Footprints of Fitness Functions in Search-Based Software Testing

Carlos Oliveira
Faculty of Information Technology, Monash University, Melbourne, Australia
carlos.guimaraes@monash.edu

Aldeida Aleti
Faculty of Information Technology, Monash University, Melbourne, Australia
aldeida.aleti@monash.edu

Yuan-Fang Li
Faculty of Information Technology, Monash University, Melbourne, Australia
yuanfang.li@monash.edu

Mohamed Abdelrazek
School of Information Technology, Deakin University, Melbourne, Australia
mohamed.abdelrazek@deakin.edu.au

ABSTRACT

Testing is technically and economically crucial for ensuring software quality. One of the most challenging testing tasks is to create test suites that will reveal potential defects in software. However, as the size and complexity of software systems increase, the task becomes more labour-intensive and manual test data generation becomes infeasible. To address this issue, researchers have proposed different approaches to automate the process of generating test data using search techniques; an area that is known as Search-Based Software Testing (SBST).

SBST methods require a fitness function to guide the search to promising areas of the solution space. Over the years, a plethora of fitness functions have been proposed. Some methods use control information, others focus on goals. Deciding on what fitness function to use is not easy, as it depends on the software system under test. This work investigates the impact of software features on the effectiveness of different fitness functions. We propose the Mapping the Effectiveness of Test Automation (META) Framework which analyses the footprint of different fitness functions and creates a decision tree that enables the selection of the appropriate function based on software features.

CCS CONCEPTS

•Computing methodologies → Search methodologies;

KEYWORDS

Search Based Software Engineering, Genetic Algorithms

ACM Reference format:
Carlos Oliveira, Aldeida Aleti, Yuan-Fang Li, and Mohamed Abdelrazek. 2019. Footprints of Fitness Functions in Search-Based Software Testing. In *Proceedings of Genetic and Evolutionary Computation Conference, Prague, Czech Republic, July 13–17, 2019 (GECCO '19),* 9 pages.
DOI: 10.1145/3321707.3321880

GECCO '19, Prague, Czech Republic

DOI: 10.1145/3321707.3321880

1 INTRODUCTION

Testing is a widely used software validation technique. Among tasks in this process, the most expensive one is the generation of test data that will fulfill a testing criterion [16], accounting for approximately 40% of the total software development budget [49]. Automated test data generation can significantly reduce the cost of testing, thus decrease the overall cost of the entire software development process.

A variety of automated test data generation techniques have been developed in the past few decades. Random test data generators [8, 39, 42] are some of the earliest techniques, which automatically create random inputs until an acceptable one is found. Since test data is devised at random with no knowledge of the software structure or information on the test requirements being incorporated into the generation process, random test data generators may often fail [31]. The focus of this work is on search-based software testing (SBST) technique, which use a fitness function to guide the search for high quality test cases. SBST methods have been successfully applied to many testing problems, including functional testing [20, 23], non-functional testing [34], temporal testing [46], and mutation testing [19].

SBST has drawn significant interest from both the research community and industrial organisations, such as Microsoft, Nokia, Ericsson, Motorola, and IBM [30]. This interest was motivated by the advantages offered by search methods. Firstly, these are generic methods which are ready for adaptation to any testing problem for which a test criterion can be measured. Secondly, search algorithms are robust, capable of coping with noise, partial data and inaccurate fitness [18]. Finally, the search space of test data generation can be very large, hence exact algorithms are impractical.

The application of SBST requires the design of an appropriate fitness function, which measures the quality of generated solutions and guides the search process to promising areas of the search space. To this end, researchers have proposed different definitions of fitness functions, which use different measures, such as structural coverage [36], approach level [31], distance calculation [12, 37], or the combination of more than one measure [41, 45]. Due to the limited time practitioners have in producing test suites for software systems, it becomes important that the chosen fitness function is as effective as possible, and helps find the best solution that can possibly be found in the given amount of time. The question then arises: 'What makes a fitness function effective?'. In this paper we address two key research challenges (RC):

RC1 Determining the most significant software features that have an impact on the effectiveness of fitness functions.

RC2 Determining whether software features can be used for selecting the most suitable fitness function.

To this end, we introduce the Mapping the Effectiveness of Test Automation (META) Framework. The META framework visualizes the performance of fitness functions when applied to software systems with different features. We employ both real and artificial datasets. The artificial dataset is created using a genetic programming approach which evolves classes with different features that make them difficult to cover when using some of the fitness functions. This helps highlight the strengths and weaknesses of different approaches. The META framework creates a footprint of the different fitness functions which is used to explain when and why a fitness function is more effective. Finally, a decision tree is created that can be used by practitioners to select a suitable fitness function based on features of the software system at hand.

2 SEARCH BASED SOFTWARE TESTING

The problem of generating test suites can be defined as a Search Based Software Testing (SBST) problem by: stating the fitness function, such as branch and method coverage; deciding on the solution representation, e.g., binary or string representation; and designing the search operators, such as mutation and crossover [1]. A candidate solution is a test case consisting of a sequence of input values, passed to the program upon execution to observe its behavior. A set of test cases will form a search space. A test adequacy criterion for structure testing is a testing aim that can be numerically measured and assessed, e.g. covered branches or statements. The test criterion is coded as a fitness function, which is used to evaluate the performance of candidate test inputs. In order to assess the fitness of candidate solutions, the program is executed for the inputs generated. The fitness function plays a vital role in the performance of search techniques, as it shapes the fitness landscape. A well-defined fitness function increases the likelihood of finding high quality solutions and reaching high overall coverage, which results in fewer system resources [7, 43].

2.1 Goal-oriented Approaches

Korel [21] developed the goal-oriented approach to alleviate the problem of a path's infeasibility. The idea is to concentrate purely on branches that influence the execution of the goal node, ignoring branches with no influence. The search process determines whether the program's execution should continue through the current branch, or via an alternative branch. Search algorithms are used to automatically find new inputs that will change flow execution. As this approach is based solely on the flow graph of the program, this makes some nodes difficult to reach for some programs [11], because the execution of a certain goal node could require prior execution of other nodes in the program.

2.2 Chaining Approaches

Chaining approaches [11, 22] extends the goal-oriented approach [21] by using program dependency concepts, combined with a program flow graph. The aim is to find solutions to branch predicates by identifying a chain of nodes that affect the execution of the target node. The main contribution of the chaining approach is the use of data dependencies, which improves the efficiency of the search process [15]. Although the chaining approach can be effective for a larger class of programs, the use of the 'find-any-path' concept could present some drawbacks. Firstly, it is hard to predict the coverage to be provided because different paths exercise different branches, resulting in different levels of coverage [10]. Secondly, the search time will significantly increase if there is a high number of paths which need to be considered when processing a chain.

2.3 Coverage-oriented Approaches

Various forms of coverage measures are used in coverage-oriented approaches: (i) statement coverage – estimates the percentage of program statements covered during testing, (ii) branch coverage – measures the extent to which branch statements in the code are covered during the test, and (iii) path coverage – measures the number of feasible paths through the graph produced during the test. Watkins [44] concentrates on full path coverage. Test data that follow previously uncovered paths are assigned higher fitness values than those that pass via paths which have already been covered. The penalisation of executed paths, however, does not exploit the information in the branch predicates [41].

2.4 Distance-oriented Approaches

Branch distance-oriented approaches exploit information from branch predicates, which evaluate how far a predicate is from obtaining its opposite value [25]. The work of Xanthakis et al. [48] was the first to apply GAs in the generation of structural test data. This method follows similar lines to earlier work by Miller and Spooner [25] and it therefore suffers from related problems, such as the limited ability to detect path infeasibility. A tester chooses a path, from which the branch predicates are extracted. A GA is employed to find test data which satisfy all branch predicates in the path. The fitness function sums up all branch distances.

Tracey et al. [40] employ simulated annealing to generate structural test data. The fitness function is the branch distance, which indicates how close the current execution is to adopting the desired branch according to the decision made. If the search stagnates, the approach attempts to generate test data for the next target node. Unlike Korel's approach [20], the newly generated test data do not need to conform to an already successful sub-path. However, this leads to the search losing information about its progress [24], since a solution that deviates from the desired path at an early stage of the search is assigned similar fitness values to those which deviate at a later stage.

The main criticism of branch distance-oriented techniques is that control information about the target node is not included in the fitness function. This may cause the search to get stuck in local optima, thereby making it difficult to obtain full coverage [24, 45]. The control-oriented approaches discussed in the next section will address this problem.

2.5 Control-Oriented Approaches

Control-oriented approaches use a control dependency graph to determine predicate paths for the intended node. Pargas et al. [31] apply a GA for statement and branch coverage, guided by the

control dependencies in the program. For a goal node, a sequence of control-dependent nodes is specified, which should be exercised for the execution of the goal node. The fitness function is equivalent to the number of successful control-dependent node executions.

It is worth noting that using only control structures in fitness functions will form plateaux on the fitness landscape [24]. As there is no distance information that can be exploited, this will result in insufficient guidance towards unexplored structures. If the solutions fail to fulfil any of the branch predicates, no branch distance information will be given on how to ascend the fitness landscape during the search process.

3 FITNESS LANDSCAPES

The different approaches to defining a fitness function presented in the previous sections use different information from the software under test. Their effectiveness, as a results, would depend on whether that information is critical in the software under test. Naturally, a goal oriented approach, which focuses on branches would be more effective in generating test cases for software systems with higher number of branches. The way the fitness function is defined is crucial to how effective it is in guiding the search method to high-quality test cases. This is due to the fact that it will shape the fitness landscape of the problem being solved.

The concept of fitness landscape refers to the topological description of the search space of a problem [1, 4]. Formally, a fitness landscape is represented by a triplet $\{S, N, F\}$ where S is the set of potential solutions for the given problem, also known as the *search space*, N is neighbourhood relation, defined as a subset of $S \times S$. Two solutions are neighbours (i.e. $\langle s, s' \rangle \in N$) if it is possible to reach one solution by applying a search operator to the other. finally $F : S \rightarrow \mathfrak{R}$ is the fitness function.

Different fitness functions give rise to different landscapes, which has an impact on problem difficulty [17, 26, 27]. Finding the best suited fitness function for a problem can also be seen as the problem of finding a function able to create a landscape with the most helpful gradients to exploit and guide the search algorithms [2, 3, 5, 6]. We expect that in the search-based software testing problem, the definition of the fitness function impacts the shape of the fitness landscape, and as a result, the effectiveness of the search procedure.

4 META FRAMEWORK

The proposed *Mapping the Effectiveness of Test Automation (META)* Framework is presented in Figure 1. META Framework is composed of three main components:

i) *The CUT Space*, which includes the classes under test (CUTs) and the structural-based complexity features extracted from the CUTs.
ii) *The Performance Space* which contains the selected fitness functions and the metrics used to assess their performance (generation time, branch coverage, mutation coverage, etc.);
iii) *The Fitness Function Effectiveness space* which identifies CUT features that have an impact on the effectiveness of fitness functions, and maps the strengths and weaknesses of the fitness functions as footprints in a 2-D space.

The META framework is inspired from the Algorithm Selection Problem [29, 38] and the *No Free Lunch theorem*, which informs us that there does not exist a single algorithm that can be expected to outperform all other algorithms on all problem instances [47]. Hence, if method A is superior over method B in solving a particular set of problems, then one may claim that there exists other untested problems where method B outperforms method A. Empirical studies in the area of SBST should focus on identifying conditions under which an algorithm is expected to succeed or fail instead of claiming superiority of a method over another.

We have extended and adapted these ideas to assess the strengths and weaknesses of fitness functions used in SBST, and built the META tool, which can be used to predict which fitness function from a portfolio of different functions is likely to perform best based on measurable features of a collection of CUTs.

4.1 CUT Space

The CUT space contains the set of classes under test and the features used to characterise these classes.

4.1.1 Classes Under Test (CUTs). One important step of our framework is the dataset that is used to train the META model. We examined the diversity of common datasets used in SBST (e.g., SF110 [13]). We observed that these benchmark instances are not diverse enough, as the three fitness functions had similar performance, suggesting one of the following: (i) the fitness functions are all essentially the same, (ii) the instances are not revealing the unique strengths and weaknesses of each fitness function as much as is desired, or (iii) the features are not discriminant enough. Therefore, we propose a method to generate new CUTs, in order to enrich the repository's diversity.

While there is no doubt that these problem repositories have had a tremendous impact on SBST studies, and have improved research practice by ensuring comparability of performance evaluations, there is also concern that these repositories may not be a representative sample of the larger population of software testing problems. It is important to challenge whether existing datasets are enabling us to evaluate fitness function performance in an unbiased manner, and therefore we seek new tools and methodologies that enable us to generate new problem instances that drive improved understanding of the strengths and weaknesses of different approaches. The development of such methodologies to support objective assessment of different fitness functions is at the core of the META framework. This is achieved by generating artificial CUTs.

The generated CUTs must be diverse and large enough to uniformly cover a wide degree of problem difficulty, that is for all fitness functions there must exist both easy and hard instances, and the transition from easy to hard should be densely covered [28]. The most obvious way to artificially generate CUTs is to select and sample an arbitrary probability distribution. However, this approach lacks control, as there is no guarantee that the resulting dataset will have specific features. Hence, we employ a different method, initially introduced for machine learning problems [28], where datasets are evolved using a genetic algorithm to lie at target locations in the instance space.

In this work, we use a Genetic Programming (GP) algorithm to evolve branch predicates that are easy and hard for different

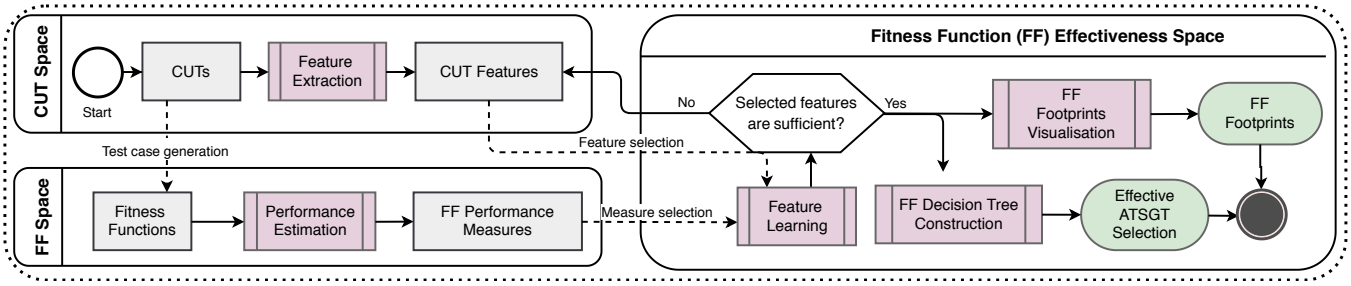


Figure 1: The main components of the Mapping the Effectiveness of Test Automation (META) Framework

fitness functions. The GP uses a variable length (n) solution representation $[(vt_0, ct_0, cv_0), (vt_1, ct_1, cv_1), ..., (vt_n, ct_n, cv_n)]$, where each gene (vt_i, ct_i, cv_i) has three components: variable type vt, comparator type ct, and variable value vv.

The GP algorithm was set to only generate classes that are similar to real world CUTs (unrealistic CUTs were discarded). The CUTs have one method and one nested set of conditional statements up to 4 levels and 14 branches. The possible values are: **1.** variable types include double, long, integer and boolean; **2.** comparator types include $=, \neq, >, <, \leq, \geq$; 3. variable value ranges include Boolean (0, 1), Double $[-1.7 * 10^{308}, +1.7 * 10^{308}]$, Integer $[-2^{31}, +2^{31}]$, and Long $[-2^{63}, +2^{63}]$.

The GP uses mutation and crossover operators to evolve classes of various characteristics and levels of difficulty for our fitness functions. The mutation operator modifies the the variable type, the comparator type or the comparator value. For example, given a parent solution $s = [(vt_0, ct_0, vv_0), (vt_1, ct_1, vv_1), ..., (vt_n, ct_n, vv_n)]$, the mutation operator generates a new candidate solution $s' = [(vt_0, ct_0, vv_0), (vt'_1, ct_1, vv_1), ..., (vt_n, ct_n, vv_n)]$ by mutating the variable type vt_1 into vt'_1.

The crossover consists in exchanging predicates of two parent solutions to generate two new solution. For example, given two parent solutions $s^1 = [(vt^1_0, ct^1_0, vv^1_0), (vt^1_1, ct^1_1, vv^1_1), ..., (vt^1_n, ct^1_n, vv^1_n)]$ and $s^2 = [(vt^2_0, ct^2_0, vv^2_0), (vt^2_1, ct^2_1, vv^2_1), ..., (vt^2_n, ct^2_n, vv^2_n)]$, the crossover operator generates the following two new candidate solutions $s'^1 = [(vt^2_0, ct^2_0, vv^2_0), (vt^1_1, ct^1_1, vv^1_1), ..., (vt^1_n, ct^1_n, vv^1_n)]$ and $s'^2 = [(vt^1_0, ct^1_0, vv^1_0), (vt^2_1, ct^2_1, vv^2_1), ..., (vt^2_n, ct^2_n, vv^2_n)]$.

The main steps of the method are presented in Algorithm 1.

4.1.2 CUT Features. Useful features of CUTs are measurable properties that (i) can be computed in polynomial time and (ii) are expected to expose what makes a SBST hard for a given objective function. At the same time, features must correlate to algorithm performance, measure diverse aspects of the CUTs, and be uncorrelated with one another. The feature set should be small in size, yet it should comprehensively measure aspects of the CUTs that either challenge objective functions or make their task easy. For each CUT, we measure features of the predicates, such as number of variables in a predicate, number of inequalities, and type of variables. The full list of features used in this work includes the number of: **1.** comparators in a class: equalities, inequalities and ranges (higher and less than); **2.** equalities; **3.** inequalities; **4.** ranges (e.g., $1 < x < 3$); **5.** variables; **6.** boolean variables; **7.** integer variables; **8.** long variables; **9.** double variables; **10.** equalities with boolean variables; **11.** inequalities with boolean variables; **12.** equalities with integer variables; **13.** inequalities with integer variables; **14.** ranges with integer variables; **15.** equalities with long variables; **16.** inequalities with long variables; **17.** ranges with long variables; **18.** equalities with double variables; **19.** inequalities with double variables; **20.** ranges with double variables.

We postulate that it is characteristics of branches in a CUT that make it harder or easier for a search-based software testing method to cover. The META framework will identify the specific characteristics that have an impact.

Algorithm 1 Generating CUTs with Genetic Programming.

```
1:  procedure EvolveHardCUTs(dataSet, OF1, OF2)
2:      Input: n, m_r, c_r ▷ n is population size, m_r is mutation rate, c_r is
    crossover rate, FF1 and FF2 are the fitness functions.
3:      P ← RandomSolutions(n)                          ▷ P is the population
4:      for i ← 1 to n do
5:      Q ← ∅                                  ▷ Q is the Auxiliary Population
6:      while !TerminationCodition do
7:          for i ← 1 to n do
8:              p_1, p_2 ← RouletteWheelSelection(P)
9:              o_1, o_2 ← UniformCrossover(p_1, p_2, c_r)
10:             o'_1 ← Mutation(o_1, m_r)
11:             o'_2 ← Mutation(o_2, m_r)
12:             f(o'_1) ← EvaluateFitness(o'_1,FF1, FF2)
13:             f(o'_2) ← EvaluateFitness(o'_2,FF1, FF2)
14:             Insert(o'_1, o'_2, Q)
15:         R ← P ∪ Q
16:         RankSolutions(R)
17:         P ← SelectBestSolutions(R)
18:     Return(P)
19: procedure EvaluateFitness(c, FF1, FF2)
20:     bc_1 ← execute(c, FF1)
21:     bc_2 ← execute(c, FF2)          ▷ bc_1, bc_2 are the branch coverage
    performance
22:     Return(bc_1/bc_2)
```

4.2 Fitness Function Space

In this space, we define the fitness functions that we evaluate, and the performance measures used to assess the effectiveness of the different fitness functions.

4.2.1 Coverage-oriented. The coverage oriented function we employ in our investigation was originally proposed for branch coverage [36]. This function is primarily concerned with ensuring

that the highest possible level of coverage is achieved. Given a test suite T and a set of branches B of the program being tested, the coverage level $f_1(b,T)$ for each branch $b \in B$ on test suite T is:

$$f_1(b,T) = \begin{cases} 0 & \text{if both branches are covered,} \\ 0.5 & \text{if the predicate is executed once,} \\ 1 & \text{otherwise.} \end{cases} \quad (1)$$

The overall fitness function of a test suite is:

$$\text{CLF} = |M| - |M_T| + \sum_{b \in B} f_1(b,T), \quad (2)$$

where M is the set of methods in the object and M_T is the set of methods executed during the test. The difference $|M| - |M_T|$ is used to reward coverage of methods in the test objects which have no branch statements.

4.2.2 Distance-oriented. This fitness function [12] uses the branch distance measurement which reflects how close a branch's predicate is to being reached. Given a set of branches B, the minimal branch distance for each branch $b \in B$ in test suite T is defined as:

$$f_2(b,T) = \begin{cases} 0 & \text{if the branch is covered,} \\ d(b,T) & \text{if the predicate is executed at least twice,} \\ 1 & \text{otherwise,} \end{cases} \quad (3)$$

where $d(b,T)$ is 0 if at least one of the branch's values (true or false) has been covered, and > 0 otherwise. The fitness function is then calculated as:

$$\text{BDF} = |M| - |M_T| + \sum_{b \in B} f_2(b,T), \quad (4)$$

where M is the set of methods and M_T is the set of methods executed by test suite T.

4.2.3 Control-oriented. The control oriented function [31] uses a control dependency graph to compute an individual's fitness value. The fitness value is equivalent to the number of successful control dependent node executions towards the intended branch.

Let dn be the number of control dependent nodes for the current target branch, and en be the number of successfully executed control-dependent nodes; the fitness function $f_3(b,T)$ for each branch $b \in B$ in test suite T is defined as $f_3(b,T) = \text{norm}(dn - en)$, where norm is a normalisation function in the range [0, 1]. The fitness of a test suite T is calculated as:

$$\text{CFF} = |M| - |M_T| + \sum_{b \in B} f_3(b,T), \quad (5)$$

where M is the set of methods in T and M_T is the set of methods executed during the test.

4.3 Fitness Function Effectiveness Space

4.3.1 Feature Learning. The feature learning step of the META framework identifies the most significant features of CUTs that impact the effectiveness of the fitness functions. The input to this step are the collection of CUT features and the performance of the fitness functions, measured as the percentage of the branches covered. The output is the set of features that best describe why a certain fitness function is effective. We use a genetic algorithm to search for groups of features containing between 3 and 10 features that best explain the weaknesses and strengths of the fitness functions. The main steps of the approach are presented in Algorithm 2.

Algorithm 2 Feature Selection with a Genetic Algorithm

```
1:  procedure FeatureLearning(features, OFP)
2:      Input: n, m_r, c_r ▷ n is population size, m_r is mutation rate, c_r is
    crossover rate, OFP is OF performance.
3:      P ← RandomSolutions(n)                          ▷ P is the population
4:      for i ← 1 to n do
5:      Q ← ∅                                           ▷ Q is the Auxiliary Population
6:      while !TerminationCodition do
7:          for i ← 1 to n do
8:              p_1, p_2 ← RouletteWheelSelection(P,F)
9:              o_1, o_2 ← UniformCrossover(p_1, p_2, c_r)
10:             o'_1 ← Mutation(o_1, m_r)
11:             o'_2 ← Mutation(o_2, m_r)
12:             f(o'_1) ← EvaluateFitness(o'_1,OFP)
13:             f(o'_2) ← EvaluateFitness(o'_2,OFP)
14:             Insert(o'_1, o'_2, Q)
15:         R ← P ∪ Q
16:         RankSolutions(R)
17:         P ← SelectBestSolutions(R)
18:     Return(P)
19: procedure EvaluateFitness(s, OFP)
20:     Input: s                                        ▷ set of features to be evaluated
21:     2D_coordinates ← PCA(s, OFP)
22:     f(s) ← SVM(2D_coordinates, OFP)
23:     Return(f(s))                                    ▷ Return the fitness of the set of features
```

An individual solution contains the set of features that are used to characterise the CUTs. The crossover operator (line 9) takes two sets of features and combines them at different random positions. Next mutation operator is applied to both solutions (lines 10 and 11 by replacing a random feature from the set of features with a new feature. The fitness function described by the second procedure in Algorithm 2 is based on the accuracy of a Support Vector Machine (SVM) applied on the reduced 2D CUT sub-space. Principal Component Analysis (PCA) is used in order to reduce the CUT sub-space from n to 2 dimensions.

4.3.2 Footprints Visualisation. Once the most significant features are identified, they are used to visualise the footprints of the fitness functions, as shown in Figure 2, with each CUT represented as a point in the space.

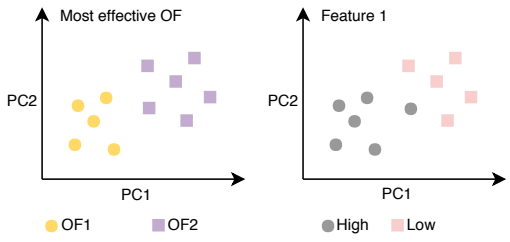


Figure 2: Strengths and weaknesses of fitness functions.

We apply PCA as a dimensionality reduction technique on the optimal subset of features. The aim is to plot the performance of the fitness functions (OF1 and OF2) across the CUT space in 2D, which is likely to reveal where the methods are performing well, and where they are sub-optimal. In this example, SVM would produce a highly accurate prediction score and consequently, the set of features used to create this space, would receive a high fitness score. A high fitness means that the selected features were able to identify accurately the characteristics associated to good and bad performance.

The plot on the left visualises the effectiveness of two objective functions over all CUTs. The circles are the CUTs where OF1 is the most effective, whereas the squares represent the CUTs where the second technique is the most effective.

Next, we plot the same CUTs in a 2-dimensional map based on how they score in terms of the most significant features identified in the feature learning step (plotted on the right in Figure 2). As shown by the legend, the circles indicate that the respective CUTs score highly according to Feature 1 (the most significant feature), while the CUTs with low Feature 1 are represented as squares. Looking at both plots, this example shows that OF1 is effective when Feature 1 is high, whereas OF2 works best when Feature 1 is low.

4.3.3 Decision Tree Construction. Using the most significant CUT features, a decision tree is constructed, which can be used to select the most effective objective function for new software systems based on their features. We use the clustering algorithm DBSCAN [9] to identify the areas in the CUT space where the different objective functions are effective. In the next step, the C4.5 [32] algorithm is used to generate the decision tree with the features identified in the feature learning phase.

5 EXPERIMENTAL SETUP

Experiments were performed on a 10 computers running a Linux operating system, with two cores of 2.5 GHz and eight gigabytes of memory. The objective functions were implemented in EvoSuite [12] and used to guide the search in the test suite generation.

Each trial was repeated 10 times to take the random nature of the search technique into consideration. The time-out was two minutes per class, stated as the best trade-off between time and branch coverage [14]. Three fitness functions that are part of the experiments are CLF (described in Section 4.2.1), BDF (Section 4.2.2), and CFF (Section 4.2.3).

Branch coverage is used as a performance measure. An objective function is superior if its branch coverage is at least 1% higher than the other techniques. There were no cases where objective functions performed equally.

5.1 SBST Tool

This study employs EvoSuite [12] as a search-based testing tool for Java projects. EvoSuite evolves candidate test suites aimed at covering all test goals, while at the same time minimising the total size of the suite (i.e. reducing the number of test cases and their length). As such, when there is a tie between test suites with respect to their fitness values, EvoSuite chooses the test suite which is composed of a cumulatively lower number of statements.

5.2 Optimisation Method

The objective functions were used to guide a state of the art optimisation method, the Whole Suite with Archive (WSA) [35]. WSA starts by generating a set of solutions, which are uniformly, randomly initialized. Formally, let t denote a test case, which consists of a sequence of statements $t = \langle s_1, s_2, ..., s_l \rangle$ of length l. A statement s_i can be a constructor, a field, a primitive, a method, or an assignment. A solution is defined as a test suite T, which is a collection $T = \{t_1, t_2, ..., t_n\}$ of test cases. An optimal solution T^* is a test suite that covers all possible branches and lines of code, i.e., 100% coverage.

Since the number of test cases in a test suite and the number of statements in a test case may vary, the solution representation is of a variable size. The solutions are evolved in iterations until a stopping criterion is achieved, which usually is a predefined number of function evaluations. The Genetic Algorithm used by EvoSuite has four genetic operators that are applied to solutions at every iteration: crossover, mutation, selection, and replacement. Crossover creates two new solutions by combining test cases from two test suites in the population. The mutation operator is applied after the crossover operator, at a test suite level and at a test case level. Test suites are mutated by changing each of the test cases with a probability $1/n$, where n is the number of test cases in the test suite. In addition, new test cases are added to the test suite at random. Mutation of test cases is performed by either adding, changing, or removing statements from a test case with a probability $1/l$, where l is the number of statements in a test case.

A rank-based selection procedure is employed to select the parent solutions that will undergo recombination and mutation procedures. Solutions are ranked based on the fitness function. When there is a tie between solutions, shorter test suites are assigned better ranks. As a result, solutions with better branch coverage and shorter length have a higher chance of projecting their 'genes' to the next generation. Similar to the original studies with EvoSuite, an elitist strategy is used as a replacement procedure [12], which selects the best solutions to create the next generation.

6 RESULTS

The GP algorithm evolved 202 classes. CFF did not outperform the other objective functions in any of these classes. It is possible that CFF is superior in CUTs with unrealistic features, such as nested if conditions with a tree size larger than 10. However, our CUT generator was set up in a way that unrealistic CUTs were discarded. An example of a CUT is shown in Listing 1.

Listing 1: An example of an evolved CUT

```
public class Evolved_CUT_1 {
public void method1 (Double number00,
Long number10, Integer number20,
Integer number30,Long number40,
Boolean number50,Boolean number60){
if( number00 <= 2.6234427914696525E307D){
 System.out.println("b1");
 if( number10 < -4282759360621669546L){
  System.out.println("c1");
  if( number30 == 1640314761){
```

```
    System.out.println("d1"); }
  else { System.out.println("d2"); }
 } else { System.out.println("c2");
  if( number40 > 550514008264203262L){
   System.out.println("d3");
  } else { System.out.println("d4"); }
 } } else { System.out.println("b2");
 if( number20 > 1937584) {
  System.out.println("c3");
  if( number50 == false) {
   System.out.println("d5");
  }  else { System.out.println("d6"); }
 } else { System.out.println("c4");
  if( number60 == true){
   System.out.println("d7");
  } else { System.out.println("d8"); }
 } } } }
```

The coverage achieved by BDF and CLF for this example are shown in Figures 3 and 4. The nodes represent parts of the code where the execution branches out (if-else statement). The labels next to the arrows are the number of times that a branch has been executed by test cases (out of 30 test cases) generated using the two objective functions. In this example, BDF outperforms CLF, as it has generated a higher number of test cases that can cover most of the branches.

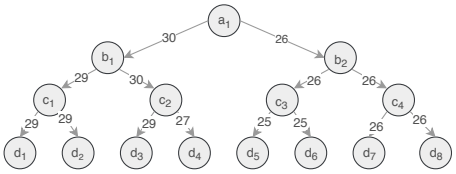


Figure 3: Coverage achieved by BDF.

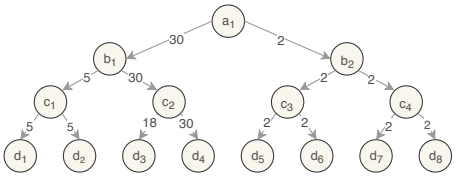


Figure 4: Coverage achieved by CLF.

Applying the feature learning method, the SVM identified the footprints where BDF and CLF perform well with **76% accuracy**. The META framework identified the following optimal features which best capture the difficulty in generating test cases:

- i Integer Variables with ≤ and ≥ comparator
- ii Equalities with Double Variables
- iii Double Variables with ≤ and ≥ comparator
- iv Equalities with Long Variables
- v Inequalities with Long Variables
- vi Long Variables with ≤ and ≥ comparator.

Using these six features the META framework created the CUT Space and identified the footprints of the two objective functions. Using the footprint visualisation method a 2d effectiveness map is created, as shown in Figure 5a. Each point is a CUT, which is colored red if CLF is the most effective objective function, and blue if BDF is the winner. **The first two components used to visualise the CUT space explain 47.5% of the variation in the data.** The values of these two components are as follows:

$$\begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} -.28 & .21 & -.75 & -.25 & .50 & .76 \\ -.77 & .55 & .11 & .59 & .01 & -.13 \end{bmatrix} \begin{bmatrix} \text{i} \\ \text{ii} \\ \text{iii} \\ \text{iv} \\ \text{v} \\ \text{vi} \end{bmatrix} \quad (6)$$

These are the footprints of the two objective functions. The best separation is provided by the second principal component (PC2) axis. Therefore, the features that contribute the most to the PC2 are the most important ones. This gives the answer to the first research challenge:

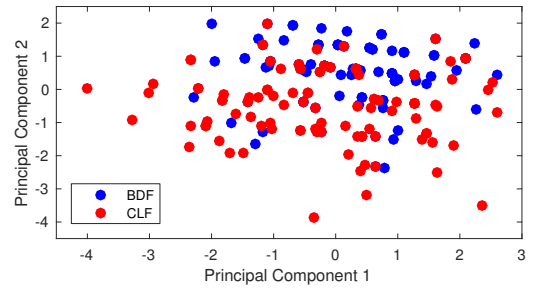
RC1: The most significant features that have an impact on the effectiveness of BDF and CLF are **integer variables with ≤ and ≥ comparator**, **equalities with long variables**, and **equalities with double variables**.

Figures 5b, 5c and 5d show the same principal components, however, the CUTs are now colored according to the most significant features. Figure 5b shows how the CUTs score according to the Integer Variables with <= and >= comparator. When we consider both Figures 5a and 5b side by side, it becomes clear that BDF is effective in generating test suite for classes that have a high number of Integer Variables with <= and >= comparator, whereas CLF is more effective in CUTs that score low according to this feature. In a similar way, Figures 5c and 5d show that BDF is more effective when both the number of equalities with long type and the number of equalities with double type are low, while CLF is more effective when the CUTs score low in these two features.

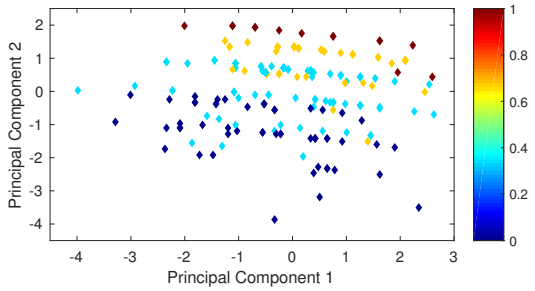
We did not find a superior average performance of CLF over BDF inside the CLF footprint area. In the cases that BDF is superior to CLF, the performance difference is usually higher than 30%. CLF is better in most of cases inside the CLF footprint area, however the performance difference reaches a maximum of 11%. In the majority of cases the difference ranges between 2 and 5%. On the other hand, BDF presents a significant superior performance inside BDF footprint area. In the best case, the average branch coverage reaches a difference of 55%. This means that BDF is on average better than CLF, but there are CUTs where CLF outperforms BDF.

Next, a decision tree is constructed which can be used for objective function selection when solving new SBST problems. The goal is to achieve optimal classification of the Objective Functions with minimal number of decisions. The rules defining the decision tree for Objective Function selection are shown in Figure 6.

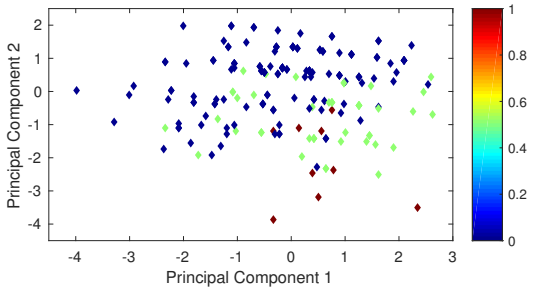
The decision tree can be used to select the appropriate objective function when a new software project requires the generation of test cases. The results were validated for consistency and accuracy using 10-fold cross validation technique. Results from the classifier are presented in Table 1.



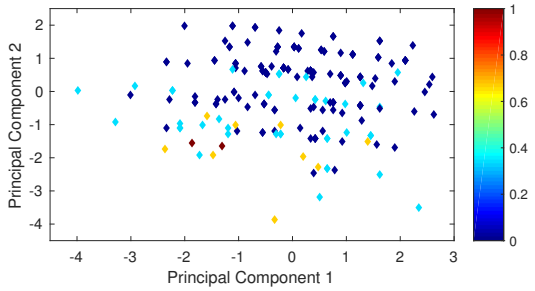
(a) Effectiveness Map



(b) Integer Variables with ≤ & ≥ comparator.



(c) Number of Equalities with Long type.



(d) Number of equalities with Double type.

Figure 5: Visualisation of the footprints of fitness functions. The principal components are defined in Eq. 6.

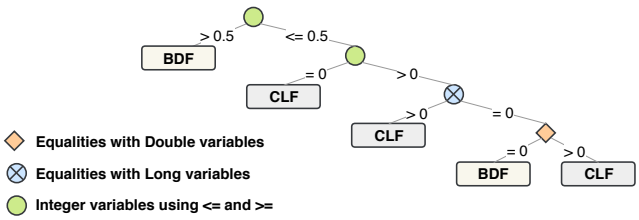


Figure 6: Objective Function decision tree.

Table 1: 10-fold cross validation of the decision tree.

Fitness function	Precision	Recall	F-Score
CLF	76%	71%	74%
BDF	74%	77%	75%
Average	75%	74%	74%

Precision denotes the proportion of predicted positive cases that are correctly real positives [33]. Recall is the proportion of real positive cases that are correctly predicted positive [33]. As there is always a quality compromise between Precision and Recall, being desirable but different features, the F-Measure is used as a harmonic mean to counter this problem. It references the true positives to the arithmetic mean of predicted positives and real positives, being a constructed rate normalized to an idealized value [33]. In summary, the decision tree selects the most effective fitness function with high accuracy, hence the answer to the second research question is:

RC2: The **integer variables with <= and >= comparator**, **equalities with long variables**, and **equalities with double variables** can accurately predict the most suitable fitness function, hence features can be used for fitness function selection.

Analyzing Figure 6, we observe that BDF has issues when dealing with equalities using Long and Double variables. CLF is superior in most of the cases where the branch predicate present equalities with Double or Long. While BDF has superior performance when the number of Integer variables using <= or >= is higher than 0.5.

7 CONCLUSION

We investigated the effectiveness of fitness functions widely used in Search-Based Software Testing. Fitness functions are crucial in guiding the search algorithm and finding high quality test cases. We developed the META framework which identifies features of classes under test (CUTs) that impact the effectiveness of different fitness functions. The framework creates a decision tree based on the most significant features, which can be used for fitness function selection. Beyond the challenge of accurately predicting which fitness function is likely to perform best for a given CUT, based on the relationship between CUT features and fitness function performance, the META framework also explains why.

REFERENCES

[1] Aldeida Aleti and Lars Grunske. 2015. Test data generation with a Kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software* 103 (2015), 343 – 352. DOI: http://dx.doi.org/10.1016/j.jss.2014.11.035 Special Issue.

[2] Aldeida Aleti and Irene Moser. 2011. Predictive Parameter Control. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM,

561–568. DOI: http://dx.doi.org/10.1145/2001576.2001653
[3] Aldeida Aleti and Irene Moser. 2013. Entropy-based Adaptive Range Parameter Control for Evolutionary Algorithms. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. ACM, 1501–1508. DOI: http://dx.doi.org/10.1145/2463372.2463560
[4] Aldeida Aleti, Irene Moser, and Lars Grunske. 2016. Analysing the fitness landscape of search-based software testing problems. *Automated Software Engineering* (2016), 1–19.
[5] Aldeida Aleti, Irene Moser, Indika Meedeniya, and Lars Grunske. 2014. Choosing the appropriate forecasting model for predictive parameter control. *Evolutionary computation* 22, 2 (2014), 319–349.
[6] Aldeida Aleti, Irene Moser, and Sanaz Mostaghim. 2012. Adaptive range parameter control. In *2012 IEEE Congress on Evolutionary Computation*. IEEE, 1–8.
[7] André Baresel, Harmen Sthamer, and Michael Schmidt. 2002. Fitness Function Design To Improve Evolutionary Structural Testing.. In *Genetic and Evolutionary Computation Conference*, Vol. 2. 1329–1336.
[8] David L. Bird and Carlos Urias Munoz. 1983. Automatic generation of random self-checking test cases. *IBM systems journal* 22, 3 (1983), 229–245.
[9] B Borah and DK Bhattacharyya. 2004. An improved sampling-based DBSCAN for large spatial databases. In *Intelligent Sensing and Information Processing, 2004. Proceedings of International Conference on*. IEEE, 92–96.
[10] Jon Edvardsson. 1999. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*. 21–28.
[11] Roger Ferguson and Bogdan Korel. 1995. Software test data generation using the chaining approach. In *International Test Conference*. IEEE, 703–709.
[12] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *Software Engineering, IEEE Transactions on* 39, 2 (2013), 276–291.
[13] 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 (2014), 8.
[14] 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 (2014), 8:1–8:42 pages. DOI: http://dx.doi.org/10.1145/2685612
[15] Matthew J Gallagher and V Lakshmi Narasimhan. 1997. Adtest: A test data generation suite for ada software systems. *Software Engineering, IEEE Transactions on* 23, 8 (1997), 473–484.
[16] Kamran Ghani and John A Clark. 2009. Automatic test data generation for multiple condition and MCDC coverage. In *Software Engineering Advances, International Conference on*. IEEE, 152–157.
[17] Marius Gheorghita, Irene Moser, and Aldeida Aleti. 2013. Characterising fitness landscapes using predictive local search. In *Proceedings of the 15th annual conference companion on genetic and evolutionary computation*. ACM, 67–68.
[18] Mark Harman and Phil McMinn. 2010. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on* 36, 2 (2010), 226–247.
[19] Yue Jia and Mark Harman. 2008. Constructing subtle faults using higher order mutation testing. In *Source Code Analysis and Manipulation, International Working Conference on*. IEEE, 249–258.
[20] Bogdan Korel. 1990. Automated software test data generation. *Software Engineering, IEEE Transactions on* 16, 8 (1990), 870–879.
[21] Bogdan Korel. 1992. Dynamic method for software test data generation. *Software Testing, Verification and Reliability* 2, 4 (1992), 203–213.
[22] Bogdan Korel. 1996. Automated test data generation for programs with procedures. In *ACM SIGSOFT Software Engineering Notes*, Vol. 21. ACM, 209–215.
[23] Nashat Mansour and Miran Salame. 2004. Data generation for path testing. *Software Quality Journal* 12, 2 (2004), 121–136.
[24] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing Verification and Reliability* 14, 2 (2004), 105–156.
[25] Webb Miller and David L. Spooner. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* 2, 3 (1976), 223.
[26] I Moser, M Gheorghita, and A Aleti. 2016. Identifying Features of Fitness Landscapes and Relating Them to Problem Difficulty. *Evolutionary computation* (2016).
[27] Irene Moser, Marius Gheorghita, and Aldeida Aleti. 2016. Investigating the correlation between indicators of predictive diagnostic optimisation and search result quality. *Information Sciences* 372 (2016), 162–180.
[28] Mario A. Muñoz, Laura Villanova, Davaatseren Baatar, and Kate Smith-Miles. 2018. Instance spaces for machine learning classification. *Machine Learning* 107, 1 (2018), 109–147. DOI: http://dx.doi.org/10.1007/s10994-017-5629-5
[29] Carlos Oliveira, Aldeida Aleti, Lars Grunske, and Kate Smith-Miles. 2018. Mapping the Effectiveness of Automated Test Suite Generation Techniques. *IEEE Transactions on Reliability* 99 (2018), 1–15.
[30] Alessandro Orso and Gregg Rothermel. 2014. Software testing: a research travelogue (2000–2014). In *Future of Software Engineering*. ACM, 117–132.
[31] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. 1999. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability* 9, 4 (1999), 263–282.
[32] Kemal Polat and Salih Güneş. 2009. A novel hybrid intelligent method based on C4. 5 decision tree classifier and one-against-all approach for multi-class classification problems. *Expert Systems with Applications* 36, 2 (2009), 1587–1592.
[33] D. M. W. Powers. 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.
[34] Peter Puschner and Roman Nossal. 1998. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symposium*. IEEE, 134–143.
[35] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893. DOI: http://dx.doi.org/10.1007/s10664-015-9424-2
[36] Marc Roper. 1997. Computer aided software testing using genetic algorithms. (1997).
[37] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. 2015. Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?. In *Genetic and Evolutionary Computation Conference*. ACM, 1367–1374.
[38] Kate A. Smith-Miles, Davaatseren Baatar, Brendan Wreford, and Rhyd Lewis. 2014. Towards objective measures of algorithm performance across instance space. 45 (2014), 12–24.
[39] Pascale Thevenod-Fosse and Helene Waeselynck. 1993. STATEMATE applied to statistical software testing. In *ACM SIGSOFT Software Engineering Notes*, Vol. 18. ACM, 99–109.
[40] Nigel Tracey, John Clark, Keith Mander, and John McDermid. 1998. An automated framework for structural test-data generation. In *Automated Software Engineering*. IEEE, 285–288.
[41] Nigel James Tracey. 2000. *A search-based automated test-data generation framework for safety-critical software*. Ph.D. Dissertation. Citeseer.
[42] Jeffrey Voas, Larry Morell, and Keith Miller. 1991. Predicting where faults can hide from testing. *Software* 8, 2 (1991), 41–48.
[43] Alison Watkins and Ellen M Hufnagel. 2006. Evolutionary test data generation: a comparison of fitness functions. *Software: Practice and Experience* 36, 1 (2006), 95–116.
[44] Alison Lachut Watkins. 1995. The automatic generation of test data using genetic algorithms. In *Software Quality Conference*, Vol. 2. 300–309.
[45] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.
[46] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. 1997. Testing real-time systems using genetic algorithms. *Software Quality Journal* 6, 2 (1997), 127–135.
[47] David H Wolpert and William G Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67–82.
[48] S Xanthakis, C Ellis, C Skourlas, A Le Gall, S Katsikas, and K Karapoulios. 1992. Application of genetic algorithms to software testing. In *International Conference on Software Engineering and its Applications*. 625–636.
[49] Man Xiao, Mohamed El-Attar, Marek Reformat, and James Miller. 2007. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering* 12, 2 (2007), 183–239.