Evolution of Evolvability in Genetic Programming. In K.E.Jr. KKinnear, editor, *Advances of Genetic Programming*, pages 47–74. M.I.T. Press, 1994.

[3] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1100–1104, 2019.

[4] Andrea Arcuri. On the Automation of Fixing Software Bugs. In *Proceedings of the ACM International Conference on Software Engineering*, pages 1003–1006, Leipzig, 2008.

[5] Andrea Arcuri. Evolutionary Repair of Faulty Software. *Applied Software Computing*, 11(4):3494–3514, 2011.

[6] Andrea Arcuri and Xin Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *Proceedings of the IEEE Congress on Evolutionary Computing*, pages 162–168, Hong Kong, 2008.

[7] Earl T Barr, Yuriy Brun, Prem Devanbu, Mark Harman, and Federica Sarro. The Plastic Surgery Hypothesis. In *Proceedings of the ACM SIGSOFT International Symposim on the Foundation of Software Engineering*, pages 306–317, Hong Kong, 2014.

[8] Aymeric Blot and Justyna Petke. Empirical Comparison of Search Heuristics for Genetic Improvement of Software. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, XX, 2021.

[9] Yossi Borenstein and Riccardo Poli. Decomposition of Fitness Functions in Random Heuristic Search. In C.R. Stephens, M. Toussaint, D. Whitley, and P.F. Stadtler, editors, *Foundations of Genetic Algorithms*, volume 4436, pages 123–137. Springer, Berlin, Heidelberg, 2007.

[10] George E P Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time Series Analysis Forecasting and Gontrol*. Wiley, 4 edition, 1970.

[11] Robert S Boyer, Bernard Elspas, and Karl N Levitt. SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, Los Angeles, 1975.

[12] Alexander E I Brownlee, Justyna Petke, Brad Alexander, Earl T Barr, Markus Wagner, and David R White. Gin: Genetic Improvement Research Made Easy. In *Genetic and Evolutionary Computation Conference*, volume 19, Prague, 2019.

[13] Bobby R Bruce. Energy Optimisation via Genetic Improvement A SBSE technique for a new era in Software Development. In *Proceedings of the Genetic and Evolutionary Computing Conference*, pages 819–820, Madrid, 2015.

[14] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, 2013.

[15] James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61:190–210, 2006.

[16] Yuval Davidor. Epistasis Variance: Suitability of a Representation to Genetic Algorithms. *Complex Systems*, 4(4):369–383, 1990.

[17] Richard A Demillo and A. Jefferson Offutt. Constrain-based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[18] Edsger W. Dijkstra. Structured Programming, 1969.

[19] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing Better Fitness Functions for Automated Program Repair. In *Proceedings of the ACM Genetic and Evolutionary Computing Conference*, pages 965–972, Portland, 2010.

[20] Stephanie Forrest, Thanhvu Nguyen, Westley Weimer, and Claire Le Goues. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the ACM Genetic and Evolutionary Computing Conference*, pages 947–954, Montreal, 2009.

[21] Gordon Fraser. A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator. In *Search-Based Software Engineering*, pages 106–130. Springer, 2018.

[22] Sumit Gulwani. Dimensions in Program Synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24, 2012.

[23] 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. 8, 2017.

[24] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *Proceedings of International Conference of Software Testing*, pages 1–12, Graz, 2015.

[25] Mark Harman and Justyna Petke. GI4GI: Improving Genetic Improvement Fitness Functions. In *Proceedings of the Genetic and Evolutionary Computing Conference*, Madrid, 2015.

[26] W K Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97, 1970.

[27] Ying Huang, Wei Li, Furong Tian, and Xiang Meng. A fitness landscape ruggedness multiobjective differential evolution algorithm with a reinforcement learning strategy. *Applied Soft Computing Journal*, 96, 2020.

[28] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 9 2010.

[29] Yue Jia, Mark Harman, William B Langdon, and Alexandru Marginean. Grow and Serve: Growing Django Citation Services Using SBSE. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 269–275, Bergamo, 2015.

[30] Terry Jones. *Evolutionary Algorithms, Fitness Landscapes and Search by*. PhD thesis, University of New Mexico, 1995.

[31] Sun-Woo Kim, John A Clark, and John A McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method ‡. *Software Testing, Verification and Reliability*, 11:207–225, 2001.

[32] Motoo Kimura. Evolutionary Rate at the Molecular Level. *Nature*, 217(5129):624–626, 1968.

[33] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 7 1976.

[34] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 8 1990.

[35] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the Kitchen Sink from Software. In *Proceedings of the Genetic and Evolutionary Computing Conference*, pages 833–838, Madrid, 2015.

[36] William B Langdon and Mark Harman. Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In *Proceedings of the ACM Genetic and Evolutionary Computing Conference*, pages 805–810, Madrid, 2015.

[37] William B Langdon and Mark Harman. Optimising Existing Software with Genetic Programming. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 19(1):118, 2015.

[38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions of Software Engineering*, 38(1):54–72, 1 2012.

[39] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing Security Policies: Going Beyond Functional Testing. In *IEEE International Symposium on Software Reliability*, pages 93–102, Trollhättan, 2007.

[40] R. Lipton. Fault Diagnosis of Computer Programs. Technical report, Carnegie Mellon University, 1971.

[41] A. A. Lovelace. Sketch of the Analytical Engine Invented by Charles Babbage by L. F. Menabrea of Turin, Officer of the Military Engineers, With Notes by the Translator, 1843.

[42] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2019.

[43] Zohar Manna and Richard Waldinger. Knowledge and Reasoning in Program Synthesis. *Artificial Intelligence*, 6(2):175–208, 1975.

[44] Zohar Manna and Richard J Waldinger. Toward Automatic Program Synthesis. *Article in Communications of the ACM*, 14(3):151–165, 1971.

[45] Xiaoguang Mao, Yan Lei, and Yuhua Qi. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *Proceedings of the IEEE International Conference on Software Maintainability and Evolution*, pages 180–189, Eindhoven, 2013.

[46] A Marginean, J Bader, S Chandra, M Harman, Y Jia, K Mao, A Mols, and A Scott. SapFix: Automated End-to-End Repair at Scale. 2019.

[47] Alexandru Marginean, Earl T Barr, Mark Harman, and Yue Jia. Automated Transplantation of Call Graph and Layout Features into Kate. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 262–268, Bergamo, 2015.

[48] Paul Marrow. Evolvability: Evolution, Computation, Biology. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program (GECCO-99 Workshop on Evolvability)*, pages 30–33, 1999.

[49] J. McCarthy. Towards a mathematical theory of computation. In *Proceedings of the International Foundation of Information Processes Conference*, pages 21–28, Amsterdam, 1962.

[50] Phil McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 6 2004.

[51] Peter Merz. Advanced Fitness Landscape Analysis and the Performance of Memetic Algorithms. *Evolutionary Computing*, 12(3):303–325, 2004.

[52] Michael Orlov and Moshe Sipper. Genetic Programming in the Wild: Evolving Unrestricted Bytecode. In *Proceedings of the ACM Genetic and Evolutionary Computing Conference*, pages 1043–1050, Montreal, 2009.

[53] Justyna Petke, Brad Alexander, Earl T Barr, Alexander E I Brownlee, Markus Wagner, and David R White. A Survey of Genetic Improvement Search Spaces. 2019.

[54] Justyna Petke and Alexander E I Brownlee. Software Improvement with Gin: a Case Study. 2019.

[55] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. Genetic Improvement of Software : A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2018.

[56] Justyna Petke, William B Langdon, and Mark Harman. Applying Genetic Improvement to MiniSAT. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 257–262, St. Petersburg, 2013.

[57] Collard Philippe, Verel Sébastien, and Clergue Manuel. Local search heuristics: Fitness Cloud versus Fitness Landscape. In *European Conference on Artificial Intelligence*, pages 973–974, Amsterdam, 2007.

[58] Erik Pitzer and Michael Affenzeller. A Comprehensive Survey on Fitness Landscape Analysis. *Recent advances in intelligent engineering systems*, pages 161–191, 2012.

[59] Erik Pitzer, Michael Affenzeller, and Andreas Beham. A Closer Look Down the Basins of Attraction. In *2010 UK Workshop on Computational Intelligence*, pages 1–6, 2010.

[60] Christian M. Reidys and Peter F. Stadler. Neutrality in Fitness Landscapes. *Applied Mathematics and Computation*, 117(2-3):321–350, 2001.

[61] Sophie Rochet. Epistasis in Genetic Algorithms Revisited. *Intelligent Systems*, 102(1-4):133–155, 11 1997.

[62] Richard L. Sauder. A General Test Data Generator for COBOL. In *Proceedings of the AFIPS Spring Joint Computation Conference*, pages 317–323, 1962.

[63] Eric Schulte, Westley Weimer, and Stephanie Forrest. Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair. In *Proceedings on the Genetic and Evolutionary Computing Conference*, pages 847–854, Madrid, 2015.

[64] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. Technical report, 2005.

[65] P. B. Sheridan. The arithmetic translator-compiler of the IBM Fortran automatic coding system. *Commun. ACM*, 2(2):9–21, 1959.

[66] P. Shiu and Fernando Q. Gouvea. p-Adic Numbers: An Introduction. *The Mathematical Gazette*, 79(484):215, 3 1995.

[67] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic Programming for Shader Simplification. *ACM Transactions on Graphics*, 30(6):1–12, 12 2011.

[68] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the ACM ESEC/SIGSOFT International Symposium on the Foundation of Software Engineering*, pages 532–543, Bergamo, 2015.

[69] Tom Smith and Paul Layzell. Fitness Landscapes and Evolvability. *Evolutionary Computing*, 10(1):1–34, 2002.

[70] Dennis G. Sprokholt. *Superoptimization of WebAssembly Process Graphs*. PhD thesis, Utrecht University, 2021.

[71] Peter Stadler. Linear operators on correlated landscapes. *Journal de Physique I, EDP Sciences*, 4(5):681–696, 1994.

[72] Peter F Stadler. Landscapes and Their Correlation Functions. *Journal of Mathematical Chemistry*, 20(1):1–45, 1996.

[73] Peter F Stadler. Fitness Landscapes. In *Biological Evolution and Statistical Physics*, pages 183–204. Springer, 2002.

[74] J.E. Stoy. *Denotational Semantics: The Scott—Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, 1981.

[75] Ei-Ghazali Talbi. *METAHEURISTICS FROM DESIGN TO IMPLEMENTATION*. Wiley, Chichester, 2009.

[76] Vesselin K Vassilev, Terence C Fogarty, and Julian F Miller. Information Characteristics and the Structure of Landscapes. *Evolutionary Computing*, 8(1):31–60, 2000.

[77] Nadarajen Veerapen, Fabio Daolio, and Gabriela Ochoa. Modelling Genetic Improvement Landscapes with Local Optima Networks. 6, 2017.

[78] Eelco Visser. Program Transformation with Stratego/XT. In *Domain-Specific Program Generation*, volume 3016, pages 216–238. Springer, Berlin, Heidelberg, 2004.

[79] Edward D Weinberger. Correlated and Uncorrelated Fitness Landscapes and How to Tell the Difference. *Biological Cybernetics*, 63(5):325–336, 1990.

[80] Edward D Weinberger. Local properties of Kauffman's N-k model: A tunably rugged energy landscape. *PHYSICAL REVIEW A*, 44(10):6399–6413, 11 1991.

[81] David R White. GI in No Time. *ACM Reference*, 3, 2017.

[82] David R White, Andrea Arcuri, and John A Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 8 2011.

[83] David R White and Jeremy Singer. Rethinking Genetic Improvement Programming. In *Proceedings of the ACM Genetic and Evolutionary Computing Conference*, pages 845–846, Madrid, 2015.

[84] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep Parameter Optimisation. In *Proceedings of the Genetic Evolutionary Computing Conference*, pages 1375–1382, Madrid, 2015.

[85] Mohamed El Yafrani, Marcella S R Martins, Mehdi El Krari, Markus Wagner, Myriam R B S Delgado, Belaïd Ahiod, Ricardo Lüders, and El Krari. A fitness landscape analysis of the Travelling Thief Problem. In *Proceedings of the ACM Genetic and Evolutionary Computing Conference*, Kyoto, 2018.