Reformulating Branch Coverage as a Many-Objective Optimization Problem

Annibale Panichella*, Fitsum Meshesha Kifetew†‡, Paolo Tonella†
†Delft University of Technology, The Netherlands
‡Fondazione Bruno Kessler, Trento, Italy
University of Trento, Trento, Italy
a.panichella@tudelft.nl, kifetew@fbk.eu, tonella@fbk.eu

Abstract—Test data generation has been extensively investigated as a search problem, where the search goal is to maximize the number of covered program elements (e.g., branches). Recently, the whole suite approach, which combines the fitness functions of single branches into an aggregate, test suite-level fitness, has been demonstrated to be superior to the traditional single-branch at a time approach.

In this paper, we propose to consider branch coverage directly as a many-objective optimization problem, instead of aggregating multiple objectives into a single value, as in the whole suite approach. Since programs may have hundreds of branches (objectives), traditional many-objective algorithms that are designed for numerical optimization problems with less than 15 objectives are not applicable. Hence, we introduce a novel highly scalable many-objective genetic algorithm, called MOSA (Many-Objective Sorting Algorithm), suitably defined for the many-objective branch coverage problem.

Results achieved on 64 Java classes indicate that the proposed many-objective algorithm is significantly more effective and more efficient than the whole suite approach. In particular, effectiveness (coverage) was significantly improved in 66% of the subjects and efficiency (search budget consumed) was improved in 62% of the subjects on which effectiveness remains the same.

Keywords: Evolutionary testing, many-objective optimization, branch coverage.

I. INTRODUCTION

Test automation, and input data generation in particular, has received substantial attention from researchers. When instantiated for branch coverage, the problem can be formulated as: find a set of test cases which maximize the number of covered branches in the software under test (SUT). Solutions based on search meta-heuristics (aka, search-based testing) deal with this problem by targeting one branch at a time and typically using genetic algorithms (GA) to generate test cases that get closer and closer to the target branch [1], [2], under the guidance of a fitness function, measuring the distance between each test case trace and the branch. With this approach, a single-objective GA is performed multiple times, changing the target branch each time, until all branches are covered or the total search budget is consumed. In the end, the final test suite is obtained by combining all generated test cases in a single test suite, including those responsible for accidental coverage.

Such a single-target strategy presents several issues [3]. For example, some targets may be more difficult to cover than others, thus, an improper allocation of the testing budget might affect the effectiveness of the search process (e.g., targeting infeasible goals will by definition fail and the related effort is wasted [3]). To overcome these limitations, Fraser and Arcuri [3] have recently proposed the whole suite (WS) approach that uses a search strategy based on (i) a different representation of candidate solutions (i.e., test suites instead of single test cases); and (ii) a new single fitness function that considers all testing goals simultaneously. From the optimization point of view, WS applies the sum scalarization approach that combines multiple target goals (i.e., multiple branch distances) into a single, scalar objective function [4], thus allowing for the application of single-objective meta-heuristics such as standard GA.

Previous works on numerical optimization have shown that the sum scalarization approach to many-objective optimization has a number of drawbacks, among which the main one is that it is not efficient for some kinds of problems (e.g., problems with non-convex region in the search space) [4]. Further studies have also demonstrated that many-objective approaches can be more efficient than single-objective approaches when solving the same complex problem [5], [6], i.e., a many-objective reformulation of complex problems reduces the probability of being trapped in local optima, also leading to a better convergence rate [5]. This remains true even when the multiple objectives are eventually aggregated into a single objective for the purpose of selecting a specific solution at the end of the (many-objective) search [5].

Stemming from the consideration that many-objective approaches can be more efficient when solving complex problems [5], [6], in this paper we propose to explicitly reformulate the branch coverage criterion as a many-objective optimization problem, where different branches are considered as different objectives to be optimized. In this new formulation, a candidate solution is a test case, while its fitness is measured according to all (yet uncovered) branches at the same time, adopting the multi-objective notion of optimality. As noted by Arcuri and Fraser [7], the branch coverage criterion lends itself to a many-objective problem, but it poses scalability problems to traditional many-objective algorithms since a typical class can have hundreds if not thousands of objectives (e.g., branches). However, we observe that branch coverage presents some peculiarities with respect to traditional (numerical) problems and we exploit them to overcome the scalability issues associated with traditional algorithms.

In this paper, we introduce a novel highly-scalable many-objective GÁ, named MOSA (Many-Objective Sorting Algorithm), that modifies the selection scheme used by existing
many-objective GA. Results achieved on 64 Java classes extracted from 16 well-known libraries show that MOSA yields significantly better results (i.e., either higher coverage or, if the same coverage is reached, faster convergence) compared to the whole suite approach. In particular, coverage was significantly higher in 66% of the subjects and the search budget consumed was significantly lower in 62% of the subjects for which the coverage was the same. The total number of branches/objectives of the subjects ranges from 50 to 1213 (215 on average). To the best of our knowledge, this is the first work in search-based software engineering and test case generation that tackles a many-objective problem with hundreds of objectives. While multiple objectives have been already considered in test case generation, such additional objectives are non-coverage related, dealing for instance with test case length, execution time, memory consumption, etc. [8], [9], [10], [11], [12], [13]. This is the first approach where branches are regarded as many-objectives to be optimized simultaneously.

The remainder of this paper is organized as follows. Section II compares the single-objective formulation of the branch coverage criterion used in WS and the many-objective reformulation proposed in this paper. Section III presents our novel many-objective algorithm MOSA. Section IV describes the design of the empirical study and reports the achieved results. Section V summarizes the main related works, while Section VI concludes the paper.

II. PROBLEM FORMULATION

This section describes the main characteristics and differences between the single-objective formulation of branch coverage used in the whole suite approach and the many-objective reformulation proposed in this paper.

A. 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 again of variable length. The fitness of a test suite is measured with respect to full branch coverage, as the sum of the individual (normalized) branch distances for all the branches in the program under test. More formally, the search problem has been formulated [3] as follows:

Problem 1: Let $B = \{b_1, \ldots, b_m\}$ be the set of branches of a program. Find a test suite $T = \{t_1, \ldots, t_n\}$ that covers all the feasible branches, i.e., one that minimizes the following fitness function:

$$\min \text{fitness}(T) = |M| - |M_T| + \sum_{b \in B} d(b, T)$$  \hspace{1cm} (1)

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 normalized branch distance for branch $b \in B$.

The fitness function above estimates how close a candidate test suite is to cover all branches of a program. The term $|M| - |M_T|$ accounts for the entry edges of unexecuted methods. The minimal normalized branch distance $d(b, t)$ for each branch $b \in B$ is defined as [3]:

$$d(b, t) = \begin{cases} 0 & \text{if } b \text{ has been covered} \\ \frac{D_{\min}(t \in T, b)}{D_{\min}(t \in T, b) + 1} & \text{if the predicated has been executed at least twice} \\ 1 & \text{otherwise} \end{cases}$$  \hspace{1cm} (2)

where $D_{\min}(t \in T, b)$ is the minimal non-normalized branch distance, computed according to any of the available branch distance computation schemes [2]; 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.

From an optimization point of view, the whole suite approach considers all the branches at the same time and aggregates all corresponding branch distances into a unique fitness function, by summing up the contributions from the individual branch distances. In other words, multiple search goals (the branches to be covered) are reduced to a single search goal by means of an aggregated fitness function. Using this kind of approach, often named scalarization [4], a problem which involves multiple goals is transformed into a traditional single-objective, scalar one, thus allowing for the application of single-objective meta-heuristics such as standard genetic algorithms.

B. Many-Objective Formulation

In this paper we reformulate the branch coverage criterion as a many-objective optimization problem, where the objectives to be optimized are the branch distances of all the branches in the class under test. More formally, in this paper we consider the following reformulation:

Problem 2: Let $B = \{b_1, \ldots, b_m\}$ be the set of branches of a program. Find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimize the fitness functions for all branches $b_1, \ldots, b_m$, i.e., minimizing the following $m$ objectives:

$$\begin{align*}
\min f_1(t) &= a_1(b_1, t) + d(b_1, t) \\
& \vdots \\
\min f_m(t) &= a_l(b_m, t) + d(b_m, t)
\end{align*}$$  \hspace{1cm} (3)

where each $d(b_i, t)$ denotes the normalized branch distance of test case $t$ for branch $b_i$, while $a(b_i, t)$ is the corresponding approach level (i.e., the minimum number of control dependencies between the statements in the test case trace and the branch). Vector $(f_1, \ldots, f_m)$ is also named fitness vector.

In this formulation, a candidate solution is a test case, not a test suite, and it is scored by branch distance + approach level, computed for all branches in the program. Hence, its fitness is a vector of $m$ values, instead of a single aggregate score. In many-objective optimization, candidate solutions are evaluated in terms of Pareto dominance and Pareto optimality [4]:

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
functions satisfy the following conditions:
\[ \forall i \in \{1, \ldots, m\}, f_i(x) \leq f_i(y) \]
and
\[ \exists j \in \{1, \ldots, m\}, f_j(x) < f_j(y) \]

Conceptually, the definition above indicates that \( x \) is preferred to (dominates) \( y \) if and only if \( x \) is better on one or more objectives (i.e., it has a lower branch distance + approach level for one or more branches) and it is not worse for the remaining objectives. 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 optimization 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 optimization 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 maximize the total coverage by covering previously uncovered branches, i.e., test cases having one or more objective scores equal to zero, i.e., \( f_i(t) = 0 \). 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.

### III. Algorithm

This section briefly summarizes the main previous works on many-objective optimization and highlights their limitations in the context of many-objective branch coverage. Then, it introduces our novel many-objective genetic algorithm, suitably defined for the many-objective branch coverage problem.

#### A. Existing Many Objective Algorithms

Multi-objective algorithms have been successfully applied within the software engineering community to solve problems with two or three objectives, such as software refactoring, test case prioritization, etc. However, it has been demonstrated that classical multi-objective evolutionary algorithms, such as the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [14] and the improved Strength Pareto Evolutionary Algorithm (SPEA2) [15], are not effective in solving optimization problems with more than three-objectives.

To overcome their limitations, new algorithms have been recently proposed that modify the Pareto dominance relation to increase the selection pressure. For example, Laumanns et al. [16] proposed the usage of an \( \varepsilon \)-dominance relation (\( \varepsilon \)-MOEA) instead of the classical one. Although this approach has been shown to be 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 optimization process significantly [17]. Zitzler and Künzli [18] proposed the usage of the hypervolume indicator instead of the 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 infeasible with hundreds of objectives as in the case of branch coverage. Yang et al. [19] 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 difference to determine the mutual relationship of individuals in a grid environment. Di Pierro et al. [20] used a preference order-approach (POGA) as an optimality criterion in the ranking stage 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. Yuan et al. [21] proposed \( \theta \)-NSGA-III, an improved version of the classical NSGA-II, where the non-dominated sorting scheme is based on the concepts of \( \theta \)-dominance to rank solutions in the environmental selection phase, which ensures both convergence and diversity.

All the many-objective algorithms mentioned above have been investigate mostly for numerical optimization problems with less than 15 objectives. Moreover, they are designed to produce a rich set of optimal trade-offs between different optimization goals, by considering both proximity to the real Pareto optimal set and diversity between the obtained trade-offs [16]. As explained in Section II-B, this is not the case for branch coverage, since we are interested in finding only the test cases having one or more objective scores equal to zero (i.e., \( f_i(t) = 0 \)), while the trade-offs are useful just for maintaining diversity during the optimization process. Hence, there are two main peculiarities that have to be considered in many-objective branch coverage as compared to more traditional many-objective problems: (i) not all Pareto optimal test cases (trade-offs between objectives) have a practical utility, hence, the search has to focus on a specific sub-set of the optimal solutions (those intersecting the \( m \) axes); (ii) for a given level of branch coverage, shorter test cases (i.e., test cases with a lower number of statements) are preferred.

#### B. MOSA: a New Many Objective Sorting Algorithm

Previous research in many-objective optimization [22] has shown that many-objective problems are particularly challenging because the proportion of non-dominated solutions increases exponentially with the number of objectives, i.e., all or most of the individuals are non-dominated. 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 [22]. 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 branch coverage, this means focusing the search effort on the test cases that are closer to one or more uncovered branches of a program. To this aim, we propose the following preference criterion in order to impose an order of preference among non-dominated test cases:

**Definition 3:** Given a branch \( b_i \), a test case \( x \) is preferred over another test case \( y \) (also written \( x \prec b_i y \)) if and only if...
Algorithm 1: MOSA

Input:
B = \{b_1, \ldots, b_m\} the set of branches of a program.
Population size M

Result: A test suite \( T \)

begin
1 \( t \leftarrow 0 \) \quad \text{current generation}
2 \( P_t \leftarrow \text{RANDOM-POPULATION}(M) \)
3 archive \( \leftarrow \text{UPDATE-ARCHIVE}(P_t) \)
4 while not (search_budget_consumed) do
5 \( Q_t \leftarrow \text{GENERATE-OFFSPRING}(P_t) \)
6 \( R_t \leftarrow P_t \cup Q_t \)
7 \( F \leftarrow \text{PREFERENCE-SORTING}(R_t) \)
8 \( P_{t+1} \leftarrow \emptyset \)
9 \( d \leftarrow 0 \)
10 while \( |P_{t+1}| + |F| \leq M \) do
11 \( \text{CROWDING-DISTANCE-ASSIGNMENT}(F_d) \)
12 \( P_{t+1} \leftarrow P_{t+1} \cup F_d \)
13 \( d \leftarrow d + 1 \)
14 Sort\( (F_d) \text{according to the crowding distance} \)
15 \( P_{t+1} \leftarrow P_{t+1} \cup F_d[1 : (M - |P_{t+1}|)] \)
16 archive \( \leftarrow \text{UPDATE-ARCHIVE}(\text{archive} \cup P_{t+1}) \)
17 \( t \leftarrow t + 1 \)
18 \( T \leftarrow \text{archive} \)
end

the values of the objective function for \( b_1 \) satisfy the following condition:
\[
f_i(x) < f_i(y)
\]
where \( f_i(x) \) denotes the objective score of test case \( x \) for branch \( b_i \) (see Section II). The best test case for a given branch \( b_i \) is the one preferred over all the others for such branch \( (x_{\text{best}}, y, \forall y \in T) \). The set of best test cases across all uncovered branches \( \{x | \exists y : x <_b y, \forall y \in T\} \) defines a subset of the Pareto front that is given priority over the other non-dominated test cases in our algorithm. When there are multiple test cases with the same minimum fitness value for a given branch \( b_i \), we use the test case length (number of statements) as a secondary preference criterion.

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-domination relation, and it increases the selection pressure by giving higher priority to the best test cases across uncovered branches. Since none of the existing many-objective algorithms considers this preference ordering, which is a peculiarity of the branch coverage criterion, in this paper we present a novel many-objective genetic algorithm, which we name MOSA (Many Objective Sorting Algorithm), incorporating the proposed preference criterion during the selection process.

As shown in Algorithm 1, MOSA starts with an initial set of randomly generated test cases that forms the initial population (line 3 of Algorithm 1). The population then evolves toward better test cases through subsequent iterations, called generations. To produce the next generation, MOSA first creates new test cases, called offspring, 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 6 of Algorithm 1).

A new population is generated using a selection operator, which selects from parents and offspring according to the values of the objective scores. Such a selection is performed by considering both the non-domination relation and the proposed preference criterion (function PREFERENCE-SORTING, at line 8 of Algorithm 1). In particular, the PREFERENCE-SORTING function, whose pseudo-code is provided in Algorithm 2, determines the test case with the lowest objective score (branch distance + approach level) for each uncovered branch \( b_i \), i.e., the test case that is closest to cover \( b_i \) (lines 2-7 of Algorithm 2). All these test cases are assigned rank 0 (i.e., they are inserted into the first non-dominated front \( F_0 \)), 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 ranked according to the traditional non-dominated sorting algorithm used by the NSGA-II [14], starting with a rank equal to 1 and so on (line 8-12 of Algorithm 2). It is important to notice that the routine FAST-NONDOMINATED-SORT assigns the ranks to the remaining test cases by considering only the non-domination relation for the uncovered branches, i.e., by focusing the search toward the interesting sub-region of the search space.

Once a rank is assigned to all candidate test cases, the crowding distance is used in order to make a decision about which test case to select: the test cases having a higher distance from the rest of the population are given higher probability of being selected. Specifically, the loop at line 11 in Algorithm 1 and the following lines 15 and 16 add as many test cases as possible to the next generation, according to their assigned ranks, until reaching the population size. The algorithm first selects the non-dominated test cases from the first front \( (F_0) \); if the number of selected test cases is lower than the population size \( M \), the loop selects more test cases from the second front \( (F_1) \), and so on. The loop will stop when adding test cases from current front \( F_d \) exceeds the population size \( M \). At end of the loop (lines 15-16), 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 \( F_d \) according to the descending order of crowding distance.

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 branches of the program under test. Specifically, after each generation MOSA stores every test case that covers previously uncovered branches in the archive as a candidate test case to form the final test suite (line 4 and 17 of Algorithm 1). To this aim, at the end
Algorithm 3: UPDATE-ARCHIVE

Input: A set of candidate test cases $T$
Result: An archive $A$

```plaintext
begin
  $A \leftarrow \emptyset$
  for $b_i \in B$ do
    $\text{best_length} \leftarrow \infty$
    $t_{\text{best}} \leftarrow \emptyset$
    for $t_j \in T$ do
      $\text{score} \leftarrow$ objective score of $t_j$ for branch $b_i$
      $\text{length} \leftarrow$ number of statements in $t_j$
      if $\text{score} == 0$ and $\text{length} \leq \text{best_length}$ then
        $t_{\text{best}} \leftarrow \{t_j\}$
        $\text{best_length} \leftarrow \text{length}$
      end if
    end for
    if $t_{\text{best}} \neq \emptyset$ then
      $A \leftarrow A \cup t_{\text{best}}$
    end if
  end for
end
```

of each generation function UPDATE-ARCHIVE (reported in Algorithm 3 for completeness) updates the set of test cases stored in the archive with the new test cases forming the last generation. This function considers both the covered branches and the length of test cases when updating the archive: for each covered branch $b_i$ it stores the shortest test case covering $b_i$ in the archive.

In summary, generation by generation MOSA focuses the search towards the uncovered branches of the program under test (both PREFERENCE-SORTING and FAST-NONDOMINATED-SORT routines analyze the objective scores of the candidate test cases considering the uncovered branches 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 towards suboptimal regions of the search space [23].

C. Graphical Interpretation of Preference Criterion

Let us consider the simple program shown in Figure 1-a and let us assume that the uncovered goals are the true branches of nodes 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 optimization goals, 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 $t$ corresponds to some point in a two-dimensional objective space as shown in Figure 1-b and 1-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, the function $f_2$ is greater than zero unless $b$ and $c$ are equal.

Let us consider the scenario reported in Figure 1-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 1-b are non-dominated and form the first non-dominated front $F_0$. Therefore, all such test cases have the same probability to be 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 $F_0$, it might happen that $A$ and/or $B$ are not kept for the next generation, while other, less useful test cases in $F_0$ are preserved. This scenario is quite common in many-objective optimization, where the number of non-dominated solutions increases exponentially with the number of objectives [22]. However, from the branch coverage point of view the two test cases $A$ or $B$ are better (fitter) than all other test cases, because they are the closest to cover 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, the first non-dominated front $F_0$ will contain only test cases $A$ and $B$ (see Figure 1-c), while all other test cases will be assigned to other, successive fronts.
IV. EMPIRICAL EVALUATION

The goal of the empirical evaluation is to assess the effectiveness and efficiency of MOSA, in comparison with state-of-the-art single-objective approaches, and in particular the whole test suite optimization (WS) approach implemented by EvoSuite [3]. Specifically, we investigated the following research questions:

- **RQ1 (effectiveness):** What is the coverage achieved by MOSA vs. WS?
- **RQ2 (efficiency):** What is the rate of convergence of MOSA vs. WS?

RQ1 aims at evaluating the benefits introduced by the many-objective (re)formulation of branch coverage and to what extent the proposed MOSA algorithm is able to cover more branches if compared to an alternative, state-of-the-art whole suite optimization approach. With RQ2 we are interested in analyzing to what extent the proposed approach is able to reduce the cost required for reaching the highest coverage.

A. Prototype Tool

We have implemented MOSA in a prototype tool by extending the EvoSuite test data generation framework. In particular, we implemented an extended many-objective GA as described in Section III. All other details (e.g. test case encoding schema, genetic operators, etc.) are those implemented in EvoSuite [3].

The experimental results reported in this section are obtained by using this prototype tool. The tool, along with a replication package, is available for download here: http://selab.fbk.eu/kifetew/mosa.html.

B. Subjects

In our empirical evaluation we used 64 Java classes from 16 widely used open source projects, many of which were used to evaluate the whole suite approach [3]. We tried to incorporate a diverse set of classes with varying levels of complexity and functionality. Table II (columns 2, 3, and 4) summarizes the details of the subjects. As running experiments on all classes from all the projects is computationally expensive, we selected classes randomly from the respective projects, with the only restriction that the total number of branches in the class should be at least 50. As can be seen from Table II, the total number of branches ranges from 50 to 1213 (on average around 215 branches). In our proposed many-objective formulation of the test data generation problem, each branch represents an objective to be optimized. Hence the set of classes used in our experiments present a significant challenge for our algorithm, in particular with respect to scaling to a large number of objectives.

C. Metrics

For comparing the two techniques, we use coverage as a measure of effectiveness and consumption of search budget as a measure of efficiency. Coverage (branch coverage) of a technique for a class is computed as the number of branches covered by the technique divided by the total number of branches in the class. Efficiency (search budget) is measured in the number of executed statements. Efficiency is used as a secondary metric, hence we compare efficiency only in cases where there is no statistically significant difference in effectiveness (i.e., coverage).

D. Experimental Protocol

For each class, each search strategy (WS or MOSA) is run and the effectiveness (coverage) and efficiency (budget consumption) metrics are collected. On 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 branch coverage is reached, the search budget is finished, or the total allocated time is elapsed. To allow reliable detection of statistical differences between the two strategies, each run is repeated 100 times. Consequently we performed a total of 2 (search strategies) × 64 (classes) × 100 (repetitions) = 12,800 experiments. Statistical significance is measured with the non-parametric Wilcoxon Rank Sum test [24] 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 higher branch coverage (RQ1) or a lower number of executed statements (RQ2). Other than testing the null hypothesis, we used the Vargha-Delaney ($\hat{A}_{12}$) statistic [25] to measure the effect size, i.e., the magnitude of the difference between the coverage levels (or number of executed statements for RQ2) achieved with different algorithms. The Vargha-Delaney ($\hat{A}_{12}$) statistic also classifies the obtained effect size values into four different levels (negligible, small, medium and large) that are easier to interpret.

There are several parameters that control the performance of the algorithms being evaluated. We adopted the default parameter values used by EvoSuite[3], as it has been empirically shown [26] that the default values, which are also commonly used in the literature, give reasonably acceptable results. The values for the important search parameters are listed in Table I.

E. Results

Table II summarizes the results (coverage) of the experiment. The coverage, averaged over 100 runs, that was achieved by each strategy is shown along with $p$-values obtained from the Wilcoxon test. Effect size metrics are also shown, indicating the magnitude of the difference. We can see from the table that out of the 64 classes, WS was statistically significantly better in 9 cases while MOSA was better in 42 cases. In the remaining 13 cases, no statistically significant difference was observed. In particular, from the analysis of the results reported in Table II we can notice that using MOSA coverage increases between 2% and 53%. It is worth noticing that such a result indicates a notable improvement if we consider the actual number of branches in our subjects. For example, if we consider class Conversion extracted from the Apache
TABLE III. BUDGET CONSUMED BY EACH APPROACH TO ACHIEVE THE BEST COVERAGE. P-VALUES AND EFFECT SIZE STATISTICS ARE ALSO SHOWN. \( \hat{A}_{12} \) LESS THAN 0.5 MEANS MOSA IS BETTER THAN WS, \( \hat{A}_{12} > 0.5 \) MEANS WS IS BETTER THAN MOSA AND \( \hat{A}_{12} = 0.5 \) MEANS THEY ARE EQUAL. STATISTICALLY SIGNIFICANT VALUES ARE PRINTED IN BOLDFACE.

<table>
<thead>
<tr>
<th>No</th>
<th>Class</th>
<th>WS</th>
<th>MOS</th>
<th>p-value</th>
<th>( \hat{A}_{12} )</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Monitor</td>
<td>294</td>
<td>906</td>
<td>0.000</td>
<td>0.34</td>
<td>Large</td>
</tr>
<tr>
<td>2</td>
<td>TDoubleShortMapDecorator</td>
<td>193701</td>
<td>158788</td>
<td>0.003</td>
<td>0.38</td>
<td>Small</td>
</tr>
<tr>
<td>3</td>
<td>TShortByteMapDecorator</td>
<td>206375</td>
<td>157995</td>
<td>0.000</td>
<td>0.32</td>
<td>Medium</td>
</tr>
<tr>
<td>4</td>
<td>TShortHash</td>
<td>203317</td>
<td>179764</td>
<td>0.078</td>
<td>Negligible</td>
<td>Negligible</td>
</tr>
<tr>
<td>5</td>
<td>LinearMath</td>
<td>99049</td>
<td>58050</td>
<td>0.000</td>
<td>0.26</td>
<td>Large</td>
</tr>
<tr>
<td>6</td>
<td>Option</td>
<td>31102</td>
<td>347356</td>
<td>0.436</td>
<td>Negligible</td>
<td>Negligible</td>
</tr>
<tr>
<td>7</td>
<td>TreeList</td>
<td>425663</td>
<td>286717</td>
<td>0.000</td>
<td>0.27</td>
<td>Medium</td>
</tr>
<tr>
<td>8</td>
<td>FunctionUtils</td>
<td>408573</td>
<td>416136</td>
<td>0.719</td>
<td>Negligible</td>
<td>Negligible</td>
</tr>
<tr>
<td>9</td>
<td>SchorTransformer</td>
<td>42033</td>
<td>31865</td>
<td>0.029</td>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>10</td>
<td>BrentOptimizer</td>
<td>89938</td>
<td>78094</td>
<td>0.000</td>
<td>0.33</td>
<td>Medium</td>
</tr>
<tr>
<td>11</td>
<td>SAXOutputter</td>
<td>353790</td>
<td>195672</td>
<td>0.000</td>
<td>0.07</td>
<td>Large</td>
</tr>
<tr>
<td>12</td>
<td>JDOMResult</td>
<td>29544</td>
<td>20335</td>
<td>0.691</td>
<td>Negligible</td>
<td>Negligible</td>
</tr>
<tr>
<td>13</td>
<td>NamespaceStack</td>
<td>273263</td>
<td>137880</td>
<td>0.000</td>
<td>0.20</td>
<td>Large</td>
</tr>
</tbody>
</table>

Overall Average

<table>
<thead>
<tr>
<th>WS</th>
<th>MOS</th>
<th>p-value</th>
<th>( \hat{A}_{12} )</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>202814</td>
<td>159181</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No. cases significantly better 17/13 10/13

For the 13 classes on which there was no significant difference in coverage between WS and MOSA, we compared the amount of search budget consumed by each search strategy (RQ2). Since for these 13 classes none of the two strategies achieved 100% coverage, they both consume the entire search budget. For this reason, we recorded the amount of search budget consumed to achieve the best coverage. This serves as an indicator of the convergence rate of the search strategy. Table III summarizes the result of the comparison of the budget consumed to achieve the final (highest) coverage. Out of 13 cases, WS consumed a significantly lower budget in only one case, while MOSA consumed significantly lower budget in 8 cases. In the remaining 4 cases, there was no statistically significant difference in budget consumption as well. Furthermore, we can see from Table III that MOSA reached the best overall mean efficiency across the 13 classes, with a reduction in budget consumption of 22%, with the minimum reduction of 18% yielded for TDoubleShortMapDecorator and the maximum one of 50% achieved for NamespaceStack. The \( \hat{A}_{12} \) effect size values shown in Table III confirm this analysis: in the majority of cases where MOSA outperforms WS, the magnitude of the difference in terms of efficiency is either large (3 out of 8 cases) or medium (3 out of 8 cases).

According to the results and the analyses reported above we can answer the research questions considered in this experiment as follows:

In summary, we can conclude that MOSA achieves higher branch coverage than WS (RQ1) and that when both achieve the same coverage, MOSA converges to such coverage more quickly than WS (RQ2).

F. Qualitative analysis

Figure 3 shows an example of branch covered by MOSA but not by WS for the class MatrixUtils extracted from the Apache commons math library, i.e., the false branch of line 165 of method createFieldMatrix. The related branch condition checks the size of the input matrix \( d \) indirectly by calling the second method createFieldDiagonalMatrix as reported in Figure 3-a. However, by analyzing all the test cases generated by WS during the search process we found that TC1 is not the closest test case to the false branch of line 165 across all generations. For example, at some generation WS generated a test case TC2 with a lower branch distance \( d = 0.9997 \) that is reported in Figure 3-c. As we can see, TC1 executes the line 162 because the input data is null, while the test case TC2 executes the branch condition in line 165 and the corresponding true branch in line 166. However, TC2 was generated within a candidate test suite with a poor whole suite fitness \( f = 170.0 \), which was also the worst candidate test suite in its generation. Thus, in the next generation this test suite (with the promising TC2) is not selected to form the next generation and this promising test case is lost. By manual
investigation we verified that this scenario is quite common, especially for classes with a large number of branches to cover. As we can see from this example, the whole suite fitness is really useful in increasing the global number of covered branches, but when aggregating the branch distances of uncovered branches, the individual contribution of single promising test cases may remain unexploited.

Unlike WS, MOSA selects the best test case (and not test suites) within the current population for each uncovered branch. Therefore, in a similar scenario it would place TC2 test suites) within the current population for each uncovered branch. Therefore, in a similar scenario it would place TC2 will be assigned to front preference criterion MOSA covers the false branch of line 165, while WS does not.

Unlike WS, MOSA selects the best test case (and not test suites) within the current population for each uncovered branch. Therefore, in a similar scenario it would place TC2 in the first non-dominated front $F_0$ according to the proposed preference criterion. Thus, generation by generation test case TC2 will be assigned to front $F_0$ until it is replaced by a new test case that is closer to covering the target branch. Eventually, MOSA covers the false branch of line 165, while WS does not.
There is a potential threat to external validity with respect to the generalization of our results. We carried out experiments on 64 Java classes taken from 16 widely used open source projects with a total number of branches ranging from 50 to 1213. While these classes exhibit a reasonable degree of diversity, further experiments on a larger set of subjects would increase the confidence in the generalization of our results. We also evaluated the performances of the compared techniques with subjects having less than 50 branches, obtaining consistent results.

V. RELATED WORK

The application of search algorithms for test data generation has been the subject of increasing research efforts. As a result, several techniques and tools have been proposed. Existing works on search based test data generation rely on the single objective formulation of the problem, as discussed in Section II. In the literature, two variants of the single objective formulation can be found: (i) targeting one branch at a time [2], and (ii) targeting all branches at once (whole suite approach [3]). The first variant (i.e., targeting one branch at a time) has been shown to be inferior to the whole suite approach [3], [7], mainly because it is significantly affected by the inevitable presence of unreachable or difficult targets. Consequently, we focused on the whole suite formulation as a state-of-the-art representative of the single objective approach.

In the related literature, previous works applying multi-objective approaches in evolutionary test data generation have been reported. However, all previous works considered branch coverage as a unique objective, while other additional domain-specific goals have been considered as further objectives the tester would like to achieve [8], such as memory consumption, execution time, test suite size, etc. For example, Harman et al. [9] proposed a search-based multi-objective approach in which the first objective is branch coverage (each goal is still targeted individually) and the second objective is the number of collateral targets that are accidentally covered. Moreover, no specific heuristics are used to help cover these other targets. Ferrera et al. [10] proposed a multi-objective approach that considers two conflicting objectives: the coverage (to maximize) and the oracle cost (to minimize). They also used the targeting one branch at a time approach for maximizing the branch coverage criterion, i.e., their approach selects one branch at a time and then runs GA for finding the test case with minimum oracle cost that covers such a branch. Pinto and Vergilio [11] 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 Saglitti [12] considered other two objectives to optimize: branch coverage (to maximize) and number of test cases required to reach the maximum coverage (to minimize). Lakhotia et al. [13] experimented bi-objective approaches by 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 with a single objective function, by considering one branch at a time.

It is important to notice that all previous multi-objective approaches for evolutionary test data generation used the
targeting one branch at a time approach [2]. The branch distance of a single targeted branch is one objective, considered with additional non-coverage goals. 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 [10]. Moreover, the number of objectives considered in these studies remains limited to a relatively small number.

Unlike previous multi-objective approaches to evolutionary test data generation, in this paper we proposed to consider the branch coverage by itself as a many-objective problem, where the goal is to minimize simultaneously the distances between the test cases and the uncovered branches in the class under test.

VI. CONCLUSIONS AND FUTURE WORK

We have reformulated branch coverage as a many-objective problem, where different branches are considered as different objectives to be optimized. Our novel many-objective genetic algorithm, MOSA, exploits the peculiarities of branch coverage with respect to traditional many-objective problems to overcome scalability issues when dealing with hundreds of objectives (branches).

An empirical study conducted on 64 Java classes extracted from widely used open source libraries demonstrated that the proposed algorithm, MOSA, (i) yielded strong, statistically significant improvements (i.e., either higher coverage or faster convergence) with respect to the whole suite approach, and (ii) is highly scalable to programs with even more than one thousand of branches. Specifically, the improvements can be summarized as follows: coverage was significantly higher in 66% of the subjects and the search budget consumed was significantly lower in 62% of the subjects for which coverage was the same.

Based on the promising results achieved in this paper, we intend to investigate different directions for future work. First, we will replicate the study, considering further java classes to corroborate the results reported in this paper. Second, we will investigate whether other adequacy testing criteria (e.g., statement coverage, mutation testing, etc.) can be reformulated as many-objective problems and then solved using the proposed novel algorithm (MOSA). We also plan to combine the proposed many-objective branch coverage with other non-coverage criteria, such as execution time [11] and memory consumption [13].

REFERENCES