Hypervolume-Based Search for Test Case Prioritization

Dario Di Nucci[1(✉)], Annibale Panichella[2], Andy Zaidman[2], and Andrea De Lucia[1]

[1] University of Salerno, Salerno, Italy
{ddinucci,adelucia}@unisa.it
[2] Delft University of Technology, Delft, The Netherlands
{a.panichella,a.e.zaidman}@tudelft.nl

Abstract. Test case prioritization (TCP) is aimed at finding an ideal ordering for executing the available test cases to reveal faults earlier. To solve this problem greedy algorithms and meta-heuristics have been widely investigated, but in most cases there is no statistically significant difference between them in terms of effectiveness. The fitness function used to guide meta-heuristics condenses the cumulative coverage scores achieved by a test case ordering using the Area Under Curve (AUC) metric. In this paper we notice that the AUC metric represents a simplified version of the hypervolume metric used in many objective optimization and we propose HGA, a Hypervolume-based Genetic Algorithm, to solve the TCP problem when using multiple test criteria. The results shows that HGA is more cost-effective than the additional greedy algorithm on large systems and on average requires 36 % of the execution time required by the additional greedy algorithm.

Keywords: Test case prioritization · Genetic algorithm · Hypervolume

1 Introduction

Regression testing is aimed at verifying that software changes do not affect the unchanged parts compromising their behaviours. Many approaches have been proposed in literature for reducing the effort of regression testing [22,29], which remains a particular expensive maintenance activity. One of these approaches is *test case prioritization* (TCP) [11,25]. TCP is aimed at finding an ordering for executing the available test cases, with the goal of executing those test cases that are more likely to reveal faults earlier [12]. Most of the proposed techniques for TCP are based on a coverage criterion [29], such as branch coverage [25], used as a surrogate to prioritize test case with the idea that test cases having higher code coverage also have a higher probability to reveal faults. Once a coverage criterion is chosen, search algorithms can be applied for finding the ordering maximizing the selected criterion.

M. Barros and Y. Labiche (Eds.): SSBSE 2015, LNCS 9275, pp. 157–172, 2015.
DOI: 10.1007/978-3-319-22183-0_11

Greedy Algorithms have been widely investigated in literature for test case prioritization, such as simple greedy algorithms [29], additional greedy algorithms [25], 2-optimal greedy algorithms [22], or hybrid greedy algorithms [15]. 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 [22], such as the Average Percentage Block Coverage (APBC), or the Average Percentage Statement Coverage (APSC). Each fitness function condenses the cumulative coverage scores achieved by a test case ordering using the Area Under Curve (AUC) metric. As such, multiple points are condensed in a single scalar value that can be used as a fitness function of 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 [20,21]. Previous work shows that in most cases there is no statistically significant difference between genetic algorithms and additional greedy approaches in terms of effectiveness, i.e., in terms of the capability of the generated orderings to reveal regression faults earlier [22].

In this paper we notice that the AUC metrics used in the related literature for TCP represents a simplified version of the *hypervolume* metric [2], which is a widely know metric in many-objective optimization. Indeed, in many-objective optimization the problem of condensing multiple criteria has already been investigated by using the more general concept of *hypervolume under manifold* [2], which represents a generalization for the higher dimensional objective space of the AUC-based metrics used in previous TCP studies. Indeed, we show that the *hypervolume* metric allows to condense not only a single cumulative code coverage (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, in only one scalar value. Therefore, in this paper we propose HGA, a Hypervolume-based Genetic Algorithm to solve the TCP problem when using multiple criteria. To show the applicability of the proposed algorithm, we conducted an empirical study involving six real world open-source programs. The results achieved shows that HGA is more cost-effective than the additional greedy algorithm on large systems and on average requires 36 % of the execution time required by the additional greedy algorithm. As a further contribution, we also show that for TCP the computation of the hypervolume metric is polynomial with respect to the number of the testing criteria, while in general for traditional optimization problems it is exponential.

2 Background and Related Work

Test Case Prioritization (TCP) has been widely investigated in literature. The most investigated direction regards the choice of a proper testing criterion to use for generating a test case ordering aimed at maximizing the real fault detection rate. Since the fault capability can not be known to the tester in advance until the test cases are executed according to the chosen ordering, researchers have

proposed to use surrogate metrics which are in some way correlated with the fault detection rate [29]. Code coverage is one of the most widely used prioritization criterion, such as branch coverage [25], statement coverage [11], block coverage [8], and function or method coverage [13]. Other prioritization criteria were also used instead of structural coverage, such as interaction [3], clustering-based [5], and requirement coverage [27].

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 [15,32]: the *total* strategy selects test cases according to the number of code elements they cover, whereas the *additional* strategy iteratively selects a next test case that yields the maximal coverage of code elements not yet covered by previously selected test cases. Recently, Hao *et al.* [15] and Zhang *et al.* [32] 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.* [10] and Malishevsky *et al.* [23] 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, history faults coverage and execution cost [24,29].

Other than greedy algorithms, meta-heuristics have been investigated as alternative search algorithms to test case prioritization. Li *et al.* [22] compared additional greedy algorithm, 2-optimal greedy, hill climbing and genetic algorithms for code coverage based TCP. To allow 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). Each of these metrics condenses the cumulative coverage scores (e.g., branch coverage) achieved when considering the test cases in the given order sequentially [22] 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 [22]. Since this metric allows 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 in [22], in most cases the difference between the effectiveness of permutation-based genetic algorithms and additional greedy approaches is not significant.

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, in a further paper Li *et al.* [21] 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. Similarly, Islam *et al.* [20] 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. Both these two multi-objective approaches to test case prioritization [20,21] have important drawbacks. Firstly, they can provide hundreds of orderings representing trade-offs between AUC metrics and not between the selected testing criteria. Furthermore, no guidelines are given to guide the decision maker in selecting the ordering to use. 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 [18]. Therefore, other non classical many-objective solvers must be investigated in order to deal with multiple (many) testing criteria. Finally, in [20–22] there is no empirical evidence of the effectiveness of NSGA-II with respect to simple heuristics, such as greedy algorithms, in terms of cost-effectiveness.

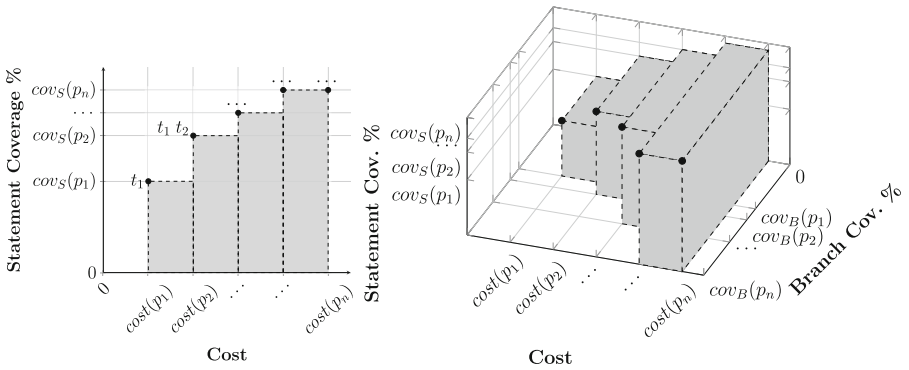
In this paper we notice that the most natural way to deal with the multi-objective TCP problem is represented by the *hypervolume*-based solvers since the AUC metrics used in the related literature for TCP represent a specific simplified version of the *hypervolume* metric [2]. Indeed, in many-objective optimization the *hypervolume* metrics is widely used to condense points from a higher dimensional objective space in only one scalar value. Instead of using the Area Under Curve to condense multiple points, the hypervolume metric uses the more general concept of *hypervolume under manifold* for this aim [2]. For these reasons, in this paper we propose to use an hypervolume metric to solve the multi-objective TCP problem. Moreover, we determine that 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.

3 Hypervolume Indicator for TCP

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 [2]. Therefore, instead of optimizing the objective functions directly, indicator-based algorithms are aimed at finding a set of solutions that maximizes the underlying quality indicator [2]. 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 [2]). For two-objective problems, the *hypervolume* corresponds to the area under curve, i.e., the proportion of the area that is dominated by a given set of candidate solutions.

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, 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 Fig. 1(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 $cov_S(t_1)$ with cost equal to $cost(t_1)$ (first cumulative point p_1); the second test case in the ordering t_2 reaches a new cumulative statement coverage $cov_S(t_1, t_2) = cov_S(t_1) \cup cov_S(t_2)$ with $cost(t_1, t_2) = cost(t_1) + cost(t_2)$ (second cumulative point p_2); and so on. 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 Fig. 1(a)). These points are of *weakly monotonically increasing* since both cumulative coverage and cumulative cost increase when adding a new test case from the ordering, i.e., $cov_S(t_1) \leqslant cov_S(t_1, t_2)$ and $cost(t_1) \leqslant cost(t_1, t_2)$.



(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 $P(\tau)$.

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 Fig. 1(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:

Definition 1. *We say that a point X dominates another point Y (also written $X >_p Y$) if and only if the values of the objective functions satisfy:*

$$\begin{gathered} cost(X) \leqslant cost(Y) \;\; and \;\; cov_S(X) > cov_S(Y) \\ or \\ cost(X) < cost(Y) \;\; and \;\; cov_S(X) \geqslant cov_S(Y) \end{gathered} \tag{1}$$

Two different orderings correspond to two different 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 Fig. 1(a). Similarly, if we consider a third testing criteria (such as branch coverage $cov_B(p_i)$) 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 Fig. 1(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 $cov_B(p_i)$. 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 $F = \{cost, f_1, \ldots, f_m\}$ a set of testing criteria used to prioritize the test case in T, where *cost* denotes the execution cost of each test case while $f_1, \ldots, f_m$ are the remaining m testing criteria to maximize. Given a permutation τ of test case in T we can compute the corresponding set of cumulative points $P(\tau) = \{p_1, \ldots, p_n\}$ obtained by cumulating the scores $cost, f_1, \ldots, f_m$ achieved by each test case in the order τ.

Definition 2. *The hypervolume dominated by a permutation $P(\tau)$ of test cases can be computed as follows:*

$$I_H(\tau) = \sum_{i=1}^{(n-1)} [cost(p_{i+1}) - cost(p_i)] \times f_1(p_i) \times \cdots \times f_m(p_i) \tag{2}$$

where $[cost(p_i) - cost(p_{i+1})] \times f_1(p_i) \times \cdots \times f_m(p_i)$ measure 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 3. *The fraction of the hypervolume dominated by a permutation* $P(\tau)$ *of test cases is:*

$$I_{HP}(\tau) = \frac{\sum_{i=1}^{(n-1)} [cost(p_{i+1}) - cost(p_i)] \times f_1(p_i) \times \cdots \times f_m(p_i)}{cost(p_n) \times f_1^{max} \times \ldots f_m^{max}} \quad (3)$$

where $cost(p_n)$ is the execution cost of the whole test suite T. 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. In this paper we consider the $I_{HP}(\tau)$ metric as suitable fitness function to guide search algorithms, such as genetic algorithm, in finding the optimal ordering τ in multi-objective test case prioritization. As such, we propose a new genetic algorithm named HGA (Hypervolume-based Genetic Algorithm).

Since in TCP a candidate test case ordering corresponds to a set of *monotonically* increasing cumulative scores (as described in the previous section) we can use the Eq. 3 for computing the dominated hypervolume instead of the more expensive algorithm used in traditional many-objective optimization [2]. Specifically, 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), the $I_{HP}(\tau)$ 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 hypervolume metric requires a more complex algorithm which is exponential with respect to the number of objectives m [2], or testing criteria for TCP.

The $I_{HP}(\tau)$ metric proposed in this paper can be viewed as a generalization of the cumulative scores used in previous 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 [22]. Under the 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 testing criterion. A similar consideration can be performed for all the other cumulative fitness functions used in previous work on search-based test case prioritization [20–22].

4 Empirical Evaluation

The *goal* of this study is to evaluate the Hypevolume-based GA, with the *purpose* of solving the test case prioritization problem. The *quality focus* of the study

is represented in terms of three—possibly conflicting—testing criteria which are pursued when performing test case prioritization, namely execution cost (to minimize), statement coverage (to maximize), and past fault coverage (to maximize). The *context* of our study consists of six open-source and industrial programs available from the Software-artifact Infrastructure Repository (SIR) [19]: four GNU open-source programs `bash`, `flex`, `grep`, and `sed`; and two programs of the Siemens suite, namely `printtokens`, and `printtokens2`. Their main characteristics are summarized in Table 1. We selected these programs since they have been used in previous work on regression testing [4,7,22,30,31], hence, allowing us—wherever possible—to compare results. For two programs extracted from the Siemens suite, SIR provides a large number of test suites but with a limited number of test cases. Therefore, in the context of our study we considered all the available test cases.

Table 1. Programs used in the study.

Program	LOC	# of Test Cases	Description
bash	59,846	1,200	Shell language interpreter
flex	10,459	567	Fast lexical analyser
grep	10,068	808	Regular expression utility
printtokens	726	4,130	Lexical analyzer
printtokens2	520	4,115	Lexical analyzer
sed	14,427	360	Non-interactive text editor

The empirical evaluation is steered by the following research questions:

RQ$_1$: *What is the cost-effectiveness of HGA, compared to cost-aware additional greedy algorithms?* This research question aims at evaluating to what extent faults (*effectiveness*) can be detected earlier (lower execution *cost*) using the test cases ordering obtained by HGA, in comparison with a baseline technique namely two-and three-objective additional greedy algorithms. This reflects the software engineer's needs to obtain the maximum advantage from testing even if it is prematurely stopped at some point.

RQ$_2$: *What is the efficiency of HGA, compared to cost-aware additional greedy algorithms?* With this second research question we are interested in comparing the running time (*efficiency*) required by HGA to find an optimal ordering, in comparison with two- and three-objective additional greedy algorithms.

Testing Criteria. To answer our research questions we consider three objectives widely used in previous test case prioritization work [17,22]:

– *Statement coverage criterion.* We measure statement coverage achieved by each test case using the `gcov` tool part of the GNU C compiler (`gcc`).

- *Execution cost criterion.* In this paper we approximate the execution cost by counting the number of executed instructions in the code, instead of measuring the actual execution time, similarly as done in previous work [24,29]. To this aim we use the `gcov` tool to measure the execution frequency of each source code instruction.
- *Past fault coverage criterion.* As for the past fault coverage criterion, we consider the versions of the programs with seeded faults available in the SIR repository [19]. SIR also specifies whether or not each test case is able to reveal these faults. Such information can be used to assign a past fault coverage value to each test case subset, computed as the number of known past faults that this subset is able to reveal in the previous version.

Problem Formulation. Using the three test case prioritization criteria described above, we examine two and three-objective formulations of TCP problem. The *two-objective TCP problem* is aimed at finding an optimal ordering of test cases which (i) minimizes the execution cost and (ii) maximizes the statement coverage. In this case the $I_{HP}(\tau)$ metric corresponds to the area under curve delimited by the two criteria. For the *three-objective TCP problem* we consider the past faults coverage as third criteria to be maximized. Thus, in this second case the $I_{HP}(\tau)$ metric corresponds to the volume under manifolds delimited by the three criteria. We note that it is possible to formulate other criteria by just providing a clear mapping between tests and criterion-based requirements. The formulations are used to illustrate how the Hypervolume-based metric introduced in this paper 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 the fitness function $I_{HP}(\tau)$.

Evaluated Algorithms. For the two-objective formulation of the test case prioritization problem, we compare HGA instantiated with two criteria and the *additional greedy* algorithm used by Yoo and Harman [30] and by Rothermel *et al.* [25], which considers at the same time coverage and cost by maximizing the coverage per unit of time of the selected test cases (cost cognizant additional greedy). Note that, after reaching the maximum coverage with the additional greedy, there are possible remaining un-prioritized test cases that cannot add additional coverage. These remaining test cases could be ordered using any algorithm; in this work we re-apply *additional greedy* algorithm as done in previous work [22]. Similarly, for what concerns the three-objective formulation of the test case prioritization problem, we compare HGA instantiated for three criteria with the *additional greedy* algorithm used by Yoo and Harman [30], which conflates code coverage, execution cost and past coverage in one objective function to be minimized. Also in this case the *additional greedy* is re-applied multiple times in order to have a complete test cases ordering.

Implementation. All the algorithms have been implemented using *JMetal* [9], a Java-based framework for multi-objective optimization with metaheuristics. In details, we use the *Parallel Genetic Algorithm* which evaluates the individuals in parallel using multiple threads, thus reducing the execution time. We use

a population of 100 individuals that are initially randomly generated within the solution space. At each generation, *offsprings* are generated by combining pairs of fittest individuals with probability $p_c = 0.90$ by using the *PMX-Crossover*, which swaps the permutation elements at a given crossover point. As mutation operator we use the *SWAP-Mutation*, which randomly swaps two chosen permutation elements within each offspring. The fittest individuals are selected using the *tournament selection* with *tournament size* equal to 10. The algorithm ends when reaching 250 generations. To account for the inherent randomness nature of GAs [1], we performed 20 independents runs for each program under study and for each TCP problem.

Evaluation Metrics. To address $\mathbf{RQ}_1$ we use the *cost-cognizant average fault detection percentage* metric (APFD_c) proposed by Elbaum *et al.* [10]. This metric measures the effectiveness of a given test cases ordering by summing up the costs of the first test cases that are able to reveal the faults [10]. The higher the AFDP_c value, the lower the average cost needed to detect the same number of faults. Since we performed 20 independent runs, we reports the mean and the standard deviation of the APFD_c scores achieved for each program under study and for each TCP problem. We statistically analyze the obtained results, to check whether the differences between the APFD_c scores produced by the compared algorithms over different independent runs are statistically significant or not. To this aim we use the *Welch's t test* [6] with a p-value threshold of 0.05 for both the TCP problems. Welch's t-test is generally used to test two groups with unequal variance, e.g., in our case the variance of the APFD_c produced by the additional greedy and HGA is different[1]. Significant p-values indicate that the corresponding null hypothesis can be rejected in favour of the alternative ones. Other than testing the null hypothesis, we use the Vargha-Delaney ($\hat{\text{A}}_{12}$) statistical test [28] to measure the effect size, i.e., the magnitude of the difference between the APFD_c achieved with different algorithms. To address $\mathbf{RQ}_2$ we compare the average running time required by each algorithm for each software program used in the empirical study. The execution time was measured using a machine with Intel Core i7 processor running at 2.40GHz with 12GB RAM.

5 Empirical Results

Table 2 reports the AFDP_c values for the two-objective and three-objective test case prioritization problem obtained by (i) the additional greedy algorithm, and (ii) HGA. Specifically, the table reports mean size and standard deviation over 20 independent runs of the algorithms. In both problem formulations, HGA is more cost-effective than the additional greedy algorithm for 4 out of 6 programs since the mean AFDP_c is higher. In particular, for the two-objective formulation there is an improvement in terms of AFDP_c ranging between 5 % and 11 %,

[1] Since the additional greedy is a deterministic algorithm, the variance over 20 independent runs is zero. Conversely, because of the random inheritance of GAs, HGA does not reach a zero variance.

Table 2. Test case prioritization problem: $AFDP_c$ achieved by HGA and additional greedy in two and three objective formulations. The best result for each program is highlighted in bold face.

Program	2-Objective			3-Objective		
	Add. Greedy	HGA		Add. Greedy	HGA	
		Mean	St. Dev.		Mean	St. Dev.
bash	0.658	**0.705**	0.046	0.658	**0.743**	0.053
flex	0.604	**0.677**	0.116	0.507	**0.578**	0.100
grep	0.793	**0.815**	0.023	0.793	**0.816**	0.039
printtokens	**0.588**	0.287	0.091	**0.496**	0.203	0.052
printtokens2	**0.733**	0.462	0.326	**0.312**	0.275	0.253
sed	0.787	**0.831**	0.081	0.688	**0.744**	0.103

while in the three-objective formulation the improvement ranges between 3 % and 12 %. This has practical implications from the tester's perspective since the test cases orderings obtained by HGA detect more faults at the same (or lower) execution cost. Conversely, the additional greedy algorithm has better performance on `printtokens` and `printtokens2` for both two- and three-objective TCP problems. Indeed, for these programs we can observe that the $AFDP_c$ values obtained by the additional greedy are substantially higher than the mean $AFDP_c$ values achieved by HGA (+27 % for two-objective problem and +3.27 % for the three-objective one on `printtokens2`). For these two programs we can also observe that HGA has a higher variability when compared with the other programs as demonstrated by the higher standard deviations: for example the standard deviation for `printtokens` is larger than 25 % for both two- and three-objective TCP while for `bash` it is less than 6 %. This high variability can be due to the fact that `printtokens` and `printtokens2` are two very small programs with less than 1,000 lines of code while their test suites are very large because they contain more than 4,000 test cases. Hence, for these programs reaching the maximum statement coverage and past faults coverage requires the execution of only 30 test cases on average, i.e., less than 1 % of the whole test suites. For the two considered coverage criteria the majority of test cases are equivalent (i.e., they have the same code and past fault coverage) but only few of them are really able to detect new faults. the obtained ordering. Thus, HGA might select other test cases that are equivalent in terms of code (or past fault) coverage but that have different fault detection capabilities. For the additional greedy algorithm this is not the case since it always generates the same test case ordering. This analysis highlights that the poor performance of HGA for `printtokens` and `printtokens2` can be due to the used testing criteria more than the algorithm itself.

To provide statistical support to our preliminary analysis, Table 3 reports the results of the Welch's t-test and the Vargha-Delaney ($\hat{A}_{12}$) statistic, obtained by

Table 3. Welch's t-test p-values of the hypothesis HGA > Additional Greedy for the two and three objective test case prioritization problem. $p-values$ that are statistically significant (i.e., $p - value < 0.05$) are reported in bold face. $\hat{A}_{12} > 0.5$ means HGA is better than Additional Greedy; $\hat{A}_{12} < 0.5$ means Additional Greedy is better than HGA, and $\hat{A}_{12} = 0.5$ means they are equal.

Program	2-Objective			3-Objective		
	p-value	$\hat{A}_{12}$	Magnitude	p-value	$\hat{A}_{12}$	Magnitude
bash	< **0.01**	0.88	Large	< **0.01**	0.95	Large
flex	< **0.01**	0.70	Medium	< **0.01**	0.75	Large
grep	< **0.01**	0.85	Large	< **0.01**	0.85	Large
printtokens	1	0.10	Large	1	0.10	Large
printtokens2	1	0.30	Large	0.73	0.40	Small
sed	**0.01**	0.85	Large	**0.01**	0.80	Large

comparing (across the 20 GA runs) the $AFDP_c$ value yielded by the algorithms under investigation. As expected, HGA is statistically better than the additional greedy in 4 cases out of 6 for both two- or three-objective TCP problems. For these cases the effect size ($\hat{A}_{12}$) is always *large* with the only exception of `flex` where for the two-objective TCP problem the effect size is *medium*. For `printtokens` we can observe that HGA is statistically worse than the additional greedy algorithm for both two- and three-objective TCP problems and the magnitude of the difference is also *large* according to the $\hat{A}_{12}$ statistic. For `printtokens2` there is no statistically significant difference between HGA and the greedy algorithm for the three-objective TCP problem while for the two-objective problem there is a statistically significant difference in favour of the additional greedy.

Table 4. Average Execution Time for Algorithms

Program	2-Objective		3-Objective	
	Add. Greedy	HGA	Add. Greedy	HGA
bash	2h 21 min 57s	**3 min 1s**	2h 46 min 40s	**11 min 13s**
flex	2 min 19s	**43s**	2 min 46s	**51s**
grep	9 min 41s	**2 min 19s**	11 min 21s	**2 min**
printtokens	2 min 47s	**2s**	3 min 19s	**11s**
printtokens2	3 min 11s	**1s**	6 min 51s	**5s**
sed	25s	**12s**	30s	**16s**

To answer our $\mathbf{RQ}_2$, Table 4 reports the mean execution time required by each algorithm for each software program used in the empirical study. For both

two- and tree-objective formulation, we can note that HGA requires less execution time for its convergence with respect to the additional greedy algorithm. Specifically, HGA on average takes 36 % of the execution time required by the additional greedy for the same software system. This is an important improvement if we also consider that HGA is not only much faster than the additional greedy algorithm, but it also provides orderings that are able to reveal more faults ($\mathbf{RQ}_1$).

It is important to highlight that the running times of the additional greedy algorithms reported in this paper are substantially higher than the running times reported in previous studies for test case selection using the same additional greedy algorithms and for the same software systems [24,31]. For example in [31] the average running time of the two-objective additional greedy algorithm for `grep` is 20 seconds against 11 min and 21 seconds reported in this paper. This huge difference concerns the different stop conditions used to end the additional greedy algorithm in test case selection and test case prioritization problems. In the test case selection problem the additional greedy ends when the maximum code coverage is reached, thus, as reported by Harrold *et al.* [16] the execution time of $O(\mid T \mid \cdot \max \mid T_i \mid)$, where $\mid T \mid$ represents the size of the original test suite, while $\max \mid T_i \mid$ denotes the cardinality of the largest group of test cases which is able to reach the maximum coverage. For TCP the additional greedy algorithm does not end when the maximum coverage is reached but it is re-applied until all test cases are selected in order to obtain a complete test case ordering. Thus, for TCP the running time of the (re-started) additional greedy algorithm is $O(|T| \times |T|)$ motivating the higher execution time reported in this paper. These findings are particularly interesting since in previous works on multi-objective test case selection [24,31] the additional greedy algorithm turned out to be faster than genetic algorithms with the only exception of large programs [24]. For multi-objective TCP problems we highlight that genetic algorithms, and HGA in particular, are always faster than the additional greedy algorithm independently of the size of the program and the test suites. We also note that despite the lower running time, HGA is more cost-effective than the additional greedy algorithm ($\mathbf{RQ}_1$).

6 Threats to Validity

This section discusses the threats to the validity of our empirical evaluation, classifying them into *construct*, *internal*, and *external* 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. In order to evaluate the optimality of the experimented algorithms (HGA, and additional greedy) we used the APFD_c [10], a well-know metric used in previous work on multi-objective test case prioritization [13,14]. Another construct validity threat involves the correctness of the measures used as test criteria: statement coverage, faults 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 was measured by counting the number of source code blocks expected to be executed by the test cases, while the original fault coverage information was extracted from the SIR repository [19].

Internal Validity. To address the random nature of the GAs themselves [1], we ran HGA 20 times for each subject program (as done in previous work [7,22,30]), and considered the mean $APFD_c$ scores. The tuning of the GA's parameters is another factor that can affect the internal validity of this work. In this study we used the same genetic operators and the same parameters used in previous work on test case prioritization [20,22].

External Validity. We considered 6 programs from the SIR, that were also used in most previous work on regression testing [4,7,22,26,31]. However, in order to corroborate our findings, replications on a wider range of programs and optimization techniques are desirable. Also, there may be optimization algorithms or formulations of the test case prioritization problem not considered in this study that could produce better results. In this paper we compared HGA with the additional greedy algorithm in order to evaluate the benefits of the proposed algorithms over the most used. Moreover, in order to make the results more generalizable, we evaluated all the algorithms with respect to solving two different formulations of the test case prioritization problem with two and three objectives to be optimized.

7 Conclusion and Future Work

This paper proposes a hypervolume-based genetic algorithm (HGA) to solve multi-criteria test case prioritization. Specifically, we use the concept of *hypervolume* [2], 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 [20–22]. Indeed, the proposed *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 [22].

To show the applicability of HGA we instantiated the TCP problem using three different testing criteria. The empirical study conducted on six real-world open source programs demonstrated that the proposed algorithm is not only much faster than greedy algorithms, but is also able to generate test case orderings allowing to reveal more regression faults at the same level of execution cost for large software programs. This denotes an important finding since previous search-based approaches based on AUC metrics did not statistically outperform greedy algorithms in terms of effectiveness [22].

Given the promising results obtained in this paper, we plan to apply the *hypervolume* metric when considering up to three testing criteria in order to investigate its scalability with respect to greedy algorithms for higher dimensional TCP problems. We also plan to replicate the study, considering more

and different software systems and different coverage criteria to corroborate the results reported in this paper. Then, we plan to incorporate diversity measures proposed in previous studies on multi-objective test case selection [7,24] to improve the performances of HGA for software systems with highly redundant test suites, where greedy algorithms are particularly competitive. Finally, we plan to apply the proposed meta-heuristic also for other test case optimitization problems.

References

1. Arcuri, A., Briand, L.C.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of International Conference on Software Engineering (ICSE), pp. 1–10. ACM (2011)
2. Auger, A., Bader, J., Brockhoff, D., Zitzler, E.: Theory of the hypervolume indicator: optimal μ-distributions and the choice of the reference point. In: Proceedings of SIGEVO workshop on Foundations of Genetic Algorithms (FOGA), pp. 87–102. ACM (2009)
3. Bryce, R.C., Colbourn, C.J., Cohen, M.B.: A framework of greedy methods for constructing interaction test suites. In: Proceedings International Conference on Software Engineering (ICSE), pp. 146–155 (2005)
4. Chen, T.Y., Lau, M.F.: Dividing strategies for the optimization of a test suite. Inf. Process. Lett. **60**(3), 135–141 (1996)
5. Cohen, M., Dwyer, M., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Trans. Softw. Eng. **34**, 633–650 (2008)
6. Conover, W.J.: Practical Nonparametric Statistics, 3rd edn. Wiley, New York (1998)
7. De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A.: On the role of diversity measures for multi-objective test case selection. In: Proceedings of International Workshop on Automation of Software Test (AST), pp. 145–151 (2012)
8. Do, H., Rothermel, G., Kinneer, A.: Empirical studies of test case prioritization in a junit testing environment. In: 15th International Symposium on Software Reliability Engineering, pp. 113–124. IEEE Computer Society (2004)
9. Durillo, J.J., Nebro, A.J.: jMetal: a java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011)
10. Elbaum, S., Malishevsky, A., Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of International Conference on Software Engineering (ICSE), pp. 329–338. IEEE (2001)
11. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), pp. 102–112. ACM (2000)
12. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. Softw. Eng. Notes **25**, 102–112 (2000)
13. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)
14. Elbaum, S., Rothermel, G., Kanduri, S., Malishevsky, A.: Selecting a cost-effective test case prioritization technique. Softw. Qual. J. **12**(3), 185–210 (2004)
15. Hao, D., Zhang, L., Zhang, L., Rothermel, G., Mei, H.: A unified test case prioritization approach. ACM Trans. Softw. Eng. Methodol. **24**(2), 10:1–10:31 (2014)

16. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2**, 270–285 (1993)
17. Huang, Y.C., Huang, C.Y., Chang, J.R., Chen, T.Y.: Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. In: Proceedings of Annual Computer Software and Applications Conference (COMPSAC), pp. 413–418. IEEE (2010)
18. Hughes, E.: Evolutionary many-objective optimisation: many once or one many? IEEE Congr. Evol. Comput. **1**, 222–227 (2005)
19. Hyunsook Do, S.G.E., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Softw. Eng.: Int. J. **10**, 405–435 (2005)
20. Islam, M., Marchetto, A., Susi, A., Scanniello, G.: A multi-objective technique to prioritize test cases based on latent semantic indexing. In: Proceedings of European Conf. on Software Maintenance and Reengineering (CSMR), pp. 21–30. IEEE (2012)
21. Li, Z., Bian, Y., Zhao, R., Cheng, J.: A fine-grained parallel multi-objective test case prioritization on GPU. In: Ruhe, G., Zhang, Y. (eds.) SSBSE 2013. LNCS, vol. 8084, pp. 111–125. Springer, Heidelberg (2013)
22. Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. **33**(4), 225–237 (2007)
23. Malishevsky, A.G., Ruthruff, J.R., Rothermel, G., Elbaum, S.: Cost-cognizant test case prioritization. Technical report, Department of Computer Science and Engineering (2006)
24. Panichella, A., Oliveto, R., Di Penta, M., De Lucia, A.: Improving multi-objective test case selection by injecting diversity in genetic algorithms. IEEE Trans. Softw. Eng. **41**(4), 358–383 (2015)
25. Rothermel, G., Untch, R., Chu, C., Harrold, M.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)
26. Rothermel, G., Harrold, M.J., von Ronne, J., Hong, C.: Empirical studies of test-suite reduction. Softw. Test. Verif. Reliab. **12**, 219–249 (2002)
27. Srikanth, H., Williams, L., Osborne, J.: System test case prioritization of new and regression test cases. In: International Symposium on Empirical Software Engineering (2005)
28. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)
29. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012)
30. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA), pp. 140–150. ACM (2007)
31. Yoo, S., Harman, M.: Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. J. Syst. Softw. **83**(4), 689–701 (2010)
32. Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H.: Bridging the gap between the total and additional test-case prioritization strategies. In: Proceedings of International Conference on Software Engineering (ICSE), pp. 192–201. IEEE (2013)