Information and Software Technology 104 (2018) 236–256

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A large scale empirical comparison of state-of-the-art search-based test case generators

Annibale Panichella[a], Fitsum Meshesha Kifetew[*,b], Paolo Tonella[c]

[a] Delft University of Technology, Netherlands
[b] Fondazione Bruno Kessler, Italy
[c] Università della Svizzera Italiana(USI), Switzerland

ARTICLE INFO

Keywords:
Test case generation
Search-based testing
Large-scale evaluation

ABSTRACT

Context: Replication studies and experiments form an important foundation in advancing scientific research. While their prevalence in Software Engineering is increasing, there is still more to be done.
Objective: This article aims to extend our previous replication study on search-based test generation techniques by performing a large-scale empirical comparison with further techniques from the state of the art.
Method: We designed a comprehensive experimental study involving six techniques, a benchmark composed of 180 non-trivial Java classes, and a total of 21,600 independent executions. Metrics regarding effectiveness and efficiency of the techniques were collected and analyzed by means of statistical methods.
Results: Our empirical study shows that single target approaches are generally outperformed by multi-target approaches, while within the multi-target approaches, DynaMOSA/MOSA, which are based on many-objective optimization, outperform the others, in particular for complex classes.
Conclusion: The results obtained from our large-scale empirical investigation confirm what has been reported in previous studies, while also highlighting striking differences and novel observations. Future studies, on different benchmarks and considering additional techniques, could further reinforce and extend our findings.

1. Introduction

The progress made in recent years in the area of automated test generation has resulted in a number of techniques and tools that are able to achieve significant levels of code coverage. In particular, search-based software testing (SBST) has seen an increasing number of competitive, alternative techniques, as well as the creation of a testing tool competition [1–3]. A consequence of such a strong momentum is that the experimental results reported in the respective scientific articles, documenting the techniques and tools, need to be replicated and enhanced by the community. Both replication studies and large-scale empirical evaluations play an important role in advancing scientific research. Replication studies increase confidence in experimental results by evaluating reported techniques in different scenarios than the original settings [4]. Furthermore, large-scale empirical studies help to address threats to external validity, such as the generalizability of the reported results [5].

In the context of white-box unit-testing, various search-based techniques have been proposed and empirically evaluated over the years [5–12]. The earliest search-based strategy for test generation is the single-target approach, which attempts to satisfy one coverage target (e.g., branch) at a time by using search algorithms [6,7,10–12]. More recent approaches consider all coverage targets (e.g., all branches) at the same time by either evolving test suites rather than test cases (whole-suite approach) or by relying on many-objective algorithms. The former uses an aggregate fitness function that combines the distance functions of all yet uncovered targets [5,8,9] and is minimized by means of classical meta-heuristic (e.g., genetic) algorithms. The latter applies a truly many-objective search to the test generation problem [13–15].

Previous empirical studies by Fraser and Arcuri [8,16] showed that the whole-suite approach (WS) is able to cover more branches than the single-target approach. However, there are very few cases (classes) in which the single-target approach can cover branches that are left uncovered by WS. In our previous work [13], we introduced a different strategy to target all branches at once, relying on a many-objective genetic algorithm, named MOSA (Many-Objective Sorting Algorithm), that regards all uncovered branches as independent objectives to optimize. Our empirical study showed that MOSA can often achieve higher branch coverage than WS [13].

* Corresponding author.
E-mail addresses: a.panichella@tudelft.nl (A. Panichella), kifetew@fbk.eu (F.M. Kifetew), paolo.tonella@usi.ch (P. Tonella).

https://doi.org/10.1016/j.infsof.2018.08.009
Received 1 February 2018; Received in revised form 17 June 2018; Accepted 17 August 2018
Available online 21 August 2018

While most recent effort has been devoted to improving multi-target approaches [8,13,16–18], a novel single-target approach has been proposed in 2016 by Scalabrino et al. [19]. Their study describes an empirical comparison between LIPS (Linearly Independent Path-based Search) and MOSA [19], both re-implemented in a tool called OCELOT. According to their results, LIPS can achieve higher coverage and is more efficient than MOSA [19]. Their empirical investigation is particularly interesting because it tries to further investigate the advantages and disadvantages of adopting a single-target vs. many-objective approach.

Since comparing single- vs. multi-target approaches is a key research question in search-based testing, in our conference paper we decided to replicate and extend the empirical study by Scalabrino et al. [19]. The results of our replication study [20] were published and presented at the 9th annual Symposium on Search Based Software Engineering (SSBSE'17) held in Paderborn, Germany. Our aim was to replicate the comparative experimental results reported by Scalabrino et al. [19]. The outcome of the replication study turned out to be quite different from what was reported in the original study [20].

While our replication study highlighted several open challenges of single-target approaches, it compared only LIPS and MOSA, leaving several important questions unaddressed: *Does LIPS overcome some of the limitations of the classical single-target approach? More generally, how do LIPS and MOSA perform compared to existing search-based single/multi-target approaches?* Our conference paper focused on generating test data for Java static methods with numerical inputs [20] to replicate the original study [19], which targeted procedural, stateless methods. Yet, another important unaddressed question is: *How do LIPS and MOSA perform when testing stateful Java classes, rather than single static/stateless methods?* Generating test cases for Java classes requires the synthesis of method sequences in addition to the generation of input parameters for each method call [21]: *How do LIPS and MOSA perform when generating non-numerical test inputs?*

Another motivation for the present empirical work is that further new search-based techniques have been published in the last two years [14,15,22–24]. For example, Rojas et al. [22] introduced the whole suite with Archive approach (WSA), which is a variant of WS that (1) incorporates the *archive* strategy of MOSA, and (2) updates the fitness computation by removing already covered targets from the fitness function computation. A large-scale empirical study showed that WSA outperforms both its previous variant WS and the classical single-target approach [22]. In our recent work, we also further extended MOSA by introducing DynaMOSA [14], a many-objective genetic algorithm which dynamically updates the subset of uncovered branches being considered as search objectives. In DynaMOSA, coverage targets are selected as objectives depending on their position in the control dependency hierarchy. This variant has been proven to be more effective and efficient than its predecessors MOSA and WSA. Finally, Arcuri [15,23] proposed a novel many-objective local search algorithm, named MIO, that generates system-level test cases. An empirical study with artificial programs and seven RESTful API web services showed that MIO has an overall superior performance compared to MOSA and WSA, although it is not the best on all problems [23].

These recent advances pose new research questions that were left unanswered in our previous replication study [20]. For example: *Does WSA outperform LIPS, which is the most recent single-target approach? Are the most recent multi-target algorithms superior to the single-target ones? How do different algorithms scale when generating test cases/suites for large classes?* Furthermore, the new search-based algorithms introduced in very recent papers were not fully compared either with any single-target approach (e.g., MIO vs. LIPS) or with other multi-target approaches (e.g., DynaMOSA vs. MIO): *How does MIO perform compared to other search-based algorithms for unit-testing? How does it perform compared to single-target approaches?* Answering these open questions requires a comprehensive large-scale empirical study to provide further evidence regarding the strengths and weaknesses of techniques and tools available for the generation of unit test cases.

In the current article, we expand our previous replication study by conducting a large-scale empirical comparison among further state-of-the-art SBST approaches with the purpose of shedding light on the aforementioned open questions. With respect to the conference version, this article makes the following main contributions:

- We re-implemented the MIO algorithm in EvoSuite by migrating and adapting its original source code from EvoMaster [23].
- We extend the techniques being compared from the two (LIPS and MOSA) in the conference version, to six techniques, by adding DynaMOSA, WSA, MIO, and Single-target;
- We extend the benchmark by including additional subjects, available from the recent literature. The new benchmark is one order of magnitude larger than the one used in our previous study and includes Java classes (instead of static-methods) with non-primitive input data types.

The remainder of this article is organized as follows: Section 2 presents the necessary background, in particular, it briefly introduces the techniques considered for the empirical study. Section 3 presents the design of the empirical study, and Section 4 presents the results of the study. Section 5 discusses closely related work. Finally, Section 6 concludes the paper and outlines potential future works.

2. Background

In unit testing, the goal is to devise a test suite that thoroughly exercises the class (or program) under test (CUT). The quality of the test suite is measured according to some adequacy criteria [6], e.g., *branch coverage* [21], *statement coverage* [6], *mutation score* [18], and *input/output diversity* [6]. In white-box testing, adequacy criteria correspond to coverage metrics measuring the percentage of code elements of the CUT (e.g., statements) that are exercised by a test suite. More generally, the test case generation problem (TCG) can be formulated as follows:

Definition 1. Let $\Omega = \{\omega_1, \cdots, \omega_m\}$ be the set of test targets/requirements (e.g., branches) to satisfy for a given CUT. Find a suite of test cases $T = \{t_1, \cdots, t_n\}$ that maximizes the number of test requirements in Ω that are satisfied.

In search-based software testing, test targets (or coverage targets) are encoded as distance (or fitness) functions to minimize. Such functions are obtained by applying well-established *heuristics* that measure how far a given test case t is from covering each target $\omega \in \Omega$. The heuristics vary depending on the chosen coverage criterion. For example, a well-known heuristic for branch and statement coverage is a combination of *approach level* [25] and *branch distance* [26]. The former heuristic measures the number of control dependencies that separate the execution trace of t from the target branch. The latter measures the distance to satisfying the conditional expression where the execution of t diverges from the path to the target ω. Further heuristics exist for other coverage criteria, such as *infection* and *propagation* distance for mutation coverage [18], *input* and *output* distance for input/output diversity [27].

Once the appropriate functions have been defined, search algorithms are then used as tools for generating test cases, by minimizing the fitness functions for the CUT, i.e., covering the corresponding test targets. The most widely-investigated category of search algorithms in TCG are global solvers, which include genetic algorithms [5,21], ant colony [28], and particle swarm optimization [29]. In addition, local solvers have been also investigated in the related literature, such as hill climbing [30], simulated annealing [31], and Korel's alternating variable method [26].

Let us consider the CUT depicted in Fig. 1. It contains four branches: two branches for the `if`-condition at line 1 and two further branches for the `if`-condition at line 3. Each branch has its own fitness function to optimize, according to the corresponding branch distance and approach

```
   Instructions
s  int example(int a, int b, int c)
   {
1  if (a == b)
2     return 1;
3  if (b == c)
4     return -1;
5  return 0;
   }
```

Fig. 1. Example program.

level. For example, the fitness function for the true branch of the `if` at line 1 is:

$$f(\omega_{\langle 1,2\rangle}) = approach\_level(\omega_{\langle 1,2\rangle}) + \frac{branch\_distance(\omega_{\langle 1,2\rangle})}{branch\_distance(\omega_{\langle 1,2\rangle}) + 1} \quad (1)$$

$$= \frac{|a-b|}{|a-b|+1} \quad (2)$$

Reaching 100% branch coverage for the CUT in Fig. 1 requires more than a single test case. This is because any test data covering the true branch of the first `if`-condition cannot also cover the true branch of the second `if`-condition, i.e., the two branches are in contrast with each other. An early attempt to address this problem is the so-called *one target at a time* approach: each branch is solved separately with a single run of a search algorithm (e.g., genetic algorithm). Therefore, multiple targets are solved through independent searches and the final suite is built by collecting the test cases produced across such different searches.

Despite its theoretical simplicity, this strategy comes with a number of limitations that are widely-described in the literature [5,13,14], such as (i) how to allocate the search budget for each target, (ii) how to handle infeasible branches, (iii) how to decide which target to solve first. For these reasons, researchers have proposed more sophisticated strategies to the TCG problem, each differing from the other on how multiple targets are handled during the search. Most recent advances include assigning priorities to the targets [19], applying a sum-scalarization approach [8] or many-objective solvers [13–15].

In this paper, we focus on the most recent works in TCG that propose novel strategies for handling multiple targets, i.e., alternatives to the classical *one target at a time* approach. In particular, we consider the following five techniques: the whole-suite approach [22], MOSA [13], DynaMOSA [14], LIPS [19] and MIO [15]. Table 1 summarizes the main characteristics of these techniques, including the chromosome definition, the underlying search algorithm and the number of targets (e.g., branches) they address directly in each search run. The next subsections provide a brief overview of these different approaches to the TCG problem.

Table 1
Characteristics of the different algorithms.

Algorithm name	No. of Targets	Chromosome	Search algorithm	Ref.
ONETARGET	One Branch	Test Case	Genetic Algorithms	[22]
LIPS	One Branch	Test Case	Genetic Algorithms	[19]
WS	All Branches	Test Suite	Genetic Algorithms	[8]
MOSA	All Branches	Test Case	Many-obj. Genetic Algorithms	[13]
DynaMOSA	All Branches	Test Case	Many-obj. Genetic Algorithms	[14]
MIO	All Branches	Test Case	Many-obj. Local Search	[15]

2.1. The traditional single-target approach

In the *single-target* (or one target at a time) approach, the overall search budget (either time or number of iterations) is divided among the test targets. Each target is assigned a portion of the overall budget (local budget) in which a search algorithm (e.g., genetic algorithm) is executed in an attempt to cover it. In the case of genetic algorithms, test cases (chromosomes) are iteratively evolved using genetic operators, i.e., *selection, crossover*, and *mutation*. In each independent search, only one fitness function is optimized and the search stops when either a zero-fitness value is reached (i.e., the corresponding test target is covered) or the local budget is consumed. The final test suite is therefore built by collecting the test cases that cover some targets and that come from one of the different, independent searches.

However, not all targets require the same search budget: some targets may require more time than others depending on how difficult it is to synthesize test cases that cover them. Furthermore, the CUT may contain infeasible targets, which represent a waste of search effort as their local budget could be better spent optimizing feasible targets. Therefore, the order in which targets are selected for the search might strongly impact the final coverage. Unfortunately, it is impossible to determine a priori neither the branches which are infeasible, nor those feasible but most difficult to cover. Therefore, in the literature, no order is specified (e.g., [30]) and targets are usually selected in a random order.

Another important weakness of the *single-target* approach is related to collateral coverage, the phenomenon in which test cases generated when solving a given target may accidentally cover other targets. As demonstrated by Arcuri et al. [32], if collateral coverage is not properly detected, search algorithms applied within the *single-target* approach are asymptotically equivalent to or worse than pure random search.

2.2. Linearly independent path-based search (LIPS)

LIPS is a variant of the traditional single-target approach recently proposed by Scalabrino et al. [19] for branch coverage. It relies on genetic algorithms to optimize (cover) one branch (target) at a time. The key contributions of LIPS over the standard single-target approach are: (i) dynamic budget allocation; (ii) the order of the branches to optimize; (iii) seeding tests from previous GA runs.

LIPS starts with an initial, randomly generated test case tc_0, which is executed against the CUT. The uncovered branches of all decision nodes in the execution path of tc_0 are added to a *worklist* in the order in which they are encountered. Thus, the worklist is an ordered queue containing the branches that can be potentially considered as search targets. The overall search budget is uniformly divided across all branches in the queue and the first target to optimize is the last branch added to the worklist. Then, an initial population that includes tc_0 is randomly generated and further evolved using a genetic algorithm (GA), according to the fitness function corresponding to the current target [19].

Whenever a newly generated test *tc* covers the current target, it is added to the final test suite, and all the uncovered branches of the decision points in the path covered by *tc* are added to the worklist. The worklist is also updated in case of collateral coverage: branches in the worklist that are covered by chance in each GA iteration are removed from the worklist and the corresponding covering test cases are added to the test suite. Since the size of the worklist changes over time, the budget allocated to each (yet to cover) target is dynamically updated after each GA iteration and is re-computed using the formula SB/n [19], where SB is the budget that remained available after last target selection and n is the number of remaining uncovered branches that were never selected before.

If the current target is not covered within the allocated local budget SB/n, the search stops and the uncovered target is added at the front of the worklist, so that it will be selected again only when all the other branches in the worklist are covered, using the residual (if any) search

budget. Then, a new target is selected from the worklist and GA is re-started by using the last population from the previous GA run (seeding strategy). The population is therefore evolved based on the fitness function of the newly selected branch. The overall search terminates when all the branches are covered (i.e., the worklist is empty) or the total search budget is consumed [19].

Thanks to the dynamic budget allocation and the worklist, infeasible or difficult targets do not consume the overall search budget, because they are allocated only a fraction of the entire search budget. Moreover, even if they are not covered in the generation cycle allocated to them, they remain still coverable in successive generations thanks to collateral coverage (i.e., coverage achieved by a test case generated for a different target) or in case some residual budget remains at the end, due to easy to cover targets considered late in the process.

Originally, LIPS is defined for procedural programs written in C and for branch coverage only. Moreover, the length of the chromosome used by LIPS is fixed, which means that data structures with variable size (e.g., arrays) are assigned a predefined, fixed size. This may prevent coverage of targets requiring a specific, special value of size (e.g., a condition that checks if an array has size zero). In the conference version of our paper [20], we further improved LIPS for testing object-oriented programs. In particular, we described how to address the problem of generating method sequences [21] and input data with variable length. Interested readers can find further details about LIPS in the original paper by Scalabrino et al. [19]. The pseudo-code of LIPS is available in our prior conference paper [20].

2.3. Many-Objective Sorting Algorithm (MOSA)

MOSA is a many-objective genetic algorithm that we proposed in our prior work [13] for Java classes and implemented in EvoSuite[1]. A test case in MOSA is a method sequence (including input data) of variable length, which is evaluated against all uncovered branches. MOSA targets all uncovered branches at once by considering them as different (many) objectives to be optimized in parallel. It shares the same main loop with NSGA-II [33], which is one of the most popular multi-objective genetic algorithms. However, it differs on three key aspects: (i) it selects test cases according to a preference criterion suitably defined for the test case generation problem; (ii) it considers as objectives only the yet uncovered targets (i.e., the set of optimization objectives changes across generations); (iii) it uses an archive to store all test cases satisfying one or more previously uncovered branches. The pseudo-code of MOSA is provided in our previous papers [13,14,20].

As any other GA, MOSA starts with an initial randomly-generated population. Tests in a given population are combined to create new tests (*offspring*) via *crossover* and *mutation*. Then, parents and offspring are selected to form the next generation according to their ranks, as determined by the preference-based sorting algorithms. The preference criterion assigns the highest priority to test cases that are closest to one of the uncovered branches, according to the corresponding branch distance and approach level scores. When there are multiple test cases with the same objective scores, the preference criterion uses the test case length as secondary selection criterion [13], i.e., shorter tests are preferred. In the sorting, the best test cases according to the preference criterion are assigned a rank $R = 0$, while the remaining tests are ranked based on the criteria of the non-dominated sorting algorithm NSGA-II. Then, the new population for the next generation is formed by selecting the top N tests having the smallest ranking. During the selection, more diverse test cases are given higher priority compared to the other tests with the same ranking. The diversity is measured using the *crowding distance*, as defined by Deb et al. for NSGA-II [33].

Finally, MOSA uses an *archive*, to keep track of the shorter test cases that cover the branches of the program under test. Whenever new test cases are generated (either at the beginning of the search or when creating an offspring), MOSA stores those tests that cover previously uncovered targets in the *archive*, as candidates to form the final test suite.

2.4. Dynamic Many-Objective Sorting Algorithm (DynaMOSA)

DynaMOSA [14], is an enhanced variant of MOSA designed to improve its scalability for large CUTs. The difference between DynaMOSA and MOSA concerns the number of objectives that are optimized in each generation. In MOSA, the set of objectives to optimize corresponds the full set of test targets Ω. This set is incrementally reduced as soon as some targets are covered. Instead, DynaMOSA uses the *control dependency graph* (CDG) to determine which targets can be covered only after satisfying other, previous yet uncovered targets in the graph and which ones are free of control dependencies from yet uncovered targets.

At the beginning of the search, DynaMOSA selects as initial set of objectives only those targets that are free of control dependencies (e.g., root branches), which is typically a (small) subset of all the targets, i.e., $\Omega^* \subseteq \Omega$. Therefore, test cases are evaluated only according to the objectives in Ω^*. They are ranked using the *preference sorting algorithm* and the *crowding distance*, as done in MOSA. Then, the set of objectives to optimize Ω^* is *dynamically updated* in each generation according to the test execution results and the CDG. When an objective $\omega_i \in \Omega^*$ is covered by a newly generated test t, ω_i is removed from Ω^* and t is saved into the archive. Furthermore, all uncovered targets that are control dependent on ω_i are added to Ω^*.

In this way, the ranking performed by DynaMOSA is identical to the one in MOSA (a formal proof of this is provided in [14]), while convergence to the final test suite is achieved faster since the number of objectives to be optimized concurrently is kept smaller. Intuitively, the ranking of MOSA is unaffected by the exclusion of targets that are controlled by uncovered targets because such excluded targets induce a ranking of the test cases which is identical to the one induced by the controlling nodes [14].

2.5. Whole-suite approach with archive strategy (WSA)

The *whole-suite* approach (WS) was the first alternative to the single-target approach. It was proposed by Fraser and Arcuri [5,9]. Instead of evolving individual test cases, WS evolves entire test suites using a genetic algorithm. A test suite consists of a variable number of individual test cases, thus allowing the optimization of all targets simultaneously. The fitness function for a test suite is obtained by summing-up the minimum distances between test cases in the suite and test targets. For branch coverage, the test suite level fitness is defined as follows [9]:

$$f(T) = |M| - |M_T| + \sum_{\omega \in \Omega} d(\omega, T) \tag{3}$$

where $d(\omega, T)$ denotes the minimum normalized branch distance among all test case in T for the target ω, $|M|$ is the total number of methods in the CUT, and $|M_T|$ is the number of methods executed by the test cases in T. This single fitness function allows the application of genetic algorithms, given some definition of genetic operators that work at both test case and test suite levels. For example, the mutation operator has (i) to mutate a test suite by adding, removing, or changing a test case, and (ii) to mutate a test case by adding, removing and changing statements. The final test suite is the one having the best fitness score from the last GA iteration. As explained in our previous work [14], the whole-suite approach corresponds to the *sum-scalarization* approach in multi-objective optimization.

Recently, Rojas et al. [22] proposed a more efficient variant of the whole-suite approach and based on the Archiving strategy (WSA). WSA implements a strategy that combines the original whole-suite approach

[1] https://github.com/EvoSuite/evosuite/tree/master/client/src/main/java/org/evosuite/ga/metaheuristics/mosa

with the use of an archive. The main loop remains unchanged: candidate suites are iteratively evolved using a genetic algorithm according to one single fitness function that sums up the contributions of the individual test cases. The first main difference between WS and WSA lies in the usage of an archive (as in MOSA) to store test cases covering one or more targets [22]. Therefore, the final test suite is no longer the best suite (individual) from the last generation of GA but it is composed of all test cases stored in the archive, possibly coming from different suites. A second important difference between WSA and WS regards the computation of the fitness function: in WSA the fitness function score is computed by considering only the uncovered targets (as in MOSA/DynaMOSA).

Rojas et al. [22] empirically demonstrated the superiority of WSA over the original WS. Therefore, in this paper, we consider the latest variant of the whole-suite approach with archive (WSA).

2.6. Many Independent Objects (MIO)

The Many Independent Objective (MIO) algorithm [15] is a novel multi-population evolutionary algorithm that has been recently proposed for system testing. MIO tries to address the scalability issues of test case generators in the case of complex programs with hundreds/thousands of branches, which are very common in the context of system or integration testing [15].

MIO keeps one population of tests for *each* testing target (e.g., branch). Test cases within the same population are ranked *exclusively* according to the fitness function for the target they are associated with. At the beginning of the search, all populations are empty and are iteratively filled with randomly generated tests until reaching a fixed population size. In each iteration, with a given probability, MIO either samples and mutates an existing test case (scenario 1) or generates a new test case at random (scenario 2). In the first scenario, a test case t is randomly sampled from one of the populations related to uncovered targets. Then, t is mutated and added to *all* the populations associated with uncovered targets (not only to the population it originally comes from); t is therefore evaluated and ranked independently in each population according to the corresponding fitness function. In the second scenario, a new test case is randomly generated and therefore added to all populations for fitness evaluation and ranking.

To keep the populations size constant, each population P_ω with exceeding tests is shortened by removing the worst test based on the fitness for the corresponding target ω. In the original paper [15], such a threshold is set to ten test cases per population. Whenever a target ω is covered, the size of its population P_ω is shrunk to one and no more sampling is done from that population. At the end of the search, a test suite is created based on the best tests in each population.

The balance between *exploration* and *exploitation* is maintained in MIO with an adaptive strategy which dynamically updates the search parameters (e.g., sampling probability) during the search, similarly to Simulated Annealing [15]. MIO also uses *feedback-directed sampling* to effectively manage *infeasible branches*. For each population there is a counter, initialized to zero; every time an individual is sampled from a population P_ω, its counter is increased by one. Every time a new, better test is successfully added to P_ω, the counter for that population is reset to zero. When sampling a test from one of the populations, the population with the lowest counter is chosen. This helps focus the sampling on populations (one per testing target) for which there have been recent improvements in the achieved fitness value. This is particularly effective to prevent spending significant search time on (likely) infeasible targets [15].

Despite the name given to the algorithm, the targets are not treated as fully independent objectives in MIO. In fact, each newly generated test case (created either by mutating existing tests or randomly) is always evaluated against all uncovered targets (since it is copied in all populations), to handle the collateral coverage problem. Moreover, similarly to MOSA, DynaMOSA, and WSA, MIO always focuses the search on the uncovered targets (as tests are sampled from populations associated with uncovered targets).

While MIO has been originally evaluated for system testing, in this paper we evaluate it in the context of unit-level test case generation, given its high scalability for very complex programs [15]. Further details about our re-implementation of MIO within the EvoSuite framework are reported in Section 3.3.

3. Empirical evaluation

This section describes the large-scale empirical study we carried out to compare the six state-of-the-art approaches for TCG described in Section 2. Section 3.1 details the protocol we followed to select the CUTs. Section 3.2 presents the research questions and the performance metrics we use to answer them. Section 3.3 provides further information about the EvoSuite framework, the implementation of the alternative algorithms, and the parameter settings. Finally, Section 3.4 details the experimental procedure.

3.1. Benchmark

We carried out our evaluation on the SF110 corpus [5], which contains open-source projects from the `SourceForge.net` repository. The corpus contains 100 projects of different size, complexity and application domains. It also includes as additional projects the 10 most popular projects from the `SourceForge.net` repository. The corpus was constructed to be a statistically representative sample of open-source projects [5] and it has been widely-used in the literature to evaluate testing tools [5,14,18,22,34].

The computation time required to perform the evaluation on the entire SF110 corpus is potentially high because of (i) the very large number of classes in the corpus (23,886 Java classes in total), (ii) the number of repetitions required for statistical analysis, (iii) the number of TCG approaches to compare. Therefore, we randomly sampled non-trivial classes from the corpus as benchmark for our experiment. To this aim, we applied the same selection procedure used in previous studies [2,14], which filters the classes based on their McCabe's cyclomatic complexity [35]. The McCabe's cyclomatic complexity for a given Java method m is equal to the number of independent paths in its control flow graph, which also equates the number of branches in m plus one. Classes containing only branchless methods (having McCabe's complexity equal to one) are trivial because they can be fully covered with a single test case that just calls them all.

To challenge the TCG approaches considered in our evaluation, we excluded all non-trivial classes. We first pruned the SF110 corpus by filtering out all classes that contain exclusively methods with a cyclomatic complexity lower than five. This filtering is particularly required for SF110 since, as showed by Shamshiri et al. [36], the majority of classes in this corpus is trivial and does not require the usage of search-based techniques. From the filtered corpus, we then randomly sampled 175 classes. We further added the five largest classes (i.e., with the largest number of branches) to the corpus with the goal of assessing the performance of TCG approaches when dealing with very complex classes.

Table 2 provides some descriptive statistics about the projects and the classes in our benchmark, including the number of classes per project and the average number of branches per class. In total, our benchmark contains 180 classes with a total number of 34,949 branches. The number of branches ranges between 11 (for the class `ExtrasPatternParser` from the `ext4j` project) and 7938 (for the class `JavaParser` from the `jmca` project), while the median number of branches per class is 68.

3.2. Research questions and performance metrics

We investigate the following research questions:

- **RQ1**: *How do the different search-based algorithms perform in terms of effectiveness?*
- **RQ2**: *How do the different search-based algorithms perform over time?*

Table 2
Java projects and number of classes from SF110 selected in our study.

ID	Project name	No. of classes	# Branches		
			Min	Mean	Max
1	tullibee	2	20	21	21
2	a4j	2	31	78	125
4	rif	1	21	21	21
5	templateit	2	31	43	54
7	sfmis	2	32	57	82
10	water-simulator	1	137	137	137
11	imsmart	1	15	15	15
12	dsachat	2	69	84	98
13	jdbacl	2	188	193	197
14	omjstate	1	30	30	30
15	beanbin	2	41	44	47
17	inspirento	2	27	61	95
18	jsecurity	3	36	91	170
19	jmca	4	199	2515	7938
21	geo-google	2	23	27	30
22	byuic	4	39	357	739
24	saxpath	2	55	270	484
26	jipa	2	23	79	134
27	gangup	2	25	29	32
29	apbsmem	2	41	215	388
31	xisemele	1	27	27	27
32	httpanalyzer	2	56	128	200
33	javaviewcontrol	3	32	875	2373
35	corina	3	69	165	290
36	schemaspy	2	17	199	380
37	petsoar	2	16	32	47
38	javabullboard	2	98	120	141
39	diffi	2	20	28	35
40	glengineer	2	23	69	115
41	follow	2	20	28	35
42	asphodel	1	42	42	42
43	lilith	4	23	221	646
44	summa	2	30	201	372
45	lotus	2	24	26	28
46	nutzenportfolio	2	21	35	48
47	dvd-homevideo	3	29	55	84
49	diebierse	2	27	54	81
50	biff	1	817	817	817
51	jiprof	4	76	451	824
52	lagoon	2	63	64	65
54	db-everywhere	2	13	83	153
55	lavalamp	1	18	18	18
56	jhandballmoves	2	26	29	31
57	hft-bomberman	2	13	71	128
58	fps370	2	70	111	151
59	mygrid	2	28	28	28
60	sugar	2	24	38	51
61	noen	2	67	69	71
62	dom4j	2	73	247	420
63	objectexplorer	2	17	96	175
64	jtailgui	2	13	18	23
65	gsftp	2	29	60	91
66	openjms	2	34	67	99
67	gae-app-manager	1	47	47	47
68	biblestudy	2	32	35	37
69	lhamacaw	2	23	47	70
70	echodep	2	188	193	198
71	ext4j	2	11	75	139
72	battlecry	2	78	180	281
73	fim1	2	25	49	73
74	fixsuite	2	35	54	72
75	openhre	2	50	74	97
77	io-project	1	66	66	66
78	caloriecount	3	25	94	232
79	twfbplayer	2	74	115	156
80	wheelwebtool	4	40	272	817
81	javathena	4	49	206	329
82	ipcalculator	2	55	79	103
83	xbus	2	17	22	27
84	ifx-framework	1	72	72	72
85	shop	4	41	109	192
86	at-robots2-j	2	37	80	123
87	jaw-br	1	26	26	26
88	jopenchart	2	20	56	92
89	jiggler	1	33	33	33
100	jgaap	1	23	23	23
101	netweaver	2	69	71	73
102	squirrel-sql	2	21	36	51
103	sweethome3d	4	154	338	618
104	vuze	2	119	127	134
105	freemind	2	29	119	208
106	checkstyle	1	37	37	37
107	weka	4	29	338	809
108	liferay	1	78	78	78
109	pdfsam	2	19	30	41
110	firebird	3	98	444	1040

- **RQ3**: *Does the class size affect the performance of the different search algorithms?*

The first two research questions focus on a direct comparison of the performance of each of the six TCG approaches described in Section 2. In **RQ1**, we measure the effectiveness as the final percentage of branches covered by the different approaches, after a given, fixed search time. In **RQ2**, we compare their behaviors over time measured as the amount of coverage increase as the search time proceeds. Finally, **RQ3** investigates whether the performance of the TCG approaches varies with regards to the size (number of branches) of the class under tests. This analysis helps us better understand whether and in which measure the TCG approaches scale to complex classes.

3.3. Implementation and parameter setting

Three of the algorithms in our evaluation (i.e., single target approach, the whole-suite approach, and MOSA) are already implemented in EvoSuite [8,9] and publicly available on GitHub[2]. For DynaMOSA, we used its implementation available in our GitHub fork of the EvoSuite repository[3], which was also used in our previous paper [14]. For the remaining two algorithms, namely LIPS and MIO, we re-implemented them in EvoSuite as part of this large-scale empirical study. All algorithms are implemented in the same version of EvoSuite (v1.0.6), downloaded from GitHub on September 15th, 2017. The remainder of this section describes our re-implementation of LIPS and MIO and provides further details about the parameter setting.

Implementation of LIPS in EvoSuite. We have re-implemented LIPS in EvoSuite as described in our conference paper [20]. The main differences between the original version of LIPS [19] and our re-implementation regard the encoding schema and the genetic operators, for which we use the default test-case-level operators available in EvoSuite. In EvoSuite [8], a test case is a sequence of statements, which is composed of method calls (i.e., constructors, protected and public methods), variable declarations, and object instantiations (e.g., arrays, strings) and assignment statements. Test cases are bred with *single-point* crossover (for MOSA, DynaMOSA, and single-target approach), and changed with *uniform mutation* (for WSA, MOSA, and DynaMOSA). The mutation operator removes, changes, or adds statements from/in/to each test case. Therefore, the length of the test cases can vary during the search, so that the number of method calls, the length of input arrays, strings, etc., can change. Selection is done using *tournament selection*,

[2] https://github.com/EvoSuite/evosuite.
[3] https://github.com/apanichella/evosuite.

the same operator originally proposed for LIPS [19].

In the original LIPS implementation [19], the length of the chromosomes is fixed a priori, which might prevent coverage of specific branches. In addition, the original genetic operators [19] are *blend-crossover* (BLX) and *polynomial mutation*, which can be applied only to chromosomes with fixed length and containing only numerical values [37]. Since our re-implementation of LIPS in EvoSuite does not have such constraints, we deem it as superior to the original implementation and eventually able to cover more branches. In our conference paper [20], we showed empirically that our conjecture is true by comparing the results of our re-implementation with the results reported in the original study. Instead, the core novelties of LIPS, namely the order by which branches are selected as targets and the dynamic allocation of the search budget, are kept identical to the original formulation. Our re-implementation of LIPS is publicly available for download on GitHub[4].

Implementation of MIO in EvoSuite. We have re-implemented MIO in EvoSuite by adapting the original code available in EvoMaster[5] and written in Kotlin. The re-implementation was straightforward: we used the IntelliJ facilities to automatically convert Kotlin code to Java and then we adapted the code by introducing calls to the EvoSuite APIs. Fortunately, a semi-automated translation from Kotlin to Java is possible because programs written in these two programming languages run on the same Java virtual machine, after being compiled into bytecode. We also carefully reviewed the adapted code and verified that it strictly adheres to the original description of MIO [15] and to the original implementation available in EvoMaster.

Parameter setting. For the parameter settings, we adopted the default values used by EvoSuite and suggested by the related literature [5,14,15,20,27,34]:

Encoding Schema: The single-target approach, MOSA, DynaMOSA, MIO, and LIPS evolve test cases and, therefore, use the encoding scheme for test cases provided by EvoSuite. Differently, WSA evolves test suites and, thus, it uses the encoding scheme for test suites, not the one for test cases. In EvoSuite, a suite is a pool of test cases and each test case is a sequence of statements. The length of each test suite and of its composing test cases can vary across the generations.

Population Size: We used a fixed population size of 50 test cases for the single-target approach, LIPS, MOSA and DynaMOSA [14,20,34]; for WSA, the population is fixed to 50 test suites [5,34]; finally, in MIO the size of the population is set to 10 test cases, as suggested by the author of MIO [15].

Crossover probability: test cases (or test suites for WSA) are recombined using the *single-point crossover* with crossover probability $p_c = 0.75$ [5,14]. In MOSA, DynaMOSA, single-target, and LIPS, the *single-point crossover* generates two offspring test cases by exchanging statements between the two parent test cases; in WSA, offspring suites are generated by exchanging test cases between the two parent suites. This parameter is not applicable to MIO since it does not use any crossover operator.

Mutation probability: test cases (or test suites for WSA) are mutated using *uniform mutation* with mutation probability $p_m = 1/size$, where *size* is the number of statements contained in a given test case (or the number of test cases in a test suite) [34]. In WSA, suites are mutated by adding, removing and changing test cases. All other algorithms in our study work at the test-case level and therefore the mutation operator adds, removes and changes statements to/from/in each test case.

Selection Operator: test cases (or test suites in WSA) are selected for reproduction using *tournament selection*. In the single-target approach, the selection is done according to the single fitness function optimized in each GA run. In MOSA and DynaMOSA, the selection is performed by taking into account *dominance ranking* and *crowding distance* [14]. In WSA, the selection is based on the suite-level fitness function. In MIO, test cases are selected for mutation using *feedback-directed target selection*.

Termination criterion: all TCG approaches were set with the same maximum search budget of two minutes [27]. Therefore, the search is terminated when either all branches in a CUT are covered or the search timeout is reached. It should be noticed that a search timeout of two minutes was commonly used in large-scale experiments in test case generation [5,22,38–40]. This is because large-scale studies lead to very extensive computational costs, due to (i) the large number of CUTs, (ii) the number of algorithms in the comparison, and (iii) the number of repetitions required for a proper statistical analysis. Moreover, such a setting has been shown to provide a *reasonable trade-off between time and branch coverage* [5].

3.4. Experimental procedure

To account for the non-determinism of the TCG approaches, each approach was run 20 times on each CUT. Therefore, we performed 20 (repetitions) × 6 (TCG approaches) × 180 (classes) = 21,600 runs in total. In total, this setting required (21,600 runs × 2 min) / (60 min × 24 h) ≈ 28 days of computation time. In each run, we stored the maximum branch coverage achieved by the TCG approach executed in that run. Branch coverage is measured by executing the generated test suites using the execution engine of EvoSuite and by post-processing the test cases using EvoSuite's optimization facilities [5,17]. In particular, EvoSuite minimizes the generated test cases by removing spare statements that do not contribute to the final coverage. Moreover, candidate assertions are added using a mutation-based strategy [5]. Then, the resulting post-processed test suite is re-executed and the final branch coverage is collected (**RQ1**).

Other than collecting the coverage at the end of the search process, we also collected the percentage of covered branches at fixed intervals of 10 s. Therefore, in each run, we collected 12 data points allowing us to measure the performance of a TCG approach over time. We then measured efficiency (**RQ2**) using the *area under the curve* (AUC), computed with the trapezoidal rule:

$$AUC = \frac{\sum_{i=0}^{11} [cov_i + cov_{i+1}]^* \Delta Time}{2 \times TotalTime} \quad (4)$$

where cov_i is the coverage score achieved at the ith time point, $\Delta Time$ is the time elapsed between two consecutive data points (i.e., 10 s), and $TotalTime$ is the total search budget (i.e., 120 s). The AUC metric ranges between [0; 1] and higher values indicate better efficiency, i.e., a TCG approach reaches higher coverage in less time.

For the statistical analysis, we use the Friedman test to assess whether different TCG approaches achieve statistically different branch coverage (**RQ1**) and AUC scores (**RQ2**). The Friedman test is a non-parametric test for multi-problem analysis, i.e., it ranks the TCG approaches across all problems (CUTs). It has been used in the fifth edition of the SBST tool competition [2] and it is widely used in evolutionary computation to assess, compare and rank evolutionary algorithms [41]. In case of a significant p-value from the Friedman test, we applied the post-hoc Conover's procedure [42]. Such a procedure is a test for pairwise multiple comparisons, i.e., it is applied to discover which pairs of TCG approaches statistically differ. Finally, we use the Wilcoxon rank sum test [42] for each CUT separately, and for each pair of TCG approaches. This allows us to count the number of CUTs for which a TCG approach (e.g., LIPS) significantly outperforms another TCG approach (e.g., single-target approach) in terms of branch coverage (**RQ1**) and AUC scores (**RQ2**). For all statistical tests, we used the

[4] https://github.com/apanichella/evosuite/tree/LIPS_replication.
[5] https://github.com/EMResearch/EvoMaster.

Table 3
Average branch coverage achieved over 20 independent runs.

PID	Projectname	No. of classes	Dyna-MOSA	MOSA	MIO	WSA	LIPS	ST
1	tullibee	2	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.69
2	a4j	2	0.31	0.31	**0.32**	**0.32**	0.26	0.04
4	rif	1	**1.00**	**1.00**	**1.00**	**1.00**	0.99	0.68
5	templateit	2	**0.63**	**0.63**	0.62	0.60	0.57	0.48
7	sfmis	2	**0.69**	0.67	0.68	0.64	0.60	0.42
10	water-simulator	1	**0.12**	**0.12**	0.11	0.11	0.08	0.00
11	imsmart	1	1.00	1.00	1.00	1.00	1.00	1.00
12	dsachat	2	**0.23**	0.20	0.15	0.19	0.12	0.02
13	jdbacl	2	**0.29**	0.14	0.27	0.00	0.14	0.06
14	omjstate	1	**1.00**	**1.00**	0.99	**1.00**	0.99	0.74
15	beanbin	2	**0.74**	**0.74**	0.69	0.69	0.47	0.27
17	inspirento	2	**0.86**	**0.86**	0.83	0.85	0.77	0.44
18	jsecurity	3	**0.61**	**0.61**	0.54	0.57	0.46	0.12
19	jmca	4	**0.57**	0.54	0.40	0.50	0.25	0.06
21	geo-google	2	0.80	**0.81**	**0.81**	**0.81**	0.68	0.42
22	byuic	4	**0.45**	0.44	0.37	0.43	0.35	0.11
24	saxpath	2	**0.92**	0.91	0.88	0.87	0.85	0.56
26	jipa	2	**0.80**	0.79	0.79	0.76	0.76	0.58
27	gangup	2	0.25	**0.25**	**0.25**	**0.25**	**0.25**	0.03
29	apbsmem	2	**0.50**	**0.50**	0.49	**0.50**	0.47	0.21
31	xisemele	1	**0.55**	**0.55**	**0.55**	**0.55**	0.50	0.14
32	httpanalyzer	2	**0.41**	**0.41**	**0.41**	**0.41**	**0.41**	0.17
33	javaviewcontrol	3	0.47	0.46	0.39	**0.49**	0.34	0.21
35	corina	3	**0.04**	0.01	**0.04**	0.00	**0.04**	0.00
36	schemaspy	2	**0.21**	0.11	0.02	0.07	0.03	0.00
37	petsoar	2	**0.76**	**0.76**	**0.76**	**0.76**	**0.76**	0.58
38	javabullboard	2	**0.64**	**0.64**	0.63	**0.64**	0.51	0.19
39	diffi	2	0.91	0.90	**0.92**	0.90	0.88	0.75
40	glengineer	2	0.81	**0.82**	0.78	0.77	0.46	0.18
41	follow	2	0.52	**0.53**	0.50	0.46	0.46	0.17
42	asphodel	1	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	0.00
43	lilith	4	**0.64**	0.54	0.58	0.55	0.45	0.27
44	summa	2	**0.74**	0.72	0.72	0.69	0.37	0.03
45	lotus	2	**0.46**	**0.46**	**0.46**	**0.46**	0.39	0.17
46	nutzenportfolio	2	0.21	0.21	0.20	**0.22**	0.20	0.03
47	dvd-homevideo	3	**0.09**	**0.09**	**0.09**	0.08	**0.09**	0.00
49	diebierse	2	**0.63**	0.62	0.59	0.57	0.46	0.25
50	biff	1	**0.13**	0.06	0.00	0.12	0.02	0.00
51	jiprof	4	**0.63**	0.62	0.56	0.55	0.23	0.09
52	lagoon	2	**0.16**	0.15	0.14	**0.16**	0.15	0.02
54	db-everywhere	2	**0.33**	0.32	0.26	0.29	0.23	0.04
55	lavalamp	1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.99
56	jhandballmoves	2	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**	0.05
57	hft-bomberman	2	**0.51**	0.50	0.50	0.33	0.31	0.27
58	fps370	2	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.00
59	mygrid	2	**0.82**	**0.82**	**0.82**	**0.82**	0.73	0.47
60	sugar	2	**0.83**	0.82	0.80	0.82	0.80	0.57
61	noen	2	**0.76**	0.75	0.73	0.75	0.48	0.31
62	dom4j	2	**0.49**	0.47	0.44	0.45	0.26	0.05
63	objectexplorer	2	**0.50**	0.49	0.38	0.23	0.29	0.17
64	jtailgui	2	**0.59**	**0.59**	0.56	0.56	0.32	0.01
65	gsftp	2	**0.22**	**0.22**	0.19	0.20	0.11	0.04
66	openjms	2	**0.64**	**0.64**	0.63	0.59	0.56	0.34
67	gae-app-manager	1	**0.15**	**0.15**	**0.15**	**0.15**	0.14	0.02
68	biblestudy	2	**0.84**	**0.84**	0.83	**0.84**	0.81	0.58
69	lhamacaw	2	0.09	**0.10**	**0.10**	**0.10**	0.08	0.00
70	echodep	2	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.00
71	ext4j	2	**0.95**	**0.95**	**0.95**	**0.95**	0.90	0.68
72	battlecry	2	**0.34**	0.31	0.14	**0.34**	0.07	0.00
73	fim1	2	**0.13**	**0.13**	**0.13**	**0.13**	0.12	0.00
74	fixsuite	2	0.24	0.24	**0.25**	0.21	0.20	0.02
75	openhre	2	**0.99**	**0.99**	**0.99**	**0.99**	0.95	0.73
77	io-project	1	**0.89**	0.82	**0.89**	0.87	0.54	0.23
78	caloriecount	3	**0.83**	**0.83**	0.75	0.72	0.46	0.27
79	twfbplayer	2	0.87	**0.88**	0.83	0.84	0.60	0.26
80	wheelwebtool	4	**0.78**	0.77	0.74	0.75	0.57	0.22
81	javathena	4	**0.34**	0.33	0.31	0.34	0.28	0.06
82	ipcalculator	2	**0.71**	0.70	0.70	0.68	0.53	0.26
83	xbus	2	**0.25**	**0.25**	**0.25**	**0.25**	0.22	0.04
84	ifx-framework	1	**0.43**	**0.43**	**0.43**	**0.43**	0.41	0.12
85	shop	4	**0.49**	**0.49**	0.47	0.46	0.26	0.04
86	at-robots2-j	2	**0.56**	0.50	0.46	0.36	0.43	0.07
87	jaw-br	1	**0.19**	**0.19**	**0.19**	0.15	0.17	0.01
88	jopenchart	2	**0.40**	0.39	0.37	0.37	0.22	0.05
89	jiggler	1	**1.00**	**1.00**	0.93	**1.00**	0.88	0.64
100	jgaap	1	**0.83**	0.69	0.63	0.63	0.61	0.32
101	netweaver	2	**0.93**	**0.93**	**0.93**	**0.93**	0.77	0.48
102	squirrel-sql	2	**0.50**	0.46	**0.50**	0.48	0.43	0.10
103	sweethome3d	4	**0.06**	**0.06**	**0.06**	**0.06**	0.04	0.00
104	vuze	2	**0.02**	**0.02**	**0.02**	0.01	**0.02**	0.00
105	freemind	2	**0.50**	0.36	0.45	0.21	0.36	0.12
106	checkstyle	1	**0.11**	0.09	0.10	0.07	**0.11**	0.01
107	weka	4	**0.52**	0.51	0.46	0.50	0.40	0.16
108	liferay	1	**1.00**	**1.00**	**1.00**	0.86	**1.00**	0.55
109	pdfsam	2	**0.32**	**0.32**	**0.32**	**0.32**	**0.32**	0.07
110	firebird	3	**0.70**	**0.70**	0.60	0.64	0.58	0.27
	1st Quartile		**0.25**	0.24	**0.25**	0.21	0.20	0.03
	Median		**0.52**	0.51	0.50	0.50	0.40	0.15
	3rd Quartile		**0.80**	0.78	0.75	0.75	0.59	0.33

significance level $\alpha = 0.05$. The p-values obtained from the post-hoc Conover's procedure and by the Wilcoxon test are adjusted with the Holm-Bonferroni [42]. Such a correction is required when performing multiple comparisons.

To answer **RQ3**, we used the two-way permutation test [43] (significance level $\alpha = 0.05$) to verify whether there exists any statistically significant interaction between branch coverage (or AUC scores) achieved by each TCG approach with size (number of branches) of the CUTs. The two-way permutation test and all other statistical tests used in our evaluation are non-parametric, i.e., they do not require data being normally distributed.

4. Results

In this section, we discuss the results of our empirical evaluation and we answer the research questions formulated in Section 3.2. Reporting the results at class-level is not feasible due to the large number of CUTs in our benchmark. Therefore, we report the results at project-level: the average branch coverage and AUC scores achieved by each TCG approach over all the classes belonging to the same project. The detailed results at class-level are reported in the Appendix. For brevity, we refer to the single-target and the whole-suite approaches as ST and WSA, respectively. For the other four TCG approaches, we use their original acronyms.

4.1. RQ1: How do the different search-based algorithms perform in terms of effectiveness?

Table 3 summarizes the average branch coverage (**RQ1**) achieved by the six TCG approaches (i.e., ST, LIPS, WSA, MOSA, DynaMOSA, and MIO) over 20 independent executions. The table reports the results grouped by project and highlights in boldface the highest branch coverage for each project. Furthermore, Table 4 shows the results of the pairwise multiple comparisons performed with the Wilcoxon test and adjusting the p-values with the Holm- Bonferroni corrections procedure. This table reports the number of classes (CUTs) on which a given TCG approach (row) significantly outperforms another TCG approach (column) in terms of branch coverage (effectiveness).

As we can notice, ST produced the lowest average branch coverage among all TCG approaches, which is on average 23% across all projects and 21% across all classes. These percentages are sensibly lower than the scores achieved by the other TCG approaches: at class-level, DynaMOSA, MOSA, LIPS, WSA, and MIO achieved an average coverage of 52%, 50%, 40%, 48%, and 48%, respectively. For only two classes,

Table 4
Number of classes in which an algorithm A (row) outperforms another algorithm B (column) in terms of branch coverage as indicated by the Wilcoxon test (p-value ≤ 0.05) .

	DynaMOSA	LIPS	MIO	MOSA	ST	WSA
DynaMOSA	–	122	79	41	177	76
LIPS	2	–	17	11	175	14
MIO	12	87	–	19	175	40
MOSA	6	105	74	–	175	59
ST	0	0	0	1	–	0
WSA	11	103	53	22	174	–

namely *HTMLFilter* (project `imsmart`) and `ExtrasPatternParser` (project `ext4j`, ST reached 100% coverage. However, we can notice that these classes are relatively small, having 15 and 11 branches to cover in total, respectively.

As indicated in Table 4, on 174 (comparing with WSA) to 177 (comparing with DynaMOSA) classes, out of 180, ST reached a significantly lower branch coverage than the other TCG approaches. In the remaining few classes, there is no significant difference between ST and the other approaches. All these latter classes are, however, the smallest in our benchmark, with less than 30 branches in total. There is one single exception to this general rule: for the class `JPodThmbnailCallable` (project `pdfsam`), ST achieved significantly higher branch coverage than MOSA. This class has only 19 branches and provides an implementation of the Java interface `callable` aimed to generate thumbnails with `JPod`. For such a class, ST yielded a branch coverage of 0.60% while MOSA could not generate any test case in most of the runs. It is worth noting that this class is particularly challenging for all TCG approaches in our study since none of them was able to create test suites with branch coverage higher than 0.70%. On the other hand, MOSA outperformed ST in 175 classes out of 180 with an average difference in branch coverage ranging between 1% and 87%, being 31% on average. Similar observations can be done by comparing the other TCG approaches with ST, which achieved at least -20% branch coverage on average.

> *The single-target approach is the least effective TCG approach, as it achieved the lowest branch coverage for the vast majority of CUTs.*

LIPS is a recent variant of the single-target approach [19]. According to our results, it outperforms ST in 175 classes out of 180 with an average difference in branch coverage of +20%, ranging between 1% for class `TestHaSMETSProfile` (project `echodep`) and 61% for class `CascadeInternal-FramePositioner` (project `squirrel-sql`). These improvements are due to (i) the different order by which targets (i.e., branches) are selected, and (ii) *collateral coverage* detection, implemented in LIPS. To better explain these two aspects, let us focus on the class with the largest difference, i.e., `Cascade-InternalFramePositioner`, which has 21 branches to cover. Both ST and LIPS divided the overall search budget of 120 s among all branches. However, LIPS started with a randomly generated test case, that covered 9 (42%) trivial branches. The search budget was then divided among the remaining yet to cover branches (i.e., 12). GA was then initialized with a random population for the first branch in the worklist and performed on average 25 generations per branch (i.e., with a local budget of about 120 s/12 branches = 10 s per branch). In this way, LIPS covered on average 14–15 branches with the aid of collateral coverage detection. Instead, ST directly started GA with a randomly selected branch and, thus, with it had only 42% of probability to start with one of the trivial branches. Since collateral coverage is not detected in ST, trivial branches have been covered only when they were selected as search targets in one of the independent searches. As a further consequence, the average local budget for each branch was 120 s/21 branches ≈ 5.71 s per branch, a budget that is too small for the non- trivial targets.

Despite these improvements over ST, LIPS performs poorly compared to all TCG approaches that simultaneously optimize all targets at the same time. According to the Wilcoxon test (see Table 4), DynaMOSA, MOSA, WSA produced higher branch coverage than LIPS in more than 57% of the classes in our benchmark (68% for DynaMOSA); MIO outperformed LIPS in 48% of the CUTs. In detail, DynaMOSA achieved on average +12% branch coverage, with minimum difference of 1% for class `GroupPanel` (project `gangup`) and maximum difference of 65% for class `DirectoryScanner` (project `caloriecount`); MOSA led to an improvement in branch coverage ranging between +1% for class `RoomController` (project `sweethome3d`) and 64% for class `DirectoryScanner` (the same holds for DynaMOSA); MIO improved branch coverage over LIPS by 8% on average, with a minimum of 1.10% (the same case occurred with MOSA) and a maximum 54% (the same case occurred with MOSA and DynaMOSA). WSA outperformed LIPS with an average difference in branch coverage of +8%, the minimum and the maximum difference were +1.36% (the same case occurred with DynaMOSA) and +55% (for class `JMCAAnalyzer`, project `jmca`), respectively. The largest difference between LIPS and the multi-target TCG approaches are observable for classes with a large number of branches, i.e., with more than 75 branches on average. Vice versa, LIPS statistically outperformed DynaMOSA in 2 classes, MIO in 17 classes, MOSA in 11 classes and WSA in 14 classes. On all these classes the coverage differences were very small, being lower than 1% on average.

To provide a qualitative analysis of these results, let us consider the class `JavaParserTokenManager` from project `jmca`, containing 1707 branches. This is one of the largest classes in our benchmark, with many non-trivial branches. The initial test case generated by LIPS could cover (on average) only 122 branches (7% of the targets) and therefore the overall search budget of 120 s was divided among the remaining 1585 branches. This corresponds to a very small local search budget of 120 s/1585 branches ≈ 0.07 s per each uncovered branch. This local budget is so small that a full GA iteration was never completed: every time the GA was initialized, only a few new test cases could be generated before the search timeout was reached. This led to covering only the trivial branches that could be easily satisfied with randomly generated tests, i.e., without any evolution (or GA generation). This resulted in covering 398 branches on class `JavaParserTokenManager` in 120 s with LIPS. Instead, all TCG approaches considering all branches at once achieved larger coverage scores as they do not face the issue of splitting the search budget among targets. DynaMOSA (the best TCG approach for this class) covered on average 858 (as twice as many) branches within the same search budget of 120 s.

> *LIPS is significantly more effective than its predecessor, the single-target approach (ST). However, it is inferior to all TCG approaches that simultaneously optimize all targets/branches at once. This is due to the problem of splitting the search budget among many independent sub-searches, which turns out to be ineffective for large/complex classes.*

In the comparison between DynaMOSA and WSA, our results confirm previous findings [13,14,44]: the former achieved higher or equal branch coverage than the latter in most of the classes. In 42% of the classes, DynaMOSA achieved statistically higher branch coverage than WSA, with an average difference of +9%, a minimum of 1% for class `IndexedString`, project `diffi`, and a maximum difference of 54% for class `Attribute-ModelComparator` from project `objectexplorer`. In 51% of the classes, there is no statistically significant

difference between DynaMOSA and WSA in terms of branch coverage. For the remaining 11 classes (i.e., the remaining 6%), WSA produced a higher coverage, with an average difference of +4%. Looking at the size of the CUTs with statistical difference between DynaMOSA and WSA, we observe that the former performed better than the latter on larger classes. For example, let us consider the largest class in our benchmark, i.e., `JavaParser` from project `jmca`, having 7938 branches. In this class, WSA covered 1837 branches in 120 s while DynaMOSA covered 2252 (+416) branches within the same search budget, i.e., more than three additional branches per second.

Our results also confirm previous findings [14,44] for what regards the comparison between DynaMOSA and MOSA on branch coverage. In the majority of the classes (74%) there is no statistically significant difference between the two many-objective algorithms (see Table 4). In 46 classes (23%), DynaMOSA produced test suites with significantly higher coverage, while in the remaining six classes (3%) MOSA was significantly better than its recent variant. For what regards the magnitude of the difference, DynaMOSA led to an increment in branch coverage over MOSA ranging between 1% (class `TreeView` from `fixsuite`) and 37% (class `ConditionTableModel` from `lilith`); the average improvement is 5%. In the few cases where MOSA outperformed DynaMOSA, the differences are always lower than 2%. These results are quite expected since DynaMOSA was defined specifically to address very large classes with many search targets [14]. Indeed, DynaMOSA outperformed its predecessor in CUTs with more than 100 branches (on average), while the opposite is true only in few classes with less than 60 branches one average.

> *Our results confirm previous findings* [14,44]: *DynaMOSA produces higher or equal branch coverage as compared to both WSA and MOSA. Larger differences are observable for larger classes with hundreds/thousands of branches.*

Let us now focus on the comparison between DynaMOSA and MIO. As shown in Table 4, the former had significantly better coverage than the latter in 79 classes out of 180 (44%), with an average difference of 8%; the minimum difference is +1%, the maximum one is +40%. On the other hand, MIO provided better coverage in 12 classes (6%), with an average difference of 2.60% in branch coverage. For the remaining 89 classes (49%), there is no statistically significant difference between the two TCG approaches.

We identified two main reasons why DynaMOSA achieved often a better coverage than MIO. First, DynaMOSA generates offsprings using both *uniform mutation* and *single-point crossover*; instead, MIO relies only on the *mutation* operator to form new test cases from existing ones.[6] By performing a manual investigation, we discovered that crossover often plays an important role in achieving high coverage. To better explain this point, let us consider the class with the largest difference between DynaMOSA and MIO, i.e., class `bcGenerator` from project `battlecry`. MIO was able to cover on average 72 branches out of 281 (26%) while DynaMOSA covered 186 branches (on average) in 120 s. We can notice that by disabling the crossover in DynaMOSA (i.e., using the crossover probability $p_c = 0$), the resulting branch coverage is substantially lower, being 20% on average across 10 independent runs. This shows that crossover can help TCG achieve high coverage by increasing test case diversity. In fact, with crossover offsprings are generated by shuffling statements of two different test cases. On the other hand, the mutation operator changes (on average) only one statement in a given test case [14,15]. Although MIO generates random test cases over time (with a given probability), the new random tests are never mixed (via crossover) with existing ones. Therefore, the diversity injected by random tests is beneficial if and only if random testing is by itself capable of covering new branches.

[6] MIO can be viewed as a many-objective variant of the classical $(\mu + \lambda)$ Evolutionary Algorithm with sub-populations [15].

Table 5
Ranking produced by Friedman's (smaller values of Rank indicate better coverage) for branch coverage. We also report the statistical significance by the Conover's post-hoc procedure.

	Rank	Approach	Statistically better than
(1)	2.05	DynaMOSA	(2), (3), (4), (5), (6)
(2)	2.63	MOSA	(3), (4), (5), (6)
(3)	3.10	WSA	(4), (5), (6)
(4)	3.24	MIO	(5), (6)
(5)	4.10	LIPS	(6)
(6)	5.87	ST	–

Moreover, MIO uses *directed sampling selection* (see Section 2) to decide the sub-population (in other words, the target/branch) from which to sample a test case to mutate. Such a selection samples more frequently targets for which the fitness functions had improved over the latest iterations. Hence, complex targets are sampled less frequently [15] and the resulting strategy consists of (i) focusing the search on the easiest targets/branches first and (ii) not sampling from sub-populations associated with infeasible branches. While this strategy may help in case of truly infeasible branches, it may be detrimental in case of branches that are feasible but for which approach level and branch distance do not provide strong guidance. As shown by Samshiri et al. [36], this scenario is often common in Java code. Therefore, using the variation of fitness functions over time as heuristics to detect likely infeasible branches is often not effective in white-box unit testing.

> *DynaMOSA produces higher or equal branch coverage than MIO. The usage of the crossover operator helps DynaMOSA increase test case diversity without destroying the test case structure, with a corresponding lower probability of being trapped in local optima. Moreover, the directed sampling selection of MIO can be detrimental in case of feasible branches with poor fitness guidance.*

To further assess the statistical significance of our findings, we use Friedman's test [42] to compare the branch coverage scores yielded by the different TCG approaches across all CUTs in our benchmark. According to this test, the six approaches significantly differ from each other in terms of branch coverage (p-value $< 10^{-12}$). The corresponding Friedman's ranking and the results of the post-hoc procedure are reported in Table 5. The lowest (best) rank is obtained by DynaMOSA, which is significantly better than all the other TCG approaches as indicated by the post-hoc procedure. MOSA achieves the second best rank, followed by WSA, MIO, and LIPS, respectively. Each TCG approach in Table 5 significantly outperforms all approaches with higher (worst) ranking (all p-values are $< 10^{-12}$ after applying the Holm-Bonferroni correction procedure).

4.2. RQ2: How do the different search-based algorithms perform over time?

Table 6 reports the average AUC scores (**RQ2**) achieved by ST, LIPS, WSA, MOSA, DynaMOSA, and MIO over 20 independent runs. The table shows the results grouped by project and the highest score for each project is highlighted in boldface (as done for **RQ1**). Table 7 summarizes the results of the pairwise multiple comparisons based on the Wilcoxon test, with p-values adjusted according to the Holm-Bonferroni procedure. More specifically, it reports the number of CUTs in which each TCG approach achieved significantly higher AUC score compared to another TCG approach (i.e., better effectiveness over time).

Results in terms of AUC are very similar to the results we obtained for branch coverage (i.e., for **RQ1**). First, ST yielded the lowest average AUC score, which is on average 0.17 at project-level and 0.14 at class-

Table 6
Average AUC scores achieved over 20 independent runs .

PID	Project Name	No. of Branches	Dyna-MOSA	MOSA	MIO	WSA	LIPS	ST
1	tullibee	2	0.88	0.86	**0.89**	0.84	0.82	0.42
2	a4j	2	**0.29**	0.28	**0.29**	**0.29**	0.22	0.02
4	rif	1	0.91	0.90	**0.92**	0.87	0.86	0.52
5	templateit	2	**0.57**	0.55	0.54	0.51	0.52	0.39
7	sfmis	2	**0.63**	0.60	0.61	0.57	0.55	0.38
10	water-simulator	1	**0.09**	**0.09**	0.08	0.06	0.06	0.00
11	imsmart	1	0.97	0.96	**0.98**	0.93	0.98	0.85
12	dsachat	2	**0.16**	0.14	0.10	0.13	0.09	0.01
13	jdbacl	2	**0.22**	0.10	0.16	0.00	0.06	0.03
14	omjstate	1	**0.91**	**0.91**	**0.91**	0.83	0.89	0.53
15	beanbin	2	0.63	**0.64**	0.55	0.55	0.4	0.17
17	inspirento	2	**0.74**	**0.74**	0.71	0.67	0.63	0.30
18	jsecurity	3	**0.48**	0.46	0.41	0.40	0.37	0.07
19	jmca	4	**0.40**	0.36	0.25	0.32	0.18	0.03
21	geo-google	2	**0.67**	**0.67**	**0.67**	0.62	0.58	0.31
22	byuic	4	**0.40**	0.38	0.32	0.35	0.33	0.12
24	saxpath	2	**0.80**	0.79	0.72	0.71	0.77	0.44
26	jipa	2	**0.72**	0.71	0.71	0.61	0.69	0.47
27	gangup	2	**0.23**	**0.23**	**0.23**	0.22	0.22	0.04
29	apbsmem	2	**0.41**	0.37	0.39	0.31	0.34	0.09
31	xisemele	1	**0.48**	**0.48**	0.46	0.37	0.42	0.09
32	httpanalyzer	2	0.37	0.39	**0.40**	0.36	0.37	0.23
33	javaviewcontrol	3	**0.38**	0.37	0.33	0.36	0.3	0.17
35	corina	3	**0.03**	0.01	0.02	0.00	0.01	0.00
36	schemaspy	2	**0.08**	0.05	0.01	0.02	0.02	0.00
37	petsoar	1	0.95	0.95	**0.96**	0.89	0.96	0.83
38	javabullboard	2	**0.59**	0.58	0.55	0.52	0.46	0.13
39	diffi	2	**0.82**	0.81	0.80	0.72	0.79	0.54
40	glengineer	2	**0.64**	0.63	0.61	0.53	0.32	0.08
41	follow	2	**0.40**	0.39	0.39	0.31	0.34	0.09
42	asphodel	1	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	0.00
43	lilith	4	**0.55**	0.45	0.47	0.37	0.37	0.14
44	summa	2	**0.64**	0.60	0.55	0.56	0.28	0.01
45	lotus	2	**0.43**	0.42	0.42	0.39	0.38	0.17
46	nutzenportfolio	2	**0.18**	**0.18**	0.17	0.16	0.16	0.02
47	dvd-homevideo	3	**0.08**	**0.08**	**0.08**	0.07	0.07	0.00
49	diebierse	2	**0.48**	0.47	0.45	0.40	0.33	0.14
50	biff	1	0.06	0.01	0.00	**0.07**	0.01	0.00
51	jiprof	4	**0.46**	0.45	0.33	0.33	0.17	0.05
52	lagoon	2	**0.13**	0.12	0.12	0.13	0.12	0.02
54	db-everywhere	2	**0.27**	0.22	0.18	0.21	0.17	0.03
55	lavalamp	1	**0.96**	**0.96**	0.95	0.93	**0.96**	0.84
56	jhandballmoves	2	**0.18**	**0.18**	**0.18**	0.17	**0.18**	0.04
57	hft-bomberman	2	**0.37**	0.36	0.34	0.17	0.18	0.04
58	fps370	2	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.00
59	mygrid	2	**0.75**	0.74	0.71	0.69	0.67	0.39
60	sugar	2	**0.77**	0.75	0.74	0.69	0.74	0.45
61	noen	2	**0.61**	0.57	0.54	0.55	0.4	0.12
62	dom4j	2	**0.43**	0.41	0.35	0.38	0.23	0.04
63	objectexplorer	2	0.31	**0.32**	0.26	0.17	0.2	0.05
64	jtailgui	2	0.41	**0.43**	0.35	0.35	0.2	0.01
65	gsftp	2	**0.16**	**0.16**	0.15	0.14	0.09	0.02
66	openjms	2	**0.52**	**0.52**	0.50	0.46	0.45	0.21
67	gae-app-manager	1	0.13	**0.14**	**0.14**	0.12	0.13	0.02
68	biblestudy	2	**0.79**	**0.79**	0.77	0.73	0.75	0.47
69	lhamacaw	2	0.08	0.08	0.08	**0.09**	0.07	0.00
70	echodep	2	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.00
71	ext4j	2	**0.89**	0.87	0.85	0.81	0.84	0.56
72	battlecry	2	**0.29**	0.22	0.08	0.21	0.06	0.00
73	fim1	2	0.10	0.10	0.10	**0.11**	0.09	0.00
74	fixsuite	2	0.18	0.18	**0.19**	0.09	0.14	0.00
75	openhre	2	**0.92**	**0.92**	0.89	0.87	0.87	0.58
77	io-project	1	**0.69**	0.63	0.67	0.60	0.44	0.16
78	caloriecount	3	**0.62**	0.58	0.54	0.49	0.34	0.12
79	twfbplayer	2	**0.74**	**0.74**	0.68	0.68	0.53	0.17
80	wheelwebtool	4	**0.57**	0.56	0.55	0.47	0.42	0.13
81	javathena	4	**0.28**	0.26	0.24	0.23	0.23	0.04
82	ipcalculator	2	**0.59**	0.54	0.55	0.54	0.46	0.17
83	xbus	2	0.22	0.22	**0.23**	0.21	0.2	0.03
84	ifx-framework	1	**0.39**	0.38	0.37	0.38	0.37	0.08
85	shop	4	**0.39**	0.38	0.36	0.26	0.2	0.02
86	at-robots2-j	2	**0.41**	0.37	0.33	0.24	0.27	0.06
87	jaw-br	1	**0.15**	0.14	0.12	0.09	0.09	0.01
88	jopenchart	2	**0.30**	0.29	0.28	0.27	0.18	0.04
89	jiggler	1	0.78	0.85	0.77	**0.86**	0.76	0.42
100	jgaap	1	**0.70**	0.57	0.53	0.48	0.46	0.16
101	netweaver	2	**0.82**	0.80	0.79	0.75	0.66	0.35
102	squirrel-sql	2	**0.41**	0.38	0.40	0.38	0.32	0.06
103	sweethome3d	4	**0.04**	**0.04**	**0.04**	**0.04**	0.03	0.00
104	vuze	2	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.00
105	freemind	2	**0.42**	0.30	0.37	0.14	0.31	0.07
106	checkstyle	1	**0.10**	0.08	0.10	0.07	0.08	0.00
107	weka	4	0.38	**0.39**	0.35	0.34	0.29	0.05
108	liferay	2	**0.48**	**0.48**	0.45	0.38	0.42	0.21
109	pdfsam	2	**0.28**	0.27	0.26	0.22	0.25	0.05
110	firebird	3	**0.59**	**0.59**	0.49	0.51	0.49	0.17
	1st Quartile		**0.22**	0.19	0.18	0.16	0.16	0.02
	Median		**0.41**	0.39	0.38	0.36	0.32	0.07
	3rd Quartile		**0.64**	0.62	0.59	0.38	0.37	0.17

Table 7
Number of classes in which an algorithm A (row) outperforms another algorithm B (column) in terms of AUC as indicated by the Wilcoxon test (p-value ≤ 0.05).

	DynaMOSA	LIPS	MIO	MOSA	ST	WSA
DynaMOSA	–	152	109	92	179	158
LIPS	8	–	18	15	178	61
MIO	33	127	–	50	177	121
MOSA	12	131	85	–	176	137
ST	0	0	1	2	–	5
WSA	3	81	32	11	170	–

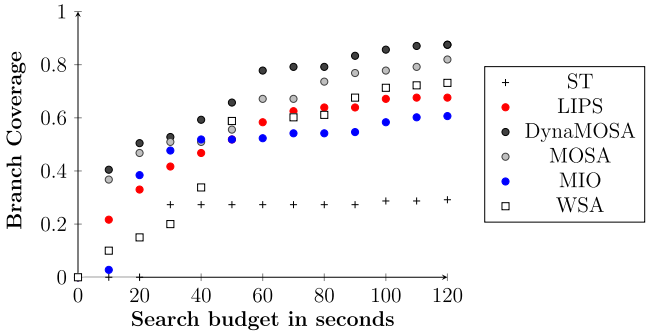

Fig. 2. Percentage of covered branches over search time for the class `JavaCharStream`.

level. These scores are very low as compared to all other TCG approaches: DynaMOSA achieved an average AUC score of 0.43 (+0.29) at class-level while MOSA, LIPS, WSA and MIO yielded the scores 0.41 (+0.27), 0.33 (+19), 0.37 (+0.22), and 0.39 (+0.25), respectively. According to Table 7, TS is significantly inferior to the other TCG approaches in at least 170 classes out of 180 (179 if compared with DynaMOSA). Vice versa, on only one (compared to MIO), two (compared to MOSA) and five (compared to WSA) classes out of 180 ST had a significantly higher AUC score than the alternative approaches. All these (few) classes have either very small size (less than 50 branches) or are poorly covered by all TCG approaches.

For example, one of the CUTs falling in the latter scenario is class `DBUtil` from project `jdbacl` (197 branches). ST obtained a very low AUC score of 0.02 (on average) for this CUT while WSA did not generate any test case in most of its runs. `DBUtil` is challenging for all other TCG approaches as well, since none of them achieved an AUC score greater than 0.06. The few covered branches are actually very

trivial since they are covered by the initial population of GA (i.e., by few randomly generated tests). We can further notice that one generation of WSA is more expensive than a single generation of ST since the former evaluates more test cases (contained in a fixed number of test suites) than the latter (fixed number of test cases). In such a scenario, ST achieves maximum – yet, poor – coverage earlier than WSA, only because its initialization phase is faster. So, the higher AUC of ST is just a consequence of its faster generation of the initial, random population.

> *The single-target approach is the least efficient TCG approach, as it achieved the lowest AUC scores for the large majority of the CUTs. It can be more efficient than WSA only for CUTs where maximum coverage is reached by the random tests generated for the initial population (which happens quite rarely).*

The AUC scores obtained by LIPS further confirm its superiority over the classical single-target approach (ST). As reported in Table 7, LIPS outperformed ST in 178 classes out of 180, with an improvement ranging between 0.01 (class `Scanner` from project `biff`) and 0.51 (class `PlotAxis` from project `apbsmem`), being 0.20 on average. Let us consider as a qualitative example the convergence graph for class `JavaCharStream` from `jmca`, depicted in Fig. 2. The graph shows, for each TCG approach, the convergence curve corresponding to the median AUC value across 20 independent runs. As we can notice, LIPS always provides a substantially higher coverage than ST over the entire search time window. The large differences are mainly due to both (i) the different order in which branches are selected (being random in ST and based on CDG in LIPS), and (ii) collateral coverage detection. Since ST does not consider collateral coverage [22], branches that are accidentally covered remain undetected and their covering test cases are not stored to form the final test suite. As a consequence, ST tends to achieve lower branch coverage than LIPS over time; this results in a higher AUC score for the latter approach over the former.

However, LIPS does not perform well in comparison with the TCG approaches that optimize for all branches simultaneously. For the class in Fig. 2, LIPS produced a lower convergence curve than MOSA and DynaMOSA. In general, multi-target approaches are significantly more efficient (according to the AUC metric) than LIPS in at least 45% of the classes in our benchmark (up to 84% for DynaMOSA). In details, the average difference between DynaMOSA and LIPS is 0.13 on average, with a minimum of 0.01 for class `GlobalPreferencesSheet` (project `squirrel-sql`) and a maximum of 0.47 for class `DirectoryScanner` (project `caloriecount`); MOSA improved the AUC scores from 0.01 for class `IFXObject` (project `ifx-framework`) to 0.43 for class `Block` (project `glengineer`); MIO led to an average improvement in AUC scores equal to 0.10, with a minimum of 0.01 for class `FontChooserDialog` (project `fim1`) and a maximum of 0.42 for the class `Block` (the same holds for MOSA).

For what regards the comparison between LIPS and WSA, results are mixed. In 34% of the classes, the former outperformed the latter, while the opposite scenario is true for the other 45% of CUTs. There are 63 CUTs for which LIPS and WSA are statistically equivalent in terms of effectiveness (**RQ1**), i.e., they reached the same branch coverage at the end of the search budget. However, in 47 of these CUTs, LIPS is usually much quicker than WSA at the beginning of the search. This is because LIPS evaluates 50 test cases in each generation; instead, WSA evaluates 50 test suites, with many test cases each. For instance, let us consider again class `JavaCharStream`, whose corresponding convergence graph is depicted in Fig. 2. By looking at the first 40 s of the search, we can notice that LIPS has higher coverage than WSA; between 60 s and 80 s the two approaches perform equally and then in the last 40 s of

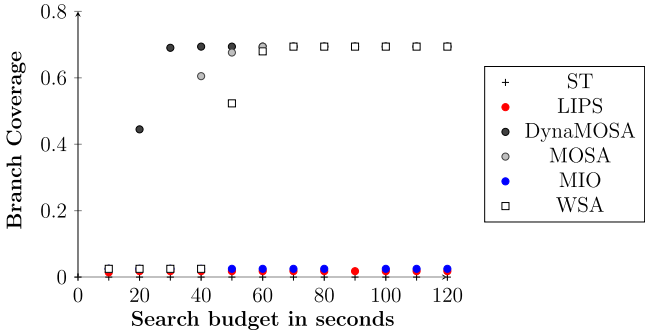

Fig. 3. Percentage of covered branches over search time for the class `bcGenerator`.

search WSA outperforms LIPS. It is worth noting that `JavaCharStream` has 216 branches and therefore LIPS assigned less than one second to each branch. Around 30%-40% of the branches can be covered with random tests and LIPS quickly detects them via collateral coverage. Instead, WSA needs one full generation (i.e., the execution of 50 randomly generated suites) before updating the archive. This overhead, due to the suite-level genetic operators, might initially penalize WSA in case the class has many trivially covered branches.

> *LIPS is more efficient than the single-target approach (ST) but it is less efficient then all multi-target TCG approaches. LIPS can be more efficient than WSA on classes with many easy-to-cover branches when a short search budget is available for the search.*

For what concerns the comparison between DynaMOSA and its predecessor MOSA, the Wilcoxon test revealed that the former significantly outperformed the latter in 51% of the CUTs (see Table 7. The difference between the two many-objective approaches in terms of AUC scores ranges between 0.01 (class `Queue` on project `biblestudy`) and 0.37 (class `ConditionTableModel` on project `lilith`); the average difference is 0.04. Only for 12 classes, MOSA is significantly more efficient (better AUC) than DynaMOSA. However, the magnitude of the difference for these classes is small, being 0.02 on average. These results are quite expected, as DynaMOSA was designed to reduce the computational cost associated with many-objective operators, namely *crowding distance* and *non-dominated sorting* [14], by dynamically defining the search objectives. For classes with a small number of statically determined objectives, the difference between the two approaches is negligible. For very large classes, with hundreds or thousands of branches, handling a lower number of objectives at a time leads to a significantly higher efficiency. Let us consider as an example the largest CUT in our benchmark: `JavaParser` (7938 branches) from project `jmca`. DynaMOSA could perform on average 51 generations in 120 s and achieved an average AUC score equal to 0.21; instead, MOSA performed on average 38 generations (−25%) in the same time window, thus, resulting in a lower AUC value of 0.17 on average.

Our results further confirm the results of previous findings [14] for what regards the comparison between DynaMOSA and WSA. Indeed, DynaMOSA achieved statistically higher AUC scores than WSA in 88% of the classes, with an average difference of +9%, a minimum of 0.01 for class `QuotaDetailsParser` (project `gae-app-manager`), and a maximum of 0.48 for class `JDayChooser` (project `freemind`). As in **RQ1**, larger differences in AUC scores are observable for very large classes. For example, for class `JavaParser` (the largest in our study) WSA yielded an average AUC value equal to 0.09 while DynaMOSA

```
public class bcGenerator{
  ...
  public List getLyrics() {
    if ((module != null) && (module.isInitialized())) {
      ...
      voice.sing("* " + fill(parseGrammar("TITLE")) + " *");
      ...
    } else {...}
    return voice.getLyrics();
  }
}
```

Fig. 4. Example of branch covered by DynaMOSA but not by MIO for class *bcGenerator* of project `battlecry`.

obtained the value 0.22. On only three classes, WSA produced higher AUC scores, with a maximum improvement of just 0.01.

> *DynaMOSA is significantly more efficient than both its predecessor MOSA and WSA. Similar to* ***RQ1*** *, the larger differences are obtained for larger classes, with hundreds/thousands of branches.*

Finally, we observe that DynaMOSA had significantly higher AUC scores than MIO in 109 classes out of 180 (61%). The average difference is 0.07, the minimum difference is 0.01 (class `CascadeInternalFramePositioner` from `squirrel-sql`) and the maximum is 0.43 (class `bcGenerator` from `battlecry`). MIO significantly outperformed DynaMOSA in 33 classes out of 180 (18%), with an average difference of 0.03. In the remaining 42 classes (23%) the two TCG approaches are statistically equivalent according to the results of the Wilcoxon test.

To better understand why DynaMOSA is often more efficient than MIO, let us analyze class `bcGenerator`. Fig. 3 depicts the median convergence curves achieved by all TCG approaches in our study. As we can observe from Fig. 3, MIO achieves very low branch coverage (< 2%) over the entire search time (i.e., 120 s) while DynaMOSA quickly covered 69% of the branches within the first 30 s. DynaMOSA is also the fastest TCG approach to reach this coverage value, followed by MOSA and WSA. The remaining approaches failed to reach such a high branch coverage.

As already explained in Section 4.1, the large difference between MIO and DynaMOSA is due to both *single-point crossover* (used in DynaMOSA but not in MIO) and *directed sampling selection* (used by MIO only). To provide further evidence about the drawbacks of using *directed sampling selection*, let us consider class `bcGenerator` (whose convergence graph is depicted in Fig. 3). This class contains 281 branches, one public constructor, two public methods, and 16 private methods. These private methods can be covered only by satisfying the true branch (b_t) of the `if`-statement of the public method `getLyrics()`, whose code is shown in Fig. 4. The code nested in b_t calls two private methods, i.e., `fill()` and `parseGrammar()`, which invoke all the remaining private methods. Covering b_t requires satisfying two conditions: the attribute `module` should not be `null` and should be initialized. These two conditions, however, do not have a branch distance with smooth gradient, as they can have only true/false values (this is the well-known flag-variable problem in TCG). Since there is no proper guidance toward satisfying b_t, the corresponding fitness function does not decrease over time (except when it is actually covered) and therefore MIO labels it as a *likely-infeasible* branch, ignoring its corresponding population in later iterations (*directed sampling selection*). Since covering b_t is critical for covering all branches of the private methods, ignoring its population penalizes MIO. Instead, DynaMOSA does not make any assumption about the feasibility of b_t, which remains one of the objectives to optimize. When b_t is covered, DynaMOSA updates the set of objectives by removing b_t (and saving its covering test in the archive) and adding the root branches of the private methods `fill()` and `parseGrammar()`.

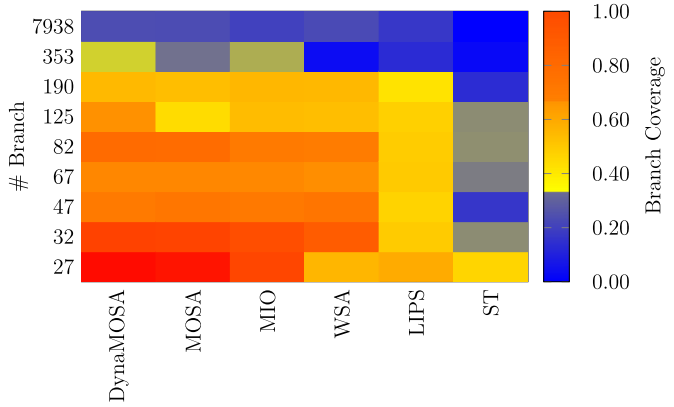

Fig. 5. Heatmap showing the interaction between branch coverage and CUT size (number of branches to cover) for the different TCG approaches.

> *DynaMOSA is often more efficient than MIO. Indeed, it achieved significantly higher AUC scores for the majority of CUTs in the benchmark. MIO is often penalized by the directed sampling selection strategy in case of feasible branches with poor fitness guidance.*

Table 8
Ranking produced by the Friedman's (smaller values of Rank indicate better coverage) for the AUC metric. We also report statistical significance by the Conover's post-hoc procedure.

	Rank	Approach	Statistically better than
(1)	1.71	DynaMOSA	(2), (3), (4), (5), (6)
(2)	2.46	MOSA	(3), (4), (5), (6)
(3)	2.77	MIO	(4), (5), (6)
(4)	3.99	WSA	(5), (6)
(5)	4.21	LIPS	(6)
(6)	5.85	ST	–

The results of the Friedman's test [42] revealed that the six approaches significantly differ from one another in terms of efficiency (p-value $< 10^{-12}$). Then, we applied the corresponding Friedman's ranking and the Conover's post-hoc procedure to determine for which pair of TCG approaches such a significant difference holds. The results of these tests are reported in Table 8. As we can notice, the lowest (best) rank is obtained by DynaMOSA, which is significantly better than all the other TCG approaches. MOSA achieves the second best rank, followed by MIO, WSA, and LIPS, respectively. Each TCG approach in Table 5 significantly outperforms all approaches with higher (worse) rank (all p-values are $< 10^{-12}$ after applying the Holm-Bonferroni correction procedure).

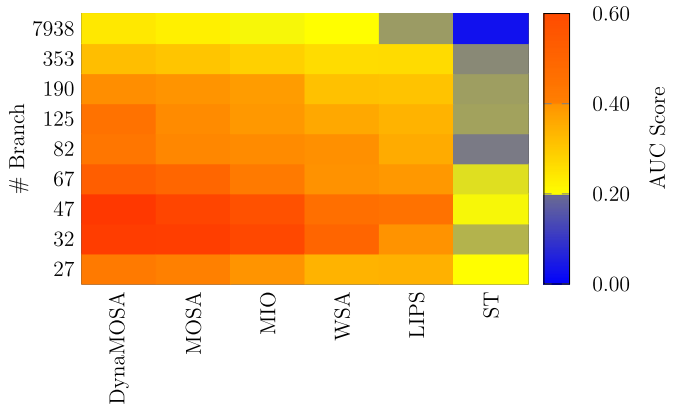

Fig. 6. Heatmap showing the interaction between AUC scores and CUT size (number of branches to cover) for the different TCG approaches.

4.3. RQ3: Does the class size affect the performance of the different search algorithms?

Fig. 5 plots the relation among class size (measured as the number of branches in each CUT), TCG approach and corresponding average branch coverage. The figure depicts a heatmap that clusters the CUTs according to their number of branches (*y*-axis); each entry (or cell) shows the average branch coverage (color) achieved by each TCG approach (*x*-axis) for the CUTs within the same cluster. For clustering, we sorted CUTs in descending order of size (i.e., number of branches) and grouped them in clusters of 20 CUTs each. Cells with red colors denote an average coverage close to 100% while cells in blue color correspond to clusters of CUTs with an average coverage close to 0%.

We can observe that the performance of all TCG approaches decreases when the size (i.e., number of branches) of the classes increases. This is clearly due to the fact that we limited our search budget to 120 s, while larger time should be spent for very large CUTs. ST achieves a very low coverage independently of the size of the CUTs: it achieved a coverage always lower than 20% with the exception of CUTs with less than 27 branches. For CUTs with more than 190 branches, ST achieved almost zero coverage. LIPS performed better than ST, but its results are not comparable with DynaMOSA, MOSA, MIO, and WSA. For very small classes with less than 27 branches, LIPS had a larger coverage than WSA. In general, DynaMOSA shows the best coverage among all TCG approaches. It is equivalent to MOSA for CUTs with up to 82 branches, while the heatmap shows better coverage than DynaMOSA on larger classes.

We have run the two-way *permutation test* to verify whether there is any significant interaction between TCG approach, branch coverage, and CUTs size. To this aim, we used the implementation of the permutation test available in `R` (package `lmPerm`), setting the number of iterations to 10^8. We followed the recommendation of having a very large number of iterations to increase the reliability of the test results [42]. The permutation test revealed that (i) the total number of branches (class size) does not significantly affect the coverage achieved by the different TCG approaches (p-value = 0.99); (ii) the choice of the TCG approach significantly impacts the final branch coverage (p-value $< 10^{-16}$); (iii) there is no significant interaction between TCG approach, class size and final branch coverage (p-value = 1.00). In other words, the choice of the TCG approach is the only factor to take into account when the goal is to achieve high coverage. This finding is expected given the very large superiority of multi-target TCG approaches over the single-target ones (i.e., LIPS and ST).

Fig. 6 shows the heatmap for the following three dimensions: (i) number of branches in the CUTs (*y*-axis); (ii) TCG approach (*x*-axis); and (iii) average AUC scores achieved by each TCG approach in each group of CUTs (color of each entry). We again sorted the CUTs in descending order of size and grouped them in clusters of 20 CUTs each. Better AUC scores correspond to red entries in the heatmap.

The plot shows that ST is the least efficient TCG approach, with an average AUC score which is sensibly lower than all other TCG approaches. On the other hand, DynaMOSA achieves the best average AUC values, especially for CUTs with more than 67 branches (for smaller classes, it turns out to be equivalent to MOSA). MIO is more efficient than WSA independently of the size of the CUTs. As explained in Sections 4.1 and 4.2, this is due to the overhead of the WSA genetic operators that work at the test-suite level.

To determine which factor (or combination of factors) significantly impacts the achieved AUC score, we applied the two-way *permutation test* using the same methodology followed for analyzing branch coverage (i.e., using `R`'s `lmPerm` package with 10^8 iterations). According to the test, similarly to the case of branch coverage, the only factor that significantly affects the AUC scores is the choice of the TCG approach (p-value $< 10^{-16}$). Instead, the size of the CUTs (i.e., number of branches) has no significant impact on the achieved AUC scores (p-value = 1.00). Moreover, there is no significant interaction between TCG approach, class size, and AUC score.

4.4. Threats to validity

Threats to *construct validity* concern the relation between theory and experimentation. The comparison of the six test generation approaches is based on performance metrics that are widely adopted in the literature: branch coverage and AUC. These metrics give a reasonable estimation of the effectiveness (coverage metrics) and efficiency (AUC) of the test case generation techniques.

Threats to *internal validity* concern factors that are likely to influence our results. One such threat comes from the inherent randomness of GA. To minimize its effect, we repeated each execution 20 times and reported average performance together with rigorous statistical analysis to support our findings. Another potential threat comes from GA parameters. We used parameter values suggested by the original authors of the techniques. For parameters not set by the original authors, we resorted to default values used in EvoSuite and in the related literature [34]. All experimented algorithms are implemented in the same tool, EvoSuite, thus, they share the same implementations of the various search operators. This minimizes potential confounding effects arising from using different tools with varying implementations. LIPS, originally developed for C programs, was re-implemented in EvoSuite so that it could work on Java programs. Our re-implementation was found to be superior to the original one when compared using the original experimental subjects, as reported in our conference paper [20].

Threats to *conclusion validity* arise from the relationship between treatment and outcome. We analyzed the results from our experiments by means of appropriate statistical tests coupled with enough repetitions of the experiments to enable the statistical tests. In particular, we used the non-parametric Friedman test for ranking the approaches across all the subjects, together with the post-hoc Conover's procedure. We used the Wilcoxon rank sum test to identify the number of subjects on which one approach outperforms another. For all tests, we used significance level $\alpha = 0.05$, adjusted with the Holm-Bonferroni method for multiple comparisons. We drew conclusions only when results were significant according to these tests.

Threats to *external validity* concern the generalization of the results. We used a benchmark of 180 Java classes, chosen randomly from the SF110 benchmark based on their cyclomatic complexity. These software projects have been used in previous work on test case generation (e.g., [18,34]). However, generalization of the results reported in this paper should be taken with care, as remarked in the detailed analysis presented in Section 4. Future experiments on different benchmarks could further increase confidence in the results. Another potential threat to the external validity is related to the choice of the search budget. We used a budget of 120 s as suggested in previous large-scale studies [5,22,38–40]. Nevertheless, using different search budgets may

lead to different results. To address this threat, we use the AUC as a second performance metric in addition to the branch coverage achieved at the end of the search. The AUC metric provides an overview of the coverage achieved over time and, therefore, is less sensitive to different, shorter budgets.

5. Related works

The conference version of this article [20] was mainly focused on replicating a previous study [19] with the objective of shedding light on results reported in the replicated study. In the current article, we further expand our empirical investigation to include state of the art search-based test generation approaches. This study goes in the direction of recent efforts in the community to produce comprehensive empirical evidence on the strengths and weaknesses of techniques and tools available for the generation of test cases.

Recent work in search-based test generation includes comparative empirical investigations, albeit only considering a small number of approaches at a time. For instance, Rojas et. al [22] reported an empirical investigation of the effectiveness of the whole-suite approach with respect to the single-target approach. They also explored the *archive*-enabled variant of the whole-suite approach. Their results, on a set of 100 Java classes, showed that for the large majority of the cases, the whole-suite approach gives better performance than single-target, while also confirming the presence of a few cases where the single-target outperforms whole-suite. Furthermore, their results also showed that the presence of an *archive* results in better results for whole-suite [22]. The empirical investigation reported in this work did not include MOSA.

A more comprehensive and detailed empirical investigation of evolutionary algorithms for test case generation was recently reported by Campos et. al [44]. The study involved several variants of evolutionary algorithms (Standard GA, Monotonic GA, Steady-State GA), $\mu + \lambda$ EA), MOSA, DynaMOSA, as well as random search. All algorithms were implemented in EvoSuite. Particular emphasis was given to combinations of different coverage criteria. The results reported reveal that GAs outperform random search, in particular on complex subjects, and specifically the $\mu + \lambda$ EA variant gave better performance over the other variants. Furthermore, results show that for branch coverage MOSA outperformed the whole-suite approach, while for multiple criteria its overall coverage was lower.

Another detailed investigation of various search algorithms and their combinations, all implemented in EvoSuite, was reported by Gay [45]. In this study, the author reported the impact of various coverage criteria on the resulting test suites' fault revealing capability, using the Defects4j faults dataset [46]. The findings of this study show that applying all criteria in combination is mostly ineffective. The same holds when applying a single criterion. Rather, a basic structural criterion (e.g., branch coverage), combined with additional, secondary criteria, specific to the desired testing objective, was found to be most effective at revealing faults. Test generation was performed using the default algorithm and settings in EvoSuite.

Other past efforts produced empirical results on different aspects of search-based test generators (e.g., [47]). Such studies investigated different aspects of the test generation problem, suggesting the need for improved test generation techniques and tools. The present article goes towards the same goal, by providing sound empirical results which serve as a solid basis for future research in the area. In our study, we have considered search-based test generation techniques available in the state of the art. As new techniques and tools keep on being introduced, such empirical studies need to be continuously conducted to investigate in detail the new techniques with respect to the existing ones.

6. Conclusion and future works

In this article, we have presented the detailed empirical investigation that we carried out to compare six state-of-the-art techniques for search-based test case generation. The techniques considered, i.e., TS, LIPS, WSA, MIO, MOSA and DynaMOSA, cover different approaches in the area of search-based test generation, spanning from the early approach in which every target (branch) is optimized in isolation (ST) and its later evolution (LIPS), to some recent approaches that optimize simultaneously all the targets at once (WSA, MIO, MOSA, DynaMOSA).

While different approaches have their own pros and cons, from our empirical investigation, the following conclusions, in line with what we reported previously [14,19,44], can be drawn: First, the single-target approaches considered in our study are ineffective when compared to multi-target approaches. This poor performance is due to the inefficient budget allocation especially for large and complex classes, which remains an open issue for single-target strategies. While approaches that optimize all targets simultaneously are generally superior to single target approaches, among them DynaMOSA outperforms the others both in terms of effectiveness and efficiency. Further details have been investigated as to why one technique outperforms the other. All analyses are supported by rigorous statistical evidence based on a large number of independent executions. Our benchmark is reasonably large, consisting of 180 non-trivial Java classes sampled from the SF110 corpus of Java classes, and conclusions were drawn only when there is statistically significant evidence.

In future work, the comparison of such techniques could be further extended beyond coverage metrics by, for example, considering fault revealing capabilities of the techniques. This could be achieved, for instance, by using existing real fault benchmarks (e.g., Defects4j [46]) to measure the effectiveness of the generated test suites in exposing real faults in the subject applications. Furthermore, mutation analysis could also be used to measure the fault revealing capability of generated test suites by simulating real faults via mutants [18], in particular for subjects where real faults are not readily available.

Appendix A

Table A.9
Average branch coverage achieved over 20 independent runs.

PID	Project	Subject	# Br.	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
1	tullibee	ComboLeg	21	1.00	1.00	1.00	1.00	0.79	1.00
1	tullibee	ExecutionFilter	20	1.00	1.00	1.00	1.00	0.59	1.00
2	a4j	FileUtil	125	0.52	0.53	0.54	0.54	0.08	0.42
2	a4j	Product	31	0.10	0.10	0.10	0.10	0.01	0.10
4	rif	WebServiceDescriptor	21	1.00	1.00	1.00	1.00	0.68	0.99
5	templateit	Region	31	1.00	1.00	1.00	1.00	0.93	1.00
5	templateit	WorkbookParser	54	0.26	0.25	0.24	0.21	0.02	0.14

(*continued on next page*)

Table A.9 (*continued*)

PID	Project	Subject	# Br.	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
7	sfmis	Base64	32	1.00	1.00	1.00	1.00	0.83	1.00
7	sfmis	Loader	82	0.38	0.34	0.35	0.27	0.02	0.20
10	water-simulator	ConsumerAgent	137	0.12	0.12	0.11	0.11	0.00	0.08
11	imsmart	HTMLFilter	15	1.00	1.00	1.00	1.00	1.00	1.00
12	dsachat	Handler	98	0.42	0.36	0.25	0.34	0.04	0.20
12	dsachat	InternalChatFrame	69	0.04	0.04	0.04	0.04	0.00	0.04
13	jdbacl	DBUtil	197	0.04	0.00	0.04	0.00	0.02	0.06
13	jdbacl	SQLUtil	188	0.53	0.28	0.49	0.00	0.11	0.22
14	omjstate	Transition	30	1.00	1.00	0.99	1.00	0.74	0.99
15	beanbin	MethodReflectionCriteria	47	0.83	0.83	0.72	0.72	0.32	0.57
15	beanbin	ReflectionSearch	41	0.66	0.66	0.66	0.66	0.22	0.37
17	inspirento	MainMenu	27	0.74	0.74	0.74	0.74	0.48	0.72
17	inspirento	XmlElement	95	0.98	0.97	0.93	0.97	0.39	0.81
18	jsecurity	AntPathMatcher	170	0.75	0.75	0.63	0.72	0.21	0.56
18	jsecurity	BasicHttpAuthenticationFilter	36	0.44	0.44	0.44	0.44	0.09	0.42
18	jsecurity	DefaultWebSecurityManager	67	0.63	0.63	0.55	0.55	0.07	0.39
19	jmca	JMCAAnalyzer	199	0.61	0.59	0.50	0.60	0.01	0.06
19	jmca	JavaCharStream	216	0.87	0.84	0.66	0.72	0.20	0.66
19	jmca	JavaParser	7938	0.28	0.25	0.17	0.23	0.00	0.04
19	jmca	JavaParserTokenManager	1707	0.50	0.46	0.28	0.45	0.04	0.23
21	geo-google	AddressToUsAddressFunctor	30	0.60	0.62	0.62	0.62	0.08	0.36
21	geo-google	GeoStatusCode	23	1.00	1.00	1.00	1.00	0.76	1.00
22	byuic	JavaScriptCompressor	561	0.21	0.22	0.11	0.20	0.02	0.11
22	byuic	Parser	739	0.44	0.43	0.32	0.38	0.02	0.29
22	byuic	ScriptOrFnScope	39	0.77	0.77	0.69	0.77	0.32	0.69
22	byuic	YUICompressor	87	0.36	0.36	0.35	0.36	0.08	0.31
24	saxpath	Axis	55	1.00	1.00	1.00	1.00	0.83	1.00
24	saxpath	XPathLexer	484	0.85	0.83	0.76	0.73	0.28	0.70
26	jipa	Main	134	0.59	0.58	0.57	0.52	0.18	0.52
26	jipa	Variable	23	1.00	1.00	1.00	1.00	0.98	1.00
27	gangup	AudioManager	32	0.47	0.47	0.47	0.47	0.07	0.47
27	gangup	GroupPanel	25	0.04	0.03	0.03	0.04	0.00	0.03
29	apbsmem	Main	388	0.00	0.00	0.00	0.00	0.00	0.00
29	apbsmem	PlotAxis	41	1.00	1.00	0.99	0.99	0.43	0.94
31	xisemele	OperationsHelperImpl	27	0.55	0.55	0.55	0.55	0.14	0.50
32	httpanalyzer	HttpAnalyzerView	200	0.02	0.01	0.02	0.02	0.00	0.02
32	httpanalyzer	Password	56	0.80	0.80	0.80	0.80	0.35	0.80
33	javaviewcontrol	JVCParser	221	0.44	0.45	0.32	0.43	0.06	0.16
33	javaviewcontrol	JVCParserTokenManager	2373	0.08	0.03	0.02	0.12	0.00	0.01
33	javaviewcontrol	TokenMgrError	32	0.88	0.89	0.82	0.93	0.56	0.85
35	corina	GrapherPanel	290	0.00	0.00	0.00	0.00	0.00	0.00
35	corina	SiteDB	137	0.05	0.02	0.04	0.01	0.00	0.04
35	corina	StandardPlot	69	0.08	0.02	0.07	0.00	0.00	0.07
36	schemaspy	Table	380	0.03	0.03	0.02	0.02	0.00	0.01
36	schemaspy	TableMeta	17	0.39	0.19	0.02	0.12	0.01	0.05
37	petsoar	CreditCardInfo	16	1.00	1.00	1.00	1.00	0.99	1.00
37	petsoar	DefaultLuceneDocumentFactory	47	0.51	0.51	0.51	0.51	0.17	0.51
38	javabullboard	JDBCUtils	141	0.35	0.35	0.35	0.36	0.02	0.26
38	javabullboard	StringUtils	98	0.94	0.93	0.91	0.92	0.37	0.77
39	diffi	IndexedString	20	0.99	0.97	0.97	0.98	0.89	0.96
39	diffi	StringIncrementor	35	0.84	0.83	0.87	0.82	0.61	0.79
40	glengineer	Block	23	0.95	0.95	0.94	0.94	0.29	0.52
40	glengineer	GroupAgent	115	0.67	0.70	0.62	0.59	0.07	0.39
41	follow	SearchableTextPane	35	0.73	0.75	0.68	0.63	0.29	0.62
41	follow	TabbedPane	20	0.31	0.31	0.33	0.28	0.06	0.29
42	asphodel	DefaultRepositoryManager	42	0.02	0.02	0.02	0.02	0.00	0.02
43	lilith	AccessEvent	134	0.91	0.89	0.86	0.87	0.34	0.64
43	lilith	ConditionTableModel	80	0.86	0.49	0.66	0.65	0.28	0.61
43	lilith	MethodRenderer	23	0.78	0.77	0.78	0.69	0.45	0.56
43	lilith	ViewActions	646	0.01	0.01	0.00	0.01	0.00	0.01
44	summa	ExposedTimSort	372	0.64	0.63	0.60	0.54	0.01	0.20
44	summa	NamedCollatorComparator	30	0.84	0.81	0.85	0.84	0.06	0.54
45	lotus	Game	24	0.42	0.42	0.42	0.42	0.08	0.28
45	lotus	Phase	28	0.50	0.50	0.50	0.50	0.27	0.50
46	nutzenportfolio	AuswertungGrafik	21	0.32	0.32	0.30	0.33	0.05	0.30
46	nutzenportfolio	NaOpNuDaoService	48	0.10	0.10	0.10	0.10	0.00	0.10
47	dvd-homevideo	Capture	29	0.10	0.10	0.10	0.10	0.00	0.10
47	dvd-homevideo	Convert	52	0.11	0.11	0.10	0.08	0.00	0.11
47	dvd-homevideo	Menu	84	0.06	0.06	0.06	0.06	0.00	0.06
49	diebierse	DefaultSettingsController	27	0.31	0.28	0.29	0.23	0.03	0.06
49	diebierse	Drink	81	0.95	0.96	0.88	0.92	0.48	0.86
50	biff	Scanner	817	0.13	0.06	0.00	0.12	0.00	0.02
51	jiprof	ClassReader	817	0.51	0.44	0.40	0.44	0.01	0.07
51	jiprof	LocalVariablesSorter	87	0.61	0.62	0.55	0.45	0.04	0.11

(*continued on next page*)

Table A.9 (*continued*)

PID	Project	Subject	# Br.	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
51	jiprof	MethodWriter	824	0.55	0.52	0.43	0.46	0.01	0.11
51	jiprof	Profile	76	0.87	0.88	0.87	0.86	0.32	0.62
52	lagoon	LagoonCLI	65	0.31	0.28	0.27	0.30	0.04	0.28
52	lagoon	LagoonGUI	63	0.02	0.02	0.02	0.02	0.00	0.02
54	db-everywhere	DBEHelper	153	0.35	0.35	0.26	0.28	0.02	0.18
54	db-everywhere	SapdbTableList	13	0.30	0.30	0.27	0.31	0.06	0.27
55	lavalamp	TimeOfDay	18	1.00	1.00	1.00	1.00	0.99	1.00
56	jhandballmoves	CreateMovePdfAction	26	0.04	0.04	0.04	0.04	0.00	0.04
56	jhandballmoves	HandballModelReader	31	0.35	0.35	0.35	0.35	0.10	0.35
57	hft-bomberman	ForwardingObserver	13	0.95	0.92	0.89	0.60	0.54	0.54
57	hft-bomberman	ServerGameModel	128	0.07	0.07	0.10	0.07	0.00	0.07
58	fps370	Fps370Panel	151	0.01	0.01	0.01	0.00	0.00	0.01
58	fps370	TederFrame	70	0.01	0.01	0.01	0.01	0.00	0.01
59	mygrid	AvailableJobsResponse	28	0.82	0.82	0.82	0.82	0.48	0.73
59	mygrid	Fail	28	0.82	0.82	0.82	0.82	0.45	0.73
60	sugar	FSPathExplorer	51	0.65	0.64	0.59	0.64	0.26	0.59
60	sugar	FSPathResult	24	1.00	1.00	1.00	1.00	0.88	1.00
61	noen	DaikonFormatter	71	0.64	0.63	0.61	0.57	0.11	0.34
61	noen	ProbeInformation	67	0.89	0.87	0.84	0.93	0.50	0.63
62	dom4j	AbstractElement	420	0.94	0.92	0.84	0.89	0.11	0.50
62	dom4j	DOMWriter	73	0.03	0.03	0.03	0.00	0.00	0.03
63	objectexplorer	AttributeModelComparator	17	0.99	0.97	0.74	0.45	0.35	0.56
63	objectexplorer	ExplorerFrameEventConverter	175	0.02	0.02	0.02	0.02	0.00	0.02
64	jtailgui	IndexFileAction	13	0.31	0.30	0.30	0.30	0.03	0.23
64	jtailgui	JTailPanel	23	0.87	0.87	0.81	0.82	0.00	0.42
65	gsftp	FtpApplet	29	0.03	0.03	0.03	0.03	0.00	0.03
65	gsftp	SSHSCPGUIThread	91	0.40	0.40	0.35	0.37	0.07	0.18
66	openjms	DefaultConnectionPool	99	0.28	0.28	0.28	0.19	0.00	0.15
66	openjms	SocketRequestInfo	34	1.00	1.00	0.98	1.00	0.67	0.97
67	gae-app-manager	QuotaDetailsParser	47	0.15	0.15	0.15	0.15	0.02	0.14
68	biblestudy	Queue	37	1.00	1.00	1.00	1.00	0.99	1.00
68	biblestudy	Verse	32	0.69	0.69	0.66	0.69	0.16	0.62
69	lhamacaw	DisplayableListPanel	70	0.04	0.06	0.08	0.04	0.00	0.04
69	lhamacaw	MacawWorkBench	23	0.14	0.14	0.13	0.16	0.00	0.11
70	echodep	HaSMETSWebValidator	198	0.01	0.01	0.01	0.01	0.00	0.01
70	echodep	TestHaSMETSProfile	188	0.01	0.01	0.01	0.01	0.00	0.01
71	ext4j	ExtrasPatternParser	11	1.00	1.00	1.00	1.00	1.00	1.00
71	ext4j	Request	139	0.91	0.91	0.90	0.90	0.35	0.80
72	battlecry	battlecryGUI	78	0.03	0.03	0.03	0.03	0.00	0.03
72	battlecry	bcGenerator	281	0.66	0.59	0.26	0.66	0.00	0.12
73	fim1	FontChooserDialog	25	0.16	0.16	0.16	0.16	0.01	0.15
73	fim1	UpdateUserPanel	73	0.11	0.11	0.11	0.11	0.00	0.09
74	fixsuite	Library	35	0.21	0.22	0.24	0.20	0.02	0.23
74	fixsuite	TreeView	72	0.27	0.26	0.26	0.23	0.02	0.17
75	openhre	ExpressionImpl	50	1.00	1.00	1.00	1.00	0.93	1.00
75	openhre	User	97	0.98	0.98	0.97	0.99	0.53	0.90
77	io-project	ClientGroup	66	0.88	0.82	0.89	0.87	0.23	0.54
78	caloriecount	DirectoryScanner	232	0.83	0.82	0.70	0.71	0.13	0.18
78	caloriecount	SimpleComboBox	26	0.70	0.70	0.67	0.66	0.24	0.39
78	caloriecount	SimpleKeyListenerHelper	25	0.97	0.98	0.87	0.81	0.45	0.81
79	twfbplayer	BattleStatistics	156	0.88	0.89	0.89	0.89	0.12	0.53
79	twfbplayer	SimpleSector	74	0.86	0.87	0.76	0.80	0.39	0.66
80	wheelwebtool	ClassReader	817	0.53	0.51	0.45	0.49	0.02	0.19
80	wheelwebtool	ClassWriter	174	0.93	0.91	0.87	0.88	0.24	0.69
80	wheelwebtool	DynamicSelectModel	40	0.67	0.67	0.67	0.62	0.19	0.51
80	wheelwebtool	FieldWriter	58	1.00	1.00	0.95	0.99	0.44	0.90
81	javathena	ConfigurationManagement	190	0.50	0.46	0.41	0.48	0.05	0.42
81	javathena	FromClient	49	0.39	0.39	0.40	0.39	0.14	0.38
81	javathena	Login	255	0.33	0.33	0.29	0.35	0.03	0.21
81	javathena	UserManagement	329	0.15	0.15	0.14	0.15	0.00	0.12
82	ipcalculator	BinaryCalculate	103	0.90	0.90	0.85	0.86	0.39	0.74
82	ipcalculator	WhoIS	55	0.53	0.49	0.54	0.49	0.14	0.33
83	xbus	MessageHandler	27	0.31	0.33	0.33	0.33	0.04	0.28
83	xbus	XBUSClassLoader	17	0.18	0.18	0.18	0.18	0.04	0.15
84	ifx-framework	IFXObject	72	0.43	0.43	0.43	0.43	0.12	0.41
85	shop	JSJshop	114	0.31	0.32	0.38	0.51	0.02	0.19
85	shop	JSPredicateForm	87	0.67	0.66	0.54	0.49	0.03	0.30
85	shop	JSState	41	0.39	0.39	0.42	0.38	0.07	0.24
85	shop	JSTerm	192	0.59	0.58	0.53	0.48	0.03	0.33
86	at-robots2-j	Robot	123	0.51	0.47	0.51	0.18	0.05	0.42
86	at-robots2-j	RobotRenderer	37	0.61	0.52	0.41	0.53	0.10	0.43
87	jaw-br	Abrir	26	0.19	0.19	0.19	0.15	0.01	0.17
88	jopenchart	CoordSystemUtilities	92	0.50	0.46	0.44	0.42	0.03	0.16
88	jopenchart	DefaultChart	20	0.30	0.33	0.30	0.32	0.07	0.28

(*continued on next page*)

Table A.9 (*continued*)

PID	Project	Subject	# Br.	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
89	jiggler	Clip	33	1.00	1.00	0.93	1.00	0.64	0.88
100	jgaap	jgaapGUI	23	0.83	0.69	0.63	0.63	0.32	0.61
101	netweaver	HeapInfo	73	0.93	0.93	0.93	0.93	0.48	0.76
101	netweaver	J2EEApplicationAlias	69	0.93	0.93	0.93	0.93	0.48	0.78
102	squirrel-sql	CascadeInternalFramePositioner	21	0.95	0.90	0.96	0.96	0.20	0.82
102	squirrel-sql	GlobalPreferencesSheet	51	0.06	0.01	0.03	0.00	0.00	0.05
103	sweethome3d	HomeController	618	0.05	0.05	0.06	0.05	0.00	0.03
103	sweethome3d	Room3D	353	0.00	0.00	0.00	0.00	0.00	0.00
103	sweethome3d	RoomController	154	0.03	0.03	0.03	0.04	0.00	0.02
103	sweethome3d	SweetHome3D	228	0.16	0.16	0.15	0.15	0.00	0.10
104	vuze	FeatureManagerUIListener	134	0.02	0.02	0.03	0.01	0.00	0.02
104	vuze	SWTSkinObjectContainer	119	0.01	0.01	0.01	0.01	0.00	0.01
105	freemind	ExportHook	29	0.31	0.32	0.31	0.27	0.04	0.31
105	freemind	JDayChooser	208	0.68	0.40	0.59	0.15	0.21	0.41
106	checkstyle	ProjectConfiguration	37	0.11	0.09	0.10	0.07	0.01	0.11
107	weka	Evaluation	809	0.44	0.43	0.37	0.39	0.00	0.13
107	weka	FindWithCapabilities	247	0.63	0.61	0.53	0.63	0.16	0.54
107	weka	ICSSearchAlgorithm	267	0.25	0.24	0.21	0.21	0.00	0.19
107	weka	Memory	29	0.76	0.76	0.74	0.76	0.48	0.74
108	liferay	DLSyncWrapper	78	1.00	1.00	1.00	0.86	0.55	1.00
109	pdfsam	HelpCmdLineHandler	41	0.63	0.63	0.63	0.63	0.13	0.63
109	pdfsam	JPodThmbnailCallable	19	0.01	0.00	0.01	0.00	0.01	0.01
110	firebird	AbstractJavaGDSImpl	1040	0.28	0.27	0.21	0.23	0.00	0.15
110	firebird	EncodingFactory	195	0.96	0.96	0.79	0.87	0.49	0.92
110	firebird	FBProcedureCall	98	0.85	0.87	0.81	0.83	0.33	0.67

Table A.10
Average AUC scores achieved over 20 independent runs.

PID	Project	Subject	No. of Branches	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
1	tullibee	ComboLeg	21	0.92	0.92	0.93	0.88	0.59	0.91
1	tullibee	ExecutionFilter	20	0.84	0.79	0.86	0.80	0.26	0.74
2	a4j	FileUtil	125	0.48	0.48	0.49	0.49	0.04	0.37
2	a4j	Product	31	0.09	0.09	0.09	0.08	0.00	0.08
4	rif	WebServiceDescriptor	21	0.91	0.90	0.92	0.87	0.52	0.86
5	templateit	Region	31	0.95	0.94	0.91	0.90	0.77	0.95
5	templateit	WorkbookParser	54	0.18	0.16	0.16	0.12	0.02	0.09
7	sfmis	Base64	32	0.97	0.97	0.96	0.94	0.74	0.98
7	sfmis	Loader	82	0.28	0.24	0.25	0.20	0.01	0.13
10	water-simulator	ConsumerAgent	137	0.09	0.09	0.08	0.06	0.00	0.06
11	imsmart	HTMLFilter	15	0.97	0.96	0.98	0.93	0.85	0.98
12	dsachat	Handler	98	0.28	0.25	0.16	0.22	0.02	0.15
12	dsachat	InternalChatFrame	69	0.04	0.04	0.04	0.04	0.00	0.03
13	jdbacl	DBUtil	197	0.03	0.00	0.01	0.00	0.01	0.01
13	jdbacl	SQLUtil	188	0.40	0.20	0.30	0.00	0.04	0.10
14	omjstate	Transition	30	0.91	0.91	0.91	0.83	0.53	0.89
15	beanbin	MethodReflectionCriteria	47	0.65	0.66	0.49	0.52	0.17	0.46
15	beanbin	ReflectionSearch	41	0.61	0.61	0.62	0.58	0.17	0.34
17	inspirento	MainMenu	27	0.65	0.65	0.69	0.65	0.35	0.55
17	inspirento	XmlElement	95	0.84	0.83	0.73	0.69	0.25	0.71
18	jsecurity	AntPathMatcher	170	0.63	0.58	0.49	0.59	0.12	0.48
18	jsecurity	BasicHttpAuthenticationFilter	36	0.39	0.40	0.38	0.34	0.06	0.36
18	jsecurity	DefaultWebSecurityManager	67	0.43	0.40	0.36	0.28	0.03	0.27
19	jmca	JMCAAnalyzer	199	0.38	0.34	0.27	0.41	0.01	0.05
19	jmca	JavaCharStream	216	0.64	0.59	0.45	0.50	0.09	0.49
19	jmca	JavaParser	7938	0.21	0.17	0.09	0.09	0.00	0.02
19	jmca	JavaParserTokenManager	1707	0.36	0.34	0.20	0.31	0.01	0.17
21	geo-google	AddressToUsAddressFunctor	30	0.40	0.41	0.44	0.36	0.03	0.24
21	geo-google	GeoStatusCode	23	0.94	0.94	0.89	0.89	0.60	0.92
22	byuic	JavaScriptCompressor	561	0.15	0.12	0.08	0.09	0.02	0.09
22	byuic	Parser	739	0.37	0.36	0.25	0.31	0.01	0.26
22	byuic	ScriptOrFnScope	39	0.73	0.73	0.61	0.68	0.37	0.69
22	byuic	YUICompressor	87	0.33	0.32	0.32	0.33	0.07	0.29
24	saIath	Axis	55	0.94	0.95	0.93	0.90	0.72	0.97
24	saxpath	XPathLexer	484	0.65	0.62	0.51	0.53	0.17	0.58
26	jipa	Main	134	0.46	0.46	0.44	0.31	0.10	0.41
26	jipa	Variable	23	0.97	0.97	0.97	0.90	0.84	0.97
27	gangup	AudioManager	32	0.44	0.44	0.43	0.42	0.08	0.42
27	gangup	GroupPanel	25	0.03	0.02	0.02	0.02	0.00	0.02

(*continued on next page*)

Table A.10 (*continued*)

PID	Project	Subject	No. of Branches	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
29	apbsmem	Main	388	0.00	0.00	0.00	0.00	0.00	0.00
29	apbsmem	PlotAxis	41	0.82	0.73	0.77	0.62	0.18	0.69
31	xisemele	OperationsHelperImpl	27	0.48	0.48	0.46	0.37	0.09	0.42
32	httpanalyzer	HttpAnalyzerView	200	0.01	0.01	0.01	0.01	0.00	0.01
32	httpanalyzer	Password	56	0.74	0.77	0.78	0.71	0.46	0.74
33	javaviewcontrol	JVCParser	221	0.33	0.30	0.23	0.29	0.03	0.12
33	javaviewcontrol	JVCParserTokenManager	2373	0.03	0.01	0.01	0.05	0.00	0.01
33	javaviewcontrol	TokenMgrError	32	0.79	0.79	0.77	0.74	0.47	0.76
35	corina	GrapherPanel	290	0.00	0.00	0.00	0.00	0.00	0.00
35	corina	SiteDB	137	0.03	0.02	0.02	0.00	0.00	0.02
35	corina	StandardPlot	69	0.06	0.01	0.04	0.00	0.00	0.02
36	schemaspy	Table	380	0.02	0.01	0.01	0.01	0.00	0.01
36	schemaspy	TableMeta	17	0.14	0.09	0.00	0.04	0.00	0.03
37	petsoar	CreditCardInfo	16	0.95	0.95	0.96	0.89	0.83	0.96
38	javabullboard	JDBCUtils	47	0.33	0.33	0.32	0.32	0.01	0.23
38	javabullboard	StringUtils	141	0.85	0.84	0.78	0.72	0.25	0.69
39	diffi	IndexedString	98	0.87	0.86	0.82	0.81	0.59	0.83
39	diffi	StringIncrementor	20	0.78	0.76	0.78	0.63	0.48	0.76
40	glengineer	Block	35	0.77	0.76	0.75	0.69	0.12	0.32
40	glengineer	GroupAgent	23	0.51	0.51	0.47	0.36	0.04	0.31
41	follow	SearchableTextPane	115	0.55	0.53	0.51	0.44	0.14	0.47
41	follow	TabbedPane	35	0.25	0.25	0.26	0.19	0.04	0.22
42	asphodel	DefaultRepositoryManager	20	0.02	0.02	0.02	0.02	0.00	0.02
43	lilith	AccessEvent	42	0.73	0.69	0.62	0.60	0.19	0.53
43	lilith	ConditionTableModel	134	0.78	0.41	0.58	0.37	0.14	0.47
43	lilith	MethodRenderer	80	0.69	0.69	0.69	0.52	0.25	0.46
43	lilith	ViewActions	23	0.01	0.00	0.00	0.00	0.00	0.00
44	summa	ExposedTimSort	646	0.55	0.51	0.41	0.47	0.00	0.14
44	summa	NamedCollatorComparator	372	0.73	0.69	0.70	0.66	0.01	0.43
45	lotus	Game	30	0.37	0.36	0.37	0.30	0.07	0.29
45	lotus	Phase	24	0.48	0.48	0.48	0.47	0.26	0.48
46	nutzenportfolio	AuswertungGrafik	28	0.26	0.26	0.26	0.23	0.03	0.23
46	nutzenportfolio	NaOpNuDaoService	21	0.09	0.09	0.08	0.08	0.00	0.09
47	dvd-homevideo	Capture	48	0.10	0.09	0.10	0.09	0.00	0.09
47	dvd-homevideo	Convert	29	0.09	0.09	0.08	0.06	0.00	0.07
47	dvd-homevideo	Menu	52	0.05	0.05	0.05	0.05	0.00	0.05
49	diebierse	DefaultSettingsController	84	0.15	0.13	0.18	0.13	0.01	0.03
49	diebierse	Drink	27	0.81	0.81	0.72	0.68	0.28	0.64
50	biff	Scanner	81	0.06	0.01	0.00	0.07	0.00	0.01
51	jiprof	ClassReader	817	0.40	0.36	0.06	0.14	0.00	0.02
51	jiprof	LocalVariablesSorter	817	0.27	0.28	0.30	0.18	0.01	0.04
51	jiprof	MethodWriter	87	0.46	0.42	0.30	0.34	0.00	0.09
51	jiprof	Profile	824	0.70	0.74	0.67	0.65	0.20	0.52
52	lagoon	LagoonCLI	76	0.26	0.23	0.23	0.25	0.04	0.23
52	lagoon	LagoonGUI	65	0.01	0.01	0.01	0.01	0.00	0.01
54	db-everywhere	DBEHelper	63	0.27	0.17	0.12	0.15	0.01	0.10
54	db-everywhere	SapdbTableList	153	0.28	0.27	0.24	0.27	0.05	0.23
55	lavalamp	TimeOfDay	13	0.96	0.96	0.95	0.93	0.84	0.96
56	jhandballmoves	CreateMovePdfAction	18	0.03	0.03	0.04	0.03	0.00	0.03
56	jhandballmoves	HandballModelReader	26	0.32	0.33	0.33	0.32	0.08	0.32
57	hft-bomberman	ForwardingObserver	31	0.69	0.67	0.61	0.32	0.08	0.31
57	hft-bomberman	ServerGameModel	13	0.05	0.05	0.07	0.01	0.00	0.04
58	fps370	Fps370Panel	128	0.01	0.01	0.01	0.00	0.00	0.01
58	fps370	TederFrame	151	0.01	0.01	0.01	0.01	0.00	0.01
59	mygrid	AvailableJobsResponse	70	0.75	0.74	0.69	0.69	0.39	0.67
59	mygrid	Fail	28	0.75	0.74	0.72	0.69	0.39	0.68
60	sugar	FSPathExplorer	28	0.58	0.55	0.53	0.48	0.18	0.51
60	sugar	FSPathResult	51	0.96	0.96	0.95	0.90	0.71	0.96
61	noen	DaikonFormatter	24	0.52	0.48	0.47	0.42	0.07	0.27
61	noen	ProbeInformation	71	0.69	0.67	0.61	0.68	0.18	0.54
62	dom4j	AbstractElement	67	0.83	0.79	0.67	0.75	0.08	0.43
62	dom4j	DOMWriter	420	0.02	0.02	0.02	0.00	0.00	0.02
63	objectexplorer	AttributeModelComparator	73	0.61	0.62	0.50	0.32	0.10	0.39
63	objectexplorer	ExplorerFrameEventConverter	17	0.02	0.02	0.02	0.02	0.00	0.02
64	jtailgui	IndexFileAction	175	0.24	0.24	0.23	0.21	0.02	0.17
64	jtailgui	JTailPanel	13	0.58	0.62	0.47	0.49	0.00	0.24
65	gsftp	FtpApplet	23	0.03	0.03	0.03	0.03	0.00	0.03
65	gsftp	SSHSCPGUIThread	29	0.29	0.29	0.27	0.26	0.04	0.14
66	openjms	DefaultConnectionPool	91	0.14	0.15	0.16	0.10	0.00	0.09
66	openjms	SocketRequestInfo	99	0.91	0.89	0.84	0.82	0.41	0.81
67	gae-app-manager	QuotaDetailsParser	34	0.13	0.14	0.14	0.12	0.02	0.13
68	biblestudy	Queue	47	0.97	0.96	0.96	0.91	0.82	0.97
68	biblestudy	Verse	37	0.62	0.61	0.58	0.55	0.11	0.54
69	lhamacaw	DisplayableListPanel	32	0.04	0.04	0.04	0.04	0.00	0.04

(*continued on next page*)

Table A.10 (*continued*)

PID	Project	Subject	No. of Branches	Dyna-MOSA	MOSA	MIO	WS	Single	LIPS
69	lhamacaw	MacawWorkBench	70	0.13	0.13	0.12	0.14	0.00	0.10
70	echodep	HaSMETSWebValidator	23	0.01	0.01	0.01	0.00	0.00	0.00
70	echodep	TestHaSMETSProfile	198	0.01	0.01	0.01	0.01	0.00	0.01
71	ext4j	ExtrasPatternParser	188	0.97	0.96	0.96	0.92	0.88	0.97
71	ext4j	Request	11	0.80	0.77	0.73	0.70	0.24	0.70
72	battlecry	battlecryGUI	139	0.02	0.02	0.02	0.02	0.00	0.02
72	battlecry	bcGenerator	78	0.56	0.42	0.13	0.40	0.00	0.09
73	fim1	FontChooserDialog	281	0.14	0.13	0.14	0.13	0.00	0.13
73	fim1	UpdateUserPanel	25	0.07	0.07	0.06	0.09	0.00	0.04
74	fixsuite	Library	73	0.16	0.17	0.18	0.06	0.00	0.15
74	fixsuite	TreeView	35	0.20	0.20	0.19	0.12	0.01	0.13
75	openhre	ExpressionImpl	72	0.96	0.95	0.94	0.93	0.78	0.95
75	openhre	User	50	0.89	0.88	0.84	0.81	0.38	0.79
77	io-project	ClientGroup	97	0.69	0.63	0.67	0.60	0.16	0.44
78	caloriecount	DirectoryScanner	66	0.61	0.56	0.45	0.51	0.05	0.14
78	caloriecount	SimpleComboBox	232	0.53	0.48	0.53	0.44	0.12	0.31
78	caloriecount	SimpleKeyListenerHelper	26	0.72	0.71	0.65	0.51	0.19	0.56
79	twfbplayer	BattleStatistics	25	0.78	0.78	0.76	0.77	0.07	0.47
79	twfbplayer	SimpleSector	156	0.71	0.70	0.61	0.59	0.26	0.59
80	wheelwebtool	ClassReader	74	0.43	0.39	0.31	0.37	0.01	0.11
80	wheelwebtool	ClassWriter	817	0.74	0.70	0.66	0.60	0.14	0.57
80	wheelwebtool	DynamicSelectModel	174	0.52	0.51	0.51	0.39	0.18	0.34
80	wheelwebtool	FieldWriter	40	0.60	0.62	0.71	0.54	0.18	0.66
81	javathena	ConfigurationManagement	58	0.42	0.40	0.35	0.37	0.04	0.38
81	javathena	FromClient	190	0.32	0.31	0.32	0.29	0.09	0.29
81	javathena	Login	49	0.25	0.23	0.20	0.19	0.01	0.17
81	javathena	UserManagement	255	0.12	0.10	0.09	0.07	0.00	0.09
82	ipcalculator	BinaryCalculate	329	0.76	0.74	0.70	0.70	0.23	0.65
82	ipcalculator	WhoIS	103	0.41	0.34	0.40	0.39	0.10	0.28
83	xbus	MessageHandler	55	0.28	0.28	0.29	0.26	0.03	0.26
83	xbus	XBUSClassLoader	27	0.16	0.17	0.17	0.16	0.04	0.14
84	ifx-framework	IFXObject	17	0.39	0.38	0.37	0.38	0.08	0.37
85	shop	JSJshop	72	0.29	0.30	0.38	0.25	0.02	0.18
85	shop	JSPredicateForm	114	0.52	0.52	0.38	0.26	0.01	0.20
85	shop	JSState	87	0.32	0.32	0.34	0.26	0.04	0.20
85	shop	JSTerm	41	0.41	0.39	0.35	0.28	0.01	0.24
86	at-robots2-j	Robot	192	0.37	0.31	0.28	0.05	0.04	0.15
86	at-robots2-j	RobotRenderer	123	0.46	0.43	0.38	0.44	0.07	0.39
87	jaw-br	Abrir	37	0.15	0.14	0.12	0.09	0.01	0.09
88	jopenchart	CoordSystemUtilities	26	0.33	0.31	0.30	0.27	0.02	0.10
88	jopenchart	DefaultChart	92	0.27	0.28	0.27	0.27	0.05	0.26
89	jiggler	Clip	20	0.78	0.85	0.77	0.86	0.42	0.76
100	jgaap	jgaapGUI	33	0.70	0.57	0.53	0.48	0.16	0.46
101	netweaver	HeapInfo	23	0.81	0.80	0.78	0.72	0.32	0.63
101	netweaver	J2EEApplicationAlias	73	0.83	0.81	0.80	0.78	0.37	0.68
102	squirrel-sql	CascadeInternalFramePositioner	69	0.79	0.76	0.78	0.76	0.13	0.61
102	squirrel-sql	GlobalPreferencesSheet	21	0.04	0.01	0.02	0.00	0.00	0.03
103	sweethome3d	HomeController	51	0.02	0.02	0.02	0.02	0.00	0.02
103	sweethome3d	Room3D	618	0.00	0.00	0.00	0.00	0.00	0.00
103	sweethome3d	RoomController	353	0.02	0.01	0.01	0.02	0.00	0.01
103	sweethome3d	SweetHome3D	154	0.13	0.13	0.12	0.11	0.00	0.08
104	vuze	FeatureManagerUIListener	228	0.02	0.02	0.02	0.01	0.00	0.01
104	vuze	SWTSkinObjectContainer	134	0.01	0.01	0.01	0.01	0.00	0.01
105	freemind	ExportHook	119	0.29	0.29	0.28	0.21	0.03	0.27
105	freemind	JDayChooser	29	0.55	0.31	0.46	0.07	0.10	0.35
106	checkstyle	ProjectConfiguration	208	0.10	0.08	0.10	0.07	0.00	0.08
107	weka	Evaluation	37	0.34	0.33	0.25	0.29	0.00	0.09
107	weka	FindWithCapabilities	809	0.49	0.46	0.44	0.47	0.10	0.42
107	weka	ICSSearchAlgorithm	247	0.13	0.13	0.13	0.13	0.00	0.10
107	weka	Memory	267	0.57	0.64	0.59	0.47	0.08	0.55
108	liferay	DLSyncWrapper	29	0.92	0.92	0.86	0.74	0.42	0.81
108	liferay	FilterMapping	78	0.05	0.04	0.03	0.03	0.00	0.03
109	pdfsam	HelpCmdLineHandler	41	0.55	0.53	0.51	0.44	0.09	0.50
109	pdfsam	JPodThmbnailCallable	19	0.00	0.00	0.00	0.00	0.00	0.00
110	firebird	AbstractJavaGDSImpl	1040	0.24	0.22	0.16	0.20	0.00	0.13
110	firebird	EncodingFactory	195	0.80	0.78	0.63	0.65	0.31	0.73
110	firebird	FBProcedureCall	98	0.75	0.76	0.69	0.67	0.21	0.61

References

[1] U. Rueda, R. Just, J.P. Galeotti, T.E.J. Vos, Unit testing tool competition: round four, Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST '16, ACM, New York, NY, USA, 2016, pp. 19–28, https://doi.org/10.1145/2897010.2897018.

[2] A. Panichella, U.R. Molina, Java unit testing tool competition: fifth round, Proceedings of the 10th International Workshop on Search-Based Software Testing (SBST), IEEE Press, 2017, pp. 32–38.

[3] U. Molina, F. Kifetew, A. Panichella, Java unit testing tool competition — sixth round, Proceedings of the 10th International Workshop on Search-Based Software Testing (SBST), (2018), pp. 22–29.

[4] F. Shull, J.C. Carver, S. Vegas, N.J. Juzgado, The role of replications in empirical software engineering, Empir. Software Eng. 13 (2) (2008) 211–218, https://doi.org/10.1007/s10664-008-9060-1.

[5] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, ACM Trans. Softw. Eng. Methodol. 24 (2) (2014) 8:1–8:42, https://doi.org/10.1145/2685612.

[6] P. McMinn, Search-based software test data generation: a survey, Softw. Test. Verif. Reliab. 14 (2) (2004) 105–156.

[7] F.M. Kifetew, A. Panichella, A.D. Lucia, R. Oliveto, P. Tonella, Orthogonal exploration of the search space in evolutionary test case generation, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), (2013), pp. 257–267.

[8] G. Fraser, A. Arcuri, Whole test suite generation, IEEE Trans. Software Eng. 39 (2) (2013) 276–291.

[9] G. Fraser, A. Arcuri, EvoSuite: automatic test suite generation for object-oriented software, Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, (2011), pp. 416–419.

[10] M. Harman, A multiobjective approach to search-based test data generation, Association for Computer Machinery, ACM Press. To, 2007, pp. 1029–1036.

[11] J. Ferrer, F. Chicano, E. Alba, Evolutionary algorithms for the multi-objective test data generation problem, Software Pract. Experience 42 (11) (2012) 1331–1362.

[12] G. Pinto, S. Vergilio, A multi-objective genetic algorithm to test data generation, 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), 1 (2010), pp. 129–134.

[13] A. Panichella, F.M. Kifetew, P. Tonella, Reformulating branch coverage as a many-objective optimization problem, 8th IEEE International Conference on Software Testing, Verification and Validation, ICST, (2015), pp. 1–10.

[14] A. Panichella, F. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Trans. Software Eng. (99) (2017) 1, https://doi.org/10.1109/TSE.2017.2663435. PP

[15] A. Arcuri, Many Independent Objective (MIO) Algorithm for Test Suite Generation, Springer International Publishing, Cham, pp. 3–17. doi:10.1007/978-3-319-66299-2_1.

[16] A. Arcuri, G. Fraser, On the effectiveness of whole test suite generation, Search-Based Software Engineering, Lecture Notes in Computer Science 8636 Springer International Publishing, 2014, pp. 1–15.

[17] S. Panichella, A. Panichella, M. Beller, A. Zaidman, H.C. Gall, The impact of test case summaries on bug fixing performance: an empirical investigation, Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 547–558, https://doi.org/10.1145/2884781.2884847.

[18] G. Fraser, A. Arcuri, Achieving scalable mutation-based generation of whole test suites, Empirical Software Eng. 20 (3) (2015) 783–812, https://doi.org/10.1007/s10664-013-9299-z.

[19] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, A. De Lucia, Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach?, Springer International Publishing, Cham, pp. 64–79. doi:10.1007/978-3-319-47106-8_5.

[20] A. Panichella, F.M. Kifetew, P. Tonella, LIPS vs MOSA: A replicated empirical study on automated test case generation, Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9–11, 2017, Proceedings, (2017), pp. 83–98, https://doi.org/10.1007/978-3-319-66299-2_6.

[21] P. Tonella, Evolutionary testing of classes, ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04, ACM, 2004, pp. 119–128.

[22] J.M. Rojas, M. Vivanti, A. Arcuri, G. Fraser, A detailed investigation of the effectiveness of whole test suite generation, Empirical Software Eng. 22 (2) (2017) 852–893, https://doi.org/10.1007/s10664-015-9424-2.

[23] A. Arcuri, Test suite generation with the many independent objective (mio) algorithm, Inf. Software Technol. (2018).

[24] G. Grano, T.V. Titov, S. Panichella, H.C. Gall, How high will it be? using machine learning models to predict branch coverage in automated testing, 2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE@SANER 2018, Campobasso, Italy, March 20, 2018, (2018), pp. 19–24.

[25] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Inf. Software Technol. 43 (14) (2001) 841–854.

[26] B. Korel, Automated software test data generation, IEEE Trans. Software Eng. 16 (8) (1990) 870–879.

[27] J.M. Rojas, J. Campos, M. Vivanti, G. Fraser, A. Arcuri, Combining multiple coverage criteria in search-based unit test generation, International Symposium on Search Based Software Engineering, Springer, 2015, pp. 93–108.

[28] K. Ayari, S. Bouktif, G. Antoniol, Automatic mutation test input data generation via ant colony, Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, 2007, pp. 1074–1081.

[29] A. Windisch, S. Wappler, J. Wegener, Applying particle swarm optimization to software testing, Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, 2007, pp. 1121–1128.

[30] M. Harman, P. McMinn, A theoretical and empirical study of search-based testing: local, global, and hybrid search, IEEE Trans. Software Eng. 36 (2) (2010) 226–247.

[31] N. Tracey, J. Clark, K. Mander, J. McDermid, An automated framework for structural test-data generation, Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on, IEEE, 1998, pp. 285–288.

[32] A. Arcuri, M.Z. Iqbal, L. Briand, Formal analysis of the effectiveness and predictability of random testing, Proceedings of the 19th international symposium on Software testing and analysis, ACM, 2010, pp. 219–230.

[33] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast elitist multi-objective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comp. 6 (2000) 182–197.

[34] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical investigation in search-based software engineering, Empirical Software Eng. 18 (3) (2013) 594–623.

[35] T.J. McCabe, A complexity measure, IEEE Trans. Software Eng. (4) (1976) 308–320.

[36] S. Shamshiri, J.M. Rojas, G. Fraser, P. McMinn, Random or genetic algorithm search for object-oriented test suite generation? Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, ACM, New York, NY, USA, 2015, pp. 1367–1374, https://doi.org/10.1145/2739480.2754696.

[37] K. Deb, D. Deb, Analysing mutation schemes for real-parameter genetic algorithms, Int. J. Artif. Intell. Software Comput. 4 (1) (2014) 1–28.

[38] A. Arcuri, G. Fraser, Java enterprise edition support in search-based junit test generation, International Symposium on Search Based Software Engineering, Springer, 2016, pp. 3–17.

[39] J. Campos, G. Fraser, A. Arcuri, R. Abreu, Continuous test generation on guava, International Symposium on Search Based Software Engineering, Springer, 2015, pp. 228–234.

[40] J.P. Galeotti, G. Fraser, A. Arcuri, Improving search-based test suite generation with dynamic symbolic execution, Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, IEEE, 2013, pp. 360–369.

[41] S. García, D. Molina, M. Lozano, F. Herrera, A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization, J. Heuristics 15 (6) (2008) 617, https://doi.org/10.1007/s10732-008-9080-4.

[42] W.J. Conover, Practical Nonparametric Statistics, 3rd edition, Wiley, 1998.

[43] R.D. Baker, Modern permutation test software, Marcel Decker, 1995. Randomization Tests, chapter Appendix.

[44] J. Campos, Y. Ge, G. Fraser, M. Eler, A. Arcuri, An empirical evaluation of evolutionary algorithms for test suite generation, Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9–11, 2017, Proceedings, (2017), pp. 33–48, https://doi.org/10.1007/978-3-319-66299-2_3.

[45] G. Gay, Generating effective test suites by combining coverage criteria, Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9–11, 2017, Proceedings, (2017), pp. 65–82, https://doi.org/10.1007/978-3-319-66299-2_5.

[46] R. Just, D. Jalali, M.D. Ernst, Defects4j: a database of existing faults to enable controlled testing studies for java programs, International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21, - 26, 2014, (2014), pp. 437–440, https://doi.org/10.1145/2610384.2628055.

[47] S. Shamshiri, J.M. Rojas, G. Fraser, P. McMinn, Random or genetic algorithm search for object-oriented test suite generation? Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11–15, 2015, (2015), pp. 1367–1374, https://doi.org/10.1145/2739480.2754696.