Coverage Goal Selector for Combining Multiple Criteria in Search-Based Unit Test Generation

Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang

Abstract—Unit testing is critical to the software development process, ensuring the correctness of basic programming units in a program (e.g., a method). Search-based software testing (SBST) is an automated approach to generating test cases. SBST generates test cases with genetic algorithms by specifying the coverage criterion (e.g., branch coverage). However, a good test suite must have different properties, which cannot be captured using an individual coverage criterion. Therefore, the state-of-the-art approach combines multiple criteria to generate test cases. Since combining multiple coverage criteria brings multiple objectives for optimization, it hurts the test suites' coverage for certain criteria compared with using the single criterion. To cope with this problem, we propose a novel approach named **smart selection**. Based on the coverage correlations among criteria and the subsumption relationships among coverage goals, smart selection selects a subset of coverage goals to reduce the number of optimization objectives and avoid missing any properties of all criteria. We conduct experiments to evaluate smart selection on 400 Java classes with three state-of-the-art genetic algorithms under the 2-minute budget. On average, smart selection outperforms combining all goals on 65.1% of the classes having significant differences between the two approaches. Secondly, we conduct experiments to verify our assumptions about coverage criteria relationships. Furthermore, we experiment with different budgets of 5, 8, and 10 minutes, confirming the advantage of smart selection over combining all goals.

Index Terms—SBST, software testing, test generation.

1 Introduction

Unit testing is a common way to ensure software quality by testing individual units or components of a software system in isolation from the rest of the system. Manually writing unit tests can be a tedious and error-prone process. Hence, developers and researchers put much effort into automatically generating test cases for programming units in recent years.

Search-based software testing (SBST) is considered a promising approach to generating test cases. It generates test cases with genetic algorithms (e.g., Whole Suite Generation (WS) [1], MOSA [2], DynaMOSA [3]) based on the coverage criterion (e.g., branch coverage). The execution of a genetic algorithm depends on fitness functions, which quantify the degree to which a solution (i.e., one or more test cases) achieves its goals (i.e., satisfying a certain coverage criterion). Each coverage criterion has a corresponding group of fitness functions, and each fitness function describes whether or how far a test case covers a specific coverage goal (e.g., a branch).

The Problem and Motivation. As claimed in [4], a good test suite must possess different properties that cannot be easily captured by any individual coverage criterion. Therefore, to generate a good test suite, multiple coverage criteria should be considered in SBST. Hence, the state-of-the-art approach [4] combines multiple coverage criteria to guide genetic algorithms. This method involves eight coverage criteria (see Sec. 2.1), which we call the **original combination** in this paper. However, combining multiple criteria leads to more objectives for optimization, which could impact the effectiveness of the genetic algorithms [2], [5], [6]. For example, it can increase the probability of being trapped in local optima. As a result, the coverage of the generated test suite decreases for certain criteria compared with using a single criterion. Fig. 1 shows (see Sec. 4.2) the average coverage gaps between the original combination and each constituent criterion when applying WS [1] into 85 large Java classes (i.e., with at least 200 branches). The average gap of the eight criteria is 8.2%, Branch coverage (BC) decreases by 10%, and Exception coverage (EC) decreases by 26%. Note that since

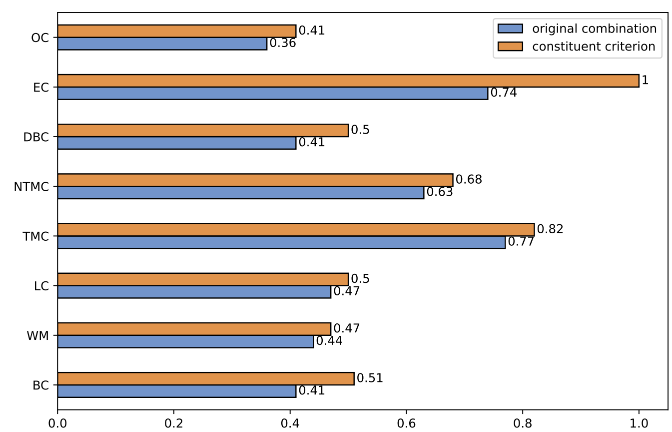


Fig. 1: Partial Data of Coverage Gaps

Z. Zhou and J. He are with School of Information Science and Technology, ShanghaiTech University, China
Y. Zhou, C. Fang, and Z. Chen are with State Key Laboratory for Novel Software Technology, Nanjing University, China
X. Luo is with the Hong Kong Polytechnic University, Hong Kong SAR, China
Y. Tang is with University of Glasgow, United Kingdom
Y. Tang (yutian.tang@glasgow.ac.uk) is the corresponding author.

the total exceptions in a class cannot be determined [4], we normalize the exception coverage values of two approaches (22.08 vs. 29.74) by dividing them by the larger one.

Targets. To address this problem, a competent approach should achieve the following targets: (1) **T1: GA Effectiveness.** It should select a subset of coverage goals from multiple coverage criteria. This subset should enhance the effectiveness of guiding genetic algorithms (GAs); and (2) **T2: Property Consistency.** This subset should prevent the omission of any properties captured by these coverage criteria.

Our Solution. To achieve these targets, we propose a novel approach named **smart selection** (see Sec. 3). In this paper, we have considered the eight coverage criteria mentioned above. However, instead of directly combining them, in smart selection, we first group them into four groups based on coverage correlations (see Sec. 3.2). Next, we select one representative criterion that is more effective in guiding the genetic algorithms from each group (T1) (see Sec. 3.3). These selected coverage criteria (SC)' coverage goals are denoted as $Goal(SC)$. To keep the property consistency (T2), for each criterion (c) of unselected criteria (USC), we select a subset $Goal(c)_{sub}$ from its coverage goals based on the goals' subsumption relationships (see Sec. 3.4). Finally, we combine $Goal(SC)$ and $\bigcup_{c \in USC} Goal(c)_{sub}$ to guide the test case generation process.

Contribution. In summary, the contribution of this paper includes:

• To the best of our knowledge, this is the *first* paper that uses coverage correlations to address the coverage decrease caused by combining multiple criteria in SBST.

• We implement smart selection atop EvoSuite. It is integrated into three search algorithms (i.e., WS, MOSA, and DynaMOSA).

• We conduct experiments on 400 Java classes to compare smart selection and the original combination with the 2-minute time budget. On average of three algorithms (WS in Sec. 4.2, MOSA in Sec. 4.3, and DynaMOSA in Sec. 4.4), smart selection outperforms the original combination on 77 (121/78/32) classes, accounting for 65.1% (85.8%/65%/44.4%) of the classes having significant differences between the two approaches. The counterpart data of the 85 large classes is 34 (50/35/16), accounting for 86.1% (98%/87.5%/72.7%). Second, we conduct experiments to compare smart selection with/without the subsumption strategy on 173 classes (Sec. 4.7).

Major Extensions. The article is the extended version of our previous paper [7], which was published at the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE). This article introduces the following extended contributions:

• We add the statistical information of our experimental subjects, including their distributions of the number of branches and lines (Sec. 4.1).

• In Sec. 3.2, we propose three rules to determine whether two criteria have a coverage correlation so that we can cluster criteria into several groups. To verify our rules, we conduct experiments in Sec. 4.5. The experimental results show that, on average, the Pearson Correlation Coefficient [8] of the coverage values of the criteria from the same groups is much higher than that of the criteria from the different groups (0.88 versus 0.41), confirming the effectiveness of our rules.

• In Sec. 3.3, we select one criterion that is more representative and effective in guiding the genetic algorithms from each group by analyzing and comparing their fitness functions. We conduct experiments in Sec. 4.6 to verify our selection by using two deliberate criteria combinations to guide GAs. The experimental results show that, in most cases, the coverage resulting from smart selection is higher than that of these two criteria combinations, confirming our choice.

• We investigate how smart selection performs under different search budgets (Sec. 4.8).

Skeleton. The rest of this paper is organized as follows: In Sec. 2, we introduce the background of SBST. Our methodology, smart selection, is illustrated in Sec. 3. Sec. 4 presents our evaluation of smart selection. We discuss the parameter tuning of smart selection and the threats to validity in Sec. 5. We present the related work in Sec. 6 and conclude this paper in Sec. 7.

Online Artifact. The online artifact of this paper can be found at: https://doi.org/10.5281/zenodo.7601316.

2 BACKGROUND

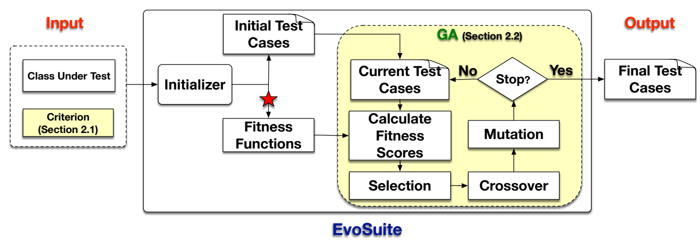


Fig. 2: Overview of Unit Tests Generation in EvoSuite

SBST and EvoSuite SBST generates test cases using a genetic algorithm (see Sec. 2.2) by specifying the coverage criterion (see Sec. 2.1). EvoSuite [9] is a state-of-art SBST tool for Java. In this section, we use EvoSuite as an example to illustrate the key idea of SBST. Fig. 2 shows an overview of EvoSuite. The red star in this figure is mentioned later in Sec. 3.1.

Input. Evosuite takes two major inputs: (1) the class under test (CUT) and (2) a coverage criterion (Sec. 2.1).

Test Generation. The test generation process consists of two main stages: (1) The initializer extracts all the necessary information required by the genetic algorithm, such as method signatures (including names and parameter types), from the CUT. Based on this information and the coverage criterion, the initializer generates initial test cases and fitness functions. Typically, each GA requires one or more specific fitness functions. A fitness function evaluates how close a test case covers a coverage goal (e.g., a branch); (2) After a specific genetic algorithm is invoked, it selects test cases based on the scores returned by fitness functions. The GA then creates new test cases using the crossover and mutation operations [1]. This process of selecting, mutating, and crossing over test cases continues until all fitness functions reach the optima or the given budget is exhausted.

Output. After running the genetic algorithm, EvoSuite outputs the final test cases.

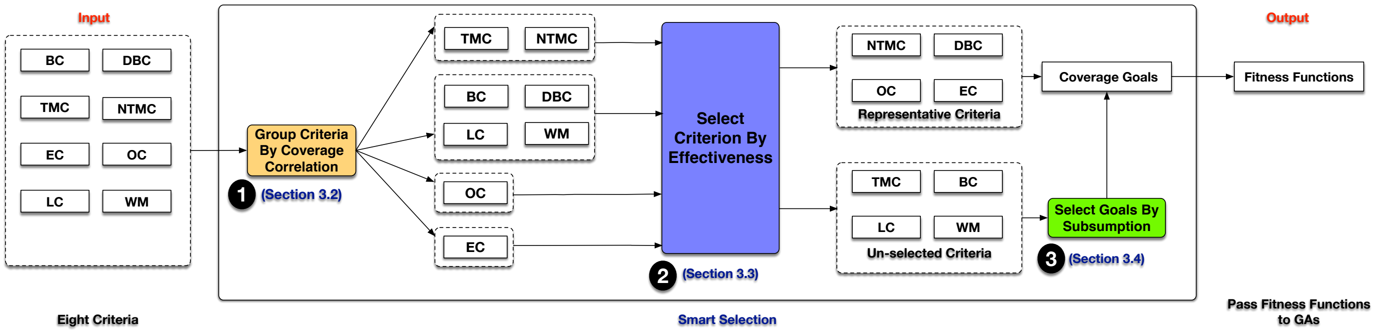


Fig. 3: Overview of **Smart Selection**

2.1 Coverage Criteria

In this research, we discuss eight criteria as follows. The reason to choose these coverage criteria is that they are EvoSuite's default criteria and have been widely used in many previous studies [4], [10], [11].

Branch Coverage (BC) BC checks the number of branches of conditional statements covered by a test suite.

Direct Branch Coverage (DBC) The only difference between DBC and BC is that a test case must directly invoke a public method to cover its branches. DBC treats others just as BC [4].

Line Coverage (LC) LC checks the number of lines covered by a test suite.

Weak Mutation (WM) WM checks how many mutants are detected by a test suite [12], [13]. A mutant is a variant of the CUT generated by a mutation operator. For example, RC is an operator that replaces a constant value with different values [14].

Top-Level Method Coverage (TMC) TMC checks the number of methods covered by a test suite with a requirement: A public method is covered only when it is directly invoked by test cases.

No-Exception Top-Level Method Coverage (NTMC) NTMC is TMC with an extra requirement: A method is not covered if it exits with any exceptions.

Exception Coverage (EC) EC checks the number of exceptions triggered by a test suite.

Output Coverage (OC) OC measures the diversity of the return values of a method. For example, a *boolean* return variable's value can be *true* or *false*. OC's coverage is 100% only if the test suite captures these two return values.

For each criterion, EvoSuite extracts a group of coverage goals from the CUT and assigns a fitness function to each coverage goal. For example, EvoSuite extracts all branches for BC, e.g., the true branch of a predicate $x == 10$. A simplified fitness function of this branch can be the branch distance [15], $|x-10|$, which measures the distance between the actual value and the expected value of x in the test case. This distance value represents how far a test case covers the branch. More details of these criteria and their fitness functions can be found in [4].

2.2 Genetic Algorithms

In this research, we discuss three GAs (i.e., WS, MOSA, DynaMOSA) as follows. All of them are integrated into EvoSuite and perform well in many SBST competitions [16], [17], [18]. These algorithms share the same inputs and outputs but differ in how to use fitness functions.

WS. WS [1] directly evolves test suites to fit all coverage goals. Consequently, WS can exploit the collateral coverage [19] and not waste time on infeasible goals (e.g., dead code). Collateral coverage means that a test case generated for one goal can implicitly also cover any number of other coverage goals. Hence, WS's fitness function is the sum of all goals' fitness functions.

MOSA. WS sums fitness scores of all coverage goals as a scalar value. However, this scalar value is less monotonic and continuous than a single goal's fitness score, which increases the probability of being trapped in local optima. To overcome this limit, Panichella et al. [2] formulates SBST as a many-objective optimization problem and propose MOSA, a variant of NSGA-II [20]. In general, MOSA maintains a fitness vector for each test case, where each item in the fitness vector represents a fitness function value for the test case. Based on Pareto dominance [21], MOSA sorts and selects test cases by the fitness vectors.

DynaMOSA. Based on MOSA, DynaMOSA [3] adopts control dependency graph to reduce the coverage goals evolved in search. A goal is selected to be part of the evolving process only when the branch goals it depends on are already covered. Hence, DynaMOSA's fitness vectors are often smaller than those of MOSA. Empirical studies have shown that DynaMOSA outperforms WS and MOSA [3], [11].

2.3 Combining Coverage Criteria

With a single criterion (e.g., BC) alone, SBST can generate test cases that reach higher code coverage but fail to meet users' expectations [4]. Therefore, Rojas et al. [4] proposed combining multiple criteria to guide SBST to generate a test suite. We use the example of replacing BC with the combination of eight criteria to demonstrate the changes in GAs. Before the combination, the fitness function of WS is $f_{BC} = \sum_{b \in B} f_b$, where B is the set of all branches. The fitness vector of MOSA/DynaMOSA is $[f_{b_1}, ..., f_{b_n}]$. After the combination, the fitness function of WS is $f_{BC} + ... + f_{OC}$. The fitness vector of MOSA/DynaMOSA is $[f_{b_1}, ..., f_{b_n}, ..., f_{o_1}, ..., f_{o_m}]$, where o_i is a output coverage goal.

3 Smart Selection

The Problem and Motivation: The main side-effect of combining multiple criteria is that the generated test suite's coverage decreases for certain criteria. This is due to the increase in optimization objectives, which results in a *larger* search space and reduces the search weight of each objective. Moreover, some criteria have fitness functions that are not monotonic and continuous, such as LC and WM (see Sec. 3.3), which makes the search space *harder*. Therefore, we propose smart selection to relieve the coverage decrease by providing a smaller and easier search space for GAs.

3.1 Overview

Fig. 3 shows the process: ❶ group criteria by coverage correlation (Sec. 3.2), ❷ select representative criteria by effectiveness to guide SBST (Sec. 3.3), and ❸ select representative coverage goals from unselected criteria by subsumption relationships (Sec. 3.4). The red star in Fig. 2 shows the position of smart selection in EvoSuite.

The inputs are the eight criteria (see Sec. 2.1). The output is a subset of fitness functions, i.e., the corresponding fitness functions of our selected coverage goals. This subset is used to guide GAs.

3.2 Grouping Criteria by Coverage Correlation

The first step is clustering these eight criteria. The standard of whether two criteria can be in one group or not is: whether these two criteria have a coverage correlation. Based on this standard, we can divide these criteria into several groups. For two criteria with coverage correlation, if a test suite achieves a high coverage under one of the criteria, then this test suite may also achieve a high coverage under the other criterion. Hence, we can select one of these two criteria to guide SBST. Thus, grouping sets the scope for choosing representative criteria.

We determine that two criteria have a coverage correlation by the following rules:

•**Rule 1:** If a previous study shows that two criteria have a coverage correlation, we adopt the conclusion:

① **BC and WM:** Gligoric et al. [22] find that "branch coverage performs as well as or better than all other criteria studied, in terms of ability to predict mutation scores". Their work shows that the average Kendall's τ_b value [23] of coverage between branch coverage and mutation testing is 0.757. Hence, we assume that BC and WM have a coverage correlation.

② **DBC and WM:** Since BC and WM have a coverage correlation, we assume that DBC and WM have a coverage correlation too.

③ **LC and WM:** Gligoric et al. [22] find that statement coverage [4], [22] can be used to predict mutation scores too. Line coverage is an alternative for statement coverage in Java since Java's bytecode instructions may not directly map to source code statements [4]. Hence, we assume that LC and WM have a coverage correlation.

•**Rule 2:** Two criteria, A and B, have a coverage correlation if they satisfy two conditions: (1) A and B have the same coverage goals; (2) A test covers a goal of A only if it covers the counterpart goal of B and it satisfies A's additional requirements:

④ **DBC and BC:** DBC is BC with an additional requirement (see Sec. 2.1).

⑤ **NTMC and TMC:** NTMC is TMC with an additional requirement (see Sec. 2.1).

•**Rule 3:** Two criteria, A and B, have a coverage correlation if, for an arbitrary test suite, the relationship between these two criteria' coverage (i.e., C_A and C_B) can be formulated as:

$$C_B = \Theta C_A, \tag{1}$$

where Θ is a nonnegative random variable and $E\Theta \approx 1$:

⑥ **BC and LC:** Intuitively, when a branch is covered, then all lines in that branch are covered. But this is not always true. When a line exits abnormally (e.g., it throws an exception.), the subsequent lines are not covered either. First, we discuss the coverage correlation of branch and line coverage in the absence of abnormal exiting. Let B be the set of branches of the CUT, L be the set of lines, and T be a test suite. For any $b \in B$, let L_b be the set of lines **only** in the branch b (i.e., we don't count the lines in its nested branches). Consequently, $L = \bigcup_{b\in B} L_b$. Let $B^{'}$ be the set of covered branches. Let $L^{'}$ be the set of covered lines. The coverage values measured by branch and line coverage are:

$$C_{Branch} = \frac{|B^{'}|}{|B|}, C_{Line} = \frac{|L^{'}|}{|L|} = \frac{\sum\limits_{b\in B'} |L_b|}{\sum\limits_{b\in B} |L_b|}. \tag{2}$$

Hence, the relationship of C_{Branch} and C_{Line} is:

$$\frac{C_{Line}}{C_{Branch}} = \frac{\sum\limits_{b\in B'} |L_b|}{\sum\limits_{b\in B} |L_b|} \div \frac{|B^{'}|}{|B|} = \frac{\sum\limits_{b\in B'} |L_b|}{|B'|} \div \frac{\sum\limits_{b\in B} |L_b|}{|B|}. \tag{3}$$

Suppose we treat branches with different numbers of lines equally in generating T. Then we have:

$$\frac{\sum\limits_{b\in B'} |L_b|}{|B'|} \approx \frac{\sum\limits_{b\in B} |L_b|}{|B|}, \tag{4}$$

i.e.,

$$\frac{C_{Line}}{C_{Branch}} \approx 1. \tag{5}$$

As a result, branch coverage and line coverage have a coverage correlation in the absence of abnormal exiting. With abnormal exiting, the coverage measured by line coverage decreases. Assuming that any line can exit abnormally, we can formulate the coverage relationship as:

$$C_{Line} = \Theta C_{Branch}, \tag{6}$$

where Θ is a random variable. In this research, instead of analyzing Θ precisely, we only need to check whether $E\Theta \approx 1$. Previous work [4] shows that, on average, when 78% of branches are covered, test suites can only find 1.75 exceptions. Hence, we assume that $E\Theta \approx 1$, i.e., BC and LC have a coverage correlation.

⑦ **DBC and LC:** We assume that DBC and LC have a coverage correlation since BC and LC have a coverage correlation.

Output. We cluster the eight criteria into four groups: (1) BC, DBC, LC, and WM; (2) TMC and NTMC; (3) EC; and (4) OC.

3.3 Selecting Representative Criterion by Effectiveness to guide SBST

In this step, among the criteria in each group, we select a criterion to represent the others. The criteria within a group differ in the ability to guide SBST. If we only select one criterion with the best effectiveness to guide SBST, SBST will be more efficient in generating unit tests. To select the best criterion to guide SBST in each group, we need to compare the effectiveness of the criteria in guiding SBST. A criterion's effectiveness in guiding SBST largely depends on the continuity of monotonicity of its fitness functions [24], [25]. Hence, we need to analyze and compare the fitness functions of the criteria.

Group1: BC, DBC, LC, and WM. We use branch coverage as the baseline and divide them into three pairs for discussion. The reason to use branch coverage as the baseline is that branch coverage has been widely used to guide unit test generation [1], [2], [3] due to the monotonic continuity of its fitness functions. For a branch goal b and a test case t, its fitness function is [1]:

$$f_{bc}(b,t) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d(b,t)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise,} \end{cases} \tag{7}$$

where $\nu(x)$ is a normalizing function in $[0,1]$ (e.g., $\nu(x) = x/(x+1)$). $d(b,t)$ is a function to provide a branch distance to describe how far a test case covers this goal [15]. To avoid an oscillate situation of a predicate [1], $f_{bc}(b,t)$ uses $\nu(d(b,t))$ only when a predicate is executed at least twice. Note that the final fitness function used by many previous studies [3], [16] is f_{bc} plus the approach level (AL), the number of control dependencies from a test's execution trace to the target [3]. We omit AL in comparing these criteria' fitness functions since its usage is the same in different criteria.

WS uses the sum of all fitness functions as one fitness function (Sec. 2.2). Hence, for WS, branch coverage's fitness function is:

$$d_1(b,T) = \min\{f_{bc}(b,t) | t \in T\}, \tag{8}$$

$$f_{BC}(T) = \sum_{b \in B} d_1(b,T), \tag{9}$$

where B denotes all branches of the CUT.

•BC vs. LC. Based on line coverage's definition (see Sec. 2.1), a line l's fitness function can be:

$$f_{lc}(l,t) = \begin{cases} 0 & \text{if the line has been covered,} \\ 1 & \text{otherwise.} \end{cases} \tag{10}$$

For WS, line coverage's fitness function is:

$$f_{LC}(T) = \nu(|L| - |CL|), \tag{11}$$

where L is the set of all lines and CL is the set of covered lines.

These two fitness functions are not continuous and monotonic since they only tell whether the lines are covered. To overcome this limit, EvoSuite uses branch coverage's fitness functions to augment line coverage's fitness functions [4]. A line l's fitness function is:

$$f_{lc}(l,B,t) = \begin{cases} 0 & \text{if the line has been covered,} \\ 1 + \min\{f_{bc}(b,t) | b \in B\} & \text{otherwise,} \end{cases} \tag{12}$$

where B is the set of branches that l depends on [3]. For WS, line coverage's fitness function is:

$$f_{LC}(T) = \nu(|L| - |CL|) + f_{BC}(T). \tag{13}$$

We call Equation 10 and 11 *def-based* (definition-based) fitness functions and call Equation 12 and 13 *augmented* fitness functions.

Firstly, we compare branch coverage's fitness functions with line coverage's *def-based* fitness functions. Line coverage's *def-based* fitness functions are not continuous and monotonic since they only tell whether the lines are covered. Therefore, branch coverage is more effective than line coverage in guiding SBST when we use line coverage's *def-based* fitness functions. After the augmentation, line coverage's *def-based* fitness functions disturb the continuity and monotonicity of branch coverage's fitness functions, undermining branch coverage's effectiveness to guide SBST. As a result, BC is better than LC in the effectiveness to guide SBST.

•BC vs. WM. Based on weak mutation's definition (see Sec. 2.1), a mutant's fitness function is:

$$f_{wm}(\mu,t) = \begin{cases} 1 & \text{if mutant } \mu \text{ was not reached,} \\ \nu(id(\mu,t)) & \text{if mutant } \mu \text{ was reached,} \end{cases} \tag{14}$$

where $id(\mu,t)$ is the infection distance function. It describes how distantly a test case triggers a mutant's different state from the source code. Different mutation operators have different infection distance functions [14]. A mutant's fitness function is always 1 unless a test case reaches it (i.e., the mutated line is covered). Hence, like line coverage, EvoSuite uses the same way to augment weak mutation's fitness functions [3], [14]. As the conclusion of comparing BC and LC, BC is better than WM in the effectiveness to guide SBST.

•BC vs. DBC. Direct branch coverage (DBC) is branch coverage with an extra requirement: A test case must directly invoke a public method to cover its branches. Based on branch coverage's fitness function, we can obtain DBC's one: For a branch in a public method, when the method is not invoked directly, the fitness function always returns 1. Otherwise, the fitness function is the same as branch coverage's one. It is easy for SBST to generate a test case that invokes a public method directly. Hence, we consider BC and DBC to be nearly equal in guiding SBST.

Order of Group1. Above all, we get a rough order of this group: (1)BC and DBC; (2) LC and WM. Since we only need one representative, the rough order satisfies our need.

The Representative Criterion of Group1. We choose DBC to represent this group instead of BC. The reason is: When a test case covers a goal of DBC, the test case covers the counterpart of BC. As a result, DBC can fully represent BC. The opposite may not hold.

Group2: TMC and NTMC. Like the relationship between branch coverage and direct branch coverage, no-exce. top-level method coverage is top-level method coverage with an extra requirement: A method must be invoked without triggering exceptions.

The Representative Criterion of Group2. We choose NTMC to represent this group. The reason is the same as why we choose DBC to represent group 1: NTMC can fully represent TMC. The opposite does not hold.

Group3: EC and Group 4: OC. Since group 3 only contains EC, we choose EC to represent group 3. Similarly, we choose OC to represent group 4.

Output. The representative criteria are DBC, NTMC, EC, and OC.

3.4 Selecting Representative Coverage Goals by Subsumption Relationships

After selecting the representative criteria in the previous step, there are four unselected criteria: LC, WM, BC, and TMC. To keep property consistency for each unselected criterion, we select a subset from its coverage goals. This subset can represent all properties required by this criterion, ensuring GA archives [3] those tests that fulfill the properties beyond the representative criteria. We have another requirement for these subsets: they should be as small as possible. These unselected criteria' fitness functions are less continuous and monotonic than the ones of the representative criteria (see Sec. 3.3). Therefore, to minimize the negative effects on guiding SBST, these subsets should be as small as possible.

Two coverage goals, G_1 and G_2, having the subsumption relationship denotes that if a test suite covers one coverage goal, it must cover another goal. Specifically, G_1 subsuming G_2 represents that if a test suite covers G_1, it must cover G_2. According to this definition, for a criterion, if the coverage goals not subsumed by others are covered, all coverage goals are covered. Hence, These coverage goals form the desired subset.

LC. For the lines in a basic block, the last line subsumes others. Hence, these last lines of all basic blocks form the desired subset. Since Sec. 3.2 shows that BC/DBC and LC have a strong coverage correlation and DBC is the representative criterion, we do a tradeoff to shrink this subset: We add an integer parameter *lineThreshold*. If a basic block's lines are less than *lineThreshold*, we skip it. In this research, we set *lineThreshold* as 8 (Sec. 5.1 discusses it).

WM. The process to extract the subset from weak mutation's all mutants can be divided into three parts: ① We select the key operators from EvoSuite's all implemented mutation operators; ② From the key operators we filter out the **equal-to-line** operators; ③ For the remaining operators, we select the subsuming mutants by following the previous work [26].

① **Select Key Operators.** Offutt et al. [27] find that five key operators achieve 99.5% mutation score. They are UOI, AOR, ROR, ABS, and LCR. EvoSuite does not implement LCR (an operator that replaces the logical connectors) and ABS (an operator that inserts absolute values) [14]. Hence, we select three operators: UOI, AOR, and ROR (see Table 1).

TABLE 1: **Evosuite**'s Partial Mutation Operators

Operator	Usage
UOI	Insert unary operator
AOR	Replace an arithmetic operator
ROR	Replace a comparison operator

② **Filter out Equal-To-Line Operators.** For each mutation operator, EvoSuite designs an infection distance function to describe how far a mutant's different state from the source code is triggered [14]. Some infection distance functions always return 0. For example, UOI only adds 1 to, subtracts 1 to, or negates a numerical value, so the infection distance is always 0. Therefore, if a test case covers the line mutated by UOI, the mutant is killed. We refer to this type of operator an **equal-to-line** operator. Among three key operators, only UOI is an equal-to-line operator [14]. Since line coverage has been dealt with, we filter out it.

③ **Select Subsuming Mutants.** The remaining operators are AOR and ROR. We choose one of the existing approaches [26], [28], [29], [30], [31] to select subsuming mutants for them. These approaches can be classified into three categories: (1) Manual analysis: Just et al. [30] establish the subsumption relationships for ROR and LCR by analyzing all possible outputs of their mutants. However, this approach can not be applied to non-logical operators [31]; (2) Dynamic analysis: Guimarães et al. [31] establish the subsumption relationships by running an exhaustive set of tests. This approach requires many tests, which we can not provide; (3) Static analysis: Gheyi and Souza et al. [26], [32] encode a theory of subsumption relations in the Z3 theorem prover to identify the subsumption relationships. We adopt this approach because (i) This approach can be applied to both AOR and ROR; (ii) Using the Z3 prover to identify the subsumption relationships is a once-for-all job. We can hardcode their results into EvoSuite.

BC and TMC. For a coverage goal of branch coverage, there is a subsuming goal from direct branch coverage (see Sec. 3.2). As a result, the subset for branch coverage is empty since we select direct branch coverage as the representative (see Sec. 3.3). Similarly, the subset for top-level method coverage is empty too.

Output. For four unselected criteria, we select four subsets of their coverage goals. Two of them are empty. Finally, smart selection joins these subsets with the representative criteria to get their fitness functions for guiding GAs.

4 Evaluation

4.1 Experiment Setting

The evaluation focuses on the performance of smart selection. Our evaluation aims to answer the following research questions:

- **RQ1:** How does smart selection perform with WS?
- **RQ2:** How does smart selection perform with MOSA?
- **RQ3:** How does smart selection perform with DynaMOSA?
- **RQ4:** Do criteria within the same criteria group exhibit a coverage correlation?
- **RQ5:** Are representative criteria efficient in guiding SBST?
- **RQ6:** How does the subsumption strategy affect the performance of smart selection?
- **RQ7:** How does smart selection perform under different search budgets?

TABLE 2: Overview of Java projects and classes in our evaluation

Project	Package	Classes	Branches			Lines		
			25%	mean	75%	25%	mean	75%
a4j	-	**1**	**125**	**125**	**125**	**2,038**	**2,038**	**2,038**
apbsmem	-	**1**	**390**	**390**	**390**	**4,323**	**4,323**	**4,323**
at-robots2-j	-	**1**	**125**	**125**	**125**	**832**	**832**	**832**
battlecry	-	**2**	**128**	**179**	**230**	**1,767**	**1,947**	**2,126**
biff	-	**1**	**817**	**817**	**817**	**7,012**	**7,012**	**7,012**
caloriecount	-	**1**	**232**	**232**	**232**	**1,425**	**1,425**	**1,425**
celwars2009	-	**1**	**360**	**360**	**360**	**2,139**	**2,139**	**2,139**
checkstyle	-	**1**	**50**	**50**	**50**	**412**	**412**	**412**
classviewer	-	**3**	**114**	**154**	**202**	**671**	**1,219**	**1,665**
commons-cli	-	**2**	**139**	**146**	**152**	**696**	**788**	**880**
commons-codec	-	**1**	**504**	**504**	**504**	**2,385**	**2,385**	**2,385**
commons-collections	-	**2**	**122**	**155**	**188**	**351**	**426**	**501**
commons-lang	-	**12**	**137**	**353**	**370**	**868**	**1,759**	**2,037**
commons-math	-	**17**	**75**	**120**	**149**	**630**	**1,105**	**1,350**
corina	-	**1**	**55**	**55**	**55**	**282**	**282**	**282**
dcparseargs	-	**1**	**80**	**80**	**80**	**571**	**571**	**571**
dsachat	-	**2**	**76**	**83**	**90**	**868**	**880**	**892**
dvd-homevideo	-	**2**	**60**	**68**	**76**	**1,269**	**1,308**	**1,347**
feudalismgame	-	**1**	**788**	**788**	**788**	**4,873**	**4,873**	**4,873**
fim1	-	**1**	**73**	**73**	**73**	**1,140**	**1,140**	**1,140**
firebird	-	**2**	**123**	**147**	**171**	**666**	**751**	**836**
fixsuite	-	**1**	**74**	**74**	**74**	**631**	**631**	**631**
fps370	-	**1**	**70**	**70**	**70**	**668**	**668**	**668**
freemind	-	**1**	**208**	**208**	**208**	**1,621**	**1,621**	**1,621**
gfarcegestionfa	-	**2**	**87**	**99**	**111**	**664**	**826**	**988**
glengineer	-	**1**	**115**	**115**	**115**	**658**	**658**	**658**
guava	-	**8**	**84**	**143**	**160**	**213**	**490**	**742**
httpanalyzer	-	**1**	**56**	**56**	**56**	**656**	**656**	**656**
ifx-framework	-	**1**	**72**	**72**	**72**	**462**	**462**	**462**
inspirento	-	**1**	**95**	**95**	**95**	**649**	**649**	**649**
io-project	-	**1**	**66**	**66**	**66**	**282**	**282**	**282**
ipcalculator	-	**2**	**67**	**79**	**91**	**1,164**	**1,252**	**1,340**
javaml	-	**3**	**51**	**52**	**54**	**390**	**409**	**424**
javathena	-	**1**	**255**	**255**	**255**	**2,278**	**2,278**	**2,278**
javaviewcontrol	-	**2**	**760**	**1,298**	**1,835**	**3,078**	**4,394**	**5,710**
jclo	-	**1**	**133**	**133**	**133**	**987**	**987**	**987**
jcvi-javacommon	-	**1**	**61**	**61**	**61**	**240**	**240**	**240**
jdom	-	**5**	**63**	**110**	**91**	**510**	**591**	**583**
jfreechart	-	**8**	**102**	**263**	**242**	**754**	**1,881**	**1,843**
jiggler	-	**3**	**65**	**82**	**96**	**465**	**711**	**836**
jipa	-	**1**	**134**	**134**	**134**	**783**	**783**	**783**
jiprof	-	**4**	**84**	**451**	**818**	**628**	**2,612**	**4,344**
jmca	-	**3**	**207**	**707**	**961**	**1,122**	**2,307**	**2,978**
jopenchart	-	**1**	**92**	**92**	**92**	**1,139**	**1,139**	**1,139**
hadoop	*org*	**45**	**70**	**151**	**156**	**121**	**271**	**313**
hadoop	*org.apache*	**65**	**70**	**343**	**158**	**105**	**565**	**295**
hadoop	*org.apache.hadoop*	**40**	**62**	**220**	**194**	**133**	**403**	**331**
hadoop	*org.apache.hadoop.crypto*	**1**	**82**	**82**	**82**	**337**	**337**	**337**
hadoop	*org.apache.hadoop.mapreduce*	**3**	**78**	**90**	**101**	**184**	**204**	**218**
hadoop	*org.apache.hadoop.mapreduce.v2.api*	**3**	**80**	**91**	**110**	**136**	**160**	**196**
hadoop	*org.apache.hadoop.security*	**1**	**74**	**74**	**74**	**159**	**159**	**159**
hadoop	*org.apache.hadoop.thirdparty.com*	**18**	**71**	**108**	**122**	**108**	**154**	**174**
hadoop	*org.apache.hadoop.thirdparty.com.google*	**4**	**74**	**114**	**127**	**141**	**196**	**239**
hadoop	*org.apache.hadoop.thirdparty.org*	**1**	**119**	**119**	**119**	**163**	**163**	**163**
hadoop	*org.apache.hadoop.yarn*	**22**	**59**	**103**	**117**	**99**	**186**	**208**
hadoop	*org.apache.hadoop.yarn.api*	**14**	**56**	**107**	**134**	**106**	**183**	**212**
hadoop	*org.apache.hadoop.yarn.server*	**11**	**55**	**105**	**101**	**98**	**193**	**191**
hadoop	*org.apache.hadoop.yarn.server.api*	**4**	**95**	**117**	**138**	**176**	**208**	**242**
hadoop	*org.apache.hadoop.yarn.server.applicationhistoryservice*	**1**	**55**	**55**	**55**	**95**	**95**	**95**
hadoop	*org.apache.hadoop.yarn.server.federation.store*	**1**	**69**	**69**	**69**	**109**	**109**	**109**
hadoop	*org.apache.hadoop.yarn.server.nodemanager*	**2**	**84**	**91**	**98**	**170**	**187**	**204**
hadoop	*org.apache.hadoop.yarn.server.nodemanager.containermanager*	**4**	**59**	**60**	**63**	**99**	**112**	**131**

TABLE 2: Continued

Project	Package	Classes	Branch			Statement		
			25%	mean	75%	25%	mean	75%
hadoop	*org.apache.hadoop.yarn.server.resourcemanager*	**2**	**106**	**127**	**147**	**246**	**281**	**315**
jsecurity	-	**1**	**170**	**170**	**170**	**725**	**725**	**725**
lagoon	-	**2**	**63**	**64**	**64**	**758**	**841**	**924**
lhamacaw	-	**1**	**70**	**70**	**70**	**714**	**714**	**714**
liferay	-	**1**	**78**	**78**	**78**	**462**	**462**	**462**
lilith	-	**1**	**134**	**134**	**134**	**496**	**496**	**496**
newzgrabber	-	**2**	**125**	**173**	**220**	**1,110**	**1,257**	**1,404**
noen	-	**2**	**68**	**69**	**70**	**386**	**456**	**526**
objectexplorer	-	**1**	**175**	**175**	**175**	**987**	**987**	**987**
openhre	-	**1**	**50**	**50**	**50**	**220**	**220**	**220**
quickserver	-	**2**	**70**	**71**	**72**	**553**	**618**	**683**
resources4j	-	**1**	**176**	**176**	**176**	**1,539**	**1,539**	**1,539**
saxpath	-	**2**	**162**	**269**	**376**	**344**	**574**	**804**
schemaspy	-	**1**	**380**	**380**	**380**	**2,241**	**2,241**	**2,241**
shop	-	**3**	**100**	**131**	**153**	**833**	**998**	**1,145**
sugar	-	**1**	**51**	**51**	**51**	**484**	**484**	**484**
summa	-	**1**	**372**	**372**	**372**	**1,906**	**1,906**	**1,906**
sweethome3d	-	**2**	**271**	**387**	**503**	**1,544**	**2,152**	**2,760**
trove	-	**9**	**77**	**137**	**255**	**378**	**652**	**852**
twfbplayer	-	**2**	**94**	**115**	**135**	**631**	**839**	**1,047**
twitter4j	-	**6**	**64**	**117**	**115**	**399**	**1,493**	**823**
vuze	-	**1**	**134**	**134**	**134**	**675**	**675**	**675**
weka	-	**3**	**257**	**441**	**538**	**1,505**	**3,606**	**4,777**
wheelwebtool	-	**3**	**116**	**349**	**495**	**1,031**	**2,002**	**2,817**
Overall	-	**400**	**67**	**201**	**171**	**144**	**725**	**749**

Environment. All experiments are conducted on three machines with Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz and 128 GB RAM.

Subjects. We randomly select Java classes from 2 sources: the benchmark of DynaMOSA [3] and Hadoop [33]. Following the previous work [2], the only restriction of randomly selecting classes is that the class must contain at least 50 branches, aiming to filter out the trivial classes. As a result, we select 400 classes: 158 from the benchmark of DynaMOSA and 242 from Hadoop. Table 2 shows the statistical data of these classes' branches and lines grouped by the projects. The second column, Package, is only for the Hadoop project: Since Hadoop contains too many classes (more than half of all), we individually present its Java Packages' statistical data, not the whole project; The third column, Classes, presents the class number of this group (a Java project or a Java Package); The fourth column, Branches, contains three sub-columns, showing the 25th percentile, mean, and 75th percentile of the branches of this group's classes. The last column is similar to the fourth column but shows the counterpart data of lines.

Baseline for RQ1-3. We have two baselines: (1) the original combination, used to be compared with smart selection on each Java class; (2) a single constituent criterion, used to show the data of coverage decrease caused by the above two combination approaches. A single constituent criterion means that we only use each criterion of these eight criteria (see Sec. 2.1) to guide GAs. There is one exception: when the constituent criterion is exception or output coverage, we combine this criterion and branch coverage to guide GAs. The reason is that only exception or output coverage is weak in the effectiveness of guiding the GAs [4], [10]. Branch coverage can guide the GAs to reach more source lines of the CUT, increasing the possibility of triggering exceptions or covering output goals.

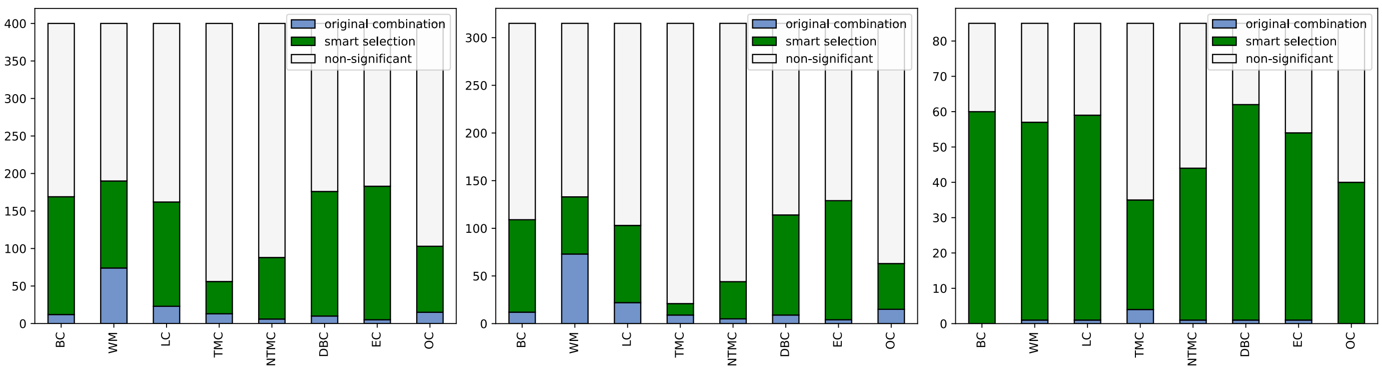
Configuration for RQ1-3. EvoSuite provides many parameters (e.g., crossover probability, population size [1]) to run the algorithms. In this paper, we adopt EvoSuite's default parameters to run smart selection and other baselines.

Smart selection introduces a new parameter *lineThreshold* (see Sec. 3.4). It controls smart selection to skip basic blocks with less than *lineThreshold* lines. We set *lineThreshold* as 8. The discussion on this value is in Sec. 5.1. For each Java class, we run EvoSuite with ten approaches: (1) smart selection, (2) the original combination, and (3) each constituent criterion of all eight criteria. We run each approach for 30 rounds per Java class, and each run's search budget is 2 minutes.

4.2 RQ1: How does smart selection perform with WS?

Motivation. In this RQ, we evaluate smart selection (**SS**) with WS. First, we compare the performance of SS and the original combination (**OC**). Next, we use the coverage of each constituent criterion (**CC**) to show the coverage decrease caused by SS and OC. Furthermore, we show these approaches' differences in the resulted suite sizes (i.e., the number of tests in a test suite).

Methodology. EvoSuite records the coverage for generated unit tests. For each class, we obtain 10 coverage data sets: One data set records the coverage of the eight criteria when using SS; One data set records the coverage of the



(a) All Classes (b) Small Classes (c) Large Classes

Fig. 4: Significant case summary of smart selection and the original combination with WS

eight criteria when using OC; The rest data sets record the coverage when using each CC.

For each Java class, we follow previous research work [4] to use Mann-Whitney U Test to measure the statistical difference between SS and OC. Then, we use the Vargha-Delaney $\hat{A}_{ab}$ [34] to evaluate whether a particular approach a outperforms another approach b ($\hat{A}_{ab} > 0.5$ and the significant value p is smaller than 0.05).

Result. ►All Classes.

TABLE 3: Average coverage results for each approach with WS

(a) All Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	55%	53%	57%	NTMC	71%	70%	71%
WM	59%	57%	59%	DBC	55%	53%	56%
LC	60%	58%	60%	EC	15.92	14.52	16.52
TMC	84%	83%	84%	OC	44%	43%	45%

(b) Small Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	58%	57%	59%	NTMC	73%	72%	72%
WM	62%	61%	62%	DBC	57%	56%	58%
LC	62%	61%	62%	EC	13.29	12.48	12.95
TMC	85%	85%	85%	OC	46%	45%	46%

(c) Large Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	45%	41%	51%	NTMC	67%	63%	68%
WM	48%	44%	47%	DBC	45%	41%	50%
LC	50%	47%	50%	EC	25.69	22.08	29.74
TMC	79%	77%	82%	OC	39%	36%	41%

TABLE 4: Average test suite size of each approach with WS

approach	SS	OC	CC (Average)
size (All Classes)	51.35	47.77	31.59
size (Small Classes)	37.27	36.39	19.43
size (Large Classes)	103.53	89.95	76.64

Significant Cases. Fig. 4 (a) shows the comparison results of SS and OC on all 400 Java classes. SS outperforms OC on 121 (30.3%) classes (a.k.a., SS-outperforming classes) on average for each coverage. OC outperforms SS on 20 (4.9%) classes (a.k.a., OC-outperforming classes). These two approaches have no significant difference on 259 (64.8%) classes (a.k.a., No-significant classes) on average.

Average Coverage. Table 3 (a) shows the average coverage of all classes with three approaches. For exception coverage, the table shows the average number of the triggered exceptions since we can not know the total number of exceptions in a class [4]. The green number represents the highest coverage at a given criterion. SS outperforms OC for eight criteria' coverage. Among three approaches, SS reaches the highest coverage for four criteria. CC reaches the highest coverage for all criteria.

Average Suite Size. The first row of Table 4 shows the test suites' average sizes of all classes. Compared to CC (average suite size of all constituent criteria), the size of OC increases by 51.2% ($(47.77-31.59)/31.59$). Compared to OC, the size of SS increases by 7.4% ($(51.35-47.77)/47.77$).

►Small Classes.(< 200 branches)

Significant Cases. Fig. 4 (b) shows the comparison results of SS and OC on 315 small Java classes. For each criterion, on average, SS-outperforming classes are 71 (22.5%). OC-outperforming classes are 19 (5.9%). No-significant classes are 226 (71.6%).

Average Coverage. Table 3 (b) shows the average coverage of small classes. SS outperforms OC for seven criteria' coverage. SS reaches the highest coverage for five criteria.

Average Suite Size. The second row of Table 4 shows the average suite sizes of small classes. Compared to CC, the size of OC increases by 87.3%. Compared to OC, the size of SS increases by 2.4%.

►Large Classes. ($\geq$ 200 branches)

Significant Cases. Fig. 4 (c) shows the comparison results of SS and OC on 85 large Java classes. For each criterion, on average, SS-outperforming classes are 50 (59.1%). The number of OC-outperforming classes is 1 (1.3%). No-significant classes are 34 (39.6%).

Average Coverage. Table 3 (c) shows the average coverage of large classes. SS outperforms OC for eight criteria' coverage. SS reaches the highest coverage for two criteria.

Average Suite Size. The third row of Table 4 shows the average suite sizes of large classes. Compared to CC, the size of OC increases by 17.4%. Compared to OC, the size of SS increases by 15.1%.

Analysis. SS outperforms OC statistically, especially on the large classes. There is one exception: On the small classes, the number (73) of OC-outperforming classes in weak mutation is more than the counterpart number (60) (see Fig. 4 (b)). On average, each of those 73 classes has 82 branches and 321 mutants, while each of those 60 classes has 115 branches and 381 mutants. It also supports that SS outperforms OC on the large classes. Furthermore, in most cases, the average coverage of CC is higher than the

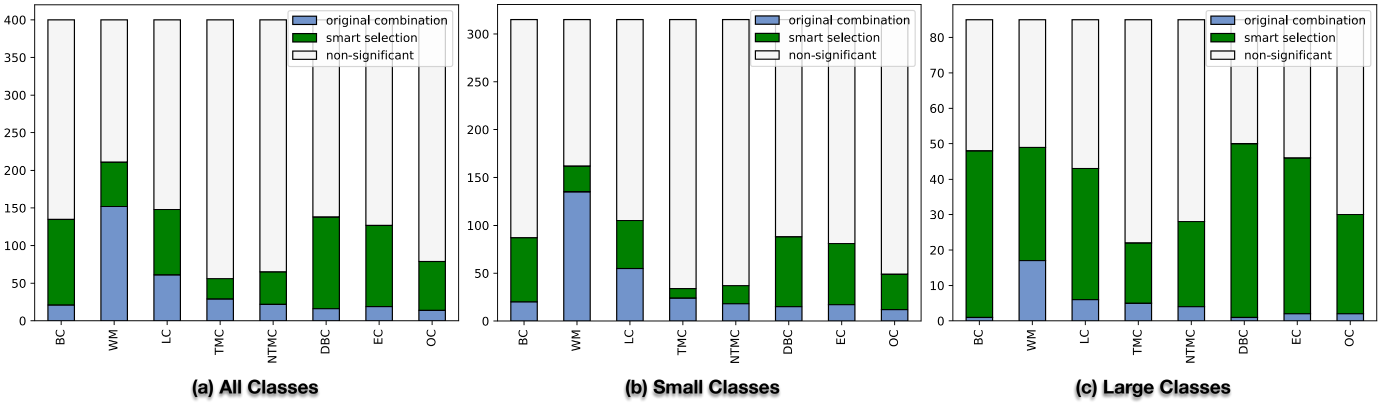


Fig. 5: Significant case summary of smart selection and the original combination with MOSA

one of OC. SS narrows the coverage gap between them. For example, the biggest gap is 10%, happening in the large classes' branch coverage. SS narrows the gap by 4% (see Table 3 (c)). These facts confirm that combining criteria offers more objectives for optimization, affecting the efficacy of GAs. The largeger classes bring more objectives, leading to a higher impact. The suite size increase brought by OC/SS to CC is significant, confirming the experimental results of the work proposing OC [4]. The main reason is that the GA (not only WS but also MOSA/DynaMOSA) needs more tests for more goals. With the coverage increase, the suite size of SS is also greater than OC. Compared with the extent of the suite increase brought by OC to CC, we regard that it is reasonable.

Answer to RQ1

With WS, SS outperforms OC statistically, especially on the large classes (i.e., the classes with no less than 200 branches).

4.3 RQ2: How does smart selection perform with MOSA?

Motivation. In this RQ, we evaluate smart selection with MOSA.

Methodology. The methodology is the same as RQ1's.

Result. ►**All Classes.**

TABLE 5: Average coverage results for each approach with MOSA

(a) All Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	57%	56%	58%	NTMC	71%	71%	69%
WM	60%	60%	60%	DBC	57%	55%	57%
LC	61%	60%	61%	EC	16.95	16.15	16.41
TMC	84%	83%	82%	OC	44%	44%	45%

(b) Small Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	59%	58%	60%	NTMC	73%	72%	71%
WM	62%	62%	62%	DBC	58%	58%	59%
LC	63%	63%	63%	EC	13.28	13.07	12.73
TMC	85%	85%	84%	OC	45%	45%	46%

(c) Large Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	49%	46%	51%	NTMC	66%	64%	64%
WM	52%	49%	51%	DBC	50%	46%	51%
LC	54%	52%	53%	EC	30.54	27.53	30.03
TMC	79%	78%	77%	OC	40%	38%	42%

TABLE 6: Average test suite size of each approach with MOSA

approach	SS	OC	CC (Average)
size (All Classes)	57.03	54.46	31.47
size (Small Classes)	38.27	38.83	19.85
size (Large Classes)	126.56	112.38	74.53

Significant Cases. Fig. 5 (a) shows the comparison results of SS and OC on all 400 Java classes. For each criterion, on average, SS-outperforming classes are 78 (19.5%). OC-outperforming classes are 42 (10.4%). No-significant classes are 280 (70.1%).

Average Coverage. Table 5 (a) shows the average coverage of all classes. SS outperforms OC for five criteria' coverage. Among three approaches, SS reaches five criteria' highest coverage.

Average Suite Size. The first row of Table 6 shows the average suite sizes of all classes. Compared to CC, the size of OC increases by 73.1%. Compared to OC, the size of SS increases by 4.7%.

►**Small Classes.**

Significant Cases. Fig. 5 (b) shows the comparison results of SS and OC on 315 small Java classes. For each criterion, on average, SS-outperforming classes are 43 (13.8%). OC-outperforming classes are 37 (11.7%). No-significant classes are 235 (74.5%).

Average Coverage. Table 5 (b) shows the average coverage of small classes. SS outperforms OC for three criteria' coverage. SS reaches three criteria' highest coverage.

Average Suite Size. The second row of Table 6 shows the average suite sizes of small classes. Compared to CC, the size of OC increases by 95.6%. OC is nearly equal to SS.

►**Large Classes.**

Significant Cases. Fig. 5 (c) shows the comparison results of SS and OC on 85 large Java classes. For each criterion, on average, SS-outperforming classes are 35 (40.9%). OC-outperforming classes are 5 (5.6%). No-significant classes are 46 (53.5%).

Average Coverage. Table 5 (c) shows the average coverage of large classes. SS outperforms OC for eight criteria' coverage. SS reaches five criteria' highest coverage.

Average Suite Size. The third row of Table 6 shows the average suite sizes of large classes. Compared to CC, the

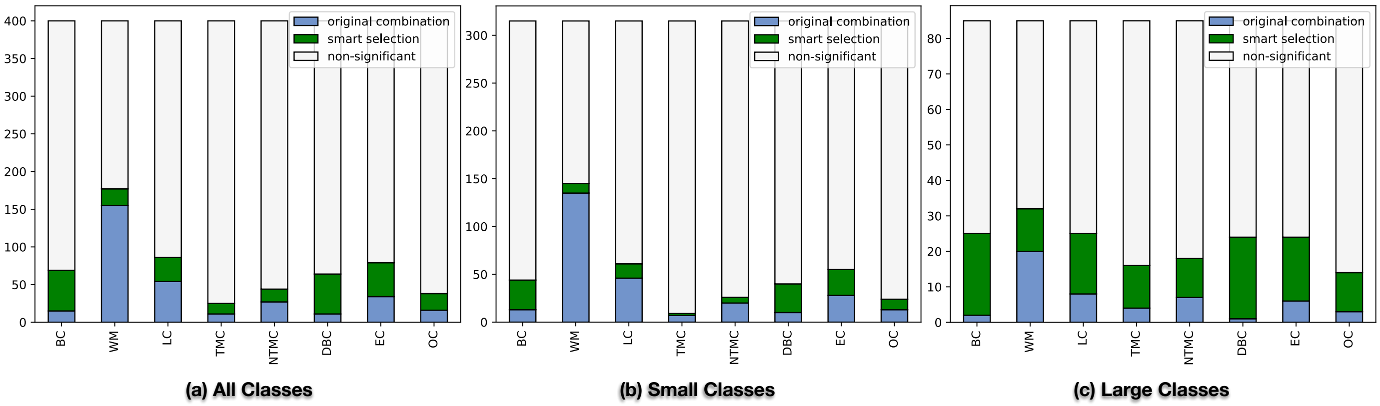


Fig. 6: Significant case summary of smart selection and the original combination with DynaMOSA

size of OC increases by 50.7%. Compared to OC, the size of SS increases by 12.6%.

Analysis. SS outperforms OC on the large classes like WS. But the advantage of SS is unnoticeable on the small classes. The coverage gap between CC and OC is not significant as the gap in WS. SS nearly bridges this gap. The biggest gap is 5%, happening in branch coverage of the large classes. SS narrows this gap by 3%. The suite size gap between SS and OC is smaller than on WS, which is consistent with the fact that SS and OC have a smaller coverage gap on MOSA. These facts show the advantage of multi-objective approaches (e.g., MOSA) over single-objective approaches (e.g., WS) [2], [5], [6]. However, the advantage of SS on the large classes indicates that too many objectives also affect the multi-objective algorithms.

Answer to RQ2

With MOSA, SS outperforms OC statistically on the large classes. Smart selection has only a slight advantage on the small classes.

4.4 RQ3: How does smart selection perform with DynaMOSA?

Motivation. In this RQ, we evaluate smart selection with DynaMOSA.

Methodology. The methodology is the same as RQ1's. Note that the difference is that we add BC to the guiding criteria no matter whether the approach contains BC. For example, if the constituent criterion is LC, we will combine BC with it to guide DynaMOSA. Similarly, we add BC to the representative group (DBC, LC, EC, OC) selected by smart selection. The reason is that DynaMOSA can not run without BC since it needs the goals of branch coverage to build the control dependency graph [3].

Result. ►All Classes.

Significant Cases. Fig. 6 (a) shows the comparison results of SS and OC on all 400 Java classes. For each criterion, on average, SS-outperforming classes are 32 (8.1%). OC-outperforming classes are 40 (10.1%). No-significant classes are 328 (81.8%).

Average Coverage. Table 7 (a) shows the average coverage of all classes with three approaches. SS outperforms OC for one criterion's coverage, i.e., exception coverage. Among three approaches, SS reaches the highest coverage for three criteria.

TABLE 7: Average coverage results for each approach with DynaMOSA

(a) All Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	58%	58%	58%	NTMC	71%	71%	70%
WM	60%	61%	62%	DBC	57%	57%	58%
LC	61%	62%	62%	EC	17.15	17.14	16.64
TMC	83%	83%	81%	OC	45%	45%	45%

(b) Small Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	60%	59%	60%	NTMC	72%	73%	72%
WM	63%	63%	64%	DBC	59%	59%	59%
LC	63%	64%	64%	EC	13.42	13.42	12.81
TMC	84%	85%	82%	OC	46%	46%	46%

(c) Large Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	51%	51%	53%	NTMC	66%	66%	65%
WM	53%	52%	54%	DBC	51%	50%	53%
LC	54%	54%	55%	EC	30.98	30.93	30.81
TMC	79%	79%	78%	OC	41%	41%	42%

TABLE 8: Average test suite size of each approach with DynaMOSA

approach	SS	OC	CC (Average)
size (All Classes)	61.13	60.59	39.2
size (Small Classes)	38.9	39.59	23.71
size (Large Classes)	143.51	138.44	96.58

Average Suite Size. The first row of Table 8 shows the average suite sizes of all classes. Compared to CC, the size of OC increases by 54.7%. OC is nearly equal to SS.

►Small Classes.

Significant Cases. Fig. 6 (b) shows the comparison results of SS and OC on 315 small Java classes. For each criterion, on average, SS-outperforming classes are 17 (5.2%). OC-outperforming classes are 34 (10.8%). No-significant classes are 265 (84%).

Average Coverage. Table 7 (b) shows the average coverage of small classes. SS outperforms OC for one criterion's coverage (branch coverage). SS reaches two criteria' highest coverage.

Average Suite Size. The second row of Table 8 shows the average suite sizes of small classes. Compared to CC, the size of OC increases by 66.9%. OC is nearly equal to SS.

►Large Classes.

Significant Cases. Fig. 6 (c) shows the comparison results

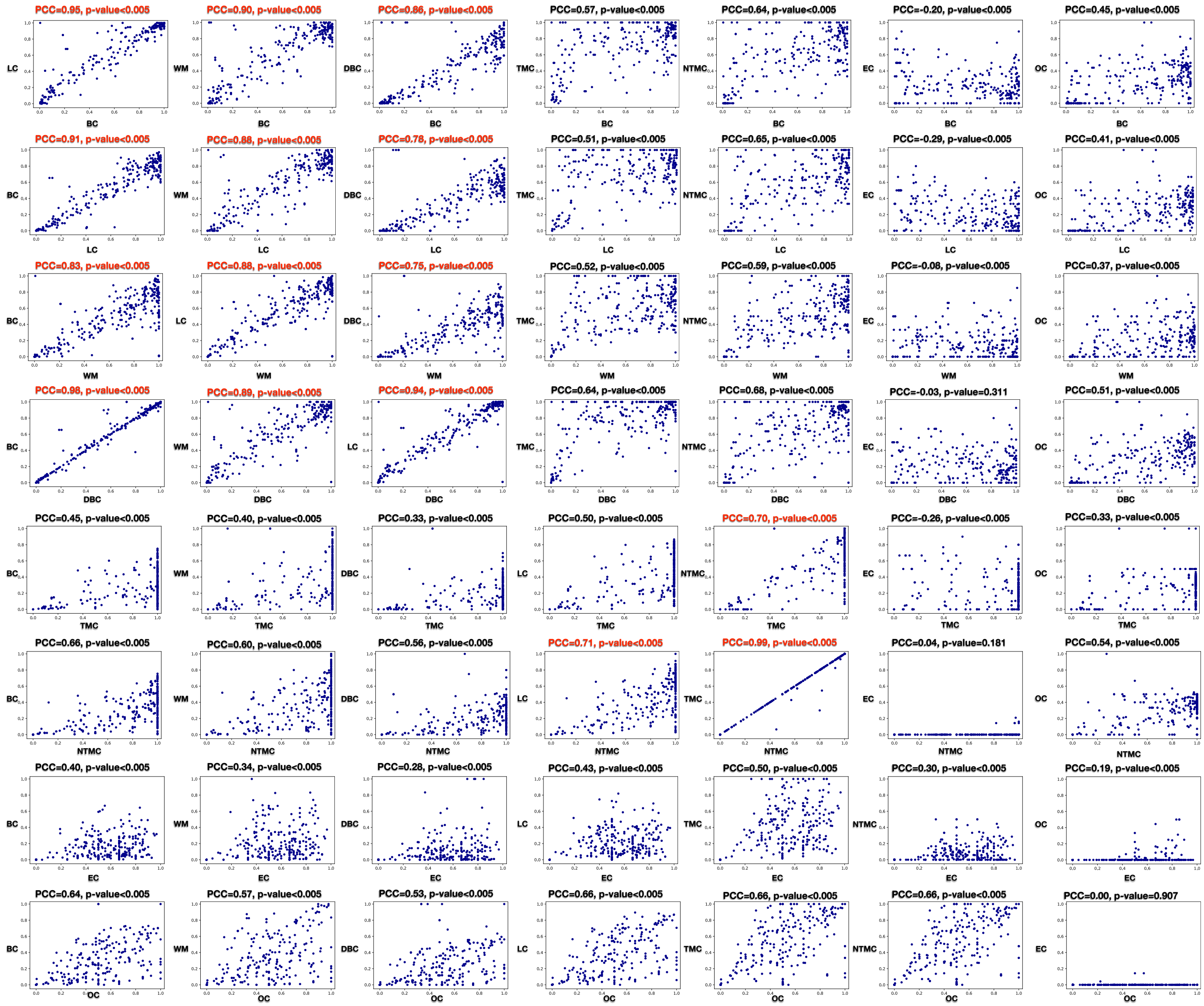


Fig. 7: Coverage correlation data for each criterion pair when the GA is WS, including the Pearson Correlation Coefficient (PCC) with its p-value (The text is marked with red color when PPC ≥ 0.7) and the scatter plot of data points in which the X-axis of the scatter plot represents the coverage values of the criterion guiding GA, and the Y-axis represents that of the criterion to be measured for its coverage

of SS and OC on 85 large Java classes. For each criterion, on average, SS-outperforming classes are 16 (18.7%). OC-outperforming classes are 6 (7.5%). No-significant classes are 63 (73.8%).

Average Coverage. Table 7 (c) shows the average coverage of large classes. SS outperforms OC for three criteria' coverage. SS reaches three criteria' highest coverage.

Average Suite Size. The third row of Table 8 shows the average suite sizes of large classes. Compared to CC, the size of OC increases by 43.3%. Compared to OC, the size of SS increases by 3.7%.

Analysis. SS still outperforms OC on the large classes, but not as obvious as WS and MOSA. In addition, SS is almost the same or slightly worse than OC on the small classes. Furthermore, the coverage gaps among the three approaches are not significant. The gap in the suite size between SS and OC is slight as in the coverage. One reason is that DynaMOSA selects the uncovered goals into the iteration process only when its branch dependencies are covered (see Sec. 2.2). Hence, the number of optimization objectives is reduced. Therefore, an increase in the goals has a much smaller impact on DynaMOSA's coverage performance than on WS and MOSA. Another reason is that we add BC to SS's guiding criteria because DynaMOSA can only run with BC. Consequently, SS's reduction in optimization objectives is undermined. Due to the same cause, we combine BC with an arbitrary constituent criterion as the guiding criteria, thus improving CC's ability to guide DynaMOSA. As a result, the differences between the three approaches are narrowed.

Answer to RQ3

With DynaMOSA, the coverage of SS and OC is close for most criteria.

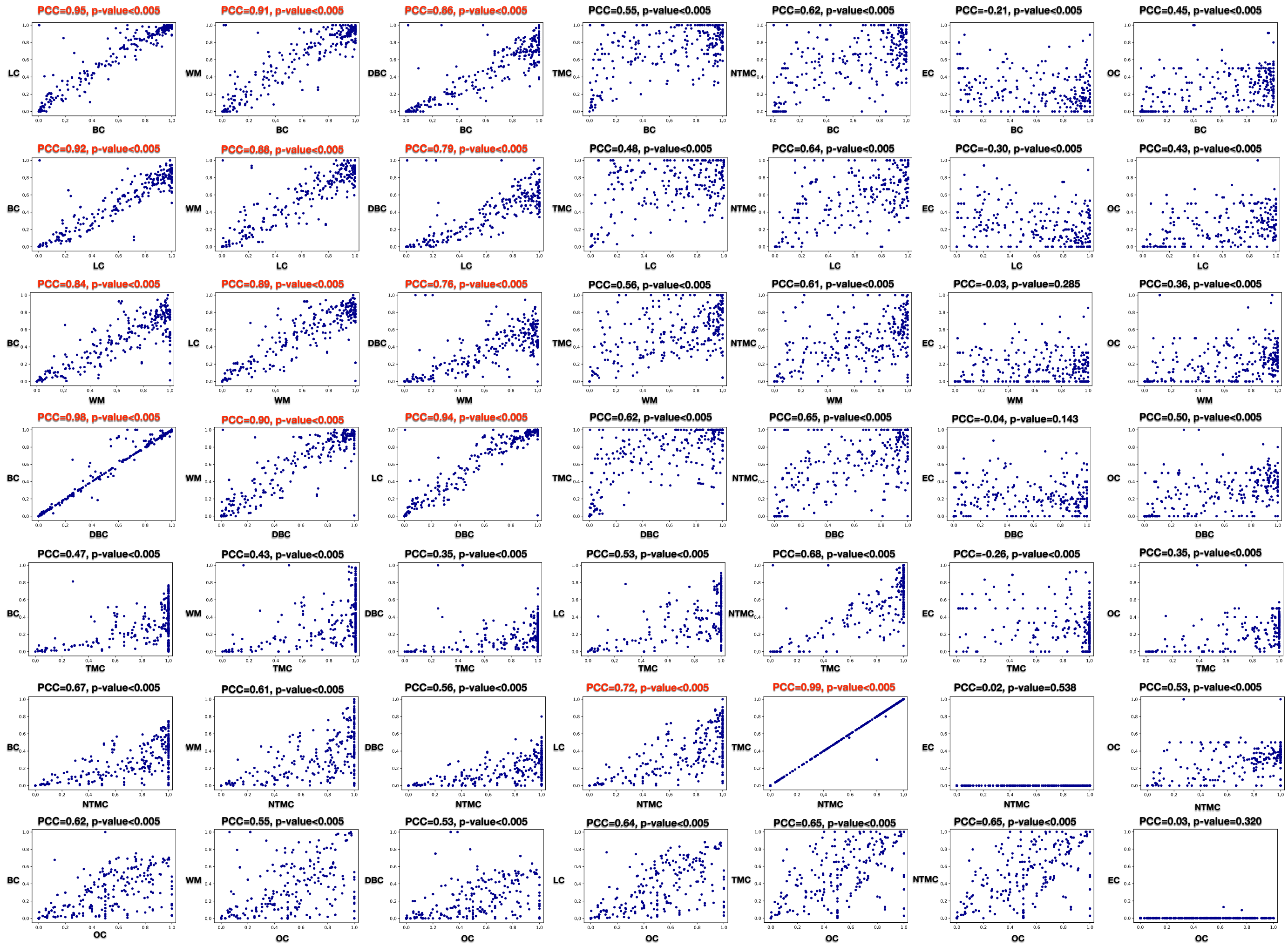


Fig. 8: Coverage correlation data for each criterion pair when the GA is MOSA

4.5 RQ4: Do criteria within the same criteria group exhibit a coverage correlation?

Motivation. One fundamental assumption of smart selection is that the criteria within the same criteria group have a strong coverage correlation (see Sec. 3.2) so that we can choose a representative criterion to represent a group, thus reducing the goals optimized by GAs. In this RQ, we aim to verify this assumption.

Methodology. In this RQ, we iterate to run EvoSuite, and each iteration has a specific four-tuple configuration, including a Java class, a GA, a criterion to guide GA, and a search budget. The Java class has 400 options (i.e., 400 Java classes used in RQ1-3), the GA has two options (WS and MOSA), the criterion has eight options (BC, LC, DBC, WM, TMC, NTMC, EC, and OC), and the budget has four options: 120 seconds, 300 seconds, 480 seconds, and 600 seconds. After running EvoSuite, we record the coverage score of all eight coverage criteria to calculate the correlation of a pair of criteria. Note that (1) DynaMOSA cannot run without BC (see Sec. 4.4). In other words, we can not calculate the correlation of a pair of criteria without the interference of BC. As a result, DynaMOSA is not used in this RQ; (2) MOSA can not run only with EC because EvoSuite can not provide any of EC's fitness functions at the beginning of running GA since EvoSuite does not know any exceptions thrown in the subject under test until GA randomly generates a unit test catching one. However, MOSA needs at least one fitness function to build Pareto Fronts [2]. As a result, when the GA is MOSA, the criterion to guide GA only has seven options, i.e., there are $8 \times 7 = 64$ criterion pairs when the GA is WS while the counterpart number is $7 \times 7 = 49$ when the GA is MOSA; (3) The origin coverage value provided by EvoSuite for EC is the number of exceptions triggered by the test suite. We normalize it as a value ranging from 0 to 1 (like other criteria) by dividing the origin value by the maximum number of exceptions triggered among all suites for the same Java class. After running all configurations on EvoSuite, we obtain $400 \times 4 = 1600$ data points to estimate the correlation for each criterion pair and algorithm. Based on these points, we calculate the Pearson Correlation Coefficient (PCC) and the significant value p [35]. PCC ranges from -1 to $+1$, where 0 indicates no correlation. Correlation values of -1 or $+1$ suggest a perfect linear relationship. Positive values indicate that when x increases, y also increases. Conversely, negative values imply that as x increases, y decreases [8].

Result. ▶WS. Fig. 7 shows the coverage value point scatter plot, PCC, and the p-value of each criterion pair (A, B) when

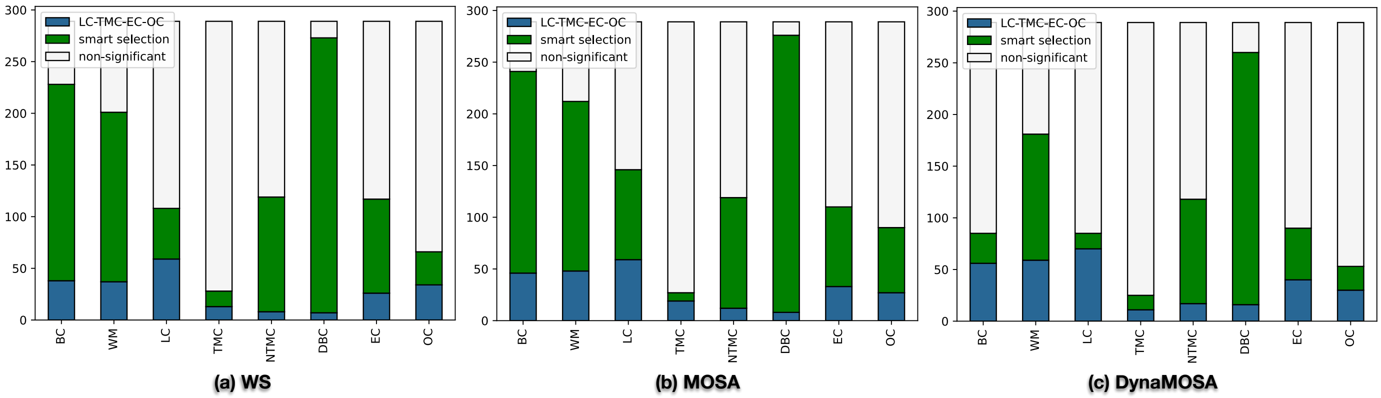


Fig. 9: Significant case summary of smart selection and LT (i.e., LC-TMC-EC-OC)

using criterion A to guide WS.

Coverage correlation of the same groups' criteria. There are 14 criterion pairs in the same groups: (BC, DBC), (DBC, BC), (BC, LC), (LC, BC), (BC, WM), (WM, BC), (DBC, LC), (LC, DBC), (DBC, WM), (WM, DBC), (LC, WM), (WM, LC), (TMC, NTMC), and (NTMC, TMC). Among them, the minimum PCC is 0.70 (TMC, NTMC), the maximum two are 0.99 (NTMC, TMC) and 0.98 (DBC, BC), and both the mean and median values are closely 0.88. Besides, all of the p-values are smaller than 0.005.

The criterion pairs among them we are most concerned with are (DBC, BC), (DBC, LC), (DBC, WM), and (NTMC, TMC) since we use DBC to represent (BC, DBC, LC, WM) and NTMC to represent (TMC, NTMC) (see Sec. 3.3). The PCCs of (DBC, BC), (DBC, LC), (DBC, WM), and (NTMC, TMC) are 0.98, 0.94, 0.89, and 0.99, respectively.

Coverage correlation of the different groups' criteria. There are $56-14=42$ criterion pairs in the different groups. Among them, the minimum PCC is -0.29 (LC, EC), the maximum is 0.71 (NTMC, LC), and the mean/median value is $0.40/0.50$.

►**MOSA.** Fig. 8 shows the coverage value point scatter plot, PCC, and the p-value of each criterion pair (A, B) when using criterion A to guide MOSA.

Coverage correlation of the same groups' criteria. Among the 14 criterion pairs in the same groups, the minimum PCC is 0.68 (TMC, NTMC), the maximum two are 0.99 (NTMC, TMC) and 0.98 (DBC, BC), and the mean/median value is $0.88/0.89$. All of the p-values are smaller than 0.005, like WS's data.

Similar to WS, the PCCs of the criterion pairs we are most concerned with (i.e., (DBC, BC), (DBC, LC), (DBC, WM), and (NTMC, TMC)) are 0.98, 0.94, 0.90, and 0.99, respectively.

Coverage correlation of the different groups' criteria. There are $49-14=35$ criterion pairs in the different groups. Among them, the minimum PCC is -0.30 (LC, EC), the maximum is 0.72 (NTMC, LC), and the mean/median value is $0.42/0.53$.

►**Difference between WS and MOSA.** For a criterion pair, assume that $\text{PCC}_{\text{WS}}/\text{PCC}_{\text{MOSA}}$ is the PCC value when the GA is WS/MOSA, and $|\text{PCC}_{\text{WS}}-\text{PCC}_{\text{MOSA}}|$ is the difference between WS and MOSA. The min, max, mean, and median differences are 0.0011 (NTMC, TMC)/0.049 (WM, EC)/0.013/0.011, respectively.

Analysis. Firstly, the coverage correlation of a criterion pair from the same group is significantly higher than that of a criterion pair from different groups, confirming the effectiveness of our criteria grouping. Secondly, the difference in the same criterion pair's correlation with different GAs is tiny, showing the correlation's independence to GAs.

Answer to RQ4

On average, the Pearson Correlation Coefficient of a criterion pair's coverage values from the same group (nearly 0.88) is significantly higher than that of a criterion pair from different groups (nearly 0.41), confirming the effectiveness of our criteria grouping.

4.6 RQ5: Are representative criteria efficient in guiding SBST?

In Sec. 3.3, we choose four criteria (DBC, NTMC, EC, and OC) to present four groups ((BC, DBC, LC, WM), (TMC, NTMC), (EC), and (OC)), respectively. Regardless of the two groups with only one criterion, The main assumptions deriving us to choose DBC and NTMC are (1) DBC is more effective in guiding GAs than EC and OC, and (2) DBC/NTMC can represent BC/TMC, but the opposite does not hold. In this RQ, we aim to verify these assumptions.

Methodology. We choose two new criteria combinations to guide GAs as two new approaches: (LC, TMC, EC, OC) (a.k.a, LT, i.e., LC to present group 1 and TMC to present group 2) and (WM, TMC, EC, OC) (a.k.a, WT). Then, we follow RQ1-3's methodology to compare smart selection with LTEO and WTEO with three algorithms: WS, MOSA, and DynaMOSA. Similar to RQ3 (Sec. 4.4), we add BC to the guiding criteria since DynaMOSA can not run without BC.

Subjects. We take the 400 classes as the experimental subjects. However, there are 111 classes on which EvoSuite with at least one of six configurations (2 combinations $\times 3$ GAs) crashed. As a result, there are remaining 289 classes as this RQ's subjects, which is still a large sample according to the previous studies [2], [3], [36].

Result. ►**WS.**

Significant Cases. Fig. 9 (a) and 10 (a) show the significant case summary for SS versus LT and SS versus WT, respectively, when the GA is WS. On average of all criteria, SS-

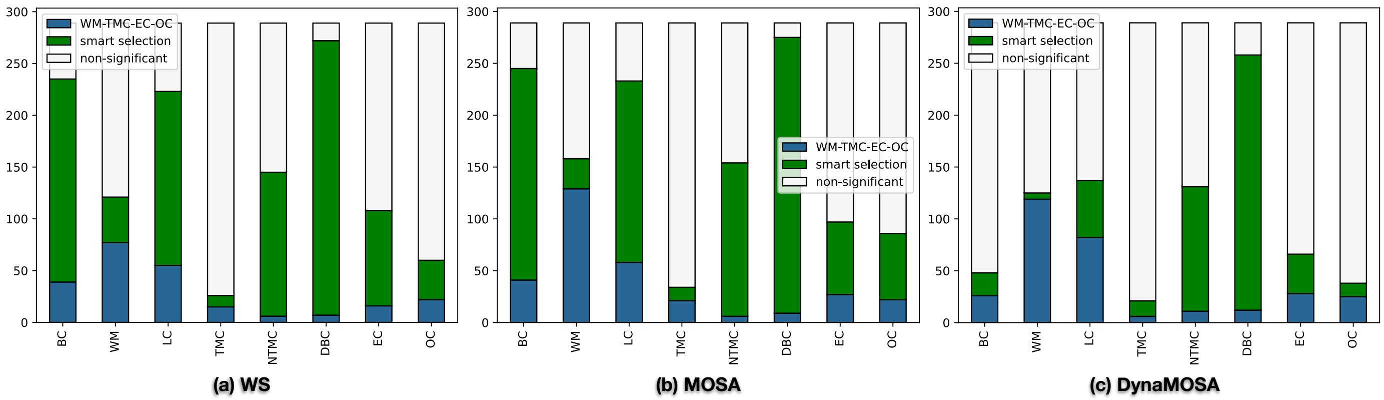


Fig. 10: Significant case summary of smart selection and WT (i.e., WM-TMC-EC-OC)

TABLE 9: Average coverage and size results for smart selection, LT, and WT

approach	SS	LT	WT	approach	SS	LT	WT
(a) WS (Suite Size: SS (47.63), LT (40.94), WT (41.91))							
BC	56%	52%	51%	NTMC	72%	69%	68%
WM	60%	58%	59%	DBC	56%	40%	40%
LC	60%	60%	58%	EC	16.12	15.51	15.56
TMC	84%	84%	84%	OC	44%	44%	44%
(b) MOSA (Suite Size: SS (53.66), LT (45.16), WT (46.10))							
approach	SS	LT	WT	approach	SS	LT	WT
BC	58%	53%	52%	NTMC	72%	69%	67%
WM	61%	59%	61%	DBC	57%	41%	41%
LC	62%	61%	59%	EC	17.18	16.84	16.89
TMC	84%	84%	84%	OC	45%	44%	44%
(c) DynaMOSA (Suite Size: SS (57.09), LT (52.54), WT (55.30))							
approach	SS	LT	WT	approach	SS	LT	WT
BC	58%	59%	58%	NTMC	72%	70%	69%
WM	62%	61%	62%	DBC	58%	47%	47%
LC	62%	62%	63%	EC	17.36	17.44	17.56
TMC	84%	84%	84%	OC	45%	45%	45%

outperforming-LT classes are 115 (40%), LT-outperforming-SS classes are 28 (10%), and no-significant classes are 146 (50%); SS-outperforming-WT classes are 119 (41%), WM-outperforming-SS classes are 30 (10%), and no-significant classes are 140 (48%).

Average Coverage. Table 9 (a) shows the average coverage for SS, LT, and WT. SS achieves the highest coverage on all criteria. Notably, SS is higher than LT and WT by 16% on direct branch coverage.

►MOSA.

Significant Cases. Fig. 9 (b) and 10 (b) show the significant case summary for SS versus LT and SS versus WT, respectively, when the GA is MOSA. On average of all criteria, SS-outperforming-LT classes are 121 (42%), LT-outperforming-SS classes are 32 (11%), and no-significant classes are 136 (47%); SS-outperforming-WT classes are 121 (42%), WM-outperforming-SS classes are 39 (13%), and no-significant classes are 129 (45%).

Average Coverage. Table 9 (b) shows the average coverage for SS, LT, and WT. Like the GA being WS, SS achieves the highest coverage on all criteria; SS is higher than LT and WT by 16% on direct branch coverage.

►DynaMOSA.

Significant Cases. Fig. 9 (c) and 10 (c) show the significant case summary for SS versus LT and SS versus WT, respectively, when the GA is DynaMOSA. On average of all criteria, SS-outperforming-LT classes are 75 (26%), LT-outperforming-SS classes are 40 (13%), and no-significant classes are 177 (61%); SS-outperforming-WT classes are 64 (22%), WM-outperforming-SS classes are 39 (13%), and no-significant classes are 186 (64%).

Average Coverage. Table 9 (c) shows the average coverage for SS, LT, and WT. Unlike the GA being WS and MOSA, the coverage values of the three approaches are close on all criteria except for direct branch coverage, on which SS is higher than LT and WT by 11%.

Analysis. Smart selection outperforms two criteria combinations (LC, TMC, EC, OC) and (WM, TMC, EC, OC) on almost all criteria (except for EC, OC, and TMC) when the GA is WS or MOSA. This result shows that the representative criteria (mainly DBC) selected from smart selection are more efficient in guiding GAs. The differences between smart selection and two criteria combinations on all criteria (except for DBC) are tiny when the GA is DynaMOSA. The main reason is that We add BC to these two criteria combinations since DynaMOSA cannot run without BC, thus improving the ability to guide DynaMOSA. However, smart selection still outperforms them on DBC by 11%, confirming our assumption in Sec. 3.3 that DBC can represent BC, but the opposite does not hold.

Answer to RQ5

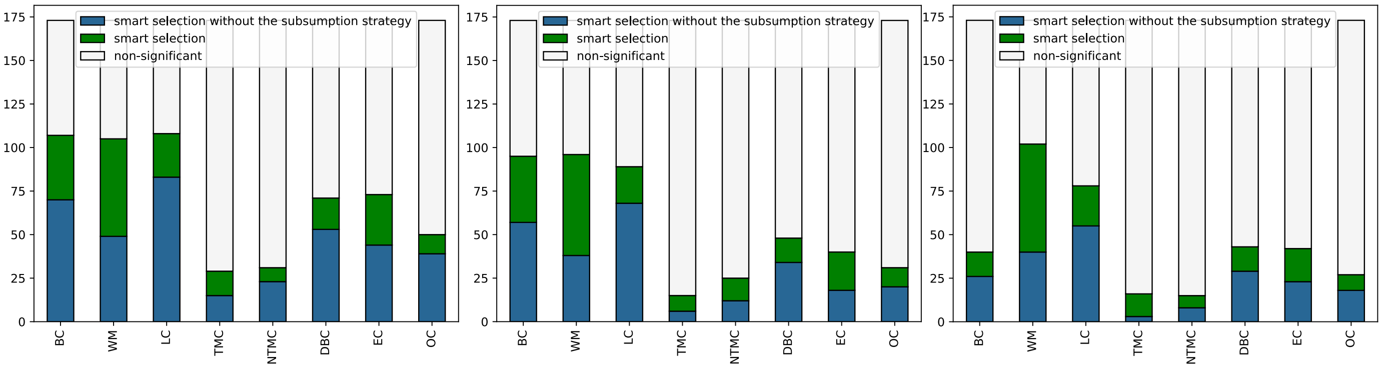
In most cases, smart selection is better than two criteria combinations (LC, TMC, EC, OC) and (WM, TMC, EC, OC), confirming that the criteria selected from the criteria groups are more efficient in guiding GA and more representative than the other criteria in the same criteria group.

4.7 RQ6: How does the subsumption strategy affect the performance of smart selection?

Motivation. We select the representative goals from line coverage and weak mutation by the subsumption relationships (see Sec 3.4). We need to test how it affects the performance of smart selection.

Subjects. From 400 classes, we select those classes that satisfy this condition: The subsumption strategy can find at least one line coverage goal and one mutant. As a result, 173 classes are selected.

Methodology. We take smart selection without the subsumption strategy (SSWS) as a new approach. To compare SS and SSWS, we follow RQ1's methodology.



(a) WS (b) MOSA (c) DynaMOSA

Fig. 11: Significant case summary of smart selection and smart selection with subsumption strategy

TABLE 10: Average coverage and size results for smart selection and smart selection without the subsumption strategy

(a) WS (Suite Size: SS (45.18), SSWS (48.08))					
approach	SS	SSWS	approach	SS	SSWS
BC	47%	48%	NTMC	63%	63%
WM	52%	53%	DBC	46%	48%
LC	51%	53%	EC	16.43	17.46
TMC	79%	78%	OC	38%	39%
(b) MOSA (Suite Size: SS (52.08), SSWS (51.66))					
approach	SS	SSWS	approach	SS	SSWS
BC	49%	49%	NTMC	63%	63%
WM	53%	53%	DBC	49%	49%
LC	53%	54%	EC	18.19	18.32
TMC	79%	78%	OC	38%	39%
(c) DynaMOSA (Suite Size: SS (57.1), SSWS (54.79))					
approach	SS	SSWS	approach	SS	SSWS
BC	50%	50%	NTMC	63%	63%
WM	54%	54%	DBC	49%	49%
LC	53%	54%	EC	18.26	18.68
TMC	79%	78%	OC	39%	39%

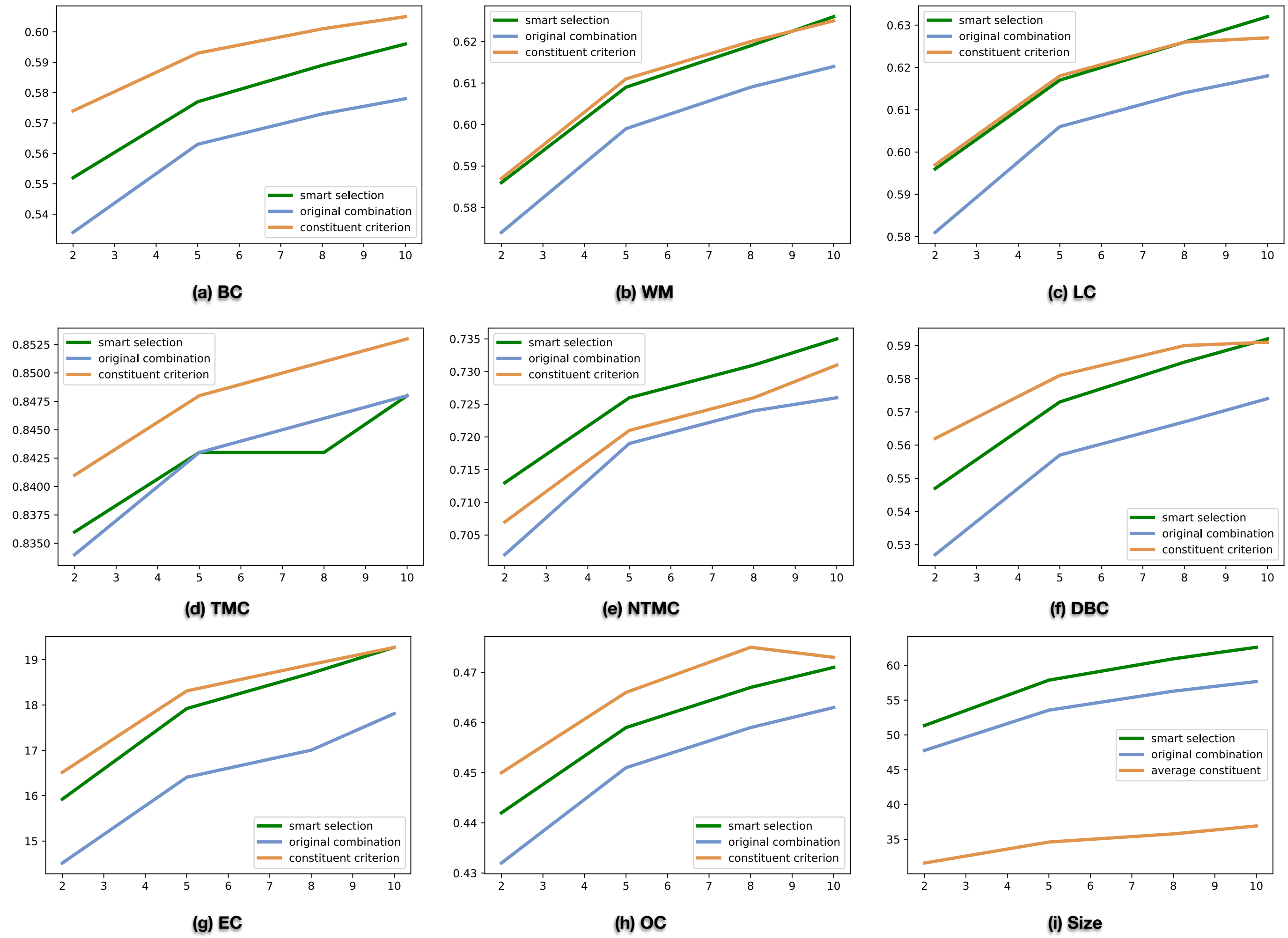
Result. ►WS.
Significant Cases. Fig. 11 (a) shows the comparison results of SS and SSWS on 173 classes with WS. For each criterion, on average, SS-outperforming classes are 25 (14.5%). SSWS-outperforming classes are 47 (27.2%). No-significant classes are 101 (58.3%).
Average Coverage. Table 10 (a) shows the average coverage for WS. SS outperforms SSWS on top-level method coverage. SSWS outperforms SS on six criteria.
►MOSA.
Significant Cases. Fig. 11 (b) shows the results with GA being MOSA. On average, SS-outperforming classes are 23 (13.3%). SSWS-outperforming classes are 32 (18.5%). No-significant classes are 118 (68.2%).
Average Coverage. Table 10 (b) shows the average coverage for MOSA. SS outperforms SSWS for one top-level method coverage. SSWS outperforms SS on three criteria.
►DynaMOSA.
Significant Cases. Fig. 11 (c) shows the results with GA being DynaMOSA. On average, SS-outperforming classes are 20 (11.6%). SSWS-outperforming classes are 25 (14.5%). No-significant classes are 128 (73.9%).
Average Coverage. Table 10 (c) shows the average coverage for DynaMOSA. SS outperforms SSWS on top-level method coverage. SSWS outperforms SS on two criteria.
Analysis. SSWS outperforms slightly SS for most criteria on WS, confirming that an increase in the objectives has a much bigger impact on WS than on MOSA and DynaMOSA. Furthermore, the results are different on line coverage and weak mutation for which SS adds subsets. For three algorithms, SSWS is better in line coverage in terms of the outperforming classes and average coverage. Contrarily, SS is better in weak mutation in terms of the outperforming classes. It indicates that the coverage correlation between (direct) branch coverage and line coverage is stronger than the one between (direct) branch coverage and weak mutation. As for the suite size, Table 10 shows that SS and SSWS are similar. Unexpectedly, SS outperforms SSWS on top-level method coverage. We analyze some classes qualitatively. For example, there is a public method named *compare* in an inner class of the class *org.apache.hadoop.mapred* [37]. The results show that 88 out of 90 test suites generated by SS cover this top-level method goal, while only 1 out of 90 test suites generated by SSWS covers this goal. We find that this method contains 8 lines, 2 branches, and 3 output goals. EvoSuite skips the branches and output goals in the inner class (lines, methods, and mutants are kept). This class has 350 branches, 48 methods, and 10 output goals. SS selects 14 lines and 216 mutants for this class (0 lines and 10 mutants for this method). As a result, if a test directly invokes this method, under SS, at most $(1+10)$ out of 638 (2%) goals are closer to being covered; under SSWS, the number is 1 out of 408 (0.2%). It explains why all algorithms with SSWS tend to ignore this method goal since the gain is tiny. We find that this scenario is common in those large classes that contain short and branch-less methods.

Answer to RQ6

Smart selection without the subsumption strategy outperforms slightly smart selection in most criteria on WS (except for WM and TMC). Smart selection outperforms slightly smart selection without the subsumption strategy in WM and TMC on three algorithms.

4.8 RQ7: How does smart selection perform under different search budgets?

Motivation. In all the above RQs, we evaluate smart selection and other baselines under a fixed search budget, i.e., 2 minutes. In this RQ, we aim to answer how smart selection



(a) BC (b) WM (c) LC

(d) TMC (e) NTMC (f) DBC

(g) EC (h) OC (i) Size

Fig. 12: The coverage and size of three strategies on WS under the various budgets (2-10 minutes).

varies and the gaps among three strategies (i.e., smart selection, the original combination, and the constituent criterion) vary under different search budgets.

Subjects. Similar to RQ1-3, we select the 400 Java classes as the subjects under test.

Configuration. Similar to RQ1-3, we run three genetic algorithms (WS, MOSA, and DynaMOSA) with smart selection and another two strategies but under three more search budgets: 5, 8, and 10 minutes. Following the previous study [4], we repeat 5 times per Java class for the 10-minute budget. We repeat 10 times per Java class for the 5-minute and 8-minute budgets. Regardless of the search budget difference, this RQ's running configuration of EvoSuite is the same as that of RQ1-3.

Result.

►WS.

Fig. 12 shows the average coverage and suite sizes of three strategies on WS under the various budgets. In Sec. 3.2, the first step of smart selection, we cluster the eight criteria into four groups: (1) BC, DBC, LC, and WM; (2) TMC and NTMC; (3) EC; and (4) OC. We present the data by groups because the coverage of criteria in different groups changes differently as the search budget increases.

①**BC-DBC-LC-WM.** Fig. 12 (a), (b), (c), and (f) show the coverage change of BC, WM, LC, and DBC. From them, we observe that (1) the coverage of three strategies increases similarly as the budget increases. For example, from 2 to 10 minutes, the BC increase of smart selection/the original combination/the constituent criterion is $4\%/4\%/3\%$. (2) The coverage gap keeps stable from 2 to 10 minutes. For example, when the budget is 2 minutes, the BC gap between smart selection and the original combination is 2%, and the gap between the constituent criterion and smart selection is 2%. When the budget is 10 minutes, the gap between smart selection and the original combination is 2% too, and the gap between the constituent criterion and smart selection is 1%; (3) Compared to the other two criteria, the LC and WM coverage of the constituent criterion and smart selection is close but keeps higher than the original combination by nearly 2% as the budget changes.

②**TMC-NTMC.** Fig. 12 (d) and (e) show the coverage change of TMC and NTMC. As the budget increases, (1) the coverage increase of the three strategies is tiny. For example, from 2 to 10 minutes, the TMC increase of smart selection/the original combination/the constituent criterion is $0.7\%/1.5\%/1.7\%$; (2) The coverage gap keeps tiny. For example, the highest coverage of TMC is 85.3%, reached by the constituent criterion with the 10-minute budget. The

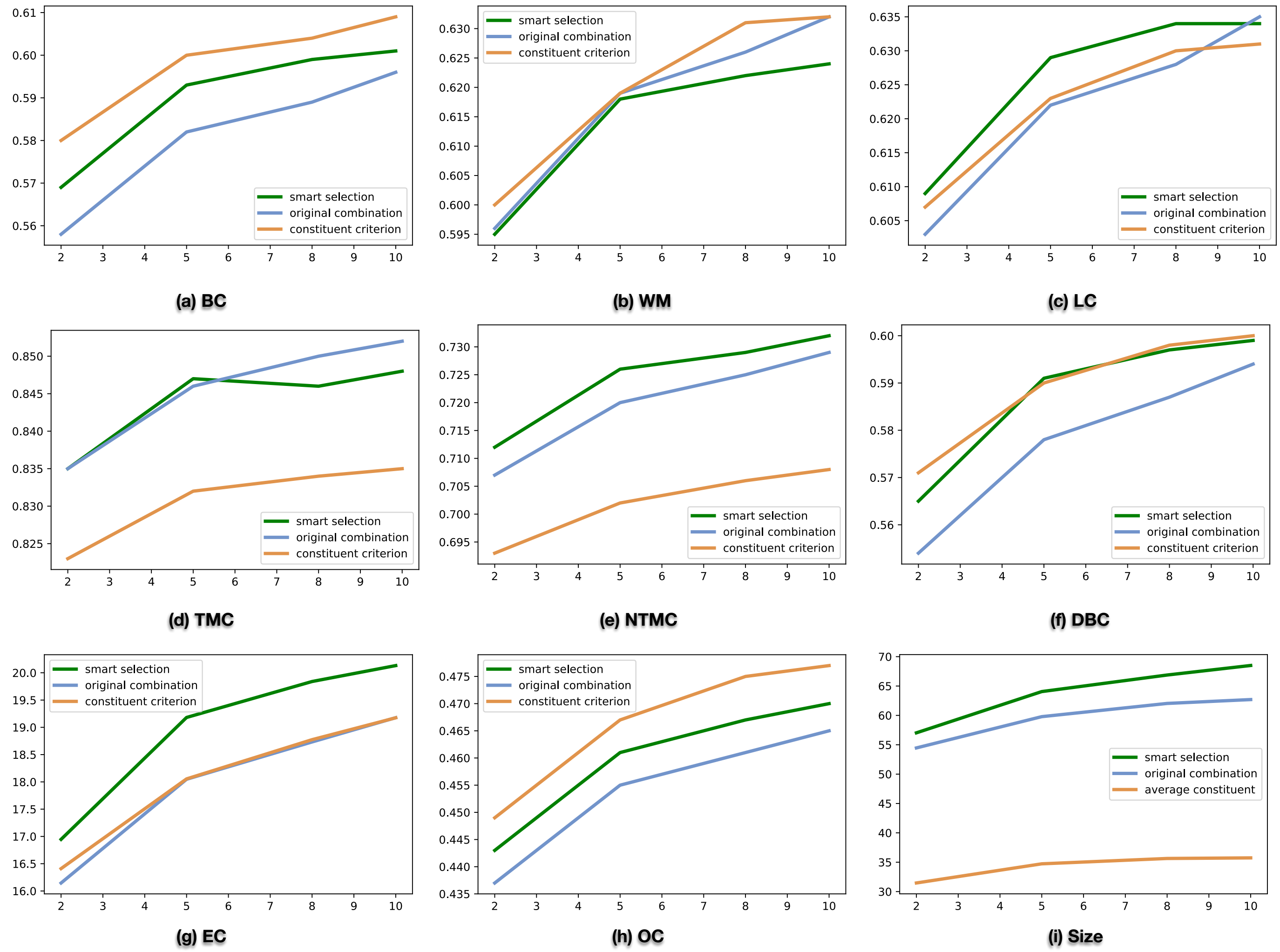


Fig. 13: The coverage and size of three strategies on MOSA under the various budgets (2-10 minutes).

lowest coverage is 83%, reached by the original combination with the 2-minute budget. The difference is only 2.3%. For comparison, the corresponding data of BC is 7%.

③④**EC and OC.** Fig. 12 (g) and (h) show the coverage change of EC and OC. The coverage changes of them are similar. As the budget increases, the coverage of the constituent criterion and smart selection is close, and there are stable coverage gaps between the original combination and the other two strategies.

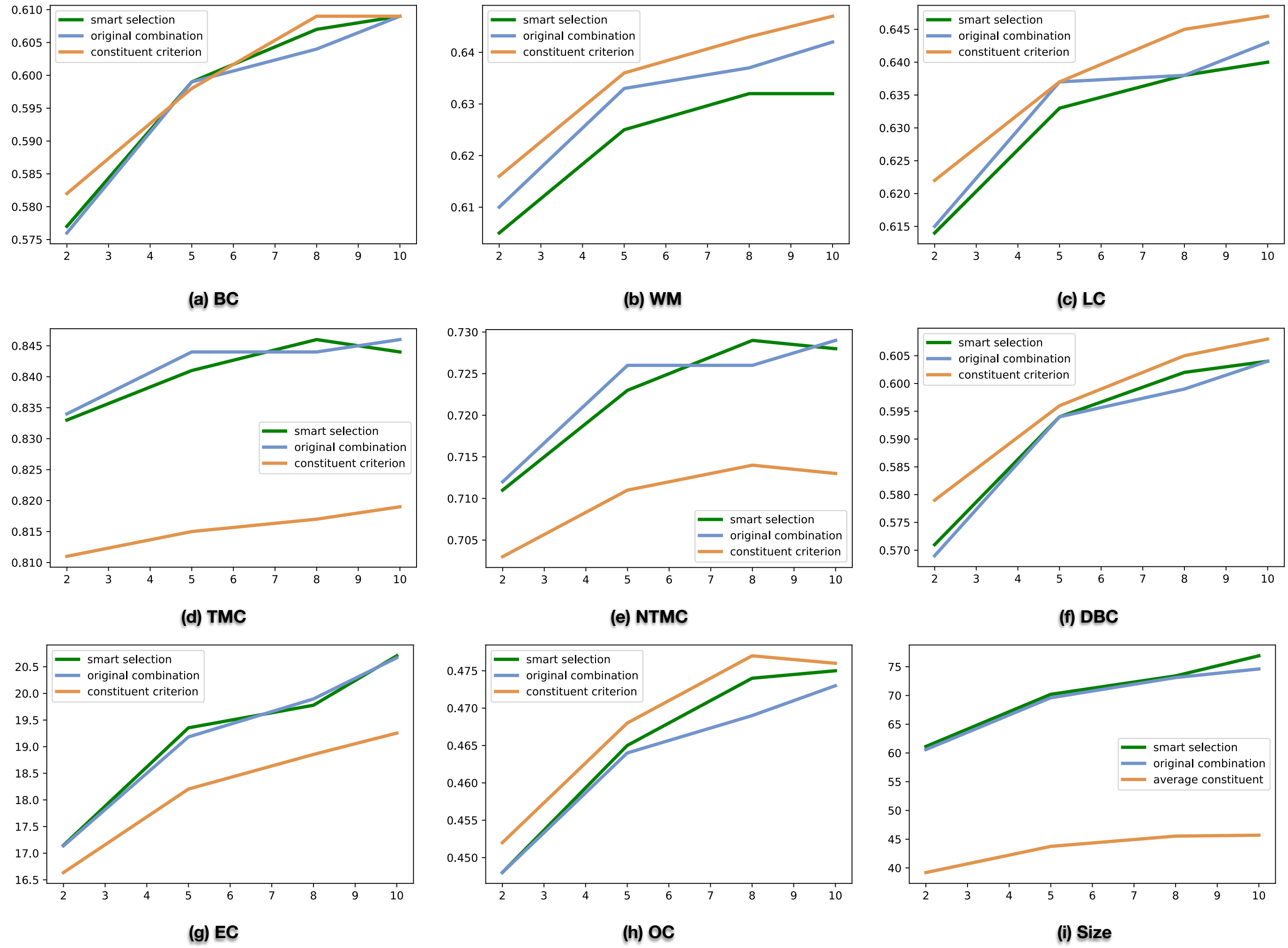
Size. Fig. 12 (i) shows the suite size change. The gaps between the three strategies are stable from 2 to 10 minutes. For all budgets, smart selection generates an average of nearly 5 more tests than the original combination. The original combination generates 19 more tests than the average constituent criterion.

Analysis. For most criteria, smart selection outperforms the original combination no matter how the budget changes. The situations of comparing the constituent criterion and the others vary on different criteria.

①**BC-DBC-LC-WM.** Firstly, the data of LC and WM shows that the coverage of smart selection and the constituent criterion is close and outperforms that of the original combination. This fact confirms the two fundamental assumptions of our methodology: (1) The negative impact of increasing the optimization goals of LC and WM on the algorithm WS exists, even if the budget is increased to 10 minutes; (2) The representation of BC/DBC to LC and WM is significant. After smart selection removes most of the goals of LC and WM, their coverage is stably higher than that of the original combination. Secondly, the data of BC and DBC shows that the constituent criterion outperforms smart selection and smart selection outperforms the original combination. This fact indicates that the other remaining fitness functions (e.g., the ones of OC) still hinder BC/DBC's fitness functions, undermining their effectiveness in guiding WS.

②**TMC-NTMC.** Firstly, although the coverage of the three strategies is not the same, the differences between them are tiny compared to group 1. Secondly, this group has significantly higher coverage than the other groups. For example, the lowest coverage of NTMC is 0.7 (see Fig. 12 (e)), but the highest coverage of BC is nearly 0.6 (see Fig. 12 (a)). These facts indicate that the goals of TMC/NTMC are easier to be covered than the others since they only require that a method is invoked (without exceptions). As a result, their coverage is more robust when combining multiple criteria.

③④**EC and OC.** For coverage of EC and OC, the difference between the constituent criterion and smart selection is tiny,



(a) BC (b) WM (c) LC (d) TMC (e) NTMC (f) DBC (g) EC (h) OC (i) Size

Fig. 14: The coverage and size of three strategies on DynaMOSA under the various budgets (2-10 minutes).

and they outperform the original combination. Hence, combining multiple criteria negatively impacts their coverage, although their own fitness functions are weak in guiding genetic algorithms.

Size. Firstly, with the budget and coverage increase, the three strategies' suite sizes also increase. Secondly, the sizes of the constituent criterion are much smaller than the sizes of the other two since the latter two strategies need to generate more tests for more goals brought by combining multiple criteria.

►MOSA.

Fig. 13 shows the average coverage and suite sizes of three strategies on MOSA under the various budgets. Similar to WS, we present data by groups.

①**BC-DBC-LC-WM.** Fig. 13 (a), (b), (c), and (f) show the coverage change of BC, WM, LC, and DBC. As the budget increases, (1) for BC and DBC, the coverage of smart selection and the constituent criterion is close and outperforms slightly that of the original combination; (2) For LC, the coverage gaps between the three strategies are tiny; (2) For WM, the coverage gap between smart selection and the other two gradually increases. The coverage of smart selection is nearly 1% behind when the budget is 10 minutes.

②**TMC-NTMC.** Fig. 13 (d) and (e) show the coverage change of TMC and NTMC. As the budget increases, (1) the coverage increase of the three strategies is tiny. For instance, from 2 to 10 minutes, the TMC coverage increase of smart selection/the original combination/the constituent criterion is $1.3\%/1.7\%/1.2\%$; (2) The coverage of smart selection and the original combination is close and stably outperforms that of the constituent criterion.

③④**EC and OC.** Fig. 13 (g) and (h) show the coverage change of EC and OC. As the budget increases, for EC, the coverage of the original combination and the constituent criterion is close and smart selection outperforms them by nearly one exception. For OC, the constituent criterion outperforms smart selection, and smart selection outperforms the original combination.

Size. Fig. 13 (i) shows the suite size change. For all budgets, smart selection generates an average of nearly 4 more tests than the original combination. The original combination generates 25 more tests than the average constituent criterion.

Analysis. Similar to WS, for most criteria, smart selection outperforms the original combination no matter how the budget changes. The situations of comparing the constituent criterion and the others vary on different criteria.

①**BC-DBC-LC-WM.** Firstly, the data of BC and DBC shows that the coverage of smart selection and the constituent criterion outperforms that of the original combination, even when the budget increases to 10 minutes. This fact confirms our findings in RQ2 (see Sec. 4.3): Too many objectives also affect the multi-objective algorithms, e.g., MOSA. Secondly, the data of WM shows that as the budget increases, the coverage of the original combination and the constituent criterion is close, and the lag of smart selection is increasing. On the contrary, the data of LC shows that The coverage lead of smart selection is obvious under most budgets. These facts confirm our findings in RQ6 (see 4.7): The coverage correlation between (direct) branch coverage and line coverage is stronger than the one between (direct) branch coverage and weak mutation.

②**TMC-NTMC.** Firstly, the coverage differences between smart selection and the original combination are tiny. This fact confirms that the coverage of TMC and NTMC is robust when combing multiple criteria. Secondly, smart selection and the original combination stably outperform the constituent criterion as the budget increases. This could be due to the weak guidance of TMC and NTMC's fitness functions since such a fitness function can only tell MOSA whether a method is covered, downgrading MOSA to the random search. As a result, when combining fitness functions with better guidance (e.g., ones of BC/DBC), MOSA has a chance to cover more methods.

③④**EC and OC.** For both EC and OC, smart selection outperforms the original combination, confirming that combining more criteria hurts their coverage. Secondly, for EC, smart selection outperforms the constituent criterion, contrary to the case of OC. This fact indicates that the fitness function of OC is better than that of EC in guiding GAs.

Size. Likewise, the three strategies' suite sizes also increase as the budget and coverage increase. Secondly, the test sizes of the constituent criterion are much smaller than the other two since more criteria call for more tests.

►**DynaMOSA.**

Fig. 14 shows the average coverage and suite sizes of three strategies on MOSA under the various budgets. Similar to WS and MOSA, we present data by groups.

①**BC-DBC-LC-WM.** Fig. 14 (a), (b), (c), and (f) show the coverage change of BC, WM, LC, and DBC. As the budget increases, (1) for BC and DBC, the coverage of three strategies is close; (2) For LC and WM, the constituent criterion outperforms the original combination, and the original combination outperforms smart selection. Note that the advantage of the constituent criterion/the original combination over the original combination/smart selection in LC is tinier than in WM.

②**TMC-NTMC.** Fig. 14 (d) and (e) show the coverage change of TMC and NTMC. Like MOSA, (1) the coverage increase of the three strategies is tiny; and (2) the coverage of smart selection and the original combination is close and outperforms that of the constituent criterion.

③④**EC and OC.** Fig. 14 (g) and (h) show the coverage change of EC and OC. For EC, the performances of smart selection and the original combination are close and are better than that of the constituent criterion by nearly one exception. For OC, the performances of the three strategies keep close, and the constituent criterion/smart selection maintains a slight advantage (nearly $0.3\%/0.2\%$) over smart selection/the original combination.

Size. Fig. 14 (i) shows the suite size change. The sizes of generated tests by smart selection and the original combination are nearly equal for all budgets. They generate nearly 25 more tests than the average constituent criterion.

Analysis. Compared to WS and MOSA, the comparing situation of the three strategies on DynaMOSA is different. For most criteria, the performance of the original combination is close to or slightly better than the other two strategies, showing the better robustness of DynaMOSA over WS and MOSA when facing combining multiple criteria.

①**BC-DBC-LC-WM.** Firstly, the data of BC and DBC shows that the coverage of the three strategies is close. Secondly, the data of WM and LC shows that the constituent criterion outperforms the original combination, and the original combination outperforms smart selection. These facts confirm our findings in RQ3 (see Sec. 4.4): DynaMOSA is more robust in handling combining multiple criteria. Especially the fact that the original combination outperforms smart selection for WM and LC shows that the fitness functions of LC and WM removed by smart selection but kept by the original combination help DynaMOSA to cover the corresponding goals of LC and WM. Furthermore, we notice that the constituent criterion outperforms the original combination for WM and LC. This fact indicates that when the fitness functions of a criterion (e.g., WM and LC) have weak guidance for GAs, combining multiple criteria still harms DynaMOSA, which is potential because more criteria lower the search weight of each fitness function.

②**TMC-NTMC.** Like MOSA, the coverage gaps between smart selection and the original combination are tiny. Secondly, smart selection and the original combination stably outperform the constituent criterion as the budget increases. This could be due to the fact that a fitness function of TMC/NTMC can only tell GAs whether a method is covered, downgrading them to the random search. As a result, when combining fitness functions with better guidance (e.g., ones of BC/DBC), DynaMOSA has a chance to cover more methods.

③④**EC and OC.** Firstly, the situation of EC is like TMC/NTMC: the performances of smart selection and the original combination are close and better than the constituent criterion, which is potential because, similar to TMC/NTMC, EC's fitness functions can only tell GAs whether a method is covered, downgrading them to the random search. Combining multiple criteria brings a more extensive search space for DynaMOSA, increasing its possibility of catching exceptions, e.g., killing a mutant may trigger an exception. Secondly, the situation of OC is like BC/DBC: the coverage of the three strategies is close, confirming the better robustness of DynaMOSA over WS and MOSA when facing more coverage criteria.

Size. Like WS and MOSA, the three strategies' suite sizes also increase as the budget and coverage increase. Secondly, the test sizes of the constituent criterion are much smaller than the other two since more criteria call for more tests. Furthermore, DynaMOSA's size gap between smart selection and the original combination is smaller than that of WS and MOSA since the coverage of these two strategies is close, showing DynaMOSA's robustness again.

Answer to RQ7

From 2 to 10 minutes:

1) **WS:** For most criteria, the coverage rank of the three strategies is (1) CC, (2) SS, and (3) OC.
2) **MOSA:** Like WS, for most criteria, OC obtains the lowest coverage.
3) **DynaMOSA:** For most criteria, the coverage of the three strategies is close.

5 Discussion

5.1 Parameter Tuning

Smart selection introduces a new parameter: *lineThreshold* (see Sec. 3.4). In handling line coverage, smart selection skips those basic blocks (BBs) with lines less than *lineThreshold*. The larger the value of this parameter, the more BBs we skip. Without considering the dead code, (direct) branch coverage fails to capture the following lines only when a certain line in a basic block exits abnormally. Previous work [4] shows that, on average, when 78% of branches are covered, test suites can only find 1.75 exceptions. It indicates that (direct) branch coverage can capture most properties of line coverage. Therefore, to minimize the impacts of line coverage goals on SBST, we prefer a larger *lineThreshold*. After statistics on the benchmarks used in DynaMOSA [3], we find that 50% of the BBs have less than 8 lines. Therefore, we set *lineThreshold* to 8.

5.2 Threats to Validity

The threat to external validity comes from the experimental subjects. We choose 158 Java classes from the benchmark of DynaMOSA [3]. [3] was published in 2018. Many classes have already become obsolete. Some projects even are no longer maintained [24]. To reduce this risk, we choose 242 classes at random from Hadoop [33], thereby increasing the diversity of the dataset. The threat to internal validity comes from the randomness of the genetic algorithms. To reduce the risk, we repeat each approach 30 (10/10/5) times for every class when the search budget is 2 (5/8/10) minutes.

6 Related Work

In this section, we introduce related studies on (1) SBST and (2) coverage criteria combination in SBST.

SBST. SBST formulates test cases generation as an optimization problem. Miller et al. [38] proposed the first SBST technique to generate test data for functions with inputs of float type. SBST techniques have been widely used in various objects under test [9], [39], [40], [41], [42], [43], [44], [45], [46], and types of software testing [47], [48], [49]. Most researchers focus on (1) search algorithms: Tonella [50] proposed to iterate to generate one test case for each branch. Fraser et al. [1] proposed to generate a test suite for all branches. Panichella et al. [2], [3], [51] introduced many-objective optimization algorithms; (2) fitness gradients recovery: Lin et al. [25] proposed an approach to address the inter-procedural flag problem. Lin et al. [24] proposed a test seed synthesis approach to create complex test inputs. Arcuri et al. [52] integrated testability transformations into API tests. Braione et al. [53] combined symbolic execution and SBST for programs with complex inputs; (3) readability of generated tests: Daka et al. [54] proposed to assign names for tests by summarizing covered coverage goals. Roy et al. [55] introduced deep learning approaches to generate test names; (4) fitness function design: Xu et al. [56] proposed an adaptive fitness function for improving SBST. Rojas et al. [4] proposed to combine multiple criteria to satisfy users' requirements.

Coverage Criteria Combination in SBST. After the work of Rojas et al. [4], Gregory Gay [10] experimented with different combinations of coverage criteria to compare the effectiveness of multi-criteria suites in detecting complex, real-world faults. Omur et al. [57] introduced the Artificial Bee Colony algorithm as a substitute for the GAs used in WS [1]. Our work aims to increase the coverage decrease caused by combining criteria [4] and is orthogonal to the latter two studies [10], [57].

7 Conclusion

We propose smart selection to address the coverage decrease caused by combining multiple criteria in SBST. We compare smart selection with the original combination on 400 Java classes. The experiment results confirm that with WS and MOSA, smart selection outperforms the original combination, especially for the Java classes with no fewer than 200 branches. However, with DynaMOSA, the differences between the two approaches are slight. Additionally, we conduct experiments to confirm our assumptions about coverage criteria relationships. We also experiment with different budgets, still showing the advantage of smart selection over the original combination.

Acknowledgements

Z. Zhou and Y. Tang are partially sponsored by Shanghai Pujiang Program (No. 21PJ1410700) and National Natural Science Foundation of China (No. 62202306). Y. Zhou is partially sponsored by National Natural Science Foundation of China (No. 62172205). C. Fang and Z. Chen are partially sponsored by Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

References

[1] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[2] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Proceedings of ICST*, 2015, pp. 1–10.

[3] ——, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, pp. 122–158, 2018.

[4] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds., 2015, pp. 93–108.

[5] J. D. Knowles, R. A. Watson, and D. W. Corne, "Reducing local optima in single-objective problems by multi-objectivization," in *International conference on evolutionary multi-criterion optimization*, 2001, pp. 269–283.

[6] D. Brockhoff, T. Friedrich, N. Hebbinghaus, C. Klein, F. Neumann, and E. Zitzler, "Do additional objectives make a problem harder?" in *Proceedings of GECCO*, 2007, pp. 765–772.

[7] Z. Zhou, Y. Zhou, C. Fang, Z. Chen, and Y. Tang, "Selectively combining multiple coverage goals in search-based unit test generation," in *Proceedings of ASE*, 2022, p. 91:1–91:12.

[8] "scipy.stats.pearsonr - scipy v1.11.1 manual," accessed: 2023-08-09. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html

[9] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of ESEC/FSE*, 2011, p. 416–419.

[10] G. Gay, "Generating effective test suites by combining coverage criteria," in *Search Based Software Engineering*, 2017, pp. 65–82.

[11] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[12] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[13] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?" in *Proceedings of ICSE*. IEEE, 2021, pp. 910–921.

[14] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, p. 783–812, 2015.

[15] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[16] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, "Evosuite at the sbst 2021 tool competition," in *Proceedings of SBST*, 2021, pp. 28–29.

[17] A. Panichella, J. Campos, and G. Fraser, "Evosuite at the sbst 2020 tool competition," in *Proceedings of ICSEW*, 2020, p. 549–552.

[18] J. Campos, A. Panichella, and G. Fraser, "Evosuite at the sbst 2019 tool competition," in *Proceedings of SBST*, 2019, pp. 29–32.

[19] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2011.

[20] A. Seshadri, "A fast elitist multi-objective genetic algorithm: Nsga-ii," *MATLAB Central*, vol. 182, 2006.

[21] K. Deb, *Multi-objective optimization*, 2014, pp. 403–449.

[22] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–33, 2015.

[23] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[24] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-based seed object synthesis for search-based unit testing," in *Proceedings of ESEC/FSE*, 2021, p. 1068–1080.

[25] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering fitness gradients for interprocedural boolean flags in search-based testing," in *Proceedings of ISSTA*, 2020, pp. 440–451.

[26] R. Gheyi, M. Ribeiro, B. Souza, M. Guimarães, L. Fernandes, M. d'Amorim, V. Alves, L. Teixeira, and B. Fonseca, "Identifying method-level mutation subsumption relations using z3," *Information and Software Technology*, vol. 132, p. 106496, 2021.

[27] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[28] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999.

[29] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Proceedings of SCAM*, 2008, pp. 249 – 258.

[30] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *Proceedings of ICST*, 2012, pp. 720–725.

[31] M. A. Guimarães, L. Fernandes, M. Ribeiro, M. d'Amorim, and R. Gheyi, "Optimizing mutation testing by discovering dynamic mutant subsumption relations," in *Proceedings of ICST*, 2020, pp. 198–208.

[32] B. Souza, "Identifying mutation subsumption relations," in *Proceedings of ASE*, 2020, pp. 1388–1390.

[33] Apache Software Foundation, "Hadoop," 2006. [Online]. Available: https://hadoop.apache.org

[34] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[35] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," *Noise reduction in speech processing*, pp. 1–4, 2009.

[36] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[37] Apache Software Foundation, "Hadoop's compare method," 2006. [Online]. Available: https://tinyurl.com/mrxvxusy

[38] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, no. 3, pp. 223–226, 1976.

[39] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *Proceedings of ICST*, 2018, pp. 394–397.

[40] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for sql queries," in *Proceedings of ICSE*, 2018, pp. 1220–1230.

[41] A. Gambi, M. Mueller, and G. Fraser, "Automatically testing self-driving cars with search-based procedural content generation," in *Proceedings of ISSTA*, 2019, pp. 318–328.

[42] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1548–1560.

[43] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu, "All your app links are belong to us: Understanding the threats of instant apps based attacks," in *Proceedings of ESEC/FSE*, 2020, p. 914–926.

[44] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of ICSE*, 2020, pp. 481–492.

[45] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: automated black-box testing of restful web apis," in *Proceedings of ISSTA*, 2021, pp. 682–685.

[46] F. U. Haq, D. Shin, L. C. Briand, T. Stifter, and J. Wang, "Automatic test suite generation for key-points detection dnns using many-objective search (experience paper)," in *Proceedings of ISSTA*, 2021, pp. 91–102.

[47] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[48] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, pp. 19–35, 2017.

[49] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proceedings of ISSTA*, 2006, pp. 1–12.

[50] P. Tonella, "Evolutionary testing of classes," in *Proceedings of ISSTA*, 2004, p. 119–128.

[51] G. Grano, C. Laaber, A. Panichella, and S. Panichella, "Testing with fewer resources: An adaptive approach to performance-aware test case generation," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2332–2347, 2019.

[52] A. Arcuri and J. P. Galeotti, "Enhancing search-based testing with testability transformations for existing apis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–34, 2021.

[53] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proceedings of ISSTA*, 2017, pp. 90–101.

[54] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of ISSTA*, 2017, pp. 57–67.

[55] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "Deeptc-enhancer: Improving the readability of automatically generated tests," in *Proceedings of ASE*, 2020, pp. 287–298.

[56] X. Xu, Z. Zhu, and L. Jiao, "An adaptive fitness function based on branch hardness for search based testing," in *Proceedings of GECCO*, 2017, pp. 1335–1342.

[57] O. Sahin, B. Akay, and D. Karaboga, "Archive-based multi-criteria artificial bee colony algorithm for whole test suite generation," *Engineering Science and Technology*, vol. 24, no. 3, pp. 806–817, 2021.