University of Passau
Innstr. 41
94032 Passau
Germany

Bachelor's Thesis

Analysing the Influence of the Fitness Function on the Fitness Landscape of the Android Black-Box Test Generation Problem

Sebastian Vogt

Course of Study:	Computer Science
Examiner:	Prof. Dr. Gordon Fraser
Supervisor:	Prof. Dr. Gordon Fraser, Sebastian Schweikl
Commenced:	October 24, 2022
Completed:	February 3, 2023

Abstract

When creating black-box tests for an Android app, it is desirable to generate the required GUI input sequences fully automatically. The MIO algorithm is an evolutionary approach to test generation, meaning that it applies concepts from biological evolution to optimize tests towards a coverage criterion (e.g. branch coverage). This coverage criterion has to be expressed in terms of a fitness function, which calculates how close a test is to covering a specific coverage goal (e.g. branch). The most common fitness function, branch distance, has proven to be complex to calculate in practice because it relies on distance calculations on the control-flow graph. Therefore we propose a simpler fitness function, code-based fitness, which only relies on information available directly in the code. A comparison of the search success of MIO with the two fitness functions showed that code-based fitness fails to cover many branches, which means that in its current form, it still lacks the capabilities of the more powerful branch distance fitness function. Additionally to the success comparison, fitness landscape analysis was applied to the MIO algorithm with the two fitness functions, which showed that the fitness landscape of code-based fitness is more neutral than the fitness landscape of branch distance. Unfortunately, this result cannot be used to explain the search behaviour.

Acknowledgments

I first want to thank my two advisors, Prof. Dr. Gordon Fraser and Sebastian Schweikl, who guided me through the whole process and always had good suggestions when I was stuck. I also want to thank Michael Auer, who always supported me when I had problems with MATE. Special thanks go to my lectors Kassian Köck, Nikolas Kirschstein, Thomas Kirz, and my father Dietmar Vogt for providing me with great feedback.

Contents

List of Figures

List of Tables

1 Introduction

When testing Android applications, it is not only important to test the individual methods and classes with white-box tests, but also to test the app under test (AUT) by interacting with its graphical user interface in the form of black-box testing. Of course, it is desirable to generate the required GUI input sequences fully automatically. An active branch of research regarding the automatic generation of both white-box and black-box tests focuses on generating tests with the help of evolutionary algorithms (e.g., [Arc18; FA12; MHJ16; PKT15; PKT17]).

All these approaches have in common that they optimize tests towards fulfilment of a *fitness function*, which calculates a heuristic value signifying how close a test is to fulfilling a specific testing criterion like line coverage or branch coverage [McM04]. Intuitively, a fitness function is a quantization of the Darwinian idea of fitness.

Often, the fitness function is not inherently part of an evolutionary algorithm and different fitness functions can be utilized to receive different search behaviours or results. A promising approach in automatic black-box test generation is the utilization of the MIO algorithm [Arc18] in conjunction with the MATE testing framework [Dev22], which was originally developed for uncovering accessibility flaws in Android apps [ERGF18]. However, there is not yet a canonical fitness function associated with this approach.

The most widely used fitness function in this area is the branch distance [Kor90] combined with the approach level [WBP02; WBS01] (e.g., [FA12; PKT15; PKT17]). However, the branch distance + approach level function is rather sophisticated and can thus become unwieldy in space and time complexity. Simpler fitness functions have also been applied in the past (e.g., [SAF+19]), but it is unclear whether the reduced complexity of simpler fitness functions outweigh the improved guidance of branch coverage + approach level or not.

Evolutionary algorithms and their configurations can of course be compared to each other empirically (e.g., [FA14; PKT18; SAF+19]), but further insight into how and why different techniques perform differently is often gained with a method called fitness landscape analysis (e.g., [AFS20; AMG17; VTG19]). Fitness landscape analysis aims at characterizing the underlying topological structure of the search space, with two important properties of fitness landscapes being *ruggedness* and *neutrality* [PA12].

The goal of this work is to compare branch distance + approach level with a proposed simpler fitness function with regards to their search behaviour in the execution of MIO and to gain further insight into the fitness function's influence on the behaviour of MIO by analysing its *ruggedness* and *neutrality*. By knowing what each fitness function's influence on the behaviour of MIO is, one can hope to make a more informed decision about which fitness function to use in reality and how the existing fitness functions can be improved for the future.

2 Background

2.1 Evolutionary Optimization

Before explaining the MIO algorithm itself in section 2.2, this section seeks to build a basic understanding of metaheuristic and evolutionary optimization and lay the groundwork for the section about the MIO algorithm.

Test generation can be modelled as a deterministic combinatorial optimization problem, which is defined by Bianchi et al. [BDGG09] as follows:

> Given a finite set S of feasible solutions x and a real valued cost function G(x), find $\min_{x \in S} G(x)$

The set S is called the **search space** [BDGG09], and in our case it contains all possible test cases. The cost function is called **fitness function** throughout this work. Some fitness functions map the best individual to the highest number instead of the lowest, in which case the minimum has to be replaced by the maximum in the above definition.

Finding an optimal solution to many optimization problems can be computationally infeasible, which is why we utilize metaheuristics, which are algorithms that find good solutions in reasonable time [BDGG09]. Metaheuristics are defined by Blum et al. [BR03] as follows:

> Metaheuristics are high level concepts for exploring **search spaces** by using different strategies. These strategies should be chosen in such a way that a dynamic balance is given between the **exploitation** of the accumulated search experience [...] and the **exploration** of the search space [...]. This balance is necessary on one side to quickly identify regions in the search space with high quality solutions and on the other side not to waste too much time in regions of the search space which are either already explored or don't provide high quality solutions. [emphasis added].

An important subset of metaheuristic algorithms are evolutionary algorithms, which are inspired by Darvinian evolution [BDGG09]. Bianchi et al. [BDGG09] explained the general idea of evolutionary algorithms as follows:

> Every iteration of the algorithm corresponds to a generation, where certain operators are applied to some individuals of the current population to generate the individuals of the population of the next generation. [...] At each generation, only some individuals are selected for being elaborated by variation operators, or for being just repeated in the next generation without any change, on the base of their fitness measure (this can be the objective function value, or some kind of quality measure). Individuals with higher fitness have a higher probability to be selected. In this metaheuristic, **exploration** is

> provided by mutation and recombination operators, while **exploitation** of obtained information on good solutions is enabled by the selection mechanism. [emphasis added, italics removed].

This means that by selecting solutions with high fitness to become part of the next generation, the information about the search space provided by exploration is exploited. Without exploitation, the search space would be explored randomly, and good solutions would be found purely by chance.

2.2 The MIO Algorithm

Some successful approaches in automatic test suite generation are the search algorithms WTS [FA12] and MOSA [PKT17], which all apply concepts of evolutionary optimization. Andrea Arcuri [Arc18] tries to improve upon the shortcomings of these algorithms by specifically tailoring the MIO algorithm to test generation problems and their properties. Sell et al. [SAF+19] have provided a good overview of the MIO Algorithm in pseudocode. A modified version of their code can be found in algorithm 2.1. Important aspects of MIO, as taken from its introductory paper [Arc18], are explained in the following sections.

Many Independent Objective

MIO stands for **M**any **I**ndependent **O**bjective algorithm. This means that instead of optimizing just the overall coverage (e.g. branch coverage) of a test, the overarching goal of the optimization is divided into a set of objectives (e.g. branches), which each have their own fitness function to be optimized. Canonically, such a fitness function maps a test to the interval $[0, 1]$, where the objective is covered if and only if the heuristic value is equal to 1. Any standard coverage function (like branch coverage) would only generate binary fitness values for each objective, so more specialized heuristics are needed for MIO to operate effectively. As the fitness function used can be viewed independently from the MIO algorithm, the further discussion of fitness functions is located in a later chapter.

For every objective MIO seeks to optimize during execution, a population of n tests is kept. The set of these populations is collectively called the *archive*. Every population in the archive always contains the best tests for the specific fitness function that have been discovered yet. In the end, a single objective is either covered or not by the best test in its corresponding population. The final test suite emerges by collecting the best test for every covered objective. In algorithm 2.1, the input parameter L is a list containing the fitness function for every objective and for every $k \in L$, $archive_k$ is the corresponding population.

The Base Procedure

In every iteration of the algorithm, a new test is either created at random with the probability P_r or an existing test is sampled from the archive and mutated m times independently with the probability $1 - P_r$ (line 3 of algorithm 2.1). This results in one or more new tests. (In algorithm 2.1 the m parameter is ignored and a test is mutated at most once instead of m times in order to simplify the

Algorithm 2.1 The MIO Algorithm

Input:

- search budget B
- random sampling probability P_r
- the amount of mutations per sampled test m (not used in the pseudocode for simplicity)
- list of fitness functions L
- population size limit n
- start of focused search F

Output: Test suite of optimized test cases

```
1: archive ← createInitialPopulation()
2: while B is not used up do
3:     if rand() < P_r then
4:         t ← createRandomIndividual()
5:     else
6:         t ← sampleIndividual(archive)
7:         t ← mutate(t)
8:     end if
9:     for each fitness function k in L do
10:        fitness ← k.getFitness(t)
11:        if objective is covered by t then
12:            archive_k ← {t}
13:            archive_k.covered ← True
14:        else if objective is partially covered then
15:            archive_k ← archive_k ∪ {t}
16:            resetCounter(k)                    // See section about feedback-directed sampling
17:            if |archive_k| > n or archive_k.covered then
18:                removeWorstTest(archive_k)
19:            end if
20:        end if
21:        updateParameters(F, P_r, n)
22:    end for
23: end while
24: return Set of all tests from the archive which fully cover the objective of their respective
    population
```

code.) For each of these tests, the following procedure is executed: For every objective, the fitness function of the newly created test is evaluated to obtain the fitness score s and the test is inserted into the corresponding population (line 9 et seq.) according to the following rules:

- If $s = 1$ (line 11 et seq.), the population size of this specific population will be permanently set to 1 and the new test will replace the current population. (If the objective was already covered by the old population, it will only be replaced if the new test is shorter or has better coverage on other objectives. This logic is left out of the pseudocode for clarity.)
- Otherwise (line 14 et seq.), the test will be inserted, and if this causes the population size to exceed the limit, the worst test will be removed (line 17). (This means that if the new test is worse than any existing tests in the population and the population is full, the new test will effectively be discarded.)

Feedback-Directed Sampling

A simple strategy of sampling tests from the archive would be to choose a random non-covered objective and then a random test from the corresponding population. (It is important to remember that an objective is non-covered as long as there does not exist a test with fitness value 1 with regard to that objective.) This would of course lead to a very thorough exploration of the search space, but in the end, having few covered objective is favourable over having many half-way covered objectives, since only fully covered objectives increase the overall coverage. For this reason, feedback-directed sampling enables MIO to focus on objectives that provide a clear uphill-direction, while neglecting objectives that are infeasible or have otherwise plateaued.

In feedback-directed sampling, every objective has a counter initialized to 0. It intuitively measures the time since the last fitness improvement for its objective. The sampling algorithm now chooses a test randomly from the non-covered population with the lowest counter value. More precisely, the counter is incremented whenever a test is sampled from its corresponding population and reset to 0 when a new test replaces an old test in its population (line 16).

Exploration/Exploitation Control

While it is important to focus the search on gradients of the fitness landscape that have already been discovered (exploitation), it is also crucial to discover such gradients in the first place (exploration). Exploration is best done in the beginning of the search, so that the end of the search can be used to maximize as many found gradients as possible. The transition from exploration to exploitation in MIO is controlled with the F parameter. It defines how much of the search budget has to be used up for the focused search (= exploitation mode) to begin. Exploration and exploitation is principally controlled by values of the P_r, n and m parameters. These values are initialized with certain values for exploration (e.g. $P_r = 0.5,\ n = 10,\ m = 1$, as suggested in the original paper). Then, these values decrease linearly to reach the values for the focused search (e.g. $P_r = 0,\ n = 1,\ m = 10$).

P_r usually decreases in the course of the search, because a lower likelihood of creating a new random test leads to a lower degree of exploration. The n value is also decreased, because a lower n value leads to the search focusing on the best test for each objective and thus a higher degree of exploitation. A high m value strengthens this effect by mutating a single sampled individual multiple times and thus focusing on improving this one before moving on to a different individual.

The Input Parameters

In this section, the input parameters of MIO are explained in the same order in which they are given in algorithm 2.1. It is also briefly mentioned how the respective parameters are used in the algorithm.

- The search budget B, in its simplest form, is just the amount of time the search takes. More generally, it represents an amount of a resource that is used up during the search. When the resource is depleted, the search ends. It is used in the main loop of the algorithm in line 2.
- The random sampling probability P_r defines how likely it is that a new test will be created in each iteration instead of an old one being mutated. It is used in line 3.
- The parameter m defines how often the same test should be mutated before a new one is sampled. At the beginning of the search, this value usually is 1. In algorithm 2.1, m is always set to 1, so as not to overcomplicate the pseudocode.
- The list of fitness functions L: For every objective MIO optimizes towards, a heuristic fitness function is given. It is 1 if and only if the objective is covered and its value approximates how far the objective is from being covered.
- The population size limit n: MIO operates on an archive of tests, which consist of one population for each objective. n is the maximum number of tests any of these populations can hold.
- The start of focused search F is the percentage of the search budget, after which the focused search begins. Focused means that instead of discovering new tests in the search space, the goal is merely to further improve existing tests. The focused search mode is set by mutating the parameters P_r, n and m, for example to $P_r = 0$, $n = 1$, $m = 10$. This change does not happen at once, the values change linearly during the first part of the search.

2.3 Configuring MIO for Android Black-Box Testing

A crucial part of MIO that is not defined in the algorithm itself is the fitness function used. Before defining the fitness function, we must define the set of objectives for the search. In this work, we choose the common testing criterion of branch coverage and thus we use the set of all non-library branches in the app under test as the set of objectives. Now we need to define a fitness function for every objective, which takes a test case as input and returns a heuristic value indicating how far the test case is from covering the base objective. Since the fitness functions for each objective are constructed in the same way, we call this construction itself simply a *fitness function*.

Next to the widely adopted branch distance + approach level function we propose a novel fitness function, which aims to reduce the computational complexity of the fitness evaluations. Practical experience shows that the fitness evaluations are a great contributor to the execution time of genetic algorithms, which is why we defined the simpler fitness function. The next two sections explain both fitness functions in detail.

2.3.1 Branch Distance Plus Approach Level

The branch distance function was introduced by Korel [Kor90] for automatic testing of Pascal programs. Originally, it was used in the context of optimizing test input towards the execution of a specific target path in the control flow graph.

Let $\{n_i\}_{i \in \mathcal{N}}$ be the set of nodes in the control flow graph and let the target path P be $< n_{k_1}, n_{k_2}, ..., n_{k_l} >$. Say the current test input x_i leads to an execution that follows the target path up to n_{k_i}. This means that at the branch $(n_{k_i}, n_{k_{i+1}})$, the control flow takes a wrong turn. The goal is now to start a local search procedure that generates a new test input x_{i+1} which executes the target path until $n_{k_{i+1}}$. This process is then repeated until the whole target path is covered.

For this we need a function that tells us how far the current test input is away from taking the branch $(n_{k_i}, n_{k_{i+1}})$ at the branching statement n_{k_i}. We assume that the branching statement n_{k_i} is of the form $x \ \Phi \ y$ with $\Phi \in \{<, \leq, >, \geq, =, \neq\}$. In any case, this can be reformulated to $F(x, y) < 0$, $F(x, y) \leq 0$ or $F(x, y) = 0$ (e.g. $x < y \Leftrightarrow x - y < 0$).

Now we have the branch distance function F, a fitness function that must be minimized, that depends solely on the testing input and that can be calculated dynamically during program execution. The branch distance function for $(n_{k_i}, n_{k_{i+1}})$ can only be used if the execution reaches the branching statement n_{k_i} in the first place. Otherwise the branch distance for a specific branch might be optimal without the branch being covered, because the control flow is somewhere else. In Korel's [Kor90] work this is not a problem, since the branch distance is by design only used as a fitness function when the corresponding branching statement has been reached. In the MIO approach meanwhile, tests are optimized towards covering a branch directly, without looking at a specific path in the control flow graph. This means that branch distance as a fitness function in MIO only provides guidance as soon as a test reaches the branching statement and is a plateau otherwise.

This is why Wegener et al. [WBS01] introduced the approach level (originally called approximation level) to provide guidance for reaching the required branching statement for a specific branch. It "is defined as the distance between the closest control dependency of the target node executed by a test and the target node in the control dependency graph" [AFS20]. To the approach level, the branch distance of the said control dependency is added to form the branch distance + approach level fitness function. In the following, the combination of branch distance and approach level is just called **branch distance fitness** for short, the approach level is implied.

2.3.2 Code Based Fitness Function

We propose a fitness function f which provides guidance for covering branches in the code. In order to receive a more lightweight function than branch distance, we do not use the control-flow graph but calculate distances solely based on features of the code itself. This is why we call it **code-based fitness**.

We define f to be a maximizing fitness function in the interval $[0, 1]$. This means that for a test case and a branch, the value of f should be one if the test case covers the branch and in $[0, 1)$ if it does not cover the branch. In the latter case, the value of f should provide an approximation of how close the test case comes to covering the objective.

For this it is helpful to have a measure of closeness between two branches. Then, the fitness of a test case regarding a target branch is just the closeness between the target branch and the closest branch reached during test execution. To define such a measure, we must first understand how branches are represented in MATE. Each branch is identified by a string that is based on the smali [Gru22] assembler language, for example `Luk/co/example/package/ExampleClass;->exampleMethod()Z->4`.

This representation consists of the following parts, delimited by an arrow (`->`):

- The full class name `Luk/co/example/package/ExampleClass;` (note that class names are always prefixed with "`L`" and suffixed with "`;`"),
- the full method name with argument types and return type `exampleMethod()Z`, where `Z` stands for boolean and denotes the return type,
- and a number n denoting the n-th branch in the bytecode, counted from top to bottom.

From here on, we represent a branch as $c_1c_2...c_lmn$, where $c_1/c_2/.../c_{l-1}$ is the package path of the class name, c_l is the class name itself, m is the method name and n is the number of the branch. We define the closeness of two branches $t = c_1c_2...c_lmn$ and $\tilde{t} = \tilde{c}_1\tilde{c}_2...\tilde{c}_{\tilde{l}}\tilde{m}\tilde{n}$, depending on the parameter α the following way: Let $p_i = c_1c_2...c_i \; \forall i \in \{1, ..., l\}$

$$closeness(t, \tilde{t}) = \begin{cases} 1 & \text{if } t = \tilde{t} \\ \frac{2}{3} + \frac{1}{3} \cdot \frac{\alpha}{|n-\tilde{n}|+\alpha} & \text{if } c_1c_2...c_lm = \tilde{c}_1\tilde{c}_2...\tilde{c}_{\tilde{l}}\tilde{m} \\ \frac{1}{3} & \text{if } c_1c_2...c_l = \tilde{c}_1\tilde{c}_2...\tilde{c}_{\tilde{l}} \\ 0 + \frac{1}{3} \cdot \frac{\max\{i | p_i = \tilde{p}_i\}}{\max\{l, \tilde{l}\}} & \text{else} \end{cases} \tag{2.1}$$

The basic structure of this function are the four basic steps 0, $\frac{1}{3}$, $\frac{2}{3}$ and 1:

- if the branches are in different root packages, the closeness is 0,
- if they are in the same class, the closeness is $\frac{1}{3}$,
- and if they are the same branch, the closeness is 1.

Between 0 and $\frac{1}{3}$, additional guidance is provided by the function $\frac{1}{3} \cdot \frac{\max\{i | p_i = \tilde{p}_i\}}{\max\{l, \tilde{l}\}}$. It is a linear function dependent on the amount of consecutive packages that match in the beginning of the class name. The longer the prefixes of the class names match, the higher this value.

Between $\frac{2}{3}$ and 1, additional guidance is provided by the function $\frac{1}{3} \cdot \frac{\alpha}{|n-\tilde{n}|+\alpha}$. This is a rational function dependent on the number of branches between the input branches. It is $\frac{1}{3}$ at 0 and converges to zero, as the input variable ($|n - \tilde{n}|$) increases. The parameter α controls how fast the convergence is. For smaller α it converges faster, for larger α, it converges slower.

Now we can define the fitness with regards to the branch $\hat{b}$ of a test case t that executes the branches $\{b_1, ..., b_n\}$ as

$$f_{\hat{b}}(t) = \max_{b \in \{b_1,...,b_n\}} closeness(b, \hat{b}). \tag{2.2}$$

2.3.3 Test Case Representation and Mutation

In addition to the fitness function, MIO does also not specify how exactly test cases look like and how the mutation function works. In MATE [Dev22], test cases are represented as action sequences filled with GUI actions like clicks [SAF+19] and we set their size to 50 actions, which is the value used by Sell et al. [SAF+19]. The initial population is created by sampling one chromosome at random. Random sampling happens by executing a random GUI action in the app 50 times and saving the executed actions to receive an action sequence. In our experiments, we use the cut point mutation function already implemented in the MATE framework [Dev22], which was also used by Sell et al. [SAF+19] in conjunction with MIO. The cut point mutation function cuts off the event sequence of a test after a random event (this may leave the sequence empty) and then randomly generates new actions from that point onwards to create a mutation.

2.4 The Fitness Landscape of MIO

The past sections were mainly concerned with explaining MIO, the test case generation algorithm used in this work. The rest of the background chapter will revolve around fitness landscape analysis, which we want to apply to MIO in order to analyse its search behaviour. By performing fitness landscape analysis on MIO with the two fitness functions respectively, we want to gain insight into how the choice of fitness function influences the search behaviour of MIO.

Sewall Wright [Wri+32] first thought out the idea of a fitness landscape as part of his work on evolutionary biology. He imagined the field of all possible gene combinations as a plane where closely related ones are located closely together. The fitness values of these gene combinations then generate a plastic landscape on top of that flat plain. This can be best imagined as a 2-d plain with the fitness values plotted in the third dimension. In this metaphor, the process of evolution can be seen as the individuals of a population exploring their neighbourhood on the fitness landscape by means of reproduction. Natural selection then leads to the population climbing up slopes in the fitness landscape instead of navigating it randomly.

At the end of a slope often lies a local optimum, which could be a (near)-optimal solution to the search problem and thus a good result of the optimization. If this is not the case however, the search budget spent on climbing that slope is often wasted, and different parts of the fitness landscape have to be explored instead. Intuitively, this problem arises more often when the fitness landscape has more local optima, which is why it is important to look at the set of local optima in fitness landscape analysis. A concept which builds upon the notion of local optima is **ruggedness**, which

is defined as "the frequency of changes in slope from up-hill to down-hill" [PA12]. When dealing with a high-ruggedness landscape, it can be expected that the evolutionary search struggles with climbing desirable optima in the presence of many local optima.

When dealing with a low-ruggedness landscape on the other hand, it is important to know whether the regions of the fitness landscape without changes in slope actually have a slope in the first place or are dominated by plateaus. The latter case can be detrimental to an evolutionary search algorithm, because the absence of a slope renders the selection mechanism useless, locally turning the search into a random exploration of the landscape. This is why in order to get a sounder view of the fitness landscape, it is important to pair the study of ruggedness with the concept of **neutrality**, which is defined as the "degree, to which a landscape contains *connected areas of equal fitness*" [AFS20].

To quantify ruggedness and neutrality, a common approach is to sample a part of the fitness landscape with a random walk, which is generated by starting with a random individual and then sampling a random neighbour on the fitness landscape n times, and then apply a specific measure to the resulting fitness sequence [PA12].

Albunian et al. [AFS20] successfully used a set of measures of ruggedness and neutrality to show that the Java unit test generation problem with an evolutionary algorithm suffers from a high degree of neutrality in the fitness landscape, which negatively affects the success of the test generation. Due to the similarity of their problem to ours, we use a subset of their measures in our own work. We use them to analyse the ruggedness and neutrality of the fitness landscapes generated by the code-based and branch distance fitness functions respectively. After a short theoretical definition of the fitness landscape of MIO, section 2.5 specifies the measures of ruggedness and neutrality used in the fitness landscape analysis of this work.

Defining the Fitness Landscape of the MIO Algorithm

While an abstract definition of fitness landscapes can be found in the survey by Pitzer and Affenzeller [PA12], the goal of this section is to provide a use-case specific definition based on the general one. Let S be the search space containing all possible tests. While a solution to the MIO algorithm is made up of multiple tests, we look at the fitness landscape per objective, where a single test is a successful solution if and only if it covers the objective.

We ignore the issues of encoding the test case and of genotype-phenotype mapping since the representation of a test case does not yield any theoretical significance and both mutations and fitness evaluations are done directly on the test cases. Let $f : S \to \mathbb{R}$ be the fitness function given by the chosen base fitness function and the objective. W.l.o.g. we assume that $f : S \to [0, 1]$ and that $f(t) = 1$ signifies that the objective is covered by test t.

To complete the notion of a fitness landscape we now need a distance function $d : S \times S \to \mathbb{R}$ on the search space. This distance is implied by the genetic operator, in our case the testCaseCutPoint-Mutation operator. This mutation operator has the property that any test can possibly be generated out of a specific test by mutation. This means we cannot define the distance function in the standard way, where the distance between two test cases is defined as the amount of mutations it takes to go from one test case to the other. Instead, we propose a stochastic definition that is based on the probability that a mutation on testcase t results in testcase t' which we denote as $\mathbb{P}_t(t')$:

$$d : S \times S \to \mathbb{R}, (t, t') \mapsto 1 - \mathbb{P}_t(t') \tag{2.3}$$

This completes the definition of a fitness landscape as $\mathcal{F} = (S, f, d)$

2.5 The Measures of Ruggedness and Neutrality

In this work, we employ a subset of the fitness landscape measures used by Albunian et al. [AFS20].

2.5.1 Measures of Ruggedness

Autocorrelation

The autocorrelation is one of the most basic measures of ruggedness and it is calculated by correlating a fitness sequence with its shifted self [PA12]. Given a fitness sequence $\{f_t\}_{t=1}^{N}$ and the shift step size k, the autocorrelation is defined as

$$r_k(\{f_t\}_{t=0}^{n}) = \frac{\sum_{i=1}^{N-k}(f_i - \bar{f})(f_{i+k} - \bar{f})}{\sum_{i=1}^{N}(f_i - \bar{f})^2} \tag{2.4}$$

where $\bar{f}$ is the mean of all fitness values [AFS20]. This measure is usually interpreted as higher autocorrelation values corresponding to a more correlated fitness landscape and thus less ruggedness [AFS20].

Information Content

Vassilev et al. [VFM00] proposed a set of fitness landscape measures that are based on the information characteristics of (random) walks. The motivation for these measures is that classical measures like autocorrelation "do not go far enough and give us only a vague notion of the structure of the landscapes" [VFM00]. All proposed new measures are based on a fitness sequence $\{f_t\}_{t=0}^{n}$, more precisely on a string derived from the sequence, which only captures its ups and downs and ignores the amplitudes. We use the *information content*, which is the most prominent of the proposed measures. This subsection is entirely based on the paper by Vassilev et al. [VFM00].

Before we can define the actual measure, we have to derive a string of ups and downs from our fitness sequence.

Definition 2.5.1 (String associated with a fitness sequence)
Let $\{f_t\}_{t=0}^{n}$ be a fitness sequence and let $\varepsilon > 0$ be the accuracy.
Then $S(\varepsilon) = s_1 s_2 s_3 \ldots s_n$ with

$$s_i = \begin{cases} \bar{1}, & \text{if } f_i - f_{i-1} < -\varepsilon \\ 0, & \text{if } |f_i - f_{i-1}| \leq \varepsilon \\ 1, & \text{if } f_i - f_{i-1} > \varepsilon \end{cases}$$

is the string associated with the fitness sequence.

Now we can define the measure itself, called the *information content*.

Definition 2.5.2 (information content (entropic measure))
The information content is defined as

$$H(\varepsilon) := -\sum_{p \neq q} P_{[pq]} \log_6 P_{[pq]}$$

with

$$P_{[pq]} := \frac{n_{[pq]}}{n}$$

where $n_{[pq]}$ is the number of occurences of the substring pq in $S(\varepsilon)$ and n is the number of substrings of length 2 in $S(\varepsilon)$.

"[T]he IC measure is meant to characterize the ruggedness of the landscape where a value close to 1 indicates a large number of peaks in the landscape, i.e., a rugged landscape" [AFS20].

2.5.2 Measures of Neutrality

Neutrality Distance

The neutrality distance merely counts the number of equal values at the beginning of a fitness sequence, so for $\{f_t\}_{t=1}^n$ it is defined as

$$nd(\{f_t\}_{t=1}^n) = \max\{k | f_0 = \ldots = f_k\} \tag{2.5}$$

and a higher neutrality distance is indicative of a more neutral landscape [AFS20].

Neutrality Volume

This measure is explained by Albunian et al. [AFS20] as follows:

> Neutrality Volume (NV) is another measure of neutrality based on the number of neighbo[u]ring areas of individuals with equal fitness during the random walk. For example, the NV of the sequence of fitness values $\{f_t\}_{t=0}^7 = \{0.3, 0.3, 0.3, 0.2, 0.2, 0.7, 0.7\}$ is 3 as there are 3 areas of equal fitness with values 0.3, 0.2 and 0.7. The NV of $\{f_t\}_{t=0}^7 = \{0.3, 0.3, 0.1, 0.2, 0.2, 0.7, 0.4\}$ is 5. The interpretation of the two cases is that the landscape in the first example is expected to be flatter than of the second example as more of the fitness values are equal.

2.5.3 Aggregating the Information Content

The information content measure relies on a value ε, which can sensibly be set to a value between $\varepsilon = 0$ and $\varepsilon = \varepsilon^*$, which is the lowest value at which the information content is 0 [VFM00]. For the analysis in this paper, we require the information content to yield a singular value, which is why we need a method to aggregate over different ε-values. Malan and Engelbrecht [ME09] have proposed a method to generate a singular information content value without having to chose a specific ε-value. The basis for their strategy is a set of n random walks, which are independently generated on the same landscape. The goal is then to approximate the theoretical formula

$$R = \max_{\varepsilon \in [0, \varepsilon^*]} \mathcal{H}(\varepsilon), \tag{2.6}$$

where the $\mathcal{H}(\varepsilon)$ denotes the theoretical information content of the entire landscape. To achieve this, the first random walk is used to calculate ε^*. Then, the information content $H_i(\varepsilon)$ of every random walk $i = 1, ..., n$ is calculated for each $\varepsilon \in \{0, \frac{\varepsilon^*}{128}, \frac{\varepsilon^*}{64}, \frac{\varepsilon^*}{32}, \frac{\varepsilon^*}{16}, \frac{\varepsilon^*}{8}, \frac{\varepsilon^*}{4}, \frac{\varepsilon^*}{2}, \varepsilon^*\} = E$. For each ε, the mean information content $\bar{H}(\varepsilon)$ over all n random walks is calculated. The final measure is then computed as the maximum of these means over all ε-values.

$$R \approx \max_{\varepsilon \in E} \bar{H}(\varepsilon) = \max_{\varepsilon \in E} \frac{1}{n} \sum_{i=1}^{n} H_i(\varepsilon). \tag{2.7}$$

Unfortunately this method is not applicable in our case, since we want to have one value for every random walk and not just one value over all repetitions of the random walk. For this reason, we propose a different method, based on the strategy by Malan and Engelbrecht, which we call direct aggregation. For every random walk, we calculate ε^* and then the final measure with the formula:

$$r = \max_{\varepsilon \in E} \{H(\varepsilon)\} \tag{2.8}$$

For our results not to be distorted, it is important that the mean of the r-values over all repetitions is approximately equal to the R-value. So it should hold that:

$$\frac{1}{n} \sum_{i=1}^{n} \max_{\varepsilon \in E} H_i(\varepsilon) \approx \max_{\varepsilon \in E} \frac{1}{n} \sum_{i=1}^{n} H_i(\varepsilon) \tag{2.9}$$

It is easy to find a counterexample that proves that this equivalence does not hold in general. This means that in order to use direct aggregation, it first has to be shown that this equality approximately holds for the given data.

3 Experimental Study

The goal of our experiments is to shed light on the influence of the fitness function on the fitness landscape and the search success of the MIO algorithm for Android black-box test generation. More precisely, we want find out how our own code-based fitness function (see section 2.3.2) compares to the branch distance fitness function (see section 2.3.1), for which we devise the following research questions:

RQ1 How do the two fitness functions compare overall regarding the neutrality and ruggedness of the fitness landscape of the Android black-box test generation problem?

RQ2 Is there a per-objective statistical difference regarding neutrality and ruggedness of the fitness landscape generated by the two fitness functions?

RQ3 How do the two fitness functions compare overall regarding the success of the test generation with MIO?

RQ4 Is there a per-objective statistical difference regarding the success of the MIO executions with the two fitness functions?

RQ5 Can performance differences between the fitness functions be explained by the properties of the fitness landscape?

The following section (3.1) explains the experiment conducted as a basis for the entire study and in the sections thereafter (3.2 - 3.6) the analysis methods, results and discussion are included for each research question. Ultimately, the threats to validity are discussed (section 3.8).

3.1 Experimental Setup

3.1.1 Dataset

Sell et al. sampled a set of 10 open source Android apps from F-Droid[1] for their evaluation of Android integration test generation algorithms in MATE [SAF+19]. They chose the apps randomly with a set of restrictions that make them suitable for use with their MATE framework.

Unfortunately, some of these apps induced technical failures like out of memory exceptions and/or excessive execution times in our specific setting. These failures are partly caused by the inefficiency of MATE when handling larger apps and partly by bugs in the technical side of interacting with the apps during runtime. In the end, we used five different apps for the data generation of the fitness landscape analysis and three of those for the MIO executions (see table 3.1, the apps used

[1] https://f-droid.org/en

for the MIO execution are printed in bold). Although we did generate some data for activitydiary and shoppinglist, the entire data analysis is based solely on the 1300 branches from the three apps in bold, unless noted otherwise. These apps were instrumented for branch coverage using the instrumentation project by Auer [Aue22].

Table 3.1: Android apps used for the experiments

app name	version	#branches
activitydiary	1.4.0	1007
periodical	**1.64**	**498**
rentalcalc	**0.5.1**	**454**
drhoffmannsoftware	**1.16-11**	**348**
shoppinglist	0.11.0	300

3.1.2 Experiment Procedure

Random Walks

For the fitness landscape analysis, we perform random walks on the fitness landscape of the Android black-box test generation problem and then use the four measures introduced in section 2.5 to analyse the neutrality and ruggedness of the landscape.

For this, we implemented the random walk algorithm in the MATE framework. Any random walk execution in MATE explores one app - the app under test - and uses one fitness function. It starts with a randomly generated test for the app under test and then repeatedly mutates the test and logs its fitness value.

We execute random walks with 750 mutation steps, resulting in fitness sequences of the length 751. This is a trade-off between the standard value from the literature of 1000 [Bar98] and the execution time of the experiments. Even with the reduced walk-length, some random walks took up to 24 hours to complete. On each app and with both fitness functions, 20 random walks are performed to get a more representative picture of the fitness landscape.

Due to the many-objective nature of MIO, we do not just receive one fitness sequence per random walk, but a sequence of fitness vectors, where the entries of a fitness vector correspond to the objectives (branches) in the app under test. We can also interpret this as receiving one fitness sequence for every random walk and objective.

To every one of these fitness sequences, our measures of neutrality and ruggedness can be applied. The resulting data is exemplified in table 3.2, which represents how the results could look like with two repetitions per app and on the apps A and B which each have 2 branches. Also, only one measure instead of all four is given.

Table 3.2: Mock results of the random walks

observation	fitness function	objective	repetition	measure
0	code-based	A branch 0	0	0.7
1	code-based	A branch 0	1	0.6
2	code-based	A branch 1	0	0.9
3	code-based	A branch 1	1	0.9
4	code-based	B branch 0	0	0.3
5	code-based	B branch 0	1	0.5
6	code-based	B branch 1	0	0.6
7	code-based	B branch 1	1	0.9
8	branch distance	A branch 0	0	0.4
9	branch distance	A branch 0	1	0.3
10	branch distance	A branch 1	0	0.6
11	branch distance	A branch 1	1	0.8
12	branch distance	B branch 0	0	0.4
13	branch distance	B branch 0	1	0.5
14	branch distance	B branch 1	0	0.6
15	branch distance	B branch 1	1	0.6

Unfortunately, the information content measure cannot be blindly applied to the random walks (see section 2.5.3). Before that, equation 2.9 must be verified for the results of our random walks. In the following, this is known as **RQ0**.

MIO Runs

To get information about how MIO performs using each fitness function respectively, we run the MIO algorithm (which is already available in MATE) on our sample apps. For each run, we chose an iteration number of 200, which is a trade-off to reduce execution times. Similarly to the random walks, the iteration number was chosen such that the execution time of one run stays under 24 hours. We execute 20 MIO runs per app and fitness function.

Since a test is only useful if it covers a specific branch, we do not obtain entire fitness sequences from our MIO runs, but rather just a success value, which is 1 if a branch was covered during the run and 0 otherwise. That means that for each fitness function and objective, we receive 20 success values. An exemplification of the resulting data analogous to table 3.2 for the random walks can be found in table 3.3.

The hyperparameters of MIO were set to their standard settings suggested in MIO's introductory paper [Arc18]. The random sampling probability P_r, amount of mutations per sampled test m and population size limit n were set to $P_r = 0.5$, $n = 10$, $m = 1$ for the exploration and to $P_r = 0$, $n = 1$, $m = 10$ for the focused search with the start of the focused search F being set to $F = 0.5$.

Table 3.3: Example results of the MIO runs

observation	fitness function	objective	repetition	success
0	code-based	A branch 0	0	0
1	code-based	A branch 0	1	0
2	code-based	A branch 1	0	0
3	code-based	A branch 1	1	1
4	code-based	B branch 0	0	1
5	code-based	B branch 0	1	0
6	code-based	B branch 1	0	0
7	code-based	B branch 1	1	1
8	branch distance	A branch 0	0	1
9	branch distance	A branch 0	1	1
10	branch distance	A branch 1	0	0
11	branch distance	A branch 1	1	1
12	branch distance	B branch 0	0	0
13	branch distance	B branch 0	1	0
14	branch distance	B branch 1	0	1
15	branch distance	B branch 1	1	0

3.1.3 RQ0 - Can the information content measure be applied to the generated data?

Before we can actually apply the information content measure to the random walks, we first have to make sure that equation 2.9 holds for our data. This is done by calculating both sides of the equation for every objective. For this calculation, we used all objectives, and not just the ones that yielded MIO success values.

The maximum difference between the two sides of the equation is 0.0049, which means that we can apply the information content as intended.

3.2 RQ1 - How do the two fitness functions compare overall regarding the neutrality and ruggedness of the fitness landscape of the Android black-box test generation problem?

3.2.1 Analysis

To answer how rugged and neutral the fitness landscape is for each fitness function, the data received (exemplified in table 3.2) is split up into a separate table for each fitness function. For both fitness functions, the distribution of each measure in the respective sub-table is then represented in a violin plot [HN98]. This gives a first idea on how the fitness landscapes compare for the two fitness functions across all objectives.

3.2.2 Results

Performing 20 random walks on each of our 1300 branches with both fitness functions resulted in 26000 measure values for each fitness function and measure.

The violin plots for neutrality distance, neutrality volume and information content can be found in figure 3.1. For the neutrality distance measure, the boxplots were hardly visible in the violin plot, which is why we plotted the kernel density plot and the boxplot separately.

For the autocorrelation measure, the violin plots for $k = 1, 2, 3$ can be found in figure 3.2. For higher k-values, the distribution did not change in a way that is relevant for the discussion, which is why only these three k-values are reported.

The autocorrelation measure is not defined for flat walks, who were exclusively observed for code-based fitness, where out of 26000 walks, 4851 were entirely flat. The ratio of flat to non-flat walks for code-based fitness can be found in a bar plot in figure 3.2.

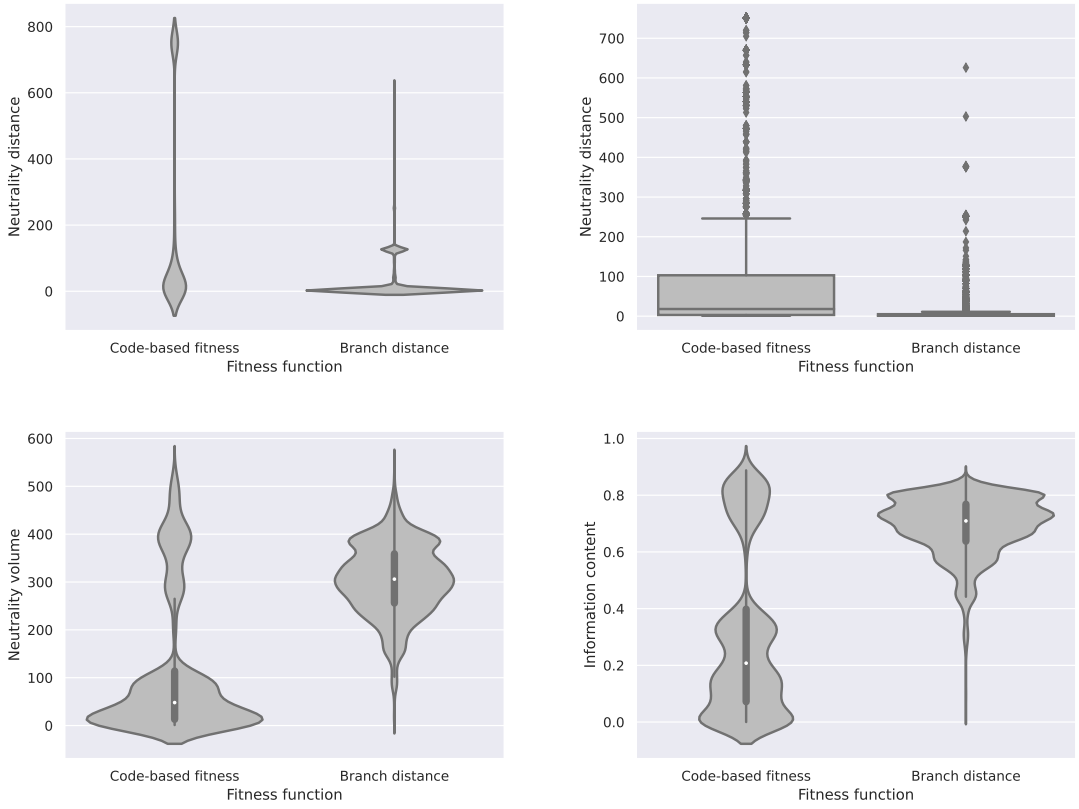

Figure 3.1: Distribution of neutrality distance, neutrality volume and information content

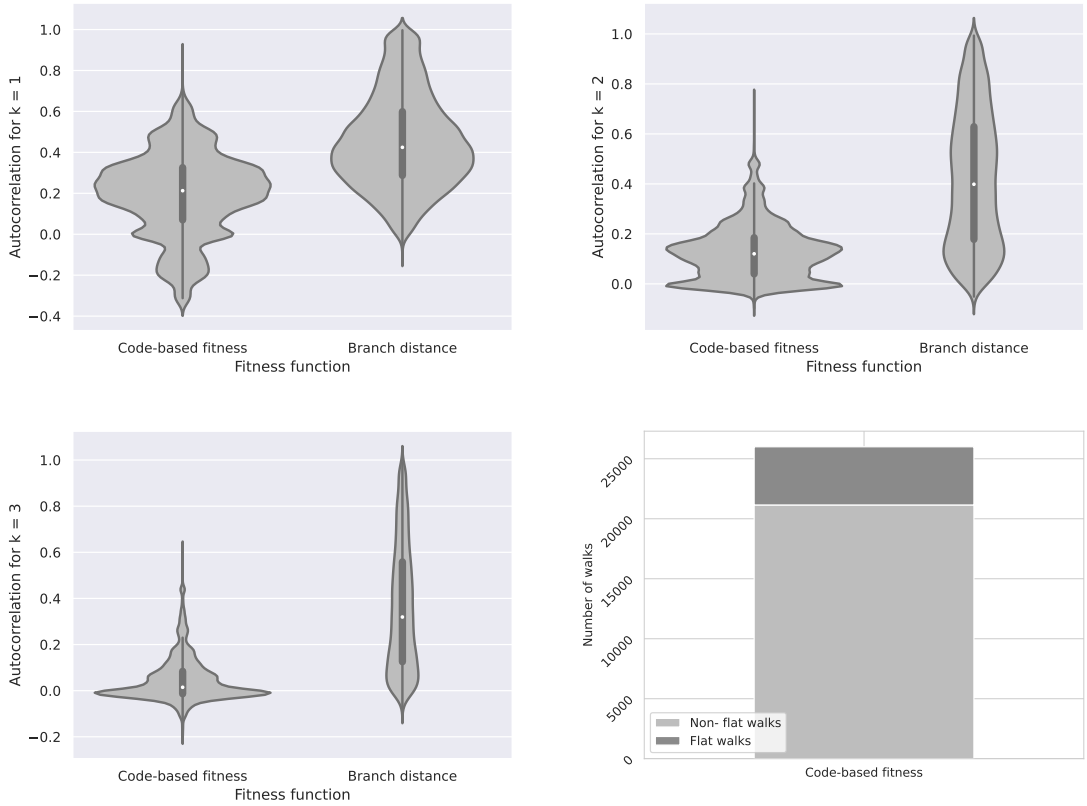

Figure 3.2: Distribution of autocorrelation

3.2.3 Discussion

The neutrality distance values for branch distance are much more concentrated close to the value 1 than the values for code-based fitness, as can be seen in figure 3.1. This means that branch distance tends to generate smaller sequences of equal fitness at the beginning of a walk than code-based fitness, suggesting that the fitness landscape of the latter is more neutral than the fitness landscape of the former. Additionally, for code-based fitness there is a sizeable set of random walks, which have a neutrality distance at the walk length of 751, which means that they were completely flat. The ratio of flat walks to non-flat walks for code-based fitness can be observed in the bar plot in figure 3.2. This is again a strong indication that the fitness landscape of the code-based fitness function is more neutral than the fitness landscape of the branch distance function.

Neutrality volume paints a similar picture. For code-based fitness, most runs have a rather low neutrality volume, while for branch distance, the distribution is much more balanced around a higher median. A lower neutrality volume means that there are less areas of equal fitness in a run, meaning that these areas are bigger (keep in mind that a slope of length n has n areas of equal fitness), suggesting that the plateaus generated by branch distance are often smaller than the plateaus generated by code-based fitness.

Regarding ruggedness, figure 3.1 shows that the fitness landscape of branch distance has a rather high information content (IC). The distribution of the IC values for code-based fitness consists of two main areas. The first one is at about the same height as the main peak for branch distance, while the second one is close to zero. An information content (close to) zero means that the walks were almost entirely made up of slopes or neutral areas. While generally lower ruggedness is seen to have a positive impact on search behaviour, in this case it seems reasonable to suppose that the lower ruggedness found in the fitness landscape of code-based fitness is mainly caused by its higher neutrality and not by smooth gradients in the fitness landscape. This means that while the code-based fitness function seems to generate a less rugged landscape, this is not expected to positively influence the search behaviour.

Looking at autocorrelation, figure 3.2 shows that for all $k = 1, 2, 3$, the autocorrelation for code-based fitness is lower than for branch distance. For higher k-values, the distribution of the autocorrelation values for code-based fitness get squished around the value 0. This suggests that code-based fitness generates a more rugged fitness landscape, which is in deviation to what the information content measure suggests. The first important point that can help reconcile these two results is that many walks for code-based fitness are not included in the violin plot for autocorrelation because they were completely flat, while they generate an information content of 0. Secondly, walks that are almost flat result in both a low autocorrelation and a low information content, which could suffice to fully explain the autocorrelation results. This again supports the previous argument, stating that the low information content of code-based fitness is due to its high neutrality and not due to smooth gradients.

Summary RQ1:

The fitness landscape of the code-based fitness function seems to contain many plateaus, while the landscape of the branch distance function seems to do so to a lesser extent.

3.3 RQ2 - Is there a per-objective statistical difference regarding neutrality and ruggedness of the fitness landscape generated by the two fitness functions?

3.3.1 Analysis

For the statistical comparison between the two fitness functions we devise a test modelled after the guidelines proposed by Arcuri and Briand [AB11]. The explained procedure is done for every fitness landscape measure, but for simplicity, the following explanation just speaks of *the measure*. For autocorrelation, $k = 1$ was used to conduct the statistical tests.

As exemplified in table 3.2, we have a list of observations, where each one belongs to one of our two fitness functions, to a certain objective (a branch of an app) and has a certain measure value. For every objective and fitness function, we have 20 measure values. The goal is now to compare the two fitness functions for statistical difference in the measure values for each objective. Since the fitness landscape measures are interval-scale results, we use the (two-sided) Mann-Whitney U-test

[MW47] to test for statistical difference between the measures of the two fitness functions, which results in a p-value for every objective. We summarize their distribution in a histogram. We also visualize the number of branches with $p < 0.05$ and $p \geq 0.05$ in a bar chart.

For all objectives, the effect size by Vargha and Delaney [VD00] is calculated. In our case, the effect size represents the probability, that code-based fitness results in higher measure values than branch distance [AB11]. Their distribution across all objectives is represented in a violin plot for both $p < 0.05$ and $p \geq 0.05$. Since the result is a distribution of a large amount of effect size values, we refrain from calculating the confidence interval for each of those values. Additional representations of the data like bar-charts are added based on the received distributions. Further information on the respective distributions of measure values is not presented, as this would result in an unwieldy amount of data.

3.3.2 Results

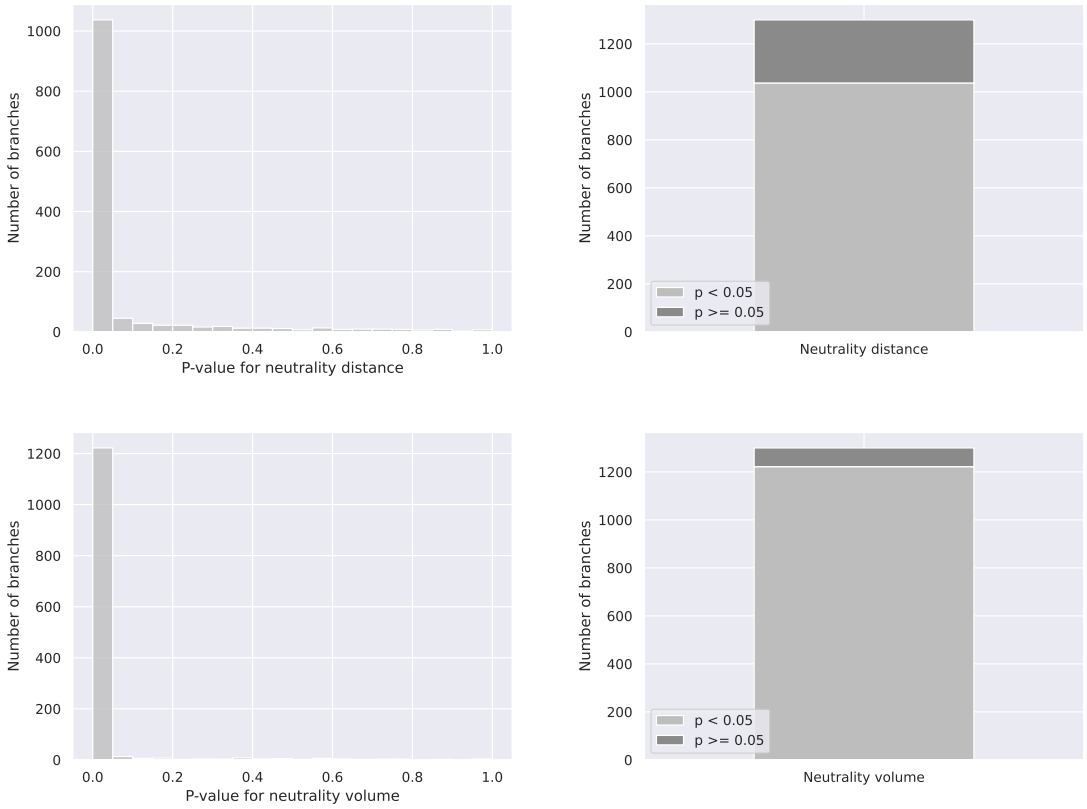

Figure 3.3: Distribution of p-values for neutrality distance and neutrality volume

For every measure, the p-values of the Mann-Whitney U-test are represented in a histogram with bin-width 0.05 and in a bar diagram with one bar for $p < 0.05$ and one for $p \geq 0.05$. The p-values for neutrality distance and neutrality volume can be found in figure 3.3. For neutrality distance, 1037 out of 1300 branches resulted in a p-value smaller than 0.05. For neutrality volume, 1222 fell into that category. The p-values for information content and autocorrelation can be found in

figure 3.4, where 1228 branches had a p-value below 0.05 for information content and 734 for autocorrelation. (For autocorrelation, only 1045 branches are included, because the other branches had only flat walks and thus no autocorrelation values for code-based fitness.)

The Vargha and Delaney effect size values for all measures can be found in figure 3.5. There is a violin plot for those objectives with $p < 0.05$ and one for $p \geq 0.05$. For neutrality volume and information content, a bar plot showing the amount of effect sizes in the intervals $[0, 0]$, $(0, 0.5)$ and $(0.5, 1]$ among the branches with $p < 0.05$ can be seen.

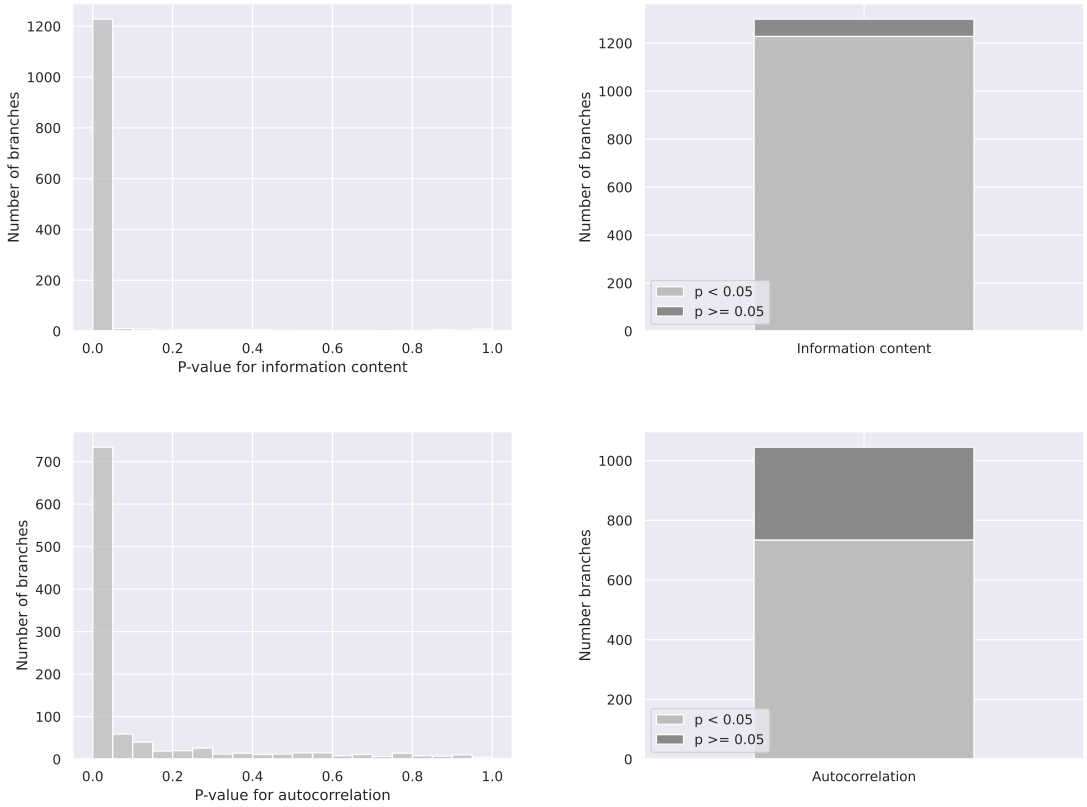

Figure 3.4: Distribution of p-values for information content and autocorrelation

3.3.3 Discussion

Figures 3.3 and 3.4 show that for all measures, the majority of branches result in a p-value smaller than 0.05. For neutrality volume and information content, almost all branches fall into that category, while for neutrality distance and especially autocorrelation, many branches have a higher p-value. This shows us that the number of 20 repetitions was enough to generate significant results, although more repetitions could have further improved the results.

For neutrality distance, the largest peak in the distribution of the effect size values is at nearly 1.0. This means that for many branches, code-based fitness is almost bound to generate higher neutrality distance values than branch distance. For some branches on the other hand, this effect is less pronounced, or even opposite. Unsurprisingly, for $p \geq 0.05$, most effect sizes are close to 0.5.

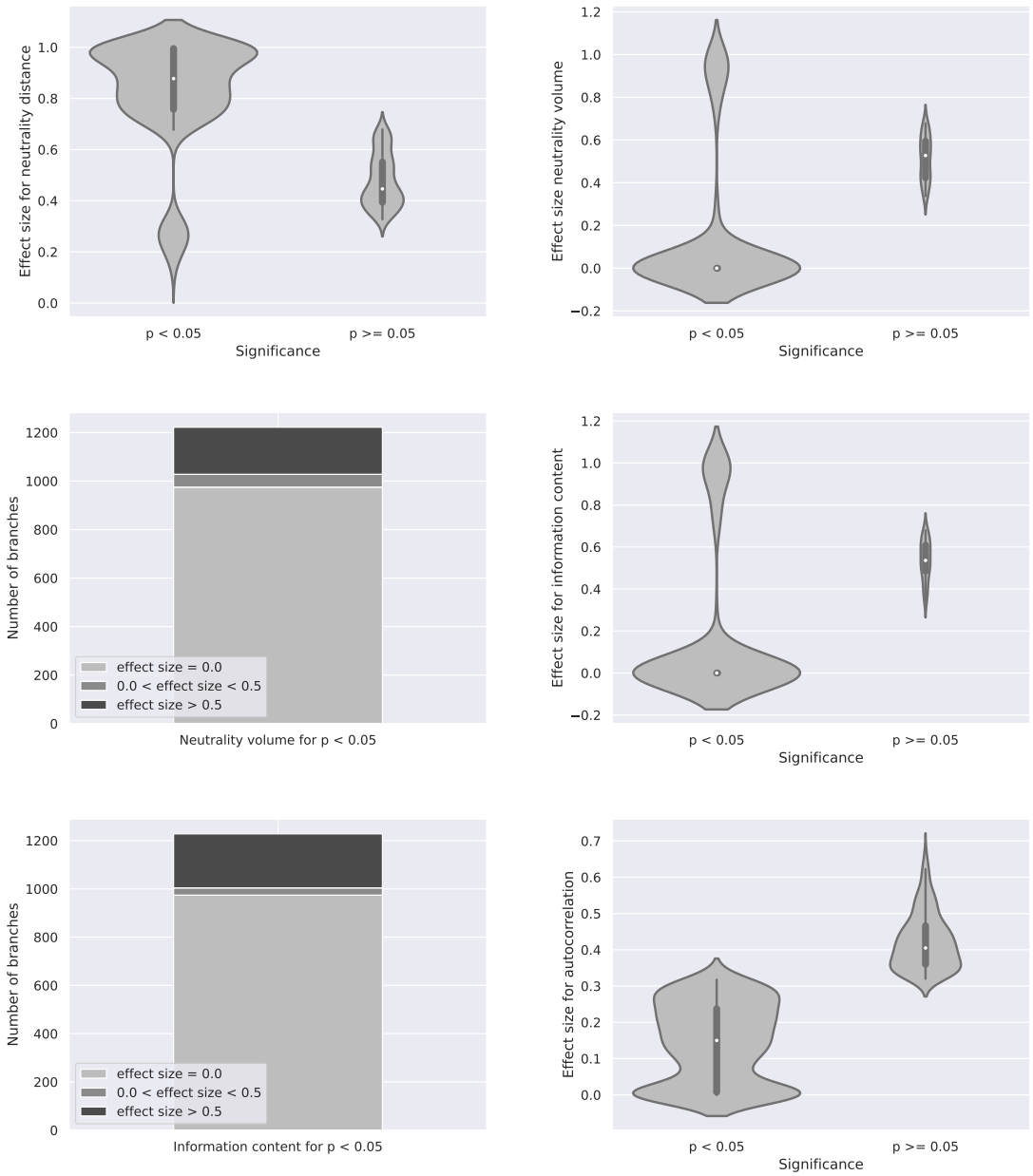

Figure 3.5: Distribution of effect sizes for neutrality distance, neutrality volume, information content and autocorrelation

Regarding neutrality volume, more than two thirds of the objectives have an effect size of 0, which means that for these objectives, code-based fitness is bound to generate lower neutrality volume values than branch distance. The data statistically supports our observations from RQ1, that the fitness landscape of code-based fitness is more neutral then the landscape of branch distance.

For the information content measure, the distribution of effect sizes is almost indiscernible from the one for neutrality volume. This means that for the majority of branches, code-based fitness generates statistically significantly lower information content values than branch distance. This statistically supports our observations from RQ1, that the fitness landscape of code-based fitness has a lower information content.

In the violin plot for autocorrelation, it is visible that all branches with $p < 0.05$ and almost all branch with $p \geq 0.05$ have a statistically significantly lower autocorrelation with code-based fitness, which again reflects the observation from RQ1.

The similarity of the violin plots between information content and the neutrality volume strengthens the suspicion raised in RQ1, that the effects observed regarding the ruggedness measures are mainly caused by the neutrality of the fitness landscape of code-based fitness.

Summary RQ2:

For the majority of branches, there is a statistically significant difference in measure values, which supports the results from RQ1. Further evidence is raised that the results of the ruggedness measures are influenced by the higher neutrality of code-based fitness rather than ruggedness itself.

<

3.4 RQ3 - How do the two fitness functions compare overall regarding the success of the test generation with MIO

3.4.1 Analysis

The first step towards representing the success of the MIO execution is to calculate the success rates from the success values. For every fitness function and objective, we divide the sum of the 20 success values by 20 to receive the proportion of runs that were successful. For both fitness functions, the distribution of these success rates over all objectives is then represented in a violin plot.

3.4.2 Results

We calculated the success rates of MIO for each of the 1300 objectives and for both fitness functions. The violin plots summarizing their distribution can be found in figure 3.6. Since the distribution of the success values for code-based fitness is very extreme, we opted to additionally represent it in a three-way bar plot which shows how many values are zero and one respectively.

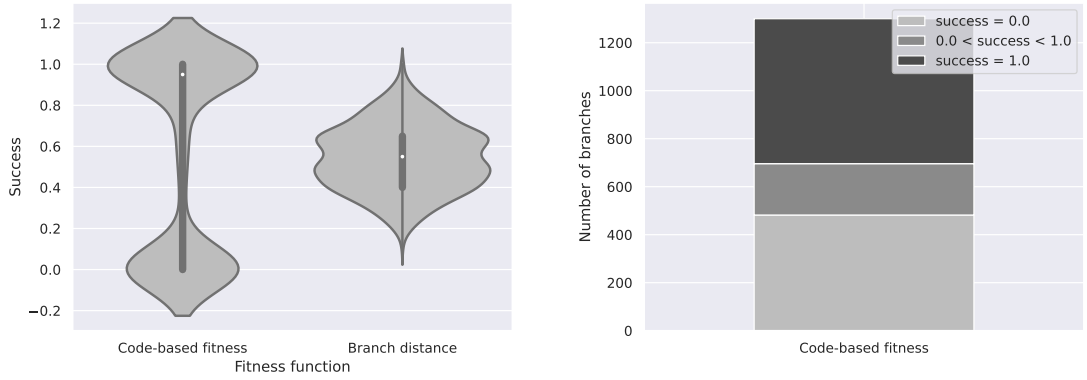

Figure 3.6: Distribution of the success values from the MIO runs

3.4.3 Discussion

Figure 3.6 shows that for code-based fitness, most branches have a success rate of either zero or one. There are more branches with success values of one than there are branches with success values of zero, which is why the median is also very close to one. For branch distance on the other hand, the distribution is rather balanced around a median of roughly 0.5. No branches had a success rate of zero and only three had a success rate of one.

While the median of the success rates of code-based fitness is much higher than the median for branch distance, one has to take into account that branch distance was to some extent successful for every branch, while code-based fitness failed entirely for many branches.

What counts in the end for the tester is the total amount of covered branches, and there are three arguments which do not support that code-based fitness is better based on this decision criterion:

- Firstly, using branch distance, all branches can surely be covered given enough search budget. For code based fitness, it is unclear, weather some branches can be covered at all.
- Secondly, the number of covered branches is a total, which means that the mean can be argued to be more meaningful from a decision-making perspective than the median [HS09]. The mean success is 0.570385 for code-based fitness and 0.543692 for branch distance, which is not a big difference.
- Thirdly, even though code-based fitness lead to many branches with a success rate of 1.0, it is a possibility that this is not caused by the fitness function but by a mechanism of MIO itself. During the search, MIO prioritizes the objectives, where it is easy to make fitness improvements over the ones, where improvements are hard to achieve. It could very well be that MIO is so successful on many branches with code-based fitness, because it saves search budget on the many unsuccessful branches.

Summary RQ3:

With the code-based fitness function, MIO entirely fails on a large set of branches, while the branch distance function is partially successful on all branches. This means that the branch distance function seems more suited for the MIO algorithm than the code-based fitness function.

3.5 RQ4 - Is there a per-objective statistical difference regarding the success of the MIO executions with the two fitness functions?

3.5.1 Analysis

The analysis for RQ4 works almost analogously to RQ2. We again followed the guidelines by Arcuri and Briand [AB11] for the definition of a statistical test.

We want to compare the two fitness functions for per-objective statistical difference in the success values. Now we are dealing with dichotomous results, which is why we use the Fisher exact test and the odds ratio ψ as a measure of effect size. The odds ratio, in our case, expresses which fitness function has a higher chance of success [AB11]. An odds ratio greater than one implies that code-based fitness has a higher chance of success than branch distance, while an odds ratio smaller than one implies the opposite. We add a $\rho = 0.5$ to each cell of the contingency table of the odds ratio, because otherwise cells with a value of zero lead to division by zero. The distribution of the effect sizes is then represented in a violin plot. For plotting the odds ratios, a logarithmic scale is used as suggested by Galbraith [Gal88].

3.5.2 Results

For the MIO runs, the p-values of the Fisher exact test are represented in a histogram with bin-width 0.05 and in a bar diagram with one bar for $p < 0.05$ and one for $p \geq 0.05$ (figure 3.7). In total, of all 1300 branches, 1097 resulted in a p-value smaller than 0.05.

The odds ratio effect size values are plotted in figure 3.8. There is a violin plot for those objectives with $p < 0.05$ and one for $p \geq 0.05$.

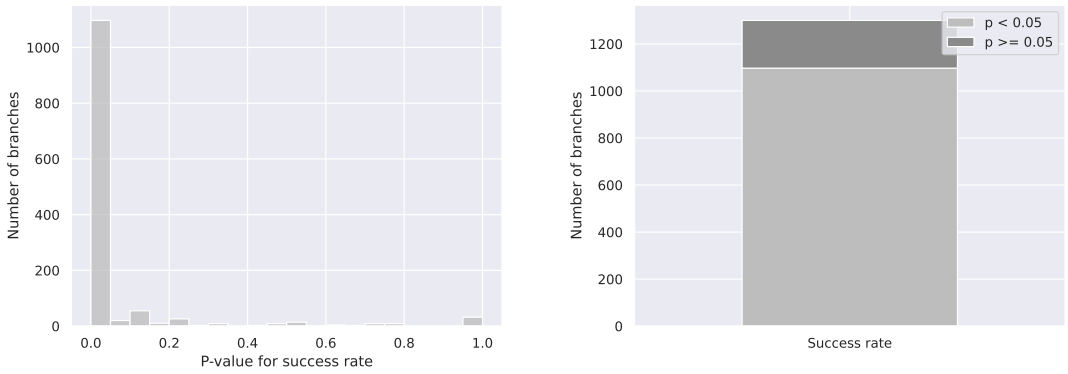

Figure 3.7: Distribution of p-values for the MIO runs

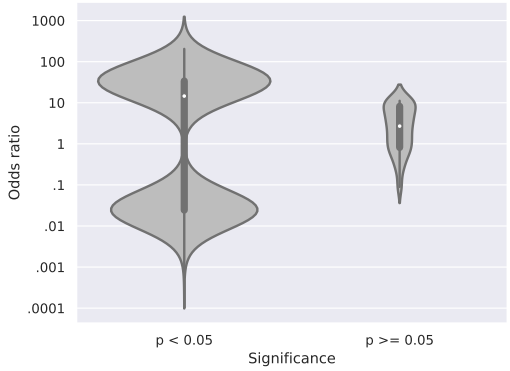

Figure 3.8: Effect size for the MIO runs

3.5.3 Discussion

Looking at figure 3.8, for the branches, where a significant difference in MIO success was found, more branches have an odds ratio > 1 than < 1, which means that there were more branches for which code-based fitness was more successful than there are branches for which branch distance was more successful. As it was established in the discussion of RQ3, this result cannot be used to argue that the code-based fitness function is better than the branch distance fitness function.

Summary RQ4:

There are more branches, for which MIO is more successful with code-based fitness than the other way around. With the argument form RQ3, it can be argued, that this does not show that code-based fitness is better than branch distance.

3.6 RQ5 - Can performance differences between the fitness functions be explained by the properties of the fitness landscape?

3.6.1 Analysis

The analysis of RQ5 is done for every fitness landscape measure, but for simplicity, the following explanation just speaks of *the measure*. Analogously, we speak of only one effect size value for every branch, while in actuality, there are 4, which have to be viewed separately.

The previous RQs resulted in an odds ratio and a Vargha-Delaney effect size for every branch. Based on this data, we now want to find out if a comparatively higher/lower measure value for code-based fitness coincides with a higher/lower success rate. For example, we want to know if branches, where code-based fitness generates a more neutral landscape than branch distance, also have a comparatively lower success rate for code-based fitness. Since we require the relationship between the two fitness functions to be factored into our analysis of association, it does not suffice to calculate the correlation coefficient between measure values and success rates. Rather, we want

to calculate Pearson's correlation between the Vargha-Delaney effect sizes and the odds ratios. This yields a scatter plot, a p-value, and a correlation coefficient. We take as a data basis all the objectives where both the measure and the MIO success yield p-values smaller than 0.05.

Since the main results from RQ1-4 were that code-based fitness generates comparatively high neutrality and comparatively many unsuccessful runs, we are especially interested in the results for the two measures of neutrality. The obvious hypothesis is, that code-based fitness being less successful in the MIO runs coincides with it having a more neutral landscape.

3.6.2 Result

The scatter plots can be found in figure 3.9 and the p- and r-values can be found in table 3.4.

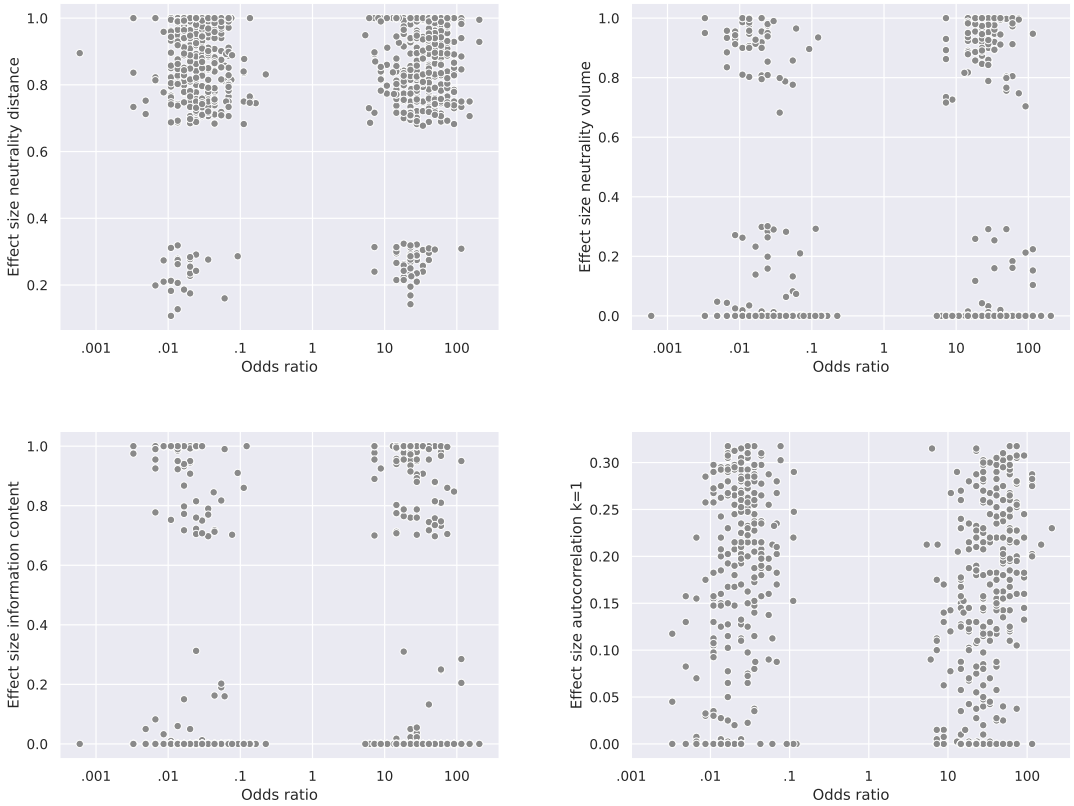

Figure 3.9: Scatter plots for RQ5

Table 3.4: P-values and correlation coefficients

measure	p-value	Pearson's r
neutrality distance	0.69	-0.013
neutrality volume	0.25	0.036
information content	0.51	0.021
autocorrelation	0.11	-0.066

3.6.3 Discussion

As we can see in figure 3.9 and table 3.4, neither is there a visible association between odds ratios and effect sizes in the scatter plots, nor do the p-values suggest a significant correlation. For autocorrelation, the p-value of 0.11 is low enough to at least cautiously suggest a correlation, but the r-value of -0.066 is negligible in magnitude. This also means that the hypothesized correlation between the difference in success and the difference in neutrality could not be found.

Summary RQ5:

No correlation between performance differences and differences in the fitness landscapes could be found.

3.7 Case Study

Unfortunately, the four applied fitness landscape measures could not supply satisfying results regarding the effect of the fitness landscape on MIO search behaviour with the two fitness functions. This is the reason why in a small case study, we look at selected random walks directly to get a more hands-on idea of the fitness landscape and gain impulses on how to improve the fitness landscape analysis for the problem class of Android test generation. The organization of this section is similar to the research questions, but the results and analysis parts are merged.

3.7.1 Analysis

The most surprising result of the research questions was that for code-based fitness, while there is both a sizeable amount of branches for which MIO is unsuccessful and a sizeable amount of branches for which the neutrality is very high, there does not seem to be a noteworthy correlation between these two effects. This is why we choose two branches for the case study, one with a mean success rate of 0.0 with code-based fitness, and one with a mean success rate of 1.0. Since flat walks are not interesting to look at, we exclude them from the case study. For better comparability, we choose the branches only from the app rentalcalc. From both categories, we sample one branch at random. The two resulting branches each yield 20 random walks with code-based fitness. The goal of this section is a hands-on analysis of the fitness sequences from the random walks.

The analysis is mainly based on the following questions:

1. Did the random walks cover the objective at any point?
2. What distinctive features do the walks have?
3. How can these features be explained by the fitness function and the app under test?

3.7.2 Results & Discussion

The sampled branches for the four categories are the 202nd branch and the 59th branch of the application. In table 3.5, these branches are listed with the mean success and (rounded) mean measure values.

Table 3.5: The branches selected for the case study

Objective	Success	Autocorr.	IC	N. distance	N. volume
protect.rentalcalc202	0	0.25	0.30	18.9	76.4
protect.rentalcalc59	1	0.18	0.14	42.7	31.4

protect.rentalcalc202

This objective was covered by every single random walk. The walks had fitness value 0.222222 most of the time but had singular peaks at 1.0. For any of the 20 walks, between 683 and 711 of the fitness values were 0.222222, while between 40 and 68 were 1.0.

What does a code-based fitness value of 0.222222 represent semantically? Most likely, the mathematical result of the fitness calculation would actually be $\frac{2}{9}$. Since $\frac{2}{9} < \frac{1}{3}$, we are in the else part of equation 2.1, which is defined as $0 + \frac{1}{3} \cdot \frac{\max\{i|p_i=\tilde{p}_i\}}{\max\{l,\tilde{l}\}}$. To receive $\frac{2}{9}$ from this equation, it must hold that $\frac{\max\{i|p_i=\tilde{p}_i\}}{\max\{l,\tilde{l}\}} = \frac{2}{3}$. Since the class containing protect.rentalcalc202 is on the third level in the package structure, it holds that $\max\{l, \tilde{l}\} = 3$, which means that $\max\{i|p_i = \tilde{p}_i\} = 2$.

This means that the random walks trivially reached the correct package, but the correct class was harder to find. As soon as code in the correct class was executed, the whole branch was covered. Unfortunately, the fitness function does not provide guidance for finding the correct class in this scenario. In the app rentalcalc, almost all classes are in the same package and many important classes correspond to exactly one activity. Code-based fitness provides no incentive for the search to discover all activities, which would cover many branches across many classes.

protect.rentalcalc59

In contrast to protect.rentalcalc202, this branch has a success rate of 1, meaning it was covered by every MIO run. Interestingly though, it was only covered by 11 of the 20 random walks, while the previous branch was covered by all random walks. For a better overview of the other fitness values that occur in the random walks, we counted the appearances of every distinct fitness value for each

random walk (table 3.6). The value 0.222222 again occurs when the search does not find the correct class, 0.333333 occurs when the correct class is found but not the correct method and 0.904762 occurs when the correct method is covered and there is a specific distance between the nearest covered branch to the target branch. Compared to the previous branch, code-based fitness provides more guidance here, which should have a positive impact on the search. This cannot fully explain though, why this branch was successfully covered by MIO compared to protect.rentalcalc202, because this branch actually generated the value 0.222222 more often than the previous one, meaning that the probability for protect.rentalcalc59 to get any fitness improvement at all is lower than the probability for protect.rentalcalc202 to be covered.

Table 3.6: Occurrences of different fitness values in every random walk for protect.rentalcalc59

	Occurrences of the fitness value...			
Walk	0.222222	0.904762	0.333333	1.000000
0	735	2	13	1
1	730	4	16	1
2	733	5	13	0
3	732	8	10	1
4	732	1	18	0
5	730	1	19	1
6	728	2	19	2
7	744	3	4	0
8	728	6	17	0
9	729	10	12	0
10	731	7	11	2
11	736	2	10	3
12	744	1	6	0
13	733	6	12	0
14	730	11	10	0
15	732	4	10	5
16	730	5	14	2
17	732	7	12	0
18	726	10	12	3
19	727	5	18	1

3.8 Threats to Validity

The set of apps used in our experiments is very small and generalizability to other apps is questionable. Also, due to time constraints, the MIO executions had to be limited to a rather low number of iterations, which might make the performance results of MIO less reliable. The number of iterations for the random walks differs from the de-facto standard of 1000 [Bar98] because of time-constraints during the execution.

4 Related Work

Albunian et al. [AFS20] performed a fitness landscape analysis on the Java unit test generation problem. They used the MOSA algorithm [PKT15], which similarly to MIO is a many-objective algorithm that operates on test cases. They use branch distance as a fitness function and a mutation operator that is based on the random deletion, insertion and editing of statements in a unit test.

In their fitness landscape analysis, they performed random walks and calculated three measures of ruggedness and three measures of neutrality. To learn about the relationship of the fitness landscape to algorithm performance, they correlated the mean landscape measures with the success rates across all branches. Our fitness landscape analysis is largely based on their work, with the largest difference being that we correlated effect sizes instead of the measures themselves because we wanted to factor in the difference between the fitness functions.

The main result of their work was that the fitness landscape of the Java unit test generation is dominated by plateaus and that this neutrality correlates with MOSA success, which suggests that the high neutrality is a major negative influence on algorithm performance. Looking at the source code, they found that the high neutrality is caused by private methods, methods that are hard to call without causing exceptions, as well as boolean comparisons, which result in a binary branch distance and thus do not provide guidance.

Similarly to Albunian et al., Aleti et al. [AMG17] analysed the fitness landscape of the Java unit test generation problem. One major difference is that they evolve whole test suites instead of single tests. This also means that the genetic algorithm they employ is not many-objective. Their fitness function is a mix of method coverage and the sum of all branch distances.

They do not employ random walks for their fitness landscape analysis, but population-based walks, which are created by logging the fitness value from the best individual in every iteration of the genetic algorithm execution. The main focus of their work is on the crossover operator and how important it is for the search, which is why they executed their genetic algorithm with and without the usage of the crossover operator.

One of the results of their work was that the fitness landscape of their problem suffers from neutrality, which they mainly attributed to the used fitness function. As a result of this neutrality, the crossover operator cannot fulfil its main use of escaping local optima. Also, they observed that the fitness function was only weakly correlated to the total branch and method coverage during the execution of the genetic algorithm. From these points, they conclude that the improvement of the fitness function has the potential to improve the overall search performance. Additionally, they conclude that the search is most successful when there is a slow but steady increase in fitness, which is not interrupted by plateaus or large jumps in fitness.

While the two previous works focus on Java unit tests, the work by Vogel et al. [VTG19] focuses on the Android black-box test generation problem. Just like in our work, their tests are sequences of GUI actions. As an algorithm, they use SAPIENZ [MHJ16] with the NSGA-II heuristic [DPAM02],

which is a many-objective algorithm. It operates on test suites, but it can optimize towards a small number of independent fitness functions. In their case, the fitness functions focus on fault revelation, total coverage and test length.

Like Aleti et al., they use fitness landscape measures that are calculated during the search for their fitness landscape analysis. These measures are mainly applied to get information about the diversity of the population during the execution of SAPIENZ. The main takeaway from the fitness landscape analysis is that the diversity of the population could be improved, which is why they extend SAPIENZ with specific measures to boost diversity. They empirically compared their extended version of SAPIENZ – SAPIENZdiv – with the baseline version, but unfortunately, the advantage of SAPIENZdiv could not be confirmed statistically.

5 Conclusion and Future Work

5.1 Conclusion

To automatically generate Android black-box tests with the MIO algorithm, a fitness function is needed, which provides both a good search performance (i.e. covers many branches in few iterations) and a low computational complexity (leading to less time spent on each iteration). In the past, branch distance, the most widely used fitness function in this context, proved to be unwieldy in space and time complexity. This is why in this work, we proposed code-based fitness, a fitness function which is simpler to compute. To compare the search performance, we executed the MIO algorithm with both fitness functions on a set of apps with a set amount of iterations and compared their success rates. To gain further insight into the search behaviour with the two fitness functions, we compared the ruggedness and neutrality of the underlying fitness landscape.

The results showed that while branch distance had decent success rates for all branches, code-based fitness had perfect success rates on many branches but also completely failed on many other branches. The main result regarding the fitness landscape was that code-based fitness generates a more neutral landscape than branch distance, even generating a set of completely flat walks. Unfortunately, we were not able to explain the performance result with the fitness landscape. A small case study showed that the code-based fitness function lacks guidance in finding the correct class when many classes are in the same package and correspond to activities in the GUI.

5.2 Future Work

Direct Analysis of MIO

The biggest open question of this work is why MIO failed on so many branches with code-based fitness. One possible way to answer this question could be to analyse an execution of MIO directly. In its simplest form, one could log the fitness of the best test for each objective in each generation and then analyse the resulting fitness sequences, but more information could be added to the analysis, e.g. whether fitness improvements are usually made by mutation or random sampling.

Bigger Dataset

The selection of apps in this work was very limited, which is why in future experiments of this kind, the number of apps should be expanded, especially to include more complex apps. For this, multiple issues with the MATE framework need to be addressed, like the high memory usage. Also, other performance bottlenecks than the fitness function should be looked into.

Fitness Function Performance Comparison

In this work, it was assumed that code-based fitness is more lightweight than branch distance because that is what both intuition and practical experience suggest. However, to get a better understanding of the computational performance of both fitness functions, an empirical experiment should be conducted.

Improve Fitness Landscape Analysis

The fitness landscape measures employed in this work could not explain the success rates of the MIO execution. To gain insight into why they did not work and maybe find measures that do work, the case study of random walks could be expanded and different measures could be tried out on the data.

Bibliography

[AB11] A. Arcuri, L. Briand. “A practical guide for using statistical tests to assess randomized algorithms in software engineering”. In: *Proceedings of the 33rd international conference on software engineering*. 2011, pp. 1–10 (cit. on pp. 33, 34, 39).

[AFS20] N. Albunian, G. Fraser, D. Sudholt. “Causes and effects of fitness landscapes in unit test generation”. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. 2020, pp. 1204–1212 (cit. on pp. 13, 20, 23–25, 45).

[AMG17] A. Aleti, I. Moser, L. Grunske. “Analysing the fitness landscape of search-based software testing problems”. In: *Automated Software Engineering* 24.3 (2017), pp. 603–621 (cit. on pp. 13, 45).

[Arc18] A. Arcuri. “Test suite generation with the Many Independent Objective (MIO) algorithm”. In: *Information and Software Technology* 104 (2018), pp. 195–206 (cit. on pp. 13, 16, 29).

[Aue22] M. Auer. *Instrumentation*. Sept. 23, 2022. URL: `https://gitlab.infosun.fim.uni-passau.de/auermich/instrumentation` (cit. on p. 28).

[Bar98] L. Barnett. “Ruggedness and neutrality-the NKp family of fitness landscapes”. In: *Artificial Life VI: Proceedings of the sixth international conference on Artificial life*. 1998, pp. 18–27 (cit. on pp. 28, 44).

[BDGG09] L. Bianchi, M. Dorigo, L. M. Gambardella, W. J. Gutjahr. “A survey on metaheuristics for stochastic combinatorial optimization”. In: *Natural Computing* 8.2 (2009), pp. 239–287 (cit. on p. 15).

[BR03] C. Blum, A. Roli. “Metaheuristics in combinatorial optimization: Overview and conceptual comparison”. In: *ACM computing surveys (CSUR)* 35.3 (2003), pp. 268–308 (cit. on p. 15).

[Dev22] M. Developers. *MATE*. Sept. 14, 2022. URL: `https://github.com/mate-android-testing` (cit. on pp. 13, 22).

[DPAM02] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197 (cit. on p. 45).

[ERGF18] M. M. Eler, J. M. Rojas, Y. Ge, G. Fraser. “Automated accessibility testing of mobile apps”. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 116–126 (cit. on p. 13).

[FA12] G. Fraser, A. Arcuri. “Whole test suite generation”. In: *IEEE Transactions on Software Engineering* 39.2 (2012), pp. 276–291 (cit. on pp. 13, 16).

[FA14] G. Fraser, A. Arcuri. “A large-scale evaluation of automated unit test generation using evosuite”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (2014), pp. 1–42 (cit. on p. 13).

[Gal88] R. Galbraith. “A note on graphical presentation of estimated odds ratios from several clinical trials”. In: *Statistics in medicine* 7.8 (1988), pp. 889–894 (cit. on p. 39).

[Gru22] B. Gruver. *smali*. Sept. 19, 2022. URL: https://github.com/JesusFreke/smali (cit. on p. 21).

[HN98] J. L. Hintze, R. D. Nelson. “Violin plots: a box plot-density trace synergism”. In: *The American Statistician* 52.2 (1998), pp. 181–184 (cit. on p. 30).

[HS09] M. M. Holt, S. M. Scariano. “Mean, median and mode from a decision perspective”. In: *Journal of Statistics Education* 17.3 (2009) (cit. on p. 38).

[Kor90] B. Korel. “Automated software test data generation”. In: *IEEE Transactions on software engineering* 16.8 (1990), pp. 870–879 (cit. on pp. 13, 20).

[McM04] P. McMinn. “Search-based software test data generation: a survey”. In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156 (cit. on p. 13).

[ME09] K. M. Malan, A. P. Engelbrecht. “Quantifying ruggedness of continuous landscapes using entropy”. In: *2009 IEEE Congress on evolutionary computation*. IEEE. 2009, pp. 1440–1447 (cit. on p. 26).

[MHJ16] K. Mao, M. Harman, Y. Jia. “Sapienz: Multi-objective automated testing for android applications”. In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 94–105 (cit. on pp. 13, 45).

[MW47] H. B. Mann, D. R. Whitney. “On a test of whether one of two random variables is stochastically larger than the other”. In: *The annals of mathematical statistics* (1947), pp. 50–60 (cit. on p. 34).

[PA12] E. Pitzer, M. Affenzeller. “A comprehensive survey on fitness landscape analysis”. In: *Recent advances in intelligent engineering systems* (2012), pp. 161–191 (cit. on pp. 13, 23, 24).

[PKT15] A. Panichella, F. M. Kifetew, P. Tonella. “Reformulating branch coverage as a many-objective optimization problem”. In: *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE. 2015, pp. 1–10 (cit. on pp. 13, 45).

[PKT17] A. Panichella, F. M. Kifetew, P. Tonella. “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets”. In: *IEEE Transactions on Software Engineering* 44.2 (2017), pp. 122–158 (cit. on pp. 13, 16).

[PKT18] A. Panichella, F. M. Kifetew, P. Tonella. “A large scale empirical comparison of state-of-the-art search-based test case generators”. In: *Information and Software Technology* 104 (2018), pp. 236–256 (cit. on p. 13).

[SAF+19] L. Sell, M. Auer, C. Frädrich, M. Gruber, P. Werli, G. Fraser. “An empirical evaluation of search algorithms for app testing”. In: *IFIP International Conference on Testing Software and Systems*. Springer. 2019, pp. 123–139 (cit. on pp. 13, 16, 22, 27).

[VD00] A. Vargha, H. D. Delaney. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132 (cit. on p. 34).

[VFM00] V. K. Vassilev, T. C. Fogarty, J. F. Miller. “Information characteristics and the structure of landscapes”. In: *Evolutionary computation* 8.1 (2000), pp. 31–60 (cit. on pp. 24, 26).

[VTG19] T. Vogel, C. Tran, L. Grunske. “Does diversity improve the test suite generation for mobile applications?” In: *International Symposium on Search Based Software Engineering*. Springer. 2019, pp. 58–74 (cit. on pp. 13, 45).

[WBP02] J. Wegener, K. Buhr, H. Pohlheim. “Automatic test data generation for structural testing of embedded software systems by evolutionary testing”. In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. 2002, pp. 1233–1240 (cit. on p. 13).

[WBS01] J. Wegener, A. Baresel, H. Sthamer. “Evolutionary test environment for automatic structural testing”. In: *Information and software technology* 43.14 (2001), pp. 841–854 (cit. on pp. 13, 20).

[Wri+32] S. Wright et al. “The roles of mutation, inbreeding, crossbreeding, and selection in evolution”. In: *Proceedings of the Sixth International Congress on Genetics* (1932) (cit. on p. 22).

All links were last followed on January 30, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature