A Test Case Prioritization Genetic Algorithm guided by the Hypervolume Indicator

Dario Di Nucci[*], Annibale Panichella[†], Andy Zaidman[†], and Andrea De Lucia[‡]

[*]Vrije Universiteit Brussel, Brussels, Belgium
[†]Delft University of Technology, Delft, The Netherlands
[‡]University of Salerno, Fisciano (SA), Italy

Abstract—Regression testing is performed during maintenance activities to assess whether the unchanged parts of a software behave as intended. To reduce its cost, test case prioritization techniques can be used to schedule the execution of the available test cases to increase their ability to reveal regression faults earlier. Optimal test ordering can be determined using various techniques, such as greedy algorithms and meta-heuristics, and optimizing multiple fitness functions, such as the average percentage of statement and branch coverage. These fitness functions condense the cumulative coverage scores achieved when incrementally running test cases in a given ordering using Area Under Curve (AUC) metrics.
In this paper, we notice that AUC metrics represent a bi-dimensional (simplified) version of the hypervolume metric, which is widely used in many-objective optimization. Thus, we propose a Hypervolume-based Genetic Algorithm, namely HGA, to solve the Test Case Prioritization problem when using multiple test coverage criteria. An empirical study conducted with respect to five state-of-the-art techniques shows that (i) HGA is more cost-effective, (ii) HGA improves the efficiency of Test Case Prioritization, (iii) HGA has a stronger selective pressure when dealing with more than three criteria.

Index Terms—Test Case Prioritization, Genetic Algorithm, Hypervolume.

1 Introduction

The goal of regression testing is to verify that software changes do not affect the behavior of unchanged parts [?]. Many approaches have been proposed in literature to reduce the effort of regression testing [?], [?], which remains a particular expensive post-maintenance activity [?]. One of these approaches is *test case prioritization* (TCP) [?], [?], whose goal is to execute the available test cases in a specific order that increases the likelihood of revealing regression faults earlier [?]. Since fault detection capability is unknown before test execution, most of the proposed techniques for TCP use coverage criteria [?] as surrogates with the idea that test cases with higher code coverage will have a higher probability to reveal faults. Once a coverage criterion is chosen, search algorithms can be applied to find the order maximizing the selected criterion.

Greedy Algorithms have been widely investigated in literature for test case prioritization, such as simple greedy algorithms [?], additional greedy algorithms [?], 2-optimal greedy algorithms [?], or hybrid greedy algorithms [?]. Other than greedy algorithms, meta-heuristics have been applied as alternative search algorithms to test case prioritization. To allow the application of meta-heuristics, proper fitness functions have been developed [?], such as the Average Percentage Block Coverage (APBC) or the Average Percentage Statement Coverage (APSC). Each fitness function measures the Area Under Curve (AUC) represented by the cumulative coverage and cost scores achieved when incrementally executing the test cases according to a specific prioritization (or order). As such, multiple points in the cost-coverage space are condensed into a single scalar value that can be used as a fitness function for meta-heuristics, such as single-objective genetic algorithms. Later work on search-based TCP also employed multi-objective genetic algorithms considering different AUC-based metrics as different objectives to optimize [?], [?], [?], [?].

We observed that the AUC metric used in the related literature for TCP represents a simplified version of the well-known *hypervolume* [?], which is a metric used in many-objective optimization. Indeed, the problem of condensing multiple points in the objective space (*i.e.,* a Pareto front) has been already investigated in many-objective optimization using the more general concept of *hypervolume under manifold* [?], which is a generalization of the AUC-based metrics used in previous TCP studies but for the higher dimensional objective space. We argue that the *hypervolume* can be used to condense not only a single cumulative code coverage criteria (as done by previous AUC metrics used in TCP literature) but also multiple testing criteria, such as the test case execution cost or further coverage criteria (*e.g., branch, and past-fault coverage*), in only one scalar value.

In our previous work [?], we introduced a Hypervolume-based Genetic Algorithm (`HGA`) to solve the TCP problem with multiple testing criteria. We conducted a preliminary study on six open-source programs and we compared `HGA` with the `Additional Greedy` algorithm [?], [?] when optimizing up to three testing criteria. Our preliminary results

This paper is an extension of "Hypervolume-based Search for Test Case Prioritization" that appeared in the Proceedings of the Symposium on Search-Based Software Engineering 2015 (SSBSE 2015), Bergamo, Italy, pp. 157-172, 2015 [?].

showed that HGA is not only much faster than the greedy algorithm but that the generated test orderings reveal more regression faults than the alternative algorithm for large software programs. However, despite these encouraging results, further studies are needed to answer the following questions: (i) *How does HGA perform compared to other state-of-the-art techniques for the TCP problem?* (ii) *To what extent does HGA scale when dealing with more than three testing criteria?* (iii) *To what extent does HGA scale when dealing with large software systems containing real faults?*

To answer the aforementioned open questions, in this paper we provide an extensive evaluation of Hypervolume-based and state-of-the-art approaches for TCP when dealing with up to five testing criteria (four objectives). In particular, we carry out a first case study to assess the *cost-effectiveness* and the *efficiency* of the various approaches. We compare HGA with respect to three state-of-the-art techniques: a cost cognizant Additional Greedy algorithm [?], [?], a single objective Genetic Algorithm based on an AUC metric (GA) [?], and Non-dominated Sorting Genetic Algorithm II (NSGA-II), a multi-objective search-based algorithm [?], [?], [?], [?].

A well-known limitation in many-objective optimization is that traditional multi-objective evolutionary algorithms (*e.g.,* NSGA-II) do not scale when handling more than three criteria. This happens because the number of non-dominated solutions increases exponentially with the number of objectives [?], [?], [?] (*selection resistance*). Therefore, we perform a second case study to assess the *selective pressure capabilities* of HGA when dealing with more than three criteria, comparing it with two many-objective search algorithms, namely Generalized Differential Evolution 3 (GDE3) [?] and Multi-objective Evolutionary Algorithm Based on Decomposition (MOEA/D-DE) [?]. Finally, we conduct a third case study with the aim of evaluating the performance of HGA when dealing with large software systems containing real faults. The studies are designed to answer the following research questions:

- **RQ$_1$:** *What is the cost-effectiveness and efficiency of HGA, compared to state-of-the-art test case prioritization techniques?*
- **RQ$_2$:** *How does HGA perform with respect to many-objective test case prioritization techniques?*
- **RQ$_3$:** *How does HGA perform on a large software system with real faults?*

Our results suggest that the solution (test ordering) produced by HGA is more cost-effective than the solution generated by Additional Greedy, GA, and NSGA-II. In terms of efficiency, HGA is much faster than GA and NSGA-II. Moreover, with respect to Additional Greedy, its efficiency does not decrease as the size of the software program and of the test suite increase. When comparing HGA with many-objective search algorithms (*e.g.,* GDE3 and MOEA/D-DE), we observe that it is not only more or equally effective, but it is also up to 3 times more efficient. Finally, when dealing with large software systems such as MySQL, we observe similar results to those achieved in the first case study.

The contributions of this paper compared to the conference paper [?] can be summarized as follows:

1) We extend the empirical evaluation by conducting two new case studies.
2) We partially replicate a previous study [?] on a large real-world software system, namely MySQL.
3) We compare our algorithm with five state-of-the-art algorithms for the Test Case Prioritization problem, namely Additional Greedy [?], [?], a Genetic Algorithm based on an AUC metric [?], Non-dominated Sorting Genetic Algorithm II [?], Generalized Differential Evolution 3 [?], and Multi-objective Evolutionary Algorithm Based on Decomposition [?].
4) We provide a comprehensive replication package [?] including all the raw data and working data sets of our studies.

In addition, we provide more details of the HGA algorithm, expand the discussion of related work, and provide a more qualitative discussion of the results. The remainder of the paper is organized as follows. Section **??** discusses the related literature, while Section **??** presents the proposed algorithm. Sections **??**, **??**, and **??** describe our empirical studies including the research questions and the results that we obtained. Section **??** discusses the threats that could affect the validity of the results achieved. Finally, Section **??** concludes the paper.

2 Background and Related Work

The Test Case Prioritization (TCP) problem consists of generating a test case ordering $\tau' \in PT$ that maximizes fault detection rate f [?]:

Definition 1. — *Given: a test suite T, the set of all permutations PT of test cases in T, and a function $f\colon PT \to \mathbb{R}$.*
Problem: find $\tau' \in PT$ such that $(\forall \tau'')(\tau'' \in PT)(\tau'' \neq \tau')[f(\tau') \geq f(\tau'')]$

However, the fault detection capability case is not known to the tester before test execution. Therefore, researchers have proposed to use surrogate metrics, which are in some way correlated with the fault detection rate [?], to determine test case execution order. They can be divided into two main categories [?]: *white-box* metrics and *black-box* metrics.

Code coverage is the most widely used metric among *white-box* ones, *e.g.,* branch coverage [?], statement coverage [?], block coverage [?], and function or method coverage [?]. Other prioritization criteria were also used instead of structural coverage, such as interactions [?], [?], requirement coverage [?], statement and branch diversity [?], [?], and additional spanning statement and branches [?]. Other than *white-box* metrics also *black-box* metrics have been proposed. For example, Bryce *et al.* proposed the *t-wise* approach that considers the maximum interactions between *t* model inputs [?], [?] [?]. Other approaches considered the input diversity calculated using NCD [?], the Jaccard distance [?], [?], and the Levenshtein distance [?], [?] between inputs. Finally, Henard *et al.* considered also the number of killed model mutants [?], [?]. Henard *et al.* [?] compared various *white-box* and *black-box* criteria for TCP, showing that there is a "little difference between black-box and white-box performance".

In all the aforementioned works, once a prioritization criterion is chosen, a greedy algorithm is used to order the test cases according to the chosen criterion. Two main greedy strategies can be applied [?] [?]: the *total* strategy selects test cases according to the number of code elements they cover, whereas the *additional* strategy iteratively selects the test case that yields the maximal coverage of code elements not covered yet by previously selected test cases. Recently, Hao *et al.* [?] and Zhang *et al.* [?] proposed a hybrid approach that combines *total* and *additional* coverage criteria showing that their combination can be more effective than the individual components. Greedy algorithms have also been used to combine multiple testing criteria such as code coverage and cost. For example, Elbaum *et al.* [?] and Malishevsky *et al.* [?] considered code coverage and execution cost, where the additional greedy algorithm was customized to condense the two objectives in only one function (coverage per unit cost) to maximize. Three-objective greedy algorithms have been also used to combine statement coverage, historical fault coverage, and execution cost [?], [?].

2.1 Search-Based Test Case Prioritization

Other than greedy algorithms, meta-heuristics have been investigated as alternative search algorithms to test case prioritization. Li *et al.* [?] compared additional greedy algorithm, hill climbing, and genetic algorithms for code coverage-based TCP. To enable the application of meta-heuristics they developed proper fitness functions: APBC (Average Percentage Block Coverage), APDC (Average Percentage Decision Coverage), or APSC (Average Percentage Statement Coverage). For a generic coverage criterion (*e.g.,* branch coverage), the corresponding fitness function is defined as follows:

Definition 2. — *Let $E = \{e_1, \ldots, e_m\}$ be a set of target elements to cover; let $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ be a given test case ordering; let TE_i be the position of the first test in τ that covers the element $e_i \in E$; the Average Percentage of Element Coverage, i.e., the AUC metric, is defined as:*

$$APEC = 1 - \frac{\sum_{i=1}^{m} TE_i}{n \times m} + \frac{1}{2 \times n} \tag{1}$$

In the definition above, the target elements in E can be branches (Equation **??** would correspond to APDC), statements (APSC), basic blocks (APBC), *etc.* Equation **??** condenses the cumulative coverage scores (*e.g.,* branch coverage) achieved when considering the test cases in the given order τ using the Area Under Curve (AUC) metric. This area is delimited by the cumulative points whose y-coordinates are the cumulative coverage scores (*e.g.,* statement coverage) achieved when varying the number of executed test cases (x-coordinates) according to a specified ordering [?].

Equation **??** relies on the assumption that all test cases have equal cost. However, such an assumption is unrealistic in practice and, as consequence, test orderings optimizing Equation **??** may become sub-optimal when measuring the test execution cost. In principle, the cost of each test could be measured as its actual execution time. As argued by previous studies [?], [?], such a measurement is not reliable because it depends on several external factors such as different hardware, application software, operating system, *etc.* Therefore, researchers used different metrics as proxy for the actual execution time, such as counting the number of executed statements [?], the number of executed basic blocks in the control flow graph [?], or estimating the monetary cost of each test case [?], or re-using the test execution measurements from past regression testing activities [?].

Given a measurement of the test execution cost, the "cost-cognizant" variant of Equation **??** has been defined in the literature [?] as follows:

Definition 3. — *Let $E = \{e_1, \ldots, e_m\}$ be a set of target elements to cover; let $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ be a given test case ordering; let $C = \{c_1, \ldots, c_m\}$ be the cost of tests in τ; let TE_i be the position of the first test in τ that covers the element $e_i \in E$; the "Cost-cognizant" Average Percentage of Element Coverage is defined as:*

$$APEC_c = \frac{\sum_{i=1}^{m} \left(\sum_{j=TE_i}^{n} c_j - \frac{1}{2} c_{TE_i} \right)}{\sum_{i=1}^{n} c_i \times m} \tag{2}$$

When assuming that all tests have the same cost (*i.e.,* $\forall c_i \in C, c_i = 1$), Equation **??** becomes equivalent to Equation **??** [?]. This "cost-cognizant" variant measures the AUC delimited by the cumulative points whose y-coordinates are the cumulative coverage scores (*e.g.,* statement coverage) while their x-coordinates are the cumulative test execution costs for a specified test ordering τ.

Since these metrics allow to condense multiple cumulative points in only one scalar value, single-objective genetic algorithms can be applied to find an ordering maximizing the AUC. According to the empirical results achieved by Li *et al.* [?], in most of the cases, the difference between the effectiveness of permutation-based genetic algorithms and additional greedy approaches is not significant.

2.2 Multi-objective Test Case Prioritization

Later works highlighted that given the multi-objective nature of the TCP problem, permutation-based genetic algorithms should consider more than one testing criterion. For example, Li *et al.* [?] proposed a two-objective permutation-based genetic algorithm to optimize APSC and execution cost required to reach the maximum statement coverage (cumulative cost). They use a multi-objective genetic algorithm, namely `NSGA-II`, to find a set of Pareto optimal test case orderings representing optimal compromises between the two corresponding AUC-based criteria.

Based on the concept of *Pareto optimality* [?], in this formulation of the problem, a test cases permutation τ_A is better than another permutation τ_B, (and vice versa), if and only if τ_A outperforms τ_B in at least one objective and it is not worse in all other objectives. Formally:

Definition 4. — *Given two permutations of test cases, τ_A and τ_B, and a set of n functions (objectives) $f : PT \rightarrow \mathbb{R}$, τ_A dominates τ_B ($\tau_A \prec \tau_B$) if an only if:*

$$\begin{gathered} f_i(\tau_A) \geq f_i(\tau_B), \forall i \in 1, 2, \cdots, n \\ \textit{and} \\ \exists i \in 1, 2, \cdots, n : f_i(\tau_A) > f_i(\tau_B) \end{gathered} \tag{3}$$

Definition 5. — *Given the concept of Pareto dominance and a set of feasible solutions Ω, a solution τ^* is Pareto optimal if a solution able to dominate it does not exist, namely:*

$$\nexists \tau_A \in \Omega : \tau_A \prec \tau^* \tag{4}$$

Definition 6. — *A Pareto Front is a set composed of Pareto optimal solutions.*

$$P^* = \{\tau^* \in \Omega\} \tag{5}$$

It is worth considering that multi-objective approaches for test case prioritization return a Pareto front of permutations, that is a set of Pareto optimal test orderings.

Islam *et al.* [?] and Marchetto *et al.* [?] used `NSGA-II` to find Pareto optimal test case orderings representing trade-offs between three different AUC-based criteria: (i) cumulative code coverage, (ii) cumulative requirement coverage, and (iii) cumulative execution cost. Similarly, Epitropakis *et al.* [?] compared greedy algorithms, `MOEAs` (`NSGA-II` e `TAEA`), and hybrid algorithms. As already done by Islam *et al.* [?] and Marchetto *et al.* [?], they considered different AUC-based fault surrogates: statement coverage (APSC), Δ-coverage (APDC), and past fault coverage (APPFD). They showed that three-objective `MOEAs` and hybrid algorithms are able to produce more effective solutions with respect to those produced by additional greedy algorithms based on a single AUC metric.

In this paper, we notice that these approaches [?], [?], [?], [?] to test case prioritization have important drawbacks. First of all, these measures are computed considering the Area Under Curve obtained plotting the value of the metric with respect to the test cases position in a Cartesian plan [?] and then computing a numerical approximation of the Area Under Curve, using the Trapezoidal rule [?]. These values are projections of a manifold of cumulative points (*e.g.*, a projection of a volume into two areas). Therefore, despite the AUC metrics being strictly dependent on each other, the different AUC metrics are calculated independently (we will show an example in Section **??**). Moving to this multi-objective paradigm where AUC metrics are treated as independent objectives has an additional overhead compared to a single-objective search. In multi-objective search, the computational complexity of computing the dominance relation for all pairs of candidate test permutations is $O(n^2 \times m)$, where n is the number of test permutations and m is the number of AUC metrics. Instead, in single-objective search, the cost of sorting the individuals to select the best ones is $O(n \times log\ n)$ for stochastic selection or $O(n)$ with tournament selection. In general, the selection in single-objective search is less expensive than the selection in a multi-objective paradigm.

Moreover, the tester has to inspect the Pareto front in order to find the most suitable solution with respect to the testing criteria but no guidelines are provided for selecting the ordering (Pareto optimal solution) to use. The Pareto efficient solutions generated by a multi-objective search are trade-offs in the space of the AUC metrics and not in the space of the original testing criteria, which are the actual aspects that decision-makers (*a.k.a.* testers) look at for regression testing purpose. Furthermore, each solution in the Pareto front represents a permutation of tests and selecting a different permutation requires re-evaluating all the test cases in that permutation.

Another important limitation of these classical multi-objective approaches is that they lose their effectiveness as the problem dimensionality increases, as demonstrated by previous work in numerical optimization [?]. Therefore, other non-classical many-objective solvers must be investigated when dealing with multiple (many) testing criteria. Finally, in [?], [?], [?], [?] there is a lack of strong empirical evidence of the effectiveness of `MOEAs` with respect to simple heuristics, such as greedy algorithms, in terms of cost-effectiveness.

In this paper, we notice that the most natural way to deal with the multi-objective TCP problem is represented by the *hypervolume*-based solvers since the AUC metrics used in the related literature for TCP represent a specific simplified version of the *hypervolume* metric [?]. Indeed, in many-objective optimization, the *hypervolume* metric is widely used to condense points from a higher dimensional objective space in only one scalar value. For these reasons, in this paper, we propose to use a hypervolume metric to solve the multi-objective TCP problem. Moreover, because of the *monotonicity* properties of the coverage criteria, the computation of the hypervolume for TCP requires polynomial time versus the exponential time required for traditional many-objective problems.

2.2.1 Hypervolume-based many-objective optimization

Multi-objective meta-heuristics have been successfully applied in the literature to solve a number of software engineering problems, such as software refactoring [?], test data generation [?], defect prediction [?], [?], and regression testing [?], [?]. These problems have often been solved with algorithms like the `Non-dominated Sorting Genetic Algorithm II` (`NSGA-II`) [?] or the improved `Strength Pareto Evolutionary Algorithm` (`SPEA2`) [?], which are very effective for problems with two or three objectives. However, handling more than four objectives is particularly challenging as the number of non-dominated solutions may exponentially increase with the number of objectives to optimize. In this scenario, the classical non-dominated sorting algorithms or other classical environmental selection mechanisms are not able to promote some solutions over the others within a given population (*selection resistance* [?], [?]) because all solutions are incomparable (*i.e.*, they do not dominate each other).

To address this problem, researchers in the evolutionary computation community developed a new class of meta-heuristics, often referred to as *many-objective algorithms*, for problems with more than three search objectives. According to a recent survey by Li *et al.* [?], strategies to address the selective resistance include *diversity-based*, *reference set based*, and *indicator-based algorithms*. For example, the `Generalized Differential Evolution 3` (`GDE3`) [?] relies on a diversity-based mechanism to improve the selection pressure. `GDE3` extends differential evolution (DE) for constrained multi-objective and many-objective optimization where the population for the next generation is generated by combining the non-dominated sorting with a pruning algorithm for the non-dominated set. The non-dominated set is pruned according to the solution diversity, which is measured with a *crowding estimation* metric based on the *nearest neighbors*. Solutions having the smallest distance to their neighbors are the most crowded ones in the non-dominated set and can be pruned out. Kukkonen and Deb [?] showed that GDE3 with crowding estimation is effective and efficient in producing well-diversified trade-offs for problems with more than two objectives.

MOEA/D is decomposition-based evolutionary algorithm [?] which decomposed a multi- or a many-objective problem into multiple single-objective problems obtained via sum-scalarization. Specifically, it specifies beforehand a set of predefined search directions uniformly distributed over the entire Pareto-optimal front. These directions are obtained by normalizing the search objectives and combining them using a weighted sum approach, where different weights are used to specify the different search directions to consider. Then, the MOEA/D promotes solutions that are closer to these directions, which therefore correspond to well-distributed reference points the search aims to reach. The idea of using predefined reference points has been proved to be so effective that it has been reused and extended in more recent many-objective algorithms, such as NSGA-III [?], the surface-based evolutionary algorithm (SEA) [?], and other decomposition-based evolutionary algorithms [?].

The closest many-objective algorithms to HGA are the indicator-based evolutionary algorithms. The first algorithm proposed in the literature and falling in this category is IBEA [?], an evolutionary algorithm that selects solutions based on a binary hypervolume indicator that compares the portion of hypervolume they dominate. Emmerich *et al.* [?] proposed SMS-EMOA, which is a *steady-state* evolutionary algorithm that combines non-dominated sorting with an hypervolume-based selection. It first uses the non-dominated sorting to determine the set of non-dominated solutions in each generation. Then, solutions with the least hypervolume contribution are discarded if the number of non-dominated solutions is larger than the fixed population size. Recently, Jiang *et al.* [?] proposed a more-efficient algorithm for the exact computation of the hypervolume. However, no analysis or proof is given about its worst-case computational complexity.

While the aforementioned hypervolume-based evolutionary algorithms help to generate better solutions than classical multi-objective algorithms (*e.g.,* NSGA-II, SPEA2), they are particularly expensive due to the algorithms used to compute the hypervolume, whose complexity is exponential in the number of objectives [?], [?]. Indeed, previous studies showed that there is no polynomial algorithm available for the exact computation of the hypervolume dominated by a generic set of non-dominated solutions [?]. To cope with the computation cost of the exact hypervolume computation, researchers have proposed various approximating strategy. Bader and Zitzler [?] used Monte Carlo simulation to approximate the exact hypervolume values. Ishibuchi *et al.* [?] used a scalarizing function-based method to approximate the hypervolume metric.

Our approach follows the line of research defined by the aforementioned hypervolume-based evolutionary algorithms. However, we introduce a novel polynomial-time algorithm for the exact computation of the hypervolume but that is applicable when the functions used in the hypervolume computation are monotonic, such as in case of the TCP problem. Therefore, defining a polynomial-time algorithm for the exact computation of the hypervolume indicator for any set of solutions (*i.e.,* for any problem) still remain an open challenge. Since our algorithm provides an exact computation of the hypervolume for TCP, there is no need for the usage of approximation strategies in our context.

3 Hypervolume Genetic Algorithm for Test Case Prioritization

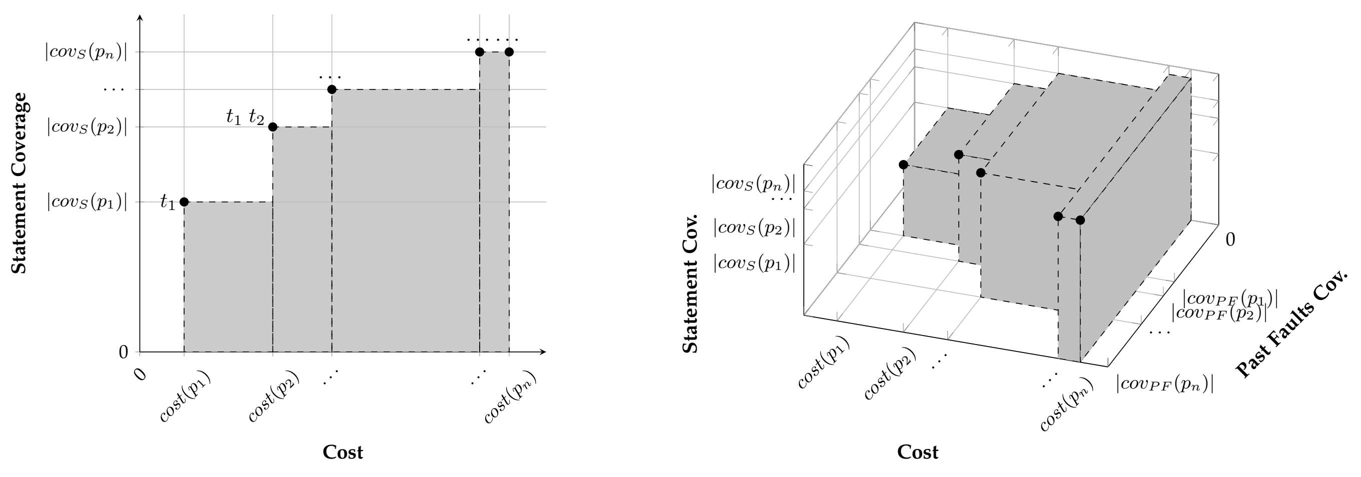
This section describes the proposed hypervolume metric for the multi-objective test case prioritization problem. It also highlights connections and differences with the AUC-based metrics used in previous work on search-based test case prioritization [?], [?], [?], [?], [?].

3.1 Hypervolume indicator

In many-objective optimization, there is a growing trend to solve many-objective problems using *quality scalar indicators* to condense multiple objectives into a single objective [?]. Therefore, instead of optimizing the objective functions directly, indicator-based algorithms are aimed at finding a set of solutions that maximize the underlying quality indicator [?]. One of the most popular indicators is the *hypervolume*, which measures the quality of a set of solutions as the total size of the objective space that is dominated by one (or more) of such solutions (combinatorial union [?]). For two-objective problems, the *hypervolume* corresponds to the area under the curve, *i.e.,* the portion of the area that is dominated by a given set of candidate solutions, while for three-objective problems it is represented by the volume.

Hypervolume in two-objective TCP. To illustrate intuitively the proposed hypervolume metric, let us consider for simplicity only two testing criteria: (i) maximizing the statement coverage and (ii) minimizing the execution cost of a test suite. When considering the test cases in a specific order, the cumulative coverage and the cumulative execution cost reached by each test case draw a set of points within the objective space.

For example, let us consider the test suite $T = \{t_1, t_2, \ldots, t_n\}$ with the following statement coverage $Cov = \{cov_S(t_1), cov_S(t_2), \ldots, cov_S(t_n)\}$ and execution cost $Cost = \{cost(t_1), cost(t_2), \ldots, cost(t_n)\}$. As depicted in Figure **??**-(a), if we consider the ordering $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ we can measure the cumulative scores as follows: the first test case t_1 covers a specific set of code statements $cov_S(p_1) = cov_S(t_1)$ with cost equal to $cost(p_1) = cost(t_1)$ (first cumulative point p_1); the second test case in the ordering t_2 reaches a new cumulative statement coverage $cov_S(p_2) = cov_S(p_1) \cup cov_S(t_2)$ with $cost(p_2) = cost(p_1) + cost(t_2)$ (second cumulative point p_2). In general, $cov_S(p_i) = cov_S(p_{i-1}) \cup cov_S(t_i)$ and $cost(p_i) = cost(p_{i-1}) + cost(t_i)$. Thus, each test case prioritization corresponds to a set of points in the two-objective space denoted by the two testing criteria, *i.e.,* statement coverage and execution cost in our example (see Figure **??**-(a)). These points are *weakly monotonically increasing* since cumulative cost increases, while cumulative coverage is stable or increases when adding a new test case from the ordering,



(a) Two testing criteria

(b) Three testing criteria

Fig. 1. Cumulative points in two- and three-objective test case prioritization. The gray area (or volume) denotes the portion of objective space dominated by the cumulative points $P(\tau)$.

i.e., $cov_S(p_i) \subseteq cov_S(p_{i+1})$ and $cost(p_i) \leqslant cost(p_{i+1})$. Note that in Figure **??**-(a) $|cov_S(p_i)|$ denotes the cardinality of the set $cov_S(p_i)$.

Given this set of points, we can measure how quickly the given ordering τ optimizes the two objectives by measuring the proportion of the *area* dominated by the corresponding cumulative points $P(\tau)$, denoted by the gray area in Figure **??**-(a). The *dominated area* is represented by all points in the objective space that are worse than the cumulative points according to the concept of *dominance* in the multi-objective paradigm in Definition **??**. Notice that by definition [**?**], the area dominated by a given point $A = (x_a, y_a)$ within the bi-dimensional objective space $F = \{cost, |cov|\}$ (*i.e.,* cumulative cost and cumulative coverage) is the rectangle (area) delimited by all points in F such that $cost \geq x_a$ and $|cov| \leq y_a$. For example, the area dominated by a cumulative point p_i in Figure **??**-(a) is the rectangle (area) delimited by $cost \geq cost(p_i)$ and $|cov| \leq |cov_S(p_i)|$. Given a set of non-dominated points $P(\tau)$ within the bi-dimensional objective space $F = \{cost, |cov|\}$, the overall dominated area is given by the union of the area (rectangle) dominated by each single point $p_i \in P(\tau)$ [**?**].

Two different orderings correspond to two different sets of cumulative points and then two different dominated areas. Therefore, we can compare the corresponding fraction of dominated areas to decide whether one candidate test case ordering is better or not than another one (fitness function): larger dominated areas imply faster statement coverage rate. In this two-objective space, the dominated area can easily be computed as the sum of the rectangles of width $[cost(p_{i+1}) - cost(p_i)]$ and height $|cov_S(p_i)|$ as reported in Figure **??**-(a).

Hypervolume in three-objective TCP. Similarly, if we consider a third testing criterion (such as past faults coverage $|cov_{PF}(p_i)|$) each candidate prioritization corresponds to a set of points in a three-dimensional space and, in this case, the dominated proportion of the objective space is represented by a volume instead of an area, as depicted in Figure **??**-(b). Since even in this three-objective space the cumulative points are always *weakly monotonically increasing*, the dominated volume can be computed as the sum of the parallelepipeds of width $[cost(p_{i+1}) - cost(p_i)]$, height $|cov_S(p_i)|$, and depth $|cov_{PF}(p_i)|$.

Hypervolume in N-objective TCP. For more than three testing criteria the objective space dominated by a set of cumulative points is called a *hypervolume* and represents a generalization of the *area* for a higher dimensional space. Without loss of generality, let $T = \{t_1, t_2, t_3, \ldots, t_n\}$ be a test suite of size n and $F = \{cost, Cov_1, \ldots, Cov_m\}$ a set of testing criteria used to prioritize the test cases in T, where *cost* denotes the execution cost of each test case while $Cov_1, \ldots, Cov_m$ are the remaining m testing criteria to maximize. Given a permutation τ of test cases in T we can compute the corresponding set of cumulative points $P(\tau) = \{p_1, \ldots, p_n\}$ obtained by cumulating the scores $cost, Cov_1, \ldots, Cov_m$ achieved by each test case in τ.

Definition 7. — *The hypervolume dominated by a permutation $P(\tau)$ of test cases can be computed as follows:*

$$I_H(\tau) = \sum_{i=1}^{n-1} \Big[[cost(p_{i+1}) - cost(p_i)] \times |Cov_1(p_i)| \times \cdots \times |Cov_m(p_i)| \Big] \tag{6}$$

where $[cost(p_{i+1}) - cost(p_i)] \times |Cov_1(p_i)| \times \cdots \times |Cov_m(p_i)|$ measures the hypervolume dominated by a generic cumulative point p_i, but non-dominated by the next point p_{i+1} in the ordering τ. Since in test case prioritization the maximum values of all the testing criteria are known (*e.g.,* the maximum execution cost or the maximum statement coverage are already known), we can express the hypervolume as a fraction of the whole objective space as follows:

Definition 8. — *The fraction of the hypervolume dominated by a permutation $P(\tau)$ of test cases is:*

$$I_{HP}(\tau) = \frac{\sum_{i=1}^{(n-1)} \Big[[cost(p_{i+1}) - cost(p_i)] \times |Cov_1(p_i)| \times \cdots \times |Cov_m(p_i)| \Big]}{cost(p_n) \times |Cov_1^{max}| \times \ldots |Cov_m^{max}|} \tag{7}$$

Algorithm 1: Hypervolume Computation

```
Input: Permutation of test cases $\tau = \langle t_1, \ldots, t_n \rangle$
       Execution cost vector $Cost = \{c_1, \ldots, c_n\}$
       Testing criteria to maximize $F = \{Cov_1, \ldots, Cov_m\}$
Result: Hypervolume score for $\tau$
1  begin
                                                   /* Initialization */
2      $I_{HP}(\tau) = 0$
3      cumCost = 0, cumCov$_1$ $= \emptyset, \ldots,$ cumCov$_n$ $= \emptyset$
4      for each i=1...(m-1) do
5          cumCost = cumCost + $c_i$
6          for each $f_i \in F$ do
7              cumCov$_i$ = cumCov$_i$ $\cup$ $Cov_i(t_i)$
8          slice = $c_{i+1}$ $\times$ |cumCov$_1$| $\times \cdots \times$ |cumCov$_m$|
9          $I_{HP}(\tau) = I_{HP}(\tau)$ + slice
           /* The loop ends when the maximum coverage is
           reached */
10         if $\forall Cov_i \in F$, $cumCov_i == Cov_i^{max}$ then
11             break
       /* Adding the remaining portion of hypervolume */
12     slice = ($cost^{max}$ - cumCost) $\times |Cov_1^{max}| \times \cdots \times |Cov_m^{max}|$
13     $I_{HP}(\tau)$ = $I_{HP}(\tau)$ + slice
                                         /* Normalizing the hypervolume */
14     for each $f_i \in F$ do
15         $I_{HP}(\tau)$ = $I_{HP}(\tau)$ / $|Cov_i^{max}|$
16     $I_{HP}(\tau)$ = $I_H(\tau)$ / $cost^{max}$
```

where $cost(p_n)$ is the execution cost of the whole test suite T and $|Cov_i^{max}|$ denotes the maximum values for the i-th coverage criterion. Such a metric ranges in the interval $[0; 1]$. It is equal to +1 in the ideal case where the test case ordering allows to reach the maximum test criteria scores independently from the execution cost value $cost(p_i)$. A higher $I_{HP}(\tau)$ mirrors a higher ability of the prioritization τ in maximizing the testing criteria with lower cost.

3.1.1 *Hypervolume complexity*

As pointed out by Auger *et al.* [?], the computation of the hypervolume indicator is usually not a trivial task and it is strongly impacted by the choice of the reference points and the distribution of solutions on the Pareto front. Despite this, it is worth noting that in the case of Test Case Prioritization a candidate test case ordering corresponds to a set of *monotonically* increasing cumulative scores. For this reason, we can use Equation **??** to compute the dominated hypervolume instead of the more expensive algorithm proposed by Auger *et al.* [?]. Indeed, the $I_{HP}(\tau)$ metric sums up the slices of dominated hypervolume delimited by two subsequent cumulative points. Thus, let m be the number of the testing criteria and let n be the number of cumulative points (corresponding to the size of the test suite), $I_{HP}(\tau)$ requires to sum the n hypervolume slices, each one computed as the multiplication of m test criteria scores. Thus, the overall computational time is $O(n \times m)$. Conversely, in traditional many-objective optimization the points delimiting the non-dominated hypervolume are non-monotonically increasing and thus, the computation of the hypervolume metric requires a more complex algorithm which is exponential with respect to the number of objectives m [?], or testing criteria for TCP.

3.1.2 *Efficient hypervolume computation*

To speed up the computation of the hypervolume metric, we use Algorithm **??**. Given a permutation of test cases τ, the corresponding execution cost array $cost$, and a set of testing criteria to maximize $Cov_1, \ldots, Cov_m$; the algorithm initializes the cumulative coverage scores (line 3 of Algorithm **??**). Such scores are then incrementally updated for each test case in the given order τ (main loop in lines 4-11). In particular, for each test t in τ, the algorithm computes the cumulative cost (line 5) and cumulative coverage scores (lines 6-7), one cumulative coverage score for each testing criterion $Cov_i \in F$. Then, the cumulative scores are used to compute the actual $I_{HP}(\tau)$ (lines 8-9). If the maximum coverage is reached earlier for all $Cov_i \in F$ (*i.e.,* before iterating over all $t \in \tau$), the loop is terminated (lines 10-11). The remaining portion of the $I_{HP}(\tau)$ metric is added in lines 12-13 of Algorithm **??**: it corresponds to the hypervolume of size $(cost^{max} - cumCost) \times |Cov_1^{max}| \times \cdots \times |Cov_m^{max}|$. Finally, $I_{HP}(\tau)$ is normalized in lines 14-16. The core idea of Algorithm **??** is to reduce the number of iterations needed to compute $I_{HP}(\tau)$ given the fact that the remaining portion of the hypervolume is known a priori when the maximum cumulative coverage is reached for all testing criteria in F.

To better understand how Algorithm **??** works, let us consider the example of the test suite shown in Table **??**. The test suite contains five test cases, whose execution time and coverage information are also shown in the table. Table **??** shows how the hypervolume is computed in each step of Algorithm **??** for the prioritization $\tau = \langle t_5, t_3, t_4, t_2, t_1 \rangle$. First, the hypervolume and the cumulative scores are initialized as specified in line 3 of Algorithm **??**. In the first iteration of the algorithm, the cumulative scores are updated based on t_5, which is the first test case in the permutation. t_5 covers four branches, five statements, and its cost is 14s. Therefore, the hypervolume score is updated according to Equation **??** as $I_{HP}(\tau)$ = (61s-14s) $\times$ 4 (branches) $\times$ 5 (statements) = 940. In the second iteration, the coverage scores are updated by considering the second test in the permutation τ, *i.e.,* t_3. Such a test covers two additional branches and six additional statements compared to t_5. Therefore, the new hypervolume is $I_{HP}(\tau)$ = 940 (previous value) + (105s-61s) $\times$ 6 (branches) $\times$ 11 (statements) = 3,844. The third test case in the permutation is t_4, which covers two additional branches and two additional statements with an additional cost of 44s. Thus, in the third iteration of Algorithm **??**, the new hypervolume value is computed as $I_{HP}(\tau)$ = 3,844 (previous value) + (124s-105s) $\times$ 8 (branches) $\times$ 13 (statements) = 5,820. The first three test cases already allow to reach 100% of branch and 100% of statement coverage; thus, the main loop in lines 4-11 of Algorithm **??** is terminated without iterating over the remaining two test cases t_1 and t_2. In the second last row of Table **??**, the hypervolume is updated according to lines 12-13 of Algorithm **??**. Specifically, $I_{HP}(\tau)$ = 5,820 (previous value) + (165s-124s) $\times$ 8 (branches) $\times$ 13 (statements) = 10,084. Finally, the hypervolume is normalized by diving $I_{HP}(\tau)$ by the hypervolume of the hyper-rectangle whose sides are equal to the overall cost and coverage achievable by running all tests in τ. Specifically, the final hypervolume score for the permutation τ is $I_{HP}(\tau)$ = 10,084 / (165s $\times$ 8 $\times$ 13) $\approx$ 0.5876.

TABLE 1
An example of test suite $T = \{t_1, t_2, t_3, t_4, t_5\}$ for a small program with eight branches, 13 statements. For every test t, we specify which branches and statement are covered by t as well as its execution cost (time in s).

Tests	Branches								Statements													Cost
	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	
t_1	X	X	X						X	X	X											41
t_2			X	X	X						X	X	X	X	X							19
t_3		X				X	X			X					X	X	X	X	X			47
t_4	X				X			X	X				X	X	X					X	X	44
t_5			X	X			X	X			X	X	X							X	X	14

TABLE 2
Walkthrough of Algorithm ?? for the test suite in Table ?? and the prioritization $\tau = \langle t_5, t_3, t_4, t_2, t_1 \rangle$.

Steps	Selected Tests	Cumulative scores			I_{HP}
		Cost	Branch Cov.	Stmt Cov.	
Initialization	-	0	0	0	0
Iteration 1	t_5	14	4	5	940
Iteration 2	t_3	61	6	11	3,844
Iteration 3	t_4	105	8	13	5,820
Adding the last part of the volume	t_1, t_2	165	8	13	10,084
Normalization	-	-	-	-	0.5876

Algorithm 2: Hypervolume Genetic Algorithm

```
Input:
Solution representation: permutation of test cases
Fitness function: $I_{HP}(\tau)$
Result: the best permutation of test cases according to $I_{HP}(\tau)$
1 begin
2     initialize population with random candidate solutions
3     evaluate each candidate solution
4     while max # of generations has not been reached do
5         select best individuals based on $I_{HP}(\tau)$ using binary
            tournament selection
6         recombine pairs of individuals using PMX-Crossover
7         mutate individuals using SWAP-Mutation
8         evaluate each candidate solution
```

3.2 Hypervolume-based Genetic Algorithm

In this paper, we consider the $I_{HP}(\tau)$ metric as a suitable fitness function to guide search algorithms in finding the optimal ordering τ in multi-objective test case prioritization. In particular, we applied the Genetic Algorithm (GA) [?], a stochastic search technique based on the mechanism of natural selection and natural genetics. We selected this algorithm because it has been used to solve a wide range of optimization problems that are not solvable in polynomial time. Moreover, with respect to other search algorithms, it is highly parallelizable [?].

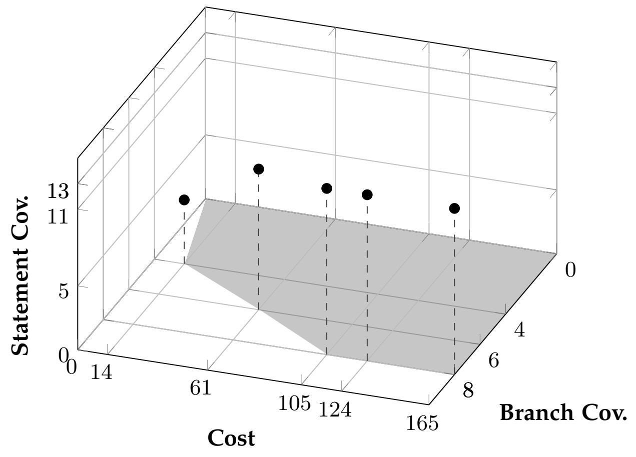
GA starts with a random population of solutions. Each individual (*i.e.,* chromosome) represents a solution of the optimization problem. The population evolves through subsequent generations where individuals are evaluated based on a fitness function to be optimized. At each generation, new individuals (*i.e.,* offsprings) are created by applying three operators: (i) a selection operator, based on the fitness function, (ii) a crossover operator, that recombines two individuals from the current generation with a given probability, and (iii) a mutation operator, which modifies the individuals with a given probability.

We propose a new genetic algorithm named `HGA` (Hypervolume-based Genetic Algorithm), depicted in Algorithm ??. Despite, GAs are commonly used for solving single-objective problems, using the *hypervolume* indicator as fitness function, it is possible to combine multiple objectives in a single one. Each solution is a permutation of integers in which each element represents a test case to be executed and the population is represented by a set of different test case permutations. The selection operator is the *binary tournament selection* (line 5), which randomly picks two individuals for the tournament and selects the one with the better fitness function. The crossover operator is the *PMX-Crossover* (line 6), which swaps the permutation elements at a given random crossover point. The mutation operator is the *SWAP-Mutation* (line 7) that randomly swaps two chosen permutation elements within each offspring. More details on the parameter settings are reported in Section ??. The fitness function that drives the GA evolution is the *hypervolume* indicator described in Section ??. `HGA` can be briefly summarized as (i) generating test cases orderings, (ii) evaluating the permutations using the $I_{HP}(\tau)$ metric, and (iii) using this value to drive the GA evolution.

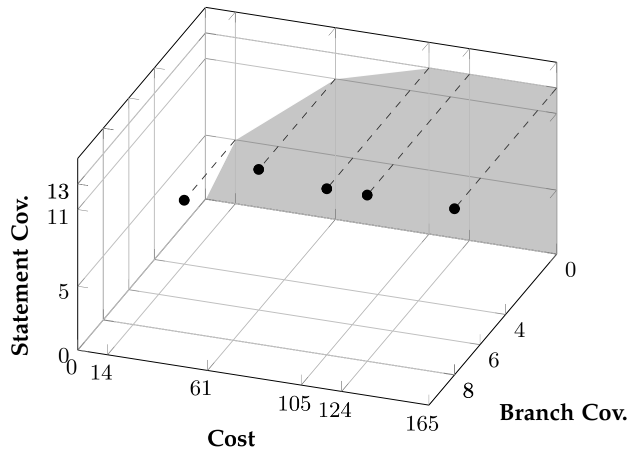
3.3 The Relationship between Hypervolume and AUC-based Metrics

The $I_{HP}(\tau)$ metric proposed in this paper can be viewed as a generalization of the AUC-based metrics (*e.g.,* APSC) used in prior work on search-based test case prioritization. For example, the APSC metric measures the average cumulative fraction of statements coverage as the Area Under Curve delimited by the test case ordering with respect to the cumulative statement coverage scores [?]. In light of the proposed hypervolume metric, APSC can be viewed as a simplified version of $I_{HP}(\tau)$ where all test cases have execution cost equal to one and only the statement coverage is considered as a testing criterion. A similar consideration can be made for all the other cumulative fitness functions used in previous work on search-based test case prioritization [?], [?], [?].

Finally, as explained in Section ??, despite the AUC metrics being strictly dependent on each other, they are calculated independently in test case prioritization based on multi-objective Genetic Algorithms. Indeed, these values are projections of a manifold of cumulative points (*e.g.,* a projection of a volume into two areas). For example, let us consider again the example of the test suite in Table ??. Figure ??-(a) depicts the cumulative coverage and cost scores for the prioritization $\tau = \langle t_5, t_3, t_4, t_2, t_1 \rangle$. Applying AUC-based metrics to assess the fitness of τ require us to compute two metrics, *i.e.,* APBC$_c$ and APSC$_c$. These metrics correspond to the grey areas in Figure ??, which correspond to the projections of the hypervolume on the geometric plane Statement-Cost and Branch-Cost. An important dif-



(a) Branch Coverage



(b) Statement Coverage

Fig. 2. Cumulative points in three-objective test case prioritization. The gray areas denote the Area Under Curve for the two projections of the cumulative score for the permutation in Table **??** onto planes [Cost × Branch Cov.] and [Cost × Statement Cov.].

ference between the AUC-based metrics (*e.g.,* APSC) and $I_{HP}(\tau)$ lies in how they measure the area dominated by a given test case permutation/ordering $P(\tau)$. The AUC-based metrics provide an over-estimation of the area dominated by $P(\tau)$ using the trapezoidal rule [**?**] (see Figure **??**). Instead, $I_{HP}(\tau)$ uses the rectangular rule, thus, strictly satisfying the definition of *dominance* in multi- and many-objective optimization (see Definition **??** and Figure **??**).

3.3.1 Supporting the decision making

Prior studies focused on AUC-based metrics in a multi-objective paradigm with the theoretical motivation that having multiple Pareto optimal solutions helps to accommodate for different views from decision-makers that may profit of the variants to refine their views during the solution evaluation process. In the following, we show, through an example, that choosing trade-offs in the space of the AUC-based metrics is different from choosing trade-offs among the original testing criteria being condensed in these metrics. To this aim, let us consider again the example of test suite shown in Table **??**. Let us suppose we are interested in finding a test case prioritization that optimizes three testing criteria: execution time, branch, and statement coverage. Let us also assume we used the AUC-based metrics and multi-objective optimization as suggested in prior studies. First, we notice that the three original testing criteria correspond to only two objectives: the cost-cognizant average percentage of branch coverage (APBC$_c$) and the cost-cognizant average percentage of statement coverage (APSC$_c$).

Remark 1: *in general, n testing criteria for the test case prioritization problem correspond to $n-1$ search objectives when using the cost-cognizant AUC-based metrics.*

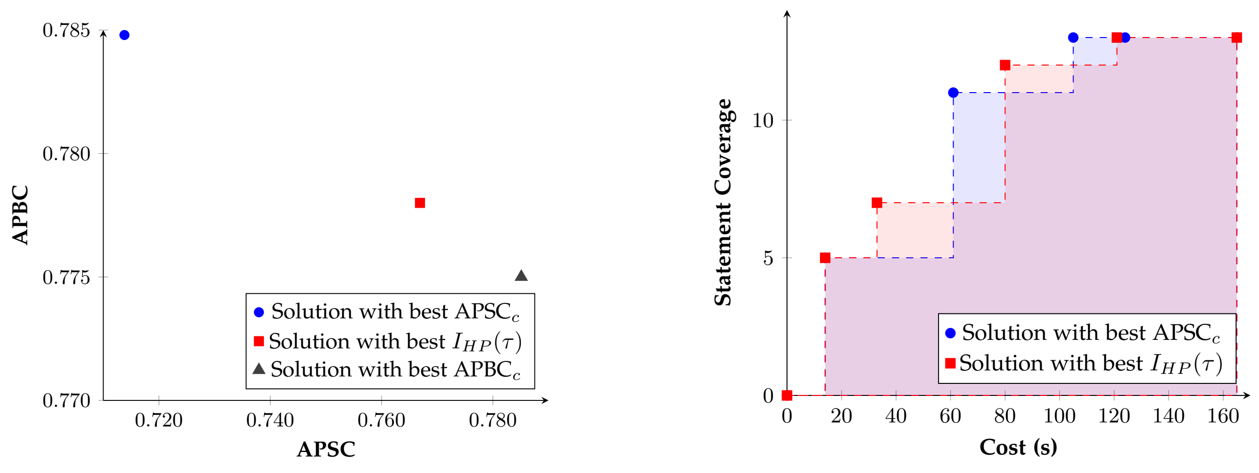
In our example, there are 120 possible permutations and we can use an exhaustive search to find the Pareto optimal ones. Among these possible test permutations, there are only three permutations that are Pareto optimal. The first two optimal solutions are $\tau_B = \{T_5, T_3, T_4, T_2, T_1\}$ and $\tau_S = \{T_5, T_2, T_1, T_3, T_4\}$. The former is the best permutation for APBC$_c$ while the latter is the best solution for APSC$_c$. Graphically, the two permutations τ_B and τ_S correspond to the two corners of the Pareto front as shown in Figure **??**-(a). The third Pareto optimal solution is the permutation $\tau_H = \{T_5, T_2, T_3, T_1, T_4\}$, which corresponds to the solutions with the largest hypervolume score.

Let us assume that the decision-maker wants to give higher priority to statement coverage over branch coverage and therefore he/she chooses the solution τ_S with the largest statement coverage rate. In theory, we may conclude that τ_S is indeed the best test prioritization for statement coverage. To refute this hypothesis, let us now look at the solution τ_S projected in the space of the original coverage criteria (for simplicity we consider only cost and statement coverage) rather than in the objective space (*i.e.,* the space of the AUC-metrics). Figure **??**-(b) depicts the statement coverage over execution cost achieved by τ_S compared to the solution with τ_H. As we can observe, τ_S achieves the maximum statement coverage earlier than τ_H (105s for the former compared to 121s for the latter). However, τ_H achieves better statement coverage than τ_S during the first 60s of test execution time. Similarly, the best permutation is τ_H if our goal is to reach higher statement coverage in 100s of test execution time.

Remark 2: *when not enough resources are available to run the entire test suite, choosing a solution among the trade-offs produced with AUC-based metrics can lead to suboptimal results.*

Furthermore, we notice that any test case permutation is by itself a set of trade-offs in the space of the original testing criteria. Indeed, τ_S corresponds to six points/trade-offs between execution cost and statement coverage as already shown in Figure **??**. Similarly, τ_H corresponds to six trade-offs in the space of the testing criteria. Decision-makers can, in theory, choose not only which permutation to select but also whether stopping the execution of the test suite earlier (if he/she has not enough resources and time to run the entire suite). In our example, analyzing the trade-offs in the space of execution cost and statement coverage depicted in Figure **??** provides better insights about the pros and the cons of the two permutations τ_S and τ_H when varying the amount of resources (time) we want to spend on regression testing.

Remark 3: *each test permutation is by definition a set of trade-offs in the space of the testing criteria.*



(a) Pareto optimal test permutation in the space of the AUC-based scores

(b) Test permutations in the space of the testing criteria

Fig. 3. Comparison of the Pareto optimal solutions (test prioritizations) for the test suite in Table **??** and the solution with the best I_{HP} score.

4 Evaluating the Hypervolume Genetic Algorithm with Less than Three Criteria

We conduct a first empirical study to assess the performances of HGA. In particular, we investigate the following high-level research question:

RQ$_1$: *What is the cost-effectiveness and efficiency of HGA, compared to state-of-the-art test case prioritization techniques?*

To better clarify it, we detailed it in two research questions:

- **RQ**$_{1.1}$: *What is the cost-effectiveness of HGA, compared to state-of-the-art test case prioritization techniques?* This research question aims at evaluating to what extent the test case ordering obtained by HGA is able to detect faults (*effectiveness*) earlier (lower execution *cost*) in comparison with three state-of-the-art techniques: a cost cognizant additional greedy algorithm [?], [?], a single objective genetic algorithm based on an AUC metric (GA) [?], and a multi-objective search based algorithm namely NSGA-II [?] used in prior test case prioritization [?], [?]. This reflects the developers' needs to discover regression faults with minimum cost.
- **RQ**$_{1.2}$: *What is the efficiency of HGA, compared to state-of-the-art test case prioritization techniques?* With this second research question, we are interested in comparing the running time (*efficiency*) required by HGA to find an optimal test ordering, in comparison with the three experimented test case prioritization techniques.

4.1 Study Design

This subsection describes the design of the study.

4.1.1 Context of the Study

The *context* consists of five GNU utilities —namely Bash, Flex, Grep, GZip, Sed— from the Software-artifact Infrastructure Repository (SIR) [?]. The characteristics of these five programs are reported in Table **??**, including their size (in terms of lines of code), test suite size, and type of faults. In total, the selected programs have a size ranging between $5,680$ and $59,846$ LOC, while the number of test cases varies between 214 and $1,061$. We selected these programs since they have been used in previous work on regression testing [?], [?], [?], [?], [?], [?]. Moreover, they have different size, number of tests, and context applications. As faults, we consider the seeded faults that are available in SIR. Please consider that, when seeding the faults, the authors of the repository assumed that the programmer that made the changes inserted the faults. Thus, the seeded faults can be located only within the changes between versions (calculated with the assistance of a diff tool)[1]. More specifically, SIR provides a list of seeded faults with the corresponding test-fault coverage information. In our study, we considered the non-trivial faults, *i.e.,* faults that can be exposed by a very few test cases, as suggested in the SIR guidelines [?]. For the sake of this analysis, we always selected the largest *hard* matrices (*i.e.,* matrices of faults that are killable by few tests) in case of multiple fault matrices available in the SIR repository.

4.1.2 Testing Criteria

To answer our research questions, we considered different testing criteria widely used in previous test case prioritization work [?], [?], [?]:

- **Statement coverage criterion.** We measured statement coverage achieved by each test case using gcov, a profiling tool that is part of the GNU C compiler (gcc).
- **Execution cost criterion.** To compute the execution cost, we could just measure the test case execution time. However, this measure depends on several external factors such as different hardware, application software, operating system, etc. In this paper, we addressed this issue by counting the number of executed instructions in the production code, instead of measuring the actual execution time. To this aim, we used gcov to measure the execution frequency of each source code instruction for the programs from the GNU. Notice that approximating the execution cost as the number

1. https://sir.unl.edu/content/c-fault-seeding.php

TABLE 3
Programs used in the study.

Program	Description	Version	LOC	# Tests	# Faults	Language	Fault Type
Bash	Shell Language Interpreter	V2	59,846	1,061	5	C	Seeded
Flex	Fast Lexical Analyzer	V2	10,459	567	15	C	Seeded
Grep	Regular Expression Utility	V2	10,068	809	10	C	Seeded
GZip	Compression Tool	V2	5,680	214	11	C	Seeded
Sed	Non Interactive Text Editor	V2	14,427	360	5	C	Seeded

of executed instructions is a standard procedure in the related literature [?], [?].

- **Past faults coverage criterion.** We considered the previous versions of the programs with seeded faults available in the SIR repository [?]. SIR also specifies whether or not each test case is able to reveal these faults. Such information can be used to assign a past faults coverage value to each test case, computed as the number of known past faults that each test is able to reveal in the previous version.

Notice that the goal of our analysis is not to determine which coverage criteria have the higher likelihood of revealing regression faults. Therefore, we selected those that have been widely used in prior studies (*e.g.*, [?], [?], [?], [?]). Nevertheless, it is possible to formulate other criteria by just providing a clear mapping between tests and coverage-based requirements. The criteria used in this study serve to illustrate how the Hypervolume-based metric can be applied to any number and kind of testing criteria to be satisfied, where further criteria just represent additional axes to be considered when computing $I_{HP}(\tau)$. Using the testing criteria described above, we examined two different formulations of the TCP problem:

- **Two-criteria (Single-objective).** The goal is to find an optimal ordering of test cases which (i) minimizes the *execution cost* and (ii) maximizes the *statement coverage*.
- **Three-criteria (Two-objective).** For this formulation, we considered the *past faults coverage* as a third criterion to be maximized.

4.1.3 Evaluated Algorithms

We compared HGA with three state-of-the-art algorithms, namely (i) Additional Greedy [?], [?], [?], (ii) GA [?], and (iii) NSGA-II [?], [?]. In particular, we compared HGA with Additional Greedy and GA in the single objective formulation (two criteria) and with Additional Greedy and NSGA-II in the two-objective formulation (three criteria).

Additional Greedy. This algorithm instantiated for the TCP problem [?], [?] considers coverage and cost at the same time by maximizing the coverage per unit of time of the selected test cases (cost cognizant additional greedy). Similarly, for what concerns the three-criteria formulation of the problem, we used the algorithm proposed by Yoo and Harman [?], [?], [?], which conflates code coverage, execution cost and past coverage in one objective function to minimize.

Additional Greedy is an iterative deterministic search algorithm that starts with an empty order of test cases $\tau_0 = \langle\rangle$; then, it selects the test case t_{max} having the highest value of code coverage per time unit (greedy step), *i.e.*, $\tau_1 = \langle t_{max}\rangle$. In each of the subsequent iterations, it selects the test case yielding the largest (additional) increment of code coverage per time unit compared to the order τ_i built in the last previous iteration of the algorithm. The loop ends when the highest coverage per time unit is reached, *i.e.*, when adding any un-selected test does not lead to an increment in coverage. To complete the test order, the un-prioritized test cases that do not contribute to the additional coverage could be ordered using any strategy (*e.g.*, using a random order). In this work, we recursively re-applied the Additional Greedy algorithm to the un-prioritized tests until all are ordered, as done in previous work [?].

When multiple coverage criteria are used (as for the three-criteria formulation), the additional coverage per unit time of each test t is computed using the following equation:

$$g(t) = \frac{1}{m} \times \frac{1}{cost(t)} \times \sum_{i=1}^{i=m} f_i(t) \tag{8}$$

where $F = \{f_1, \ldots, f_m\}$ is the set of coverage criteria to consider and $cost(t)$ denotes the execution cost of the test t.

Genetic Algorithm. Genetic Algorithms (GAs) represent a class of search techniques based on the natural selection processes defined by Darwin's theory of biological evolution. A typical GA procedure starts with an initial population P of individuals. Selected pairs of individuals are combined and mutated to generate new individuals that will be part of the population of the next generation. A GA is an approximated algorithm that does not guarantee to converge. For this reason, the search continues for a number of generations until a stop condition is reached. Individuals of the population are represented by their chromosome (*e.g.*, the sequences of their variables/parameters). We selected a GA because it is one of the best single-objective algorithms for the test case prioritization problem [?].

NSGA-II. The Non-dominated Sorting Genetic Algorithm II [?] is a computationally fast and elitist multi-objective evolutionary algorithm based on a non-dominated sorting approach. As any population-based evolutionary algorithms, NSGA-II starts with a set of solutions (test case orderings in our case) randomly generated within the solution space. At each generation, *offsprings* are generated by combining pairs of fittest individuals through three genetic operators: *selection*, *crossover* and *mutation*. To form the population for the next generation, parents and offsprings are ordered using the non-dominated sorting algorithm, which assigns to each candidate solution a fitness score that combines the *non-dominance relation* (see Equation **??**) and the *crowding distance*. The individuals are sorted and the fittest ones are selected to form the new population. The process is repeated until a maximum number of iterations (also called *generations*) is reached. We selected NSGA-II because it has been widely used in literature and for regression testing in particular [?], [?], [?]. Moreover, our choice was guided by the fact that NSGA-II has been proven to be particularly suited for prioritization problems [?], [?], [?].

When applying Genetic Algorithm and NSGA-II to the TCP problem [?], the objective functions to optimize are AUC-based metrics. Therefore, each coverage criterion is condensed with execution cost information by applying Equation **??**. This results in a single AUC-based metric per each coverage (+cost) criterion. For example, the AUC-based metric to optimize for statement coverage is the cost cognizant variant of Average Percentage of Statements Coverage (APSC$_c$):

$$\text{APSC}_c = \frac{\sum_{i=1}^{m}\left(\sum_{j=\text{TS}_i}^{n} c_j - \frac{1}{2}c_{\text{TS}_i}\right)}{\sum_{i=1}^{n} c_i \times m} \tag{9}$$

where $T = \{t_1, t_2, \ldots, t_n\}$ is the test suite to be optimized, with cost $C = \{c_1, c_2, \ldots, c_n\}$, TS$_i$ is the first test case in an ordering T' of T that is able to cover the statement i.

4.1.4 Implementation Details and Parameter Setting

All the algorithms have been implemented using *JMetal* [?], a Java-based framework for multi-objective optimization with meta-heuristics. To reduce the execution time needed to perform the experiments, we pre-processed the coverage data using the lossless *coverage compaction algorithm* proposed by Epitropakis *et al.* [?]. This technique improves the performance of all the algorithms reducing the size of the coverage matrices by a factor between 7 and 488 [?].

We used the default parameters values used in previous studies on TCP [?], [?]. This is because previous studies [?], [?] demonstrated that default values are a reasonable choice, even considering that parameter tuning is a long and expensive process that in the context of search-based software engineering does not assure better performances. In particular, we use the following (default) parameter values:

- **Population size**: 250 individuals.
- **Selection**: *binary tournament selection*. It randomly picks two individuals for the tournament and selects the fittest one. The winner of each tournament is the solution with the best $I_{HP(\tau)}$ (Equation **??**) in HGA or the permutation with the best AUC-based metric for GA. For NSGA-II, the winner of the tournament is the test case with the best *non-dominance rank*, or with the highest *crowding distance* at the same level of *non-dominance rank*.
- **Crossover**: *PMX-Crossover* with crossover probability of $p_c = 0.90$. This operator swaps elements at a given random crossover point.
- **Mutation**: *SWAP-Mutation* that randomly swaps two chosen permutation elements within each offspring with a mutation probability of $p_m = 1/n$, where n is the number of test cases.
- **Stopping criterion**: 100 generations, corresponding to $25,000$ fitness evaluations.

To account for the inherently random nature of search-based algorithms [?], we performed 30 independent runs for each program and for each search algorithm in our study.

4.1.5 Evaluation Metrics

To address **RQ**$_{1.1}$ we used the *cost-cognizant Average Percentage of Faults Detected* metric (APFD$_c$) proposed by Elbaum *et al.* [?]. This metric measures the ability of a test permutation to reveal faults earlier [?]. The larger the APFD$_c$, the lower the average cost needed to detect the same number of faults. Since we performed 30 independent runs, we report the mean and the standard deviation of the APFD$_c$ scores achieved for each program and for each formulation. It is worth noting that for NSGA-II we report the mean and the standard deviation of all the solutions in the Pareto set. The *cost-cognizant Average Percentage of Faults Detected per unit cost* can be computed as follows:

$$\text{APFD}_c = \frac{\sum_{i=1}^{m}\left(\sum_{j=\text{TF}_i}^{n} c_j - \frac{1}{2}c_{\text{TF}_i}\right)}{\sum_{i=1}^{n} c_i \times m} \tag{10}$$

where $T = \{t_1, t_2, \ldots, t_n\}$ is the test suite to be optimized, with cost $C = \{c_1, c_2, \ldots, c_n\}$ and TF$_i$ is the first test case in an ordering T' of T that reveals fault i.

To address **RQ**$_{1.2}$, we compared the average running time required by each algorithm to converge. The execution time was measured using a machine with Intel Core i7 processor running at 2.40GHz with 12GB RAM.

We statistically analyzed the results, to check whether the differences between the APFD$_c$ scores (or the running time) are statistically significant or not. To this aim, we used two different statistical tests: (i) *Welch's t-test*, and (ii) *Wilcoxon t-test* [?]. In particular, we used the *Welch's t-test* to compare HGA with Additional Greedy because the distributions of the two groups have unequal variance. Instead, we applied the *Wilcoxon t-test* when comparing HGA with GA and NSGA-II. In both cases, we considered a p-value threshold of 0.05. Significant p-values indicate that the corresponding null hypothesis can be rejected in favor of the alternative ones. Other than testing the null hypothesis, we used the Vargha-Delaney ($\hat{A}_{12}$) statistical test [?] to measure the effect size. $\hat{A}_{12} > 0.5$ indicates the distribution by HGA is larger than the distribution by a state-of-the-art algorithm; $\hat{A}_{12} < 0.5$ means the opposite; and $\hat{A}_{12} = 0.5$ means they are equal. For **RQ1**, $\hat{A}_{12} > 0.5$ is in favor of HGA while $\hat{A}_{12} < 0.5$ are preferable for **RQ2**.

4.2 Results of the empirical study

This subsection discusses the results of our first study, thus, answering the research questions.

4.2.1 Results for Two-criteria (Single objective) formulation

Table **??** reports the APFD$_c$ values and the running time obtained by HGA and the state-of-the-art algorithms on the five programs from the Software-artifact Infrastructure Repository (SIR) [?].

Results for RQ$_{1.1}$. From the comparison between HGA and Additional Greedy, we observe that the former achieves statistically higher APFD$_c$ scores than the latter in four out of five programs (*i.e.,* $\hat{A}_{12} > 0.5$ and p-value < 0.05). Moreover, the $\hat{A}_{12}$ statistics reveal that in all these cases the effect size is large. The improvements range between a minimum of +0.60% and a maximum of +41.10% achieved for Sed and GZip, respectively.

TABLE 4
Results for two-criteria (single objective) formulation: APFD$_c$ and running time achieved by Additional Greedy, GA, and HGA. For each baseline, in parenthesis is shown the median difference with respect to HGA. Results are highlighted with ↓ when one algorithm is statistically worse than HGA; ↑ when the opposite is true.

Program	**Add. Greedy**	**GA**		**HGA**		**HGA $\neq$ Add. Greedy**			**HGA $\neq$ GA**		
		Median	**St. Dev.**	**Median**	**St. Dev.**	**p-value**	**$\hat{A}_{12}$**	**Effect Size**	**p-value**	**$\hat{A}_{12}$**	**Effect Size**
Bash	0.948 (+0.021) ↑	0.920 (-0.007)	0.040	0.927	0.036	$<$ **0.01**	**0.23**	**Large**	0.51	0.55	Negligible
Flex	0.453 (-0.245) ↓	0.699 (+0.001)	0.001	0.698	0.001	$<$ **0.01**	**1.00**	**Large**	0.45	0.44	Negligible
Grep	0.476 (-0.010) ↓	0.485 (-0.004)	0.011	0.489	0.009	$<$ **0.01**	**0.93**	**Large**	0.11	0.62	Small
GZip	0.119 (-0.416) ↓	0.602 (-)	0.116	0.602	0.108	$<$ **0.01**	**1.00**	**Large**	0.20	0.43	Negligible
Sed	0.989 (-0.006) ↓	0.994 (-0.001)	0.001	0.995	0.001	$<$ **0.01**	**1.00**	**Large**	0.26	0.58	Small

Program	**Add. Greedy**	**GA**		**HGA**		**HGA $\neq$ Add. Greedy**			**HGA $\neq$ GA**		
		Mean	**St. Dev.**	**Mean**	**St. Dev.**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**
Bash	2s ↑	25s ↓	1s	17s	2s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
Flex	1s ↑	10s ↓	<1s	5s	1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
Grep	1s ↑	16s ↓	1s	5s	<1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
GZip	<1s ↑	1s ↓	<1s	<1s	<1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
Sed	<1s ↑	3s ↓	<1s	1s	<1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**

Instead, `Additional Greedy` produced a significantly higher APFD$_c$ score for `Bash` although the difference is quite small: -2.10% on average. `Bash` is a particular program since all algorithms almost achieve an optimal APFD$_c$, which is very close to one. This is due to the fact that the fault-revealing tests are always run early by the solutions generated by the search algorithms despite the very large size of the test suite. Regression faults in this program can be detected by only few test cases (1.5% of tests on average), which have, however, very large statement coverage and therefore are selected very early. Therefore, while the difference between `Additional Greedy` and `HGA` are statistically significant, they are negligible in practice as APFD$_c$ are very close to being optimal for both the two algorithms.

When comparing `HGA` with `GA`, we notice that in none of the programs we can reject the null hypothesis for the *Wilcoxon test*. However, the Vargha-Delaney ($\hat{A}_{12}$) tests reveal that, although not significant, `HGA` is better than `GA` with a `small` effect size in two programs. This means that `HGA` is able to produce test permutations that are competitive with those generated by `GA`. These results are expected since, as explained in Section **??**, the hypervolume and the AUC-based metric are equivalent. Their difference lies in how the area under the curve is computed: using the rectangular rule for `HGA` and the trapezoidal rule for `GA`.

Results for RQ$_{1.2}$. The comparison between `HGA` and `Additional Greedy`, shows that on in all the programs considered from the SIR repository the `Additional Greedy` algorithm is statistically faster (*i.e.,* $\hat{A}_{12} > 0.5$ and p-value < 0.05) than `HGA`, despite being less cost-effective as demonstrated in **RQ$_{1.1}$**. In all these cases, according to the $\hat{A}_{12}$ statistics the effect size is `large`. The improvements range between a minimum of 2 times and a maximum of 8.5 times achieved for `GZip` and `Bash`, respectively. It is worth noting that the performance of `Additional Greedy` is strongly influenced by the number of test cases. Indeed, in these programs, the number of test cases ranges between 214 for `GZip` and $1,061$ for `Bash`.

To verify whether the (positive and negative) differences between the execution time of the two algorithms significantly interact with the test suite size, we applied the *permutation test* [**?**]. It corresponds to a non-parametric version of the Analysis of Variance (ANOVA) test and, thus, it does not require that the distributions under analysis are normally distributed. For the test, we used the implementation available in `R`, and its package `lmPerm` in particular, with a large number of iterations (10^8) to have stable results [**?**]. The permutation test revealed that there is a statistically significant interaction between the execution time of the two algorithms and the number of the test cases to prioritize (p-value=4.14×10^{-4}). In other words, the larger the test suite, the more time `Additional Greedy` needs in terms of execution time.

From the comparison between `GA` and `HGA`, we can notice that for all the programs we can reject the null hypothesis of the *Wilcoxon t-test*. Looking at the Vargha-Delaney ($\hat{A}_{12}$) statistics, in all programs, `HGA` outperforms (is more efficient than) `GA` with `large` effect size. Indeed, `GA` requires between 2.00 (*e.g.,* `GZip`) to 3.20 times (*e.g.,* `Grep`) the execution times required for `HGA`. On average `HGA` is 1.89 times faster than `GA`. As we already noticed in the comparison with `Additional Greedy`, the number of test cases strongly influences the performance of `GA`. Indeed, as the number of test cases increases the ratio between the time required by `GA` and `HGA` increases. These observations are also confirmed by the permutation test: the differences (improvements/worsening) between the execution time of `HGA` and `GA` significantly interacts with the test suite size (p-value=2.90×10^{-5}).

4.2.2 Results for Three-criteria (Two-objective) formulation

Table **??** reports the APFD$_c$ values and the running time obtained by `HGA`, `Additional Greedy`, and `NSGA-II`.

Results for RQ$_{1.1}$. We observe that `HGA` outperforms `Additional Greedy` in four out of five programs with a `large` effect size. `HGA` improves APFD$_c$ values up to +48.40% with respect to `Additional Greedy`, while in the opposite case the difference is low (*e.g.,* -2.00% on Bash). Looking at the results obtained when comparing `NSGA-II` with `HGA`, we notice that in four cases out of five we can reject the null hypothesis. In three of those cases, `HGA` outperforms `NSGA-II`: in two cases with a `small` effect size and in another one with a `medium` effect size.

Only for `Bash`, `Additional Greedy` produces a better test permutation with respect to `HGA` with a `medium` effect

TABLE 5
Results for three-criteria (two-objective) formulation: APFD$_c$ and running time achieved by Additional Greedy, NSGA-II, and HGA. For each baseline, in parenthesis is shown the median difference with respect to HGA. Results are highlighted with ↓ when one algorithm is statistically worse than HGA; ↑ when the opposite is true.

Program	Add. Greedy	NSGA-II			HGA		HGA ≠ Add. Greedy			HGA ≠ NSGA-II		
		# Sol.	Median	St. Dev.	Median	St. Dev.	p-value	$\hat{A}_{12}$	Effect Size	p-value	$\hat{A}_{12}$	Effect Size
Bash	0.948 (+0.020) ↑	12	0.921 (-0.007)	0.033	0.928	0.033	**< 0.01**	**0.27**	**Medium**	0.32	0.55	Negligible
Flex	0.453 (-0.246) ↓	13	0.698 (-0.001) ↓	0.004	0.699	0.001	**< 0.01**	**1.00**	**Large**	**< 0.01**	**0.69**	**Medium**
Grep	0.476 (-0.014) ↓	9	0.486 (-0.004) ↓	0.009	0.490	0.008	**< 0.01**	**0.87**	**Large**	**0.04**	**0.61**	**Small**
GZip	0.118 (-0.484) ↓	50	0.405 (-0.197) ↓	0.131	0.602	0.081	**< 0.01**	**1.00**	**Large**	**0.05**	**0.61**	**Small**
Sed	0.989 (-0.004) ↓	14	0.994 (+0.001) ↑	0.001	0.993	0.001	**< 0.01**	**1.00**	**Large**	**< 0.01**	**0.22**	**Large**

Program	Add. Greedy	NSGA-II		HGA		HGA ≠ Add. Greedy			HGA ≠ NSGA-II		
		Mean	St. Dev.	Mean	St. Dev.	p-value	$\hat{A}_{12}$	Magnitude	p-value	$\hat{A}_{12}$	Magnitude
Bash	2s ↑	21s ↓	2s	15s	1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
Flex	<1s ↑	9s ↓	<1s	4s	<1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
Grep	1s ↑	14s ↓	1s	6s	1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**
GZip	<1s ↑	1s ↓	<1s	<1s	<1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.02**	**Large**
Sed	<1s ↑	3s ↓	<1s	1s	<1s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.00**	**Large**

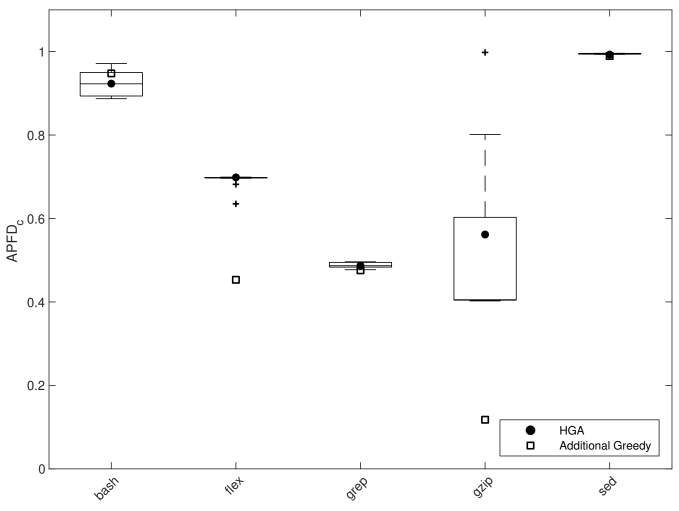


Fig. 4. APFD$_c$ scores achieved by Additional Greedy (□), NSGA-II (boxplots), and HGA (•) on the three-criteria (two-objective) formulation of the TCP problem.

size. Furthermore, there is also only one program (*i.e.*, `Sed`), in which `NSGA-II` is better than `HGA` with `large` effect size. Both these two programs are characterized by very large APFD$_c$ scores for all search algorithms (*i.e.*, APFD$_c$ >0.90). To shed light on these close-to-being-optimal results, we manually investigated the permutations generated by the algorithms. While the regression faults are non-trivial for both the two programs —they are detectable by 1.5% of tests on `Bash` and 16% of tests in case of `Sed`— all fault-revealing test case have very high statement coverage and are, therefore, always selected very early. The differences in APFD$_c$ scores between the three algorithms are due to very few test cases that differ in the corresponding test permutations. Given the fact that all algorithms achieve very high APFD$_c$ scores, these differences are negligible in practice although statistical significant.

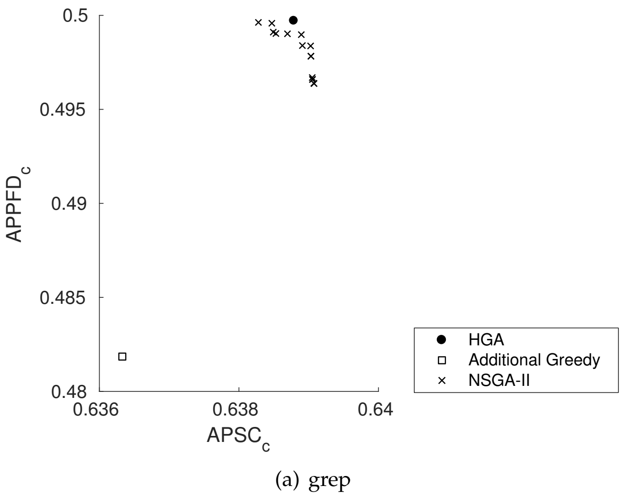
Notice that for the comparison above we considered all Pareto-optimal solutions produced by `NSGA-II` (between nine and 50 solutions). However, different solutions in the Pareto fronts may provide different APFD$_c$ scores. Figure **??** compares the APFD$_c$ scores by HGA (single points) with the boxplots of `NSGA-II` (*i.e.*, the distributions of APFD$_c$ scores of the entire Pareto front). The purpose of this comparison is two-fold: (i) we want to measure whether the majority of the solutions by `NSGA-II` are better than the single solution by HGA; and (ii) we want to measure the variability of the APFD$_c$ values by `NSGA-II`.

As we can observe, in four systems, the single solution provided by `HGA` is better or equal to the median solution of `NSGA-II`. For `GZip`, we observe a huge variation in the APFD$_c$ distribution yielded by `NSGA-II`: it ranges between 0.40 and 1.00, with a median value of 0.400. For this project, choosing a proper solution from the Pareto front is very critical since not all its solutions have better APFD$_c$ scores than `HGA`. In particular, only 26% of the Pareto front is more cost-effective than the single solution achieved by `HGA`. Once again, no guideline exists that helps the testers choosing the most cost-effective solutions in the Pareto front as the APFD$_c$ scores can be computed only by executing all test permutations.

To better understand how the three algorithms optimize the selected testing criteria, Figure **??** plots —for `Grep` and `Sed`— the Pareto front produced by `NSGA-II`, and the single solutions generated by `Additional Greedy` and `HGA` with respect to the objectives optimized by `NSGA-II` (APSC$_c$ and APPFD$_c$). The complete set of plots for all the programs in our study is available in our online appendix [**?**]. This comparison allows understanding whether the solutions generated by one algorithm dominate (*i.e.*, are better than) the solutions produced by an alternative algorithm in the space of the AUC-metrics. It is possible to notice that in all the cases the solutions generated by `NSGA-II` and `HGA` always dominate the solutions produced by `Additional Greedy`.

Furthermore, we observe that the single solution yielded by `HGA` is never dominated by the Pareto front generated by `NSGA-II`. Vice versa, for `Grep`, the single solution by `HGA` dominates the majority of the Pareto fronts produced by `NSGA-II`. These results are very unexpected considering that `NSGA-II` explicitly optimizes APSC$_c$ and APPFD$_c$ as two contrasting objectives. Instead, `HGA` optimizes the hypervolume indicator, which generalizes and combines APSC$_c$ and APPFD$_c$ as discussed in Section **??**.

This program demonstrates that optimizing the AUC-based metrics is not always directly related to having a better hypervolume score and neither a better fault detection



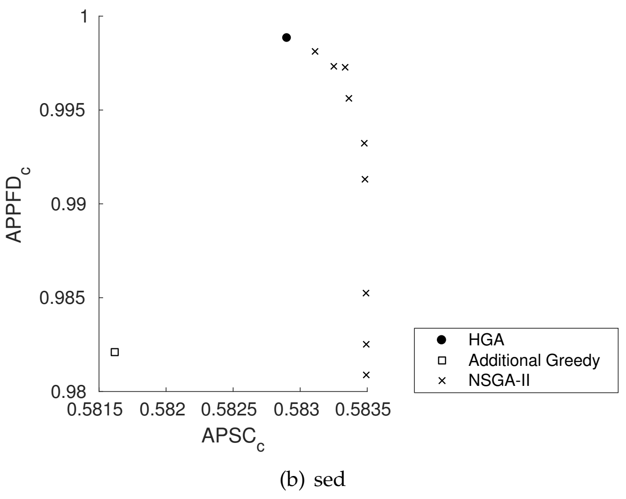


Fig. 5. Pareto frontiers achieved for the three-criteria (two-objective) formulation of TCP.

capability. Considering that there are no guidelines to select the best solution (permutation) from the Pareto front, this result poses a question on whether it is worth at all to use multi-objective algorithms for TCP.

Different observations can be made for Sed. More specifically, the single solution by `HGA` is one corner point of the Pareto front produced by `NSGA-II`. This means that none of the two algorithms dominated the other for this program. However, the solution by `HGA` corresponds to the Pareto optimal solution with the best APPFD$_c$ value (*i.e.*, the one with the highest rate of past fault coverage). Since the number of past faults is usually smaller than the number of statements to cover, the hypervolume metric may prefer solutions that overfit the past faults rather than optimizing the overall structural coverage. This remark may explain why the solutions by `NSGA-II` have higher fault detection capability than `HGA`. Clearly, investigating different weighting strategies for past-fault coverage in the computation of the hypervolume is part of our future agenda.

Results for RQ$_{1.2}$. For what regards the running time, the results for the three-criteria formulation are in line with those achieved for the two-criteria formulation. Indeed, `Additional Greedy` is more efficient than `HGA` in all the programs with a `large` effect size. Moreover, `HGA` is always statistically more efficient than `NSGA-II` with a `large` effect size. The differences between the two meta-heuristics are due to: (i) the efficient algorithm for the hypervolume computation in `HGA` and (ii) the different cost of their selection procedures as explained in Section **??**. Namely, the non-dominated sorting in `NSGA-II` is more expensive than the environmental selection implemented `HGA` for single-objective algorithms.

Summary for RQ$_1$. `HGA` outperforms `Additional Greedy` in most of the cases in terms of cost-effectiveness but it is less efficient. On the two-criteria formulation, as expected from the theory, `HGA` and `GA` are equivalent in terms of fault detection capability. However, the former is more efficient than the latter thanks to our algorithm for the fast computation of the hypervolume metric. On the three-criteria formulation, `HGA` is often more effective and always more efficient than `NSGA-II`.

5 Evaluating the Hypervolume Genetic Algorithm with up to Five Criteria

This section discusses the second empirical study we carried out to assess the performances of `HGA` compared to state-of-the-art many-objective algorithms when handling up to five testing criteria. Thus, we formulated the additional following high-level research questions:

RQ$_2$*: How does HGA perform with respect to many-objective test case prioritization techniques?*

To better clarify it, we detailed it in two research questions:

- **RQ$_{2.1}$**: *What is the cost-effectiveness of HGA, compared to many-objective test case prioritization techniques?* This research question aims at evaluating the selective pressure of `HGA`, that is "the degree to which the better individuals are favored during the computation" [**?**]. In particular, similarly to RQ$_{1.1}$, it analyses to what extent the test case ordering obtained by `HGA` is able to detect faults (*effectiveness*) earlier (lower execution *cost*) in comparison to two state-of-the-art many-objective algorithms, namely `GDE3` and `MOEA/D-DE`.
- **RQ$_{2.2}$**: *What is the efficiency of HGA, compared to many-objective test case prioritization techniques?* Similarly to RQ$_{1.2}$, with this research question, we are interested, in comparing the running time (*efficiency*) required by `HGA` to find an optimal test ordering compared to the alternative many-objective algorithms in cases that require a strong selective pressure.

5.1 Study Design

5.1.1 Context of the Study and Testing Criteria

The context of the study is the same as that of the first empirical study in Section **??**. For the testing criteria, we added two additional criteria with respect to the first study, namely branch and function coverage criteria. In particular, we considered the following criteria:

- **Branch and function coverage criterion.** Also, in this case, we measured statement coverage achieved by each test case using the `gcov` tool part of the GNU C compiler (`gcc`).

With these additional testing criteria, we examined two different many-objective formulations of the TCP problem:

- **Four-criteria (Three-objective).** The goal is to find an optimal ordering of test cases which (i) minimizes the execution cost, (ii) maximizes the statement coverage, (iii) maximizes the past faults coverage, and (iv) maximizes the branch coverage. We applied this formulation for the ten programs from SIR [**?**].
- **Five-criteria (Four-objective).** For this formulation, we considered the *function coverage* as a fifth criterion to be maximized. We applied this formulation on the same programs already used for the four-objective formulation.

5.1.2 Evaluated Algorithms and Parameter Setting

We compared the results of `HGA` with the those achieved by two algorithms, namely (i) `GDE3` [**?**] and (ii) `MOEA/D-DE` [**?**]. The former implements a diversity-based mechanism to address the problem of *selective resistance*, while the latter is a reference-point based mechanism to guarantee well distributed Pareto fronts for many-objective problems. These two algorithms inspired many other many-objective meta-heuristics and are representative for the two classes of algorithms discussed in section **??**. As already highlighted in the first study in Section **??**, it is worth noting that for `GDE3` and `MOEA/D-DE` the objective functions to optimize are AUC-based metrics.

For `GDE3` and `MOEA/D-DE`, we used their implementation available in *JMetal* [**?**] and preprocessed the coverage data using the lossless *coverage compaction algorithm* proposed by Epitropakis *et al.* [**?**]. For both algorithms, we used their default parameters values [**?**], [**?**]:

- **Population size**: 250 individuals as for `HGA`.
- **Selection**: For `GDE3` and `MOEA-D/DE`, the fittest individuals are selected using the *differential evolution selection operator*.
- **Crossover**: we used the *PMX-Crossover* with crossover probability $p_c = 0.90$ `GDE3` and `MOEA-D/DE` need also to set another parameter, namely CR. This parameter indicates how single sub-problems are separable (*i.e.,* the lower the value, the more the problems are separable). We applied the default values (*e.g.,* 0.50 for `GDE3` and 1.00 `MOEA-D/DE`).
- **Mutation**: as mutation operator, we used the *SWAP-Mutation* with permutation probability $p_m = 1/n$, where n is the number of test cases, *i.e.,* the same operator used with the same probability used in the previous study. `GDE3` and `MOEA-D/DE` need an additional parameter F. This scaling factor controls the speed and robustness of the search (*i.e.,* with a lower value the algorithm converges faster, but it has a higher risk of stacking in a local optimum). Also, in this case, we applied the default value (*i.e.,* 0.50).
- **Stopping criterion**: the evolutionary algorithms end when reaching 100 generations, corresponding to $25,000$ fitness evaluations.

We used default parameters considering that previous studies [**?**], [**?**] demonstrated that they are a reasonable choice, even considering that parameter tuning is a long and expensive process that in the context of search-based software engineering does not assure better performance.

To account for the inherently random nature of search-based algorithms [**?**], we performed 30 independents runs for each program and for each search algorithm in our study.

5.1.3 Evaluation Metrics

We used the same evaluation metrics used in Section **??**. In particular, for $\mathbf{RQ}_{2.1}$ we relied on the same evaluation metrics used for $\mathbf{RQ}_{1.1}$, while for $\mathbf{RQ}_{2.2}$ we relied on the same evaluation metrics used for $\mathbf{RQ}_{1.2}$.

5.2 Results of the empirical study

This section discusses the results of our second study, thus, answering the research questions.

5.2.1 Results for Four-criteria (Three-objective) formulation

Table **??** reports the APFD$_c$ values and the running time obtained by `HGA` and the state-of-the-art algorithms for the five programs from the Software-artifact Infrastructure Repository (SIR) [**?**].

Results for $\mathbf{RQ}_{2.1}$. We observe that `HGA` achieves equal or better APFD$_c$ values with respect to `GDE3` and `MOEA-D/DE`. In particular, the *Wilcoxon t-test* revealed that in three out of five programs the differences between `HGA` and `GDE3` are statistically significant (in two cases with `large` effect size and in one case with `large` effect size). Notice that `GDE3` returns many solutions (test permutations), whose number ranges between 20 (*e.g.,* `GZip`) and 38 (*e.g.,* `Flex`). Figure **??** shows that the APFD$_c$ scores achieved by `GDE3` may vary across the solutions in the Pareto fronts. In all the programs, the solution by `HGA` is more or equally cost-effective than the median score yielded by all the solutions by `GDE3`. For example, on `Bash` the APFD$_c$ scores by `GDE3` vary between 0.65 and 0.98, with a median value of 0.875. On this project, the solution by `HGA` outperforms 70% of the Pareto optimal solutions obtained with `GDE3`.

When comparing `HGA` with `MOEA-D/DE` we notice that in three out of five cases the null hypothesis cannot be rejected according to the *Wilcoxon t-test*. For the remaining programs, `HGA` is better than `MOEA-D/DE` (in one case with `large` effect size and in the other one with `small` effect size). Moreover, not all Pareto efficient solutions yielded by `MOEA-D/DE` achieve the same APFD$_c$ scores as shown in Figure **??**. One exemplary case is observable for `GZip`: for this project, the APFD$_c$ scores of the Pareto optimal solutions by `MOEA-D/DE` range between 0.40 and 1.00, with a mean value of 0.602. Instead, the single solution by `HGA` is better than 75% of Pareto optimal solutions generated by `MOEA-D/DE`.

TABLE 6
Results for four-criteria (three-objective) formulation: APFD$_c$ and running time achieved by MOEA-D/DE, GDE3, and HGA. For each baseline, in parenthesis is shown the median difference with respect to HGA. Results are highlighted with $\downarrow$ when one algorithm is statistically worse than HGA; $\uparrow$ when the opposite is true.

Program	GDE3			MOEA/D-DE			HGA		HGA $\neq$ GDE3			HGA $\neq$ MOEA-D/DE		
	# Sol.	Median	St. Dev.	# Sol.	Median	St. Dev.	Median	St. Dev.	p-value	$\hat{A}_{12}$	Effect Size	p-value	$\hat{A}_{12}$	Effect Size
Bash	22	0.875 (-0.027) $\downarrow$	0.057	250	0.913 (+0.011)	0.033	0.902	0.045	**< 0.01**	**0.66**	**Small**	0.14	0.42	Small
Flex	38	0.690 (-0.008) $\downarrow$	0.017	250	0.698 (-)	0.003	0.698	0.001	**< 0.01**	**0.96**	**Large**	0.11	0.41	Small
Grep	25	0.486 (-0.001)	0.001	250	0.486 (-0.001)	0.009	0.487	0.011	0.86	0.49	Negligible	0.62	0.47	Negligible
GZip	20	0.602 (-)	0.129	250	0.602 (-) $\downarrow$	0.100	0.602	0.122	0.25	0.56	Negligible	**0.02**	**0.62**	**Small**
Sed	32	0.984 (-0.009) $\downarrow$	0.012	250	0.992 (-0.001) $\downarrow$	0.002	0.993	0.001	**< 0.01**	**0.95**	**Large**	**< 0.01**	**0.78**	**Large**

Program	GDE3		MOEA-D/DE		HGA		HGA $\neq$ GDE3			HGA $\neq$ MOEA/D-DE		
	Mean	St. Dev.	Mean	St.Dev	Mean	St. Dev.	p-value	$\hat{A}_{12}$	Magnitude	p-value	$\hat{A}_{12}$	Magnitude
Bash	1min 25s $\downarrow$	6s	1min 7s $\downarrow$	4s	51s	5s	**<0.01**	**0.00**	**Large**	**<0.01**	**0.00**	**Large**
Flex	16s $\downarrow$	1s	13s $\downarrow$	1s	7s	1s	**<0.01**	**0.00**	**Large**	**<0.01**	**0.00**	**Large**
Grep	37s $\downarrow$	2s	32s $\downarrow$	1s	19s	1s	**<0.01**	**0.00**	**Large**	**<0.01**	**0.00**	**Large**
GZip	1s $\downarrow$	<1s	1s $\downarrow$	<1s	<1s	<1s	**<0.01**	**0.00**	**Large**	**<0.01**	**0.00**	**Large**
Sed	6s $\downarrow$	<1s	6s $\downarrow$	<1s	2s	<1s	**<0.01**	**0.00**	**Large**	**<0.01**	**0.00**	**Large**

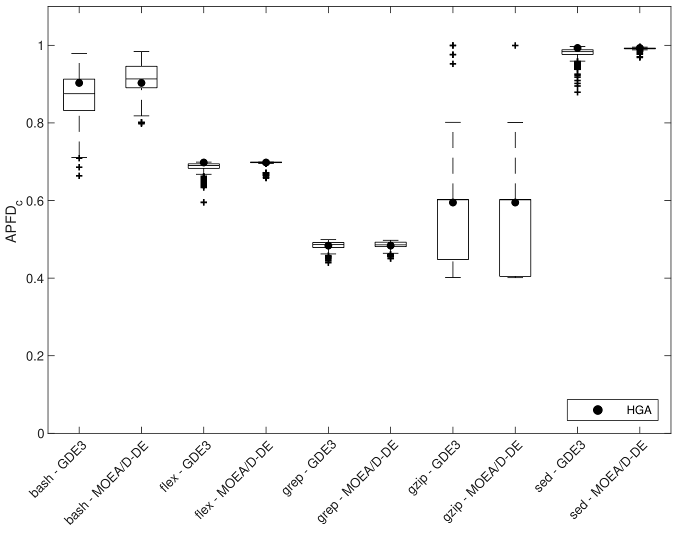


Fig. 6. APFD$_c$ scores achieved by GDE3 (boxplots), MOEA/D-DE (boxplots), and HGA (●) on the four-criteria (three-objective) formulation of the TCP problem.

Figure **??** plots —for `Flex` and `Sed`— the Pareto fronts produced by `GDE3` and `MOEA-D/DE` as well as the single solutions generated by `HGA` with respect to the AUC-based metrics. As we can observe, the solutions by `GDE3` and `MOEA-D/DE` do not dominate the solution generated by `HGA`. Vice versa, the solution generated by `HGA` dominates a large portion of the solutions by these algorithms. Using the hypervolume indicator as fitness function, `HGA` is able to optimize the AUC-based metrics even if the baselines use such metrics as objectives to optimize. This further poses the question of whether it is worth using multi-objective or many-objective algorithms for the test case prioritization problem, given the difficulty to discriminate the best solution among those produced by these algorithms. The complete set of plots of all the programs is available in our online appendix. [**?**].

Results for RQ$_{2.2}$. In all the programs, the `HGA` is statistically faster (*i.e.,* $\hat{A}_{12} < 0.5$ and p-value < 0.05) than the `GDE3` with `large` effect size. The improvements range between a minimum of 1.65 times and a maximum of 3.00 times, achieved on Bash and Sed, respectively. On average, `HGA` is 1.81 times faster than `GDE3`. Similarly to the results achieved for `NSGA-II`, the number of test cases strongly influences the performance of `GDE3`. Indeed, the ratio between the execution time needed by `GDE3` and the execution time required by `HGA` increases as the number of test cases grows. This insight is further confirmed by the permutation test: the differences (improvements/worsening) between the execution time of `HGA` and `GDE3` significantly interacts with the test suite size (p-value=7.9×10^{-5}).

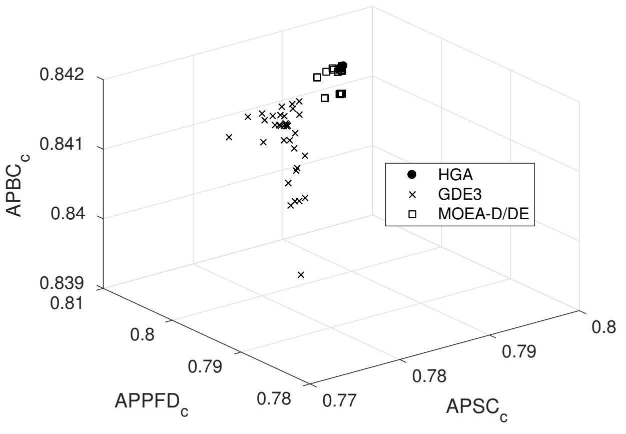
These results are confirmed when comparing `HGA` with `MOEA-D/DE`. Indeed, for all the programs we can reject the null hypothesis for the *Wilcoxon t-test* with `large` effect size. `MOEA-D/DE` requires between 1.30 (*e.g.,* `Bash`) to 3 times (*e.g.,* `Sed`) the execution times required for `HGA`. On average `HGA` is 1.49 times faster than `MOEA-D/DE`. To assess the interaction between the number of test cases and the performance gap between `HGA` and `MOEA-D/DE`, we performed the permutation test achieving a p-value equal to 2.20×10^{-16}.

By comparing the running time of `HGA` across the different programs, we can notice that `HGA` took more time to converge on `Bash` (51s) and `Grep` (19s) with respect to the other programs in our study. The computation cost of HGA is polynomial to (i) the number of criteria, (ii) to the population size, (iii) the cost of computing the hypervolume metric. While the number of criteria and the population size is the same for all programs, the cost of computing the hypervolume metric varies. Indeed, the cost of computing such a metric depends on two factors: (1) the test suite size (i.e., the total number of test cases) and (2) the percentage of test cases in the test suite required to reach the maximum coverage. Compared to the other programs, `Bash` and `Grep` have the largest test suites in our study. Furthermore, reaching the maximum coverage scores requires to run almost all test cases in their suites. In such a scenario, the algorithm for computing the hypervolume metric (Algorithm **??**) performs a large number of iterations (equal to the number of test cases needed to reach the maximum coverage score.

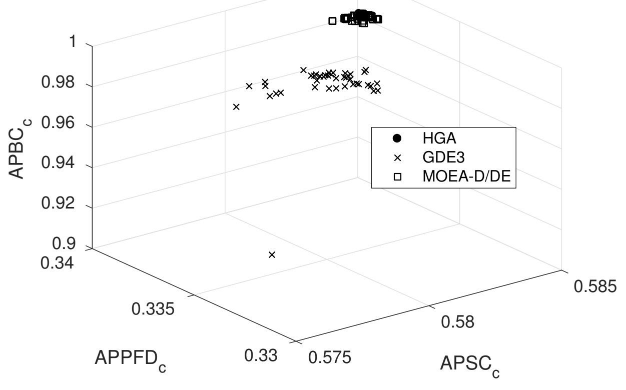
5.2.2 *Results for Five-criteria (Four-objective) formulation*

Table **??** reports the APFD$_c$ values and the running time obtained by `HGA` and the state-of-the-art algorithms for the five programs from the Software-artifact Infrastructure Repository (SIR) [**?**].

Results for RQ$_{2.1}$. The results are very similar to those achieved in the four-criteria formulation. Indeed, when



(a) flex



(b) sed

Fig. 7. Pareto frontiers achieved for the four-criteria (three-objective) formulation of TCP.

TABLE 7
Results for five-criteria (four-objective) formulation: APFD$_c$ and running time achieved by MOEA-D/DE, GDE3, and HGA. For each baseline, in parenthesis is shown the mean difference with respect to HGA. Results are highlighted with ↓ when one algorithm is statistically worse than HGA; ↑ when the opposite is true.

Program	GDE3			MOEA/D-DE			HGA		HGA ≠ GDE3			HGA ≠ MOEA-D/DE		
	# Sol.	Median	St. Dev.	# Sol.	Median	St. Dev.	Median	St. Dev.	p-value	$\hat{A}_{12}$	Effect Size	p-value	$\hat{A}_{12}$	Effect Size
Bash	61	0.849 (-0.048) ↓	0.017	250	0.911 (+0.014)	0.044	0.897	0.045	< **0.01**	**0.83**	**Large**	0.22	0.41	Small
Flex	73	0.684 (-0.014) ↓	0.005	250	0.697 (-0.001)	0.002	0.698	0.001	< **0.01**	**1.00**	**Large**	0.06	0.64	Small
Grep	98	0.484 (-)	0.003	250	0.482 (-0.002)	0.006	0.484	0.010	0.21	0.59	Small	0.11	0.62	Small
GZip	144	0.532 (-0.069) ↓	0.084	250	0.591 (-0.010)	0.118	0.601	0.147	**0.01**	**0.69**	**Medium**	0.84	0.52	Negligible
Sed	114	0.981 (-0.011) ↓	0.004	250	0.991 (-0.001)	0.001	0.992	0.002	< **0.01**	**1.00**	**Large**	0.13	0.62	Small

Program	GDE3		MOEA-D/DE		HGA		HGA ≠ GDE3			HGA ≠ MOEA/D-DE		
	Mean	St. Dev.	Mean	St.Dev	Mean	St. Dev.	p-value	$\hat{A}_{12}$	Magnitude	p-value	$\hat{A}_{12}$	Magnitude
Bash	1min 7s ↓	2s	1min 6s ↓	2s	45s	1s	<**0.01**	**0.00**	**Large**	<**0.01**	**0.00**	**Large**
Flex	13s ↓	<1s	12s ↓	<1s	6s	<1s	<**0.01**	**0.00**	**Large**	<**0.01**	**0.00**	**Large**
Grep	52s ↓	5s	37s ↓	3s	26s	3s	<**0.01**	**0.00**	**Large**	<**0.01**	**0.00**	**Large**
GZip	1s ↓	<1s	1s ↓	<1s	<1s	<1s	<**0.01**	**0.00**	**Large**	<**0.01**	**0.00**	**Large**
Sed	7s ↓	<1s	6s ↓	<1s	2s	<1s	<**0.01**	**0.00**	**Large**	<**0.01**	**0.00**	**Large**

comparing `HGA` and `GDE3`, we notice that on 4 out of 5 programs the former algorithm achieves statistically better scores than the latter (three times with `large` effect size and one time with `medium` effect size). Analyzing the results for `MOEA-D/DE`, we observe that in all cases, we cannot reject the null hypothesis and, thus, the results achieved by `HGA` and `MOEA-D/DE` are comparable in terms of APFD$_c$.

It is important to highlight that Table **??** reports the median APFD$_c$ achieved by all the solutions and across all the independent runs. However, the table does not describe the distributions of the scores. To this aim, Figure **??** compares these distributions. We can notice that the solution by `HGA` is more cost-effective than the solutions produced with `GDE3` and `MOEA-D/DE`. Moreover, the performance of these algorithms has a large variation. For example, on `GZip`, the scores for `GDE3` vary between 0.20 and 1.00 while for `MOEA-D/DE` they vary between 0.40 and 1.00. Choosing a proper solution in these contexts is very hard. Indeed, for `GZip` only four out of 144 solutions by `MOEA-D/DE` are more cost-effective than the solution achieved by `HGA`. Moreover, there is no guideline that helps in choosing the most cost-effective solutions in the Pareto front as the fault detection capability can be computed only a posteriori.

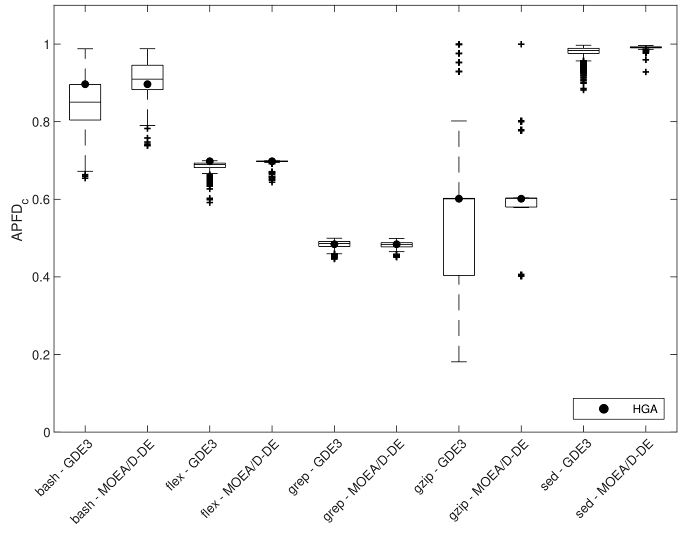


Fig. 8. APFD$_c$ scores achieved by GDE3 (boxplots), MOEA/D-DE (boxplots), and HGA (●) on the five-criteria (four-objective) formulation of TCP problem.

Results for RQ$_{2.2}$. The comparison between `HGA` and `GDE3` shows that, in all the programs, the former algorithm is statistically faster (*i.e.,* $\hat{A}_{12} < 0.5$ and p-value < 0.05) than the latter with `large` effect size. The improvements range between a minimum of 1.49 times and a maximum of 3.50 times, achieved on `Bash` and on `Sed` respectively. On average, `HGA` is 1.75 times faster than `GDE3`.

When comparing `HGA` and `MOEA-D/DE`, we notice that for all the programs we can reject the null hypothesis. According to the Vargha-Delaney ($\hat{A}_{12}$) statistics, in all cases, HGA is more efficient than MOEA-D/DE with large effect size. MOEA-D/DE requires between 1.42 (*e.g.,* Grep) to 3 times (*e.g.,* Sed) the execution times required for HGA. On average HGA is 1.53 times faster than MOEA-D/DE.

Summary for RQ$_2$. HGA often outperforms GDE3 and MOEA-D/DE in terms of cost-effectiveness. The single solution provided by HGA is not dominated in the objectives space by those generated by the many-objective algorithms. Finally, HGA is more efficient than GDE3 (up to 1.65 and 1.75 times for four and five criteria formulations) and MOEA-D/DE (up to 3 and 3.50 times depending on the formulation).

6 Evaluating the Hypervolume Genetic Algorithm on a Large Software System

We conduct a third empirical study to assess the performances of HGA, partially replicating a previous study [?]. In particular, we investigate the following research questions:

RQ$_3$: *How does HGA perform on a large software system with real faults?*

To better clarify it, we detailed it in two research questions:

- **RQ$_{3.1}$**: *What is the cost-effectiveness of HGA on a large software system with real faults?* This research question aims at evaluating to what extent the test case ordering obtained by HGA is able to detect faults (*effectiveness*) earlier (lower execution *cost*) in comparison with two state-of-the-art techniques: a cost cognizant additional greedy algorithm [?], [?], a single objective genetic algorithm based on an AUC metric (GA) [?], and a multi-objective search based algorithm namely NSGA-II [?] used in prior test case prioritization [?], [?] on a large software system, namely MySQL, containing real faults.

- **RQ$_{3.2}$**: *What is the efficiency of HGA on a large software system with real faults?* With this second research question, we are interested in comparing the running time (*efficiency*) required by HGA to find an optimal test ordering, in comparison with the three experimented test case prioritization techniques on a large software system, namely MySQL, containing real faults.

6.1 Study Design

The *context* consists of MySQL, a large real-world system that has been previously studied by Epitropakis *et al.* [?]. MySQL is developed in Java. It comprises $1,283,433$ LOC and has $2,005$ test cases. We used the same real faults from the original study [?], where the authors collected 20 real faults from issue tracker of the software system with "closed" status and available fix patches.

We considered the same testing criteria used by Epitropakis *et al.* [?]. In particular, we considered *statement coverage*, *Δ-coverage*, *past faults coverage*, and *execution cost*. We evaluated three formulations of the TCP problem:

- **Two-criteria (Single-objective)** that (i) minimizes the *execution cost* and (ii) maximizes the *statement coverage*.
- **Three-criteria (Two-objective)** that considers the *past faults coverage* as a third criterion to be maximized.
- **Four-criteria (Three-objective)** that considers *Δ-coverage* as a fourth criterion to be maximized.

In particular, we used the statement coverage matrix and the execution cost array provided by Epitropakis *et al.* [?] and built using the software profiling tool Valgrind. The *Δ-coverage* criterion represents the difference of statement coverage between two consecutive versions of a program. The conjecture behind the use of this information is that changed lines of code are more likely to introduce faults in the system. It was calculated by applying the diff program between two consecutive coverage matrices. It is worth to notice that Epitropakis *et al.* [?] considered only the Four-criteria formulation and that we added the Two- and Three-criteria formulations for sake of completeness.

We compared the results of HGA with those achieved by Additional Greedy [?], [?], [?], (ii) GA [?], and (iii) NSGA-II [?], [?]. More details on these algorithms are provided in Section **??**.

As done for our previous studies, we implemented these algorithms using *JMetal* [?]. Moreover, as in the original study [?], we pre-processed the coverage data using the lossless *coverage compaction algorithm* proposed by Epitropakis *et al.* [?].

We set up the parameters of the algorithms using the same values as the original study [?]. In particular, with respect to our two previous studies, we ran the algorithm for 200 generations (*i.e.,* $50,000$ fitness evaluations). We performed 30 independents runs for each program and for each search algorithm. Finally, we used the same evaluation metrics used in Section **??** and **??**.

6.2 Results of the empirical study

This section discusses the results of our third study, thus, answering the research question.

Results for RQ$_{3.1}$. Table **??** reports the APFD$_c$ values and the running time obtained by HGA and the state-of-the-art algorithms on MySQL. In all the formulations, HGA achieves a higher AFDP$_c$ with respect to Additional Greedy. This difference is statistically significant (p-value< 0.05) with a large effect size. Looking at the two-criteria formulation, we can notice that there is no statistically significant difference between HGA and GA. This result is expected since, as explained in Section **??** and empirically evaluated in Section **??**, the hypervolume and the AUC-based metric are equivalent. More interesting are the comparisons on the three- and four-criteria formulations between HGA and NSGA-II. Even in these cases there are no statistically significant differences in terms of cost-effectiveness demonstrating that even on a large software system, despite using only one fitness function (*i.e.,* the hypervolume indicator), HGA is competitive with NSGA-II.

Since NSGA-II produces multiple solutions, Figure **??** compares the APFD$_c$ values of NSGA-II with those of HGA and Additional Greedy for the three- and four-criteria formulations. As we can observe, the single solution produced by Additional Greedy is worse than all solutions generated by NSGA-II as well as the one produced by HGA. In both the formulations, the distribution of APFD$_c$ scores

TABLE 8
Results for two- (single objective), three- (two objectives), and four-criteria (three objectives) formulation on MySQL: APFD$_c$ and running time achieved by Additional Greedy, NSGA-II, GA (for Two-Criteria), and HGA. For each baseline, in parenthesis is shown the median difference with respect to HGA. Results are highlighted with ↓ when one algorithm is statistically worse than HGA; ↑ when the opposite is true.

Metric	**Add. Greedy**	**NSGA-II (GA for Two-Criteria)**			**HGA**		**HGA ≠ Add. Greedy**			**HGA ≠ NSGA-II (GA for Two-Criteria)**		
		# Sol.	**Median**	**St. Dev.**	**Median**	**St. Dev.**	**p-value**	**$\hat{A}_{12}$**	**Effect Size**	**p-value**	**$\hat{A}_{12}$**	**Effect Size**
Two-Criteria AFDC$_c$	0.462 (-0.205) ↓	-	0.646 (-0.021)	0.065	0.667	0.050	$<$ **0.01**	**1.00**	**Large**	0.06	0.64	Small
Three-Criteria AFDC$_c$	0.475 (-0.181) ↓	12	0.664 (-0.008)	0.053	0.656	0.055	$<$ **0.01**	**1.00**	**Large**	0.66	0.48	Negligible
Four-Criteria AFDC$_c$	0.475 (-0.170) ↓	47	0.652 (+0.007)	0.039	0.645	0.040	$<$ **0.01**	**1.00**	**Large**	0.23	0.44	Negligible

		Mean	**St. Dev.**	**Mean**	**St. Dev.**	**p-value**	**$\hat{A}_{12}$**	**Effect Size**	**p-value**	**$\hat{A}_{12}$**	**Effect Size**
Two-Criteria Execution Time	5s ↑	2min 34s ↓	5s	1min 17s	7s	**<0.01**	**1.00**	**Large**	**<0.01**	**<0.01**	**Large**
Three-Criteria Execution Time	5s ↑	2min 47s ↓	2min 22s	1min 37s	6s	**<0.01**	**1.00**	**Large**	**<0.01**	**0.12**	**Large**
Four-Criteria Execution Time	7s ↑	2min 43s ↓	2min 12s	1min 27s	5s	**<0.01**	**1.00**	**Large**	**<0.01**	**<0.01**	**Large**

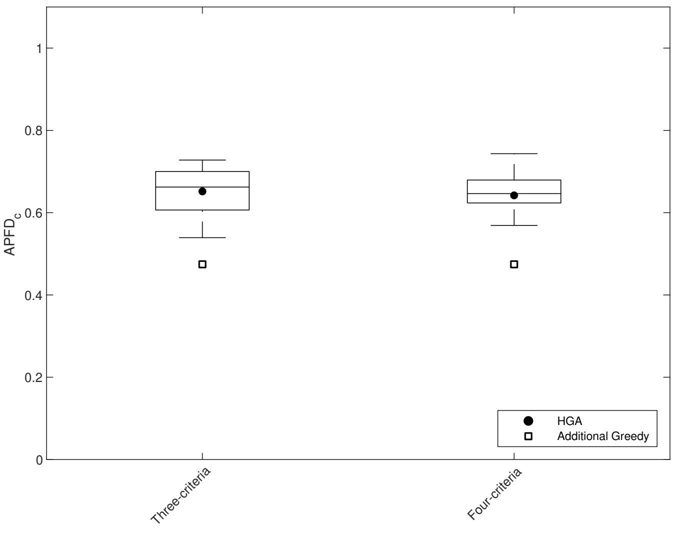


Fig. 9. APFD$_c$ scores achieved by Additional Greedy (□), GA ($*$), NSGA-II (boxplots), and HGA (●) for MySQL.

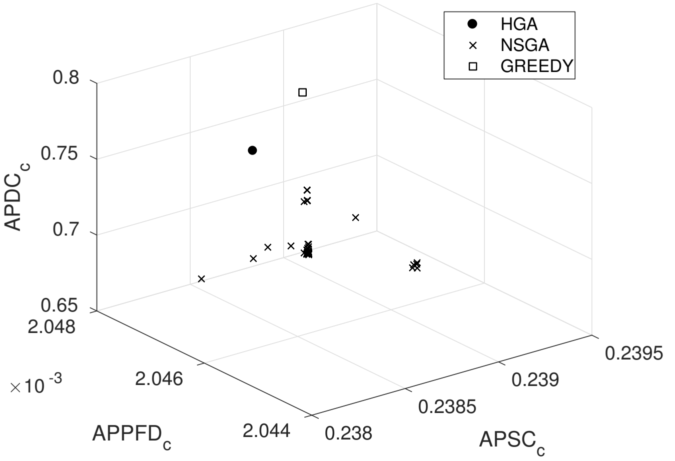


Fig. 10. Pareto frontiers achieved for the four-criteria (three-objective) formulation of the TCP on MySQL.

by NSGA-II presents a large variation with a median value that is very close to the value obtained by HGA.

Figure **??** plots, the Pareto front produced by NSGA-II and the solutions generated by Additional Greedy and HGA on the four-criteria formulation with respect to APSC$_c$, APDC$_c$ and APPFD$_c$, (*i.e.*, the objectives optimized by NSGA-II). The solution by HGA is able to dominate the whole Pareto front of NSGA-II and the solution of Additional Greedy. Even on this large systems, we can notice that the better results produced by NSGA-II in terms of cost-effectiveness (Figure **??**) are not always related to the AUC metrics that are optimized.

Results for RQ$_{3.2}$. Table **??** reports the results in terms of efficiency. More specifically, for all the formulations, the comparison between HGA and Additional Greedy shows that Additional Greedy algorithm is statistically faster (*i.e.*, $\hat{A}_{12} > 0.5$ and p-value < 0.05) with large effect size than HGA, despite being less cost-effective as already shown. Even when comparing HGA with GA and NSGA-II, we can reject the null hypothesis of the *Wilcoxon t-test*. In particular, on the four-criteria formulation, NSGA-II requires 87% more execution time than HGA.

Summary for RQ$_3$. HGA is more cost-effective than Additional Greedy. The solution provided by HGA is not dominated by those generated by NSGA-II. Additional Greedy is statistically more efficient than NSGA-II and HGA, while HGA is faster than GA and NSGA-II.

7 Threats to Validity

This section discusses the threats to the validity of our empirical evaluation, classifying them into *construct*, *internal*, *external*, and *conclusion* validity.

Construct Validity. In this study, they are mainly related to the choice of the metrics used to evaluate the characteristics of the different test case prioritization algorithms. To evaluate the optimality of the experimented algorithms (*e.g.*, HGA, Additional Greedy, GA, NSGA-II, GDE3, and MOEA-D/DE) we used the APFD$_c$ [**?**], a well-known metric used in previous work on multi-objective test case prioritization [**?**], [**?**]. Another construct validity threat involves the correctness of the measures used as test criteria: statement coverage, fault coverage and execution cost. To mitigate such a threat, the code coverage information was collected using two open-source profiler/compiler tools (GNU gcc and gcov). The execution cost has been measured by counting the number of source code blocks expected to be executed by the test cases [**?**], [**?**], while the original fault coverage information has been extracted from the SIR repository [**?**].

Internal Validity. To address the random nature of the GAs themselves [**?**], we run HGA, GA, NSGA-II, GDE3, and MOEA-D/DE 30 times for each subject program (as done in previous work [**?**], [**?**], [**?**]), and considered the median APFD$_c$ scores. The tuning of the EA's parameters is another factor that can affect the internal validity of this work. In this study, we use the same genetic operators and the same parameters used in previous work on test case prioritization [?], [?]. It is worth remarking that previous studies [?], [?] demonstrated that default values are a reasonable choice, even considering that parameter tuning is a long and expensive process that in the context of search-based software engineering does not assure better performances.

External Validity. We consider six open source and proprietary programs, that were used in previous work on regression testing [?], [?], [?], [?], [?], [?]. In details, we firstly compared `HGA` on two different formulations of the test case prioritization problem, with respect to three state-of-the-art algorithms for test case prioritization (*e.g.,* `Additional Greedy`, `GA`, and `NSGA-II`). Secondly, we look at two new formulations of the problem considering more criteria and comparing with two many-objective meta-heuristic algorithms (*i.e.,* `GDE3` and `MOEA/D-DE`). Finally, we partially replicated the study by Epitropakis *et al.* [?] on a large software system (*e.g.,* `MySQL`), comparing `HGA` with `Additional Greedy` and `NSGA-II`

Conclusion Validity. We interpret our findings using appropriate statistical tests. In particular, to test the significance of the differences we used (i) *Welch's t-test* [?] and (ii) *Wilcoxon t-test* [?], while to estimate the magnitude and the effect size of the observed differences we used the Vargha-Delaney statistic [?]. Conclusions are based only on statistically significant results.

8 Conclusion and Future Work

This paper proposed a hypervolume-based genetic algorithm (`HGA`) to improve multi-criteria test case prioritization. Specifically, we use the concept of *hypervolume* [?], which is widely investigated in many-objective optimization, to generalize the traditional Area Under Curve (AUC) metrics used in previous work on test case prioritization [?], [?], [?], [?], [?]. Indeed, the *hypervolume* metric condenses multiple testing criteria through the proportion of the objective space, while AUC based metrics can manage only one cumulative code coverage criterion per time [?].

We performed three empirical studies with three main goals. First of all, we aimed at evaluating the cost-effectiveness and efficiency of `HGA`, compared to three state-of-the-art algorithms for the Test Case Prioritization problem, namely `Additional Greedy` [?], `GA` [?], and `NSGA-II` [?], [?]. Secondly, we intended to analyze the degree to which they handle the selective pressure as the number of objectives grows. Thus we compared `HGA` with two many-objective evolutionary algorithms, *i.e.,* `GDE3` [?] and `MOEA/D-DE` [?]. Finally, we aimed at analyzing the performance of `HGA` in terms of cost-effectiveness and efficiency when dealing with large software systems with respect to two state-of-the-art algorithms such as `Additional Greedy` [?] and `NSGA-II` [?], [?].

Our results show that `HGA` is more or equally cost-effective than the state-of-the-art approaches in most cases. The single solution provided by the algorithm is able to dominate most of the solutions provided by `NSGA-II` in terms of cost-effectiveness. Moreover, the performance of `HGA` does not decrease when larger programs and more objectives are considered. Looking at the execution time we note that the efficiency of `Additional Greedy` is strictly related to the number of test cases, while `HGA` is faster than `GA` and `NSGA-II` in all the considered programs and formulations. Moreover, we show that, in terms of cost-effectiveness, `HGA` is equivalent or better than `GDE3` and `MOEA/D-DE`, while being much more efficient in terms of execution time.

As future work, we plan to incorporate diversity measures proposed in previous studies on multi-objective test case selection [?], [?] to improve the performance of `HGA` for software systems with highly redundant test suites, where greedy algorithms are particularly competitive. We plan to apply the proposed `HGA` also for other test case optimization problems, such as *Test Suite Minimization* and *Test Case Selection*. Finally, starting from the considerations made in the empirical studies, we plan to perform a new empirical study to investigate which testing criteria are more able to discover new faults.

Acknowledgments

We thank Epitropakis, Yoo, Harman, and Burke for providing us the data needed for replicating the study on the MySQL system.

Dario Di Nucci is a research fellow at the Software Languages Lab of the Vrije Universiteit Brussel in Belgium. He received the PhD in Management and Information Technology from the University of Salerno in 2018 advised by Prof. Andrea De Lucia with a thesis entitled "Methods and Tools for Focusing and Prioritizing the Testing Effort". His research interests are within the Software Engineering topic and include software maintenance and evolution, software testing, search based software engineering, green mining, mining software repositories, and empirical software engineering.

Annibale Panichella is an Assistant Professor in the Software Engineering Research Group (SERG) at Delft University of Technology (TU Delft) in Netherlands. He is also a research fellow in the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, where he worked as Research Associate until January 2018. He received the Ph.D. in Software Engineering from the University of Salerno in 2014 with the thesis entitled "Search-based software maintenance and testing". 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.

Andy Zaidman is an associate professor at the Delft University of Technology, The Netherlands. He obtained his M.Sc. (2002) and Ph.D. (2006) in Computer Science from the University of Antwerp, Belgium. His main research interests are software evolution, program comprehension, mining software repositories and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 Andy Zaidman was the laureate of a prestigious Vidi career grant from the Dutch science foundation NWO.

Andrea De Lucia received the Laurea degree in computer science from the University of Salerno, Italy, in 1991, the M.Sc. degree in computer science from the University of Durham, U.K., in 1996, and the Ph.D. degree in electronic engineering and computer science from the University of Naples Federico II, Italy, in 1996. He is a Full Professor of software engineering at the Department of Computer Science of the University of Salerno, the Head of the Software Engineering Lab, and the Director of the International Summer School on Software Engineering. Previously he was at the Department of Engineering and the Research Centre on Software Technology of the University of Sannio, Italy. His research interests include software maintenance and testing, reverse engineering and re-engineering, source code analysis, code smell detection and refactoring, mining software repositories, defect prediction, empirical software engineering, search-based software engineering, traceability management, collaborative development, workflow and document management, and visual languages. He has published more than 250 papers on these topics in international journals, books, and conference proceedings and has edited books and journal special issues. Prof. De Lucia serves on the editorial boards of international journals and on the organizing and program committees of several international conferences. He is a senior member of the IEEE Computer Society and was member-at-large of the executive committee of the IEEE Technical Council on Software Engineering.