Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets

Annibale Panichella*, Fitsum Meshesha Kifetew†, Paolo Tonella†
*SnT - University of Luxembourg, Luxembourg
†Fondazione Bruno Kessler, Trento, Italy
annibale.panichella@uni.lu, kifetew@fbk.eu, tonella@fbk.eu

Abstract—
The test case generation is intrinsically a multi-objective problem, since the goal is covering multiple test targets (e.g., branches). Existing search-based approaches either consider one target at a time or aggregate all targets into a single fitness function (whole-suite approach). Multi and many-objective optimisation algorithms (MOAs) have never been applied to this problem, because existing algorithms do not scale to the number of coverage objectives that are typically found in real-world software. In addition, the final goal for MOAs is to find alternative trade-off solutions in the objective space, while in test generation the interesting solutions are only those test cases covering one or more uncovered targets.
In this paper, we present DynaMOSA (Dynamic Many-Objective Sorting Algorithm), a novel many-objective solver specifically designed to address the test case generation problem in the context of coverage testing. DynaMOSA extends our previous many-objective technique MOSA (Many-Objective Sorting Algorithm) with dynamic selection of the coverage targets based on the control dependency hierarchy. Such extension makes the approach more effective and efficient in case of limited search budget.
We carried out an empirical study on 346 Java classes using three coverage criteria (i.e., statement, branch, and strong mutation coverage) to assess the performance of DynaMOSA with respect to the whole-suite approach (WS), its archive-based variant (WSA) and MOSA. The results show that DynaMOSA outperforms WSA in 28% of the classes for branch coverage (+8% more coverage on average) and in 27% of the classes for mutation coverage (+11% more killed mutants on average). It outperforms WS in 51% of the classes for statement coverage, leading to +11% more coverage on average. Moreover, DynaMOSA outperforms its predecessor MOSA for all the three coverage criteria in 19% of the classes with +8% more code coverage on average.

Index Terms—Evolutionary Testing; Many-Objective Optimisation; Automatic Test Case Generation

1 Introduction

Automated structural test case generation aims at producing a set of test cases that maximises coverage of the test goals in the software under test according to the selected adequacy testing criterion (e.g., branch coverage). Search-based test case generators use meta-heuristic optimisation algorithms, such as *genetic algorithms* (GA), to produce new test cases from the previously generated (initially, random) ones, so as to reduce and eventually nullify their distance from each of the yet uncovered targets [52], [37]. The search for a test sequence and for test input data that bring the test execution closer to the current coverage target is guided by a fitness function, which quantifies the distance between the execution trace of a test case and the coverage target. The actual computation of such a distance depends on the type of coverage criterion under consideration. For instance, for branch coverage, the distance is based on the number of control dependencies that separate the execution trace from the target (*approach level*) and on the variable values evaluated at the conditional expression where the execution diverges from the target (*branch distance*).

In traditional evolutionary testing, to cover all targets a meta-heuristic algorithm is applied multiple times, each time with a different target, until all targets are covered or the total search budget is consumed. The final test suite is thus composed of all the test cases that cover one or more targets, including those that *accidentally* cover previously uncovered targets (*collateral coverage*).

Searching for one target at a time is problematic in several respects. First of all, targets may require higher or lower search effort depending on how difficult it is to produce test cases that cover them. Hence, uniformly distributing the search budget across all targets might be largely sub-optimal. Even worse, the presence of unfeasible targets wastes entirely the search time devoted to their coverage. The *whole-suite* (WS) approach [17], [3] to test case generation is a recent attempt to address these issues. WS optimises entire test suites, not just individual test cases, and makes use of a single fitness value that aggregates the fitness values measured for the test cases contained in a test suite, so as to take into consideration all coverage targets simultaneously. In fact, each test case in a test suite is associated with the target closest to its execution trace. The sum over all test cases of such minimum distances from the targets provides the overall, test-suite-level fitness. The additive combination of multiple targets into a single, scalar objective function is known as sum scalarization in

the theory of optimisation [11]. The aim is to apply single-objective algorithms, like GA, to an intrinsically multi-objective problem.

While being more effective than targeting one target at a time, the WS approach suffers all the well-known problems of sum scalarization in many-objective optimisation, among which the inefficient convergence occurring in the non-convex regions of the search space [11]. On the other hand, there are single-objective problem instances on which many-objective algorithms have been shown to be much more convenient than the single-objective approach. In fact, reformulating a complex single-objective problem as a many-objective problem defined on multiple but simpler objectives can reduce the probability that the search remains trapped in local optima, eventually leading to faster convergence [30], [21]. However, there are two main challenges to address when applying many-objective optimisation to test case generation: (i) none of the available multi or many-objective solvers can scale to the number of objectives (targets) typically found in coverage testing [3]; and (ii) multi and many-objective solvers are designed to increase the diversity of the solutions, not just to fully achieve each objective individually (reducing to zero its distance from the target), as required in test case generation [33].

To overcome the aforementioned limitations, in our previous work [42] we introduced MOSA (Many-Objective Sorting Algorithm), a many-objective GA tailored to the test case generation problem. MOSA has three main features: (i) it uses a novel *preference criterion* instead of ranking the solutions based on their Pareto optimality; (ii) it focuses the search only on the yet uncovered coverage targets; (iii) it saves all test cases satisfying one or more previously uncovered targets into an *archive*, which contains the final test suite when the search ends.

Recently, Rojas et al. [48] developed the Whole Suite with Archive approach (WSA), a hybrid strategy that incorporates some of MOSA's routines inside the traditional WS approach. While WSA still applies the sum scalarization and works at the test suite level, it incorporates an archive strategy which operates at the test case level. Moreover, it also focuses the search only on the yet to cover targets. From an empirical point of view, WSA has been proved to be statistically superior to WS and to the one-target at a time approaches. However, from a theoretical point of view, it does not evolve test suites anymore since the final test suite given to developers is artificially synthesised by taking those test cases stored in the archive rather than returning the best individual (i.e., test suite) from the last generation of GAs [48]. Rojas et al. [48] arose the following still unanswered questions: (1) *To what extent do the benefits of MOSA stem from the many-objective reformulation of the problem or from the use of the archiving mechanism?* (2) *How does MOSA performs compared to WSA since both of them implement an archiving strategy?*

In this paper, we provide an in-depth analysis of many-objective approaches to test case generation, thus, answering the aforementioned open questions. First, we present DynaMOSA (Many-Objective Sorting Algorithm with Dynamic target selection) which extends MOSA with the capability to dynamically focus the search on a subset of the yet uncovered targets, based on the control dependency hierarchy. Thus, uncovered targets that are reachable from other uncovered targets positioned higher in the hierarchy are removed from the current vector of objectives. They are reinserted in the objective vector dynamically, when their parents are covered. Since DynaMOSA optimises a subset of the objectives considered by MOSA using the same many-objective algorithm and preference criterion, DynaMOSA is ensured to be equally or more efficient than MOSA.

Second, we conduct a large empirical study involving 346 Java classes sampled from four different datasets [18], [53], [49], [42] and considering three well known test adequacy criteria: statement coverage, branch coverage and mutation coverage. We find that DynaMOSA achieves significantly higher coverage than WSA in a large number of classes (27% and 28% for branch and mutation coverage respectively) with an average increment, for classes where statistically significant differences were observed, of +8% for branch coverage and +11% for mutation coverage. It also produces higher statement coverage than WS[1] in 51% of the classes, for which we observe +11% of covered statements on average. As predicted by the theory, DynaMOSA also improves MOSA for all the three criteria. This happens in 19% of the classes with an average coverage improvement of 8%.

This paper extends our previous work [42] with the following novel contributions:

1) DynaMOSA: a novel algorithm for dynamic target selection, which is theoretically proven as subsuming MOSA.
2) Larger scale experiments: one order of magnitude more classes are considered in the new experiments.
3) Empirical comparison of DynaMOSA with MOSA and WSA to provide support to the usage of many-objective solvers in addition to the archiving strategy, which is implemented in all the three approaches.
4) Empirical assessment of the selected algorithms with respect to different testing criteria, i.e., branch coverage, statement coverage and strong mutation coverage.
5) Direct comparison with Pareto dominance ranking based algorithms, i.e., NSGA-II enriched with an archiving strategy.
6) Empirical assessment of our novel preference criterion, to understand if the higher effectiveness of DynaMOSA is due to the preference criterion alone or to its combination with dominance ranking.

The remainder of this paper is organised as follows. Section 2 presents the reformulation of the test case generation problem as a many-objective problem. Section 3 presents the many-objective test generator MOSA, followed by its extension, DynaMOSA, with dynamic selection of the coverage targets. Section 4 presents the description of the experiments we conducted for the evaluation of the proposed algorithm. Section 5 reports the results obtained from the experiments, while Sections 6 and 7 provide additional empirical analyses and discuss the threats to validity,

1. For statement coverage, we consider WS as baseline since no implementation of WSA is available.

respectively. Section 8 summarises the research works most closely related to ours. Section 9 draws conclusions and identifies possible directions for future work.

2 PROBLEM FORMULATION

This section presents our many-objective reformulation of the structural coverage problem. First the single-objective formulation used in the *whole-suite* approach [17] is described, followed by our many-objective reformulation, highlighting its main differences and peculiarities.

2.1 Single Objective Formulation

In the *whole-suite* approach, a candidate solution is a test suite consisting of a variable number of individual test cases, where each test case is a sequence of method calls of variable length. The fitness function measures how close the candidate test suite is to satisfy all *coverage targets* (aka, *test goals*). It is computed with respect to full coverage (e.g., full branch coverage), as the sum of the individual distances to all coverage targets (e.g., branches) in the program under test. More formally, the problem of finding a test suite that satisfies all test targets has been defined as follows:

***Problem* 2.1:** *Let $U = \{u_1, \ldots, u_k\}$ be the set of structural test targets of the program under test. Find a test suite $T = \{t_1, \ldots, t_n\}$ that minimises the fitness function:*

$$\min f_U(T) = \sum_{u \in U} d(u, T) \tag{1}$$

where $d(u, T)$ denotes the minimal distance for the target u according to a specific distance function.

In general, the function to minimise d is such that $d(u, T) = 0$ if and only if the target goal is covered when a test suite T is executed. The difference between the various coverage criteria affects the specific distance function d used to express how distant the execution traces are from covering the test targets in U when all test cases in T are executed.

In **Branch Coverage**, the test targets to cover are the conditional statement branches in the class under test. Therefore, Problem 2.1 can be instantiated as finding a test suite that covers all branches, using as function d the traditional branch distance [37] for each individual branch to be covered. More formally [17]:

***Problem* 2.2:** *Let $B = \{b_1, \ldots, b_k\}$ be the set of branches in a class. Find a test suite $T = \{t_1, \ldots, t_n\}$ that covers all the feasible branches, i.e., one that minimises the following fitness function:*

$$\min f_B(T) = |M| - |M_T| + \sum_{b \in B} d(b, T) \tag{2}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods by all test cases in T and $d(b, T)$ denotes the minimal normalised branch distance for branch $b \in B$.

The term $|M| - |M_T|$ accounts for the entry edges of the methods that are not executed by T. The minimal normalised branch distance $d(b, t)$ for each branch $b \in B$ is defined as [17]:

$$d(b,t) = \begin{cases} 0 & \text{if } b \text{ has been covered} \\ \dfrac{D_{min}(t \in T, b)}{D_{min}(t \in T, b) + 1} & \text{if the predicate has been executed at least twice} \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

where $D_{min}(t \in T, b)$ is the minimal non-normalised branch distance, computed according to any of the available branch distance computation schemes [37]; minimality here refers to the possibility that the predicate controlling a branch is executed multiple times within a test case or by different test cases. The minimum is taken across all such executions.

In **Statement Coverage**, the optimal test suite is the one that executes all statements in the code. To reach a given statement s, it is sufficient to execute a branch on which such a statement is directly control dependent. Thus, the distance function d for a statement s can be measured using the branch distances for the branches to be executed in order to reach s. More specifically, the problem is [17]:

***Problem* 2.3:** *Let $S = \{s_1, \ldots, s_k\}$ be the set of statements in a class. Find a test suite $T = \{t_1, \ldots, t_n\}$ that covers all the feasible statements, i.e., one that minimises the following fitness function:*

$$\min f_S(T) = |M| - |M_T| + \sum_{b \in B_S} d(b, T) \tag{4}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods by all test cases in T; B_S is the set of branches that hold a direct control dependency on the statements in S; and $d(b, T)$ denotes the minimal normalised branch distance for branch $b \in B_S$.

In **Strong Mutation Coverage**, the test targets to cover are *mutants*, i.e., variants of the original class obtained by injecting *artificial* modifications (mimicking real faults) into them. Therefore, the optimal test suite is the one which is able to cover (*kill*, in the mutation analysis jargon) all mutants. A test case *strongly kills* a mutant if and only if the observable object state or the method return values differ between original and mutated class under test. Such a condition is usually checked by means of automatically generated assertions, which assert the equality of method return values and object state with respect to those observed when the original class is tested. Hence, such assertions fail when evaluated on the mutant if a different return value or object state is produced. An effective fitness function f for strong mutation coverage can be defined based on the notions of *infection* and *propagation*. The execution state of a mutated class instance is regarded as *infected* if it differs from the execution state of the original class when compared immediately after the mutated statement. This ensures that the mutant is producing some immediate effect on the class attributes or method variables, but it does not ensure that such an effect will propagate to an externally observable difference. Infection *propagation* accounts for the possibility of observing a different state between mutant and original class in the statements that follow the mutated one along the execution trace. The corresponding formal definition is the following [19]:

***Problem* 2.4:** *Let $M = \{m_1, \ldots, m_k\}$ be the set of mutants for a class. Find a test suite $T = \{t_1, \ldots, t_n\}$ that kills all mutants, i.e., one that minimises the following fitness function:*

$$\min f_M(T) = f_{B_M}(T) + \sum_{m \in M} (d_i(m, T) + d_p(m, T)) \quad (5)$$

where $f_{B_M}(T)$ is the whole-suite fitness function for all branches in T holding a direct control dependency on a mutant in M; d_i estimates the distance toward a state infection; and d_p denotes the propagation distance.

From an optimisation point of view, in the *whole-suite* approach the fitness function f considers all the targets at the same time and aggregates all corresponding distances into a unique fitness function by summing up the contributions from the individual target distances, i.e., the minimum distance from each individual target. In other words, multiple search targets are conflated into a single search target by means of an aggregated fitness function. Using this kind of approach, often named *scalarization* [11], a problem which involves multiple targets is transformed into a traditional single-objective, scalar one, thus allowing for the application of single-objective meta-heuristics such as standard GA.

2.2 Many-Objective Formulation

In this paper, we reformulate the coverage criteria for test case generation as many-objective optimisation problems, where the objectives to be optimised are the individual distances from all the test targets in the class under test. More formally, in this paper we consider the following reformulation:

***Problem* 2.5:** *Let $U = \{u_1, \ldots, u_k\}$ be the set of test targets to cover. Find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimise the fitness functions for all test targets $u_1, \ldots, u_k$, i.e., minimising the following k objectives:*

$$\begin{cases} \min f_1(t) = d(u_1, t) \\ \vdots \\ \min f_k(t) = d(u_k, t) \end{cases} \quad (6)$$

where each $d(u_i, t)$ denotes the distance of test case t from covering the test target u_i. Vector $\langle f_1, \ldots, f_k \rangle$ is also named a fitness vector.

According to this new generic reformulation, branch coverage, statement coverage and strong mutation coverage can be reformulated as follows:

***Problem* 2.6:** *Let $B = \{b_1, \ldots, b_k\}$ be the set of branches of a class. Find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimises the fitness functions for all branches $b_1, \ldots, b_k$, i.e., minimising the following k objectives:*

$$\begin{cases} \min f_1(t) = al(b_1, t) + d(b_1, t) \\ \vdots \\ \min f_k(t) = al(b_k, t) + d(b_k, t) \end{cases} \quad (7)$$

where each $d(b_j, t)$ denotes the normalised branch distance for the branch, executed by t, which is closest to b_j (i.e., which is at minimum number of control dependencies from b_j), while $al(b_j, t)$ is the corresponding approach level (i.e., the number of control dependencies between the closest executed branch and b_j).

***Problem* 2.7:** *Let $S = \{s_1, \ldots, s_k\}$ be the set of statements in a class. Find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimises the fitness functions for all statements $s_1, \ldots, s_k$, i.e., minimising the following k objectives:*

$$\begin{cases} \min f_1(t) = al(s_1, t) + d(b(s_1), t) \\ \vdots \\ \min f_k(t) = al(s_k, t) + d(b(s_k), t) \end{cases} \quad (8)$$

where each $d(b(s_j), t)$ denotes the normalised branch distance of test case t for the branch closest to $b(s_j)$, the branch that directly controls the execution of statement s_j, while $al(s_j, t)$ is the corresponding approach level.

***Problem* 2.8:** *Let $M = \{m_1, \ldots, m_k\}$ be the set of mutants for a class. Find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimises the fitness functions for all mutants $m_1, \ldots, m_k$, i.e., minimising the following k objectives:*

$$\begin{cases} \min f_1(t) = al(m_1, t) + d(b(m_1), t) + d_i(m_1, t) + \\ \qquad + d_p(m_1, t) \\ \vdots \\ \min f_k(t) = al(m_k, t) + d(b(m_k), t) + d_i(m_k, t) + \\ \qquad + d_p(m_k, t) \end{cases} \quad (9)$$

where $d(b(m_j), t)$ and $al(m_j, t)$ denote the normalised branch distance and the approach level of test case t for mutant m_j respectively; $d_i(m_j, t)$ is the distance from state infection, while $d_p(m_j, t)$ measures the propagation distance.

In this formulation, a candidate solution is a test case, not a test suite, which is scored by a single objective vector containing the *distances from all yet uncovered test targets*. Hence, the fitness is a vector of k values, instead of a single aggregate score. In many-objective optimisation, candidate solutions are evaluated in terms of *Pareto dominance* and *Pareto optimality* [11]:

***Definition* 1:** *A test case x dominates another test case y (also written $x \prec y$) if and only if the values of the objective function vector satisfy the following conditions:*

$$\forall i \in \{1, \ldots, k\} \;\; f_i(x) \leq f_i(y)$$
$$\text{and}$$
$$\exists j \in \{1, \ldots, k\} \text{ such that } f_j(x) < f_j(y)$$

Conceptually, the definition above indicates that x is preferred to (dominates) y if and only if x is strictly better on one or more objectives and it is not worse for the remaining objectives. For example, in branch coverage x is preferred to (dominates) y if and only if x has a lower branch distance + approach level for one or more branches and it is not worse for the remaining branches.

Figure 1 provides a graphical interpretation of Pareto dominance for a simple case with two test targets to cover, i.e., the problem is bi-objective. All test cases in the grey rectangle (A and C) are dominated by D, because D is better for both the two objectives f_1 and f_2. On the other hand, the test case D does not dominate B because it is closer to cover f_1 than B, but it is worse than B on the other test target, f_2. Moreover, D does not dominate E because it is worse for the test target f_1. Similarly, the test case B does not dominate D and E but it dominates A. Thus, B, D, and E are non-dominated by any other test case, while A and

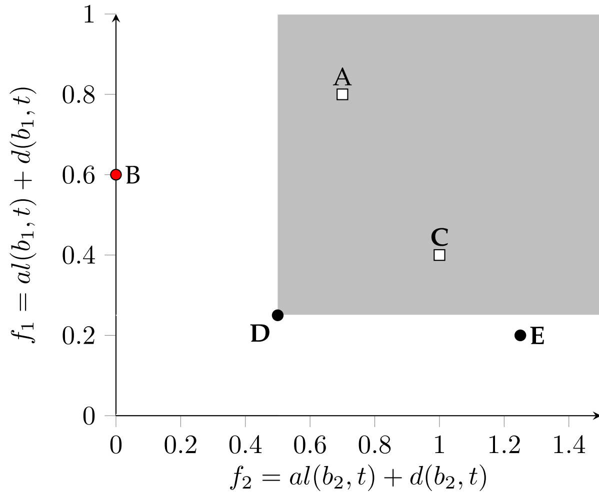


Fig. 1. Graphical interpretation of Pareto dominance.

C are dominated by either D or B. The test case B satisfies (covers) the target f_2, thus, it is a candidate to be included in the final test suite.

Among all possible test cases, the optimal test cases are those non-dominated by any other possible test case:

***Definition* 2:** *A test case x^* is Pareto optimal if and only if it is not dominated by any other test case in the space of all possible test cases (feasible region).*

Single-objective optimisation problems have typically one solution (or multiple solutions with the same optimal fitness value). On the other hand, solving a multi-objective problem may lead to a set of Pareto-optimal test cases (with different fitness vectors), which, when evaluated, correspond to trade-offs in the objective space. While in many-objective optimisation it may be useful to consider all the trade-offs in the objective space, especially if the number of objectives is small, in the context of coverage testing we are interested in finding only the test cases that contribute to maximising the total coverage by covering previously uncovered test targets, i.e., test cases having one or more objective scores equal to zero: $f_i(t) = 0$, as test B in Figure 1. These are the test cases that intersect any of the m Cartesian axes of the vector space where fitness vectors are defined. Such test cases are candidates for inclusion in the final test suite and represent a specific sub-set of the Pareto optimal solutions.

3 Algorithm

Search algorithms aimed at optimising more than three objectives have been widely investigated for classical numerical optimisation problems. A comprehensive survey of such algorithms can be found in a recent paper by Li *et al.* [34]. The following subsections provide a discussion of the most relevant related works on many-objective optimisation and highlight how our novel many-objective algorithm overcomes their limitations in the context of *many-objective structural coverage*. In particular, Section 3.1 provides an overview on traditional many-objective algorithms, while Section 3.2 presents the many-objective algorithms that we have developed for solving the many-objective reformulation of structural coverage criteria.

3.1 Existing Many-Objective Algorithms

Multi-objective algorithms, such as the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [13] and the improved Strength Pareto Evolutionary Algorithm (SPEA2) [63], have been successfully applied within the software engineering community to solve problems with two or three objectives, including software refactoring, test case prioritisation, etc. However, such classical multi-objective evolutionary algorithms present scalability issues because they are not effective in solving optimisation problems with more than three-objectives [34].

To overcome their limitations, a number of many-objective algorithms have been recently proposed in the evolutionary computation community, which modify multi-objective solvers to increase the selection pressure. According to Li *et al.* [34], existing many-objective solvers can be categorised into seven classes: *relaxed dominance based*, *diversity-based*, *aggregation-based*, *indicator-based*, *reference set based*, *preference-based*, and *dimensionality reduction* approaches. For example, Laumanns *e*t al. [33] proposed the usage of ϵ-dominance (ϵ-MOEA), which is a relaxed version of the classical dominance relation that enlarges the region of the search space dominated by each solution, so as to increase the likelihood for some solutions to be dominated by others [34]. Although this approach is helpful in obtaining a good approximation of an exponentially large Pareto front in polynomial time, it presents drawbacks and in some cases it can slow down the optimisation process significantly [27].

Yang *e*t al. [58] introduced a Grid-based Evolutionary Algorithm (GrEA) that divides the search space into hyperboxes of a given size and uses the concepts of grid dominance and grid distance to improve diversity among individuals by determining their mutual relationship in a grid environment. Zitzler and Künzli [62] proposed the usage of the hypervolume indicator instead of Pareto dominance when selecting the best solutions to form the next generation. Even if the new algorithm, named IBEA (Indicator Based Evolutionary Algorithm), was able to outperform NSGA-II and SPEA2, the computation cost associated with the exact calculation of the hypervolume indicator in a high-dimensional space (i.e., with more than five objectives) is too expensive, making it unfeasible with hundreds of objectives as in the case of structural test coverage (e.g., in mutation coverage a class/program may have hundreds of mutants).

Zhang and Li [61] proposed a Decomposition based Multi-objective Evolutionary Algorithm (MOEA/D), which decomposes a single many-objective problem into many single-objective sub-problems by employing a series of weighting vectors. Specifically, each sub-problem is obtained by using a specific weighting vector that aggregates different objectives using a weighted-sum approach. Different weighting vectors assign different importance (weights) to objectives, thus, delimiting the searching direction of these aggregation-based algorithms. However, the selection of weighting vectors is still an open problem especially for problems with a very large number of objectives [34].

Di Pierro *e*t al. [14] used a preference order-approach (POGA) as an optimality criterion in the ranking phase of NSGA-II. This criterion considers the concept of efficiency of order in subsets of objectives and provides a higher selection pressure towards the Pareto front than Pareto dominance-based algorithms.

Deb and Jain [12] proposed NSGA-III, an improved version of the classical NSGA-II algorithm, where the crowding distance is replaced with a reference-set based niche-preservation strategy. It results in a new Pareto ranking/sorting based algorithm that produces well-spread and diversified solutions.

Yuan *e*t al. [59] developed θ-NSGA-III, which enriches the non-dominated sorting scheme with the concepts of θ-dominance to rank solutions in the environmental selection phase, so as to ensure both convergence and diversity.

All the many-objective algorithms mentioned above have been investigated mostly for numerical optimisation problems with less than 15 objectives. Moreover, they are designed to produce a rich set of optimal trade-offs between different optimisation goals, by considering both proximity to the real Pareto optimal set and diversity between the obtained trade-offs [33]. As explained in Section 2.2, this is not the case of structural coverage for test case generation. The interesting solutions for this problem are test cases having one or more objective scores equal to zero (i.e., $f_i(t) = 0$). Trade-offs between objectives scores are useful just for maintaining diversity during the optimisation process. Hence, there are two main peculiarities that have to be considered in many-objective structural coverage as compared to more traditional many-objective problems. First, not all non-dominated test cases have a practical utility since they represent trade-offs between objectives. Instead, in structural coverage the search has to focus on a specific sub-set of the non-dominated solutions: those solutions (test cases) that satisfy one or more coverage targets. Second, for a given level of structural coverage, shorter test cases (i.e., test cases with a lower number of statements) are preferred to reduce the oracle cost [6], [17] and to avoid the *bloat* phenomenon [17].

3.2 The Proposed Many-Objective Algorithms

Previous research in many-objective optimisation [55], [34] has shown that many-objective problems are particularly challenging because of the *dominance resistance* phenomenon, i.e., most of the solutions are incomparable since the proportion of non-dominated solutions increases exponentially with the number of objectives. As a consequence, it is not possible to assign a preference among individuals for selection purposes and the search process becomes equivalent to a random one [55]. Thus, problem/domain specific knowledge is needed to impose an order of preference over test cases that are non-dominated according to the traditional non-dominance relation. For test case generation, this means focusing the search effort on those test cases that are closer to one or more uncovered targets (e.g., branches) of the program. To this aim, we propose the following *preference criterion* in order to impose an order of preference among non-dominated test cases:

Algorithm 1: NSGA-II

```
   Input:
   U = {u_1, ..., u_m} the set of coverage targets of a program.
   Population size M
   Result: A test suite T
 1 begin
 2     t ⟵ 0 // current generation
 3     P_t ⟵ RANDOM-POPULATION(M)
 4     while not (search_budget_consumed) do
 5         Q_t ⟵ GENERATE-OFFSPRING(P_t)
 6         R_t ⟵ P_t ⋃ Q_t
 7         𝔽 ⟵ FAST-NONDOMINATED-SORT(R_t)
 8         P_{t+1} ⟵ ∅
 9         d ⟵ 1
10         while | P_{t+1} | + | 𝔽_d | ⩽ M do
11             CROWDING-DISTANCE-ASSIGNMENT(𝔽_d)
12             P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d
13             d ⟵ d + 1
14         Sort(𝔽_d) //according to the crowding distance
15         P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d[1 : (M− | P_{t+1} |)]
16         t ⟵ t + 1
17     S ⟵ P_t
```

***Definition* 3:** *Given a coverage target u_i, a test case x is preferred over another test case y (also written $x \prec_{u_i} y$) if and only if the values of the objective function for u_i satisfy the following condition:*

$$f_i(x) < f_i(y) \quad OR \quad f_i(x) = f_i(y) \wedge size(x) < size(y)$$

where $f_i(x)$ denotes the objective score of test case x for coverage target u_i (see Section 2), and $size$ measures the test case length (number of statements). The *best test case* for a given coverage target u_i is the one preferred over all the others for such target ($x_{\text{best}} \prec_{u_i} y, \ \forall y \in T$). The *set of best test cases* across all uncovered targets ($\{x \mid \exists\ i : x \prec_{u_i} y, \ \forall y \in T\}$) defines a subset of the Pareto front that is given priority over the other non-dominated test cases in our algorithms. When there are multiple test cases with the same minimum fitness value for a given coverage target u_i, we use the test case length (number of statements) as a secondary preference criterion. We opted for this secondary criterion since generated tests require human effort to check the candidate assertions (oracle problems). Therefore, minimising the test cases is a valuable (secondary) objective to achieve [6], [17] since smaller tests involve a lower number of method calls (and corresponding covered paths) to manually analyse.

Our preference criterion provides a way to distinguish between test cases in a set of non-dominated ones, i.e., in a set where test cases are incomparable according to the traditional non-dominance relation, and it increases the selection pressure by giving higher priority to the best test cases with respect to the currently uncovered targets. Since none of the existing many-objective algorithms considers this preference ordering, which is a peculiarity of test case generation, in this paper we define our novel many-objective GA by incorporating the proposed *preference criterion* in the main loop of NSGA-II, a widely known Pareto efficient multi-objective genetic algorithm designed by Deb *et al.* [13].

In a nutshell, NSGA-II starts with an initial set of random solutions (random test cases), also called *chromosomes*, which represents a random sample of the search space (line 3 of Algorithm 1). The population then evolves through a series of *generations* to find better test cases. To produce the next generation, NSGA-II first creates new test cases,

called *offsprings*, by combining parts from two selected test cases (*parents*) in the current generation using the *crossover* operator and randomly modifying test cases using the *mutation* operator (function GENERATE-OFFSPRING, at line 5 of Algorithm 1). Parents are selected according to a *selection* operator, which uses Pareto optimality to give higher chance of reproduction to non-dominated (fittest) test cases in the current population. The *crowding distance* is used in order to make a decision about which test cases to select: non-dominated test cases that are far away from the rest of the population have higher probability of being selected (lines 10-15 of Algorithm 1). Furthermore, NSGA-II uses the *FAST-NONDOMINATED-SORT* algorithm to preserve the test cases forming the current Pareto frontier in the next generation (*elitism*). After some generations, NSGA-II converges to "stable" test cases, which represent the Pareto-optimal solutions to the problem.

The next sections describe in detail our proposed many-objective algorithms starting from NSGA-II, first describing the MOSA algorithm followed by its extension, DynaMOSA.

3.2.1 MOSA: a Many Objective Sorting Algorithm

MOSA (Many Objective Sorting Algorithm) replaces the traditional *non-dominated sorting* with a new ranking algorithm based on our *preference criterion*. As shown in Algorithm 2, MOSA shares the initial steps of NSGA-II: it starts with an initial set of randomly generated test cases that forms the initial *population* (line 3 of Algorithm 2); then, it creates new test cases using *crossover* and *mutation* (function GENERATE-OFFSPRING, at line 6 of Algorithm 2).

Algorithm 2: MOSA

```
Input:
U = {u_1, ..., u_m} the set of coverage targets of a program.
Population size M
Result: A test suite T
1  begin
2    t ⟵ 0 // current generation
3    P_t ⟵ RANDOM-POPULATION(M)
4    archive ⟵ UPDATE-ARCHIVE(P_t, ∅)
5    while not (search_budget_consumed) do
6      Q_t ⟵ GENERATE-OFFSPRING(P_t)
7      archive ⟵ UPDATE-ARCHIVE(Q_t, archive)
8      R_t ⟵ P_t ⋃ Q_t
9      𝔽 ⟵ PREFERENCE-SORTING(R_t)
10     P_{t+1} ⟵ ∅
11     d ⟵ 0
12     while | P_{t+1} | + | 𝔽_d | ⩽ M do
13       CROWDING-DISTANCE-ASSIGNMENT(𝔽_d)
14       P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d
15       d ⟵ d + 1
16     Sort(𝔽_d) //according to the crowding distance
17     P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d[1 : (M− | P_{t+1} |)]
18     t ⟵ t + 1
19   T ⟵ archive
```

Algorithm 3: PREFERENCE-SORTING

```
Input:
A set of candidate test cases T
Population size M
Result: Non-dominated ranking assignment 𝔽
1  begin
2    𝔽_0 // first non-dominated front
3    for u_i ∈ U and u_i is uncovered do
4      // for each uncovered target we select the best test case
         according to the preference criterion
5      t_best ⟵ test case in T with minimum objective score for u_i
6      𝔽_0 ⟵ 𝔽_0 ⋃ {t_best}
7    T ⟵ T − 𝔽_0
8    if |𝔽_0| > M then
9      𝔽_1 ⟵ T
10   else
11     U* ⟵ {g ∈ U : u is uncovered}
12     𝔼 ⟵ FAST-NONDOMINATED-SORT(T, {u ∈ U*})
13     d ⟵ 0 //first front in 𝔼
14     for All non-dominated fronts in 𝔼 do
15       𝔽_{d+1} ⟵ 𝔼_d
```

Selection. Differently from NSGA-II, in MOSA the selection is performed by considering both the non-dominance relation and the proposed *preference criterion* (function PREFERENCE-SORTING, at line 9 of Algorithm 2). In particular, the PREFERENCE-SORTING function, whose pseudo-code is provided in Algorithm 3, determines the test case with the lowest objective score (e.g., branch distance + approach level for branch coverage) for each uncovered target $u_i \in U$, i.e., the test case that is closest to cover u_i (lines 2-6 of Algorithm 3). All these test cases are assigned rank 0 (i.e., they are inserted into the first non-dominated front $\mathbb{F}_0$), so as to give them a higher chance of surviving in to the next generation (*elitism*). The remaining test cases (those not assigned to the first rank) are then ranked according to the traditional non-dominated sorting algorithm used by NSGA-II [13], starting with a rank equal to 1 and so on (line 11-15 of Algorithm 3). To speed-up the ranking procedure, the traditional non-dominated sorting algorithm is applied only when the number of test cases in $\mathbb{F}_0$ is smaller than the population size M (condition in line 8). Instead, when the condition $|\mathbb{F}_0| > M$ is satisfied, the PREFERENCE-SORTING function returns only two fronts: the first front (rank 0) with all test cases selected by our *preference criterion*; and the front with rank 1 that contains all remaining test cases in T, i.e., $\mathbb{F}_1 = T - \mathbb{F}_0$.

Dominance. It is important to notice that the routine FAST-NONDOMINATED-SORT assigns the ranks to the remaining test cases by considering only the non-dominance relation for the *uncovered* targets, i.e., by focusing the search toward the interesting sub-region of the remaining search space. Such non-dominance relation is computed by a specific *dominance comparator* which iterates over all the *uncovered* targets when deciding whether two test cases t_1 and t_2 do not dominate each other or whether one (e.g., t_1) dominates the other (e.g., t_2). While the original *dominance comparator* defined by Deb *et al.* [13] iterates all over the objectives (targets), in MOSA we use the *dominance comparator* depicted in Algorithm 4. Such a comparator iterates only over the *uncovered* targets (loop condition in line 4 of Algorithm 4). Moreover, as soon as it finds two uncovered targets for which two test cases t_1 and t_2 do not dominate each other (lines 11-12 Algorithm 4), the iteration is terminated without analysing further uncovered targets.

Crowding distance. Once a rank is assigned to all candidate test cases, the *crowding distance* is used to give higher probability of being selected to some test cases in the same front. As measure for the crowding distance, we use the *sub-vector dominance assignment* proposed by Köppen et al. [31] for many-objective optimisation problems. Specifically, the loop at line 12 in Algorithm 2 and the following lines 16 and 17 add as many test cases as possible to the next generation, according to their assigned ranks, until reaching

Algorithm 4: DOMINANCE-COMPARATOR

```
Input:
Two test cases to compare t_1 and t_2
U = {u_1, ..., u_m} the set of coverage targets of a program.
1  begin
2      dominates1 ⟵ false
3      dominates2 ⟵ false
4      for u_i ∈ U and u_i is uncovered do
5          f_i^1 ⟵ values of u_i for t_1
6          f_i^2 ⟵ values of u_i for t_2
7          if f_i^1 < f_i^2 then
8              dominates1 ⟵ true
9          if f_i^2 < f_i^1 then
10             dominates2 ⟵ true
11         if dominates1 == true & dominates2 == true then
12             break
13     if dominates1 == dominates2 then
14         // t_1 and t_2 do not dominate each other
15     else
16         if dominates1 == true then
17             // t_1 dominates t_2
18         else
19             // t_2 dominates t_1
```

Algorithm 5: UPDATE-ARCHIVE

```
Input:
A set of candidate test cases T
An archive A
Result: An updated archive A
1  begin
2      for u_i ∈ U do
3          t_best ⟵ ∅
4          best_length ⟵ ∞
5          if u_i already covered then
6              t_best ⟵ test case in A covering u_i
7              best_length ⟵ number of statements in t_best
8          for t_j ∈ T do
9              score ⟵ objective score of t_j for target u_i
10             length ⟵ number of statements in t_j
11             if score == 0 and length ≤ best_length then
12                 replace t_best with t_j in A
13                 t_best ⟵ t_j
14                 best_length ⟵ length
15     return A
```

the population size. The algorithm first selects the non-dominated test cases from the first front ($\mathbb{F}_0$); if the number of selected test cases is lower than the population size M, the loop selects more test cases from the second front ($\mathbb{F}_1$), and so on. The loop stops when adding test cases from current front $\mathbb{F}_d$ exceeds the population size M. At end of the loop (lines 16-17), when the number of selected test cases is lower than the population size M, the algorithm selects the remaining test cases from the current front $\mathbb{F}_d$ according to the descending order of crowding distance.

Archiving. As a further peculiarity with respect to other many-objective algorithms, MOSA uses a second population, called *archive*, to keep track of the best test cases that cover targets of the program under test. Specifically, whenever new test cases are generated (either at the beginning of search or when creating offsprings) MOSA stores those tests that satisfy previously uncovered targets in the *archive* as candidate test cases to form the final test suite (line 4 and 7 of Algorithm 2). To this aim, function UPDATE-ARCHIVE (reported in Algorithm 5 for completeness) updates the set of test cases stored in the *archive* with the new generated test cases. This function considers both the covered targets and the length of test cases when updating the *archive*: for each covered target u_i it stores the shortest test case covering u_i in the *archive*.

In summary, generation by generation MOSA focuses the search towards the uncovered targets of the program under test (both PREFERENCE-SORTING and FAST-NONDOMINATED-SORT routines analyse the objective scores of the candidate test cases considering the uncovered targets only); it also stores the shortest covering test cases in an external data structure (i.e., the *archive*) to form the final test suite. Finally, since MOSA uses the *crowding distance* when selecting the test cases, it promotes *diversity*, which represents a key factor to avoid premature convergence toward sub-optimal regions of the search space [28].

3.2.2 Graphical Interpretation

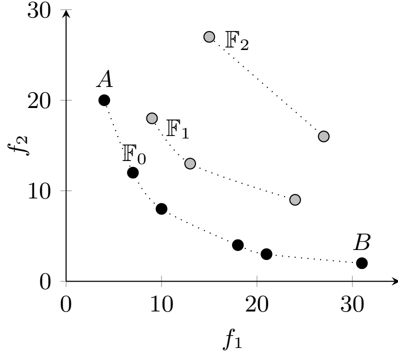
Let us consider the simple program shown in Figure 2-a. Let us assume that the coverage criterion is branch coverage and that the uncovered targets are the true branches of statements 1 and 3, whose branch predicates are `(a == b)` and `(b == c)` respectively. According to the proposed many-objective formulation, the corresponding problem has two residual optimisation objectives, which are $f_1 = al(b_1) + d(b_1) = abs(a - b)$ and $f_2 = al(b_2) + d(b_2) = abs(b - c)$. Hence, any test case produced at a given generation g corresponds to some point in a two-dimensional objective space as shown in Figure 2-b and 2-c. Unless both a and b are equal, the objective function f_1 computed using the combination of approach level and branch distance is greater than zero. Similarly, function f_2 is greater than zero unless b and c are equal.

Let us consider the scenario reported in Figure 2-b where no test case is able to cover the two uncovered branches (i.e., in all cases $f_1 > 0$ and $f_2 > 0$). If we use the traditional non-dominance relation between test cases, all test cases corresponding to the black points in Figure 2-b are non-dominated and form the first non-dominated front $\mathbb{F}_0$. Therefore, all such test cases have the same probability of being selected to form the next generation, even if test case A is the closest to the Cartesian axis f_2 (i.e., closest to cover branch b_2) and test case B is the closest to the Cartesian axis f_1 (branch b_1). Since there is no preference among test cases in $\mathbb{F}_0$, it might happen that A and/or B are not kept for the next generation, while other, less useful test cases in $\mathbb{F}_0$ are preserved. This scenario is quite common in many-objective optimisation, where the number of non-dominated solutions increases exponentially with the number of objectives [55]. However, from the branch coverage point of view the two test cases A and B are better (fitter) than all other test cases, because they are the closest to covering each of the two uncovered branches.

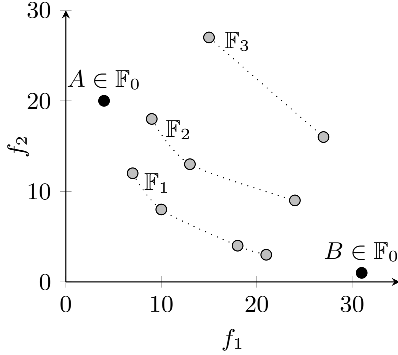
Our novel *preference criterion* gives a higher priority to test cases A and B with respect to all other test cases, guaranteeing their survival in the next generation. In particular, using the new ranking algorithm proposed in this paper (see Algorithm 3), the first non-dominated front $\mathbb{F}_0$ will contain only test cases A and B (see Figure 2-c), while all other test cases will be assigned to other, successive fronts. When there are multiple test cases that are closest to one axis,

	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;`
	`}`

(a) Example program



(b) Ranking based on the traditional non-dominance relation

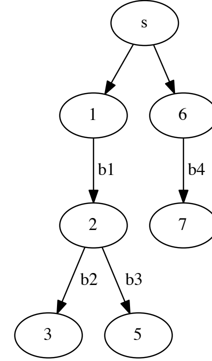


(c) Ranking based on the proposed *preference criterion*

Fig. 2. Graphical comparison between the non-dominated ranks assignment obtained by the traditional *non-dominated sorting algorithm* and the ranking algorithm based on the *preference criterion* proposed in this paper.

	Instructions
s	`int example(int a, int b, int c)`
	`{`
	`int x = 0;`
1	`if (a == b)    //` b_1
2	`   if (a > c) //` b_2
3	`      x = 1;`
4	`   else       //` b_3
5	`      x = 2;`
6	`if (b == c)    //` b_4
7	`   x = -1;`
8	`return x;`
	`}`

(a) Example program



(b) Control dependency graph

Fig. 3. Since b_1 holds a control dependency on b_2, b_3, targets b_2, b_3 are dynamically selected only once b_1 is covered.

associated with a given uncovered branch, the *preference criterion* provides a further strategy to impose an order to the non-dominated test cases by assigning rank 0 to the shortest test case only.

3.2.3 DynaMOSA: Dynamic Selection of the Optimisation Targets

One main limitation in MOSA is that it considers all coverage targets as independent objectives to optimise since the first generation. However, there exist structural dependencies among targets that should be considered when deciding which targets/objectives to optimise. For example, some targets can be satisfied if and only if other related targets are already covered.

To better explain this concept, let us consider the example in Figure 3. The four branches to be covered, b_1, b_2, b_3, b_4, are not independent from each other. In fact, coverage of b_2, b_3 can be achieved only after b_1 has been covered, since the former targets are under the control of the latter. In other words, there is a control dependency between b_1 and b_2, b_3, which means that the execution of b_2, b_3 depends on the outcome of the condition at node 2, which in turn is evaluated only once target b_1 is covered. If no test case covers b_1, the ranking in MOSA is determined by the fitness function $f_1 = d(b_1)$. When tests are evaluated for the two dependent branches b_2 and b_3, the respective fitness functions will be equal to $f_1 + 1$, since the only difference from coverage of b_1 consists of a higher approach level (in this case, +1), while the branch distance d is the same. Since the values of f_2 and f_3 are just shifted by a constant amount (the approach level) with respect to f_1, the test case ranking is the same as the one obtained when considering f_1 alone. This means that taking into account objectives f_2, f_3 during preference sorting is useless, since they do not contribute to the final ranking.

The example illustrated above shows that coverage targets can be organised into a priority hierarchy based on the following concepts in standard flow analysis:

***Definition* 4** (Dominator)**:** *A statement s_1 dominates another statement s_2 if every execution path to s_2 passes through s_1.*

***Definition* 5** (Post-dominator)**:** *A statement s_1 post-dominates another statement s_2 if every execution path from s_2 to the exit point (`return` statement) passes through s_1.*

***Definition* 6** (Control dependency)**:** *There is a control dependency between program statement s_1 and program statement s_2*

iff: (1) s_2 is not a postdominator of s_1 and (2) there exists a path in the control flow graph between s_1 and s_2 whose nodes are postdominated by s_2.

Condition (1) in the definition of control dependency ensures that s_2 is not necessarily executed after s_1. Condition (2) ensures that once a specific path is taken between s_1 and s_2, the execution of s_2 becomes inevitable. Hence, s_1 is a decision point whose outcome determines the inevitable execution of s_2.

***Definition* 7** (Control dependency graph)**:** *The graph $G = \langle N, E, s \rangle$, consisting of nodes $n \in N$ that represent program statements and edges $e \in E \subseteq N \times N$ that represent control dependencies between program statements, is called control dependency graph. Node $s \in N$ represents the entry node, which is connected to all nodes that are not under the control dependency of another node.*

The definition of control dependency given above can be easily extended from program statements to arbitrary coverage targets. For instance, given two branches to be covered, b_1, b_2, we say b_1 holds a control dependency on b_2 if b_1 is postdominated by a statement s_1 which holds a control dependency on a node s_2 that postdominates b_2.

In DynaMOSA, the control dependency graph is used to derive which targets are independent from any others (targets that are free of control dependencies) and which ones can be covered only after satisfying previous targets in the graph. The difference between DynaMOSA and MOSA are illustrated in Algorithms 6. At the beginning the search process, DynaMOSA selects as initial set of objectives only those targets that are free of control dependencies (line 2). Once the initial population of random test cases is generated (line 4), the current set of targets U^* is update using the routine UPDATE-TARGET highlighted in Algorithm 7. This routine is also called at each iteration in order to update the current set of targets U^* to consider at each generation depending on the results of the execution of the offspring (line 10 in Algorithms 6). Therefore, fitness evaluation (line 8), preference sorting (line 12), and crowding distance (line 16) are computed only considering the targets in $U^* \subseteq U$.

The routine UPDATE-TARGET is responsible for dynamically updating the set of selected targets U^* so as to include any uncovered targets that are control dependent on the newly covered target. It iterates over all targets in $U^* \subseteq U$ (loop in lines 2-6 of Algorithm 7) and in case of newly covered targets (condition in line 3) it visits the control flow graph to find all control dependent targets. Indeed, Algorithm 7 uses a standard graph visit algorithm, which stops its visit whenever an uncovered target is encountered (lines 7-13). In such a case, the encountered target is added to the set of targets U^*, to be considered by DynaMOSA in the next generation. If the encountered target is covered, the visit continues recursively until the first uncovered target is found or all targets have been already visited (lines 9-11). This ensures that only the first uncovered target, encountered after visiting a covered target, is added to U^*. All following uncovered targets are just ignored, since the graph visit stops[2].

2. For simplicity, we only consider the case of targets associated with graph edges, but the same algorithm works if targets are associated with nodes

Algorithm 6: DynaMOSA

Input:
$U = \{u_1, \ldots, u_m\}$ the set of coverage targets of a program.
Population size M
$G = \langle N, E, s \rangle$: control dependency graph of the program
$\phi : E \rightarrow U$: partial map between edges and targets
Result: A test suite T

```
1  begin
2      U* ⟵ targets in U with not control dependencies
3      t ⟵ 0 // current generation
4      P_t ⟵ RANDOM-POPULATION(M)
5      archive ⟵ UPDATE-ARCHIVE(P_t, ∅)
6      U* ⟵UPDATE-TARGETS(U*, G, φ)
7      while not (search_budget_consumed) do
8          Q_t ⟵ GENERATE-OFFSPRING(P_t)
9          archive ⟵ UPDATE-ARCHIVE(Q_t, archive)
10         U* ⟵UPDATE-TARGETS(U*, G, φ)
11         R_t ⟵ P_t ⋃ Q_t
12         𝔽 ⟵ PREFERENCE-SORTING(R_t, U*)
13         P_{t+1} ⟵ ∅
14         d ⟵ 0
15         while | P_{t+1} | + | 𝔽_d | ⩽ M do
16             CROWDING-DISTANCE-ASSIGNMENT(𝔽_d, U*)
17             P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d
18             d ⟵ d + 1
19         Sort(𝔽_d) //according to the crowding distance
20         P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d[1 : (M− | P_{t+1} |)]
21         t ⟵ t + 1
22     T ⟵ archive
```

Algorithm 7: UPDATE-TARGETS

Input:
$G = \langle N, E, s \rangle$: control dependency graph of the program
$U^* \subseteq U$: current set of targets
$\phi : E \rightarrow U$: partial map between edges and targets
Result:
U^*: updated set of targets to optimise

```
1  begin
2      for u ∈ U* do
3          if u is covered then
4              U* ⟵ U* − {u}
5              e_u ⟵ edge in G for the target u
6              visit(e_u)

7  Function visit(e_u ∈ E)
8      for each unvisited e_n ∈ E control dependent on e_u do
9          if φ(e_n) is not covered then
10             U* ⟵ U* ∪ {φ(e_n)}
11             set e_n as visited
12         else
13             visit(e_n)
```

In this way, the ranking performed by MOSA remains unchanged (a formal proof of this is provided below), while convergence to the final test suite is achieved faster, since the number of objectives to be optimised 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.

There are two main conditions that must be satisfied to justify the dynamic selection of the targets produced by the execution of Algorithm 6:

1) Since at each generation, the set U^* used by DynaMOSA contains all targets with minimum approach level, for each remaining target u_i in $U - U^*$ there should exist a target $u^* \in U^*$ such that $f(u_i) = f(u^*) + K$.

2) The computation cost of the routine UPDATE-TARGETS in Algorithm 7 should be negligible if compared to cost of computing preference sorting and crowding distance for the all uncovered targets.

The first condition is ensured by Theorem 1, presented in the following. The second is not ensured theoretically, but it was found empirically to hold in practice.

DynaMOSA generates test cases with minimum distance from U^*, the set of uncovered targets directly reachable through a control dependency edge of one or more test case execution trace. On the other hand, MOSA generates test cases with minimum distance from the full set of uncovered targets, U. The two minimisation processes are equivalent if and only if the test cases at minimum distance from the elements of U^* are also the test cases at minimum distance from U. The following Theorem provides a theoretical justification for the dynamic selection of a subset of the targets produced by the execution of Algorithm 7, by proving that the two sets of test cases optimised respectively by MOSA (T_{min}) and DynaMOSA (T_0) are the same.

***Theorem* 1:** *Let U be the set of all uncovered targets; T_{min} the set of test cases with minimum approach level from U and T_0 the test cases with approach level zero; U^* be the set of uncovered targets at approach level zero from one or more test cases $t \in T_0$, then the two sets of test cases T_{min} and T_0 are the same, i.e., $T_{min} = T_0$.*

Proof: *Since the approach level is always greater than or equal to zero, the approach level of the elements of T_0 is ensured to be minimum. Hence, $T_0 \subseteq T_{min}$. Let us now prove by contradiction that there cannot exist any uncovered target u associated with a test case t whose approach level from u is strictly greater than zero, such that t does not belong to T_0. Let us consider the shortest path from the execution trace of t to u in the control dependency graph. The first node u' in such path belongs to U^*, since it is an uncovered target and it has approach level zero from the trace of t. Indeed, it is an uncovered target because otherwise the test case t' that covers it would have an execution trace closer to u than t. Since it satisfies the relation approach_level(t, u') = 0, test case t must be an element of T_0. Hence, T_{min} cannot contain any test case not in T_0 and the equality $T_{min} = T_0$ must hold.* □

Since the same set of test cases are at minimum distance from either U or U^*, the test case selection performed by MOSA, based on the distance of test cases from U, is algorithmically equivalent to the test case selection performed by DynaMOSA, based on the distance of test cases from U^*.

The fact that DynaMOSA iterates over a lower number of targets ($U^* \subseteq U$) with respect to MOSA has direct consequences on the computational cost of each single generation. In fact, in each generation the two many-objective algorithms share two routines: the PREFERENCE-SORTING and the CROWDING-DISTANCE-ASSIGNMENT routines, as shown in Algorithms 2 and 6. For the fronts following the first one, the routine PREFERENCE-SORTING relies on the traditional FAST-NON-DOMINATED-SORT by Deb et al. [13] , which has an overall computation complexity of $O(M^2 \times N)$ where M is the size of the population and N is the number of objectives. Thus, for MOSA the cost of such routine is $O(M^2 \times |U|)$ where U is the set of uncovered targets in a given generation. For DynaMOSA, the computational cost is reduced to $O(M^2 \times |U^*|)$ with $U^* \subseteq U$ being the set of uncovered targets with minimum approach level. For trivial classes where all targets have no structural dependencies (i.e., classes with only branchless methods), the cost of PREFERENCE-SORTING will be the same since $U^* = U$. However, for non-trivial classes we would expect $U^* \subset U$ leading to a reduced computational complexity.

A similar analysis can be performed for the other shared routine, i.e., CROWDING-DISTANCE-ASSIGNMENT. According to Köppen et al. [31], the cost of such a routine when using the *sub-vector dominance assignment* is $O(M^2 \times N)$ where M is the size of the population while N is the number of objectives. Therefore, for MOSA the cost of such routine is $O(M^2 \times |U|)$ where U is the set of uncovered targets in a given generation; while for DynaMOSA it is $O(M^2 \times |U^*|)$. For non-trivial classes, DynaMOSA will iterate over a lower number of targets as long as the condition $U^* \subset U$ holds.

Moreover, MOSA requires to compute the fitness scores (e.g., approach level and branch distances in branch coverage) for all uncovered targets U and for each newly generated test case. In particular, for each test case MOSA requires to compute the distances between execution traces and all targets in U. Instead, DynaMOSA focuses only on coverage targets with minimum approach levels, thus, requiring to compute the fitness scores for $U^* \subseteq U$ only.

The additional operations of DynaMOSA with respect to MOSA are the construction of the intra-procedural control dependency graph and its visit for target selection. The cost for intra-procedural control dependency graph construction is paid just once for all test case generation iterations and the computation of the graph can be carried out offline, before running the algorithm. The visit of the control dependency graph for coverage target selection can be performed very efficiently (for structured programs the graph is a tree). As a consequence, the saving in the computation of PREFERENCE-SORTING, CROWDING-DISTANCE-ASSIGNMENT and fitness scores performed by DynaMOSA vs. MOSA is expected to dominate the overall execution time of the two algorithms, while the visit of the control dependency graph performed by DynaMOSA is expected to introduce a negligible overhead. This was confirmed by our experiments. In fact, in our experiments the intra-procedural control dependency graphs of the methods under test and their visit consumed a negligible amount of execution time in comparison with the time taken by the extra computations performed by MOSA.

4 Empirical Evaluation

This section details the empirical study we carried out to evaluate the proposed many-objective test case generation algorithms with respect to three testing criteria: (i) *branch* coverage, (ii) *statement* coverage, and (iii) *strong-mutation* coverage.

4.1 Subjects

The *context* of our study is a random selection of classes from four test benchmarks: (i) the SF110 corpus [18]; (ii) the SBST tool contest 2015 [53]; (iii) the SBST tool contest 2016 [49]; and (iv) the benchmark used in our previous conference paper [42].

The SF110 benchmark[3] is a corpus of Java classes composed of a representative sample of 100 projects from the SourceForge.net repository, augmented with 10 of the most popular projects in the repository, for a total of 23,886 classes. We selected this benchmark since it has been widely used in the literature to assess test case generation tools [18], [51], [48]. However, Shamshiri et al. [51] showed that the vast majority of classes in the FS110 corpus are trivial to cover fully, even with random search algorithms. In fact, they observed that most of the classes in this corpus do not require the usage of evolutionary algorithms. Hence, for our evaluation we selected only non-trivial classes from this benchmark. To this aim, we first computed the McCabe's cyclomatic complexity [36] for each method in FS110 by means of the extended `CKJM` library[4]. The McCabe's cyclomatic complexity of a method is defined as the number of independent paths in the control flow graph and can be shown (for structured programs) to be equal to the number of branches plus one. We found that 56% of the classes in SF110 are trivial, containing only methods with cyclomatic complexity equal to 1. These are branchless methods that can be fully covered by a simple method call. Therefore, we pruned this benchmark by removing all classes whose methods have a cyclomatic complexity lower than five[5]. This filtering resulted in a smaller benchmark composed of 96 projects and 23% of classes from the original SF110 corpus. Among those 96 projects, 18 projects contain only a single non-trivial class after our filter. For our experiment, we randomly sampled 2 classes from each of the 78 projects containing multiple non-trivial classes, and we selected the available non-trivial classes from each of the remaining 18 projects. In addition, we randomly sampled two further classes for each of the most popular projects (11 projects), and one additional class for each of the eight larger projects in the repository. This resulted in 204 non-trivial classes.

From the latest two editions of the SBST unit test tool contests [53], [49] we selected 97 subjects: 46 classes from the third edition [53] and 51 classes from the fourth edition [49] of the contest. From the original set of classes in the two competitions, we removed all duplicate classes belonging to different versions of the same library since we would not expect any relevant difference over different versions in terms of branch, statement, or mutation coverage. We selected these two benchmarks because they contain java classes with different characteristics and belonging to different application domains. In addition, they have been recently used to assess the performance of different test generation techniques.

Our last benchmark is the one used in our previous conference paper [42], from which we consider all classes used in our previous empirical study. While the original dataset includes 64 classes, in the present study we report the results for 45 classes, since 19 classes are shared with the SF110 corpus and with the latest SBST unit test tool contest [49].

TABLE 1
Java projects and classes in our study

Project Name	Classes	Branches			Statements			Mutants		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
a4j	2	31	78	125	231	1134	2038	15	477	940
apbsmem	2	41	215	390	283	2303	4323	315	371	427
asphodel	1	42	42	42	353	353	353	84	84	84
at-robots2-j	2	37	81	125	512	672	832	233	336	439
battlecry	2	78	179	281	1588	1947	2306	349	604	859
beanbin	2	41	44	47	242	298	355	114	128	142
biblestudy	2	33	35	37	280	381	483	151	193	236
biff	1	817	817	817	7012	7012	7012	488	488	488
bpmail	1	27	27	27	215	215	215	51	51	51
byuic	4	39	357	740	296	2038	4846	59	281	664
caloriecount	3	25	94	232	58	541.67	1425	5	311	909
celwars2009	2	11	185	360	135	1137	2139	39	448	857
checkstyle	6	9	41	111	92	294	412	16	214	517
classviewer	3	60	155	235	327	1219	2315	208	451	630
commons-cli	2	133	146	159	605	788	972	323	407	492
commons-codec	1	504	504	504	2385	2385	2385	725	725	725
commons-collections	3	28	113	221	276	389	576	289	671	904
commons-lang	14	40	309	1163	216	1543	5348	125	571	1086
commons-math	21	26	104	266	165	952	3184	57	467	979
compiler	9	73	293	758	121	1178	3734	151	448	787
corina	1	55	55	55	282	282	282	152	152	152
db-everywhere	2	13	83	153	97	900	1704	13	297	582
dcparseargs	1	80	80	80	571	571	571	220	220	220
diebierse	2	27	54	81	250	636	1022	39	213	388
diffi	2	20	27	35	152	248	344	214	241	268
dsachat	2	69	83	98	857	880	904	319	375	432
dvd-homevideo	3	29	55	84	947	1188	1386	52	82	98
ext4j	2	11	75	139	69	463	857	58	161	265
feudalismgame	3	11	285	788	104	1796	4873	99	159	226
fim1	2	25	49	73	170	655	1140	37	100	163
firebird	3	99	445	1040	581	2698	6591	0	245	387
fixsuite	2	35	54	74	335	483	631	108	204	301
follow	2	20	27	35	108	214	321	449	489	529
fps370	2	70	110	151	668	1320	1973	726	807	888
freemind	2	29	118	208	434	1027	1621	132	476	820
gangup	1	32	32	32	438	438	438	191	191	191
geo-google	2	24	27	30	44	95	147	20	59	99
gfarcegestionfa	2	76	99	123	503	826	1150	146	515	884
glengineer	2	23	69	115	325	491	658	154	267	380
gsftp	2	29	60	91	191	650	1110	104	471	839
guava	11	28	115	325	46	385	1114	53	390	886
heal	2	38	106	175	448	1155	1863	88	174	261
hft-bomberman	2	13	70	128	134	491	849	21	86	151
hibernate	1	4	4	4	62	62	62	6	6	6
httpanalyzer	2	56	128	200	656	2824	4992	407	537	667
ifx-framework	1	72	72	72	462	462	462	139	139	139
imsmart	1	15	15	15	69	69	69	29	29	29
inspirento	2	27	61	95	327	488	649	221	331	441
io-project	1	66	66	66	282	282	282	96	96	96
ipcalculator	2	55	79	103	1077	1252	1428	300	362	425
javabullboard	2	98	119	141	715	942	1170	432	445	458
javaml	7	12	39	56	120	303	448	58	194	273
javathena	4	49	206	330	197	1995	4279	65	372	784
javaviewcontrol	3	32	876	2373	179	2989	7027	91	446	712
javex	1	173	173	173	1023	1023	1023	208	208	208
jaw-br	2	28	38	48	196	232	269	61	78	95
jclo	1	133	133	133	987	987	987	383	383	383
jcvi-javacommon	2	29	45	61	232	236	240	169	251	334
jdbacl	2	188	193	198	1339	1475	1611	592	653	715
jdom	5	50	111	264	168	592	1116	18	298	895
jfree-chart	12	13	235	1005	74	1647	6610	18	488	907
jgaap	1	23	23	23	1245	1245	1245	59	59	59
jhandballmoves	2	26	29	32	234	307	380	23	41	60
jiggler	4	33	70	118	459	648	1205	211	555	985
jipa	2	24	79	134	124	453	783	87	241	395
jiprof	4	76	451	824	430	2612	5299	160	499	774
jmca	4	199	2515	7939	966	5887	16624	0	329	559
joda	13	53	214	697	366	1003	1892	152	469	866
jopenchart	2	20	56	92	241	690	1139	72	300	529
jsci	4	84	245	436	760	2331	3756	223	597	980
jsecurity	3	36	91	170	207	444	725	167	272	438
jtailgui	2	13	18	23	158	244	331	21	107	193
jwbf	2	45	48	52	225	312	399	101	109	117
lagoon	2	63	64	65	675	841	1007	80	97	115
lavalamp	1	18	18	18	122	122	122	61	61	61
lhamacaw	2	23	46	70	416	565	714	115	144	174
liferay	2	78	85	93	406	434	462	55	160	266
lilith	1	134	134	134	496	496	496	183	183	183
lotus	2	24	26	28	122	123	125	12	31	51
mygrid	2	28	28	28	113	113	113	35	35	35
netweaver	2	69	71	73	249	284	320	46	100	155
newzgrabber	3	32	126	268	404	973	1552	48	301	625
noen	2	67	69	71	316	456	597	145	200	256
nutzenportfolio	2	21	34	48	817	856	895	78	104	131
objectexplorer	2	17	96	175	47	517	987	48	147	246
omjstate	1	30	30	30	139	139	139	34	34	34
openhre	2	50	73	97	220	577	934	143	198	253
openjms	2	34	67	101	190	540	891	70	129	189
pdfsam	2	19	30	41	248	313	378	57	83	110
petsoar	2	16	31	47	61	182	303	56	79	102
quickserver	4	69	218	501	488	1669	3697	15	257	747
resources4j	1	176	176	176	1539	1539	1539	714	714	714
rif	1	21	21	21	94	94	94	56	56	56
saxpath	2	55	269	484	115	574	1034	128	393	659
schemaspy	2	17	198	380	140	1190	2241	21	311	602
scribe	6	2	13	37	14	113	355	1	24	47
sfmis	1	32	32	32	437	437	437	427	427	427
shop	4	41	108	192	316	828	1329	175	303	465
squirrel-sql	2	21	36	51	51	90	130	93	100	108
sugar	2	24	37	51	113	298	484	8	75	142
summa	4	30	197	372	185	1222	2114	95	357	680
sweethome3d	4	155	339	619	841	1913	3368	165	433	675
tartarus	3	228	344	514	1164	2094	3775	638	835	942
templateit	2	31	42	54	123	258	394	59	86	113
trans-locator	2	23	28	34	297	424	551	214	339	464
trove	10	0	124	281	232	683	1842	109	683	999
tullibee	2	20	20	21	97	101	106	175	186	198
twfbplayer	2	74	115	156	424	839	1255	383	402	422
twitter4j	7	25	104	319	227	1313	6320	107	257	401
vuze	2	119	126	134	657	666	675	292	387	482
water-simulator	2	59	98	137	888	1162	1436	274	606	939
weka	4	29	338	809	274	2773	7809	126	405	793
wheelwebtool	4	40	272	817	371	1598	3944	77	392	679
wikipedia	4	19	35	72	61	325	796	52	143	394
xbus	2	18	23	28	170	219	269	54	68	83
xisemele	1	27	27	27	152	152	152	70	70	70
xmlenc	2	182	698	1214	693	1439	2186	466	657	848
Total	346									

In total, the selected 346 Java classes have 360,941 statements, 61,553 branches, and 117,816 mutants that are considered as coverage targets in our experiment. Table 1 reports the characteristics of the selected classes, grouped by project. Specifically, it reports the number of classes selected for each project as well as minimum, average (mean), and maximum number of statements, branches and mutants contained in

3. http://www.evosuite.org/experimental-data/sf110/
4. http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/
5. A method with cyclomatic complexity equal to five contains at least two *conditional* statements.

those classes. The number of branches in each class ranges between 2 (a class from the SBST contest) and 7,939, while the number of statements ranges between 14 and 16,624; the number of mutants ranges between 0 and 1,086 (EvoSuite was unable to inject any mutant into two of the selected classes; however, we kept them for the other two coverage criteria). It is worth noticing that, according to the survey by Li *et al.* [34], existing many-objective algorithms can deal with up to 50 objectives, while in our study most of selected classes contain hundreds and thousands of objectives (e.g., mutants to kill). This confirms the need for designing new many-objective algorithms suitable for test case generation given the very high number of coverage targets as compared to the number of objectives (≤ 50) encountered in traditional many- and multi-objective problems.

4.2 Research Questions

The empirical evaluation aims at answering the following research questions:

- **RQ**$_1$**:** *How does DynaMOSA perform compared to alternative approaches on branch coverage?*
- **RQ**$_2$: *How does DynaMOSA perform compared to alternative approaches on statement coverage?*
- **RQ**$_3$: *How does DynaMOSA perform compared to alternative approaches on strong mutation coverage?*

The three research questions above aim at evaluating the benefits introduced by the many-objective reformulation of testing criteria —branch, statement and mutation— and to what extent the proposed DynaMOSA algorithm is able to cover more test targets (**effectiveness**) when compared to alternative approaches for test generation problems. When no difference is detected in the number of covered test targets, we analyse to what extent the proposed approach is able to reach higher code coverage when considering different search time intervals (**efficiency**).

To evaluate the internal functioning of our new sorting algorithm (Algorithm 3), which combines both our *preference criterion* and *Pareto-based ranking*, we formulate the following additional research question:

- **RQ**$_4$ **(internal assessment)**: *How do preference criterion and Pareto dominance affect the effectiveness of DynaMOSA?*

In answering this research question, we consider the two sub-questions below:

- **RQ**$_{4.1}$: *What is the effectiveness of DynaMOSA's Pareto-based ranking alone, when applied without the preference criterion?*
- **RQ**$_{4.2}$: *What is the effectiveness of DynaMOSA's preference criterion alone, when applied without Pareto-based dominance ranking?*

4.3 Baseline Selection

To answer our first three research questions, we selected the following approaches for comparison with DynaMOSA:

- *Many-objective sorting algorithm* (MOSA). It is the many-objective solver we developed in our previous paper [42] for branch coverage. While in our previous paper we used MOSA only for branch coverage criterion, in this paper (see section 3.2) we have described how this algorithm can be also used for other criteria. Therefore, we use MOSA as baseline for the first three research questions.
- *Whole Suite* approach with *Archive* (WSA). It is an extension of the more traditional *whole suite* approach recently proposed by Rojas et al. [48] and implemented in EvoSuite. In particular, WSA is a hybrid approach that implements some routines of our MOSA algorithm along with the original whole suite approach. First, WSA uses an *archive* to store test cases that cover one or more coverage targets (e.g., branches) [48]. Such an archive operator works at test case level and not at test suite level, which is the usual granularity of the whole suite approach. Second, WSA focuses the search on *uncovered coverage targets only*, which is also one of the core properties we developed in MOSA (and DynaMOSA too). Finally, the final test suite is not represented by the best individual (candidate suite) from the last generation of GA, but it is artificially synthesised by taking those test cases stored in the *archive* and that come from different suites. While WSA has been shown to outperform both standard WS and *one-target* approaches, it departs from the original WS principle of *test suite evolution* [48]. For our study, we could use WSA as baseline only for **RQ**$_1$ and **RQ**$_3$ because no archive strategy has been implemented in EvoSuite for statement coverage.
- Traditional *Whole Suite* approach (WS). This is the traditional, pure whole suite approach without any archive strategy [17]. Thus, the final test suite provided to developers is the best individual (candidate suite) from the last generation of the genetic algorithm. We used WS as baseline only for **RQ**$_2$ since for statement coverage no WSA variant is available in EvoSuite. While WS could be used as additional baseline also for **RQ**$_1$ and **RQ**$_3$, we discarded this scenario since Rojas et al. [48] already showed the superiority of WSA over WS for branch coverage.

MOSA is a natural baseline to compare with, in order to investigate the benefits achieved when dynamically reducing the number of objectives to be optimised at each generation. This comparison allows us also to empirically validate our discussion on the computational cost of DynaMOSA vs. MOSA reported in Section 3. WSA is the main baseline, because it has been shown to outperform standard WS and the single-target approach. Since WSA is a hybrid solution that incorporates the archiving core routines of MOSA (and DynaMOSA), such comparison allows us to understand *to what extent benefits in effectiveness or efficiency stem from our many-objective reformulation* and not from our archiving mechanism. It is also worth noticing that WSA won the last SBST tool contest (edition 2016) against other tools, such as JTexPert [50] (evolutionary), T3 [45] (random testing). Finally, DynaMOSA, MOSA, WS and WSA are implemented in the same tool (i.e., EvoSuite) avoiding possible confounding factors [48] due to the usage of different

tools with different implementation choices (e.g., different libraries to build the control flow graph).

To address our last research question (**RQ**$_4$), we selected the following algorithms to compare against DynaMOSA:

- DynaMOSA with *Pareto ranking* alone (DynaMOSA$_{\text{Rank}}$). This variant of DynaMOSA uses only the Pareto ranking to assign test cases to different non-dominated fronts, without using our *preference criterion* to further increase the selection pressure (see Algorithm 3). Therefore, this variant corresponds to the traditional NSGA-II algorithm, enriched with the archive strategy and with a dynamic selection of coverage targets. DynaMOSA$_{\text{Rank}}$ is used as baseline to answer RQ$_{4.1}$.
- DynaMOSA with *preference criterion* alone (DynaMOSA$_{\text{Pref}}$). This variant of DynaMOSA uses our preference criterion without applying the Pareto ranking algorithm to build the other non-dominated fronts. Therefore, at each generation it creates only two fronts: the first containing all best test cases selected according to *preference criterion*, while all the remaining tests are assigned to the second front (see Algorithm 3). This variant also uses an *archive* to keep track of test cases as soon as they reach uncovered targets (see Algorithm 2). DynaMOSA$_{\text{Pref}}$ is used as baseline to answer RQ$_{4.2}$.

4.4 Prototype Tool

We have implemented our many-objective algorithms MOSA and DynaMOSA in a prototype tool that extends the EvoSuite test data generation framework [17], [48]. In particular, we implemented the two many-objective GAs as described in Section 3 within EvoSuite version `1.0.3`, downloaded from GitHub[6] on May 15th, 2016. All other details (e.g., encoding schema, genetic operators, etc.) are those implemented in EvoSuite [17], [48]. According to the encoding schema available in EvoSuite for test cases, a test case is a statement sequence $t = \langle s_1, s_2, \ldots, s_n \rangle$ of length n. Each statement has a type belonging to one of five kinds of statements: (i) *primitive statement*, (ii) *constructor statement*, (iii) *field statement*, (iv) *method statement* and (v) *assignment statement*. Note that the representation has variable size, i.e., the number of statements n in each test can vary during the GA search. Test cases are evolved through the application of *selection*, *crossover* and *mutation* operators. Pairs of test cases (parents) are selected using *tournament selection* according to their non-dominance ranks. Parents are recombined using the *single-point crossover*, which generates offsprings by exchanging statements between the two parent test cases. Finally, test cases are mutated using *uniform mutation*, which randomly *removes*, *changes* or *inserts* statements in a given test case. Since our many-objective algorithm uses Pareto-based ranking, we implemented the tournament selection operator based on *dominance ranking* and *crowding distance* as defined in NSGA-II (see Algorithm 2).

The tool, along with a replication package, is available for download here: http://selab.fbk.eu/kifetew/mosa.html.

6. https://github.com/EvoSuite/evosuite

4.5 Parameter setting

There are several parameters that control the performance of the algorithms being evaluated. We adopted the default parameter values used by EvoSuite [17], as it has been empirically shown [2] that the default values, which are also commonly used in the literature, give reasonably acceptable results. Thus, here we only report a few of the important search parameters and their values:

- **Population size**: we use the default population size in EvoSuite, which is set to 50 individuals.
- **Crossover**: we use the default crossover operator in EvoSuite, which is *single-point* crossover with *crossover probability* set to 0.75 [17].
- **Mutation**: test cases are mutated by adding, deleting, changing statements using *uniform mutation* with *mutation probability* equal to $1/size$, where $size$ is the number of statements contained in the test case to mutate [17].
- **Selection**: we use the *tournament* selection available in EvoSuite, with default *tournament size* equal to 10.
- **Search Timeout**: the search stops when the maximum search budget (`max_time`) is reached or if 100% coverage is achieved before consuming the total allocated time. We set `max_time` to five minutes for branch and statement coverage, while we increased the search timeout up to eight minutes for strong mutation coverage. We used a larger budget for strong mutation coverage because of the additional overhead required by this criterion to re-execute each test case against the target mutants.

It is important to notice that WSA, MOSA and DynaMOSA work at two different granularity levels: WSA evolves test suites while MOSA and DynaMOSA evolve test cases. Therefore, in our many-objective algorithms the number of individuals (i.e., test cases) is always the same (50 test cases). Differently, in WSA each individual is a test suite which usually contains a variable number of test cases. The population is composed of 50 test suites, where each suite can have a variable number of test cases that can be added, deleted or modified at each generation. The crossover operator also works at two different levels of granularity: the single point crossover in MOSA and DynaMOSA creates two offsprings by recombining statements from the two parent test cases; in WSA suite offsprings are generated by recombining test cases composing the two parent suites. For WS and WSA the winner of the tournament selection is the test suite with the smaller (fitter) whole-suite fitness function, while in MOSA and DynaMOSA the winner is the test case with the lowest *non-dominance rank* or with the largest *crowding distance* for test cases having the same non-dominance rank.

4.6 Experimental Protocol

For each class, and for each coverage criterion (statement, branch, mutation), each search strategy (WSA, MOSA, DynaMOSA) is executed and the resulting coverage is recorded. The three coverage criteria are implemented as fitness functions in EvoSuite and are used as guidance for the search algorithms. For strong mutation, EvoSuite uses its own mutation engine that generates mutated versions

of the code under test, which correspond to the adequacy targets (mutants to *kill*). In each execution, an overall time limit is imposed so that the run of an algorithm on a class is bounded with respect to time. Hence, the search stops when either full coverage is reached or the total allocated time is elapsed. To allow reliable detection of statistical differences between the strategies, each run is repeated 10 times. Consequently, we performed a total of 3 (search strategies) × 3 (coverage criteria) × 346 (classes) × 10 (repetitions) = 31,140 test generator executions.

To answer $\mathbf{RQ}_1$, $\mathbf{RQ}_2$, and $\mathbf{RQ}_3$ in terms of *effectiveness*, we measure the percentage of covered branches, statements, and the percentage of killed mutants:

$$branch_cov = \frac{\text{\#covered branches}}{\text{\#total branches to be covered}}$$

$$statement_cov = \frac{\text{\#covered statements}}{\text{\#total statements to be covered}}$$

$$mutation_cov = \frac{\text{\#killed mutants}}{\text{\#total mutants to be killed}}$$

To measure the aforementioned metrics, we rely on the internal engine of EvoSuite [48], which performs some post-processing by re-executing the test suites obtained at the end of the search process. During post-processing, EvoSuite removes those statements that do not contribute to the final coverage from each test case. It also removes redundant test cases from the final test suite. After this minimisation process, the final test suite is re-executed to collect its final coverage metric. We execute the three selected algorithms separately for each single coverage criterion, thus obtaining the corresponding coverage score after the post-processing.

We also statistically analyse the results, to check whether the differences between two different algorithms are statistically significant or not. To this aim we used the non-parametric Wilcoxon test [8] with a p-value threshold of 0.05. Significant p-values indicate that the null hypothesis can be rejected in favour of the alternative hypothesis, i.e., that one of the algorithms reaches a significantly higher coverage. Besides testing the null hypothesis, we used the Vargha-Delaney ($\hat{A}_{12}$) statistic [54] to measure the effect size, i.e., the magnitude of the difference between the coverage scores achieved by two different algorithms. The Vargha-Delaney ($\hat{A}_{12}$) statistic is equal to 0.50 if the two compared algorithms are equivalent, $\hat{A}_{12} \neq 0.50$ otherwise. In our case, an $\hat{A}_{12} > 0.50$ indicates that DynaMOSA reaches a higher coverage than the alternative algorithm, while $\hat{A}_{12} < 0.50$ indicates that the alternative algorithm is better than DynaMOSA.

To quantify the *efficiency* of the compared algorithms, we analyse their capability of reaching higher code coverage at different points in time. While the *effectiveness* measures the algorithm performance only at the end of the allocated time (i.e., after five or eight minutes), we want also to analyse how algorithms perform during the search. A simple way to perform such a comparison consists of plotting the percentage of branches, statements or mutants killed by each algorithm at predefined time intervals during the search process. Such a *convergence graph* allows us to compare two or more search algorithms, showing the percentage of covered test goals at the same point in time. To this aim, we collected the coverage achieved after intervals of 20 seconds for each independent run, resulting in 16 coverage points for each run (25 points for strong mutation coverage). To summarise the *efficiency* of the experimented algorithms using a single scalar value, we computed the overall *convergence rate* as the area under the curve delimited by the *convergence graph*. More formally, let $P_A = \{p_0, \ldots, p_h\}$ be the set of points in time considered for a given algorithm A. Let $cov(p_i)$ be the percentage of targets covered by A at time p_i. Let I be the time elapsed between two consecutive points p_i and p_{i+1}. The Area-Under-Curve (AUC) enclosed by these points is computed using the trapezoidal rule:

$$AUC(P_A) = \frac{\sum_{i=0}^{h-1} [cov(p_i) + cov(p_{i+1})] \times I}{2 \times TotalRunningTime} \times 100 \quad (10)$$

Such metric takes values in $[0 : 100]$. Higher AUC values are preferable because they indicate that the algorithm is faster in achieving higher levels of code coverage.

It is important to remark that we consider *efficiency* only for classes where we do not observe a significant difference in terms of *effectiveness*, i.e., branch coverage, statement coverage or mutation coverage. In other words, the *efficiency* performance metric is considered for comparison only in cases where the algorithms under analysis achieve the same coverage level. This is because focusing only on *efficiency* without considering *effectiveness* is not meaningful (low coverage can be easily achieved very efficiently). As done for *effectiveness*, we measure the statistical significance of differences in *efficiency* using the non-parametric Wilcoxon test [8] with a p-value threshold of 0.05, and the Vargha-Delaney ($\hat{A}_{12}$) statistic [54] to measure the effect size. In this case, significant p-values indicate that one of the two algorithms being compared converges quicker than the other to the final coverage. $\hat{A}_{12} > 0.50$ indicates that DynaMOSA converged to the final coverage more quickly than the alternative algorithm, while $\hat{A}_{12} < 0.50$ indicates that DynaMOSA was less efficient than the alternative algorithm, e.g., MOSA).

For the internal assessment ($\mathbf{RQ}_4$), we investigate the *effectiveness* of DynaMOSA against two of its variants: (i) a first variant, namely $\text{DynaMOSA}_{\text{Rank}}$, which uses only the Pareto-based dominance ranking; and (ii) a second variant, namely $\text{DynaMOSA}_{\text{Pref}}$, which uses our *preference criterion* alone. For DynaMOSA and for both DynaMOSA variants we use an *archive* to store test cases as soon as they cover any yet uncovered target. Therefore, any difference in efficiency can be attributed only to the different ranking strategies used by DynaMOSA or by its variants. To save space, the comparison is reported only for branch coverage, since consistent results are obtained for statement and strong mutation coverage.

Given the large number of classes considered in the experiments, we cannot report all p-values and Vargha-Delaney $\hat{A}_{12}$ scores measured for each class. Therefore, for each project in our sample we report the number of times (number of classes) the comparison is statistically significant at level 0.05, as well as the corresponding number of times (percentage) the Vargha-Delaney ($\hat{A}_{12}$) statistic is greater or lower than 0.50. Results at class level are available in our replication package for the interested readers.

5 RESULTS

This section discusses the results of our study with the aim of answering the research questions formulated in Section 4. In the following, when comparing two different algorithms we will refer to *difference* in code coverage as the arithmetic difference between the two corresponding percentage coverage scores.

5.1 RQ1: *How does DynaMOSA perform compared to alternative approaches on branch coverage?*

Table 2 summarises the results achieved by WSA, MOSA, and DynaMOSA for branch coverage. In particular, it reports the percentage of classes in each project where the effect size ($\hat{A}_{12}$) is statistically greater/lower than 0.50. In the following, we will refer to 335 classes only since in 11 classes in our sample EvoSuite crashed for some internal errors.

From Table 2 we can observe that DynaMOSA is significantly better than WSA in 93 classes out of 335, corresponding to 28% of classes in our sample. The improvement achieved in branch coverage using DynaMOSA, for classes where statistically significant differences were observed, ranges between 1% and 67%, with an average improvement of 8%. On the other hand, we observe that WSA is significantly better than DynaMOSA in 10 classes only, while no statistically significant difference is observed in the remaining 232 cases. Looking at the results at project level, we find that in 43% of the projects DynaMOSA achieved higher branch coverage than WSA on at least one class in the project.

For example, if we consider class `ControlFlowAnalysis` extracted from the *compiler* project, we can observe that WSA covers 184 branches (on average) while DynaMOSA covers 287 (+25%) branches (on average) using the same search budget. For larger classes, differences are even larger. For example, for class `JavaParserTokenManager` extracted from the project *jmca*, WSA covers 994 branches (on average) against 1,157 (+10%) branches covered (on average) by DynaMOSA using the same search budget. Project *jmca* is particularly interesting since it contains the largest classes in our sample (two of the five selected classes have more than 1,500 branches). On this project DynaMOSA is significantly better than WSA in 75% of the selected classes with $\hat{A}_{12} = 1$ in all cases.

Only in a few cases WSA is significantly better than DynaMOSA (10 classes out of 335), with an average difference, over classes with significant differences, of 3% in terms of branch coverage. The class with the highest difference is `JSJshop` extracted from the project *shop*. For this class, WSA is able to cover 93 branches (on average) against 46 branches covered by DynaMOSA. After manual investigation, we discovered that such a negative result is due to test cases discarded because of the usage of test case length (*size*) as secondary selection criterion in our *preference criterion*. In order to verify this conjecture, we ran DynaMOSA with the test *size* as secondary criterion disabled. The obtained results show that DynaMOSA covers on average 95 branches (+1.75%), with an effect size equal to 0.83 and p-value $= 0.01$. Therefore, we conclude that test case *size* as secondary preference criterion can have sometimes negative effects on the final coverage. Thus, an adaptive version of the secondary preference criterion (e.g., disabling *size* when there is no improvement of *branch*

TABLE 2
Mean **branch** coverage achieved for each project. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	Classes	Branch Coverage			$\hat{A}_{12}$ Statistics			
					DynaMOSA vs WSA		DynaMOSA vs MOSA	
		WSA	MOSA	Dyna-MOSA	%> 0.50	%< 0.50	%> 0.50	%< 0.50
a4j	2	32.64	31.24	32.24	-	50.00	-	-
apbsmem	2	50.13	50.13	50.13	-	-	-	-
asphodel	1	2.38	2.38	2.38	-	-	-	-
at-robots2-j	2	68.49	80.80	81.20	50.00	-	-	-
battlecry	2	36.07	16.85	39.81	-	-	-	-
beanbin	2	79.14	77.92	77.92	-	50.00	-	-
biblestudy	2	84.85	84.85	84.85	-	-	-	-
biff	1	32.44	30.05	31.27	-	-	-	-
byuic	3	51.01	52.76	54.56	33.33	-	66.67	-
caloriecount	3	97.28	98.99	98.99	66.67	-	-	-
celwars2009	2	9.37	9.37	9.37	-	-	-	-
checkstyle	6	84.31	87.29	88.21	33.33	-	-	-
classviewer	3	61.19	61.29	61.49	-	-	-	-
commons-cli	2	97.58	97.58	97.42	-	-	-	-
commons-codec	1	91.37	90.97	92.46	100.00	-	100.00	-
commons-collections	2	97.29	96.78	97.74	50.00	-	50.00	-
commons-lang	14	94.89	94.41	95.40	28.57	-	28.57	-
commons-math	21	71.77	67.90	75.70	47.62	-	38.10	-
compiler	9	35.42	36.72	40.93	44.44	-	44.44	-
corina	1	43.64	43.64	43.64	-	-	-	-
db-everywhere	2	40.87	47.08	47.57	50.00	-	-	-
dcparseargs	1	97.50	97.50	97.50	-	-	-	-
diebierse	2	61.11	79.63	94.44	50.00	-	50.00	-
diffi	2	92.86	97.14	97.14	50.00	-	-	-
dsachat	2	38.14	37.12	38.91	-	-	50.00	-
dvd-homevideo	3	13.03	13.03	13.03	-	-	-	-
ext4j	2	97.66	96.94	97.12	-	-	-	-
feudalismgame	3	4.39	4.39	4.39	-	-	-	-
fim1	2	8.00	8.00	8.00	-	-	-	-
firebird	3	74.02	75.24	75.62	33.33	-	33.33	-
fixsuite	2	61.78	61.74	61.80	50.00	-	-	-
follow	2	71.07	74.64	67.14	-	-	-	-
fps370	2	1.05	1.05	1.05	-	-	-	-
freemind	2	68.17	84.69	84.93	50.00	-	-	-
gangup	1	81.25	81.25	81.25	-	-	-	-
geo-google	2	92.92	95.42	97.92	50.00	-	-	-
gfarcegestionfa	1	87.80	89.43	89.43	100.00	-	-	-
glengineer	2	93.48	95.65	95.87	50.00	-	-	-
gsftp	2	37.99	38.54	37.99	-	-	-	-
guava	10	69.58	72.26	72.42	40.00	-	40.00	-
heal	2	91.86	93.43	93.71	-	-	-	-
hft-bomberman	2	54.69	60.94	58.79	50.00	-	-	50.00
hibernate	1	25.00	25.00	25.00	-	-	-	-
httpanalyzer	2	40.93	40.93	40.93	-	-	-	-
ifx-framework	1	69.44	69.44	69.44	-	-	-	-
imsmart	1	100.00	100.00	100.00	-	-	-	-
inspirento	2	87.04	86.77	87.04	-	-	-	-
io-project	1	95.45	95.45	97.73	-	-	-	-
ipcalculator	2	82.15	83.09	83.09	50.00	-	-	-
javabullboard	2	80.58	78.99	80.05	-	-	-	-
javaml	7	83.92	82.90	82.90	-	14.29	-	-
javathena	4	44.81	45.23	46.82	50.00	-	-	-
javaviewcontrol	3	68.98	61.05	71.91	66.67	-	100.00	-
javex	1	88.44	84.39	88.44	-	-	100.00	-
jaw-br	2	8.93	8.93	8.93	-	-	-	-
jclo	1	87.59	82.71	93.61	100.00	-	100.00	-
jcvi-javacommon	2	100.00	100.00	100.00	-	-	-	-
jdbacl	2	84.32	85.20	85.45	50.00	-	-	-
jdom	5	83.44	83.62	84.16	20.00	-	80.00	-
jfree-chart	12	80.57	81.63	85.16	58.33	-	33.33	-
jgaap	1	82.61	91.30	95.65	100.00	-	-	-
jhandballmoves	2	22.60	22.60	22.60	-	-	-	-
jiggler	4	88.02	88.06	87.96	-	-	-	-
jipa	2	92.54	91.79	91.98	-	50.00	-	-
jiprof	4	77.57	79.87	80.56	50.00	-	-	-
jmca	4	48.98	52.32	53.66	75.00	-	25.00	-
joda	13	82.40	82.99	84.89	38.46	-	30.77	-
jopenchart	2	95.65	99.18	98.37	50.00	-	-	-
jsci	4	88.62	92.14	92.87	75.00	-	25.00	-
jsecurity	3	74.49	75.04	75.13	33.33	-	-	-
jtailgui	2	61.04	21.91	41.47	-	-	-	-
lagoon	2	23.20	23.20	23.20	-	-	-	-
lavalamp	1	100.00	100.00	100.00	-	-	-	-
lhamacaw	2	10.84	10.84	10.84	-	-	-	-
liferay	2	98.12	97.85	97.58	-	-	-	-
lilith	1	100.00	100.00	100.00	-	-	-	-
lotus	2	70.83	70.83	70.83	-	-	-	-
mygrid	2	96.43	96.43	96.43	-	-	-	-
netweaver	2	98.59	98.23	98.59	-	-	-	-
newzgrabber	3	27.32	19.30	20.97	-	33.33	33.33	-
noen	2	95.07	97.89	96.48	-	-	-	-
nutzenportfolio	2	40.77	42.63	61.68	50.00	-	50.00	-
objectexplorer	2	50.86	50.86	50.86	-	-	-	-
omjstate	1	100.00	100.00	100.00	-	-	-	-
openhre	2	100.00	100.00	100.00	-	-	-	-
openjms	2	83.21	86.63	84.19	-	-	-	-
pdfsam	2	61.94	44.87	44.87	-	50.00	-	-
petsoar	2	83.51	79.79	80.85	-	50.00	-	-
quickserver	4	61.45	63.19	64.66	25.00	-	25.00	-
rif	1	100.00	100.00	100.00	-	-	-	-
saxpath	2	94.58	94.47	95.04	50.00	-	50.00	-
schemaspy	2	45.21	56.84	67.63	100.00	-	50.00	-
scribe	6	100.00	100.00	100.00	-	-	-	-
sfmis	1	100.00	100.00	100.00	-	-	-	-
shop	4	67.79	56.82	56.82	25.00	25.00	-	-
squirrel-sql	2	51.54	51.54	51.54	-	-	-	-
sugar	2	84.80	90.20	90.20	-	-	-	-
summa	4	35.58	38.10	37.37	25.00	-	-	-
sweethome3d	4	39.31	45.74	47.47	100.00	-	75.00	-
tartarus	3	75.42	74.69	76.60	66.67	-	100.00	-
templateit	2	62.96	62.96	62.96	-	-	-	-
trans-locator	2	49.30	51.47	51.47	50.00	-	-	-
trove	7	81.77	82.68	83.70	28.57	-	28.57	-
tullibee	2	100.00	100.00	100.00	-	-	-	-
twfbplayer	2	89.86	88.13	89.25	-	-	50.00	-
twitter4j	7	91.20	94.35	94.59	14.29	14.29	14.29	-
vuze	2	5.88	9.42	9.98	50.00	-	50.00	-
water-simulator	2	14.60	14.96	14.78	-	-	-	-
weka	4	68.20	67.65	69.59	25.00	-	25.00	-
wheelwebtool	4	88.34	90.50	90.97	50.00	-	25.00	-
wikipedia	4	67.88	65.92	66.86	-	-	-	-
xbus	2	62.40	63.29	63.29	50.00	-	-	-
xisemele	1	77.78	81.48	77.78	-	-	-	-
xmlenc	2	64.15	63.11	63.86	-	50.00	-	-
Mean over all projects		84.61	85.45	87.42				
No. cases DynaMOSA significantly better than WSA					**93 (27.76%)**			
No. cases DynaMOSA significantly worse than WSA					10 (2.96%)			
No. cases DynaMOSA significantly better than MOSA					**64 (19.10%)**			
No. cases DynaMOSA significantly worse than MOSA					1 (0.30%)			

TABLE 3
Efficiency of DynaMOSA and WSA for branch coverage. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	Classes	Efficiency		$\hat{A}_{12}$ Statistics	
		DynaMOSA	WSA	%> 0.50	%< 0.50
checkstyle	4	82.55	82.39	-	25.00
classviewer	3	58.98	57.55	33.33	-
commons-lang	10	92.19	92.43	20.00	10.00
commons-math	11	76.69	76.53	9.09	9.09
dsachat	2	29.85	29.01	50.00	-
dvd-homevideo	3	11.13	11.15	-	33.33
firebird	2	93.49	93.52	-	50.00
gangup	1	55.42	55.10	100.00	-
guava	6	72.78	71.77	16.67	-
heal	2	92.13	89.16	50.00	-
hft-bomberman	1	96.03	85.34	100.00	-
javabullboard	2	72.81	69.78	100.00	-
javaml	6	81.24	81.24	16.67	16.67
jdbacl	1	97.72	95.70	100.00	-
jdom	4	82.37	82.29	25.00	25.00
jfree-chart	5	75.31	74.50	40.00	-
jhandballmoves	2	21.15	21.25	-	50.00
joda	8	80.11	79.50	37.50	-
jsecurity	2	69.31	66.80	50.00	-
lotus	2	69.19	70.83	-	50.00
newzgrabber	2	22.21	22.55	50.00	50.00
noen	2	88.81	87.47	50.00	-
objectexplorer	2	50.86	48.74	50.00	-
twfbplayer	2	78.00	85.59	-	50.00
twitter4j	5	98.21	98.36	-	20.00
weka	3	62.82	64.01	-	33.33
wheelwebtool	2	97.27	90.46	100.00	-
wikipedia	1	94.76	94.74	100.00	-
xisemele	1	68.15	67.46	100.00	-
Mean		71.43	70.52		
No. cases DynaMOSA significantly better than WSA					**27 (11.79%)**
No. cases DynaMOSA significantly worse than WSA					13 (5.67%)

distance for some generations) might be beneficial.

For the 232 classes on which there is no statistically significant difference in branch coverage between WSA and DynaMOSA, we compare the two search strategies along the *efficiency* metric. The results of this comparison are summarised in Table 3. The table reports the median AUC metric for WSA and DynaMOSA as well as the results of the effect size ($\hat{A}_{12}$) aggregated by project. From the comparison, we observe that in 27 classes out of 232, DynaMOSA has a statistically significantly higher AUC metric compared to WSA, which means that the former would reach higher branch coverage if given a shorter search time. The average improvement for the AUC metric is 3%. The minimum improvement, 0.1%, occurs for class `Title` (from project *wikipedia*), and the maximum, 17%, for `DynamicSelectModel` (from project *wheelwebtool*). On the other hand, WSA turned out to be more efficient than DynaMOSA in 13 classes out of 232. However, for these classes the differences in terms of AUC scores are very small, being $< 0.1\%$ on average.

Let us now consider the comparison between DynaMOSA and MOSA. Results for branch coverage are reported in Table 2. We can observe that in the majority of the cases there is no statistically significant difference between the two many-objective algorithms according to the Wilcoxon test. In total, in 64 classes out of 335 (19%) DynaMOSA achieved significantly higher branch coverage, while in only one class ($< 1\%$) MOSA turned out to be better than DynaMOSA. Looking at the magnitude of the differences, we notice that in those cases where DynaMOSA achieved higher branch coverage, the improvements range between 1% and 71%, with an average increase of 7%. The largest improvement is obtained for `JTailPanel` (*jtailgui* project), for which DynaMOSA covers 21 branches (91%) while MOSA covers only 3 branches (13%). For the single class where MOSA outperformed DynaMOSA, namely

TABLE 4
Efficiency of DynaMOSA and MOSA for branch coverage. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	N. Classes	Efficiency		$\hat{A}_{12}$ Statistics	
		DynaMOSA	MOSA	%> 0.50	%< 0.50
biff	1	19.52	18.72	100.00	-
commons-lang	10	96.18	96.05	20.00	-
commons-math	13	66.51	65.77	7.69	-
compiler	5	32.24	28.62	20.00	-
diffi	2	93.54	92.52	50.00	-
gangup	1	55.42	55.10	100.00	-
geo-google	2	91.61	89.61	50.00	-
jfree-chart	8	87.78	87.20	-	12.50
jiprof	4	73.24	72.27	25.00	-
quickserver	3	67.82	67.06	33.33	-
twitter4j	6	91.25	91.22	16.67	-
wheelwebtool	3	82.88	79.23	33.33	-
Mean		71.50	70.28		
No. cases DynaMOSA significantly better than MOSA					12 (4.51%)
No. cases DynaMOSA significantly worse than MOSA					1 (0.38%)

`ServerGameModel` from *hft-bomberman*, the difference in terms of branch coverage is 4%. We observe that this class is very peculiar: it contains 128 branches and it is very expensive in execution time. Indeed, both DynaMOSA and MOSA could only execute less than 5 generations of their evolutionary algorithm (on average) within the five minutes of search budget.

In terms of *efficiency*, DynaMOSA and MOSA are statistically equivalent in the majority of those classes where there is no statistically significant difference on branch coverage. Indeed, as reported in Table 4 in only 12 classes out of 273 DynaMOSA yielded a statistically significantly higher AUC metric as compared to MOSA. More specifically, we observe that on these classes DynaMOSA produces +5% of branch coverage on average when considering a search budget lower than five minutes, which is the total search time set for branch coverage. The minimum improvement of 0.2% is yielded for class `Conversion` (from the *apache commons* library), and the maximum of 29% is achieved for `ExploitAssigns` (from project *compiler*). On the other hand, MOSA turned out to be more efficient than DynaMOSA in only one class, namely `TimePeriodValues` from *jfree-chart*, where the difference in terms of AUC is equal to just 0.21%.

Figure 4 shows an example of (average) branch coverage achieved over time by MOSA, DynaMOSA and WSA on `SortingTableModel`, a class for which we did not find any statistically significant difference in terms of efficiency or effectiveness for the three experimented approaches. From Figure 4, we can observe that in the first 20 seconds DynaMOSA is particularly efficient compared to the other alternatives. In less than 10 seconds, it reaches 96% of branch coverage. MOSA required almost 30 seconds to reach the same branch coverage, while WSA had the worst branch coverage scores within this initial time window. This result is very unexpected considering that WSA benefits from a larger population size (50 test suites with more than one test case each) with respect to DynaMOSA and MOSA (50 initial test cases in total). After consuming 60 seconds, all the three approaches reach the maximum branch coverage, which is 98% for this class. As a consequence, we did not detect any statistically significant difference in the final efficiency among DynaMOSA, MOSA and WS, but a substantial difference can be observed in the first 20 seconds (i.e., before we collect the first coverage point when computing the AUC

Fig. 4. Branch coverage over time for class *SortingTableModel* from project *caloriecount*

metric). Hence, the results in Figure 4 show that when a very limited search budget is given to the search (<20-30s), DynaMOSA reaches higher coverage much more quickly than the other two alternatives. We obtained consistent results for the other classes in our study for which we did not find any significant difference in the final effectiveness and efficiency: DynaMOSA has always the steepest coverage increase in the initial time window.

> In summary, we can conclude that DynaMOSA achieves **higher or equal branch coverage** as compared to both WSA and MOSA. Moreover, DynaMOSA converges to such coverage more quickly especially when limited time is given to the search. Finally, our many-objective reformulation provides further benefits to branch coverage as compared to the usage of the archive strategy, since it outperformed WSA, which extends WS with the archive strategy.

5.2 RQ2: *How does DynaMOSA perform compared to alternative approaches on statement coverage?*

Table 5 reports the statement coverage achieved by WS, MOSA, and DynaMOSA for the classes in our experiment, grouped by project. Table 5 also reports the percentage of classes in each project where the effect size ($\hat{A}_{12}$) is statistically greater or lower than 0.50. Since in 7 classes in our sample EvoSuite crashed for some internal errors, in the following we will refer to 339 classes only for which EvoSuite could generate tests.

For statement coverage, we can notice that DynaMOSA is significantly better than WS on 173 classes out of 339 (51%), while WS is significantly better in only 5 cases ($< 2\%$). In the remaining 160 classes, no statistically significant difference is observed according to the Wilcoxon test. At project level, we find that in 65% of the projects DynaMOSA achieves significantly higher coverage than WS on at least one class in the project. Overall, the usage of DynaMOSA leads to an increase in statement coverage equal to 11% on average, over classes where significant differences were observed. The largest improvement is obtained for class `TwitterImpl` selected from project *twitter4j*, where

TABLE 5
Mean **statement** coverage achieved for each project. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	Classes	Statement Coverage			$\hat{A}_{12}$ Statistics			
					DynaMOSA vs WS		DynaMOSA vs MOSA	
		WS	MOSA	Dyna-MOSA	%> 0.50	%< 0.50	%> 0.50	%< 0.50
a4j	2	54.92	55.05	55.08	-	-	-	-
apbsmem	2	71.41	71.41	71.41	-	-	-	-
asphodel	1	6.80	6.80	6.80	-	-	-	-
at-robots2-j	2	85.32	88.05	88.06	50.00	-	-	-
battlecry	2	25.56	29.70	37.43	-	-	-	-
beanbin	2	86.05	86.05	86.05	-	-	-	-
biblestudy	2	89.67	90.48	90.34	50.00	-	-	-
biff	1	16.53	26.65	28.52	100.00	-	100.00	-
byuic	3	40.15	53.24	54.82	66.67	-	33.33	-
caloriecount	3	87.49	98.58	99.71	66.67	-	33.33	-
celwars2009	2	30.72	30.72	30.72	-	-	-	-
checkstyle	6	91.26	95.33	96.97	66.67	-	33.33	-
classviewer	3	67.92	74.86	74.76	100.00	-	-	-
commons-cli	2	93.26	99.07	99.39	100.00	-	50.00	-
commons-codec	1	64.98	93.27	96.55	100.00	-	100.00	-
commons-collections	2	100.00	100.00	100.00	-	-	-	-
commons-lang	14	76.37	96.37	96.74	85.71	-	28.57	7.14
commons-math	21	70.02	72.46	77.72	42.86	-	14.29	-
commons-primitives	1	92.41	92.41	92.41	-	-	-	-
compiler	9	41.53	48.22	51.96	66.67	-	22.22	-
corina	1	51.38	52.73	54.08	-	-	-	-
db-everywhere	2	38.96	44.35	46.75	50.00	-	50.00	-
dcparseargs	1	98.90	100.00	100.00	100.00	-	-	-
diebierse	2	93.80	95.41	94.20	50.00	-	-	-
diffi	2	94.07	96.51	96.21	50.00	-	-	-
dsachat	2	57.13	68.11	70.74	100.00	-	-	-
dvd-homevideo	3	21.53	21.65	21.65	33.33	-	-	-
ext4j	2	96.27	96.32	96.41	-	-	-	-
feudalismgame	3	26.74	26.74	26.74	-	-	-	-
fim1	2	0.00	39.41	39.41	50.00	-	-	-
firebird	3	68.43	78.95	80.27	100.00	-	33.33	-
fixsuite	2	67.00	71.89	70.96	50.00	-	-	-
follow	2	78.68	79.34	77.13	-	-	-	-
fps370	2	8.65	8.65	8.65	-	-	-	-
freemind	2	69.23	76.77	80.93	100.00	-	50.00	-
gangup	1	61.80	62.90	62.21	-	-	-	-
geo-google	2	72.37	84.14	83.15	50.00	-	-	-
gfarcegestionfa	1	93.59	95.97	95.34	100.00	-	-	-
glengineer	2	88.58	97.10	97.01	50.00	-	-	-
gsftp	2	46.87	50.49	51.05	50.00	-	-	-
guava	10	82.31	87.38	86.29	30.00	-	-	-
heal	2	83.94	87.36	87.75	50.00	-	-	-
hft-bomberman	2	52.94	57.46	58.50	50.00	-	-	-
hibernate	1	100.00	100.00	100.00	-	-	-	-
httpanalyzer	2	88.76	88.76	88.76	-	-	-	-
ifx-framework	1	83.42	83.03	82.55	-	-	-	-
imsmart	1	100.00	100.00	100.00	-	-	-	-
inspirento	2	94.68	97.95	98.01	50.00	-	-	-
io-project	1	98.40	99.86	99.93	100.00	-	-	-
ipcalculator	2	93.38	95.08	94.80	50.00	50.00	-	-
javabullboard	2	74.99	79.19	79.25	100.00	-	-	-
javaml	7	88.36	88.35	88.34	-	-	-	-
javathena	4	40.89	47.30	49.98	75.00	25.00	25.00	-
javaviewcontrol	3	55.58	61.27	66.82	100.00	-	33.33	-
javex	1	89.12	93.98	93.88	100.00	-	-	-
jaw-br	2	51.16	51.16	51.16	-	-	-	-
jclo	1	79.83	83.67	80.19	100.00	-	-	-
jcvi-javacommon	2	99.44	99.96	100.00	50.00	-	-	-
jdbacl	2	78.80	87.68	88.06	100.00	-	50.00	-
jdom	5	94.40	96.11	96.30	40.00	-	20.00	-
jfree-chart	12	68.25	81.82	83.96	75.00	-	-	-
jgaap	1	94.64	97.39	97.46	100.00	-	-	-
jhandballmoves	2	27.61	27.61	27.61	-	-	-	-
jiggler	4	78.85	90.08	91.42	50.00	-	-	-
jipa	2	70.55	94.76	94.36	50.00	-	-	-
jiprof	4	63.58	79.82	81.21	100.00	-	-	-
jmca	4	36.36	58.90	59.33	75.00	-	-	-
joda	13	80.75	89.95	90.49	92.31	7.69	7.69	15.38
jopenchart	2	93.65	98.53	99.50	50.00	-	-	-
jsci	4	82.67	94.15	94.51	100.00	-	25.00	25.00
jsecurity	3	79.15	85.89	86.71	66.67	-	33.33	-
jtailgui	2	76.22	56.25	67.90	-	-	-	-
lagoon	2	41.97	43.59	43.70	50.00	-	50.00	50.00
lavalamp	1	100.00	100.00	100.00	-	-	-	-
lhamacaw	2	42.14	35.45	35.33	-	-	-	-
liferay	2	91.89	94.36	96.19	50.00	-	-	-
lilith	1	99.58	100.00	100.00	-	-	-	-
lotus	2	88.11	88.11	88.11	-	-	-	-
mygrid	2	97.48	98.23	98.23	50.00	-	-	-
netweaver	2	98.69	98.57	99.06	-	-	-	-
newzgrabber	3	27.18	24.94	27.02	66.67	33.33	33.33	-
noen	2	91.38	96.05	95.28	100.00	-	-	-
nutzenportfolio	2	60.63	53.37	53.91	-	-	-	-
objectexplorer	2	52.84	52.84	52.84	-	-	-	-
omjstate	1	100.00	99.71	99.86	-	-	-	-
openhre	2	99.93	100.00	100.00	50.00	-	-	-
openjms	2	91.60	92.95	93.27	50.00	-	-	-
pdfsam	2	70.61	73.31	73.31	50.00	-	-	-
petsoar	2	85.31	84.11	83.76	-	50.00	-	-
quickserver	4	56.86	66.85	69.66	50.00	-	50.00	-
rif	1	100.00	100.00	100.00	-	-	-	-
saxpath	2	88.12	93.22	94.17	50.00	-	-	-
schemaspy	2	48.00	57.43	57.03	50.00	-	-	-
scribe	6	100.00	100.00	99.71	-	-	-	-
sfmis	1	100.00	100.00	100.00	-	-	-	-
shop	4	56.08	65.87	66.09	50.00	-	-	-
squirrel-sql	2	174.73	180.40	173.13	-	-	-	-
sugar	2	88.80	92.95	94.55	50.00	-	-	-
summa	4	39.71	43.48	45.15	25.00	-	25.00	25.00
sweethome3d	4	35.94	47.53	46.77	75.00	-	-	-
tartarus	3	65.03	77.99	79.03	100.00	-	66.67	-
templateit	2	65.29	64.14	65.65	-	-	-	-
trans-locator	2	54.72	54.72	54.72	-	-	-	-
trove	10	84.71	91.72	92.44	60.00	-	30.00	-
tullibee	2	100.00	100.00	100.00	-	-	-	-
twfbplayer	2	91.94	94.08	94.96	50.00	-	-	-
twitter4j	7	77.87	95.73	96.79	71.43	-	14.29	-
vuze	2	8.98	20.80	20.97	100.00	-	-	-
water-simulator	2	31.01	45.35	45.09	100.00	-	-	50.00
weka	4	61.63	75.72	77.99	75.00	-	50.00	-
wheelwebtool	4	79.39	88.81	89.61	75.00	-	-	-
wikipedia	4	71.74	75.63	74.92	25.00	25.00	-	-
xbus	2	67.62	58.97	60.71	-	-	-	-
xisemele	1	82.24	88.49	79.61	-	-	-	100.00
xmlenc	2	72.90	76.73	77.16	100.00	-	50.00	-
Mean over all projects		81.37	92.48	92.99				
No. cases DynaMOSA significantly better than WS						**173 (51.03%)**		
No. cases DynaMOSA significantly worse than WS						6 (1.77%)		
No. cases DynaMOSA significantly better than MOSA						**40 (11.80%)**		
No. cases DynaMOSA significantly worse than MOSA						8 (2.36%)		

WS covers 883 statements (14%) on average against 6,042 statements (96%) covered by DynaMOSA on average using the same search budget. Once again, particularly high

TABLE 6
Efficiency of DynaMOSA and WS for statement coverage. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	N. Classes	Efficiency		$\hat{A}_{12}$ Statistics	
		DynaMOSA	WS	%> 0.50	%< 0.50
a4j	2	54.92	54.34	50.00	-
apbsmem	2	71.41	71.40	50.00	-
at-robots2-j	1	74.89	73.38	100.00	-
beanbin	2	84.54	83.85	100.00	-
caloriecount	1	100.00	98.95	100.00	-
checkstyle	2	94.57	94.57	-	50.00
commons-lang	2	96.24	96.22	100.00	-
commons-math	12	76.93	75.37	58.33	-
compiler	3	34.15	29.40	33.33	-
dvd-homevideo	2	19.19	19.20	-	50.00
ext4j	2	96.32	95.99	50.00	-
fixsuite	1	50.04	49.75	100.00	-
gangup	1	38.80	36.51	100.00	-
guava	7	87.89	88.54	14.29	14.29
hft-bomberman	1	91.77	74.36	100.00	-
ifx-framework	1	69.17	67.55	100.00	-
javaml	7	88.33	88.24	71.43	-
jdom	3	98.97	98.41	66.67	-
jhandballmoves	2	23.92	23.74	50.00	-
jiggler	2	91.24	81.31	50.00	-
jmca	1	13.16	11.97	100.00	-
jsecurity	1	67.15	67.12	100.00	-
lagoon	1	23.83	23.54	100.00	-
liferay	1	100.00	99.88	100.00	-
lilith	1	100.00	95.79	100.00	-
lotus	2	86.83	88.11	-	50.00
netweaver	2	96.43	95.74	100.00	-
nutzenportfolio	2	50.16	51.73	-	50.00
objectexplorer	2	52.84	49.25	50.00	-
quickserver	2	97.49	96.92	50.00	-
scribe	6	99.64	99.18	16.67	-
shop	2	54.74	54.27	50.00	-
squirrel-sql	2	67.04	63.11	100.00	-
summa	3	33.87	32.08	33.33	-
trove	4	84.75	83.80	75.00	-
wikipedia	2	89.16	88.67	50.00	-
Mean		71.09	69.53		
No. cases DynaMOSA significantly better than WS					49 (30.63%)
No. cases DynaMOSA significantly worse than WS					6 (3.75%)

differences are detected for larger classes. Indeed, for the largest class in our sample, which is `JavaParser` from project *jmca*, WS covers (on average) 2,415 statements while DynaMOSA covers 8,472 statements (+6,057 statements) using the same search budget.

Only in five classes WS is significantly better than DynaMOSA, e.g., for `CycleHandler` from project *wikipedia* and `WhoIS` from project *ipcalculator*. For these classes, the average difference (decrement) in statement coverage is of 2%. As in the case of branch coverage, we notice that by disabling the test case *size* as secondary preference criterion, DynaMOSA achieves equal or higher statement coverage for these two classes as well.

For the 160 classes with no statistically significant differences in terms of *effectiveness* (statement coverage) between WS and DynaMOSA, we compare the *efficiency* measured using our AUC metrics. The results of this comparison are summarised in Table 6. We observe that in 49 classes out of 160 (31%), DynaMOSA yielded statistically significant higher AUC scores, meaning that it is significantly able to reach higher statement coverage than WS for search time lower than 5 minutes. The average improvement for the AUC metric is 3%. The minimum improvement, 0.1%, occurs for class `FastDateFormat` (from project *apache commons lang*), and the maximum, 21%, for `ForwardingObserver` (from project *hft-bomberman*). Vice versa, WS turned out to be more efficient than DynaMOSA in 6 classes out of 160 with an average difference (decrement) in terms of AUC scores are very small, being $< 2\%$ on average.

TABLE 7
Efficiency of DynaMOSA and MOSA for statement coverage. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	N. Classes	Efficiency		$\hat{A}_{12}$ Statistics	
		DynaMOSA	MOSA	%> 0.50	%< 0.50
at-robots2-j	2	72.45	72.14	50.00	-
checkstyle	4	93.82	91.90	-	25.00
commons-cli	1	100.00	99.99	100.00	-
commons-collections	3	95.34	95.27	66.67	-
commons-lang	9	93.34	88.70	55.56	-
commons-math	17	68.69	65.05	47.06	-
compiler	7	34.39	32.10	42.86	-
dvd-homevideo	3	21.44	21.43	33.33	-
firebird	2	97.26	96.88	50.00	-
guava	8	85.80	85.69	12.50	-
ifx-framework	1	69.17	68.26	100.00	-
javaml	7	88.33	88.33	28.57	-
javathena	3	45.19	46.02	33.33	-
jdom	4	98.64	98.61	50.00	-
jfree-chart	12	75.19	72.00	25.00	16.67
joda	10	81.29	78.54	60.00	-
jopenchart	2	83.09	81.85	-	50.00
saxpath	2	93.04	92.38	50.00	-
shop	4	62.17	61.09	25.00	-
squirrel-sql	2	67.04	66.25	50.00	-
tartarus	1	73.89	72.48	100.00	-
twitter4j	6	94.07	93.35	16.67	-
weka	2	67.34	64.54	50.00	-
wheelwebtool	4	87.70	86.32	25.00	-
wikipedia	4	62.00	61.94	25.00	-
xmlenc	1	88.05	91.16	-	100.00
Mean		76.87	75.86		
No. cases DynaMOSA significantly better than MOSA					**46 (15.97%)**
No. cases DynaMOSA significantly worse than MOSA					5 (1.73%)

Regarding the comparison between DynaMOSA and MOSA, results are reported in Table 5. From the results, we observe that in 40 classes (12%) DynaMOSA leads to statistically significantly higher statement coverage with effect size $\hat{A}_{12} \gg 0.5$. In these cases, improvements range between 0.10% and 88% with an average increment of 5%. The class with the largest improvement with respect to MOSA is `SchurTransformer` selected from the *apache commons math*, for which DynaMOSA covers (on average) 1,250 statements (92%) while MOSA covers 45 statements (3%). On the other hand, MOSA is significantly better in 8 classes (2%) in our study. In such cases, the average difference in statement coverage is 2%, with a minimum of 0.07% for class `SpecialMath` from *jsci* and a maximum difference of 8% for class `OperationsHelperImpl` from *xisemele*.

The results of the comparison between DynaMOSA and MOSA in terms of *efficiency* are reported in Table 7. Among 291 classes with no statistically significant difference in statement coverage, DynaMOSA has statistically higher AUC scores in 46 classes (16%). On the other hand, in 5 classes (2%) MOSA required significantly less search time than DynaMOSA to reach the same final coverage. On the remaining 240 classes, no statistically significant difference is observed, meaning that MOSA and DynaMOSA reached the maximum statement coverage consuming similar search times.

To provide a deeper view on *efficiency*, Figure 5 plots the average statement coverage achieved by DynaMOSA, MOSA and WS over time for class `SmallCharMatcher`, a class for which we did not find any statistically significant difference in neither effectiveness nor efficiency for the three approaches. From Figure 5, we can notice that at the beginning of the search (first 15 seconds) DynaMOSA quickly reaches 98% of statement coverage against 79% reached by WS and 71% by MOSA. Between 16 and 21 seconds the three approaches have the same statement coverage (notice that we pick the first coverage point for the AUC metric

Fig. 5. Statement coverage over time for class `SmallCharMatcher` from project *guava*

at 20s); while between 22 and 58 seconds both DynaMOSA and MOSA outperform WS. Furthermore, on this class WS outperforms MOSA for a while within the first 15 seconds of search. Then, after 56 seconds (on average) all the three approaches reach 99% of statement coverage. However, if a shorter time were allocated for the search (< 20s), we notice that DynaMOSA would reach higher coverage more quickly than the other two alternative approaches. We obtained consistent results on the other classes in our study for which we did not observe significant difference in terms of final statement coverage.

In summary, we can conclude that DynaMOSA achieves **higher or equal statement coverage** as compared to both WS and MOSA. On classes with no improvement in statement coverage, DynaMOSA converges more quickly to the final coverage, especially when limited time is given to the search.

5.3 RQ3: *How does DynaMOSA perform compared to alternative approaches on strong mutation coverage?*

Results on the strong mutation coverage are reported in Table 8, grouped by software projects. For 34 classes, we could not obtain the mutation coverage score because of internal EvoSuite crashes. Therefore, in the following we refer to 312 classes only.

Results show that DynaMOSA statistically significantly outperforms WSA in 85 out of 312 classes (27%), which corresponds to 41% of the projects in our study. The average improvement, over classes where significant differences were observed, in mutation score achieved by DynaMOSA is 11%, with minimum and maximum improvement of 0.30% and 69% respectively. The class with largest improvement is `TableMeta`, from project *schemaspy*, for which the total number strong mutants to kill is equal to 21. On this class, DynaMOSA kills on average 15 mutants (71%) while WSA kills only one mutant on average. This result shows that DynaMOSA can achieve high mutation scores even on small classes (i.e., classes with a small number of mutants). WSA turned out to be statistically significantly better on 22 classes (7%), corresponding to the 16% of the projects in our sample. However, in this case the average difference in strong mutation coverage is just 5%.

TABLE 8
Mean **strong mutation** coverage achieved for each project. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	Classes	Statement Coverage			$\hat{A}_{12}$ Statistics			
					DynaMOSA vs WSA		DynaMOSA vs MOSA	
		WSA	MOSA	Dyna-MOSA	%> 0.50	%< 0.50	%> 0.50	%< 0.50
a4j	2	28.85	30.44	30.34	50.00	-	-	-
apbsmem	1	0.00	0.00	0.00	-	-	-	-
asphodel	1	0.00	0.00	0.00	-	-	-	-
at-robots2-j	2	5.79	7.09	15.24	100.00	-	50.00	-
battlecry	2	3.38	0.93	0.64	-	50.00	-	-
beanbin	2	35.95	33.67	33.85	-	50.00	-	-
biblestudy	2	68.07	78.40	70.58	50.00	-	50.00	50.00
biff	1	3.38	0.00	0.00	-	100.00	-	-
byuic	3	14.51	17.02	21.86	66.67	-	-	-
caloriecount	3	69.96	64.44	55.59	-	-	-	-
celwars2009	2	1.28	1.28	1.28	-	-	-	-
checkstyle	5	16.51	18.98	17.78	40.00	-	20.00	20.00
classviewer	3	11.17	16.92	18.44	33.33	-	-	-
commons-cli	2	50.50	37.32	50.77	-	-	50.00	-
commons-collections	2	13.49	14.62	17.54	50.00	-	-	-
commons-lang	14	34.66	30.97	41.92	57.14	-	64.29	-
commons-math	21	22.02	22.50	25.50	42.86	4.76	52.38	4.76
commons-primitives	1	28.03	28.03	30.80	-	-	-	-
compiler	9	9.58	9.25	10.32	22.22	11.11	33.33	-
corina	1	21.05	21.38	21.71	-	-	-	-
db-everywhere	2	8.93	10.27	11.17	-	-	-	-
dcparseargs	1	25.23	26.82	32.95	-	-	-	-
diebierse	1	7.69	5.13	5.13	-	-	-	-
diffi	2	14.90	15.81	16.51	-	-	-	-
dsachat	2	1.96	2.98	2.66	-	-	-	-
dvd-homevideo	3	39.96	34.04	39.45	-	-	100.00	-
ext4j	2	32.53	30.47	30.85	-	50.00	-	-
feudalismgame	3	4.06	3.61	3.98	-	-	33.33	-
fim1	2	0.61	0.61	0.61	-	-	-	-
firebird	2	54.97	49.72	57.42	-	-	100.00	-
fixsuite	2	0.46	0.80	0.46	-	-	-	-
follow	2	16.17	14.28	15.19	-	-	-	-
fps370	2	0.00	0.00	0.00	-	-	-	-
freemind	2	4.17	7.74	4.82	50.00	-	-	-
geo-google	2	59.73	70.98	72.23	-	-	-	-
gfarcegestionfa	1	7.81	6.05	6.73	-	-	-	-
glengineer	2	25.51	41.33	42.46	50.00	-	-	-
gsftp	2	3.01	4.50	4.23	50.00	-	-	-
guava	10	25.91	27.23	28.30	20.00	-	20.00	-
heal	2	60.72	56.26	59.41	-	-	50.00	-
hft-bomberman	2	29.76	38.50	33.33	-	-	-	50.00
hibernate	1	0.00	0.00	0.00	-	-	-	-
httpanalyzer	1	0.00	0.00	0.00	-	-	-	-
ifx-framework	1	58.63	56.12	56.83	-	-	-	-
imsmart	1	37.93	37.93	37.93	-	-	-	-
inspirento	2	18.81	16.77	14.06	-	-	-	-
io-project	1	53.12	54.17	55.21	-	-	-	-
ipcalculator	2	24.61	20.36	16.99	-	100.00	-	-
javabullboard	2	24.15	24.07	24.26	-	-	-	-
javaml	7	49.74	50.51	50.83	-	-	-	-
javathena	4	17.54	20.68	19.64	50.00	-	-	-
javaviewcontrol	3	30.35	29.76	31.21	33.33	66.67	-	-
javex	1	92.79	62.74	91.35	-	-	100.00	-
jaw-br	2	0.00	0.00	0.00	-	-	-	-
jclo	1	16.45	22.19	26.63	100.00	-	-	-
jcvi-javacommon	2	76.78	49.05	82.86	50.00	-	100.00	-
jdbacl	2	35.55	39.68	38.94	50.00	-	-	50.00
jdom	5	45.64	42.50	46.63	40.00	-	40.00	-
jfree-chart	12	35.02	29.62	37.62	33.33	-	66.67	-
jgaap	1	17.80	13.56	22.03	-	-	-	-
jhandballmoves	2	6.01	7.36	7.36	50.00	-	-	-
jiggler	4	8.54	8.58	11.01	25.00	-	25.00	-
jipa	2	58.40	58.85	59.54	50.00	-	-	-
jiprof	4	22.98	25.75	34.68	75.00	-	25.00	-
jmca	2	7.09	5.88	4.80	-	50.00	-	-
joda	13	34.54	29.03	38.54	46.15	-	84.62	-
jopenchart	2	23.10	26.52	27.72	50.00	-	-	-
jsci	4	24.39	20.12	25.87	50.00	25.00	25.00	-
jsecurity	3	42.20	39.70	42.05	-	-	-	-
jtailgui	2	28.34	23.81	25.00	-	-	-	-
lagoon	2	10.73	8.75	9.06	50.00	50.00	-	-
lavalamp	1	88.52	85.25	88.52	-	-	100.00	-
lhamacaw	2	0.14	0.29	0.29	-	-	-	-
liferay	2	44.76	46.44	75.22	50.00	-	50.00	-
lilith	1	93.72	85.79	88.52	-	-	-	-
lotus	2	67.65	68.14	69.61	-	-	-	-
mygrid	2	94.29	75.00	83.57	-	100.00	50.00	-
newzgrabber	3	5.57	14.17	16.79	66.67	-	33.33	-
noen	2	24.44	27.87	28.75	50.00	-	-	-
nutzenportfolio	2	43.16	2.56	43.80	-	-	100.00	-
objectexplorer	2	43.75	43.75	43.75	-	-	-	-
omjstate	1	76.47	79.41	79.41	-	-	-	-
openhre	2	62.07	60.70	63.70	-	-	50.00	-
openjms	2	45.08	40.08	45.63	50.00	-	50.00	-
pdfsam	2	35.54	27.65	27.21	-	50.00	-	-
petsoar	2	41.09	41.58	41.58	-	-	-	-
quickserver	4	59.52	63.62	64.57	50.00	-	-	-
rif	1	46.43	46.43	46.43	-	-	-	-
saxpath	2	65.72	67.77	72.13	50.00	-	50.00	-
schemaspy	2	5.09	37.24	39.62	50.00	-	50.00	-
scribe	6	42.06	42.53	59.64	50.00	-	33.33	-
sfmis	1	49.30	48.13	48.59	-	-	-	-
shop	3	17.17	19.95	19.69	-	-	-	-
sugar	2	45.86	45.86	46.04	-	-	50.00	-
summa	4	10.33	12.07	10.51	-	25.00	-	-
sweethome3d	4	1.05	1.38	1.08	25.00	-	-	-
tartarus	3	10.74	8.74	7.10	-	33.33	-	33.33
templateit	2	50.00	53.54	53.54	50.00	-	-	-
trans-locator	2	7.48	6.31	6.31	-	-	-	-
trove	7	33.65	34.18	38.50	14.29	14.29	14.29	-
tullibee	2	97.14	93.60	97.57	-	-	50.00	-
twfbplayer	2	22.36	21.68	21.68	-	100.00	-	-
twitter4j	4	59.61	56.23	66.31	50.00	25.00	50.00	-
water-simulator	1	2.98	4.37	5.91	100.00	-	-	-
weka	3	10.79	10.56	9.28	-	-	-	-
wheelwebtool	4	18.55	21.58	24.98	50.00	-	-	-
wikipedia	1	78.85	76.92	76.92	-	-	100.00	-
xbus	2	8.48	7.88	5.47	-	-	-	-
xisemele	1	21.43	22.86	22.86	100.00	-	-	-
xmlenc	2	17.79	20.61	27.39	100.00	-	50.00	-
Mean over all projects		20.51	19.20	22.74				
No. cases DynaMOSA significantly better than WSA					**85 (27.24%)**			
No. cases DynaMOSA significantly worse than WSA					22 (7.05%)			
No. cases DynaMOSA significantly better than MOSA					**82 (26.28%)**			
No. cases DynaMOSA significantly worse than MOSA					6 (1.92%)			

For the remaining 205 classes in our study, no statistically significant difference was found for mutation coverage. For these classes, we compared DynaMOSA and WSA in

TABLE 9
Efficiency of DynaMOSA and WSA for strong mutation. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	N. Classes	Efficiency		$\hat{A}_{12}$ Statistics	
		DynaMOSA	WSA	$\% > 0.50$	$\% < 0.50$
a4j	1	53.33	32.23	100.00	-
byuic	1	37.29	33.90	100.00	-
caloriecount	3	58.87	32.67	33.33	33.33
checkstyle	3	5.01	4.28	33.33	-
classviewer	2	1.62	2.22	-	50.00
commons-cli	2	46.03	41.18	100.00	-
commons-lang	6	27.33	22.36	33.33	16.67
commons-math	11	13.93	12.38	27.27	-
compiler	6	3.44	2.95	16.67	-
diffi	2	18.89	10.98	50.00	-
dsachat	2	3.01	0.20	50.00	-
dvd-homevideo	3	36.47	21.75	100.00	-
feudalismgame	3	3.91	2.03	33.33	-
fim1	2	0.33	0.61	-	50.00
firebird	2	53.35	49.17	100.00	-
fixsuite	2	0.43	1.11	-	50.00
follow	2	15.66	12.79	50.00	-
geo-google	2	60.61	55.54	50.00	-
gfarcegestionfa	1	5.15	3.67	100.00	-
guava	8	27.32	22.38	37.50	12.50
heal	2	56.47	54.63	50.00	-
ifx-framework	1	58.71	52.29	100.00	-
io-project	1	51.29	42.18	100.00	-
javabullboard	2	22.70	19.47	50.00	-
javaml	7	46.87	46.47	14.29	-
jdom	3	36.65	40.00	-	33.33
jfree-chart	8	35.62	35.42	25.00	-
jhandballmoves	1	1.87	2.58	-	100.00
jiggler	3	9.95	5.25	66.67	-
jmca	1	3.32	0.53	100.00	-
jopenchart	1	24.55	8.92	100.00	-
jsecurity	3	38.86	34.96	66.67	-
jtailgui	2	24.00	9.38	50.00	-
lavalamp	1	87.75	86.07	100.00	-
lhamacaw	2	0.79	0.28	50.00	-
lotus	2	65.48	60.60	50.00	-
newzgrabber	1	13.64	3.73	100.00	-
nutzenportfolio	2	42.17	28.03	50.00	-
omjstate	1	76.85	51.18	100.00	-
openhre	2	55.78	57.34	-	50.00
pdfsam	1	0.90	0.00	100.00	-
petsoar	2	40.74	29.08	100.00	-
quickserver	2	60.62	53.34	100.00	-
rif	1	45.51	45.89	-	100.00
saxpath	1	90.62	90.37	100.00	-
schemaspy	1	7.78	6.72	100.00	-
scribe	3	54.17	48.96	33.33	-
sfmis	1	47.50	45.95	100.00	-
shop	3	15.36	12.50	33.33	-
sugar	2	46.33	43.75	50.00	-
sweethome3d	3	0.80	0.77	33.33	-
templateit	1	99.99	99.00	100.00	-
tullibee	2	91.88	81.77	100.00	-
weka	3	7.66	9.29	33.33	33.33
wheelwebtool	2	11.79	7.57	50.00	-
xbus	2	4.65	0.00	50.00	-
Mean		31.48	22.37		
No. cases DynaMOSA significantly better than WSA					**64 (31.22%)**
No. cases DynaMOSA significantly worse than WSA					11 (5.36%)

terms of efficiency, using the AUC metric. The results of this comparison are summarised in Table 9, grouped by project. We observe that in 64 classes (31%) DynaMOSA reaches a statistically significant higher AUC score than WSA, with an average improvement of 8%, and minimum improvement of 1% for `Region` (*templateit* project) and maximum of 78% for `SimpleKeyListenerHelper` (*caloriecount* project). Vice versa, WSA is better than DynaMOSA on 11 classes (5%), with an average AUC difference of 6%. It is important to notice that on these 11 cases, the difference in efficiency is mainly due to the test prioritisation performed by WSA: WSA prioritises the test cases in a test suite according to their execution time before re-running them on each infected mutant. Hence, mutants that are killed by quicker tests in a test suite do not need to be evaluated again on the other more expensive tests in the same test suite. This strategy can be particularly relevant in the case of the strong mutation coverage, as every mutant requires the re-execution of the test suite. Differently, in DynaMOSA (and MOSA) all test cases (offsprings) are evaluated, without applying any prioritisation strategy. Hence, there is room for improvement of DynaMOSA when using it with the strong mutation coverage. In fact, we could also prioritise the offspring test cases by execution time.

TABLE 10
Efficiency of DynaMOSA and MOSA for strong mutation. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	N. Classes	Efficiency		$\hat{A}_{12}$ Statistics	
		DynaMOSA	MOSA	$\% > 0.50$	$\% < 0.50$
classviewer	3	14.18	11.45	33.33	-
commons-collections	2	15.60	14.11	50.00	50.00
commons-lang	5	27.13	26.40	40.00	-
commons-math	9	22.01	21.76	-	11.11
compiler	6	11.27	10.06	16.67	-
fixsuite	2	0.43	0.82	-	50.00
geo-google	2	60.61	65.18	-	50.00
glengineer	2	27.63	33.59	-	50.00
guava	8	29.34	28.40	-	12.50
ipcalculator	2	14.97	18.73	-	50.00
jfree-chart	4	45.04	42.90	25.00	-
jipa	2	58.25	57.44	50.00	-
jmca	2	2.93	2.41	50.00	-
joda	2	27.09	23.82	50.00	-
liferay	1	45.52	40.75	100.00	-
lilith	1	69.67	61.94	100.00	-
lotus	2	65.48	60.18	50.00	-
omjstate	1	76.85	74.51	100.00	-
quickserver	4	62.51	61.36	50.00	-
rif	1	45.51	44.35	100.00	-
saxpath	1	90.62	90.55	100.00	-
tullibee	1	84.65	71.80	100.00	-
twfbplayer	2	20.31	21.01	-	50.00
Mean		39.99	38.53		
No. cases DynaMOSA significantly better than MOSA					18 (8.04%)
No. cases DynaMOSA significantly worse than MOSA					8 (3.57%)

Let us now consider the comparison between DynaMOSA and MOSA. Results are similar to those obtained for branch and statement coverage. As reported in Table 8, DynaMOSA achieved significantly higher mutation score in 82 classes (26%) against 6 classes (2%) on which MOSA yields statistically significantly higher scores. In those classes where DynaMOSA turned out to be better, the improvements in terms of mutation score range between 0.51% and 79%, being 12% on average. In those few cases (2%) where MOSA yields higher mutation score, the average increase is 7%, with minimum of 1% for class `BrentOptimizer` (from *apache commons math*) and maximum of 23% for class `Verse` (from project *biblestudy*).

The largest improvement achieved by DynaMOSA is 79%, and it is obtained for class `AuswertungGrafik` from project *nutzenportfolio*, whose total number of mutants is 131. On this class, DynaMOSA is able to kill 103 mutants compared to zero mutants killed by MOSA on average. The explanation for such large difference is due to (i) the number of objectives evaluated at each generation, and (ii) the heavy evaluation cost of each mutant for such a class. Indeed, with MOSA all 131 mutants are evaluated since the first generation, even if not all mutants are within control of covered branches. Differently, DynaMOSA exploits the control dependencies between test targets and selects less than 100 mutants in the first generation, i.e., only mutants under root branches (and their dependencies, if covered) in the code. The remaining mutants are dynamically added as further search objectives during later generations according to the control dependencies of newly covered branches.

In terms of *efficiency*, DynaMOSA and MOSA are statistically equivalent in the majority of the classes where there is no statistically significant difference in the number of killed mutants. Indeed, as reported in Table 10, in only 18 classes out of 224 DynaMOSA yielded a significantly higher AUC

scores (average coverage over search time) as compared to MOSA; vice versa in only 8 classes MOSA produced significantly higher AUC scores than DynaMOSA.

> In summary, DynaMOSA kills **more or the same number of mutants** as compared to both WSA and MOSA. Moreover, for classes with no improvement in mutation score DynaMOSA converges more quickly.

5.4 RQ4: *How do preference criterion and Pareto dominance affect the effectiveness of DynaMOSA?*

Our approach is designed to improve the selection pressure by giving higher priority to the test cases that are close to reach uncovered targets. Specifically, selection is based on a new sorting algorithm (Algorithm 3), which combines two key ingredients: (i) traditional Pareto-based ranking and (ii) our novel *preference criterion*. Table 11 reports the results achieved on branch coverage by three variants of DynaMOSA: (i) a first variant, namely $\text{DynaMOSA}_{\text{Rank}}$, which uses only Pareto-based ranking; (ii) a second variant, namely $\text{DynaMOSA}_{\text{Pref}}$, which uses only our *preference criterion*; and (iii) full DynaMOSA, which combines both Pareto-based ranking and *preference criterion* ($\text{DynaMOSA}_{\text{Full}}$). In all three variants, we use an *archive* to keep track of test cases as soon as they reach uncovered targets (see Algorithm 5). Therefore, any difference in the final branch coverage can be interpreted as the effect of the different selection strategies used by the DynaMOSA variants.

From the comparison, we observe that $\text{DynaMOSA}_{\text{Full}}$ outperforms $\text{DynaMOSA}_{\text{Rank}}$ in 172 classes out of 335 (51%) with an average difference in branch coverage of 14% and maximum difference of 91% (for class `TableMeta` from the project *schemaspy*). In 46% of the projects there is at least one class for which $\text{DynaMOSA}_{\text{Full}}$ outperforms $\text{DynaMOSA}_{\text{Rank}}$. Since $\text{DynaMOSA}_{\text{Rank}}$ uses only Pareto-based ranking, the achieved results show that the usage of the *preference criterion* has a strong impact on the effectiveness of our many-objective algorithm.

There are only four classes in which $\text{DynaMOSA}_{\text{Rank}}$ has significantly higher branch coverage scores, namely `ScopeUtils` from *checkstyle*, `CacheBuilderSpec` from `guava`, `InternalChatFrame` from *dsachat*, and `ReflectionSearch` from *beanbin*. However, for these classes we note that the difference is small (<5% on average) especially if compared to the average improvement that can be obtained by using our preference criterion, i.e., by using $\text{DynaMOSA}_{\text{Full}}$. The explanation for these few exceptional classes is that the statistically significant differences are due to the usage of test case length (*size*) as secondary selection criterion in our *preference criterion*. Indeed, re-running $\text{DynaMOSA}_{\text{Full}}$ by disabling the test size criterion, the average branch coverage (over 10 independent runs) for `ReflectionSearch` is 73.20%, becoming statistically indistinguishable from $\text{DynaMOSA}_{\text{Rank}}$ (p-value$= 0.92$ and $\hat{A}_{12} = 0.52$).

The results of the comparison between $\text{Dyna-MOSA}_{\text{Full}}$ and $\text{DynaMOSA}_{\text{Pref}}$ are mixed: in the majority of classes (302 out of 335 classes) there is no statistically significant difference between these two variants of DynaMOSA. This means that the *preference criterion* is the most critical ingredient in our many-objective algorithm, since it achieves

TABLE 11
Projects with statistically significant difference in **branch** coverage for three variants of DynaMOSA. We report the percentage of classes with $\hat{A}_{12} \neq 0.50$ where p-value ≤ 0.05

Project Name	Classes	DynaMOSA Variants			$\hat{A}_{12}$ Statistics			
		Simple	Rank	Full	Full vs Simple		Full vs Rank	
					%> 0.50	%< 0.50	%> 0.50	%< 0.50
at-robots2-j	2	79.16	61.03	82.16	50.00	-	100.00	-
battlecry	2	41.15	20.82	30.30	-	-	50.00	-
beanbin	2	77.92	78.40	77.92	-	-	-	50.00
biff	1	30.47	20.24	31.69	-	-	100.00	-
byuic	3	54.54	47.80	54.81	-	-	66.67	-
caloriecount	3	97.22	89.21	98.98	-	-	66.67	-
checkstyle	6	82.07	78.19	86.15	16.67	16.67	66.67	16.67
classviewer	3	61.68	55.84	61.33	-	-	66.67	-
commons-cli	2	96.10	95.93	119.28	100.00	-	100.00	-
commons-collections	3	93.33	92.52	93.53	-	-	33.33	-
commons-lang	11	94.13	85.80	95.34	9.09	-	72.73	-
commons-math	21	70.90	68.57	75.57	19.05	-	71.43	-
compiler	9	37.61	29.31	41.14	11.11	11.11	88.89	-
db-everywhere	2	44.31	38.99	47.21	-	-	50.00	-
dcparseargs	1	97.50	95.23	97.50	-	-	100.00	-
diebierse	2	78.26	76.85	94.44	50.00	-	50.00	-
diffi	2	94.49	92.18	96.29	50.00	-	50.00	-
dsachat	2	38.64	40.29	38.91	-	-	50.00	50.00
dvd-homevideo	3	13.03	12.23	13.03	-	-	66.67	-
ext4j	2	97.20	95.76	97.16	-	-	50.00	-
firebird	3	76.04	68.34	75.65	-	33.33	100.00	-
fixsuite	2	59.43	55.69	61.80	50.00	-	100.00	-
follow	2	69.31	52.69	69.82	-	-	100.00	-
freemind	2	74.55	60.48	84.61	50.00	-	100.00	-
geo-google	2	85.54	90.19	96.58	50.00	-	50.00	-
glengineer	2	95.53	88.81	96.07	50.00	-	50.00	-
gsftp	2	38.50	34.14	37.71	-	-	50.00	-
guava	10	74.68	72.32	72.90	10.00	20.00	50.00	10.00
heal	2	93.78	90.14	93.40	-	-	50.00	-
ifx-framework	1	69.25	46.21	69.86	-	-	100.00	-
inspirento	2	86.36	85.28	86.67	-	-	50.00	-
ipcalculator	2	81.06	78.06	83.02	50.00	-	50.00	-
javabullboard	2	80.18	72.30	81.33	-	-	100.00	-
javathena	4	49.15	42.31	49.18	-	-	25.00	-
javaviewcontrol	3	70.98	63.31	71.83	33.33	-	66.67	-
javex	1	88.01	85.65	88.15	-	-	100.00	-
jclo	1	82.30	76.56	83.98	-	-	100.00	-
jcvi-javacommon	2	99.77	98.05	99.84	-	-	50.00	-
jdbacl	2	86.03	82.44	86.04	-	-	100.00	-
jdom	5	84.05	82.63	84.35	-	-	40.00	-
jfree-chart	11	82.33	71.35	84.98	27.27	-	63.64	-
jgaap	1	89.13	83.64	94.35	100.00	-	100.00	-
jiggler	4	86.43	80.22	86.68	-	-	25.00	-
jiprof	4	79.08	70.75	80.21	-	-	100.00	-
jmca	4	52.55	41.12	53.69	25.00	-	100.00	-
joda	13	85.49	76.02	84.81	7.69	15.38	53.85	-
jopenchart	2	97.40	62.87	97.28	-	-	100.00	-
jsci	4	93.08	86.58	92.71	-	25.00	100.00	-
jsecurity	3	72.20	67.77	75.11	33.33	-	66.67	-
jtailgui	2	46.13	31.99	59.95	50.00	-	50.00	-
lagoon	2	22.98	18.07	23.20	-	-	100.00	-
lhamacaw	2	10.37	9.46	10.84	-	-	50.00	-
liferay	1	93.33	82.90	94.95	-	-	100.00	-
lilith	1	100.00	99.05	100.00	-	-	100.00	-
lotus	2	68.66	59.79	70.83	50.00	-	50.00	-
netweaver	2	97.88	94.87	97.62	-	-	100.00	-
newzgrabber	3	21.11	17.69	20.60	-	-	33.33	-
noen	2	96.13	75.71	96.13	-	-	100.00	-
nutzenportfolio	2	43.62	41.08	54.32	50.00	-	50.00	-
openhre	2	100.00	98.83	100.00	-	-	50.00	-
openjms	2	83.65	74.44	84.28	-	-	100.00	-
petsoar	2	79.26	76.98	80.85	-	-	50.00	-
quickserver	4	63.56	57.40	64.19	-	-	50.00	-
saxpath	2	95.06	93.35	95.04	-	-	50.00	-
schemaspy	2	52.51	23.36	67.75	50.00	-	50.00	-
shop	4	58.39	56.27	59.09	25.00	25.00	-	-
sugar	2	87.89	84.40	89.12	-	-	50.00	-
summa	4	37.89	35.53	37.62	-	-	25.00	-
sweethome3d	4	25.01	19.37	50.46	75.00	-	100.00	-
tartarus	3	75.34	68.11	76.51	33.33	-	100.00	-
trans-locator	2	51.16	50.09	51.47	-	-	50.00	-
trove	8	82.11	80.43	82.06	12.50	-	-	-
twfbplayer	2	89.07	82.23	89.26	-	-	100.00	-
twitter4j	7	94.76	88.59	86.59	-	57.14	57.14	42.86
vuze	2	8.17	5.94	9.68	50.00	-	50.00	-
water-simulator	2	15.43	9.57	14.71	-	-	50.00	-
weka	4	69.50	57.70	69.62	-	-	100.00	-
wheelwebtool	4	88.88	80.98	90.30	25.00	-	75.00	-
xbus	2	67.96	47.98	64.40	-	50.00	100.00	-
xisemele	1	55.82	67.00	80.00	100.00	-	100.00	-
xmlenc	2	63.39	62.30	63.50	-	-	100.00	-
Mean over all projects		78.26	70.75	80.85				
No. cases $\text{DynaMOSA}_{\text{Full}}$ significantly better than $\text{DynaMOSA}_{\text{Pref}}$							39 (11.64%)	
No. cases $\text{DynaMOSA}_{\text{Full}}$ significantly worse than $\text{DynaMOSA}_{\text{Pref}}$							14 (4.17%)	
No. cases $\text{DynaMOSA}_{\text{Full}}$ significantly better than $\text{DynaMOSA}_{\text{Rank}}$							172 (51.34%)	
No. cases $\text{DynaMOSA}_{\text{Full}}$ significantly worse than $\text{DynaMOSA}_{\text{Rank}}$							4 (2.08%)	

the same branch coverage (and sometimes higher coverage) even if it is not combined with Pareto-based ranking. In 39 classes (12%) $\text{DynaMOSA}_{\text{Full}}$ outperforms $\text{DynaMOSA}_{\text{Pref}}$, with an average difference in branch coverage of 14% and maximum difference of 85% for class `SchurTransformer` from the *apache commons math*. In the remaining 14 classes (4%), the usage of *preference criterion* alone ($\text{DynaMOSA}_{\text{Pref}}$) leads to statistically higher branch coverage if compared to its combination with the traditional non-dominance ranking ($\text{DynaMOSA}_{\text{Full}}$).

We notice that $\text{DynaMOSA}_{\text{Pref}}$ is significantly better for extremely large classes, i.e., classes with hundreds or thousands of branches. Vice versa, $\text{DynaMOSA}_{\text{Full}}$ achieves better coverage for relatively small/medium size classes. In fact, the mean size of classes for which $\text{DynaMOSA}_{\text{Pref}}$

turned out to be significantly better than DynaMOSA$_{\text{Full}}$ is 247 branches. On the other hand, the mean size of classes for which DynaMOSA$_{\text{Full}}$ outperforms DynaMOSA$_{\text{Pref}}$ is 177.

The explanation for such findings could be that Pareto-based ranking computes the non-dominance ranks using the *Fast-Non-Dominated-Sort* algorithm, whose computational complexity is $O(M^2 \times N)$, where M is the population size and N is the number of uncovered test targets. When the number of targets becomes too large (the order of thousands), DynaMOSA$_{\text{Full}}$ spends too much time in computing the non-dominance ranks rather than in evolving the test cases. Hence, potentially better coverage scores may be achieved by developing an adaptive strategy which enables or disables Pareto-based ranking based on the number of uncovered branches considered as objectives.

> In summary, our preference criterion is a critical component of DynaMOSA, necessary to improve the selective pressure when dealing with many objectives (**RQ**$_4$). Indeed, Pareto-based ranking (even with the archive) is not sufficient to deal with hundreds of objectives, as is the case of test case generation.

6 Additional Analyses

In this section, we analyse qualitatively some examples and discuss *co-factors* that could have played an important role in the performance of the experimented algorithms. Even though these analyses are not directly related to the research questions of the study described in Sections 4-5, they helped us understand the conditions in which DynaMOSA outperforms the alternative algorithms.

6.1 Qualitative analysis

```
public static RealMatrix createRealMatrix(double[][] data)
    throws NullArgumentException,
    DimensionMismatchException, NoDataException {
131  if (data == null ||
132     data[0] == null) {
133         throw new NullArgumentException();
134      }
135  return (data.length * data[0].length <= 4096) ?
136          new Array2DRowRealMatrix(data) :
137                         new BlockRealMatrix(data);
138     }
```

(a) Target branch

```
double[][] doubleArray0 = new double[1][6];
double[] doubleArray1 = new double[6];
...
doubleArray0[0] = doubleArray1;
...
MatrixUtils.createRealMatrix(doubleArray0);
```

(b) TC1 with branch distance $d = 0.9998$

```
ouble[][] doubleArray0 = new double[50][50];
double[] doubleArray1 = new double[50];
...
doubleArray0[0] = doubleArray1;
...
MatrixUtils.createRealMatrix(doubleArray0);
```

(c) TC2 with branch distance $d = 0.9997$

Fig. 6. Example of uncovered branch for *MatrixUtils*

Figure 6 shows an example of a branch covered by DynaMOSA but not by WSA for the class `MatrixUtils` extracted from the *Apache commons math* library, i.e., the false branch of line 137 of method `createRealMatrix`. The related branch condition checks the size of the input matrix `data` and returns an object of class `Array2DRowRealMatrix` or of class `BlockRealMatrix` depending on the outcome of the branch condition. In particular, below 2^{12} elements (i.e., 4096 elements or 64×64 for a square matrix) an `Array2DRowRealMatrix` instance is built since it can store up to 32kB array. Above this threshold a `BlockRealMatrix` instance is built. At the end of the search process, the final test suite obtained by WSA has a whole suite fitness $f = 21.50$. Within the final test suite, the test case closest to cover the considered target is shown in Figure 6-b. It is a test case with branch distance $d = 0.9998$, for the branch under analysis.

This test case executes method `createRealMatrix` giving it as input an `array` with $1 \times 6 = 6$ elements ($<< 4096$). However, by analysing all the test cases generated by WSA during the search process we found that TC1 is not the closest test case to the false branch of line 137 across all generations. For example, at some generation WSA generated a test case TC2 with a lower branch distance $d = 0.9997$ that is reported in Figure 6-c. As we can see, TC2 also executes the lines 131-136, hence being equivalent to TC1 in terms of coverage. However, in TC1 the method `createRealMatrix` is called by using as input an array with $50 \times 50 = 2500$ elements, thus, such a test case is much closer to satisfy the condition `data.length * data[0].length > 4096` that lead to cover the false branch of line 137. However, TC2 was generated within a candidate test suite with a poor whole suite fitness $f = 98.37$, which happened to be the worst candidate test suite in its generation. Thus, in the next generation this test suite is not selected to form the next generation and the promising test case TC2 is lost. By manual investigation we verified that this scenario is quite common, especially for classes with a large number of targets to cover. As we can see from this example, the whole suite fitness is really useful in increasing the global number of covered goals, but when aggregating the branch distances of uncovered branches, the individual contribution of single promising test cases may remain unexploited.

Unlike WSA, DynaMOSA selects the best test *case* (instead of the best test *suite*) within the current population for each uncovered branch. Therefore, in a similar scenario it would place TC2 in the first non-dominated front $\mathbb{F}_0$ according to the proposed *preference criterion*. Thus, generation by generation test case TC2 will remain in front $\mathbb{F}_0$ until it is replaced by a new test case that is closer to covering the target branch. Over few generations, DynaMOSA covers the false branch of line 137, while WSA does not, even after 10 minutes of search budget.

6.2 Co-factors analysis for DynaMOSA vs. WSA

Figure 7 plots the relation among McCabe's cyclomatic complexity, number of coverage targets, and Vargha-Delaney $\hat{A}_{12}$ scores achieved for those classes with statistically significant differences between DynaMOSA and WSA. Basically,

the figure reports three x-y scatter plots with *circles* to express a third dimension: each circle represents a class in our study, whose x coordinate is the $\hat{A}_{12}$ score achieved when comparing DynaMOSA against WSA, the y coordinate is its McCabe's cyclomatic complexity score, while the radius of the circle is proportional to the number of coverage targets. We compute the McCabe's cyclomatic complexity score of each class as the sum of the complexity scores of its methods. Each coverage criterion —i.e., branch, statement and strong mutation — is reported in a different graph.

For branch coverage (Figure 7-a), classes with $\hat{A}_{12}<0.50$ (i.e., with WSA outperforming DynaMOSA) have low cyclomatic complexity (average <51) and contain only few branches to cover (average <85). Vice versa, classes with $\hat{A}_{12}>0.50$ (i.e., with DynaMOSA outperforming WSA) have a more variable complexity, ranging between 4 and $1,244$, and number of branches varying between 17 and $7,938$. Therefore, on this coverage criterion we observe a general trend for DynaMOSA to reach higher branch coverage ($\hat{A}_{12}>0.50$) regardless of class size and complexity.

For statement coverage (Figure 7-b), $\hat{A}_{12}>0.50$ scores are achieved mainly for classes with more than 500 statements, which represent 75% of the classes for which DynaMOSA achieves significantly higher statement coverage. For cyclomatic complexity, there is more variability since it ranges between 5 and $1,244$. Only few circles (classes) fall on the left of the $\hat{A}_{12}=0.50$ point. For those few cases, the average complexity is 20 while the total number of statements to cover is 404 on average.

Finally, for strong mutation coverage, the results are mixed: classes with both $\hat{A}_{12}<0.50$ and $\hat{A}_{12}>0.50$ show variable complexity (y coordinate) as well as a variable number of mutants to kill (circle's radius). Indeed, Figure 7-c reports large circles on the left of point $\hat{A}_{12}$=0.50 indicating that there are classes with a large number of mutants for which WSA achieves better strong mutation scores. It is important to notice that for this criterion WSA prioritises the test cases to be run according to their execution time, while DynaMOSA does not do that. For classes with an extremely high number of mutants, this strategy helps in reducing the time consumed in each generation, thus, allowing the algorithm to spend more effort evolving the test suites. However, there are many more circles on the right of point $\hat{A}_{12}$=0.50, indicating that DynaMOSA outperforms WSA in many more cases even without using any prioritising strategy based on test case execution time.

To provide statistical support to the observations made above, we used a two-way permutation test [4] to verify whether any interaction between the $\hat{A}_{12}$ statistics, cyclomatic complexity and number of targets is statistically significant or not. The two-way permutation test is a non-parametric test equivalent to the two-way Analysis of Variance (ANOVA), thus, it does not make any assumption on data distributions. For this test, we use the implementation available in the `lmPerm` package for `R`. It uses an iterative procedure to compute the p-values, thus, it can produce different results over multiple runs when few iterations are used. For this reason, we set the number of iterations to 10^8.

The two-way permutation test shows that the $\hat{A}_{12}$ scores are significantly influenced by cyclomatic complexity and number of targets for branch coverage (p-values $\in \{0.04, 0.02\}$), as well as by their combination (p-value=0.02). Therefore, on this criterion DynaMOSA improves branch coverage over WSA especially for classes with high number of branches *and/or* high computational complexity. For statement coverage, the permutation test reveals a significant interaction between $\hat{A}_{12}$ scores and cyclomatic complexity combined with number of targets (p-value=0.02). However, the two factors taken alone do not significantly interact with the $\hat{A}_{12}$ scores (p-values $\in \{0.24, 1.00\}$). Therefore, DynaMOSA provides better coverage scores (with $\hat{A}_{12} > 0.50$) especially for classes with high cyclomatic complexity *and* with large number of mutants/statements to be covered. For strong mutation coverage, the two-way permutation test shows that the $\hat{A}_{12}$ scores are not significantly influenced by cyclomatic com-

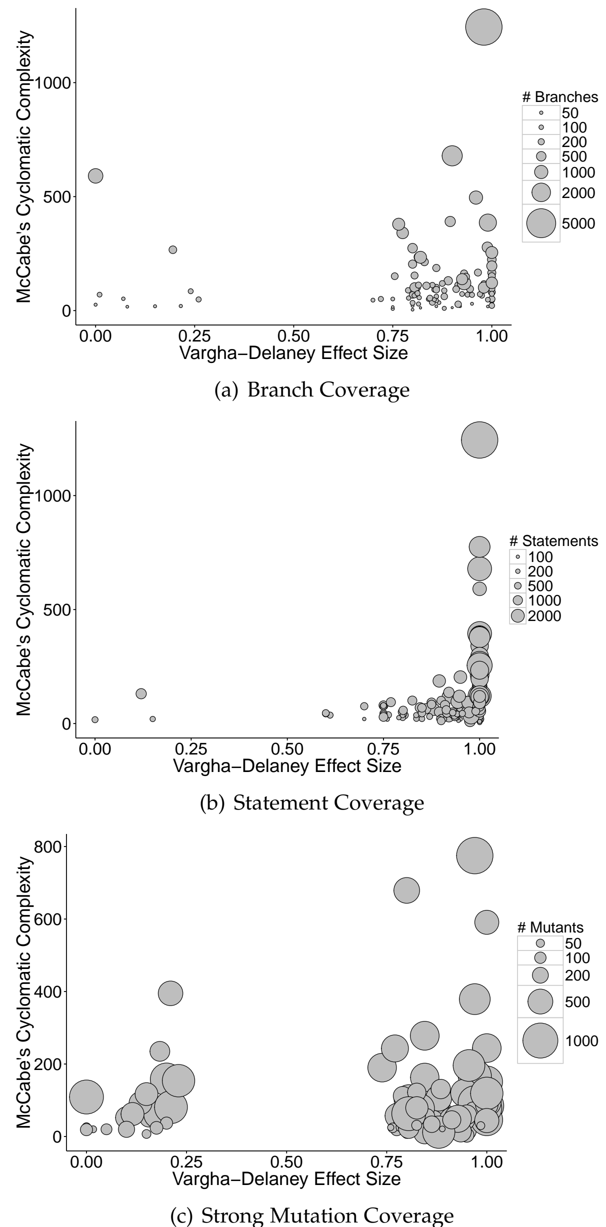


Fig. 7. DynaMOSA vs WSA: Interaction between $\hat{A}_{12}$ statistic, McCabe's cyclomatic complexity, and number of targets (branches, statements, mutants).

plexity and number of targets for branch coverage (p-values $\in \{0.98, 0.35\}$), neither by their combination (p-value=0.75).

6.3 Co-factors analysis for DynaMOSA vs. MOSA

Figure 8 reports the results of the interaction between McCabe's cyclomatic complexity (y axis), number of coverage targets (circle's radius), and Vargha-Delaney $\hat{A}_{12}$ scores (x axis) for those classes with statistically significant differences between DynaMOSA and MOSA. For branch coverage (Figure 8-a), classes with $\hat{A}_{12}$>0.50 show a variable number of branches that ranges between 21 and $2,373$, and a variable cyclomatic complexity ranging between 4 and 679. Vice versa, for the single class with $\hat{A}_{12}$<0.50 the corresponding circle has a small size (low number of branches) and low complexity scores. Therefore, for branch coverage DynaMOSA reaches higher coverage, with respect to MOSA, independently of the considered factors.

For statement coverage (Figure 8-b), $\hat{A}_{12}$<0.5 scores are achieved for classes with complexity ranging between 16 and 63, while the corresponding number of statements ranges between 152 and $1,436$. However, these two factors have higher values for classes with $\hat{A}_{12}$>0.50. Indeed, for such classes the complexity values range between 10 and 775, while the number of statements to cover ranges between 179 and $7,809$. Hence, for this criterion DynaMOSA tends to outperform MOSA for classes with both high complexity and large size.

For strong mutation, most of the classes in Figure 8-c have low complexity when $\hat{A}_{12}$<0.50 while for $\hat{A}_{12}$>0.50 we observe a larger variability in terms of complexity. There is an extreme class on the right of the line $\hat{A}_{12} = 0.50$ with a very large complexity (755) for which DynaMOSA is more effective than MOSA. For what concerns the number of mutants (radius of circles), we do not observe any relevant differences between the data points on the left and on the right sides of line $\hat{A}_{12} = 0.50$. Thus, we do not observe any influence of the number of mutants to kill on the $\hat{A}_{12}$ statistics.

To test the statistical significance of the interactions reported above, we used a two-way permutation test [4]. According to this test, for branch coverage, $\hat{A}_{12}$ scores obtained from the comparison between DynaMOSA and MOSA are not significantly influenced by cyclomatic complexity and by number of branches to cover (p-values $\in \{0.73, 0.16\}$), and neither from their combination (p-value=0.13). Similarly, for statement coverage the permutation test reveals no interaction between $\hat{A}_{12}$ statistics and both cyclomatic complexity (p-value=0.14) and number of statements (p-value=1.00). Moreover, there is no significant interaction between their combination and $\hat{A}_{12}$ scores (p-value=0.29). For strong mutation, only cyclomatic complexity has a marginal interaction with $\hat{A}_{12}$ statistics (p-value=0.07) while there is no significant influence for the number of mutants (p-value=1.00) as well as for its combination with cyclomatic complexity (p-value=0.92). Hence, DynaMOSA outperforms MOSA independently of size and complexity of the class under test for all three coverage criteria considered in this study.

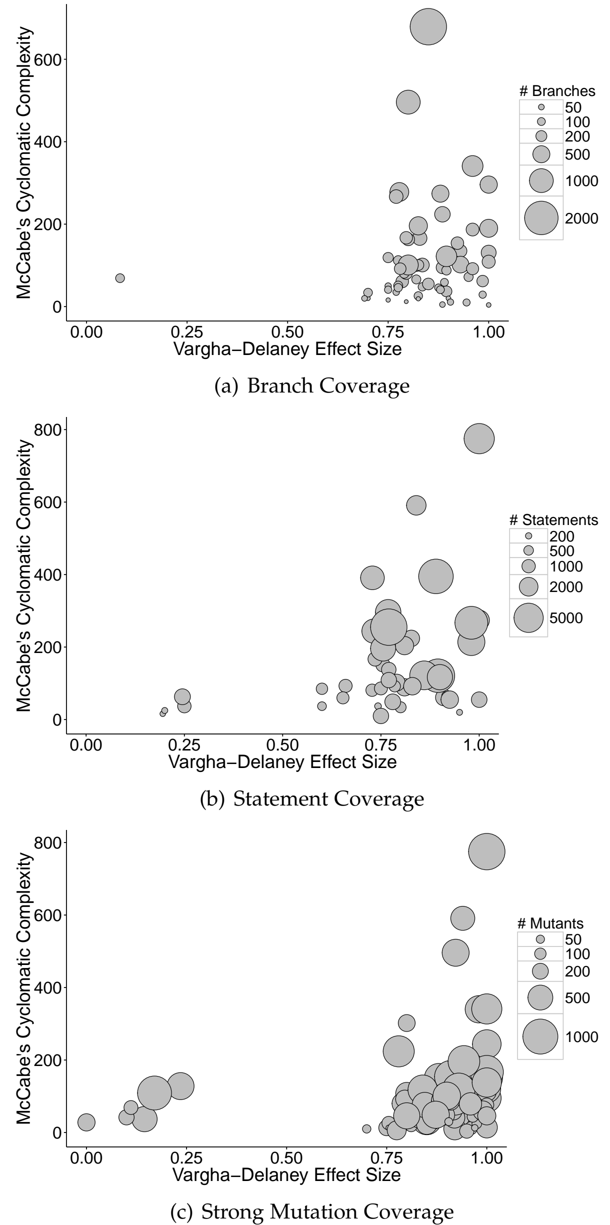


Fig. 8. DynaMOSA vs MOSA: Interaction between $\hat{A}_{12}$ statistics, McCabe's cyclomatic complexity, and number of targets (branches, statements, mutants).

7 Threats to Validity

Threats to *construct validity* regard the relation between theory and experimentation. The comparison among different search algorithms is based on performance metrics that are widely adopted in the literature: structural coverage, strong mutation scores, and running time. In the context of test case generation, these metrics give reasonable estimation of the effectiveness (coverage metrics) and efficiency (running time) of the test case generation techniques.

Threats to *internal validity* regard factors that could influence our results. To deal with the inherent randomness of GA, we repeated each execution 10 times and reported average performance together with rigorous statistical analysis to support our findings. Another potential threat arises from GA parameters. We used default parameter values

suggested in related literature since they have been shown to give reasonably acceptable results as compared to fine-tuned settings [2]. All experimented algorithms are implemented in the same tool, thus, they share the same implementation for the genetic operators. This avoids potential confounding effects due to usage of different tools with different operator implementations.

Threats to *conclusion validity* stem from the relationship between the treatment and the outcome. In analysing the results of our experiments, we have used appropriate statistical tests coupled with enough repetitions of the experiments to enable the statistical tests. In particular, we have used the two-tailed Wilcoxon test and the two-way permutation test, two non-parametric tests that do not make any assumption on the data distributions being compared. We also use the Vargha-Delaney effect size statistics for estimating the magnitude of the observed difference. We drew conclusions only when results were statistically significant according to these tests.

Threats to *external validity* affect the generalisation of our results. We carried out experiments on 346 Java classes randomly taken from 116 open-source projects belonging to four different datasets [18], [53], [49], [42]. These software projects have been used in many previous works on test case generation (e.g., [2], [19]). Moreover, in order to increase the generalisability of our findings, we evaluated all the algorithms with respect to three different, widely used, coverage criteria, i.e., branch coverage, statement coverage and strong mutation coverage.

8 Related Work

The application of search algorithms for test data and test case generation has been the subject of increasing research efforts. As a result, several techniques [5], [26], [24], [46], [56] and tools [6], [9], [40], [35], [7], [29], [57], [60], [43] have been proposed in the literature.

Search-based approaches. Existing works on search-based test generation rely on the single objective formulation of the problem, as discussed in Section 2. In the literature, two variants of the single objective formulation can be found: (i) targeting one branch at a time [1], [20], [26], [37], and (ii) targeting all branches at once (whole-suite approach) [3], [17], [19]. The first variant (i.e., targeting one branch at a time) has been shown to be inferior to the whole-suite approach [3], [17], mainly because it is significantly affected by the inevitable presence of unreachable or difficult targets. Recently, Rojas et al. [48] further improved the *whole-suite* approach by incorporating some basic routines shared with our MOSA algorithm. The corresponding approach, is a hybrid strategy which combines the traditional *evolution of test suites* with test case level operators: (1) the usage of an *archive* to keep track of test cases (and not test suites) covering some targets; and (2) synthesising the final suite by taking test cases stored in the archive rather than picking up the best individual (candidate suite) from the last generation of GA. Since the *archive* based *whole-suite* approach (WSA) has been proved to outperform both the pure *whole-suite* and targeting one branch at a time approaches, we focused on WSA as a state-of-the-art representative of the single objective approach.

Multi-objective approaches. Although in the related literature there are previous works applying multi-objective approaches in evolutionary test data generation, they all considered structural coverage as a single objective, while other, domain-specific objectives have been added as further objectives the tester would like to achieve [1], [22], such as memory consumption [32], execution time [44], test suite size [39], etc. For example, Harman *et al.* [23] proposed a search-based multi-objective approach in which the first objective is branch coverage (each coverage target is still targeted individually) and the second objective is the number of collateral targets that are accidentally covered. Ferrer *et al.* [16] proposed a multi-objective approach that considers two conflicting objectives: coverage (to be maximised) and oracle cost (to be minimised). They also used the *targeting one branch at a time* approach for the branch coverage criterion, i.e., their approach selects one branch at time and then runs GA to find the test case with minimum oracle cost that covers such a branch. Pinto and Vergilio [44] considered three different objectives when generating test cases: structural coverage criteria (*targeting one branch at a time* approach), ability to reveal faults, and execution time. Oster and Saglietti [39] considered two objectives to optimise: branch coverage (to be maximised) and number of test cases required to reach the maximum coverage (to be minimised). Lakhotia *et al.* [32] experimented bi-objective approaches, considering as objectives branch coverage and dynamic memory consumption for both real and synthetic programs. Even if they called this bi-objective formulation as *multi-objective branch coverage*, they still represent branch coverage as a single objective function, by considering one branch at a time. According to McMinn [38], there are several other potential non-coverage objectives that could be added to test case generation tools, such as minimising the oracle cost, maximising the test case diversity to increase the likelihood to expose more faults, etc.

It is important to notice that all previous multi-objective approaches for evolutionary test data generation used the *targeting one branch at a time* strategy [37]. The branch distance of a single targeted branch is one objective, considered with additional non-coverage objectives. From all these studies, there is no evidence that the usage of additional (non-coverage) objectives provides benefits in terms of coverage with respect to the traditional single-objective approach based on branch coverage alone. As reported by Ferrer *et al.* [16] the usage of such additional objectives can be even harmful for the final structural coverage. It is also important to highlight that the number of objectives considered in these studies remains limited to a relatively small number, being always ≤ 3.

Non-evolutionary approaches. Other than evolutionary approaches, other techniques have been proposed in the literature for test case and test data generation, such as *dynamic symbolic execution* [7], [29], [57], [60] and *random testing* [10], [35], [40], [41]. Dynamic symbolic execution techniques encode all constraints that should be satisfied to execute a particular path in a specific formula to be solved using constraint solvers. A solution to such a formula consists of method sequences and test data allowing to cover the corresponding path in the program [29]. Although symbolic execution has been widely applied in the liter-

ature [7], [57], there are several challenges to address for real-world programs [15], such as: *path explosion*, *complexity constraints*, *dependencies to external libraries*, and *paths related to exceptions*. Eler et al. [15] performed a large-scale study on the SF110 corpus to identify factors that negatively impact symbolic execution techniques for object-oriented software. They observed that less than 10% of java methods in SF110 have only integer or floating-point data types. Therefore, most of constraints to solve are related to complex data types (e.g., objects) posing several challenges to constraint solvers [15]. Moreover, handling calls to external libraries limits the applicability of symbolic execution for real-world software projects [15].

Direct comparison between evolutionary testing and random testing has been the subject of investigation of several researchers (e.g., [56], [25], [53], [49]) in the last decade. Most of these studies have shown that evolutionary testing outperforms random testing [56], [25], [49], which usually fails to cover hard-to-reach branches that require a quite sophisticated search [25]. For example, Fraser and Arcuri [18] conducted a large empirical study on open source projects (the SF110 corpus) and industrial projects and compared EvoSuite with Randoop [41], a popular random testing tool for Java. Their results showed that GAs (whole suite approach) lead to a large improvement over random testing [18], despite the presence of a large number of classes in SF110 that are trivial to cover. Similar results have been obtained by Shamshiri et al. [51]. On the SF110 corpus they compared the whole suite approach with a random search algorithm implemented in the same tool, i.e., EvoSuite. However, they observed that most of classes in SF110 are so trivial to cover that random search could generate test cases without much relative disadvantage for such a dataset.

Recently, Ma et al. [35] introduced an improved version of random testing, namely *guided random testing* (GRT). GRT extracts *constants* from the software under test via light-weight static analysis and reuses such *knowledge* to seed the generation process. Empirical results [35], [53] have shown GRT is able to reach competitive (and sometimes higher) coverage than the pure whole-suite approach in implemented EvoSuite. However, no empirical comparison has been performed between GRT and a more recent version of EvoSuite implementing WSA (archive-based whole suite approach), which won the latest SBST tool contest [49].

Whereas a systematic comparison of evolutionary testing (e.g., WSA, DynaMOSA) with other techniques (e.g., GRT, Randoop, etc.) would be an interesting analysis to perform, it escapes the scope of this paper. In fact, the main goal of this paper is to *determine a proper formulation of the test case generation problem in the context of evolutionary testing*. As a consequence, we have presented a new many-objective solver for evolutionary-based techniques. Further investigations and comparisons with different categories of techniques are part of our future work agenda.

Our paper. Although other existing techniques already target all branches/statements at the same time (e.g., WSA [48], GRT [35], etc.), none of them consider such coverage targets as explicit objectives to optimise. In this paper, we regard *coverage* itself as a *many-objective* problem, since the goal is to minimise simultaneously the distances between the test cases and the uncovered structural targets in the class under test. In our previous ICST 2015 paper [42] we provided a first reformulation of branch coverage as a many-objective problem. In this paper, we refine the target selection mechanism and we dynamically add yet uncovered targets that are under direct control dependency of a covered condition. This allows us to reduce the number of objectives simultaneously active during test case evolution, which further increases the effectiveness and efficiency of the proposed approach. Such extension is especially useful when the number of targets is very high, as in mutation testing or when dealing with classes that have high cyclomatic complexity/size.

Our empirical results provide also evidence that attempting all coverage targets at the same time is not equivalent to applying a many-objective solver. In fact, all algorithms investigated in our paper already attempt all coverage targets at the same time: WSA and WS consider such targets as components of a test-suite level single function to optimise; MOSA and DynaMOSA (and its variants) use such targets as independent objectives. Our results show that there exists a statistically significant difference in code coverage among the aforementioned algorithms. In particular, we found that the usage a many-objective strategy to select the candidate tests during the search process plays a paramount role on both effectiveness and efficiency in test case generation (see RQ4 on the role of the preference criterion).

9 Conclusions and Future Work

We have reformulated the test case generation problem as a many-objective optimisation problem, where different coverage targets are considered as different objectives to be optimised. Our novel many-objective genetic algorithm, DynaMOSA, exploits the peculiarities of coverage testing with respect to traditional many-objective problems to overcome scalability issues when dealing with hundreds of objectives (coverage targets). In particular, (i) it includes a new preference criterion that gives higher priority to a subset of Pareto optimal solutions (test cases), hence increasing the selective pressure; (ii) it dynamically focuses the search on a subset of the yet uncovered targets, based on the control dependency hierarchy.

Our empirical study, conducted on 346 Java classes extracted from 117 java projects belonging to four different datasets, shows that the proposed algorithm, DynaMOSA, yields strong, statistically significant improvements for coverage with respect to its predecessor, MOSA, and the whole-suite approach. Specifically, the improvements can be summarised as follows: coverage is significantly higher with respect to WSA in 28% of the classes for branch and 27% strong mutation; for these classes, DynaMOSA leads to +8% and +11% more coverage respectively. The comparison with WS shows a significant improvement for statement coverage in 51% of classes, for which the average improvement is 11%. Finally, DynaMOSA outperforms its predecessor MOSA in 12% of the classes with an average coverage improvement of 8%. Consistent results have been obtained across all three coverage criteria —i.e., branch coverage, statement coverage and strong mutation coverage. On classes with no improvement in coverage, DynaMOSA

converges more quickly, especially when limited time is given to the search. Therefore, we conclude that many-objective algorithms can be applied to test case generation, but they need to be suitably customised to deal with the hundreds and thousands of targets/objectives that are typical of test case generation.

Given the results reported in this paper, there are a few potential directions for future works. First of all, we intend to incorporate also non-coverage criteria within our many-objective algorithm, such as execution time [44] and memory consumption [32]. We also plan to further improve our preference criterion by developing adaptive strategies to enable/disable the test size (secondary non-coverage criterion) to avoid genetic drift, to enable/disable Pareto-based ranking and to activate the upper bound strategy when the number of uncovered targets/objectives is extremely high.

Recent research endeavours have resulted in improved variants of random testing techniques (e.g., [35]) which have been shown to be effective alternatives for test data generation, as opposed to being mere baselines for comparing other techniques. However, such effectiveness needs to be investigated with respect to the nature of the program under test, in particular, when the program under test exhibits complex structures. In future work, we plan to design a comprehensive experimental evaluation for investigating the comparative pros and cons of random testing techniques with respect to our DynaMOSA.

Finally, we plan to investigate the performance of our many-objective algorithm when combining multiple coverage criteria at the same time compared to the sum-scalarization strategy used by Rojas et al [47] for WS and WSA.

ACKNOWLEDGE

This work is partially supported by the National Research Fund, Luxembourg FNR/P10/03.

REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Proceedings International Conference on Software Testing, Verification and Validation (ICST)*, pages 352–361. IEEE, 2013.

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

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

[4] R. D. Baker. Modern permutation test software. In E. Edgington, editor, *Randomization Tests*. Marcel Decker, 1995.

[5] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 108–118. ACM, 2004.

[6] L. Baresi and M. Miraz. Testful: Automatic unit-test generation for java classes. In *Proceedings of the International Conference on Software Engineering - Volume 2 (ICSE)*, pages 281–284. ACM, 2010.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

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

[9] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[10] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[11] K. Deb. Multi-objective optimization. In *Search Methodologies*, pages 403–449. Springer US, 2014.

[12] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, Aug 2014.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6:182–197, 2000.

[14] F. di Pierro, S.-T. Khu, and D. Savic. An investigation on preference order ranking scheme for multiobjective evolutionary optimization. *IEEE Trans. on Evolutionary Computation*, 11(1):17–45, Feb 2007.

[15] M. M. Eler, A. T. Endo, and V. H. S. Durelli. Quantifying the characteristics of java programs that may influence symbolic execution from a test data generation perspective. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference*, COMPSAC '14, pages 181–190, Washington, DC, USA, 2014. IEEE Computer Society.

[16] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software Practise & Experience*, 42(11):1331–1362, Nov. 2012.

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

[18] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[19] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

[20] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, March 2012.

[21] J. Handl, S. C. Lovell, and J. Knowles. Multiobjectivization by decomposition of scalar cost functions. In *Parallel Problem Solving from Nature*, volume 5199, pages 31–40. Springer Berlin Heidelberg, 2008.

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

[23] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 182–191, April 2010.

[24] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 73–83, New York, NY, USA, 2007. ACM.

[25] M. Harman and P. McMinn. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

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

[27] C. Horoba and F. Neumann. Benefits and drawbacks for the use of epsilon-dominance in evolutionary multi-objective optimization. In *10th Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 641–648, New York, NY, USA, 2008. ACM.

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

[29] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[30] J. Knowles, R. A. Watson, and D. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *Evolutionary Multi-Criterion Optimization*, volume 1993 of *Lecture Notes in Computer Science*, pages 269–283. Springer Berlin Heidelberg, 2001.

[31] M. Köppen and K. Yoshida. *Substitute Distance Assignments in NSGA-II for Handling Many-objective Optimization Problems*, pages 727–741. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[32] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *9th Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1098–1105. ACM, 2007.
[33] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, Sept. 2002.
[34] B. Li, J. Li, K. Tang, and X. Yao. Many-objective evolutionary algorithms: A survey. *ACM Comput. Surv.*, 48(1):13:1–13:35, Sept. 2015.
[35] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: Program-analysis-guided random testing (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 212–223, Nov 2015.
[36] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.
[37] P. McMinn. Search-based software test data generation: a survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.
[38] P. McMinn. Search-based software testing: Past, present and future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163, 2011.
[39] N. Oster and F. Saglietti. Automatic test data generation by multi-objective optimisation. In *Computer Safety, Reliability, and Security*, volume 4166 of *Lecture Notes in Computer Science*, pages 426–438. Springer Berlin Heidelberg, 2006.
[40] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 815–816. ACM, 2007.
[41] C. Pacheco and M. D. Ernst. Randoop: Feedback-Directed Random Testing for Java. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 815–816, 2007.
[42] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.
[43] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558, New York, NY, USA, 2016. ACM.
[44] G. Pinto and S. Vergilio. A multi-objective genetic algorithm to test data generation. In *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 129–134, Oct 2010.
[45] W. Prasetya, T. Vos, and A. Baars. Trace-based reflexive testing of oo programs with t2. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 151–160, Washington, DC, USA, 2008. IEEE Computer Society.
[46] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 25–34. IEEE, 2001.
[47] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Search-Based Software Engineering*, pages 93–108. Springer, 2015.
[48] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, pages 1–42, 2016.
[49] U. Rueda, R. Just, J. P. Galeotti, and T. E. J. Vos. Unit testing tool competition: Round four. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, pages 19–28, New York, NY, USA, 2016. ACM.
[50] A. Sakti, G. Pesant, and Y. G. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, March 2015.
[51] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1367–1374, New York, NY, USA, 2015. ACM.
[52] P. Tonella. Evolutionary testing of classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128. ACM, 2004.
[53] R. Urko, T. E. J. Vos, and I. S. W. B. Prasetya. Unit testing tool competition - round three. In *8th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2015*, pages 19–24, 2015.
[54] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
[55] C. von Lücken, B. Barán, and C. Brizuela. A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational Optimization and Applications*, 58(3):707–756, 2014.
[56] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.
[57] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.
[58] S. Yang, M. Li, X. Liu, and J. Zheng. A grid-based evolutionary algorithm for many-objective optimization. *IEEE Trans. on Evolutionary Computation*, 17(5):721–736, Oct 2013.
[59] Y. Yuan, H. Xu, and B. Wang. An improved nsga-iii procedure for evolutionary many-objective optimization. In *14th Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 661–668. ACM, 2014.
[60] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.
[61] Q. Zhang and H. Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on*, 11(6):712–731, Dec 2007.
[62] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In *8th International Conference on Parallel Problem Solving from Nature (PPSN VIII*, pages 832–842. Springer, 2004.
[63] E. Zitzler, M. Laumanns, and L. Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.

Annibale Panichella received the PhD in Software Engineering from the University of Salerno in 2014 with the thesis entitled "Search-based software maintenance and testing". From March 2014 to December 2014 he was a post-doctoral researcher for the Security & Trust research unit at Fondazione Bruno Kessler in Trento (Italy). From January 2015 to September 2016, he was a post-doctoral researcher in the Software Engineering Research Group (SERG) at Delft University of Technology (TU Delft) in Netherlands. Currently, he is a Research Associate in the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. His research interests include security testing, evolutionary testing, search-based software engineering, textual analysis, and empirical software engineering. He serves and has served as program committee member of various international conference (e.g., ICSE, GECCO, ICST and ICPC) and as reviewer for various international journals (e.g., TSE, TOSEM, TEVC, EMSE, STVR) in the fields of software engineering and evolutionary computation.

Fitsum Meshesha Kifetew received his BSc and MSc degrees in Computer Science from Addis Ababa University, Ethiopia in 2003 and 2005 respectively. From 2005 to 2009 he worked as a lecturer at the Addis Ababa University. In 2009 he joined the Software Engineering unit at Fondazione Bruno Kessler (FBK) in Trento, Italy as a software engineer. In 2011 he started studying at the University of Trento towards a PhD degree, sponsored by FBK and in 2015 he obtained the PhD degree after defending his thesis "Evolutionary Test Case Generation via Many Objective Optimization and Stochastic Grammars". He is currently a postdoctoral researcher in the Software Engineering unit at FBK. His research focuses on applying search-based techniques to software engineering problems, such as test case generation and requirements prioritization. He served as program committee member of the Symposium on Search-Based Software Engineering (SSBSE) and as reviewer for various journals (TSE, STVR, IST, JSS).

Paolo Tonella is head of the Software Engineering Research Unit at Fondazione Bruno Kessler (FBK), in Trento, Italy. He is also Honorary Professor at University College London (UCL). He received his PhD degree in Software Engineering from the University of Padova, in 1999, with the thesis "Code Analysis in Support to Software Maintenance". In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is the author of "Reverse Engineering of Object Oriented Code", Springer, 2005, and of "Evolutionary Testing of Classes", ISSTA 2004. Paolo Tonella was Program Chair of ICSM 2011 and ICPC 2007; General Chair of ISSTA 2010 and ICSM 2012. He is associate editor of TSE and he is in the editorial board of EMSE and JSEP. His current research interests include code analysis, web testing, search based test case generation and the test oracle problem.