# Empirical Evaluation of Pareto Efficient Multi-objective Regression Test Case Prioritisation

Michael G. Epitropakis
Computing Science and Mathematics
University of Stirling
Stirling, UK
mge@cs.stir.ac.uk

Shin Yoo
Department of Computer Science,
University College London,
London, UK
shin.yoo@ucl.ac.uk

Mark Harman
Department of Computer Science,
University College London,
London, UK
mark.harman@ucl.ac.uk

Edmund K. Burke
Computing Science and Mathematics
University of Stirling
Stirling, UK
e.k.burke@stir.ac.uk

## ABSTRACT

The aim of test case prioritisation is to determine an ordering of test cases that maximises the likelihood of early fault revelation. Previous prioritisation techniques have tended to be single objective, for which the additional greedy algorithm is the current state-of-the-art. Unlike test suite minimisation, multi objective test case prioritisation has not been thoroughly evaluated. This paper presents an extensive empirical study of the effectiveness of multi objective test case prioritisation, evaluating it on multiple versions of five widely-used benchmark programs and a much larger real world system of over 1 million lines of code. The paper also presents a lossless coverage compaction algorithm that dramatically scales the performance of all algorithms studied by between 2 and 4 orders of magnitude, making prioritisation practical for even very demanding problems.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Algorithms

## Keywords

Test case prioritization, multi-objective evolutionary algorithm, additional greedy algorithm, coverage compaction

## 1. INTRODUCTION

Test case prioritisation [13] is useful when the tester is forced to terminate testing before all the test cases have been executed. Such premature test termination can occur because of business imperatives, such as fixed release dates, or due to budgetary constraints. Prioritisation is an attractive way to mitigate the reduction in test effectiveness that would otherwise accompany premature test termination. Indeed, a recent survey revealed increasing interest in prioritisation over other forms of regression test optimisation, such as selection and minimisation [39].

Of course, it is not known which tests will reveal which faults at prioritisation time, so some surrogate has to be employed (as in other regression test optimisation approaches). Often structural coverage is adopted as this surrogate [12, 25, 30, 44]. However, such a single objective 'coverage only' approach is limited. In practice, the tester may have multiple technical and business imperatives driving their testing, making it unrealistic to limit prioritisation to a single objective. Furthermore, even where the tester is solely concerned with the single objective of fault revelation, there are likely to be multiple coverage-based surrogates from which to choose, such as coverage of new functionality or coverage of previously revealed faults.

These practical limitations of single objective regression test optimisation have led to an upsurge in interest in multi objective regression testing [8, 19, 20, 24, 34, 37]. However, all but one of the previous studies of multi objective regression test optimisation have been concerned with test case minimisation rather than test case prioritisation. Existing empirical studies on test case prioritisation either entirely focused on the variations of the greedy algorithm [12] or considered other algorithms, such as evolutionary algorithm, exclusively for the single objective formulation of the problem [25]. The only previous study to use a multi objective prioritisation [24] was primarily concerned with extending, to prioritisation, previous work on parallel computation speed-ups for test case minimisation [41]. It, therefore, reports the speed-up achieved by parallelisation, but does not evaluate the fault detection capabilities of different single and multi objective algorithms in an empirical study.

This paper studies multi objective test case prioritisation in detail. Three objectives are considered: average percentage of coverage achieved, average percentage of coverage of changed code, and average percentage of past fault coverage.

*ISSTA'15*, July 12–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771788

The first objective is the widely-used surrogate for fault detection capability. The second objective is included based on the conjecture that prioritising differences between versions is also a natural objective, especially in the context of regression testing. Finally, the third objective is included because previous work on test suite minimisation [43] has noted that tests that have detected faults in the past may tend to be more effective than those that have not.

The evaluation of multi objective test case prioritisation has been performed on multiple versions of five SIR [10] programs (`flex`, `grep`, `gzip`, `make` and `sed`), which contain seeded faults (as in much of the previous literature [39]). Additionally, the empirical study also includes versions of a large real world system, `mysql`, which has not previously been studied in the literature and for which information concerning its real faults has been extracted. In total, the empirical study included 22 regression testing phases between versions.

The empirical study compared 7 different prioritisation algorithms in total. For multi objective test case prioritisation, the study includes new implementations of two different Multi Objective Evolutionary Algorithms (MOEAs): the widely studied NSGA-II algorithm [9] and the Two Archive multi objective algorithm [29]. In addition, the study considers three different instantiations of the current state-of-the-art single objective cost-cognisant additional greedy algorithm, each with one of the considered objectives. Finally, following existing work on test suite minimisation [38], two hybrid algorithms were formulated by seeding the initial populations of two MOEAs with solutions from the additional greedy prioritisations. These algorithms are compared using standard optimisation quality indicator metrics. The results of our study show that the MOEAs can significantly outperform the state-of-the-art with large effect sizes according to each of three quality indicators for at least 19 of the 22 versions studied in the paper.

More importantly for practitioners, the results also show that MOEAs and hybrids can significantly outperform the testing effectiveness of statement coverage based prioritisations in up to 14 out of the 22 versions studied. They find faults significantly faster in all 14 cases, often with large effect sizes, according to the standard evaluation metric, cost cognisant Average Percentage of Fault Detected (APFD$_c$) [14].

The gain in testing effectiveness may have been futile had the cost of using MOEAs been prohibitively high. Fortunately, the paper also introduces and evaluates *coverage compaction*, an algorithm for non-lossy coverage data compaction. Unlike existing work on execution profile reduction [6], the proposed compaction algorithm is deterministic, it does not affect the precision of any follow-up analysis, and it can be used as a pre-processing phase. The results show dramatic improvements in performance: after compaction, the size of the coverage data becomes smaller by a factor of between 7 and 488. With the largest studied program, `mysql`, it led to four orders of magnitude speed-up for both MOEAs and the additional greedy algorithms.

The technical contributions of this paper are as follows:

- An empirical study of two multi objective evolutionary algorithms, as well as three state-of-the-art cost-cognisant additional greedy algorithms, and two hybrids between these, in terms of their optimisation quality for the multi objective test case prioritisation problem. The empirical study uses both standard benchmark programs and a large real world open source software with over million lines of code and real faults: `mysql`.
- An evaluation of 7 prioritisation algorithms with respect to the rate of early fault detection, which is the aim of test case prioritisation. The empirical study uses the widely-studied cost cognisant Average Percentage of Fault Detection (APFD$_c$) metric to evaluate the rate of fault detection.
- The introduction and evaluation of a novel coverage compaction algorithm, which achieves up to 4 orders of magnitude speed-up for both MOEAs and greedy algorithms when applied to the coverage traces of a large system.

The rest of the paper is structured as follows. Section 2 describes the multi objective test case prioritisation problem and the coverage compaction. Section 3 outlines the research questions and how they are answered. Section 4 describes the details of the experimental setup. Section 5 presents and analyses the result of the empirical evaluation, and Section 6 discusses the threats to validity. Section 7 presents the related work, and Section 8 concludes.

## 2. MULTI OBJECTIVE TEST CASE PRIORITISATION

### 2.1 Single Objective Formulation

The aim of test case prioritisation is to find the ordering of test cases that will help the tester to achieve the maximum benefit, even if the testing procedure is prematurely halted. More formally, the Test Case Prioritisation problem can be defined as follows [32]:

DEFINITION 1. *Test Case Prioritisation Problem: Given a test suite $T$, a set of all permutations, $\Pi$, of $T$, and a function $f : \Pi \rightarrow \mathbb{R}$. The problem is to find a $\pi' \in \Pi$ such that:*

$$(\forall \pi'')(\pi'' \in \Pi)(\pi'' \neq \pi')[f(\pi') \geq f(\pi'')].$$

The set $\Pi$ represents the set of all possible permutations of $T$, and the objective function $f$ maps each ordering to a real number, which should correspond to an *award value* for the ordering under consideration. The ideal award value would represent how early faults are detected. However, this is infeasible because at the time of prioritisation faults are not known. Consequently, some surrogate for fault detection capability, such as structural coverage, is usually used for the objective function, $f$.

### 2.2 Multi Objective Formulation

Multi objective optimisation is based on the notion of Pareto optimality. With multiple objectives, an ordering of test cases $A$ is *better* than another ordering $B$, (or $A$ *dominates* $B$), only when $A$ excels $B$ in at least one objectives while not being worse of than $B$ in all other objectives. More formally, let us assume $M$ different objectives (award functions), $f_i : \Pi \rightarrow \mathbb{R}, (1 \leq i \leq M)$. An ordering $\pi_1$ is said to dominate another ordering $\pi_2$ if and only if the following is satisfied:

$$f_i(\pi_1) \geq f_i(\pi_2), \forall i \in \{1, 2, \ldots, M\} \text{ and}$$

$$\exists i \in \{1, 2, \ldots, M\} : f_i(\pi_1) > f_i(\pi_2)$$

When evolutionary algorithms are applied to single objective test case prioritisation, they produce a single ordering with the maximum fitness value. When applied to multi objective prioritisation, however, they will produce a set of orderings that are not dominated by any other in the population. This set of solutions is said to represent a Pareto front.

## 2.3 Evaluating Test Orderings

The effectiveness of test case prioritisation is measured by the rate of fault detection achieved by the produced ordering of test cases. Rothermel et al. first defined the Average Percentage of Fault Detection (APFD) evaluation metric for test case prioritisation [31], which, intuitively, measures how quickly faults are detected by the given ordering.

Definition 2. *Average Percentage of Fault Detection: Let $T$ be a test suite containing $n$ test cases, and $F$ be a set of $m$ faults detected by $T$. Let $TF_i$ be the first test case in an ordering $\pi$ of $T$ that reveals fault $i$. Given an ordering $\pi$ of the test suite $T$, the Average Percentage of Fault Detection (APFD) metric is defined as follows:*

$$APFD(\pi) = 1 + \frac{\sum_{i=1}^{m} TF_i}{nm} - \frac{1}{2n}$$

Later, Elbaum et al. extended the metric to consider both fault severity and test case execution cost, and defined the cost cognisant version, $APFD_c$ [14]. The APFD metric assumes that all faults have the same severity (i.e. they have equal cost) and all test cases take the equal effort to execute. The cost cognisant version, APFD$_c$, incorporates weightings based on varying fault severities and execution costs.

Definition 3. *Let $t_1, t_2, \ldots, t_n$ be the costs of $n$ test cases, and $f_1, f_2, \ldots, f_m$ be the severities of $m$ detected faults. $APFD_c$ is defined as follows:*

$$APFD_c(\pi) = \frac{\sum_{i=1}^{m} \left( f_i \cdot \left( \sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i} \right) \right)}{\sum_{j=1}^{n} t_j \cdot \sum_{i=1}^{m} f_i}$$

Due to the lack of robust fault severity models for the subject programs, the experiments in this paper use the test execution cost weightings of APFD$_c$ and treat all faults as sharing the same severity (i.e. $f_i = 1$ for $1 \leq i \leq m$), which yields the following:

$$\text{APFD}_c(\pi) = \frac{\sum_{i=1}^{m} \left( \sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i} \right)}{\sum_{j=1}^{n} t_j \cdot m}$$

Since MOEAs will produce multiple solutions (i.e. orderings), it is not possible to evaluate MOEAs based on a single APFD$_c$ value obtained from a single ordering. Instead, the average APFD$_c$ value from all solutions a MOEA contributes to the reference Pareto front have been calculated and used for comparisons (refer to Section 4.4 for the definition of a reference Pareto front).

## 2.4 Objectives

The current study considers a three-objective formulation of test case prioritisation. The objectives used are common surrogates for fault detection capability and have been studied before in the literature, but the trade-off between them under a multi objective formulation has not been considered before. Specifically, the study uses *statement coverage*, the *difference of the statement coverage* between two consecutive versions (hereby referred to as Δ-*coverage*), and the *historical fault information* of the test suite. For all three objectives, the rates of their realisation are measured in a similar way to APFD$_c$. To be cost cognisant, the execution cost of each test cases has been also measured. Let us briefly describe the main characteristics of each objective below.

**Statement Coverage:** one well-known and widely-used surrogate for fault detection capability in regression testing literature is structural coverage [39], which is one of the necessary conditions to detect a fault (i.e. one has to at least execute the faulty statement in order to detect it). Here, statement coverage is used as the surrogate for fault detection capability.

**Δ-coverage:** the second objective used is the information about the *difference of the statement coverage* between two consecutive versions (called Δ-*coverage*). In a regression testing scenario, one may conjecture that new regression faults are likely to originate from the changed parts of the source code in the version under test. Therefore, the coverage of the changed parts, obtained with `diff`, is a rational candidate for prioritisation.

**Fault History Coverage:** in a regression testing scenario, the tester may have information of the test history regarding which test case detected faults in the past. When aggregated over all known faults, this information can be represented in the form of coverage: a test case *covers* a known past fault if it successfully detected the fault. The rational behind this objective is that, if a test case has revealed a fault in the past, it has a better chance to reveal faults in the future. Fault history coverage has been used in previous multi objective test case minimisation [37, 43] with successful results, but has not be used in prioritisation.

**Execution Cost:** one natural candidate for execution cost, the wall-clock time, is not only noisy but also dependant on the underlying hardware and operating system, making it inaccurate and not robust. To alleviate these issues, a widely used software profiling tool called `valgrind` [27] has been used in place of the wall-clock time. `Valgrind` executes the given program in a virtual machine and can report the number of virtual instructions executed. This number provides a precise measurement of the computational effort required to execute each test case. Executions of each test case have been profiled with `Valgrind` and the number of virtual instructions were used as a representative surrogate for execution cost.

Three objective functions for test case prioritisation are defined based on these measures, as variations of APFC$_c$: Average Percentage of Statement Coverage (APSC$_c$), Average Percentage of Δ-Coverage (APDC$_c$), and Average Percentage of Fault Coverage (APFC$_c$). These are all defined by replacing $TF_i$ (i.e. index of the test case that detects fault $f_i$ in the ordering) in Definition 3 with $TS_i$, $TD_i$, and $TH_i$. They represent the index of the test case that covers the $i$th statement in the program, $i$th *changed* statement in the program, and $i$th historical fault, respectively. As with the use of APFD$_c$, the fault severity weights have been discarded. These three rate of objective realisation metrics will be hereby collectively referred to as AP*C$_c$.

## 2.5 Algorithms

The empirical study uses two different Multi Objective Evolutionary Algorithms (MOEAs) that have been previously applied to software engineering problems. The Non-dominated Sorting Genetic Algorithm II (NSGA-II) [9] is one of the most widely studied multi objective optimisation algorithms and has been applied to various domain ranging from Requirement Engineering [15] to regression testing [42]. The Two Archive Evolutionary Algorithm (TAEA) [29] was specifically designed to overcome weaknesses of NSGA-II and has also been applied to regression testing [42].

NSGA-II uses a crowding distance selection mechanism to promote diversity. Intuitively, given a set of non-dominated solutions, crowding distance selection favours the solution farthest away from the rest of the population, in order to promote diversity. It also adopts elitism to achieve fast convergence: the solutions on the Pareto front in current generation are preserved into the next generation.

The Two-Archive evolutionary algorithm is characterised by two separate archives that are used in addition to the population pool. The first is reserved for fast convergence, while the second promotes diversity. During evolution, solutions of two different types are archived: non-dominated solutions that dominate some other solutions are stored in the convergence archive, while those that do not dominate any others are stored in the diversity archive. A pruning procedure is applied when the diversity archive reaches a pre-specified size limit: the solution closest to the convergence archive will be discarded.

The empirical study also includes three different instantiations of the cost cognisant additional greedy algorithm [14, 26], each with one of the three objectives. These algorithms greedily select test cases, attempting to cover statements, modified statements, or past faults per time unit that remain as yet uncovered by test cases that occur earlier in the ordering. Specifically, they initially select a test case that has the highest value of the given coverage per time unit. Subsequently, they try to find the next test case that will increase the given coverage per time unit by the largest amount. This procedure is repeated until the highest possible coverage per time unit is reached, at which point the algorithms are recursively applied to the remaining test cases until all are ordered.

Finally, the paper also introduces two hybridisations between the MOEAs and the additional greedy algorithms. For the hybrid versions, the populations of both MOEAs are seeded with solutions produced by all three additional greedy algorithms. The expectation is that these solutions will speed up the convergence, because the additional greedy algorithms will produce better solutions than the, otherwise, random initialisation.

```
Algorithm 1: Coverage Compaction Algorithm
Input: an $n$ by $m$ binary coverage matrix, $M$
Output: a compacted $M$
(1) $i \leftarrow 0$
(2) while $i < M.width$
(3)     $cc \leftarrow M_{(,i)}$
(4)     for $j = i + 1$ to $M.width - 1$
(5)         if $cc == M_{(,j)}$
(6)             $M_{(,i)} \leftarrow M_{(,j)} + cc$
(7)             delete $M_{(,j)}$ from $M$
(8)     $i \leftarrow i + 1$
(9) return $M$
```

## 2.6 Compact Coverage

The biggest driver of computational cost for population based evolutionary algorithms is the large number of fitness evaluations required to find a solution [42]. Whereas a constructive heuristic such as the additional greedy algorithm forms only a single solution, an evolutionary algorithm, with population size of $p$, running for $l$ generations is guaranteed to perform $lp$ fitness evaluations. Moreover, when applied to the optimisation of a regression test suite with $n$ test cases covering $m$ statements, a single fitness evaluation has the complexity of $O(nm)$: the fitness function iterates over $n$ coverage traces of length $m$, either aggregating them for selected test cases (for test suite minimisation) or tracing the achieved coverage (for test case prioritisation). As a result, evolutionary algorithms applied to regression test suite optimisation includes a fixed computational cost of $O(lmnp)$.

The paper proposes a novel algorithm called *coverage compaction* to reduce this cost. Given a test suite, $n$ is fixed; $l$ and $p$ may have to be tuned to achieve desirable outcome. However, the length of traces, $m$, can be *compacted* without losing information. The compaction is based on the observation that coverage data from structured programs often show highly repetitive patterns, because some statements are executed only by the same subset of test cases, regardless of their location.

### 2.6.1 Coverage Compaction Algorithm

Algorithm 1 presents the compaction algorithm. It takes an $n$ by $m$ binary matrix that represents the coverage of $n$ test cases over $m$ statements, and outputs a compact version of the same matrix. Intuitively, each column in the compact matrix corresponds to a set of statements that are covered by the same subset of test cases. For example, consider the example in Figure 1. If three statements are covered by all test cases in the test suite, the compacted coverage matrix will contain a single column of all threes, instead of three columns of all ones.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & 1 & 2 & 0 \\ 3 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{pmatrix}$$

**Figure 1: Example of Coverage Compaction**

The coverage achieved by any subset of rows (i.e. test cases) can be calculated from the compact matrix. Let us assume that the coverage of the subset of the second and the third test case in Figure 1 needs to be calculated. With a binary coverage matrix, the aggregation of rows is performed by bitwise OR; with the compact matrix, the aggregation is performed by addition of column values, but only when the corresponding column in the accumulated coverage row is still 0. For example, Aggregating the compact trace $(3, 0, 0, 0)$ and $(3, 0, 2, 0)$ therefore results in $(3, 0, 2, 0)$, not $(6, 0, 2, 0)$. The sum of all numbers in the aggregated coverage vector, 5, is the number of statements covered by these two rows. The statement coverage can be calculated using the number of statements, which is 8: $\frac{5}{8} \cdot 100 = 62.5\%$. Notice that the column with only zero values does not contain any useful coverage information and can be discarded.

### 2.6.2 Calculating Objectives with Compact Coverage

Since compact coverage is lossless, the objective functions can be computed directly from it (and will be considerably faster). Let us consider APSC$_c$ for an example. Let the original coverage trace be a $n$ by $m$ matrix, containing coverage traces of $n$ test cases for $m$ statements. Let the compacted matrix have $m'$ columns ($m \leq m$ after compaction). The compact version, CAPSC$_c$, would then be defined as follows:

$$\text{CAPSC}_c(\pi) = \frac{\sum_{i=1}^{m'} c_i \cdot \left( \sum_{j=TS'_i}^{n} t_j - \frac{1}{2} t_{TS'_i} \right)}{\sum_{j=1}^{n} t_j \cdot m}$$

where $TS'_j (1 \leq j \leq n)$ is the index of the first test case that covers the $i$th column of the compacted coverage matrix, and $c_i (1 \leq i \leq m')$ is the non-zero entry in $i$th column. Essentially, whenever the next test case *covers* a column in the compact coverage matrix, in fact $c_i$ statements are being covered in the original program.

## 3. EMPIRICAL STUDY

### 3.1 Research Questions

This empirical study compares the performance of the two multi objective evolutionary algorithms (NSGA-II and TAEA), three instantiations of the single objective cost-cognisant additional greedy algorithm, and two hybrid MOEAs with seeding. The first research question, **RQ1**, concerns the quality of optimisation.

- **RQ1. Optimisation Quality:** Which prioritisation algorithm produces the best solutions in terms of optimisation quality? What are the differences?

**RQ1** is answered by calculating and comparing three widely used quality indicators for multi objective optimisation for the result from each algorithm. These three quality indicators asses the quality of the Pareto front produced by each algorithm, but they do not help the software engineer to determine which is better at finding faults earlier. Therefore, the second research question **RQ2** is formulated as follows:

- **RQ2. Testing Effectiveness:** Which prioritisation algorithm produces the best solutions in terms of prioritisation cost effectiveness? Which algorithm achieves the earliest fault detection?

**RQ2** is answered by calculating and comparing the standard cost effectiveness measure for test case prioritisation, the APFD$_c$ metric. The final research question, **RQ3**, concerns the efficiency of algorithms studied. MOEAs used in this paper are population based algorithms, usually demanding longer execution time due to the large number of fitness evaluation (i.e. calculations of AP*C$_c$ values). **RQ3** compares the execution time of 2 MOEAs and 3 additional greedy algorithms, with and without coverage compaction.

- **RQ3. Effort requirement/Efficiency:** How much effort is required to produce high quality test case prioritisations? How much difference does the coverage compaction make?

**RQ3** is answered by comparing the average system-clock execution time from repeated runs of 2 MOEAs and 3 additional greedy algorithms, both with and without coverage compaction.

## 4. EXPERIMENTAL SETUP

### 4.1 Subjects

The studied prioritisation techniques are evaluated using six `C/C++` open source programs as subjects. Five subject programs are Unix utilities taken from the widely-used Software-artifact Infrastructure Repository (SIR) [3,10]: `flex`, `grep`, `gzip`, `make`, and `sed`. The sixth subject program is `mysql`, one of the most popular open source relational database management systems [28].

**Table 1: Size of versions of subject programs in Lines of Code**

| Object | V1 | V2 | V3 | V4 | V5 | V6 | V7 |
|---|---|---|---|---|---|---|---|
| `flex` | 9,484 | 10,217 | 10,243 | 11,401 | 10,332 | N/A | N/A |
| `grep` | 9,400 | 9,977 | 10,066 | 10,107 | 10,102 | N/A | N/A |
| `gzip` | 4,528 | 5,055 | 5,066 | 5,185 | 5,689 | N/A | N/A |
| `make` | 14,357 | 28,988 | 30,316 | 35,564 | N/A | N/A | N/A |
| `sed` | 5,488 | 9,799 | 7,082 | 7,085 | 13,374 | 13,393 | 14,437 |
| | V0 | V1 | V2 | | | | |
| `mysql` | 1,282,282 | 1,283,361 | 1,283,504 | | | | |

Table 1 presents the size of the subject programs in Lines of Code (LOC), measured with the `cloc` [1]. All subjects have multiple consecutive versions available. The versions of the Unix tools are provided by the SIR. The three versions of `mysql` are obtained from the 5.5.X source tree: `V0` corresponds to 5.5.15, `V1` and `V2` to 5.5.16 and 5.5.17 respectively.

In total, 29 versions of six subject programs have been used to create the data required to evaluate the proposed methodologies in 22 different test case prioritisation instances. The number of instances is 22 (not 29) because the data from the previous version is required for prioritisation (i.e. prioritising `V1`s is not possible). In addition, `v0` of `mysql` was used as the baseline to collect the regression faults in `v1`.

**Table 2: Test suite size and the average number of faults per version**

| Subject: | `flex` | `grep` | `gzip` | `make` | `sed` | `mysql` |
|---|---|---|---|---|---|---|
| #Test Cases | 567 | 809 | 214 | 1,043 | 360 | 2,005 |
| #Faults | 16.6 | 11.4 | 11.8 | 8.75 | 4.57 | 20 |

Table 2 reports the size of the test suites and the average number of faults per version. Subject programs from SIR provide test suites described in Test suite Specification Language (TSL). SIR also provides versions of programs with seeded faults, descriptions of which are avaialble from SIR [3,10].

The source code of `mysql` comes with a testing framework that provides a set of test cases and the infrastructure required for their execution. Test suites for `V0` is used as the regression test suite for all subsequent versions [4]. In total, 20 real faults were manually collected from the online bug-tracking system used by `mysql` community [2]. The faults used in this paper are those with "closed" status and fix patches. These faults were then seeded back to the source code of the corresponding version by inverting and applying the fix patch.

### 4.2 MOEA Configuration

NSGA-II and TAEA share the same configuration to facilitate a fair comparison. The population size is 250. The chosen genetic operators are ones that are widely used for permutation type representation: Partially Matched Crossover (PMX) and swap mutation, as well as binary tournament selection [16,17]. The crossover rate is set to 0.9, and the mutation rate is set to $\frac{1}{n}$ where $n$ is the number of test cases. The termination criterion for both algorithms is based on the maximum available budget of fitness evaluations, which is fixed to $25,000$ for SIR subjects and $50,000$ for `mysql`. Finally, the archive size in TAEA is set to twice the size of the population. To cater for the stochastic nature of algorithms, MOEAs and hybrids are executed 30 times.

## 4.3 Measurements & Environment

Statement coverage data is obtained using the GNU `gcov` profiling tool. The $\Delta$-coverage is generated by combining the statement coverage information with the results of Unix `diff` tool applied to two consecutive versions. The execution cost of each test case is measured using the `valgrind` profiling tool, as discussed in Section 2.4.

For SIR objects, the execution time of the additional greedy algorithm as well as the MOEAs was measured on a machine equipped with AMD Opteron CPU and 16 GB of RAM, running CentOS Linux 6.4. For `mysql`, the same system could not execute neither greedy algorithms nor MOEAs due to insufficient memory; consequently, the execution time of all algorithms was measured on a cluster node equipped with Intel Xeon X7500 CPU and 1024 GB of RAM, running CentOS Linux 6.5. Execution time was measured using system clock.

## 4.4 MOEA Quality Indicators

The main goal of a MOEA is to find the *true* Pareto front of the given multi objective optimisation problem. However, it is usually infeasible to obtain the true Pareto front. The output of MOEAs are usually approximations of the true Pareto front. There are two ways to evaluate such approximations. First, the approximation should be as close as possible to the *true* Pareto front (convergence). Second, the acquired solutions should be as diverse as possible (diversity). The proximity to the true Pareto front ensures high quality of the found solutions, whereas the diversity indicates the search space has been explored as thoroughly as possible to present the decision maker with a variety of solutions.

However, the convergence to the true Pareto front cannot be measured, simply because the true front is not known. In practice, *reference* Pareto front is used as a surrogate. A reference Pareto front consists of the best non-dominated solutions found by all evaluated algorithms (in the case of this paper, all 7 algorithms). Formally, the reference Pareto front, $P_{\text{ref}}$, can be defined as follows:

Definition 4. *Reference Pareto Front: Let us assume that there exist N different Pareto fronts, $P_i (i = 1, 2, \dots, N)$, and the union of all $P_i$, $P_{\text{U}}$. The reference Pareto front, $P_{\text{ref}}$, is defined as: $P_{\text{ref}} \subset P_{\text{U}} : (\forall p \in P_{\text{ref}})(\nexists q \in P_{\text{U}})(q \succ p)$, where $\succ$ is the Pareto dominance relation.*

To answer **RQ1**, three widely studied MOEA quality indicators have been used: EPSILON, Inverted Generational Distance (IGD), and Hyper Volume (HV). EPSILON and IGD are distance based indicators that measure convergence, while HV measures diversity of a solution set. Given a Pareto front $A$ and a reference Pareto front:

- **EPSILON** measures the shortest distance that is required to transform every solution in $A$ so that it dominates the reference Pareto front [23].
- **IGD** is the average distance from solutions in the reference Pareto front to the closest solution in $A$ [36].
- **HV** is the volume of objective space dominated by solutions in $A$ [45].

With EPSILON and IGD, the lower the indicator value is, the closer $A$ is to the reference Pareto front, which adds confidence to its convergence to the true front. With HV, the higher the indicator value is, the more diverse the solutions in $A$ is (as they collectively dominate larger volume).

## 4.5 Statistical Tests

Non-parametric Wilcoxon-signed rank tests [5, 18] have been used to assess the statistical significance of differences observed in the quality indicators (the Shapiro-Wilk normality test [33] does not report evidence of normality in the samples). The null hypothesis is that the median difference between two sets of quality indicator values is zero; the alternative hypothesis is that the algorithms produce different median quality indicator values. The significance level is 0.05. In addition, the standard Bonferroni adjustment [5] have been applied to address the problem of the higher probability of Type I errors in multiple comparisons: the adjusted p-value, $p_{Bonf}$, are reported in Section 5. This is conservative, but safe, because it avoids Type I errors.

Furthermore, a non-parametric effect size measure, called Vargha and Delaney's $\hat{A}_{12}$ statistic [35], was used to assess the magnitude of any observed improvements. Intuitively, given a quality indicator measure $I$ and two algorithms $A_1$ and $A_2$ for comparison, the $\hat{A}_{12}$ measures the probability that $I(A_1)$ yields a higher value than $I(A_2)$. For example, $\hat{A}_{12} = 0.8$ suggests that algorithm $A_1$ will outperform $A_2$ in 80% of the runs. If two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. According to Vargha and Delaney [35], differences between populations can be characterised as small, medium and large when $\hat{A}_{12}$ is over 0.56, 0.64, and 0.71, respectively.

## 5. RESULTS & DISCUSSION

## 5.1 Optimisation Quality

Let us first consider the optimisation quality of the studied algorithms. Table 3 presents the complete descriptive statistics on the quantitative optimisation quality from the NSGA-II (denoted 'N'), the TAEA ('T'), the cost-cognisant additional greedy algorithms based on: statement coverage ('C'), $\Delta$-coverage ('D'), and fault history coverage ('F'), and the two hybrid algorithms with seeded solutions from the greedy approaches, NSGA-II ('NS') and TAEA ('TS'). Each column in Table 3 shows the quality indicators (EPSILON, IGD, and HV) obtained by applying all the studied algorithms on 22 versions of the subjects. The best performing cases are highlighted with **boldface** (Note that lower values indicate better performance for EPSILON and the IGD, while the opposite holds for the HV). Overall, seeded MOEAs tend to outperform others.

The first three major columns of Table 4 present a summary of the statistical significance of the differences in each quality indicator, observed between each pair of algorithms ($\alpha = 0.05$). Due to limited space full information of the statistical analysis ($p$-values, effect sizes) are provided as supplementary material in[1]. For each comparison of algorithm pair ($A$, $B$), columns $p$-value and $p_{\text{Bonf}}$ contain the number of subject versions for which the performance of $A$ is significantly superior to ('+'), equal to ('='), or inferior to ('–') that of $B$. In addition, column $\hat{A}_{12}$ contains median ($m$), mean ($\mu$), and standard deviation ($\sigma$) of the $\hat{A}_{12}$ effect size metric: large effect sizes are highlighted with **boldface** and medium effect sizes with <u>underlined</u> fonts. Boxplots of the effect size for each quality indicator are illustrated in Figure 2(a), 2(b), and 2(c).

The results answer **RQ1** in favour of MOEAs. MOEAs and their hybrid variants outperformed all variations of the

[1] `http://www.epitropakis.co.uk/issta2015/`

**Table 3: Descriptive statistics of EPSILON, HV and IGD indicators. For each subject version, the mean ($\mu$) and standard deviation ($\sigma$) of the indicators are presented for the NSGA-II (N), the TAEA (T), the additional greedy algorithms based on, statement coverage (C), $\Delta$-coverage (D), and fault history coverage (F), and the two hybrid MOEAs based on NSGA-II (NS) and TAEA (TS).**

| | | EPSILON | | HV | | IGD | |
|---|---|---|---|---|---|---|---|
| | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| flex v2 | N | 0.006 | 0.001 | 0.670 | 0.053 | 0.009 | 0.001 |
| | NS | 0.006 | 0.001 | 0.688 | 0.042 | 0.008 | 0.001 |
| | T | **0.004** | 0.002 | 0.765 | 0.049 | **0.007** | 0.001 |
| | TS | **0.004** | 0.003 | **0.787** | 0.089 | **0.007** | 0.002 |
| | C | 0.130 | 0.000 | 0.000 | 0.000 | 0.589 | 0.000 |
| | D | 0.131 | 0.000 | 0.000 | 0.000 | 0.633 | 0.000 |
| | F | 0.128 | 0.000 | 0.000 | 0.000 | 0.498 | 0.000 |
| flex v3 | N | 0.003 | 0.001 | 0.560 | 0.059 | 0.022 | 0.003 |
| | NS | 0.003 | 0.001 | 0.618 | 0.073 | 0.019 | 0.004 |
| | T | **0.001** | 0.001 | 0.658 | 0.091 | **0.018** | 0.005 |
| | TS | **0.001** | 0.001 | **0.668** | 0.083 | **0.018** | 0.004 |
| | C | 0.026 | 0.000 | 0.000 | 0.000 | 1.581 | 0.000 |
| | D | 0.030 | 0.000 | 0.000 | 0.000 | 1.311 | 0.000 |
| | F | 0.029 | 0.000 | 0.000 | 0.000 | 1.755 | 0.000 |
| flex v4 | N | **0.003** | 0.001 | 0.161 | 0.128 | 0.022 | 0.009 |
| | NS | **0.003** | 0.001 | **0.269** | 0.147 | **0.016** | 0.005 |
| | T | **0.003** | 0.003 | 0.182 | 0.235 | 0.037 | 0.035 |
| | TS | 0.004 | 0.004 | 0.179 | 0.231 | 0.053 | 0.074 |
| | C | 0.026 | 0.000 | 0.000 | 0.000 | 1.286 | 0.000 |
| | D | 0.032 | 0.000 | 0.000 | 0.000 | 1.294 | 0.000 |
| | F | 0.031 | 0.000 | 0.000 | 0.000 | 1.368 | 0.000 |
| flex v5 | N | 0.002 | 0.001 | 0.040 | 0.088 | 0.101 | 0.031 |
| | NS | **0.001** | 0.001 | **0.227** | 0.205 | **0.063** | 0.023 |
| | T | 0.004 | 0.002 | 0.012 | 0.061 | 0.199 | 0.113 |
| | TS | 0.004 | 0.002 | 0.047 | 0.151 | 0.181 | 0.086 |
| | C | 0.026 | 0.000 | 0.000 | 0.000 | 2.827 | 0.000 |
| | D | 0.029 | 0.000 | 0.000 | 0.000 | 1.698 | 0.000 |
| | F | 0.033 | 0.000 | 0.000 | 0.000 | 1.860 | 0.000 |
| grep v2 | N | 0.004 | 0.001 | 0.623 | 0.064 | 0.008 | 0.001 |
| | NS | 0.004 | 0.001 | 0.637 | 0.057 | 0.008 | 0.001 |
| | T | **0.003** | 0.001 | 0.755 | 0.120 | **0.007** | 0.002 |
| | TS | **0.003** | 0.001 | **0.759** | 0.098 | **0.007** | 0.002 |
| | C | 0.111 | 0.000 | 0.000 | 0.000 | 0.094 | 0.000 |
| | D | 0.113 | 0.000 | 0.000 | 0.000 | 0.096 | 0.000 |
| | F | 0.086 | 0.000 | 0.000 | 0.000 | 0.100 | 0.000 |
| gzip v4 | N | **0.000** | 0.000 | 0.115 | 0.139 | 0.031 | 0.008 |
| | NS | **0.000** | 0.000 | 0.425 | 0.090 | 0.015 | 0.003 |
| | T | **0.000** | 0.000 | 0.372 | 0.206 | 0.019 | 0.017 |
| | TS | **0.000** | 0.000 | **0.525** | 0.119 | **0.011** | 0.004 |

| | | EPSILON | | HV | | IGD | |
|---|---|---|---|---|---|---|---|
| | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| grep v3 | N | 0.004 | 0.001 | 0.607 | 0.045 | 0.009 | 0.001 |
| | NS | 0.004 | 0.001 | 0.614 | 0.050 | 0.009 | 0.001 |
| | T | **0.003** | 0.001 | 0.736 | 0.052 | **0.007** | 0.001 |
| | TS | **0.003** | 0.001 | **0.750** | 0.069 | **0.007** | 0.001 |
| | C | 0.067 | 0.000 | 0.000 | 0.000 | 0.174 | 0.000 |
| | D | 0.069 | 0.000 | 0.000 | 0.000 | 0.179 | 0.000 |
| | F | 0.038 | 0.000 | 0.000 | 0.000 | 0.102 | 0.000 |
| grep v4 | N | 0.004 | 0.001 | 0.777 | 0.035 | 0.004 | 0.001 |
| | NS | 0.004 | 0.000 | 0.779 | 0.021 | 0.004 | 0.000 |
| | T | **0.002** | 0.000 | 0.852 | 0.032 | **0.003** | 0.000 |
| | TS | **0.002** | 0.001 | **0.854** | 0.048 | **0.003** | 0.000 |
| | C | 0.091 | 0.000 | 0.000 | 0.000 | 0.131 | 0.000 |
| | D | 0.098 | 0.000 | 0.000 | 0.000 | 0.115 | 0.000 |
| | F | 0.039 | 0.000 | 0.000 | 0.000 | 0.092 | 0.000 |
| grep v5 | N | 0.003 | 0.001 | 0.000 | 0.000 | 0.128 | 0.029 |
| | NS | **0.002** | 0.001 | 0.024 | 0.114 | **0.059** | 0.022 |
| | T | 0.003 | 0.001 | 0.021 | 0.055 | 0.101 | 0.057 |
| | TS | **0.002** | 0.002 | **0.040** | 0.097 | 0.088 | 0.084 |
| | C | 0.022 | 0.000 | 0.000 | 0.000 | 1.995 | 0.000 |
| | D | 0.025 | 0.000 | 0.000 | 0.000 | 2.417 | 0.000 |
| | F | 0.024 | 0.000 | 0.000 | 0.000 | 0.986 | 0.000 |
| gzip v2 | N | **0.000** | 0.000 | 0.936 | 0.011 | 0.002 | 0.003 |
| | NS | **0.000** | 0.000 | 0.942 | 0.009 | **0.001** | 0.000 |
| | T | **0.000** | 0.000 | 0.924 | 0.031 | 0.003 | 0.004 |
| | TS | **0.000** | 0.000 | **0.944** | 0.009 | **0.001** | 0.000 |
| | C | 0.013 | 0.000 | 0.009 | 0.000 | 0.024 | 0.000 |
| | D | 0.013 | 0.000 | 0.000 | 0.000 | 0.024 | 0.000 |
| | F | 0.013 | 0.000 | 0.000 | 0.000 | 0.020 | 0.000 |
| gzip v3 | N | **0.000** | 0.000 | 0.160 | 0.148 | 0.018 | 0.005 |
| | NS | **0.000** | 0.000 | 0.631 | 0.103 | 0.005 | 0.002 |
| | T | **0.000** | 0.000 | 0.582 | 0.163 | 0.007 | 0.004 |
| | TS | **0.000** | 0.000 | **0.744** | 0.090 | **0.003** | 0.001 |
| | C | 0.125 | 0.000 | 0.000 | 0.000 | 0.088 | 0.000 |
| | D | 0.126 | 0.000 | 0.000 | 0.000 | 0.074 | 0.000 |
| | F | 0.116 | 0.000 | 0.000 | 0.000 | 0.165 | 0.000 |
| gzip v4 | C | 0.038 | 0.000 | 0.000 | 0.000 | 0.143 | 0.000 |
| | D | 0.053 | 0.000 | 0.000 | 0.000 | 0.143 | 0.000 |
| | F | 0.129 | 0.000 | 0.000 | 0.000 | 0.177 | 0.000 |

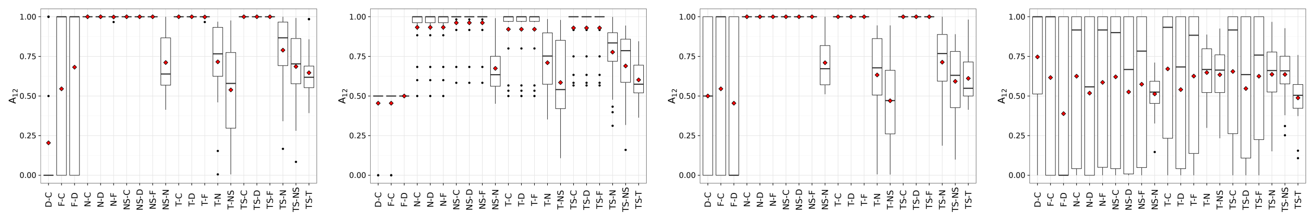
| | | EPSILON | | HV | | IGD | |
|---|---|---|---|---|---|---|---|
| | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| gzip v5 | N | **0.000** | 0.000 | **0.962** | 0.023 | **0.002** | 0.001 |
| | NS | **0.000** | 0.000 | 0.956 | 0.031 | **0.002** | 0.001 |
| | T | **0.000** | 0.000 | 0.949 | 0.048 | 0.003 | 0.001 |
| | TS | **0.000** | 0.000 | 0.917 | 0.066 | 0.003 | 0.001 |
| | C | 0.009 | 0.000 | 0.382 | 0.000 | 0.010 | 0.000 |
| | D | 0.019 | 0.000 | 0.000 | 0.000 | 0.021 | 0.000 |
| | F | 0.044 | 0.000 | 0.000 | 0.000 | 0.048 | 0.000 |
| make v2 | N | 0.003 | 0.001 | 0.140 | 0.120 | 0.024 | 0.005 |
| | NS | **0.002** | 0.000 | 0.251 | 0.198 | 0.019 | 0.007 |
| | T | 0.003 | 0.001 | 0.227 | 0.224 | 0.021 | 0.009 |
| | TS | **0.002** | 0.001 | **0.371** | 0.272 | **0.016** | 0.008 |
| | C | 0.103 | 0.000 | 0.000 | 0.000 | 1.883 | 0.000 |
| | D | 0.103 | 0.000 | 0.000 | 0.000 | 1.883 | 0.000 |
| | F | 0.116 | 0.000 | 0.000 | 0.000 | 2.116 | 0.000 |
| make v3 | N | 0.002 | 0.001 | 0.637 | 0.076 | 0.021 | 0.004 |
| | NS | 0.002 | 0.001 | 0.765 | 0.070 | 0.014 | 0.003 |
| | T | 0.002 | 0.001 | 0.675 | 0.204 | 0.022 | 0.011 |
| | TS | **0.001** | 0.001 | **0.874** | 0.067 | **0.012** | 0.004 |
| | C | 0.037 | 0.000 | 0.000 | 0.000 | 0.185 | 0.000 |
| | D | 0.037 | 0.000 | 0.000 | 0.000 | 0.200 | 0.000 |
| | F | 0.067 | 0.000 | 0.000 | 0.000 | 0.489 | 0.000 |
| make v4 | N | **0.000** | 0.000 | **0.025** | 0.060 | 0.113 | 0.086 |
| | NS | **0.000** | 0.000 | 0.080 | 0.155 | **0.098** | 0.068 |
| | T | 0.002 | 0.001 | 0.000 | 0.000 | 1.814 | 0.955 |
| | TS | **0.000** | 0.000 | 0.014 | 0.037 | 0.168 | 0.078 |
| | C | 0.551 | 0.000 | 0.000 | 0.000 | 6.873 | 0.000 |
| | D | 0.551 | 0.000 | 0.000 | 0.000 | 6.873 | 0.000 |
| | F | 0.016 | 0.000 | 0.000 | 0.000 | 15.297 | 0.000 |
| sed v2 | N | **0.002** | 0.000 | 0.613 | 0.064 | 0.021 | 0.005 |
| | NS | **0.002** | 0.000 | 0.633 | 0.059 | 0.019 | 0.004 |
| | T | **0.002** | 0.001 | 0.734 | 0.062 | **0.013** | 0.005 |
| | TS | **0.002** | 0.001 | **0.737** | 0.103 | 0.014 | 0.007 |
| | C | 0.034 | 0.000 | 0.000 | 0.000 | 0.403 | 0.000 |
| | D | 0.035 | 0.000 | 0.000 | 0.000 | 0.210 | 0.000 |
| | F | 0.041 | 0.000 | 0.000 | 0.000 | 0.337 | 0.000 |
| mysql v2 | N | 0.008 | 0.003 | 0.804 | 0.047 | 0.010 | 0.004 |
| | NS | 0.002 | 0.000 | **0.895** | 0.031 | **0.003** | 0.001 |
| | T | 0.008 | 0.004 | 0.807 | 0.154 | 0.010 | 0.006 |
| | TS | **0.001** | 0.001 | 0.866 | 0.170 | 0.004 | 0.005 |

| | | EPSILON | | HV | | IGD | |
|---|---|---|---|---|---|---|---|
| | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| sed v3 | N | 0.003 | 0.001 | 0.690 | 0.038 | **0.006** | 0.001 |
| | NS | 0.003 | 0.001 | 0.676 | 0.046 | **0.006** | 0.001 |
| | T | **0.002** | 0.001 | **0.759** | 0.042 | **0.006** | 0.001 |
| | TS | 0.003 | 0.001 | 0.736 | 0.047 | **0.006** | 0.001 |
| | C | 0.211 | 0.000 | 0.000 | 0.000 | 0.395 | 0.000 |
| | D | 0.210 | 0.000 | 0.000 | 0.000 | 0.396 | 0.000 |
| | F | 0.139 | 0.000 | 0.000 | 0.000 | 0.322 | 0.000 |
| sed v4 | N | **0.001** | 0.000 | 0.294 | 0.137 | 0.024 | 0.007 |
| | NS | **0.001** | 0.000 | 0.378 | 0.132 | **0.020** | 0.006 |
| | T | **0.001** | 0.001 | **0.459** | 0.267 | 0.022 | 0.020 |
| | TS | **0.001** | 0.001 | 0.449 | 0.280 | 0.021 | 0.019 |
| | C | 0.110 | 0.000 | 0.000 | 0.000 | 1.044 | 0.000 |
| | D | 0.123 | 0.000 | 0.000 | 0.000 | 1.475 | 0.000 |
| | F | 0.052 | 0.000 | 0.000 | 0.000 | 0.845 | 0.000 |
| sed v5 | N | 0.011 | 0.004 | 0.771 | 0.041 | **0.005** | 0.001 |
| | NS | 0.010 | 0.004 | 0.778 | 0.036 | **0.005** | 0.001 |
| | T | 0.011 | **0.004** | 0.811 | 0.042 | **0.005** | 0.001 |
| | TS | 0.009 | 0.005 | **0.831** | 0.052 | **0.005** | 0.001 |
| | C | 0.173 | 0.000 | 0.000 | 0.000 | 0.192 | 0.000 |
| | D | 0.173 | 0.000 | 0.000 | 0.000 | 0.187 | 0.000 |
| | F | 0.172 | 0.000 | 0.000 | 0.000 | 0.154 | 0.000 |
| sed v6 | N | 0.002 | 0.000 | 0.569 | 0.083 | 0.019 | 0.004 |
| | NS | 0.002 | 0.000 | 0.634 | 0.073 | 0.016 | 0.004 |
| | T | **0.001** | 0.000 | 0.772 | 0.080 | **0.012** | 0.006 |
| | TS | **0.001** | 0.000 | **0.803** | 0.078 | **0.012** | 0.005 |
| | C | 0.029 | 0.000 | 0.000 | 0.000 | 0.384 | 0.000 |
| | D | 0.034 | 0.000 | 0.000 | 0.000 | 0.455 | 0.000 |
| | F | 0.027 | 0.000 | 0.000 | 0.000 | 0.359 | 0.000 |
| sed v7 | N | **0.002** | 0.000 | 0.843 | 0.022 | **0.003** | 0.001 |
| | NS | **0.002** | 0.000 | 0.848 | 0.027 | **0.003** | 0.001 |
| | T | **0.002** | 0.000 | 0.866 | 0.027 | **0.003** | 0.001 |
| | TS | **0.002** | 0.000 | **0.881** | 0.025 | **0.003** | 0.001 |
| | C | 0.068 | 0.000 | 0.000 | 0.000 | 0.077 | 0.000 |
| | D | 0.070 | 0.000 | 0.000 | 0.000 | 0.067 | 0.000 |
| | F | 0.049 | 0.000 | 0.000 | 0.000 | 0.079 | 0.000 |
| mysql v2 | C | 0.240 | 0.000 | 0.000 | 0.000 | 0.125 | 0.000 |
| | D | 0.195 | 0.000 | 0.000 | 0.000 | 0.121 | 0.000 |
| | F | 0.017 | 0.000 | 0.000 | 0.000 | 0.055 | 0.000 |

**Table 4: A summary of the statistical significance of differences between each pair of algorithms. Column $p$-value (Wilcoxon test) and $p_{\text{Bonf}}$ (the Bonferroni adjustment) contain the number of cases where algorithm $A$ is significantly superior (+), equal (=), or inferior (–) to algorithm $B$. Column $\hat{A}_{12}$ contains the median ($m$), mean ($\mu$) and standart deviation ($\sigma$) of the $\hat{A}_{12}$ effect size metric. MOEAs consistently outperform the additional greedy algorithms in EPSILON, HV, and IGD, while matching or outperforming the APFD$_c$ of the additional greedy algorithms for the majority of the cases.**

| | EPSILON | | | | | HV | | | | | IGD | | | | | APFD$_c$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $p$-value | $p_{\text{Bonf}}$ | $\hat{A}_{12}$ | | | $p$-value | $p_{\text{Bonf}}$ | $\hat{A}_{12}$ | | | $p$-value | $p_{\text{Bonf}}$ | $\hat{A}_{12}$ | | | $p$-value | $p_{\text{Bonf}}$ | $\hat{A}_{12}$ | | |
| A vs B | +/=/– | +/=/– | $m$ | $\mu$ | $\sigma$ | +/=/– | +/=/– | $m$ | $\mu$ | $\sigma$ | +/=/– | +/=/– | $m$ | $\mu$ | $\sigma$ | +/=/– | +/=/– | $m$ | $\mu$ | $\sigma$ |
| D vs C | 4/1/17 | 4/1/17 | 0.000 | 0.205 | 0.398 | 0/20/2 | 0/20/2 | 0.500 | 0.455 | 0.147 | 11/0/11 | 11/0/11 | 0.500 | 0.500 | 0.512 | 15/2/5 | 15/3/4 | **1.000** | **0.747** | 0.404 |
| F vs C | 12/0/10 | 12/0/10 | **1.000** | 0.545 | 0.510 | 0/20/2 | 0/20/2 | 0.500 | 0.455 | 0.147 | 12/0/10 | 12/0/10 | **1.000** | 0.545 | 0.510 | 13/1/8 | 13/1/8 | **1.000** | 0.616 | 0.486 |
| F vs D | 15/0/7 | 15/0/7 | **1.000** | <u>0.682</u> | 0.477 | 0/22/0 | 0/22/0 | 0.500 | 0.500 | 0.000 | 10/0/12 | 10/0/12 | 0.000 | 0.455 | 0.510 | 8/1/13 | 8/1/13 | 0.000 | 0.389 | 0.486 |
| N vs C | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.933** | 0.144 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 13/1/8 | 13/1/8 | **0.917** | 0.625 | 0.453 |
| N vs D | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.933** | 0.144 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 10/2/10 | 10/2/10 | 0.558 | 0.518 | 0.460 |
| N vs F | 22/0/0 | 22/0/0 | **1.000** | **0.998** | 0.007 | 19/3/0 | 19/3/0 | **1.000** | **0.933** | 0.144 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 13/0/9 | 12/2/8 | **0.917** | 0.586 | 0.454 |
| NS vs C | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 20/2/0 | 20/2/0 | **1.000** | **0.962** | 0.109 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 13/1/8 | 12/2/8 | **0.900** | 0.621 | 0.453 |
| NS vs D | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 20/2/0 | 20/2/0 | **1.000** | **0.962** | 0.109 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 10/2/10 | 10/2/10 | <u>0.667</u> | 0.526 | 0.466 |
| NS vs F | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 20/2/0 | 20/2/0 | **1.000** | **0.962** | 0.109 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 11/2/9 | 11/3/8 | **0.783** | 0.574 | 0.453 |
| NS vs N | 11/11/0 | 10/12/0 | <u>0.639</u> | <u>0.711</u> | 0.198 | 10/12/0 | 7/15/0 | 0.634 | <u>0.674</u> | 0.160 | 12/10/0 | 8/14/0 | <u>0.672</u> | **0.710** | 0.171 | 2/18/2 | 0/21/1 | 0.525 | 0.513 | 0.128 |
| T vs C | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.921** | 0.164 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 14/2/6 | 12/4/6 | **0.933** | <u>0.671</u> | 0.409 |
| T vs D | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.921** | 0.164 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 12/1/9 | 10/3/9 | <u>0.683</u> | 0.540 | 0.454 |
| T vs F | 22/0/0 | 22/0/0 | **1.000** | **0.998** | 0.007 | 19/3/0 | 19/3/0 | **1.000** | **0.921** | 0.164 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 12/3/7 | 11/5/6 | **0.883** | 0.626 | 0.422 |
| T vs N | 13/7/2 | 12/8/2 | **0.766** | **0.716** | 0.257 | 13/8/1 | 11/11/0 | **0.752** | **0.710** | 0.211 | 11/8/3 | 7/13/2 | <u>0.678</u> | 0.633 | 0.255 | 13/8/1 | 6/16/0 | <u>0.667</u> | <u>0.648</u> | 0.172 |
| T vs NS | 9/6/7 | 7/10/5 | 0.579 | 0.539 | 0.318 | 9/10/3 | 8/11/3 | 0.541 | 0.585 | 0.268 | 6/8/8 | 5/10/7 | 0.471 | 0.470 | 0.290 | 9/12/1 | 7/15/0 | <u>0.664</u> | 0.634 | 0.189 |
| TS vs C | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.930** | 0.147 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 13/1/8 | 12/3/7 | **0.917** | <u>0.655</u> | 0.404 |
| TS vs D | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.930** | 0.147 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 11/1/10 | 11/1/10 | 0.635 | 0.547 | 0.441 |
| TS vs F | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 19/3/0 | 19/3/0 | **1.000** | **0.930** | 0.147 | 22/0/0 | 22/0/0 | **1.000** | **1.000** | 0.000 | 11/5/6 | 10/6/6 | **0.758** | 0.625 | 0.417 |
| TS vs N | 15/6/1 | 13/8/1 | **0.867** | **0.789** | 0.226 | 17/4/1 | 15/7/0 | **0.835** | **0.777** | 0.203 | 14/6/2 | 14/7/1 | **0.768** | **0.713** | 0.229 | 11/9/2 | 7/14/1 | 0.661 | 0.637 | 0.207 |
| TS vs NS | 12/7/3 | 10/11/1 | <u>0.703</u> | <u>0.686</u> | 0.229 | 13/7/2 | 12/9/1 | **0.786** | <u>0.690</u> | 0.224 | 10/8/4 | 7/12/3 | 0.631 | 0.593 | 0.229 | 10/10/2 | 5/17/0 | <u>0.659</u> | 0.636 | 0.169 |
| TS vs T | 6/16/0 | 5/17/0 | 0.618 | <u>0.647</u> | 0.159 | 7/15/0 | 4/18/0 | 0.574 | 0.602 | 0.133 | 7/15/0 | 5/17/0 | 0.549 | 0.611 | 0.164 | 1/19/2 | 0/20/2 | 0.504 | 0.488 | 0.150 |

additional greedy algorithms, for all subject versions, in terms of the quality indicator metrics. The additional greedy algorithms fail to produce as many solutions on or close to the reference Pareto front, evidenced by the high EPSILON and IGD values. HV values of the additional greedy algorithms are mostly nearly zero. This is as expected because, being single objective algorithms, they do not optimise for all three objectives.



(a) Epsilon (b) Hypervolume (c) IGD (d) APFD$_c$

**Figure 2: Boxplots of $\hat{A}_{12}$ for the Optimization Quality Indicators and the APFD$_c$ metric**

**Table 5: The mean ($\mu$), standard deviation ($\sigma$), and the maximum value ($\max$) of the APFD$_c$ measure for 22 subjects.**

| | Alg. | $\mu$ | $\sigma$ | max | | Alg. | $\mu$ | $\sigma$ | max | | Alg. | $\mu$ | $\sigma$ | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `flex v2` | N | **0.999** | 0.000 | **0.999** | `grep v2` | N | 0.980 | 0.004 | 0.986 | `gzip v2` | N | **1.000** | 0.001 | **1.000** |
| | NS | **0.999** | 0.000 | **0.999** | | NS | 0.981 | 0.004 | 0.993 | | NS | **1.000** | 0.000 | **1.000** |
| | T | **0.999** | 0.000 | **0.999** | | T | **0.986** | 0.005 | **0.995** | | T | **1.000** | 0.000 | **1.000** |
| | TS | **0.999** | 0.000 | **0.999** | | TS | 0.985 | 0.007 | **0.995** | | TS | **1.000** | 0.000 | **1.000** |
| | C | 0.974 | 0.000 | 0.974 | | C | 0.930 | 0.000 | 0.930 | | C | **1.000** | 0.000 | **1.000** |
| | D | 0.973 | 0.000 | 0.973 | | D | 0.931 | 0.000 | 0.931 | | D | **1.000** | 0.000 | **1.000** |
| | F | 0.978 | 0.000 | 0.978 | | F | 0.913 | 0.000 | 0.913 | | F | **1.000** | 0.000 | **1.000** |
| `flex v3` | N | 0.987 | 0.006 | 0.995 | `grep v3` | N | 0.982 | 0.004 | 0.990 | `gzip v3` | N | **1.000** | 0.000 | **1.000** |
| | NS | 0.989 | 0.006 | 0.995 | | NS | **0.983** | 0.003 | **0.992** | | NS | **1.000** | 0.000 | **1.000** |
| | T | 0.993 | 0.003 | **0.997** | | T | 0.982 | 0.004 | 0.989 | | T | **1.000** | 0.000 | **1.000** |
| | TS | **0.994** | 0.003 | **0.997** | | TS | 0.982 | 0.005 | **0.992** | | TS | **1.000** | 0.000 | **1.000** |
| | C | 0.975 | 0.000 | 0.975 | | C | 0.945 | 0.000 | 0.945 | | C | **1.000** | 0.000 | **1.000** |
| | D | 0.978 | 0.000 | 0.978 | | D | 0.951 | 0.000 | 0.951 | | D | **1.000** | 0.000 | **1.000** |
| | F | 0.972 | 0.000 | 0.972 | | F | 0.948 | 0.000 | 0.948 | | F | **1.000** | 0.000 | **1.000** |
| `flex v4` | N | **0.999** | 0.000 | 0.999 | `grep v4` | N | 0.994 | 0.001 | 0.996 | `gzip v4` | N | **1.000** | 0.000 | **1.000** |
| | NS | **0.999** | 0.000 | 0.999 | | NS | 0.994 | 0.001 | 0.996 | | NS | **1.000** | 0.000 | **1.000** |
| | T | **0.999** | 0.000 | **1.000** | | T | 0.995 | 0.002 | **0.998** | | T | **1.000** | 0.000 | **1.000** |
| | TS | **0.999** | 0.000 | **1.000** | | TS | 0.996 | 0.002 | **0.998** | | TS | **1.000** | 0.000 | **1.000** |
| | C | 0.996 | 0.000 | 0.996 | | C | 0.996 | 0.000 | 0.996 | | C | **1.000** | 0.000 | **1.000** |
| | D | 0.997 | 0.000 | 0.997 | | D | **0.997** | 0.000 | 0.997 | | D | **1.000** | 0.000 | **1.000** |
| | F | **0.999** | 0.000 | 0.999 | | F | 0.996 | 0.000 | 0.996 | | F | 0.835 | 0.000 | 0.835 |
| `flex v5` | N | **1.000** | 0.000 | **1.000** | `grep v5` | N | **1.000** | 0.000 | 1.000 | `gzip v5` | N | 0.992 | 0.005 | 0.998 |
| | NS | **1.000** | 0.000 | **1.000** | | NS | **1.000** | 0.000 | **1.000** | | NS | 0.994 | 0.005 | **1.000** |
| | T | **1.000** | 0.000 | **1.000** | | T | **1.000** | 0.000 | **1.000** | | T | 0.992 | 0.008 | **1.000** |
| | TS | **1.000** | 0.000 | **1.000** | | TS | **1.000** | 0.000 | **1.000** | | TS | 0.997 | 0.004 | **1.000** |
| | C | 0.999 | 0.000 | 0.999 | | C | **1.000** | 0.000 | **1.000** | | C | **1.000** | 0.000 | **1.000** |
| | D | **1.000** | 0.000 | **1.000** | | D | **1.000** | 0.000 | **1.000** | | D | **1.000** | 0.000 | **1.000** |
| | F | **1.000** | 0.000 | **1.000** | | F | **1.000** | 0.000 | **1.000** | | F | **1.000** | 0.000 | **1.000** |

| | Alg. | $\mu$ | $\sigma$ | max | | Alg. | $\mu$ | $\sigma$ | max | | Alg. | $\mu$ | $\sigma$ | max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `make v2` | N | 0.993 | 0.002 | 0.998 | `sed v3` | N | 0.975 | 0.004 | 0.982 | `sed v7` | N | 0.859 | 0.018 | 0.895 |
| | NS | 0.994 | 0.003 | 0.998 | | NS | 0.974 | 0.004 | 0.981 | | NS | 0.856 | 0.023 | 0.935 |
| | T | 0.994 | 0.003 | **1.000** | | T | **0.980** | 0.004 | **0.988** | | T | 0.855 | 0.019 | 0.896 |
| | TS | 0.996 | 0.002 | **1.000** | | TS | 0.978 | 0.005 | 0.986 | | TS | 0.859 | 0.023 | 0.897 |
| | C | 0.989 | 0.000 | 0.989 | | C | 0.863 | 0.000 | 0.863 | | C | **0.957** | 0.000 | **0.957** |
| | D | 0.993 | 0.000 | 0.993 | | D | 0.866 | 0.000 | 0.866 | | D | **0.957** | 0.000 | **0.957** |
| | F | **1.000** | 0.000 | **1.000** | | F | 0.880 | 0.000 | 0.880 | | F | 0.909 | 0.000 | 0.909 |
| `make v3` | N | 0.996 | 0.002 | 0.999 | `sed v4` | N | 0.996 | 0.003 | **1.000** | `mysql v2` | N | 0.706 | 0.020 | 0.734 |
| | NS | 0.997 | 0.002 | 0.999 | | NS | 0.997 | 0.002 | **1.000** | | NS | 0.709 | 0.019 | 0.753 |
| | T | 0.997 | 0.002 | **1.000** | | T | 0.996 | 0.002 | 0.999 | | T | 0.719 | 0.036 | 0.780 |
| | TS | 0.997 | 0.003 | 0.999 | | TS | 0.996 | 0.002 | 0.999 | | TS | 0.714 | 0.029 | 0.769 |
| | C | 0.987 | 0.000 | 0.987 | | C | 0.996 | 0.000 | 0.996 | | C | 0.759 | 0.000 | 0.759 |
| | D | 0.984 | 0.000 | 0.984 | | D | 0.997 | 0.000 | 0.997 | | D | **0.787** | 0.000 | **0.787** |
| | F | **1.000** | 0.000 | **1.000** | | F | **0.999** | 0.000 | 0.999 | | F | 0.777 | 0.000 | 0.777 |
| `make v4` | N | 0.998 | 0.002 | **1.000** | `sed v5` | N | 0.965 | 0.006 | 0.979 | | | | | |
| | NS | 0.994 | 0.004 | 0.999 | | NS | 0.963 | 0.005 | 0.973 | | | | | |
| | T | 0.999 | 0.002 | **1.000** | | T | 0.974 | 0.006 | 0.986 | | | | | |
| | TS | 0.995 | 0.006 | **1.000** | | TS | 0.974 | 0.006 | 0.986 | | | | | |
| | C | 0.092 | 0.000 | 0.092 | | C | **0.999** | 0.000 | **0.999** | | | | | |
| | D | 0.092 | 0.000 | 0.092 | | D | 0.997 | 0.000 | 0.997 | | | | | |
| | F | **1.000** | 0.000 | **1.000** | | F | 0.972 | 0.000 | 0.972 | | | | | |
| `sed v2` | N | 0.890 | 0.026 | 0.946 | `sed v6` | N | 0.966 | 0.006 | 0.981 | | | | | |
| | NS | 0.889 | 0.021 | 0.927 | | NS | 0.970 | 0.006 | 0.983 | | | | | |
| | T | 0.886 | 0.045 | **0.974** | | T | 0.976 | 0.006 | 0.986 | | | | | |
| | TS | 0.883 | 0.044 | 0.958 | | TS | 0.978 | 0.004 | **0.988** | | | | | |
| | C | 0.860 | 0.000 | 0.860 | | C | 0.982 | 0.000 | 0.982 | | | | | |
| | D | 0.849 | 0.000 | 0.849 | | D | **0.990** | 0.000 | 0.990 | | | | | |
| | F | **0.900** | 0.000 | 0.900 | | F | 0.958 | 0.000 | 0.958 | | | | | |

The performance difference between MOEAs and the additional greedy algorithms is statistically significant with large effect sizes in all cases: both the Wilcoxon-signed rank test $p$-values and the adjusted $p_{\text{Bonf}}$) are significant at the $\alpha = 0.05$ significance level, confirming the alternative hypothesis. Note that MOEAs and the hybrids significantly outperform the additional greedy algorithms in all cases with EPSILON and IGD, while they exhibit significant improvements in only 19 (N, T, TS) or 20 (NS) out of 22 cases for the HV.

Between two MOEAs, TAEA showed significant improvements with large effect sizes over NSGA-II for 13 out of 22 subject versions in terms of EPSILON and HV. TAEA also significantly outperformed NSGA-II for 11 out of 22 subject versions in terms of the IGD. The hybrid variants tend to produce solutions that are either significantly better than or statistically equal to their original versions. The hybrid NSGA-II outperformed its original in about 50% of the cases with medium effect sizes, and showed equal quality in other cases. Similarly, while the hybrid TAEA outperformed its original in 7 subject versions, the effect sizes are mostly small. Finally, between two hybrids, the hybrid TAEA outperformed the hybrid NSGA-II in 12 and 13 out of 22 cases with medium effect sizes in terms of EPSILON and HV respectively, and in 10 out of 22 cases with small effect sizes in terms of IGD. While the results suggest that TAEA and its seeded hybrid can outperform other algorithms, further empirical study will be required to draw a more conclusive decision.

## 5.2 Effectiveness of Prioritisation

Table 5 reports the APFD$_c$ values achieved by all 7 algorithms for the 22 subject versions. The results of statistical hypothesis tests between all pairs of algorithms are summarized in the fourth major column of Table 4. The spread of the $\hat{A}_{12}$ effect sizes is illustrated in Figure 2(d).

MOEAs and their variants still tend to outperform the cost-cognisant additional greedy algorithms in general, but not as dominantly as with quality indicators. Based on $p$-value, NSGA-II and its hybrid significantly outperform the statement coverage based prioritisation in 13 out of 22 cases, with large effect sizes, whereas TAEA and its hybrid do so in 14 and 13 cases respectively, with large effect sizes. While these numbers become smaller against other single objective prioritisations, the $\hat{A}_{12}$ metric indicates mostly medium to large effect sizes in favour of MOEAs and hybrids.

Table 4 also allows us to investigate how suitable each of the individual single objectives are as a surrogate for fault detection, by comparing the cost effectiveness of the single objective additional greedy prioritisations. Both the $\Delta$ and fault coverage outperform the statement coverage, as can be seen in the 'D vs C' and 'F vs C' comparisons. Also note that $\Delta$-coverage can outperform MOEAs and the hybrids up to 10 out of 22 cases. This provides supporting evidence to the conjecture about $\Delta$-coverage as well as to the existing work that uses fault coverage.

Finally, Figure 3 and Table 6 present the overall comparison between algorithms. The boxplot shows all APFD$_c$ values across all 22 cases, per algorithm. This provides a high

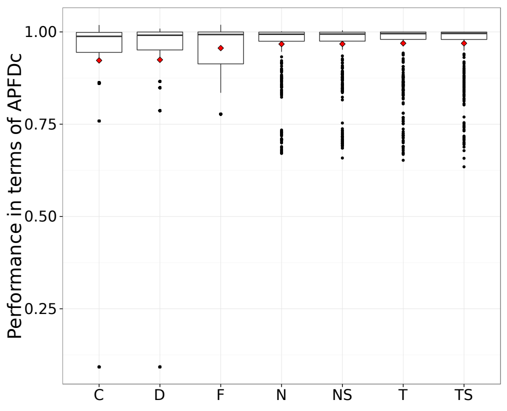


**Figure 3: Boxplots of the APFD$_c$ metric across all studied subjects. MOEAs and their variants show higher median values and smaller variances.**

**Table 6: Wilcoxon rank sum test of the APFD$_c$ measures between all algorithms, across all studied subjects. For the $p$ and $p_{\text{Bonf}}$-values in row $r$ and column $c$, the alternative hypothesis is that the algorithm in row $r$ produces higher APFD$_c$ value than the algorithm in column $c$.**

| | C | | D | | F | |
|---|---|---|---|---|---|---|
| | $p$ | $p_{\text{bonf}}$ | $p$ | $p_{\text{bonf}}$ | $p$ | $p_{\text{bonf}}$ |
| D | **<0.001** | **<0.001**$^\dagger$ | – | – | – | – |
| F | **0.001** | **0.022**$^\dagger$ | **0.009** | 0.192$^\dagger$ | – | – |
| N | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **0.002**$^\dagger$ | **0.003** | 0.061$^\dagger$ |
| NS | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **0.012**$^\dagger$ |
| T | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ |
| TS | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ |
| | N | | NS | | T | |
| | $p$ | $p_{\text{bonf}}$ | $p$ | $p_{\text{bonf}}$ | $p$ | $p_{\text{bonf}}$ |
| NS | 0.331 | 1.000 | – | – | – | – |
| T | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ | – | – |
| TS | **<0.001** | **<0.001**$^\dagger$ | **<0.001** | **<0.001**$^\dagger$ | 0.480 | 1.000 |

level comparison between algorithms. MOEAs and their variants show higher median values and smaller variances, suggesting that, for an arbitrary program and its test suite, they are more likely to result in higher APFD$_c$. Table 6 presents $p$ and $p_{\text{Bonf}}$-values from Wilcoxon signed rank test of APFD$_c$ values, aggregated across all 22 cases, between all pairs of algorithms. It confirms the observation from Figure 3 that MOEAs and their variants can outperform the single objective algorithms.

To answer **RQ2**, MOEAs are not as dominant in terms of the testing cost effectiveness measured in APFD$_c$ as in optimisation quality. However, MOEAs can still outperform statement coverage based prioritisation in up to 14 out of 22 cases. The $\hat{A}_{12}$ metric suggests that, if the decision maker chooses a solution on the reference Pareto front produced by an MOEA, there is a reasonable chance that the prioritisation will produce a higher APFD$_c$ value than that of the single objective prioritisation.

## 5.3 Efficiency

Let us now turn to **RQ3**. Table 7 shows the average sizes of coverage trace data across all versions of studied subject programs, both before ($S_O$) and after ($S_C$) the application of the coverage compaction algorithm. The reduction percentage of the coverage trace size, $R_{\%}$, is calculated as $100 \cdot (S_O - S_C)/S_O$. Also the ratio $S_O/S_C$ shows how many times smaller is the coverage trace size after compaction. In all subject programs, the coverage compaction achieves close to or over 90% of size reduction. In particular, `mysql` experiences 99.9% size reduction.

**Table 7: Coverage data sizes before ($S_O$) and after ($S_C$) compaction**

| | Coverage Trace Size | | | | | |
|---|---|---|---|---|---|---|
| | `flex` | `grep` | `gzip` | `make` | `sed` | `mysql` |
| $S_O$ | 3822.83 | 3338.00 | 1887.33 | 5901.60 | 3360.62 | 447316.67 |
| $S_C$ | 354.67 | 457.33 | 93.00 | 123.60 | 223.88 | 916.00 |
| $R_{\%}$ | **90.72** | **86.30** | **95.07** | **97.91** | **93.34** | **99.80** |
| $S_O/S_C$ | 10.78 | 7.30 | 20.29 | 47.75 | 15.01 | 488.34 |

The size reduction leads to reduction in execution time. The execution time of two MOEAs and three additional greedy algorithms have been measured for 30 times using the wall clock[2], using the same configuration described in Section 4.2. Note that, essentially, the execution time of a hybrid is roughly equal to the sum of its component, i.e. that of the corresponding MOEA and the additional greedy algorithm. Table 8 shows the mean ($\mu$) and the standard deviation ($\sigma$) of the average execution time that each algorithm takes, with and without compact coverage, along with the speed-up.

Being a constructive heuristic, it is expected that the additional greedy algorithms will be faster than MOEAs on average. Without coverage compaction, they take less than 60 seconds for all SIR subjects, and less than 2.3 hours for `mysql`. The longer execution time of $\Delta$-coverage based prioritisations can be attributed to the additional invocation of the `diff` tool to identify the modified coverage. On the other hand, both NSGA-II and TAEA take from slightly over a minute (`gzip`) to over 35 minutes (`make`) with SIR subjects. The execution time of MOEAs for `mysql` becomes pathological without compaction, as they take almost 9 days.

The application of the coverage compaction algorithm introduces a dramatic change to the performance of all algorithms. For the SIR subjects, the MOEAs terminate within 40 seconds, achieving speed-ups ranging from one to two orders of magnitude. However, the biggest speed-up is observed in the largest subject, `mysql`: their execution times are reduced from about 8.7 days to slightly over a minute, achieving 4 orders of magnitude speed-up (about 10,000 times faster). The additional greedy algorithms can also prioritise `mysql` test cases under 12 seconds after coverage compaction (a speed-up of 2 orders of magnitude). It is worth noting that the additional greedy algorithm based on $\Delta$-coverage for the `mysql` case exhibits the highest speed-up of this study, becoming about 16,000 times faster.

To answer **RQ3**, the execution time of all algorithms can be short enough to be used in most regression testing contexts, especially with the coverage compaction algorithm, which introduces speed-ups ranking from 1 to 4 orders of magnitudes. This may prove to be a pivotal contribution for future work. Without coverage compaction, a large real

[2] Note that the execution time of MOEAs for `mysql` could not be measured for 30 times, as each run took over a week. It was repeated only twice.

**Table 8: Average wall clock execution time in seconds without and with Compaction, along with Speed-up**

| | | No Compact. | | Compact. | | | | | No Compact. | | Compact. | | | | | No Compact. | | Compact. | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | Speed-up | | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | Speed-up | | Alg. | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | Speed-up |
| | N | 627.28 | 80.42 | 16.47 | 13.55 | 38.08 | | N | 79.84 | 9.09 | 8.07 | 14.93 | 9.90 | | N | 280.28 | 132.08 | 8.59 | 2.59 | 32.64 |
| `flex` | T | 628.52 | 81.66 | 19.38 | 19.22 | 32.44 | `gzip` | T | 81.24 | 10.01 | 6.14 | 17.59 | 13.23 | `sed` | T | 279.58 | 125.27 | 8.79 | 4.15 | 31.82 |
| | C | 7.21 | 0.39 | 0.56 | 0.10 | 12.96 | | C | 0.88 | 2.17 | 0.02 | 0.04 | 39.10 | | C | 3.87 | 1.78 | 0.13 | 0.05 | 29.27 |
| | D | 9.72 | 1.06 | **0.07** | 0.08 | 135.76 | | D | 0.88 | 1.26 | **0.01** | 0.04 | 61.15 | | D | 4.79 | 2.15 | 0.07 | 0.05 | 71.46 |
| | F | **0.08** | 0.03 | 0.08 | 0.13 | 0.90 | | F | **0.02** | 0.03 | 0.02 | 0.04 | 0.84 | | F | **0.04** | 0.03 | **0.03** | 0.03 | 1.71 |
| | N | 926.39 | 145.63 | 34.48 | 13.75 | 26.87 | | N | 2117.28 | 222.36 | 18.50 | 21.62 | 114.46 | | N | 758170.25 | 562.46 | 73.62 | 4.41 | 10297.90 |
| `grep` | T | 905.82 | 157.09 | 38.90 | 35.80 | 23.28 | `make` | T | 2100.43 | 220.70 | 22.29 | 28.97 | 94.23 | `mysql` | T | 756153.49 | 982.51 | 72.25 | 5.53 | 10465.36 |
| | C | 15.11 | 1.10 | 1.50 | 0.14 | 10.05 | | C | 58.81 | 16.01 | 1.36 | 2.13 | 43.26 | | C | 6974.31 | 573.29 | 11.85 | 0.55 | 588.60 |
| | D | 20.36 | 5.01 | 0.35 | 0.21 | 59.06 | | D | 59.60 | 16.31 | 0.59 | 0.51 | 101.93 | | D | 8091.21 | 350.92 | **0.49** | 0.21 | 16442.09 |
| | F | **0.21** | 0.10 | **0.13** | 0.09 | 1.68 | | F | **0.33** | 0.08 | **0.49** | 1.72 | 0.67 | | F | **15.01** | 1.21 | 0.59 | 0.25 | 25.40 |

world systems such as `mysql` will not be able to take advantage of even the single objective prioritisation, under time-limited testing scenarios such as smoke testing, let alone more expensive MOEAs.

## 6. THREATS TO VALIDITY

The main threat to internal validity derives from potential instrumentation inaccuracy. To alleviate this, widely used and tested open source tools, such as **`gcov`**, **`diff`**, and **`valgrind`**, have been adopted. To avoid any bias, the majority of the subject programs were chosen from a well-managed software repository [10], where not only the subject programs and their test suites, but also the fault seeding process is documented. With `mysql`, a mature source code branch (5.5.X) that is both well-documented and tested was chosen; all artifacts and fault information used are available from the `mysql` website and bug tracker system.

Another potential threat to internal validity lies on the selection of the optimisation algorithms. Both MOEAs were chosen because they have been successfully applied to software engineering problems [15, 42]. However, only further studies with different algorithms can eliminate this threat.

Threats to external validity are centred around the representativeness of the studied subjects, and how it affects the generalisation of the claims. The use of the standard benchmark in the literature as well as a large and complex real world system will help avoid over-fitting the results. However, wider generalisation will require further empirical studies with a disjoint set of subject programs.

## 7. RELATED WORK

Recent trends in the regression literature suggest that test case prioritisation is receiving increasing attention [39]: prioritisation does not exclude the execution of any test case completely (it will eventually accomplish the "retest-all" strategy), yet promises the maximum benefit when testing may have to terminate prematurely. Structural coverage has been widely used as the prioritisation criterion [11,13,21,44]. Coverage of differences between two versions of program has been considered as an isolated single objective for prioritisation [12], but this is the first paper to consider it as part of a multi objective formulation and compare the results to those from single objective formulations. Evolutionary algorithms have been applied to test case prioritisation [25], but only with single objective formation, apart from Li et al. [24], which is concerned with parallelisation speed-ups and not fault detection and optimization quality.

Other prioritisation criteria in the literature include time of last test execution [22], coverage augmented by human knowledge [40], and interaction coverage [7]. Any of these objectives can be added to the multi objective test case prioritisation, because the MOEAs studied in this paper are agnostic to the objective functions. This paper chooses to focus on the most widely studied fault detection surrogate, structural coverage, and its variations in the regression testing context, $\Delta$- and past fault coverage.

The size of coverage data from large systems can significantly increase the execution time of evolutionary algorithms, limiting their scalability. Generic Purpose computation on Graphics Processing Units (GPGPU) has been suggested to parallelise and, therefore, improve the scalability [24, 42]. This paper proposes compact coverage, which provides 2 to 4 orders of magnitude speed-up without any parallelism. Unlike existing work [6], compact coverage is deterministic and completely lossless. Moreover, it is compatible with the existing GPGPU parallelism, providing further scalability.

## 8. CONCLUSIONS

This paper empirically evaluates seven different test case prioritisation algorithms: three instantiations of the additional greedy algorithm with different fault detection surrogates, two multi objective formulations using MOEAs (NSGA-II and TAEA), and two hybrid algorithms that augment the MOEAs with the additional greedy seeding. These algorithms are evaluated on a set of five utility programs from the Software Infrastructure Repository (SIR), together with a larger program, `mysql`, from which 20 real faults with "closed" status have been extracted. The results show that MOEAs and hybrid algorithms can produce solutions whose prioritisation effectiveness, measured by the widely studied APFD$_c$ metric, is either equal or superior to those of solutions produced by the additional greedy algorithms.

The paper also introduces a coverage compaction algorithm that dramatically, yet losslessly, reduces coverage data size, and thereby algorithm execution time. On the largest program, `mysql`, the additional greedy algorithms can take more than two hours to prioritise without compaction, but only 12 seconds after compaction. The performance improvement is even more dramatic for the MOEAs. Their performance is improved from over eight days to a little over one minute. Since compaction can be applied to any and all regression testing approaches, these performance improvements may make an important contribution to the practical application of regression test optimisation in future work.

## 9. ACKNOWLEDGEMENTS

The authors are supported by EPSRC, EP/J017515/1 (DAASE: Dynamic Adaptive Automated Software Engineering).

## 10. REFERENCES

[1] CLOC: count lines of code. `http://cloc.sourceforge.net`.

[2] MySQL, Online Bug Repository. `http://bugs.mysql.com/`.

[3] SIR: Software-artifact Infrastructure Repository. `http://sir.unl.edu/`.

[4] The MySQL Test Framework, Version 2.0. `http://bugs.mysql.com/`.

[5] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1–10, New York, NY, USA, 2011. ACM.

[6] R. Assi and W. Masri. Lossless reduction of execution profiles using a genetic algorithm. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2014, pages 294–297, March 2014.

[7] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology*, 48(10):960–970, 2006.

[8] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *Proceedings of* $2^{nd}$ *International Symposium on Search based Software Engineering (SSBSE 2010)*, pages 143–152, Benevento, Italy, 2010. IEEE Computer Society Press.

[9] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature Conference*, pages 849–858. Springer, 2000.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[11] H. Do, G. Rothermel, and A. Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11(1):33–70, 2006.

[12] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb 2002.

[13] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 102–112, August 2000.

[14] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, pages 329–338. ACM Press, May 2001.

[15] A. Finkelstein, M. Harman, S. A. Mansouri, J. Ren, and Y. Zhang. "fairness analysis" in requirements assignments. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE '08)*, Barcelona, Catalunya, Spain, September 2008.

[16] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[17] D. E. Goldberg and R. Lingle, Jr. Alleles loci and the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 154–159, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[18] R. Grissom and J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates, Inc., Publishers., 2005.

[19] Q. Gu, B. Tang, and D. Chen. Optimal regression testing based on selective coverage of test requirements. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA 10)*, pages 419 – 426, Sept. 2010.

[20] M. Harman. Making the case for MORTO: Multi Objective Regression Test Optimization. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 111–114, Washington, DC, USA, 2011. IEEE Computer Society.

[21] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance 2005 (ICSM'05)*, pages 549–558. IEEE Computer Society Press, September 2005.

[22] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, pages 119–129. ACM Press, May 2002.

[23] J. Knowles, L. Thiele, and E. Zitzler. A tutorial on the performance assessment of stochastic multiobjective optimizers. 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland, Feb. 2006. revised version.

[24] Z. Li, Y. Bian, R. Zhao, and J. Cheng. A fine-grained parallel multi-objective test case prioritization on gpu. In G. Ruhe and Y. Zhang, editors, *Search Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2013.

[25] Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[26] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska-Lincoln, March 2006.

[27] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 89–100. ACM Press, June 2007.

[28] Oracle Corporation. `http://www.mysql.com`.

[29] K. Praditwong and X. Yao. A new multi-objective evolutionary optimisation algorithm: The two-archive

algorithm. In *Proceedings of Computational Intelligence and Security, International Conference*, volume 4456 of *Lecture Notes in Computer Science*, pages 95–104, November 2006.

[30] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 130–140. ACM Press, May 2002.

[31] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of International Conference on Software Maintenance (ICSM 1999)*, pages 179–188. IEEE Computer Society Press, August 1999.

[32] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[33] J. P. Royston. An extension of shapiro and wilk's w test for normality to large samples. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):115–124, 1982.

[34] P. G. Sapna and M. Hrushikesha. Automated test scenario selection based on levenshtein distance. In T. Janowski and H. Mohanty, editors, $6^{th}$ *Distributed Computing and Internet Technology (ICDCIT'10)*, volume 5966 of *Lecture Notes in Computer Science (LNCS)*, pages 255–266. Springer-Verlag (New York), Bhubaneswar, India, Feb. 2010.

[35] A. Vargha and H. D. Delaney. A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):pp. 101–132, 2000.

[36] D. A. V. Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithm research: A history and analysis. Technical Report TR-98-03, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 1998.

[37] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 140–150. ACM Press, July 2007.

[38] S. Yoo and M. Harman. Using hybrid algorithm for pareto effcient multi-objective test suite minimisation. *Journal of Systems Software*, 83(4):689–701, April 2010.

[39] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 22(2):67–120, March 2012.

[40] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 201–211. ACM Press, July 2009.

[41] S. Yoo, M. Harman, and S. Ur. Highly scalable multi-objective test suite minimisation using graphics card. In *LNCS: Proceedings of the 3rd International Symposium on Search-Based Software Engineering*, volume 6956 of *SSBSE*, pages 219–236, September 2011.

[42] S. Yoo, M. Harman, and S. Ur. Gpgpu test suite minimisation: search based software engineering performance improvement using graphics cards. *Empirical Software Engineering*, 18(3):550–593, 2013.

[43] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In $8^{th}$ *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, Szeged, Hungary, September 5th - 9th 2011. Industry Track.

[44] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei. Time-aware test-case prioritization using Integer Linear Programming. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA 2009)*, pages 212–222. ACM Press, July 2009.

[45] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, Nov. 1999.