An Adaptive Search Budget Allocation Approach for Search-Based Test Case Generation

SIMONE SCALABRINO, University of Molise, Italy
ANTONIO MASTROPAOLO, University of Molise, Italy
GABRIELE BAVOTA, Università della Svizzera Italiana, Switzerland
ROCCO OLIVETO, University of Molise, Italy

Search-based techniques have been successfully used to automate test case generation. Such approaches allocate a fixed search budget to generate test cases aiming at maximizing code coverage. The search budget plays a crucial role; due to the hugeness of the search space, the higher the assigned budget, the higher the expected coverage. Code components have different structural properties that may affect the ability of search-based techniques to achieve a high coverage level. Thus, allocating a fixed search budget for all the components is not recommended and a component-specific search budget should be preferred. However, deciding the budget to assign to a given component is not a trivial task.

In this paper we introduce BOT, an approach to adaptively allocate the search budget to the classes under test. BOT requires information about the branch coverage that will be achieved on each class with a given search budget. Therefore, we also introduce BRANCHOS, an approach that predicts coverage in a budget-aware way. The results of our experiments show that (i) BRANCHOS can approximate the branch coverage in time with a low error, and (ii) BOT can significantly increase the coverage achieved by a test generation tool and the effectiveness of generated tests.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; **Search-based software engineering**.

Additional Key Words and Phrases: Budget Allocation, Test Case Generation, Search-Based Software Engineering

ACM Reference Format:
Simone Scalabrino, Antonio Mastropaolo, Gabriele Bavota, and Rocco Oliveto. 2020. An Adaptive Search Budget Allocation Approach for Search-Based Test Case Generation. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 26 pages. https://doi.org/10.1145/3446199

1 INTRODUCTION

Automatic test case generation tools (TGTs) are designed to derive test cases for a given software project, and can reduce the time allocated for unit testing. Many TGTs have been defined in the literature [10, 13, 18, 19, 21, 27], most of them based on search-based techniques [11, 23, 24]. Randoop [21] and EvoSuite [10] are two well-known examples of such tools, based on random search (Randoop) and genetic algorithms (EvoSuite). These tools take as input a set of components (*e.g.,* classes) to be tested and a search budget, namely the time that can be spent searching for a

Authors' addresses: Simone Scalabrino, University of Molise, Italy, simone.scalabrino@unimol.it; Antonio Mastropaolo, University of Molise, Italy, a.mastropaolo1@studenti.unimol.it; Gabriele Bavota, Università della Svizzera Italiana, Switzerland, gabriele.bavota@usi.ch; Rocco Oliveto, University of Molise, Italy, rocco.oliveto@unimol.it.

1049-331X/2020/1-ART1 $15.00
https://doi.org/10.1145/3446199

(near-)optimal solution that maximizes code coverage. The search budget plays a fundamental role since automatic test case generation is a time-consuming activity due to the hugeness of the search space. In general, the higher the assigned search budget, the higher the *expected* code coverage.

State-of-the-art tools allow to specify a **global** search budget, meaning the overall time that the search-based approach will invest in the coverage of all targets (*e.g.,* all the branches of the classes under test). However, intuitively, some classes are easier to cover than others. For example, a class implementing a Java parser is likely to exhibit a high code complexity, with branches dealing with exceptional conditions that are unlikely to happen but that still need to be managed. The automatic generation of test cases for such a class is clearly challenging, and could require a long execution time before reaching a satisfactory coverage level. On the other hand, a simple Java bean representing a data class and having little application logic is likely to require only a few seconds in order to meet the coverage target. Thus, assuming the possibility to specify a *local* search budget rather than a *global* one (*i.e.,* assign a specific budget to each unit to test), the first class (*i.e.,* the parser) should probably be assigned with a higher search budget as compared to the second one (*i.e.,* the Java bean).

Such an intuition has been exploited by Campos *et al.* [5] when presenting the idea of Continuous Test Generation (CTG), embedding automatic test case generation in the process of continuous integration. CTG exploits historical data of the project under test to allocate a specific budget to each of its classes. For example, if a class C has been modified in a commit, then its search budget should be higher. Also, information about the coverage achieved on C in past runs of test case generation is used to decide the search budget for a new version of C. More in general, abstracting from the CTG to the automatic generation of test cases, this problem can be referred to as *Budget Optimization Problem*: given a set of classes and a global search budget, the aim is to divide the budget to optimize the global coverage. While the approach by Campos *et al.* [5] is a clear step in this direction, it can only be used when historical information is available. Indeed, there are many cases in which projects do not have test suites or they only cover a small portion of the system: Kochhar *et al.* [17] show that the majority of the open-source projects they analyzed have test suites achieving less than 25% coverage.

Ferrer *et al.* [9] proposed a metric—namely Branch Coverage Expectation (BCE)—to assess how difficult it is to generate test cases for a given class or method.

While the authors show that BCE correlates with branch coverage better than existing metrics, its output cannot be directly used for predicting the coverage achieved by a test case generation tool when run on a given component. Indeed, while BCE is a good proxy for the difficulty of testing the code artifact C, it does not take into account the search budget assigned to the test case generator for testing C.

In this paper, we introduce BOT, an approach tackling the Budget Optimization Problem. Given a set of classes to test, BOT uses a search-based algorithm to intelligently divide the global budget among the classes. BOT requires a budget-aware estimation of the coverage achieved on a given class. For this reason, we introduce BRANCHOS, an approach built on top of BCE that predicts the branch coverage that a search-based test case generation technique will achieve on a class *given a specified search budget*. Specifically, we define a set of structural metrics that we use as predictors of branch coverage "*in time*"[1] and we use machine learning to train a regressor. We evaluate both BRANCHOS and BOT on 10,349 Java classes. Our results indicate that (i) BRANCHOS is able to predict branch coverage significantly better than the considered baselines, and (ii) BOT can improve the project-level branch coverage and the effectiveness of test suites created by automatic

[1]We use "in time" to refer to the ability of the approach to predict the coverage reached by an automatic test case generation tool on a unit under test *in a given time budget.*

test case generation techniques both in an ideal scenario (simulating a perfect branch prediction approach) and in a realistic scenario (using BRANCHOS). For the latter one, however, the observed improvement is quite limited, calling for more accurate branch prediction approaches.

BOT can be used on projects on which the total coverage is insufficient, in order to run test case generation on the classes still not covered by test cases. Besides, BRANCHOS can also be used in isolation, to decide the most adequate search budget to assign to a new class for which automated test case generation will be executed.

2 RELATED WORK

In this section we presents an overview of the state of the art on automatic test case generation. Then, we describe in details the existing techniques for branch coverage prediction.

2.1 Search-Based Test Case Generation

Developers run unit tests to check the presence of faults in their code. However, manually writing test cases is a time consuming task. Researches theorized many different approaches for automatic test case generation. While many approaches were introduced in the literature to achieve this goal (*e.g.,* [13, 29]), we focus mainly of search-based techniques.

In search-based software testing, the set of possible test cases for a given program is represented as a search space, and the problem of defining test cases is solved using an optimization algorithm [20] that allows to select a solution maximizing some selected adequacy criteria. One criterion commonly used is branch coverage, but other criteria, such as exception coverage or weak mutation [25], have been targeted as well. Random search [15] is the simplest form of search-based optimization algorithm: test cases are randomly selected from the search space and the most valuable ones according to the defined adequacy criteria are selected. Despite its simplicity, random search is among the most effective approaches in many contexts [13, 28]. The main limitation of random testing is the absence of guidance. For example, when targeting branch coverage, there are cases in which the probability of randomly satisfying a given branch condition is very low: a commonly used example is the condition `a == b && b == c`, with the integers `a`, `b` and `c` as inputs. For this reason, researchers experimented more complex approaches to solve this problem. Some approaches proposed in the literature use genetic algorithms (GAs) to evolve individual test cases (*e.g.,* single target approaches) [30], whole test suites [11, 26] or many test cases in parallel [23, 24].

Single-target approaches evolve test cases to cover a single target at a time. Tonella [30] introduced a chromosome representation to evolve test cases for classes in Object-Oriented code. Fraser and Arcuri [11] used such a representation to define the first approach to evolve whole test suites instead of single test cases.

Panichella *et al.* introduced two approaches to improve the whole test suite approach: MOSA [24] and DynaMOSA [23], both using a many-objective algorithm to evolve test cases in parallel.

Some of the proposed approaches have been implemented in publicly available tools. The ones mostly used for Java are two open source tools, *i.e.,* Randoop [21] and EvoSuite [10]. Randoop is based on feedback-directed random test case generation [22], while EvoSuite implements many evolutionary strategies, but it also features random test case generation and Dynamic Symbolic Execution [13, 28]. EvoSuite is also available as plugin for Maven, Eclipse and IntelliJ IDEA.

2.2 The Budget Problem in Search-Based Test Case Generation Tools

Despite providing a great support for generating test cases, the previously mentioned tools have some limitations. First, they cannot automatically determine the expected behavior of code (*i.e.,* the "oracle problem" [2]). Randoop and EvoSuite mitigate this issue by generating oracles that reflect the current behavior of the system. Such a strategy results useful especially in the context

Table 1. Operator-based probabilities in BCE [9]: the probability is purely based on the structure of the condition.

$\Pr(a \vee b)$	$\Pr(a) + \Pr(b) - \Pr(a)\Pr(b)$
$\Pr(a \wedge b)$	$\Pr(a)\Pr(b)$
$\Pr(\neg a)$	$1 - \Pr(a)$
$\Pr(a > b)$	0.5
$\Pr(a \geq b)$	0.5
$\Pr(a < b)$	0.5
$\Pr(a \leq b)$	0.5
$\Pr(a = b)$	q
$\Pr(a \neq b)$	$1 - q$

of regression testing. The other limitation of these tools is that they require developers to decide the time they want the tool to spend searching for test cases for each unit to test. Randoop and EvoSuite offer features that allow to set the budget at the project-level, without taking into account differences among the classes and the fact that some classes are naturally more difficult to test than others. Randoop allows to specify a global search budget that will be used for testing the whole project: it generates tests for each class until the budget is over. When this happens, the tool simply stops, *i.e.,* there may be classes left untested. In EvoSuite, the developer can specify a search budget that will then be used for the test case generation of each class (*e.g.,* 60 second per class).

To address this problem, Campos *et al.* [5] introduced a technique, implemented in EvoSuite, that uses information about the coverage achieved on the classes in past runs of test case generation to intelligently decide the search budget for a new version of such a class. The main limitation of this approach is that it requires information about the past coverage to work. For the first run of EvoSuite on a previously unseen unit to test, they use an approach that proportionally divides the search budget based on the number of branches of the classes.

Our work is motivated by the will of introducing an approach for intelligently splitting the search budget assigned by the developer at project level among the classes in the case where no previous coverage information is available. To achieve such a goal, a possible way is to predict the branch coverage that will be achieved for a given search budget.

2.3 Prediction of Branch Coverage

Ferrer *et al.* [9] introduced BCE, a metric that predicts the coverage achieved by automatic test case generation techniques on a given unit. BCE consists in modeling code as a Markov chain. Especially, the authors first extract the Control Flow Graph (CFG) of a program, then they consider the basic blocks of the CFG as the states of the Markov chain and its edges as possible transitions from a state to another. The authors also add a transition from the states created from exit nodes of the CFG to the initial state to simulate the behavior of test case generation techniques, that run a program multiple times. Markov chains require a probability associated to each transition, with the specific requirement that, for all the transitions $B_{a,i}$ from the state S_a, $\sum B_{a,i} = 1$.

For all the states that have a single possible transition, the probability associated to such a transition is 1. In all the other cases, the authors assign the probabilities to the branches $B_{a,b}$ using probabilistic rules based on the operators in the branching conditions. Table 1 shows how probabilities are computed for a given logical expression in a condition. For equality/inequality tests, the authors use $q = \frac{1}{16}$.

Given the Markov chain representing a program, the authors analytically compute, for each state S_i, its stationary probability π_i, *i.e.,* the probability of randomly traversing such a state. Using the stationary probabilities, for each state S_i they compute the frequency of appearance as $E[S_i] = \frac{\pi_i}{\pi_1}$. Finally, the authors compute the expectation of traversing a branch $B_{a,b}$ from S_a to S_b in a single run using the formula $E[B_{a,b}] = E[S_a] \Pr(B_{a,b})$. Given the set B^* of the branches with branch expectation lower than $\frac{1}{2}$, BCE is defined as the mean expectation of traversing such branches of a program:

$$BCE = \frac{\sum_{b \in B^*} E[b]}{|B^*|}$$

The main difference between BRANCHOS and BCE is that the latter is meant to predict the absolute coverage achieved by test cases, while BRANCHOS is aimed at predicting the coverage given a specific search budget (*coverage in time*).

3 ADAPTIVELY ASSIGNING THE SEARCH BUDGET

Consider a software project P, composed by n classes $CS = \{C_1, C_2, \ldots, C_n\}$, and a global search budget, B, assigned by a developer to an automatic test case generation tool in order to test the classes of P. It is possible to see the partitioning of B among CS as an optimization problem (*Budget Optimization Problem*). The *constraint* of such a problem is that the sum of the budgets b_i assigned to each class C_i should be equal to B. The *objective* is to maximize the total number of branches covered in the system. The *variables* of the problem are the search budgets assigned to the classes, *i.e.,* $b_1, b_2, \ldots, b_n$. Formally, the objective function is defined as:

$$\max \sum_{i=1}^{n} BCov_t(C_i, b_i) \tag{1}$$

where $BCov_t(C_i, b_i)$ represents the total number of branches covered by a test case generation tool t on a class C_i with b_i as budget. $BCov_t(C_i, b_i)$ ranges between 0 and $Branches(C_i)$, *i.e.,* the total number of branches in C_i. $BCov_t(C_i, b_i)$ can be also expressed as $Cov_t(C_i, b_i) \times Branches(C_i)$, where $Cov_t(C_i, b_i)$ is a function that returns the percentage of branch coverage (which ranges between 0 and 1), and $Branches(C_i)$ indicates the number of branches of the class C_i. Even if more convoluted, we will mainly use such a form in the rest of the paper since it simplifies the solution we will present later. Since $Branches(C_i)$ is fixed, the main problem in optimizing this function is that $Cov_t(C_i, b_i)$ is unknown *a-priori*, *i.e.,* it is necessary to run t on C_i with b_i budget to know such a value.

We introduce BOT (Budget Optimization for Testing), an approach to solve the *Budget Optimization Problem*. BOT uses a search-based algorithm to determine the search budget allocation that will allow to achieve the maximum coverage. To do this, it requires an estimation of the branch coverage achieved on a class with a given budget (*i.e.,* $Cov_t(C_i, b_i)$ in the objective function). To estimate such a value, BOT uses BRANCHOS (BRANch Coverage HistOry Seer), a novel approach to predict branch coverage "in time". In this section we first describe BRANCHOS and the predictors we introduce to estimate branch coverage in time; then, we describe our search-based algorithm that we use in BOT to optimize the budget allocation.

3.1 BRANCHOS: Predicting Coverage in Time

The basic assumption behind BRANCHOS is that the coverage achieved by test case generation approaches in time depends on features that can be measured on the classes under test. Under this assumption, classes with similar features are likely to have similar coverage in time. Given a class C for which we want to measure the expected coverage and a virtual search budget b, BRANCHOS

computes a set of metrics on C and uses such metrics and b as features of a regression model which has a dependent variable $Cov_p(C, b)$, *i.e.,* the predicted coverage that can be achieved on C by using b as search budget.

We use the following metrics to predict the coverage:

- **Class Size**: the size of a class is known to be negatively correlated with the coverage [8]. Intuitively, the higher the number of instructions to test, the longer the time needed to cover all of them. Also, having more instructions increases the likelihood that an exception occurs, making harder to cover some instructions. For these reasons, we include the Class Size as a feature in BRANCHOS. We measure the Class Size as the total number of bytecode instructions in the class.
- **Number of Branches**: since BRANCHOS is defined to predict the branch coverage, we hypothesize that the number of possible targets in a class is a relevant factor in determining the coverage achieved in time. We measure the number of branches (#Branches) as the number of conditional branching instructions and switch instructions at bytecode level.
- **Number of Methods**: like the previously outlined metrics, the number of methods (#Methods) may be an indicator of the complexity of the class. Specifically, it could be necessary to call methods in a specific order to test some branches. Consider the class `Stack`: if on an empty stack the method `pop` is invoked before a `push` operation is performed, the code related to the deletion of the element from the top of the stack is not executed. The higher #Methods, the higher the possible permutations of method executions to be tested.
- **Number of Infinite-Domain Fields**: while a high number of methods may indicate that many different invocation sequences are possible, it does not provide information about the possible states in which the class may be. The number of states of a class C may be computed as $States(C) = \prod_{f \in fields(C)} |type(f)|$, where $fields(C)$ is the set of fields in C and $|type(f)|$ indicates the domain size of a type, *i.e.,* the number of possible values it may have. For example, an `int` variable can have 2^{32} possible values. We compute the domain size of a primitive type T as $2^{bytes(T)}$, except for boolean which only have domain size 2 by definition. The domain size of array-types is always virtually infinite in Java since the size of arrays is decided when they are instantiated and it can not be deduced from the static type. Finally, we compute the domain size of non-primitive/array types (*i.e.,* type defined as classes) as $States(C)$. Given our previous definition, it is very likely that any class has a virtually infinite number of states since it is sufficient that it contains an array field or even a `String` type, which contains array fields. Consider two classes, one with a `String` field and one with ten `String fields`: it is clear that the latter can have a larger number of possible states than the former and it is more difficult to handle for a search-based test case generation technique. For this reason, instead of counting the number of states that an object of a given class can achieve, we count the number of infinite-size fields (#Fields).
- **Number of Infinite-Domain Parameters**: besides the order of invocation of methods, the arguments passed to the methods are the main aspect that determines which branches are covered. A high number of parameters may indicate that there are many different variables affecting the code execution, and testing the interaction among all of them may be more difficult [8]. Similarly to what we do for #Fields, we only take into account the infinite-domain parameters, *i.e.,* the ones that mostly increase the number of combinations to take into account. We measure the number of infinite-domain parameters (#Parameters) of a class as the sum of the number of parameters of all the methods belonging to the class under test.
- **Number of Private Methods**: in Java, test cases can only invoke public and protected methods, but they cannot call any private method of the class under test: private methods can only be used from inside the class. Therefore, the input of such methods cannot be directly decided

by the test case generation approach, that can only indirectly cover their instructions through public methods implemented in the class. Thus, we also measure the number of private methods (#PrMethods) of the class under test.

- **Branch Coverage Expectation**: we also use BCE, the metric introduced by Ferrer *et al.* [9] detailed in Section 2. We compute BCE for all the methods in the class under test and we consider the mean as a feature in BRANCHOS.
- **Branch Switching Difficulty**: branching conditions may have very different probabilities of being evaluated as *true* or *false*. Since search-based test case generation is random at its base, knowing such probabilities *a-priori* would be of fundamental importance to predict the coverage in time. The higher the absolute difference between such a probability and 0.5, the harder it would be to switch the truth value of that condition either to *true* or *false*. We introduce PrEst, a technique for estimating the probability of satisfying a condition, and we use such an estimation to compute the Branch Switching Difficulty (BSD). Given a condition c and its estimated probability $P(c)$ (we detail how we estimate it in the next paragraph), we compute BSD as $|0.5 - P(c)|$. We compute aggregate BSD at class level using two distinct metrics: we compute the *maximum* and the *mean* BSD of the conditions of all the branching instructions in the class under test. The rationale for also using the maximum here is to consider the most challenging condition as a feature of our model.

It is worth noting that we did not consider the Cyclomatic Complexity as a metric. We did this because it is measured as a function of the number of branches, a metric we already consider.

Estimating the Probabilities of Conditions. Ferrer *et al.* [9] estimate the probabilities of conditions using simple probabilistic properties, that uniquely depend on the binary operators used in them. For example, they use 0.5 as fixed probability of satisfying all the conditions containing the <=. Table 1 in Section 2 shows all the probabilities associated to the different comparison operators (the authors use $q = \frac{1}{16}$).

Such an approach suffers from two main problems. First, it does not take into account the nature of the operands: if both a and b are parameters of the method, the probability of satisfying any condition may differ from the case where a and b are constants or are not strictly dependent on the user input (*e.g.,* they represent a timestamp). The second problem is that this approach does not take into account the context in which the condition is. Consider for instance the condition `i < list.size()`. It is reasonable to assume that these types of conditions are more frequently satisfied (*i.e.,* true value) than they are not in practice. This happens because they are frequently used in `for` loops, and in these cases the analytical probability of satisfying the condition is $\frac{\texttt{list.size}}{\texttt{list.size+1}}$, which tends to be higher than 0.5.

We try to tackle these two problems defining PrEst, a context-aware approach for estimating the probability of satisfying a condition. We achieve this goal by mining information about the coverage of conditions from other software projects in order to estimate more precisely the probabilities of new conditions.

Consider the condition C_{BI} of a branch instruction BI. Without loss of generality, we can assume that every C_{BI} appears in the form $v_i \diamond v_j$, where v_i and v_j are variables and $\diamond$ is a binary operator. Indeed, a branch instruction with a more complex condition can be broken into many branch instructions in the binary form previously presented. This is what happens in low-level program representations, like bytecode. Given this, we aim to determine the sequence of operations that involve v_i and v_j and if their value depends, in some way, on parameters and/or instance variables.

To achieve this goal, we use a lightweight version of backward slicing [3]: we select all the instructions that affect a given line (specifically, a condition); unlike backward slicing, we do not necessarily want to construct a compilable program or a program that preserves the behavior of

Listing 1. Example method for the computation of *CS*.

```
1  public void test(String x, int num) {
2      if (num > 0) {
3          x = x.downcase();
4      } else {
5          x = x.upcase();
6      }
7      if (x.equals("TEST")) { // Target condition
8          // ...
9      }
10 }
```

the original one as for a specific variable. To define the sequence of instructions in which we are interested, we create an empty stack O containing at the beginning only the comparison operator $\diamond$ of C_{BI}. We start from the variables v_i and v_j and we look for the instructions in which such variables are assigned, respectively I_i and I_j. If these instructions contain a method call or an arithmetical operation, we push them on top of O. We keep doing this also for the variables used in I_i and I_j, until we get to parameters, instance variables or already analyzed variables (it may happen in loops). At the end of this process, O contains the list of operations affecting data that are used in the condition C_{BI}. We memorize whether instance variables and/or parameters are met in this process. To remark the difference with backward slicing, let us consider the example in Listing 1: the backward slice of the target condition (line 7) includes the first `if` statement (line 2): indeed, this is needed to preserve the original behavior of the program. On the other hand, we do not consider such an instruction since `num` does not directly affect `x`, the variable we are analyzing.

Therefore, at the end, we know (i) the sequence of operations O (*i.e.,* the operations performed from the beginning of the method to the branch instruction of which we want to know the probability), and (ii) if parameters and instance variables are involved in the computation. We call the data structure containing such information *condition signature* (*CS*). We conjecture that conditions that have similar condition signatures have similar probabilities of being satisfied. We call a condition signature *controllable* if it depends on parameters and/or instance variables and *not controllable* otherwise.

Given a dataset D of condition signatures associated with the probability of being satisfied using a search-based approach, we define a method for determining the probability of a new condition signature of being satisfied. We compute the context-aware probability $\Pr_c$ of satisfying a condition C with a condition signature CS with the following formula:

$$\Pr_c(C, CS) = \frac{W \Pr_s(C) + \sum_{Q \in D} \Pr_c(Q) sim(CS_C, Q)}{W + \sum_{Q \in D} sim(CS_C, Q)} \tag{2}$$

where $sim(CS_1, CS_2) \rightarrow [0, 1]$ is a similarity function between condition signatures, $\Pr_s$ is the structural probability function defined by Ferrer *et al.* [9] and W is a parameter representing the weight of the structural probability.

In other words, we compute the probability of satisfying a condition C as the weighted sum of the probabilities of satisfying conditions with similar condition signatures, where the weights are the similarities between CS and the other condition signatures in the dataset. We also add to this weighted sum the mean $\Pr_s(C)$, the structural probability of satisfying C. We use a constant, W, as weight for $\Pr_s(C)$. We introduced W to be able to tune the level of importance we should give to $\Pr_s(C)$ when combining it with the structural probability instead of assuming such two parts as equally important. We also consider structural probabilities to handle the cases in which we find conditions with condition signatures different from the ones in the dataset.

Indeed, if there is no condition signature with high similarity, the correction over the structural probability is very small; on the other hand, if the similarity is high, the correction is more significant.

We define a similarity measure, $sim_k(CS_1, CS_2)$, based on the length of the Longest Common Subsequence (LCS) of CS_1 and CS_2. First, we compute the length of the LCS between CS_1 and CS_2 ($LLCS$). Such a measure indicates how many operations appear in both the sequences in the same order, even if one of the sequences contains more operations than the other one. Then, we normalize $LLCS$ on the length of the longest sequence between CS_1 and CS_2 and we raise the resulting value to the power of k (the role of the k parameter is, as explained later, to penalize the similarity of sequences sharing few operations):

$$sim_k(CS_1, CS_2) = \begin{cases} [\frac{LLCS(CS_1,CS_2)}{\max(|CS_1|,|CS_2|))}]^k, & \text{if } cp(CS_1, CS_2) \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

where $cp(CS_1, CS_2)$ is true only if the signatures are both *controllable* or both *not controllable*.

Consider the following example, in which we set k to 2:

$$CS_1 = \{\texttt{Integer.parseInt},\ +,\ *,\ <\}_{contr}$$

$$CS_2 = \{\texttt{Integer.parseInt},\ *,\ >\}_{contr}$$

In this case, since both the sequences are controllable, the similarity is not 0 *a priori*. We first compute $LLCS$, which is 2 in this case (*i.e.,*`Integer.parseInt`, `*`). We divide the $LLCS$ by the length of CS_1 (the longest sequence), obtaining 0.5 as a result. Finally, we compute 0.5^2, and we have that $sim_2(CS_1, CS_2) = 0.25$. It is worth noting that the higher k, the lower the similarity of sequences sharing few operations. In our context, it is important to penalize (*i.e.,* minimize) the similarity of two sequences only sharing a few operations, to minimize their impact in the computation of $\Pr_c(C)$. We tune the parameters k and W in our study.

The similarity measure is instead set to 0 when one of the signatures is controllable and the other one is not. In these cases, indeed, although the sequences of calls are identical, the truth value of only one of the conditions can be changed by the search-based technique modifying the value of a parameter and/or of an instance variable. Therefore their satisfaction probabilities may be unrelated.

It is worth noting that PrEst is computed statically: if there are alternative branches that assign a given variable in different ways, we cannot know which one will be executed. In such cases, we include in the CS the operations performed in *all* the alternative branches. Consider again the example in Listing 1: the last condition may depend on the execution of either `downcase` or `upcase`. We include in the CS both such operations. The resulting CS would be the following:

$$CS_e = \{\texttt{String.downcase},\ \texttt{String.upcase},\ \texttt{String.equals}\}_{contr}$$

Such a CS never reflects an actual execution scenario: only one of the methods will be called for a given call to the method `test`. However, if the dataset D contains a sequence that includes only one of the methods, it will still have a high similarity with the CS built in the example: indeed, the similarity measure we use, *i.e., LLCS*, considers also the cases in which there is a lack of one or more elements in the sequence, but the order is the same.

The last step to compute $\Pr_c(C)$ is the creation of dataset D of condition signatures associated with the probability of being satisfied using a search-based approach. We show how we built such a dataset and how we estimate the parameters of PrEst in Section 4.

Training the Model. A requirement for BRANCHOS is a dataset D_{cov} containing information about the coverage achieved in time for a set of classes. Such a dataset must contain triples $\langle C_i, b_j, Cov(C_i,b_j)^*\rangle$, where C_i is the class under test, b_j is the search budget consumed and $Cov^*(C_i, b_j)$ is

the coverage achieved using b_j as search budget for C_i. Then, we measure all the metrics previously described (*i.e.,* the independent variables of BRANCHOS) on all the classes in D_{cov}, defining a new dataset, D_{feat}, containing tuples $\langle C_i, M_1(C_i), ..., M_n(C_i)\rangle$, where M_i are the metrics we use. The training set used by BRANCHOS is $T = D_{cov} + D_{feat}$, where D_{cov} and D_{feat} are merged on the class to which they belong. Note that a class C_i can appear in T multiple times, with the same values for the metrics, but different budget (b_j) and, possibly, different coverage levels achieved ($Cov^*(C_i, b_j)$).

It is worth noting that BRANCHOS is designed to be independent on the search-based test case generation technique used: if a different tool or a different technique is used, it would be sufficient to build a new training set. In our experiments reported in Section 4 and Section 5 we experiment it with a specific test case generation technique, *i.e.,* MOSA.

3.2 Optimizing the Search Budget

We propose a search-based approach to optimize the budget allocation inspired by hill climbing. A *solution* represents the budget that will be assigned to each class under test and the *objective function* is an approximation of the one presented in Equation 1, in which we use the prediction provided by BRANCHOS as a proxy for $Cov_t(C_i, b_i)$, *i.e.,* the coverage that will be achieved on the class C_i using search budget b_i.

Algorithm 1 shows our optimization algorithm. It requires three parameters: (i) the list of classes C; (ii) the initial solution S, which is an array of the same length of C and indicates the budget assigned to each of them; (iii) the maximum number of iterations $maxIter$. At each iteration, we first determine, for each class, which budget increment would allow it to have the most cost-effective coverage improvement (*i.e.,* the highest improvement at the lowest budget cost). We determine such a budget cost ϵ_i for each class. To compute the cost-effectiveness we simply divide the coverage gain by the cost. For example, let us assume the following scenario: there is a class C_i with 60 seconds of budget that has a predicted coverage of 20 branches; adding 5 seconds would result in an increase of 5 branches (25 covered branches), while adding 10 seconds would result in an increase of 7 branches (27 covered branches). In this case, we set $\epsilon_i = 5$ since its cost effectiveness is higher ($\frac{5}{5} > \frac{7}{10}$).

At this point, for each class C_i, we have (i) a ϵ_i, which indicates the best budget increment, (ii) a coverage increment $BCov(C_i, S_i + \epsilon_i) - BCov(C_i, S_i)$, and (iii) the cost-effectiveness of such a change, computed as $\frac{BCov^*_t(C_i, S_i+\epsilon_i) - BCov^*_t(C_i, S_i)}{\epsilon_i}$. We pick the class for which it is possible to achieve the most cost-effective budget increment, and we call it C_h. Then, we search for the class C_l for which decrementing the budget by ϵ_h, *i.e.,* the budget required by C_h, allows to minimize the coverage loss. Finally, we try to increment the total coverage by removing ϵ_h from the budget of C_l and assigning it to C_h: if such an operation increments the total coverage, we move the budget and we repeat the procedure, otherwise we stop. The algorithm stops anyway when the maximum number of iterations is reached.

To determine the initial solution S for BOT we divide the budget proportionally to the number of branches of the classes, *i.e.,* we use the "Budget" approach defined by [5].

We preferred to use a local-search technique because it is more natural for the budget-optimization problem: such a problem has a constraint, *i.e.,* the budget used should be equal to the total budget allocated at project level. This can be easily achieved with a local-search technique: as we showed, all the operations performed at each step do not modify the total budget by design. On the other hand, using a genetic algorithm to achieve the same goal would be more problematic: it would be necessary to define a chromosome representation ensuring that all possible chromosomes are valid solutions. For example, a trivial solution would be to represent a solution as an array of integers, where the number at position i represents the budget to assign to the i-th class. However, most of

Algorithm 1 BOT algorithm.

Data: C, S, $maxIter$
Result: S
$iter \leftarrow 0$
while $iter \leq maxIter$ **do**
  for $i \in \{0, \ldots, |C|\}$ **do** ▷ Determine the best budget increase ϵ for each class
    $\epsilon_i \leftarrow \max \arg_{\epsilon} \frac{BCov_t^*(C_i, S_i+\epsilon) - BCov_t^*(C_i, S_i)}{\epsilon}$
  end for
  $h \leftarrow \max \arg_{i=1}^{n} \frac{BCov_t^*(C_i, S_i+\epsilon_i) - BCov_t^*(C_i, S_i)}{\epsilon_i}$ ▷ Class with the highest gain given its ϵ_i
  $l \leftarrow \min \arg_{i=1}^{n} BCov_t^*(C_i, S_i) - BCov_t^*(C_i, S_i - \epsilon_h)$ ▷ Class with the lowest loss given ϵ_h
  $Q \leftarrow S$ ▷ Create a test solution Q by copying the current solution
  $Q_h \leftarrow Q_h + \epsilon_h$ ▷ Try to increase the budget for class that would get the maximum gain
  $Q_l \leftarrow Q_l - \epsilon_h$ ▷ Try to reduce the budget for the class that would have the minimum loss
  if $\sum Q_i \leq \sum S_i$ **then**
    return S ▷ If the coverage of the test solution did not increase, stop
  else
    $S \leftarrow Q$ ▷ If the coverage increased, the test solution Q becomes the current solution
  end if
  $iter \leftarrow iter + 1$
end while

the solutions obtainable with this chromosome would not meet the constraint (*e.g.,* $\langle 60, 100, 100 \rangle$ would not be a valid solution if the total budget is 180).

4 EMPIRICAL STUDY DESIGN

The *goal* of this study is to investigate whether (i) BRANCHOS is able to predict branch coverage in time and (ii) using BOT it is possible to find a budget allocation that allows to improve the project-level branch coverage. The *context* consists of 10,349 Java classes from ten popular Java software systems, while EvoSuite is used as a representative instance of search-based unit testing tools.

Our study is steered by the following research questions:

- **RQ$_1$**: *What is the coverage prediction accuracy of BRANCHOS when varying the search budget?* With this research question we want to understand what is the branch coverage prediction accuracy of BRANCHOS for different search budgets.
- **RQ$_2$**: *Is BOT able to improve the effectiveness of search-based unit testing tools in its ideal form?* We study whether using BOT with an ideal prediction approach it is possible to improve the branch coverage and the effectiveness at project level.
- **RQ$_3$**: *Is BOT able to improve the effectiveness of search-based unit testing tools when used with BRANCHOS?* We study whether using BOT with BRANCHOS it is possible to improve the branch coverage and the effectiveness at project level.

4.1 Parameters Tuning

Before discussing how we answered our research questions, we detail the context and the procedure used to tune (i) the k and W parameters of the novel metric PrEst used as coverage predictor by BRANCHOS The results of the tuning will be presented at the beginning of the section discussing the achieved results. The best parameters will be used to answer our research questions.

Tuning of PrEst. Such a tuning was run on five projects. We selected the five most popular libraries in Maven by excluding those that are used by EvoSuite itself (*e.g.,*`junit`). Indeed, the version of EvoSuite we used does not work when used to generate test cases for projects it depends upon. This process resulted in the selection of the five projects listed in the top part of Table 2. From each of these projects, we extracted the 100 classes having the highest number of conditional branches and run on them a modified version of EvoSuite that stores the number of times each branch condition was evaluated both as *true* and *false*. We used a search budget of 120 seconds per class. In this context, multiple runs are not necessary, because conditions are already evaluated multiple times in a single run. We used the empirically recorded frequencies to estimate the probabilities associated with each branch condition. The instances of our dataset are composed by the branching instructions and the probability to evaluate their conditions as *true*. In total, such a dataset is composed by 2,588 pairs ⟨*branching instructions, probability*⟩. Note that we preferred to focus only on the 100 classes having the highest number of conditional branches in each project for two reasons: (i) we wanted to balance in our dataset the data points extracted from the subject systems, since the five projects have substantially different size (*i.e.,* from 123 to 1,865 classes); (ii) given the first condition and the fact that the cost of running EvoSuite on each class is the same (*i.e.,* 120 seconds), we preferred to consider the 100 classes having the highest number of conditional branches in order to maximize the number of collected pairs for our dataset.

Once built the dataset, we tune the parameters required by BRANCHOS comparing the mean squared error achieved by different versions of it. We evaluated BRANCHOS by varying k from 1 to 10 at steps of 1 and W from 0 to 10 at steps of 1, for a total of 110 $\langle k, W \rangle$ combinations. We choose the parameters k and W that allow us to achieve the lowest mean squared error. Given a branch condition c, its actual probability empirically assessed $\Pr^*(c)$ and its predicted probability, $\Pr(c)$, we compute the squared error as $e_c^2 = (\Pr(c) - \Pr^*(c))^2$ and the mean squared error as the mean of e_c^2 for all the branch conditions extracted from our dataset.

Tuning of the Search Algorithms. BOT does not rely on any parameter. However, as previously stated, the initial solution that is optimized by such an algorithm is based on the "Budget" approach defined by Campos *et al.* [5] which allocates the budget based on the number of branches of the classes under test. Such an approach requires to specify the minimum budget that should be allocated for each class. The authors use a one minute budget in a context in which the global budget for each project is 3 minutes times the number of classes. Since our global budget is 1 minute times the number of classes, we needed to use a different parameter, otherwise we would have had the same results of a fixed budget allocation approach. To set such a parameter, we used one of the 10 projects of our dataset (*i.e.,* PDFSam). We tried different values for the minimum allocation budget ranging between 5 and 55, with a step of 5. We chose the value that allows to achieve the highest number of actual covered branches for the "Budget" approach alone (*i.e.,* without running BOT): we did this since we use "Budget" as a baseline and we want to perform a fair comparison with it, *i.e.,* we did not want to choose a value that favours the optimization step.

4.2 Context of the study

To answer our research questions, we use the ten systems reported in the bottom part of Table 2. We used the ten most popular Java software systems from the SF110 dataset [12].

While all ten systems have been used to answer $\mathbf{RQ_1}$, nine were used for $\mathbf{RQ_2}$ and $\mathbf{RQ_3}$ (*i.e.,* all but PDFsam, that was used for the tuning of the search algorithms). We kept into account all the classes belonging to the main JARs of such software systems, excluding those that could not be tested (anonymous classes, interfaces and abstract classes). In total, our study context consists of 10,349 classes.

Table 2. Context of the study: for each project we report both the total number of classes and the number of classes considered in our study.

	Project	Total classes	Classes considered
Tuning	commons-io	123	100
	commons-lang	132	100
	guava	1,865	100
	mockito-all	752	100
	jackson-databind	786	100
$RQ_{1\text{-}2}$	Netweaver	210	185
	Squirrel SQL	1,582	894
	SweetHome 3D	452	159
	Vuze	3,949	1,948
	Freemind	851	411
	Checkstyle	219	134
	Weka	1,466	851
	Liferay	8,616	5,353
	PDFsam*	406	338
	Firebird	364	197
	Total	**21,773**	**10,970**

To build a dataset that we could use to answer $\mathbf{RQ_1}$, $\mathbf{RQ_2}$, and $\mathbf{RQ_3}$, we recorded the coverage achieved by EvoSuite during each second of execution on all classes of our subject system. We used MOSA [24] as test case generation approach, which is the one with the best performance available in EvoSuite. We did not use DynaMOSA [23], the evolution of MOSA, because such an approach was not implemented in the latest release of EvoSuite at the time of the experiment. We set the search budget at 300 seconds per class. In total, our D_{cov} dataset is composed by 3,104,700 instances. To take into account the randomness of the test generation process, we run EvoSuite five times for each class.

It is worth noticing that generating test cases for all the classes of our study and with such a large search budget per class is very expensive. About 13 days of computation were needed to acquire all the data running six parallel instances of EvoSuite on a dedicated virtual machine with 8 cores and 16GB of RAM. For this reason, we preferred to generate test cases for more classes rather than performing more than 5 runs for the same classes. We discuss this threat in Section 6.

4.3 Experiment Methodology

To answer $\mathbf{RQ_1}$, as a preliminary step we compute the mean coverage achieved for each search budget and class among the five EvoSuite runs. Then, we perform a cross-project validation, *i.e.,* we train the model on nine projects and we test it on the tenth one, using one project at a time as test set, to compute the coverage as predicted by BRANCHOS ($\mathbf{RQ_1}$). Such a strategy was selected to avoid training the model on classes of the system on which we will then predict the coverage. We choose Random Forest [4] as regression technique since it performed better than other techniques in our preliminary tests. Since BRANCHOS is the first approach designed to predict branch coverage in time, we do not have an actual baseline.

BCE cannot be used for this purpose either, since it cannot take the search budget as input. For this reason, we use a trivial approach as baseline: we define several *constant regressors* CP_b, one

for each possible search budget b we considered in our study. With *constant regressor* we mean a regressor that always predicts the same value, without taking into account the characteristics of the input class. The constant regressor $CP_b(C)$ always returns the mean branch coverage achieved on all the classes in the training set in exactly b seconds, regardless of the input class C. Therefore, given any class C and a search budget b, the baseline returns $CP_b(C)$. Since we run EvoSuite for 300 seconds, the baseline approach contains 300 CP_b, with $b \in \{1, \ldots, 300\}$. Also in this case, we use the mean squared error to compare the two approaches. Given a class C and a search budget b, we have the actual coverage, $Cov^*(C, b)$, and the predicted coverage, $Cov(C, b)$. We compute the error as $e_C = |Cov(C, b) - Cov^*(C, b)|$.

We use a Wilcoxon signed-rank test on the squared errors of the models to compare them. Our null hypothesis is that "*there is no difference between the errors introduced by the models*". We reject the null hypothesis, and thus we consider the difference *significant*, if the p-value of the test is lower than 0.05.

We also measure the effect size, using the Cliff's delta [6], to understand the magnitude of difference among the models. Cliff's delta δ lays in the interval [-1, 1]: the effect size is **negligible** for $|\delta| < 0.148$, **small** for $0.148 \leq |\delta| < 0.33$, **medium** for $0.33 \leq |\delta| < 0.474$, and **large** for $|\delta| \geq 0.474$. If $\delta > 0$, it means that the first distribution (in our case, always the distribution representing the results achieved by our approach) is larger than the second (the baseline), while the opposite happens otherwise. Finally, we also report two additional measures: (i) the Pearson correlation between the predicted values and the actual values: this measure indicates if a higher actual value results in a higher predicted value; (ii) PRED(25), a metric which indicates how many predictions have a relative error lower than 25%. Given the actual and the predicted vectors, p^* and p, we compute PRED(25) as:

$$PRED(25) = \sum_{i}^{n} \begin{cases} \frac{1}{n}, & \text{for } \frac{|p_i - p_i^*|}{p_i^*} \leq 0.25 \\ 0, & \text{otherwise} \end{cases}$$

To answer $\mathbf{RQ_2}$ and $\mathbf{RQ_3}$ we use our search algorithm previously described to optimize the total coverage predicted for each of the nine projects used in this research question. Since in $\mathbf{RQ_2}$ we want to evaluate BOT in an ideal scenario, we assume the existence of a perfect prediction model, which makes no mistakes ("Ideal") and we use it to run the optimization ($\text{BOT}_{\text{Ideal}}$). To do this, we always predict the coverage actually achieved by EvoSuite in the five runs we executed. In $\mathbf{RQ_3}$, instead, we evaluate BOT in the realistic scenario ($\text{BOT}_{\text{BRANCHOS}}$) in which we use the best prediction approach available (*i.e.,* BRANCHOS). For both the research questions, as a first step we run the approaches we compare (*i.e.,* $\text{BOT}_{\text{Ideal}}$, $\text{BOT}_{\text{BRANCHOS}}$, and the two baselines described below) on the nine projects to determine the search budgets they would assign to each class. Then, we use such budgets to compute the actual coverage achieved by EvoSuite distinctly in the five runs we completed. We used the coverage results obtained to answer $\mathbf{RQ_1}$ to do this, *i.e.,* we did not need to re-run EvoSuite.

It is worth noting that our approach requires some time to optimize the search budget. Thus, to take this into account in the comparison with the baselines, we compute the average time needed to run our approach and we remove it from the global budget of each project. This was done to simulate a scenario in which the developer has a given budget B available to test a system, and B includes both the time needed to run our approach as well as the time needed to run the test case generation. To do this, we preliminarily run our algorithm with the exact setting of the experiment. Then, we computed the average time needed. We found it is very fast since it takes only 2 seconds for project, on average. When experimenting both $\text{BOT}_{\text{Ideal}}$ and $\text{BOT}_{\text{BRANCHOS}}$, we remove such time from the global budget of each project. We report, for each project, the mean number of

total branches covered. For each project, we use the t-test to check if the difference between the total number of covered branches changes when using our approach and the baselines. The null hypothesis is that the project coverage achieved over different EvoSuite runs is the same: we reject such an hypothesis if the p-value is lower than 0.05. Finally, we also report the Cohen's d statistic [7] to compute the effect size of the significant differences. The effect size is *negligible* for $|d| < 0.2$, it is *small* for $0.2 \leq |d| < 0.5$, it is *medium* for $0.5 \leq |d| < 0.8$, while it is *large* otherwise. It is worth noting that, in this case, we use parametric statistics (*i.e.,* t-test and Cohen's d), which implicitly assume the normality of the distributions we compare. We can safely make this assumption since each sample of the distributions we compare (*i.e.,* the project coverage) is the sum of many random variables (*i.e.,* the class coverages): according to the Central Limit Theorem, regardless of the distribution of the single random variables, the distribution of their normalized sum tend to be normal. EvoSuite failed to generate test cases in some runs for some classes: this means we have the coverage achieved in five runs only for a subset of the classes we take into account. For $\mathbf{RQ_2}$ and $\mathbf{RQ_3}$ we only consider the classes for which EvoSuite successfully completed all the five runs, *i.e.,* 9,787 classes in total.

To check if our approach is able to improve the effectiveness of generated tests we also complete five additional runs of EvoSuite for each class to compute the strong mutation score. We compare the number of mutants killed running tests generated using our optimization strategy and the baseline which achieves the highest branch coverage. Also in this case we use the t-test to check if the number of killed mutants between the tests generated with our approach and the ones generated with the baseline is significantly different and the Cohen's d to report the effect size of significant differences.

Baselines. We compare $\text{BOT}_{\text{Ideal}}$ and $\text{BOT}_{\text{BRANCHOS}}$ with two baselines: the first one equally divides the project-level search budget among the classes (the *Simple* approach described by Campos *et al.* [5]); the second one is the *Budget* approach introduced by Campos *et al.* [5], which the authors use when no historical information is available. The *Budget* approach divides the search budget proportionally to the number of branches of the classes: a higher budget is assigned to classes with more branches to cover. Specifically, we first compute the budget per branch rate (b_{br}) dividing the total budget by the total number of branches; then, for each class C with $branches(C)$ branches, we compute the candidate budget $b_{br} \times branches(C)$. Since such a value can be extremely low for small classes (even 0) and extremely large for big classes, we set a minimum and a maximum allocable budget. If the candidate budget is not in such a range, we use the minimum or the maximum value instead and we update b_{br} based on the remaining budget and the remaining branches. We use 300 as maximum allocable budget. We tune the minimum allocable budget on the same project we use for tuning BOT (*i.e.,* PDFSam) using all the possible values between 5 and 55. We selected 40 as minimum allocable budget since it is the value that allows to achieve the highest coverage on such a project. For our algorithm we assign a project-level search budget of $60 \times |C|$, *i.e.,* 60 seconds times the number of classes of the project.

We summarize the metrics computed and the analysis performed to answer the two research questions in Table 3.

4.4 Replication Package

The data and code used in our study are made publicly available[2]. In particular, we provide: (i) all the datasets, including all the classes subject of our study, (ii) the raw data generated to answer the three research questions, (iii) the results of the tuning, and (iv) all the scripts and the programs used. We do this to make the experiment fully replicable to foster future research in this field.

[2]https://dibt.unimol.it/report/bot-tosem/

Table 3. Summary of the metrics computed and the analysis performed to answer the research questions.

	Analysis	Description	Interpretation
RQ_1	Correlation	Pearson correlation coefficient.	The higher, the better.
	PRED(25)	Percentage of predictions with a relative error of 25%.	The higher, the better.
	MSE	Mean of the squared errors.	The lower, the better.
	Wilcoxon test	Non-parametric comparison between two distributions.	Significant if the p-value is lower or equal to 0.05.
	Cliff's δ	Non parametric effect size.	The higher, the larger the difference.
RQ_{2-3}	t-test (p-value)	Parametric comparison between two distributions.	Significant if the p-value is lower or equal to 0.05.
	Cohen's d	Parametric effect size.	The higher, the larger the difference.

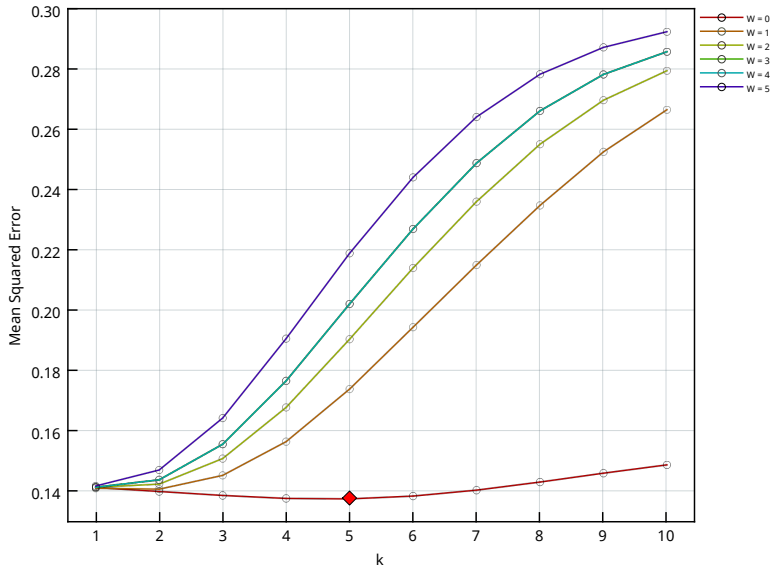

Fig. 1. Tuning of k and W parameters

5 EMPIRICAL STUDY RESULTS

In this section we discuss the results of our empirical study. Before answering our two research questions, we show the results of the parameters' tuning.

5.1 Parameter Tuning Results

PrEst. For computing PrEst we need to tune two parameters, k and W. Fig. 1 shows how different combinations of k and W perform: we only show results for $W \leq 5$, since greater values of W have worse results, in line with the trend depicted in Fig. 1. The best results (complete data in our replication package) are for $W = 0$. This shows that the estimation of probabilities using operators only, as done by Ferrer *et al.* [9], does not help our approach but, instead, it increases the error. The

Table 4. Comparison between BRANCHOS and the baseline.

	Correlation		PRED(25)		MSE		Wilcoxon test (p)	Cliff's δ
	BRANCHOS	Baseline	BRANCHOS	Baseline	BRANCHOS	Baseline		
Netweaver	**0.469**	0.052	**50.0%**	38.7%	**0.086**	0.099	**<0.001**	0.166 (small)
Squirrel SQL	**0.527**	0.054	**42.6%**	23.6%	**0.098**	0.119	**<0.001**	0.197 (small)
SweetHome 3D	**0.705**	0.053	**23.3%**	23.5%	**0.087**	0.228	**<0.001**	0.393 (medium)
Vuze	**0.592**	0.080	**45.1%**	24.4%	**0.098**	0.139	**<0.001**	0.246 (small)
Freemind	**0.472**	0.031	**25.8%**	15.7%	**0.153**	0.238	**<0.001**	0.272 (small)
Checkstyle	**0.648**	0.033	**30.6%**	20.6%	**0.090**	0.164	**<0.001**	0.282 (small)
Weka	**0.603**	0.124	35.6%	**38.2%**	0.089	0.094	**<0.001**	0.055 (negl.)
Liferay	**0.602**	0.076	**55.3%**	12.0%	**0.080**	0.125	**<0.001**	0.432 (medium)
PDFsam	**0.720**	0.041	**40.4%**	14.6%	**0.089**	0.169	**<0.001**	0.360 (medium)
Firebird	**0.730**	0.083	**59.7%**	21.4%	**0.067**	0.091	**<0.001**	0.328 (small)
Average	**0.622**	-0.151	**48.3%**	18.6%	**0.089**	0.131	-	-

best value for the k parameter is 5, for which the approach achieves a slightly lower error compared to other variants of the approach having $W = 0$. Such a quite large k exponent drastically reduces the weight of sequences that are substantially different from the one under test (see Section 3).

We also report some additional statistics for the best configuration of our approach, *i.e.,* PrEst ($W = 0$, $k = 5$). The mean squared error is 0.138 and the PRED(25) is 18.2%. The error distribution ($P(c) - P(c)^*$) of PrEst is, overall, symmetrical (skewness of 0.07, *i.e.,* slightly skewed towards positive values). This shows that the approach is accurate, although, on average, it slightly overestimates the condition probabilities. Such results may seem not very encouraging, in absolute terms, since in more than 80% of the cases the relative absolute error is greater than 25%. However, statically estimating the probability of satisfying a condition is a very challenging task. Indeed, checking if a branch is infeasible, which is an instance of the problem we try to solve, is an undecidable problem. Therefore, considering the tackled problem, we think this is an acceptable result, although there is still room for improvement.

As for the parameter required by the "Budget" approach [5], which provides the initial solution for BOT, we found that the best value for ϵ is 20 (with a coverage of 1,721 branches). It is worth noting that this is a third of the average class budget we use in our experiment (*i.e.,* one minute). Such a value is analogous to the value Campos *et al.* [5] used in their experiment (one minute for an average class budget of three minutes).

5.2 RQ_1: Coverage Prediction in Time

We show in Table 4 the comparison between BRANCHOS and the baseline we considered. The mean squared error (MSE) achieved by BRANCHOS is always lower than the MSE achieved by the baseline. The p-value of the Wilcoxon test comparing the MSE achieved by the two approaches is always 0. This confirms that the difference is always statistically significant. The magnitude of the differences is *small* in most of cases, with variations across the ten projects. For Liferay, SweetHome 3D, and PDFsam, the magnitude of the difference is *medium*, while it is *negligible* only for Weka.

In almost all systems but one the PRED(25) shows that the baseline makes larger errors. The only exception is Weka, for which the PRED(25) of the baseline is slightly higher than the one achieved by BRANCHOS. Finally, it is worth noting that the baseline has, on average, a very weak correlation with the coverage. Indeed, the overall correlation is *negative*: this means that the higher the predicted coverage value, the lower the actual coverage, when considering all the instances in our dataset. On the other hand, BRANCHOS achieves, on average, a *strong* positive correlation (~0.62). To investigate more in-depth why BRANCHOS performs poorly on Weka, we report in Table 5 the average value for the features we consider in BRANCHOS for all the projects. It is

Table 5. The mean value of each feature we use in BRANCHOS for each project.

Project	BCE	BSD_{max}	BSD_{avg}	#Branches	Class Size	#PrMethods	#Fields	#Parameters	#Methods
Netweaver	0.42	0.09	0.02	30.71	371.04	0.46	5.33	17.12	11.64
Squirrel SQL	0.40	0.08	0.02	19.40	229.87	1.44	11.73	15.33	12.11
SweetHome 3D	0.35	0.14	0.02	99.65	901.87	5.31	24.96	33.95	38.26
Vuze	0.40	0.10	0.02	39.75	369.23	0.63	3.91	13.49	14.62
Freemind	0.37	0.10	0.02	28.08	293.56	1.50	17.72	17.91	14.30
Checkstyle	0.38	0.08	0.02	18.87	181.23	0.97	6.93	6.84	8.50
Weka	0.40	0.16	0.03	69.05	699.42	0.84	15.79	19.44	20.17
Liferay	0.42	0.07	0.01	14.74	230.59	0.22	4.61	16.04	14.72
PDFsam	0.39	0.10	0.02	22.14	282.58	0.78	17.57	11.94	8.66
Firebird	0.40	0.09	0.02	50.31	460.90	0.78	6.39	19.27	17.87

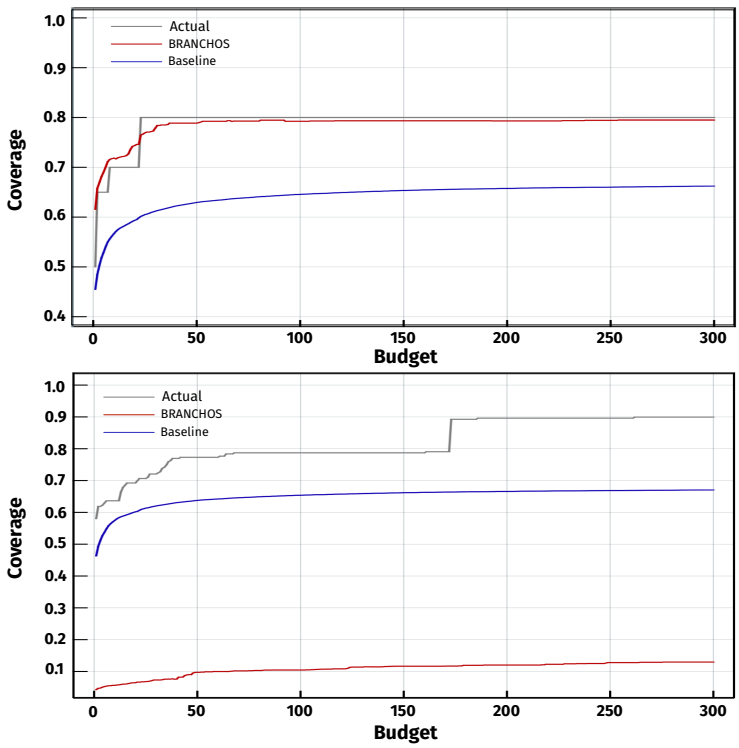

Fig. 2. Two examples of prediction from the Liferay project. In the first one, BRANCHOS achieves an almost perfect prediction of the actual coverage, while in the second one it underestimates it.

interesting to notice that, for Weka, the BSD_{max} and BSD_{avg} metrics are higher than for the other projects. Such a project contains a higher number of complex classes by nature since it is a toolbox for machine learning, with many state-of-the-art algorithms implemented: it is likely that training BRANCHOS on the other projects does not allow BRANCHOS to learn from enough complex classes. As a result, BRANCHOS is not able to provide accurate estimations when it encounters such classes.

Fig. 2 shows two examples of prediction for the project Liferay: At the top, a good prediction done by BRANCHOS (class `PortletPreferencesWrapper`); At the bottom, a bad prediction (class

`ContactModelImpl`). In the first case, BRANCHOS almost perfectly predicts the final coverage, but it also approximates well the coverage achieved with intermediate search budgets. In the second case, instead, BRANCHOS underestimates the coverage. This is probably due to the fact that some metrics, like the number of branches (91) and the number of fields (64), both quite high, deceive the regressor.

We also report in Fig. 4 the boxplot of the squared errors made by the two approaches when considering the dataset as a whole (*i.e.,* when merging all classes belonging to the ten systems in a single dataset). The median of the distribution depicted for BRANCHOS (*i.e.,* median=0.035) confirms the good accuracy of our approach. An important point to stress here is the performance achieved by the baseline. While it is clear that the baseline we exploited is trivial (*i.e.,* for each experimented search budget, the baseline predicts the mean branch coverage achieved across all classes in the training set for that specific budget) it is worth noticing that: (i) BRANCHOS is the first technique able to predict branch coverage in time (*i.e.,* taking the search budget into account), thus do not having any clear competitor; (ii) in our opinion, there is still value in showing that a very simple and straightforward approach cannot be applied to solve a complex problem such as the one we are tackling.

To understand the worth of the single metrics we considered, we built ten linear regression models, one for each feature. For a given feature f, the linear regression model L_f use f itself and the progressive budget to predict the coverage. We trained and tested the model on the whole dataset (without cross-validation) since in this case we are building a *descriptive* model rather than a *predictive* one. We report in Table 6 the correlation achieved by the models, which indicates the worth of each feature alone. As expected, the metrics specifically designed to achieve this goal are the most important ones (BCE, BSD_{max}, and BSD_{avg}). Interestingly, the number of private methods (#PrMethods) is, alone, more important than the total number of methods. Moreover, such a metric appears to be more important than the number of fields and parameters, which intuitively might appear as very correlated to the coverage that a test generation technique can achieve. This probably happens because while, on the one hand, a test generation technique has full control on parameters and some control on the fields, it has no direct control on private methods.

Finally, we wanted to check how the size of the training set influences the performance of BRANCHOS. To do this, we first defined a test set composed by a stratified sample of 500 classes, 50 from each project. Then, we tried to sample different number of classes, ranging between 500 and 9500, with a step of 500 classes, from the remaining classes. For each sample, we trained BRANCHOS and we tested it on the test set previously defined. Fig. 3 shows how the MSE changes when the size of the dataset increases. Once reached the 5,000 classes mark, the improvement achieved by adding more classes is very small, *i.e.,* the MSE is always around 0.075. This suggests that increasing even further the number of classes may have a limited effect on the performance of BRANCHOS.

Summary for RQ_1: In general, while the outcome of the comparison with the trivial baseline was quite expected, the results achieved by BRANCHOS are still satisfactory, with a correlation on the overall dataset of ~0.62, and a median squared error of 0.035.

5.3 RQ_2: Adaptive Budget Allocation in the Ideal Scenario

Table 7 reports the project-level branch coverage achieved by the algorithm with an ideal coverage prediction approach ($\text{BOT}_{\text{Ideal}}$), the one with the best prediction approach available ($\text{BOT}_{\text{BRANCHOS}}$) and the two baselines. We discuss the results obtained by $\text{BOT}_{\text{BRANCHOS}}$ in the next subsection.

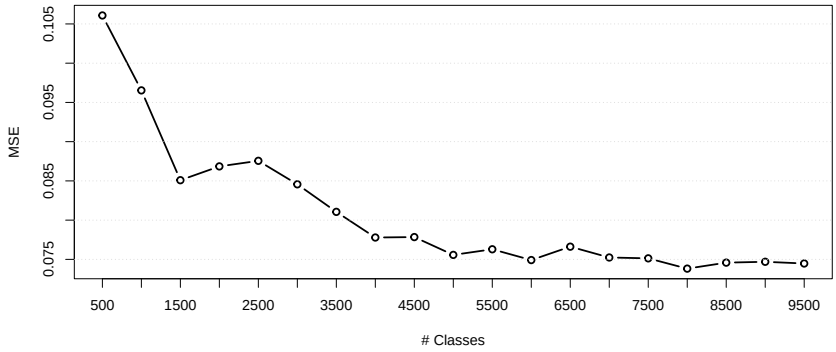

Fig. 3. Analysis of the variation of the mean squared error (y-axis) when the dataset size (x-axis) increases.

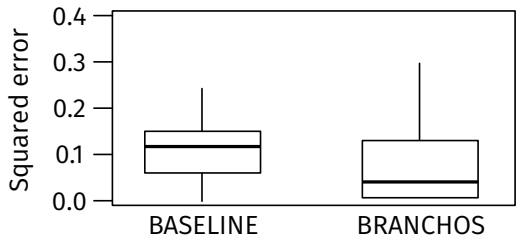

Fig. 4. Distribution of the squared errors by BRANCHOS and the baseline for the prediction in time (without outliers).

Table 6. Ranked worth of single metrics measured through the correlation between the branch coverage and linear regression models that used each feature in isolation and the progressive budget as independent variables and the coverage as dependent variable. The direction indicates the sign of the coefficient determined for the feature (△: positive, ▽: negative).

Metric	Direction	Correlation
BCE	△	0.4719
BSD_{max}	▽	0.3901
BSD_{avg}	▽	0.3671
#Branches	▽	0.3004
Class Size	▽	0.2819
#PrMethods	▽	0.2379
#Fields	▽	0.1705
#Parameters	▽	0.1197
#Methods	▽	0.1036

First, it is worth noting that the "Budget" approach [5] always achieves a higher coverage compared to the "Simple" approach. This confirms the result achieved by Campos *et al.* [5]. BOT_{Ideal} allows to improve the branch coverage at project level for all the projects we took into account. On average, we obtained an improvement of ~3.6% of the branch coverage. The project for which we obtained the lowest relative improvement was Liferay: BOT_{Ideal} allows to cover 0.93% that the "Budget"

approach could not cover. While this value is relatively small, it is worth noting that, in absolute terms, this means that our approach allows to cover 754 additional branches compared to the "Budget" approach. On the other hand, the project for which BOT_{Ideal} allows to achieve the best relative improvement is SweetHome 3D: in this case, our approach allows to cover 11.37% additional branches (390 in absolute terms).

The difference between the two approaches we compared is significant for all the projects. Besides, the effect size shows that the difference is never negligible: it is small for three projects out of nine, and large for six projects out of nine.

We also checked the difference of coverage at class level. Most of the classes (7,975) achieve the exact same coverage when using both the approaches. BOT_{Ideal} allows to increase the coverage of 1,592 classes (16.6%), while the "Budget" approach achieves a higher coverage for a single class, *i.e.,* `AssociatorEvaluation` from Weka. This happens because BOT_{Ideal} prefers to sacrifice a branch from such a class to cover more branches of another class at some step.

We report the results of the mutation analysis in Table 8. BOT_{Ideal} is always able to kill a higher number of mutants compared to the Budget approach, except for one project (Weka). BOT_{Ideal} kills a significantly higher number of mutants for five projects out of nine. For Weka, the difference is significant in favour of the baseline. For all the differences, even the ones for which we could not achieve statistically significant results, the effect size is large. We tried to understand why for such a project the Budget approach is able to kill a higher number of mutants. We found that Evosuite crashed on 12 classes for at least one of the approaches: BOT_{Ideal} allocated a high budget on ten of such classes (219 seconds each, on average) and, therefore, it "lost" such a budget. In total, for Weka, BOT_{Ideal} used a budget 995 seconds (*i.e.,* 16 minutes) lower. On average, despite such a peculiar case, BOT_{Ideal} allows to kill 3.3% more mutants compared to the baseline.

Summary for RQ_2: an ideal version of BOT would allow to allocate the budget better than the two baselines, generating tests covering, on average, 3.6% more branches, killing 3.3% more mutants, and improving the coverage of 16.6% of the classes.

5.4 RQ_3: Adaptive Budget Allocation with BRANCHOS

Like we did for $\mathbf{RQ_2}$, we report in Table 7 also the results achieved by $BOT_{BRANCHOS}$, *i.e.,* the BOT algorithm that uses the best branch coverage prediction approach available, *i.e.,* BRANCHOS. We underline the results achieved by the best *realistic* approach.

It can be noticed that $BOT_{BRANCHOS}$, as expected, always performs worse than its ideal form because of the prediction errors made by BRANCHOS. However, $BOT_{BRANCHOS}$ still allows to improve the project coverage of most of the projects (seven out of nine) compared to the best baseline, *i.e.,* the "Budget" approach [5]. The opposite happens just for two projects: Neatweaver and Checkstyle. For the former, the "Budget" approach only covers 8 additional branches, while for the latter such a difference is even smaller (2 branches). Indeed, the effect size is negligible and not significant in both the cases.

The project that obtained the smallest relative improvement is Freemind (0.12%, *i.e.,* only five additional branches covered), while the project that would benefit the most from $BOT_{BRANCHOS}$ is, again, SweetHome 3D (3.0%, *i.e.,* 103 additional branches). In general, the improvement is consistent for all the projects, but it is smaller than the one achieved in the ideal scenario: the average relative improvement is only 0.76%.

In this realistic scenario, the difference is statistically significant only for three projects out of nine, *i.e.,* SweetHome 3D, Weka (+1.6% coverage, *i.e.,* 511 additional branches), Vuze (+0.95% coverage, *i.e.,* +288 additional branches). Only in such cases, the effect size is large, while in the other

Table 7. Comparison between the branch coverage achieved by the approaches at project-level. We report in boldface the result of the best approach, while we underline the results of the best realistic approach (*i.e.,* excluding BOT_{Ideal}) for each project. We also report the statistical comparison between BOT_{Ideal}/$BOT_{BRANCHOS}$ and the best baseline (Budget [5]) over five runs (significant results in bold), along with the effect size (Cohen's d) and its magnitude (**N**egligible, **S**mall, **M**edium, or **L**arge).

Project	Covered Branches				$BOT_{BRANCHOS}$ vs Budget			BOT_{Ideal} vs Budget		
	BOT_{Ideal}	$BOT_{BRANCHOS}$	Budget [5]	Simple [5]	t-test (p)	Cohen's d		t-test (p)	Cohen's d	
Netweaver	**6,125.6**	6,088.8	6,096.4	5,911.2	**0.037**	-0.11	(N)	**0.022**	0.47	(S)
Squirrel SQL	**11,810.2**	11,688.6	11,651.4	11,339.0	0.104	0.05	(N)	**0.008**	0.21	(S)
SweetHome 3D	**3,818.0**	3,531.8	3,428.2	3,332.0	**0.018**	2.29	(L)	**0.001**	5.81	(L)
Vuze	**31,670.4**	30,458.6	30,170.8	28,751.2	**<0.000**	1.69	(L)	**<0.001**	9.01	(L)
Freemind	**3,833.8**	3,749.2	3,744.6	3,636.4	0.649	0.11	(N)	**<0.001**	2.36	(L)
Checkstyle	**1,098.8**	1,077.2	1,079.4	1,079.6	0.444	-0.04	(N)	**0.001**	0.36	(S)
Weka	**33,717.2**	32,363.6	31,852.6	30,659.6	**<0.000**	4.93	(L)	**<0.001**	12.05	(L)
Liferay	**81,960.8**	81,355.8	81,206.6	79,809.0	**0.010**	0.27	(S)	**0.001**	1.37	(L)
Firebird	**4,634.0**	4,548.0	4,504.6	4,182.8	**0.001**	0.77	(M)	**0.002**	1.88	(L)

cases it is negligible (four out of nine), small or medium (one project each). While the percentage of improvement is small, it is worth highlighting that $BOT_{BRANCHOS}$ allows to cover 1,127 branches that could not be covered using the "Budget" approach.

Also in this case we checked the difference of coverage at class level. Again, most of the classes (8,352) achieve the exact same coverage when using both the approaches. $BOT_{BRANCHOS}$ allows to increase the coverage of 730 classes (7.6%), while the "Budget" approach achieves a higher coverage for a 486 classes (5.0%). We tried to understand the characteristics of such classes: we found that the classes for which $BOT_{BRANCHOS}$ achieves a higher coverage are generally bigger in terms of number of branches (~96 branches, on average) compared to both the ones for which the "Budget" approach achieves a higher coverage (~55 branches, on average) and the ones for which the two approaches achieve the same coverage (~ 31 branches, on average).

Finally, Table 8 shows the number of mutants killed by our approach and the *Budget* approach. The tests generated with $BOT_{BRANCHOS}$ allow to kill a higher number of mutants for all the projects as compared to the baseline. Besides, the number of mutants killed is significantly higher for five projects out of nine. On average, BOT is able to kill 3.0% more mutants compared to the baseline. This shows that the approximation of the ideal BOT we defined is able to achieve a result close to the one achieved by the ideal version.

Summary for RQ_3: $BOT_{BRANCHOS}$ allocates the budget better slightly than the two baselines, generating tests covering 0.76% more branches and killing 3.0% more mutants.

5.5 Discussion

The results of our experiments show that both the approaches we introduced, *i.e.,* BRANCHOS and BOT, allow to improve the baselines we considered ($\mathbf{RQ_1}$ and $\mathbf{RQ_2}$). However, when we tried to combine them to understand the actual improvement that BOT can have in a real usage scenario, we found that such an improvement is quite slim. These results show how important is to accurately predict the branch coverage in time: while BRANCHOS is the best coverage prediction approach we experimented with, it is still insufficient to unleash the full potential of the optimization algorithm we introduced, BOT. Still, it allows to improve by 3.0% the average number of mutants killed by generated tests: this shows that, while there is still margin for improvement, our approach could be

Table 8. Comparison between the number of mutants killed by the approaches at project-level. We report in boldface the result of the best approach for each project. We also report the statistical comparison between BOT_{Ideal}/$BOT_{BRANCHOS}$ and the best baseline (Budget [5]) over five runs (significant results in bold), along with the effect size (Cohen's d) and its magnitude (**N**egligible, **S**mall, **M**edium, or **L**arge).

Project	Killed Mutants			$BOT_{BRANCHOS}$ vs Budget			BOT_{Ideal} vs Budget		
	BOT_{Ideal}	$BOT_{BRANCHOS}$	Budget [5]	t-test (p)	Cohen's d		t-test (p)	Cohen's d	
Netweaver	6,278.5	**6,331.8**	6,230.5	**0.005**	1.70	(L)	0.084	1.08	(L)
Squirrel SQL	19,474.3	**19,586.4**	18,937.3	**0.021**	1.46	(L)	**0.001**	2.40	(L)
SweetHome 3D	**4,040.0**	3,707.0	3,597.1	0.147	1.03	(L)	**<0.001**	5.68	(L)
Vuze	42,294.4	**43,229.1**	41,546.8	**0.002**	2.39	(L)	**0.047**	0.88	(L)
Freemind	**5,170.6**	5,017.5	4,793.6	**0.047**	0.91	(L)	**<0.001**	1.92	(L)
Checkstyle	**785.3**	761.7	751.9	0.694	0.32	(S)	0.114	0.98	(L)
Weka	36,421.5	**37,778.5**	37,651.4	0.444	0.27	(S)	**0.012**	-1.97	(L)
Liferay	83,307.3	**83,587.4**	81,883.8	**0.004**	2.00	(L)	**0.016**	1.47	(L)
Firebird	8,018.3	**8,084.8**	7,735.7	0.064	1.50	(L)	0.090	1.13	(L)

useful in practice. Future research should aim at improving the coverage prediction accuracy since this is crucial for achieving a higher coverage at project-level. To foster future studies in this field and to allow future researchers to tackle this problem without the cost of running a search-based test case generation tool, we release all the material we used to run the experiments, including the results of the 54,697 runs of EvoSuite in which we observed the coverage for every second of execution of the tool for 300 seconds (for a total of 15.2M data-points). Future researchers can use our datasets to device better prediction models and test BOT.

Our future work include the integration of BOT in EvoSuite to make it an easier-to-use tool for practitioners.

6 THREATS TO VALIDITY

Construct Validity. Threats to construct validity are mainly related to the measure of the coverage we consider in our study.

We run EvoSuite on each class only once. Since the technique we use is stochastic, the level of coverage measured in different runs may vary. Given the particularly time-consuming task, we had to balance the effort to spend in terms of (i) the search budget to assign to each class, (ii) the number of classes to test and (iii) the number of runs for each class. Using a large search budget, in our context, was a strict requirement needed to predict how the coverage increases over time and also to make sure that the BCE metric would not be penalized, since we expect it to correlate with the "maximum" coverage that can be reached by search-based testing algorithms on a given unit to test. We chose 300 seconds, which, in automated test case generation, is a quite high budget (*e.g.,* the default budget used by EvoSuite is 60 seconds).

Having clarified that saving time on the assigned budget was not an option, the two other factors influencing the execution time were the number of classes and the number of runs per class. Given a number of runs R for each class, the number of classes analyzable in a fixed amount decreases with the increase of R. For example, two runs would have required to halve the number of classes we considered in our study assuming a fixed amount of time available. The number of runs suggested in this context is 30 [1]. To estimate the difference among runs and to assess the possible bias introduced by our choice of maximizing the number of tested classes rather than the number of runs per class, we tried to run EvoSuite 10 times on the 10 largest classes in terms of number of branches from our dataset. As done in our study, we considered the coverage achieved given a search budget varying from 1 to 300 seconds at steps of 1 second. Given the same class

under test and the same level of budget (*e.g.,* 25 seconds), the maximum absolute variation we observed is 0.035. This means that two different runs on the same class and with the same budget achieved a difference on coverage of at most ~ 4%. When considering only the final coverage (*i.e.,* the one achieved after 300 seconds), the mean absolute variation is 0.002 and the maximum absolute variation is 0.016 (*i.e.,* meaning less than 2%).

Internal validity. Threats to internal validity concern internal factors of our study that could hinder its validity. The main threat of this category is the choice of the machine learning technique to use for our approach. We choose Random Forest [4], since it performed better than others we tested (linear regression, REPTree from Weka [14] and a Multilayer Perceptron [16]), without requiring an unreasonable amount of training time (about 40 minutes for each fold). However, this does not exclude that using other more expensive techniques, such as deep neural networks (DNN) or support vector machines (SVM), would result in better predictions.

Another problem related to the use of machine learning is the possibility of over-fitting. We limited this risk experimenting our techniques in a cross-project scenario, *i.e.,* with training data completely separated from test data. We performed statistical analysis (Wilcoxon test and effect size) to measure the difference in terms of error of the models we compared, to exclude that the differences in the prediction error between BRANCHOS and the baselines is achieved by chance.

It can be argued that using BOT requires time-intensive operations (*e.g.,* training of the classifier) that would make it not useful in practice. We release the training set we built for BRANCHOS so that it is possible to use our classifier out-of-the-box, without the need to perform further training. Since the optimization step requires some time, we removed such overhead from the project-level search budget to divide, in order to test the approach in a realistic scenario. Anyway, we found that the time needed to optimize the search budget is generally very low. Finally, we used Evosuite to compute the strong mutation score for the techniques we compared. For some of the combination class-budget we provided as input to the tool, it crashed with a `NullPointerException`. Therefore, we had to ignore such classes. This, however, could negatively impact the results. We computed the search budget used for the classes for which we have the result for each approach and we found that the baseline used, in total, a higher amount of search budget: $\text{BOT}_{\text{Ideal}}$ and $\text{BOT}_{\text{BRANCHOS}}$ used 0.14% (~ 1 hour and 7 minutes) and 0.10% (~ 47 minutes) less budget, respectively. This allows us to conclude that the actual improvement may be slightly higher than the one we reported, mostly in the cases in which classes with high budget were not considered, such as the case of Weka for $\text{BOT}_{\text{Ideal}}$ already discussed in the results.

External validity. Threats to external validity concern the generalization of our findings. To limit the risk of taking into account classes not representative enough of a typical software project, we answered $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$ by taking into account *all* the classes from the studied projects.

We did not pick a sample of the classes to have a larger dataset from which BRANCHOS could extract knowledge and to have a better assessment of what would be the performance of the compared approaches in a real scenario.

Another possible threat to the generalizability of our findings is that we used only a search-based test case generation approach (*i.e.,* MOSA [24]) implemented in one tool (EvoSuite). It is possible that for different techniques/tools (such as Randoop) the results would be different. Also, it is worth noting that BRANCHOS and BOT are mainly designed to work on search-based test case generation techniques: it may not be possible to use them on top of inherently different techniques, such as dynamic symbolic execution. Replication will be devoted to corroborate our findings.

7 CONCLUSION AND FUTURE WORK

We presented BOT, an approach to optimize the project-level branch coverage by adaptively distributing the global search budget defined at project level among the classes in the context of search-based test case generation. BOT uses a search algorithm and BRANCHOS, the first approach that predicts the branch coverage achieved by an automatic search-based test case generation approach on a given class with a given search budget (expressed as seconds). Such an approach uses machine learning to train a regressor using as features structural metrics and the assigned search budget.

We experimented both BRANCHOS and BOT, and we compared them to three baselines. The results indicated that (i) BRANCHOS is able to overcome its baseline in terms of coverage prediction error in time, (ii) using BOT with an ideal coverage prediction approach to optimize the search budget allocation it would be possible to improve the project-level branch coverage by 3.6% and the number of killed mutants by 3.3%, on average, and (iii) combining BRANCHOS and BOT allows to achieve a limited improvement in terms of branch coverage and a quite substantial improvement in terms of number of killed mutants.

Even if the results are encouraging, there is still much room for improvement, mostly in terms of coverage prediction. For this reason, we release our dataset to facilitate future research in this field. We will investigate the definition and inclusion in BRANCHOS of other structural metrics to improve its prediction power and the overall accuracy of BOT . Finally, a next step will be to integrate BOT in EvoSuite to build a fully budget-aware test case generation tool that can be simply used on whole projects. The BRANCHOS model built on the training set we used in this paper could be adopted as is, *i.e.,* without the necessity of gathering more data. In other words, this means that running BOT in practice would require virtually no effort by the developers.

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 1–10.

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2015), 507–525.

[3] David W Binkley and Keith Brian Gallagher. 1996. Program slicing. In *Advances in Computers*. Vol. 43. Elsevier, 1–50.

[4] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[5] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 55–66.

[6] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[7] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Academic press.

[8] Camila Faria de Castro, Decio de Souza Oliveira, and Marcelo Medeiros Eler. 2016. Identifying characteristics of Java methods that may influence branch coverage: An exploratory study on open source projects. In *Computer Science Society (SCCC), 2016 35th International Conference of the Chilean*. IEEE, 1–8.

[9] Javier Ferrer, Francisco Chicano, and Enrique Alba. 2013. Estimating software testing complexity. *Information and Software Technology* 55, 12 (2013), 2125–2139.

[10] 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.

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

[12] 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), 8.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. ACM, 213–223.

[14] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.
[15] Richard Hamlet. 1994. Random testing. *Encyclopedia of software Engineering* (1994).
[16] Simon Haykin and Neural Network. 2004. A comprehensive foundation. *Neural networks* 2, 2004 (2004), 41.
[17] Pavneet Singh Kochhar, Ferdian Thung, David Lo, and Julia Lawall. 2014. An empirical study on the adequacy of testing in open source projects. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1. IEEE, 215–222.
[18] Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2013. AUSTIN: An open source tool for search based software testing of C programs. *Information and Software Technology* 55, 1 (2013), 112–125.
[19] Phil McMinn. 2007. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. *Department of Computer Science, University of Sheffield, Tech. Rep. CS-07-14* (2007).
[20] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.
[21] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.
[22] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 75–84.
[23] Annibale Panichella, Fitsum 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* (2017).
[24] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 1–10.
[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, 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.
[27] Simone Scalabrino, Giovanni Grano, Dario Di Nucci, Rocco Oliveto, and Andrea De Lucia. 2016. Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach?. In *International Symposium on Search Based Software Engineering*. Springer, 64–79.
[28] Koushik Sen. 2007. Effective random testing of concurrent programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 323–332.
[29] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.
[30] Paolo Tonella. 2004. Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 119–128.