Causes and Effects of Fitness Landscapes in Unit Test Generation

Nasser Albunian
Department of Computer Science
University of Sheffield, Sheffield, UK

Gordon Fraser
Chair of Software Engineering II
University of Passau, Germany

Dirk Sudholt
Department of Computer Science
University of Sheffield, Sheffield, UK

ABSTRACT

Search-based unit test generation applies evolutionary search to maximize code coverage. Although the performance of this approach is often good, sometimes it is not, and how the fitness landscape affects this performance is poorly understood. This paper presents a thorough analysis of 331 Java classes by (i) characterizing their fitness landscape using six established fitness landscape measures, (ii) analyzing the impact of these fitness landscape measures on the search, and (iii) investigating the underlying properties of the source code influencing these measures. Our results reveal that classical indicators for rugged fitness landscapes suggest well searchable problems in the case of unit test generation, but the fitness landscape for most problem instances is dominated by detrimental plateaus. A closer look at the underlying source code suggests that these plateaus are frequently caused by code in private methods, methods throwing exceptions, and boolean flags. This suggests that inter-procedural distance metrics and testability transformations could improve search-based test generation.

KEYWORDS

Fitness landscape analysis, Search-Based Test Generation, Empirical Software Engineering, Genetic Algorithm

ACM Reference Format:
Nasser Albunian, Gordon Fraser, and Dirk Sudholt. 2020. Causes and Effects of Fitness Landscapes in Unit Test Generation. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3390194

1 INTRODUCTION

As software testing is a time-consuming, laborious, and error-prone task, developers can choose to generate tests automatically. In the context of unit testing object-oriented software, where tests are sequences of calls on a class under test, Genetic Algorithms (GAs) have been successfully applied for generating tests. Several studies [8, 9, 22] have shown that GAs are effective at generating tests that achieve high code coverage. However, they are still far from being able to satisfy all test goals (e.g., covering all branches) [5, 27].

While some general limitations are known (e.g., the challenges of generating complex parameter objects [10, 27]), there is a lack of understanding of the search behavior during the optimization, making it difficult to identify the factors that make a search problem difficult. Such an understanding can be provided by investigating the underlying structure of the search space and the influence of its features on the optimization process. The concept of the fitness landscape is among the most commonly used metaphors to give an intuitive understanding of the search space structure and help in predicting search behavior with different search problems. Analyzing the fitness landscape helps in identifying the properties that are related to the problem difficulty [1]. The two main properties of fitness landscapes that are known to have a great influence on the optimization process are *ruggedness* and *neutrality* [16]. The interplay of these properties has motivated the development of several techniques that study the structure of fitness landscapes.

The aim of this paper is to analyze the fitness landscape and investigate the impact of its properties on the generation of unit tests. More specifically, we study the influence of the two landscape properties, ruggedness and neutrality, on unit test generation. Fitness landscape analysis uses different proxy measurements to gather evidence on these properties, usually by analyzing the way fitness values change while randomly walking across the search space. In this paper, we apply the six most common such measurements to investigate random walks on a selection of 331 Java classes. By contrasting the resulting metrics with the performance of a GA on generating tests for these problem instances, we can identify how they affect the search, and what aspects of the underlying source code causes these properties.

Our experiments suggest that the landscape structure is mostly dominated by neutral areas, i.e., plateaus, which makes it harder for the search to find test inputs. Although ruggedness is often considered a negative property of the fitness landscape, in the case of unit test generation and the scale of ruggedness observed there, we find that higher ruggedness is an indicator of more informative landscapes, resulting in better performance of the search. A closer look at the causes of neutrality suggests that influential factors are (1) whether the target code is contained in private methods, for which there is no direct guidance provided by the fitness function; (2) whether the code has preconditions that are difficult to satisfy and cause exceptions when violated; and (3) the prevalence of boolean flags, which provide no guidance to the search. This suggests that the search could be improved by enhancing the existing fitness functions to consider inter-procedural distance information, by addressing the problem of generating *valid* complex objects, and by applying testability transformations.

2 BACKGROUND

2.1 Search-Based Software Testing (SBST)

Search-Based Software Testing (SBST) describes the application of meta-heuristic optimization techniques to the automation of various software testing tasks. In particular, SBST is frequently

GECCO '20, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7128-5/20/07…$15.00
https://doi.org/10.1145/3377930.3390194

applied to generate test data [18] with optimization goals based on different notions of code coverage. When testing individual functions then local search algorithms such as hill climbing have been used successfully [12]; in other domains, such as unit testing, GAs are more common [17].

In a GA, a population of candidate solutions is gradually evolved towards an optimal solution. The algorithm typically starts with a population of random individuals that will be iteratively evolved over many generations. In each generation, the processes of natural evolution are mimicked: Every individual in the population is evaluated by a fitness function, which determines how close this individual is to the desired solution. The fitter an individual, the more likely it is selected from the current population and used for recombination using crossover and mutation operators while building the next generation of the GA population.

2.2 A Genetic Algorithm for Unit Tests

In the context of generating tests for object-oriented programs, a common approach lies in evolving sets of unit tests [7] using single-objective optimization, or individual test cases using many-objective optimization [20]. The many-objective approach has been shown to generally perform best [5, 21, 22].

Representation: A solution that is represented as a test case τ consists of a sequence of calls $\tau = \langle s_1, s_2, \ldots, s_n \rangle$ on the class under test (CUT) [7]. That is, each s_j is an invocation of a constructor of the CUT, a method call on an instance of the CUT, a call on a dependency class in order to generate or modify dependency objects, or it defines a primitive value (e.g., number, string, etc.). As the ideal test case size is not known a priori, the number of statements in a test case is variable and can be changed by the search operators.

Crossover: The common crossover operator in the context of test case optimization works as follows: Given two parent test cases τ_1 and τ_2, a random value x in the range $(0, 1)$ is selected. The first offspring will contain the first $x \cdot |\tau_1|$ statements from τ_1, followed by the last $(1 - x) \cdot |\tau_2|$ statements from τ_2. The second offspring will contain the first $x \cdot |\tau_2|$ statements from τ_2, followed by the last $(1 - x) \cdot |\tau_1|$ statements from τ_1. As the size of individuals is not fixed, this operator ensures that offspring do not grow larger than their parents during crossover. Since there can be dependencies between statements within a test, the crossover possibly needs to repair the offspring to ensure validity, e.g., by generating additional statements for missing dependencies.

Mutation: When a test case τ is mutated, each statement in τ is deleted or edited with probability $\frac{1}{|\tau|}$, whereas insertion is applied at a random position with probability σ; if a statement is added, then another one is inserted with probability σ^2, then with σ^3, etc. A challenge lies in ensuring that these operations maintain the syntactic validity of the statements, for example by recursively inserting calls that create and modify dependency objects.

Fitness function: The fitness function used to guide the search is based on code coverage. Various different criteria as well as combinations of criteria have been proposed in the literature [25]. In this paper, we focus on branch coverage, because it is one of the most common coverage criteria in practice [7], and fitness functions for other criteria are typically based on branch coverage fitness calculations [32]. In the many-objective representation of the unit test generation problem [20], each branch in the CUT is considered as a single objective to be optimized. In this case, the fitness function of a branch b_i is typically calculated as follows:

$$f(\tau, b_i) = al(b_i, \tau) \; + \; \alpha(bd(b_i, \tau)) \tag{1}$$

Here τ is an individual test case to be evaluated, bd is the branch distance [15], α is a normalization function that normalizes the branch distance in the range $[0, 1]$ [2], and al is the approach level [32]:

- The branch distance [15] is the basis of many coverage-based fitness functions, and estimates a distance for a given conditional statement to become true or false. For example, when the if-condition `if(x == 42)` is executed with `x` equal to 0, then the distance to the condition evaluating to true is $|42 - x| = 42$, whereas if x is 40, then the distance is $|42 - x| = 2$. If the condition evaluates to true, then the distance is 0. The distance to the condition evaluating to false can be calculated analogously.
- The approach level [32] 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. The branch distance for $f(\tau, b_i)$ is calculated for this control dependency.

A test case τ_x is better than a test case τ_y if and only if τ_x has a lower approach level + normalized branch distance for a branch b_i.

2.3 Fitness Landscape Analysis

A greater understanding of the behavior of combinatorial optimization algorithms comes from a thorough analysis of the underlying topological structure of the search space. This topological structure over which the search is being executed is known as the *fitness landscape*, a term that was first introduced by Sewall Wright [33]. More formally, a fitness landscape (S, f, N) of a problem instance for a given optimization problem consists of a set of genotypes S that represent the problem solutions, a fitness function $f : S \to \mathbb{R}$ that maps each genotype to a numerical fitness value, and a genetic operator N that defines the neighborhood relationship between the genotypes. Given a specific landscape structure, an optimization algorithm can be thought of as navigating this structure in order to find optimal or near-optimal solutions. However, the structure of a fitness landscape is completely defined by several landscape features [16]. Among these features are *ruggedness* and *neutrality* that both have an explicit impact on the ability of the optimization algorithm at finding optimal solutions.

Ruggedness: A fitness landscape is said to be rugged if the landscape contains multiple local optima and an isolated global optimum, and if the fitness values of neighboring individuals are less correlated. In this case, the search of an optimal solution is thought to become harder as the algorithm might get trapped in local optima and result in sub-optimal solutions. Ruggedness can be analyzed based on different types of landscape walks [23], i.e., randomized explorations of the search space. Among these walks is the *random walk*, which starts at a randomly initialized individual in the landscape and then arbitrarily moves in each step to neighboring individuals using the genetic operator N. Several studies show that

the random walk is effective in describing the features of the fitness landscape using different fitness landscape analysis metrics [16, 23].
Neutrality: Ruggedness alone is not enough to measure the search difficulty if equilibrium periods dominate the process of evolution. Such periods result in a set of neighboring genotypes that have the same fitness value. The presence of these periods in a landscape defines the concept of neutrality [24]. A neutral fitness landscape can be thought of as a landscape with many plateaus. In this case, the mutation in a neutral fitness landscape produces mainly movements in genotype space with no effects on the fitness. A neighbor x of a solution y is said to be a neutral neighbor if $f(x) = f(y)$. In order to obtain a comprehensive picture of a neutral landscape, a *neutral walk* can be used, which is a variation of a random walk that accepts only neighbors with identical fitness values, i.e., the area of neutral neighbors.

2.4 Fitness Landscape Measurements

The sequence of fitness values that are obtained from a random walk can be used to analyze the structure of the fitness landscape. Based on that, different statistical measures have been proposed [23] to measure both ruggedness and neutrality:

Measure 1: Autocorrelation (AC) is a well-known measure of ruggedness that is applied on the sequence of fitness values of the random walk to measure the correlation between the fitness of each two individuals that are i steps away. It thus can be calculated as follows:

$$r(s) = \frac{\sum_{i=1}^{N-s}(f_i - \overline{f})(f_{i+s} - \overline{f})}{\sum_{i=1}^{N}(f_i - \overline{f})^2} \tag{2}$$

where N is the total number of the individuals of the random walk, s is the step size, f_i is the fitness of the i^{th} individual, and $\overline{f}$ is the mean fitness of all the individuals. The resulting value is in the range of -1 to 1. The landscape is more rugged when the AC value is close to 0 meaning that the individuals of the random walk are less correlated.

Measure 2: Neutrality Distance (ND) is a measure of neutrality in a landscape. It measures the number of neutral steps made at the start of the random walk. More formally, for a random walk $x_1, x_2, \ldots,$ ND is the largest t such that $f(x_1) = f(x_2) = \cdots = f(x_t)$.

Measure 3: Neutrality Volume (NV) is another measure of neutrality based on the number of neighboring 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.

Besides these measures, additional measures to gain further information about the structure of the landscape have been proposed [30] based on information analysis. These measures depend on the sequence of the fitness values that are obtained from the random walk. However, instead of directly using the fitness values of the random walk, the following steps are applied:

Step 1: The sequence of fitness values $\{f_t\}_{t=1}^{n}$ is first transformed into a series of fitness changes:

$$\Delta\{f_t\}_{t=1}^{n} := \{f_t - f_{t-1}\}_{t=2}^{n} \tag{3}$$

Step 2: The series of fitness changes is represented as an ensemble of objects that can be defined as a string $S(\epsilon) = s_1, s_2, \ldots, s_n$ of symbols $s_i \in \{\bar{1}, 0, 1\}$ given by:

$$s_i = \begin{cases} \bar{1}, & \text{if } x < -\epsilon \\ 0, & \text{if } |x| \leq \epsilon \\ 1, & \text{if } x > \epsilon \end{cases} \tag{4}$$

where x corresponds to each of the fitness changes that are resulted from equation 3. The parameter ϵ is a real number that is taken from the interval $[0, l_n]$, where l_n is the length of the interval of the fitness values that are obtained by the random walk.

Measure 4: Information content (IC) is designed to capture the variety of shapes in the string $S(\epsilon)$ in order to analyze the ruggedness of the landscape. It is an entropy measure of the number of consecutive symbols that are not equal in the string $S(\epsilon)$. It can be calculated using the formula:

$$H(\epsilon) = -\sum_{p \neq q} P_{[pq]} \log_6 P_{[pq]} \tag{5}$$

The probabilities $P_{[pq]}$ are frequencies of the possible blocks pq of elements from the set $\{\bar{1}, 0, 1\}$, and are defined as:

$$P_{[pq]} = \frac{n_{[pq]}}{n} \tag{6}$$

where $n_{[pq]}$ is the number of occurrences of each pq in the string $S(\epsilon)$. Note that the value of $H(\epsilon)$ increases with an increase in the number of peaks in the landscape.

Measure 5: Partial information content (PIC) is designed to analyze the modality of the landscape by filtering the string $S(\epsilon)$ into $S'(\epsilon)$ removing all zeros and all symbols that equal their preceding symbol. In this case, the new string $S'(\epsilon)$ has the form $\{\bar{1}, 1, \bar{1}, \ldots\}$. The partial information content can then be calculated as:

$$M(\epsilon) = \frac{\mu}{n} \tag{7}$$

where μ is the length of the string $S'(\epsilon)$ and n is the length of the string $S(\epsilon)$. If the landscape path is maximally multimodal, $M(\epsilon)$ is 1 as the string $S'(\epsilon)$ is identical to $S(\epsilon)$ (i.e., $S(\epsilon)$ cannot be modified). In contrast, the landscape path is flat when the $M(\epsilon)$ is 0 as there are no slopes in the landscape path.

Measure 6: Density-basin information (DBI) estimates the variety of flat areas in the landscape. It captures the information of smooth points by only considering the equal consecutive symbols in the string $S(\epsilon)$. In this context, the only possible sub-blocks of the string symbols are $00, 11, \bar{1}\bar{1}$, and the entropic measure is defined as:

$$h(\epsilon) = -\sum_{p=q} P_{[pp]} \log_3 P_{[pp]} \tag{8}$$

Therefore, a high value of $h(\epsilon)$ indicates a low density of peaks in the landscape, and thus that the landscape structure is dominated by flat areas.

3 EMPIRICAL STUDY

Our aim is to analyze how unit test generation is influenced by the fitness landscape, and understand how the landscape properties relate to features of Java classes. We therefore designed a study to answer the following research questions:

RQ 1: What are the properties of the fitness landscape for the JUnit test generation problem?
RQ 2: How do the fitness landscape properties affect the search behavior?
RQ 3: What are the underlying properties of source code that influence the fitness landscape?

3.1 Experimental Setup

Selection of Classes Under Test: Choosing a diverse set of classes is important in studying the properties of the fitness landscape since the features of Java classes might have an impact on the landscape properties. Therefore, we used the selection of 346 complex and non-trivial classes from the DynaMOSA study [21] where the complexity of classes ranges from 2 to 7939 branches. The complexity of the selected classes is intended to ensure that their branches are not covered easily in the initial population.

Unit Test Generation Tool: Among the popular tools that generate tests for Java programs using an evolutionary algorithm is EvoSuite [6]. It generates JUnit test suites for a given Java CUT and target coverage criterion using different evolutionary algorithms, with the Many-Objective Sorting Algorithm (MOSA) being the most effective algorithm for JUnit test generation [5, 22].

Experiment Procedure: To better understand the influence of the fitness landscape properties on the generation of JUnit tests, we conducted an experiment that involves (i) applying random walks on each CUT, and then (ii) applying all the six fitness landscape measures (described in Section 2.4) on the sequence of fitness values obtained by the two types of landscape walks. To perform a walk on a landscape, we applied the corresponding mutation operator in order to move from one landscape point to another where each point in a landscape corresponds to one step of the walk.

In order to perform the experiment, we implemented and ran random walk in EvoSuite. We also ran the MOSA algorithm in order to compare its performance against the fitness landscape measures. To minimise the influence of other optimizations, we used a "vanilla" configuration [6] and default settings [3] with only branch coverage as target criterion. The search stopping criterion was set to be a one minute timeout, which is EvoSuite's default search budget. As a pre-required step to run the random walk, we consider the most commonly used number of random walk steps in the literature, which is 1000 [4]. We ran EvoSuite 30 times on each class in order to account for the randomness of the algorithm under consideration and the two landscape walks.

Running this experiment on the corpus of 346 classes resulted in data for only 331 classes. This is due to the environmental dependencies of 8 classes that are difficult to fulfill by EvoSuite, and the search timeout was reached for 7 classes because of constraints that cannot be solved within a specific time [8].

RQ1 Analysis: Given a class X with n branches, a landscape walk of m steps on X is defined as a sequence $x_1, x_2, \ldots$ such that x_{i+1} is the outcome of a mutation applied to x_i where an initial individual x_i is created randomly by applying the insertion mutation repeatedly. For each step in the walk there will be n fitness values, as there are n branches in the CUT. Table 1 contains an example random walk of 6 steps on a class with 5 branches, resulting in 5 fitness values for each step. Each of the landscape measures is applied to the sequence of 6 fitness values for each branch. For example, applying the autocorrelation measure, defined in Section 2.4, on the sequence of fitness values results in 0.1668 for branch 1, 0.166415 for branch 2, 1.667 for branch 3, -0.20905 for branch 4, and 0.3064 for branch 5. To answer RQ1, we consider the distribution of these values across all branches.

Table 1: An example of applying the random walk of 6 steps on a class with 5 branches

Step	b1	b2	b3	b4	b5
1	1.5	0.99304409	0.5	0.9375	0.2235
2	1.5	0.99304409	0.5	0.9315	0.2233
3	1.4	0.99304261	0.4	0.9315	0.2229
4	1.4	0.99304261	0.4	0.9363	0.2229
5	1.4	0.99304409	0.5	0.9363	0.2229
6	1.5	0.99304409	0.5	0.9315	0.2225

RQ2 Analysis: In order to understand the influence of the landscape properties on the search behavior, we want to understand how it affects the ability of the GA to cover the branches. While the overall performance of the search is usually measured in terms of the resulting branch coverage, we need to consider individual branches, where the outcome is dichotomous (i.e., either the branch is covered, or it is not). We define a *Success Rate (SR)* for MOSA for each branch as the fraction of runs in which MOSA covers the branch at least once. For example, if we run MOSA five times and in two cases branch b_i is covered by the resulting test suite, then the SR equals $2/5 = 0.2$. However, correlating the SR value for one branch with the m values of a landscape measure of that specific branch requires the use of an appropriate measure of central tendency such as the average of the m values. This results in a single value of the landscape measure that can be correlated with the SR value. For example, consider the following results of the AC measure with the two branches where each of the five results represents the AC value of a single run of random walk: $b_1 \rightarrow \{0.90, 0.92, 0.84, 0.89, 0.90\}$ and $b_2 \rightarrow \{0.84, 0.90, 0.76, 0.56, 0.97\}$. In this case, the average of the five runs for each branch is correlated with the SR value of that branch: $b_1 \rightarrow \{\text{SR} : 0.2, \text{AC} : 0.89\}$ and $b_2 \rightarrow \{\text{SR} : 1, \text{AC} : 0.81\}$.

RQ3 Analysis: To answer the third research question we compare the success rates of the search and random walks, such that we can distinguish between branches that are trivially covered, branches that are impossible to cover, and branches that can be covered with reasonable search effort. Given this distinction, we can then compare the different branches with respect to their landscape and source code properties.

All the data presented in this paper and the scripts needed to reproduce the experiment are available at https://github.com/nasser-albunian/fitness-landscape-study.git.

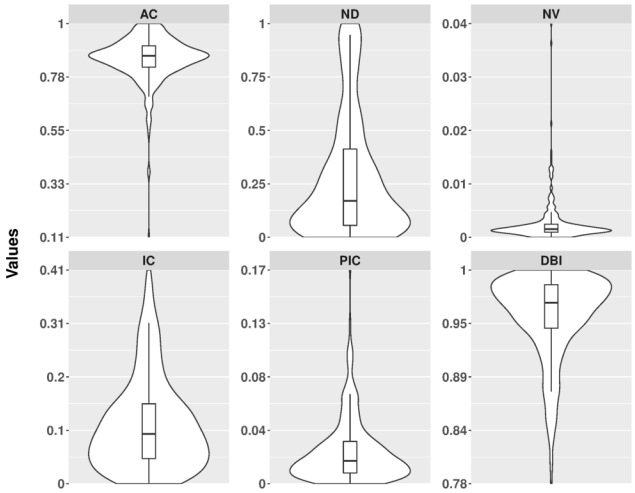

Figure 1: Results of the six fitness landscape measures applied on the branches of the 331 classes

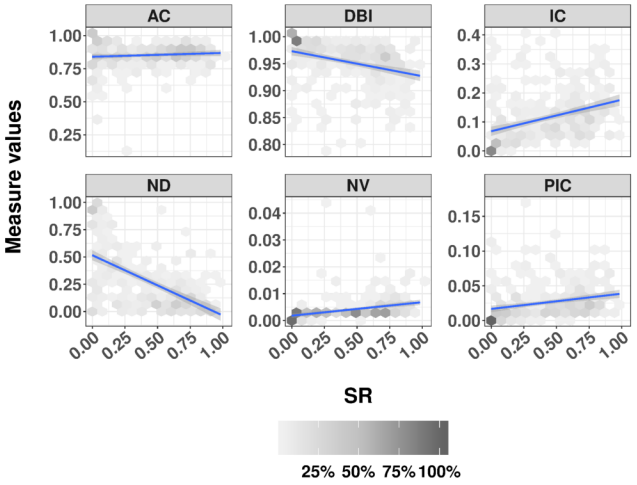

Figure 2: The Spearman correlation of SR with each of the six measures for all the branches of 331 classes. The correlation coefficient of SR and AC is 0.04, ND is -0.34, NV is 0.41, IC is 0.488, PIC is 0.476, and DBI is -0.481

3.2 Threats to Validity

To control threats of the stochastic behavior of both techniques, i.e., MOSA and the random walk, we repeated the experiment 30 times. Although we used a selection of 331 complex classes with a diverse number of branches, which was also used by previous studies [21], our results may not generalize to other classes. Choosing the number of steps of the random walk as 1000 is common practice [4]. The search budget used in running MOSA is based on EvoSuite's default search budget of one minute, which is examined previously to assess the performance of MOSA [20].

3.3 RQ1 — What are the properties of the fitness landscape for the JUnit test generation problem?

The results of applying the six fitness landscape measures on the series of fitness values obtained by the random walk are shown in Figure 1. In general, all the measures indicate that the fitness landscape is mostly dominated by plateaus, i.e., that the landscape is flat. Looking at the results of the AC measure, the AC values for most of the branches are higher than 0.6, which is interpreted as highly correlated fitness values of the random walk, and thus indicate a smooth landscape.

The ND measure indicates that, on average, the first 20% steps of the random walk are all neutral steps, which is strong evidence of plateaus in the landscape. The NV measure indicates a small number of neighboring areas of individuals with equal fitness during the random walk with most of the branches, i.e., NV $\approx$ 5, which also indicates a landscape with flat areas.

For the information-based measures, the 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. Our results show that the IC with many branches is close to 0.1. This indicates a landscape with a low number of peaks, and thus many flat areas. The PIC measure is an estimate of modality in the landscape where PIC = 0 when the landscape is flat and has no slopes, whereas PIC = 1 when the landscape is maximally multimodal, i.e., the number of optima is high. Our results reveal that the PIC values with most of the branches are lower than 0.04, indicating that the landscape is mostly flat and has few slopes. In contrast, the DBI measure estimates the variety of flat areas where the density of peaks in the landscape is low and the flat areas are more prominent when the DBI is high, i.e., close to 1. Our results show that most of the branches result in DBI higher than 0.9, indicating a landscape with a low density of peaks.

Although the fitness landscape of a large number of branches is dominated by plateaus, several branches seemed to point to the existence of rugged areas in their fitness landscape. This can be seen with the branches where the landscape measures result in lower values such as the case with the AC (< 0.4), and higher values such as the case with the IC (> 0.4), although they do not indicate a fully rugged landscape [30].

> ***RQ 1**: Neutrality seems to dominate much of the fitness landscape for most of the branches, although there are some exceptions of branches with more rugged fitness landscapes.*

3.4 RQ2 — How do the fitness landscape properties affect the search behaviour?

In order to understand the impact of the fitness landscape properties on the test generation, we investigate the Spearman correlation of the SR and each of the landscape measures, as shown in Figure 2. Each hexagon represents a set of runs of different branches in which the hexagon density increases with an increase in the number of runs in the same hexagon.

There is always a significant correlation between the SR and each of the measures with p-value < 0.001, but the difference lies in the strength of the correlation (i.e., the correlation coefficient). The strongest correlation is observed between the SR and IC (0.488); a high SR value corresponds to a high IC value. Since a high IC value indicates a large number of peaks in the landscape, this suggests

that rugged branches with few plateaus can be covered easily. This is also shown in the correlation between the SR and PIC (0.476) as a high SR value corresponds to a high PIC value. A large PIC value indicates a high landscape modality. This correlation between SR and PIC indicates that on a multimodal landscape it is easier to find the test input that covers a branch.

The third measure that shows a moderate correlation with SR is the NV (0.41) where a high SR value corresponds to a high NV value. A high NV value means that there are more neighboring areas of neutral individuals in the landscape, and thus few flat areas in the landscape. Based on that, and the correlation of SR and NV, the possibility of covering a branch becomes higher when the number of neighboring areas of neutral individuals in the landscape is high.

A negative correlation can be seen with the two measures that estimate the variety of flat areas in the landscape, ND and DBI. A negative correlation means a high SR value corresponds to a low measure value. In the case of ND, the negative correlation between SR and ND (−0.34) suggests that a large neutrality distance (that is, long sequences of neutral steps in the random walk) makes it difficult to cover a branch. However, this correlation is weaker than the correlation between the SR and each of IC, PIC, and NV measures. The negative correlation between SR and DBI (−0.481) indicates that a high SR value corresponds to a low DBI value. According to the definition of DBI, a low DBI value is an indicator of a high density of peaks and few flat areas in the landscape. The negative correlation between SR and DBI suggests that such branches are easier to cover.

Note that the correlation between the SR and the AC measure (0.04) is weaker than the correlation between SR and the other measures. The reason behind that is that measuring the correlation between the fitness values of the random walk is not always helpful in predicting the problem difficulty [14, 23], i.e., the correlation between the fitness values of the random walk does not always anticipate whether a branch is easy to cover.

***RQ 2**: While neutrality seems harmful for search performance, ruggedness does not seem to decrease search performance.*

3.5 RQ3 — What are the underlying properties of source code that influence the fitness landscape?

Having seen that landscape properties can influence the effectiveness of the search, the question now is what aspects of the code under test influence these landscape properties. In order to distinguish between cases where the search is successful simply because the problem is easy, and cases where the reason is the effectiveness of the search algorithm, Figure 3 plots the success rate of the search (MOSA) for each branch vs. the number of times that branch was covered by the random walk within the 30 repetitions. That is, the value 0 means a branch is never covered and 1 means a branch is covered by all 30 runs of either of the two techniques. Notably, a large share of the branches is either always covered (top right corner) or never covered (bottom left corner). However, there is also a substantial share of the branches on which the search is effective but the random walk is not (top left corner) – these are cases with a benign fitness landscape. Surprisingly, there are a few cases also in the bottom right corner of the plot, which were covered

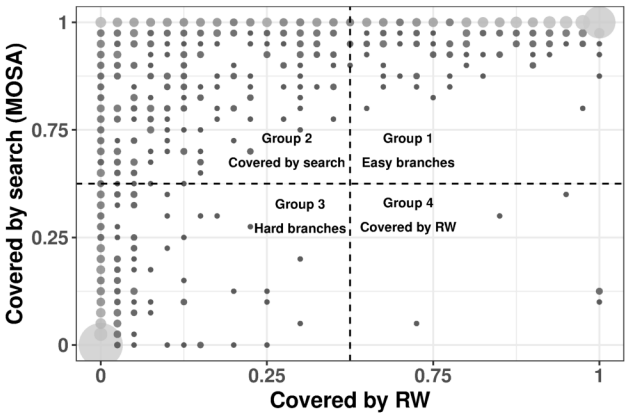

Figure 3: Four groups of the branches based on their coverage by MOSA and random walk (RW) where a large bubble size indicates a high number of branches

Table 2: The average values of the six landscape measures for the branches of the four groups

Group	AC	ND	NV	IC	PIC	DBI
Easy	0.651833	114	9	0.401357	0.092291	0.871883
Search	0.828804	129	5	0.125164	0.057825	0.903901
Hard	0.898022	516	2	0.075532	0.027528	0.960161
RW	0.851833	258	4	0.098439	0.039814	0.928281

during the random walks, but not by the search. Based on these observations, we partition the branches into four groups based on whether they were covered by more than 50% of the runs of the search and random walk, illustrated in Figure 3.

Table 2 shows the mean values of the fitness landscape metrics for the four partitions of Figure 3. The metrics show that branches that are always covered (*easy* group) result in a more rugged landscape than branches that are never covered (*hard* group), where the fitness landscape seems to be dominated by plateaus. Branches covered only by the search (*search* group) appear to result in a substantially more challenging fitness landscape than those always covered (*easy*), yet the landscape metrics confirm there are fewer plateaus than in the most challenging *hard* group. Branches in the odd *RW* group are somewhere in between according to the metrics, and there likely are reasons unrelated to the fitness landscape that cause the search to fail here.

There can be multiple reasons for plateaus in the fitness landscape. A fundamental question is whether the methods containing the branches were executed in the first place – as the fitness function only considers intra-procedural information, the fitness landscape would by definition represent a plateau as long as a method is not called. Figure 4 shows how often the method containing the branch was actually executed during the random walk. Very clearly, methods in the *easy* group (covered by both, search and random walk) are executed far more often than in the other groups. The methods containing branches covered by search are executed substantially less often, but still more often than those that are hard to cover.

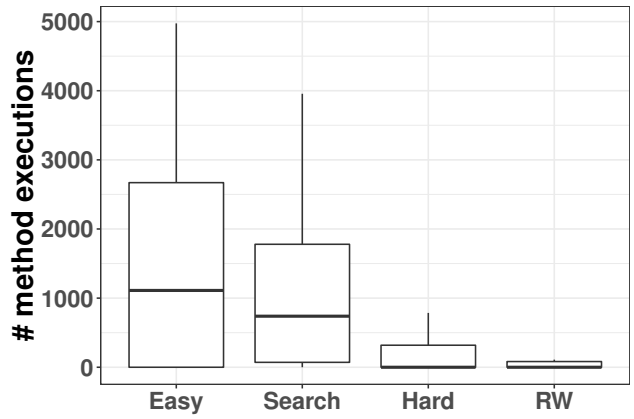

Figure 4: Number of method executions during the random walk for each branch in the four groups

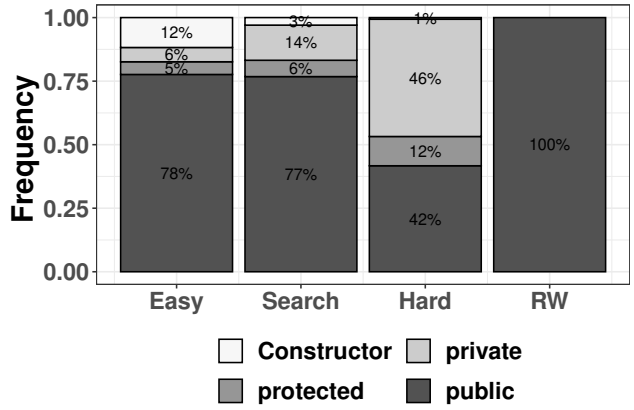

Figure 5: Types of methods containing each branch in the four groups

To understand better why methods are not called, we look at their accessibility, and whether they are methods or constructors (Figure 5): Notably, the *easy* branches contain substantially more constructors and public methods than branches in the *hard* and *search* groups. Interestingly, the few cases in the fourth group are all in public methods. Very notably, private methods are predominantly in the *hard* group, and thus not covered at all. Consequently, accessibility is a primary influential factor for the fitness landscape. This also suggests that a refined fitness function that considers inter-procedural distance information could transform the fitness landscape into a more benign one and thus improve the performance of search-based algorithms.

In those cases where methods are actually called, the branch distances could in principle provide a more nuanced fitness landscape. Since plateaus nevertheless dominate, there are two possible conjectures: Either the executions never even reach relevant branches that could provide a gradient but instead cause exceptions to be thrown by invalid complex parameter objects [27], or the source code is dominated by branches comparing references or boolean flags [11] which, by definition, do not provide gradients.

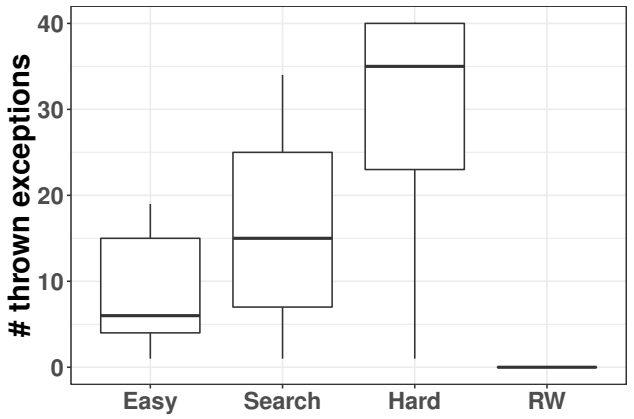

Figure 6: Number of exceptions thrown by methods containing each branch in the four groups

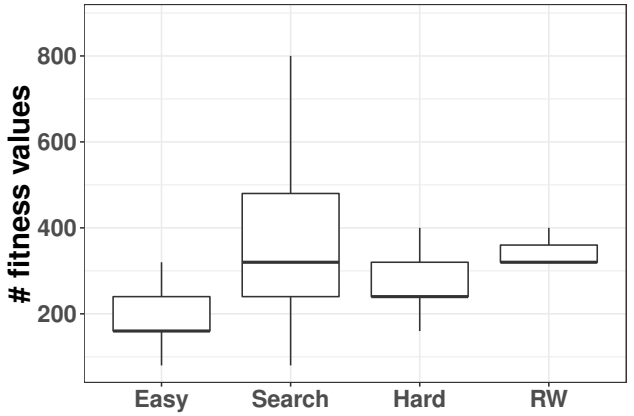

Figure 7: Number of discrete fitness values obtained by the random walk for each branch in the four groups

Figure 6 shows the number of exceptions thrown by the methods containing the branches during the random walk. As expected, the *hard* branches are in methods that are much more likely to result in exceptions (42% of methods calls), while the *easy* branches hardly result in exceptions (8% of methods calls). Branches in group *search* lie in between these two groups (28% of methods calls), and no exceptions at all were observed for the few methods only called by the random walks. Thus, exceptional behavior clearly is an important factor. A possible cause for such exceptions are dependencies on complex objects that are notoriously difficult to configure into valid configurations [27]. Methods may often have implicit preconditions on particular configurations of such valid complex objects, and the fitness function usually provides no guidance to reaching this. Fitness is typically measured only directly on the CUT and not dependency classes; a possible way to improve the fitness landscape would thus be to also consider the code underlying the dependencies, such that there is guidance towards producing valid object configurations. Alternative strategies could include improving the search operators to increase chances of producing valid object configurations, or seeding [26] valid object configurations [13, 29].

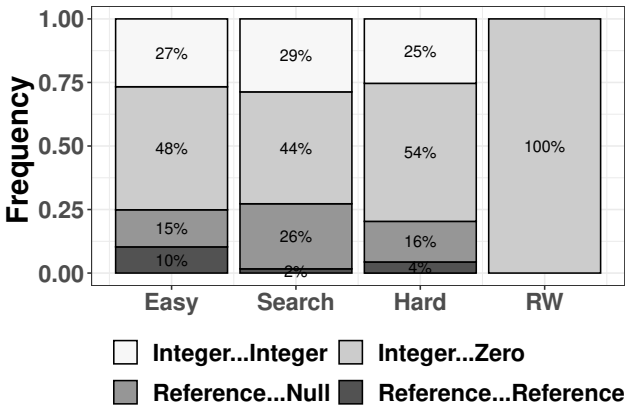

Figure 8: Classifications of the branch types in the four groups

To investigate the influence of the branch types, we first look at the number of discrete fitness values observed (Figure 7). Intuitively, any gradients along the execution to a target branch would lead to many small variations in the fitness values. This, in fact, is observed with the branches of the *search* group, which explains why the search performs well on these branches, but the random walk does not. Interestingly, however, the number of discrete fitness values is also relatively high for branches in the group that is only covered by the random walk. A possible conjecture is that these are branches requiring specific object configurations that are very difficult to produce, and only happen by chance. Since the search tries to minimize test cases as a secondary criterion while the random walk is likely to invoke many more methods on individual objects, the chances of accidentally producing a valid object configuration then simply is higher for the random walk. It is interesting to see that branches in the *easy* group result in very few distinct fitness values; it is likely that they are not embedded within complex conditional code constructs, and depend on well explored parameters. The ruggedness suggested by the fitness landscape analysis in these cases thus likely is not the result of gradients, but of frequently flipping if-conditions, such as easy reference or null comparisons.

To investigate this hypothesis, Figure 8 shows the types of if-conditions, based on their underlying Java bytecode instructions, using the classification by Shamshiri et al [28]: The most common branch type among all four groups is the "Integer-Zero" category, which is produced by the Java compiler mainly for boolean predicates such as `if(x)`, where x is a boolean variable. It is well known that such boolean predicates result in plateaus in the fitness landscape [17]. The *search* group contains slightly more "Integer-Integer" branches, which is the only category of branches that can possibly result in gradients. As expected, the *easy* group contains the most "Reference-Null" and "Reference-Reference" comparisons, thus contributing to their low difficulty and low number of discrete fitness values. The branches covered only by the random walk consist of only "Integer-Null" (i.e., boolean) branches, supporting the conjecture that these are if-conditions querying properties of complex objects that are difficult to produce. Consequently, many of the difficult aspects of the fitness landscape could thus potentially be overcome using testability transformations [11] to remove the boolean flags.

> ***RQ 3**: Plateaus in the fitness landscape are caused by lack of inter-procedural guidance, the difficulty of satisfying preconditions on complex objects, and the prevalence of boolean flags.*

4 RELATED WORK

Aleti et al. [1] previously investigated the fitness landscape in whole test suite generation [7]. In this study, the properties of the fitness landscape were analyzed using information acquired during the evolution, such as the sequence of fitness values of the best individuals and the number of fitness improvements, and then correlated with the branch and method coverage of the GA. The study results suggest that the search space has many plateaus, and the use of the crossover is useless when the landscape is dominated by plateaus. Although this confirms our findings regarding plateaus, we considered a more fine-grained objective function on a branch level, rather than aggregating all the branches into a single objective function. Moreover, our study investigates the factors that cause the fitness landscape properties such as the underlying properties of the source code.

Vogel et al. [31] studied the fitness landscape of test suite generation for mobile applications using multi-objective evolutionary search algorithms. Their fitness landscape analysis focuses on the global topology of the landscape, i.e., how solutions and the fitness are distributed, and not on local structure, i.e., ruggedness and smoothness. The analysis is based on 11 metrics that characterize the Pareto-optimal solutions, population, and connectedness of Pareto-optimal solutions. These metrics are applied after every generation of the algorithm, revealing that the search stagnates because of the lack of diversity. This type of analysis is orthogonal to the landscape analysis we applied in this paper, and could be replicated for unit test generation as well.

5 CONCLUSIONS

Understanding the performance of evolutionary algorithms in generating unit tests requires understanding the underlying structure of the fitness landscape. To this purpose, we studied the fitness landscape in terms of its ruggedness and neutrality. Our study showed that the fitness landscape is highly dominated by neutral areas, i.e., plateaus. Branches that have a large degree of neutrality in their landscape seem to be harder to cover, whereas branches that have a small degree of neutrality in their landscape seem to be easy to cover. Indeed, for this particular search problem, ruggedness does not seem to be detrimental to the search as it indicates the existence of gradients that make a branch easy to cover by GA, and possibly harder to cover by a random walk. The main causes for the often neutral fitness landscapes we identified in our analysis are (1) accessibility of the methods that contain the branches (i.e., private methods are difficult to cover), (2) the difficulty of satisfying the preconditions of methods (i.e., calling them without causing exceptions), but also (3) the classic flag problem (i.e., boolean comparisons offering no guidance) in search-based software testing. These insights offer a potential avenue to improving the fitness landscape, for example by adding inter-procedural distance information and testability transformations.

REFERENCES

[1] Aldeida Aleti, Irene Moser, and Lars Grunske. 2017. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering* 24, 3 (2017), 603–621.

[2] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147.

[3] Andrea Arcuri and Gordon Fraser. 2011. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*. Springer, 33–47.

[4] Lionel Barnett. 1998. Ruggedness and neutrality-the NKp family of fitness landscapes. In *Artificial Life VI: Proceedings of the sixth international conference on Artificial life*. 18–27.

[5] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[6] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.

[7] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[8] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2, Article 8 (Dec. 2014), 42 pages.

[9] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[10] Gregory Gay. 2016. Challenges in using search-based test generation to identify real faults in mockito. In *International Symposium on Search Based Software Engineering*. Springer, 231–237.

[11] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

[12] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering* 36, 2 (2009), 226–247.

[13] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. 2010. OCAT: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*. 159–170.

[14] Kenneth E Kinnear. 1994. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE, 142–147.

[15] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[16] Katherine M Malan and Andries P Engelbrecht. 2013. A survey of techniques for characterising fitness landscapes and some possible ways forward. *Information Sciences* 241 (2013), 148–163.

[17] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

[18] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[19] Galina Merkuryeva and Vitalijs Bolshakovs. 2011. Benchmark fitness landscape analysis. *International Journal of Simulation Systems, Science and Technology* 12, 2 (2011), 38–45.

[20] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[21] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[22] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256.

[23] Erik Pitzer and Michael Affenzeller. 2012. A comprehensive survey on fitness landscape analysis. In *Recent advances in intelligent engineering systems*. Springer, 161–191.

[24] Christian M Reidys and Peter F Stadler. 2001. Neutrality in fitness landscapes. *Appl. Math. Comput.* 117, 2-3 (2001), 321–350.

[25] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.

[26] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.

[27] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[28] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. 2015. Random or genetic algorithm search for object-oriented test suite generation?. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1367–1374.

[29] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 193–202.

[30] Vesselin K Vassilev, Terence C Fogarty, and Julian F Miller. 2000. Information characteristics and the structure of landscapes. *Evolutionary computation* 8, 1 (2000), 31–60.

[31] Thomas Vogel, Chinh Tran, and Lars Grunske. 2019. Does Diversity Improve the Test Suite Generation for Mobile Applications?. In *International Symposium on Search Based Software Engineering*. Springer, 58–74.

[32] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and software technology* 14 (2001), 841–854.

[33] Sewall Wright. 1932. The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proceedings of the Sixth Annual Congress of Genetics*, Vol. 1. 356–366.