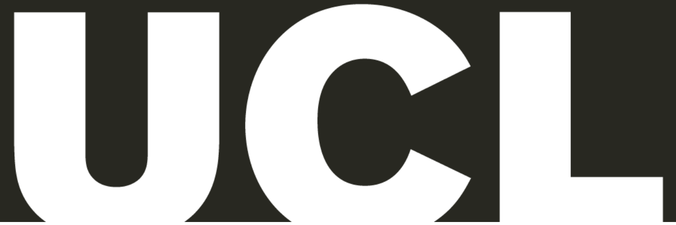
UCL DEPARTMENT OF
COMPUTER SCIENCE



Research Note

RN/14/01

Pareto Efficient Multi-Objective Regression Test Suite Prioritisation

April 23, 2014

Michael G. Epitropakis[1], *Shin Yoo*[2], *Mark Harman*[2], *Edmund K. Burke*[1]

Affiliation: University of Stirling[1], University College London[2]
E-Mail: mge@cs.stir.ac.uk, shin.yoo@ucl.ac.uk, mark.harman@ucl.ac.uk, e.k.burke@stir.ac.uk

Abstract

Test suite prioritisation seeks a test case ordering 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. We study multi objective test suite prioritisation, evaluating it on multiple versions of five widely-used benchmark programs and a much larger real world system of over 1MLoC. Our multi objective algorithms find faults significantly faster and with large effect size for 20 of the 22 versions. We also introduce a non-lossy coverage compact 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.

1 Introduction

Test case prioritisation seeks a test case ordering that favours early fault revelation [8]. Prioritisation 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 [32].

Of course, it is not known which tests will reveal which faults at test case prioritisation time, so some surrogate has to be employed (as in other regression test optimisation approaches). Often structural coverage is adopted as this surrogate [7, 20, 24, 37]. 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 and 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 [3, 14, 15, 19, 28, 31]. However, all but one of the previous studies of multi objective regression test optimisation have been concerned with test case selection rather than test case prioritisation. The only previous study to use a multi objective approach for prioritisation [19] was primarily concerned with extending, to prioritisation, previous work on parallel computation speed-ups for test case selection [34]. It therefore reports the reduction of optimisation time achieved by parallelisation, but does not study multi objective test case prioritisation in its own right.

This paper studies multi objective test case prioritisation in detail. We consider three objectives: average percentage of coverage achieved, average percentage of coverage of changed code, and average percentage of past fault coverage. The first objective is the previously widely-used surrogate for fault detection capability. The second objective is included because we conjecture that prioritising differences between versions is also an natural objective. Though considered in isolation, the trade-off between coverage of the entire program and coverage of the differences has not previous been studied [32]. Finally, the third objective is included because previous work on test case selection [36] has noted that tests that have detected faults in the past may tend to be more effective than those that have not.

We evaluated multi objective test case prioritisation on five SIR [5] programs (`flex`, `grep`, `gzip`, `make` and `sed`), which contain seeded faults (as done in much of the previous literature [32]). Additionally, we included a large real world system, `mysql`, which has not previously been studied in the literature and for which we extracted information concerning its real faults. Multiple versions of the SIR programs are available so, in total, we evaluated on 22 different versions.

We implemented two different multi objective test case prioritisation algorithms, one using the well-known NSGA-II algorithm [4] and the other using the Two Archive multi objective algorithm [23] in addition to the current state-of-the-art single-objective additional greedy algorithm. We used three standard solution quality indicator metrics to compare the solutions found by the three algorithms. The results of our study show that the multi objective test case prioritisation algorithms significantly outperform the state-of-the-art with large effect sizes according to each of three quality metrics for 21 of the 22 versions studied.

Perhaps more importantly for practising testers, we also found that for 20 of the 22 versions studied, both multi objective algorithms also outperformed the state-of-the-art in terms of test effectiveness; they find faults significantly faster (and with large effect size in all 20 cases), according to the standard evaluation metric, cost cognisant Average Percentage of Fault Detected (APFD$_c$) [8].

We also introduce and evaluate *coverage compaction*, an algorithm for non-lossy coverage data com-

paction. Coverage compaction algorithm is applicable to all regression testing problems as a pre-processing phase. Our results indicate that it can dramatically improve 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`, the compact size of the coverage data led to four orders of magnitude speed-up for multi objective evolutionary algorithms, and two orders of magnitude speed-up for the additional greedy algorithm.

The technical contributions of this paper are as follows:

- We evaluate two multi objective evolutionary algorithms, as well as the state-of-the-art additional greedy algorithm, 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`.
- We also evaluate all three 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) metric to evaluate the rate of fault detection.
- We introduce the coverage compaction algorithm, which can achieve up to 4 orders of magnitude speed-up for MOEAs 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 [26]:

Definition 2.1. Test Case Prioritisation Problem: Given a test suite T, a set of orderings (permutations), Π, 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 Π represents the set of all possible orderings (permutations) of T, and the fitness (or 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. As a result, f is usually measured using some surrogate for fault detection capability, such as structural coverage.

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 that we consider M different objectives (award functions), $f_i : \Pi \rightarrow \mathbb{R}, (1 \leq i \leq M)$. An ordering π_1 is said to dominate another ordering π_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 at the end. When applied to multi objective test case prioritisation, however, they will produce a set of orderings that are not dominated by any other orderings in the population. This set of solutions are said to form 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 Average Percentage of Fault Detection (APFD) evaluation metric for test case prioritisation [25], which, intuitively, measures how quickly faults are detected by the given ordering.

Definition 2.2. 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 π of T that reveals fault i. Given an ordering π of the test suite T, the Average Percentage of Fault Detection (APFD) metric is defined as following:

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

Later, Elbaum et al. extended the metric to consider both fault severity and test case execution cost, and defined the cost cognisant version, $APFD_c$ [9]. 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 2.3. 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:

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

Due to the lack of robust fault severity models for the subject programs, we only 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), we cannot evaluate MOEAs based on a single APFD$_c$ value obtained from a single ordering. Instead, we calculate the average APFD$_c$ value from all solutions a MOEA contributes to the reference Pareto front (please refer to Section 4.4 for the definition of a reference Pareto front).

2.4 Objectives

In the current study, we consider 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, we have used *statement coverage*, the *difference of the statement coverage* between two subsequent versions (hereby referred to as Δ-*coverage*), and the *historical fault information* of the test suite. For all three objectives, we measure the rate of their realisation, in a similar way to APFD$_c$. To be cost cognisant, we have also measured the execution cost of each test cases. We 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 [32], 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, we use statement coverage 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 expect that new regression faults are likely to originate from the changed parts of the source code in the version being tested. Therefore, the coverage of the changed parts only is a rational candidate for prioritisation criterion.

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 formulation of regression testing [31, 36] with successful results.

Execution Cost: one natural candidate for execution cost, the wall-clock time, is not only noisy but also dependant on the underlying hardware environment and operating system, making it inaccurate and not robust. To alleviate these issues, we use a widely used software profiling tool called `valgrind` [21]. `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. We profile the execution of each test case with `Valgrind` and use the number of virtual instructions as a representative surrogate for execution cost.

Based on these measures, we form three objective functions for test case prioritisation 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 2.3 with TS_i, TD_i, and TH_i. They represent the index of the test case that covers the ith statement in the program, ith *changed* statement in the program, and ith historical fault, respectively. As with the use of APFD$_c$, we only use the execution cost weighting with `valgrind` measurement and discard the fault severity weights. These three rate of realisation objective metrics will be hereby collectively referred to as AP*C$_c$.

2.5 Algorithms

We use 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) [4] is one of the most widely studied multi objective optimisation algorithms and has been applied to various domain ranging from Requirement Engineering [10] to regression testing [35]. The Two Archive Evolutionary Algorithm (TAEA) [23] was specifically designed to overcome weaknesses of NSGA-II and has also been applied to regression testing [35].

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 algorithm is characterised by two separate archives that are used in addition to the population pool. The first archive is reserved for fast convergence, while the second is maintained to promote 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.

We also implemented an additional greedy prioritisation algorithm [7]. This algorithm greedily selects test cases, maximising coverage of statements that remain as yet uncovered by test cases that occur earlier in the ordering. Specifically, the algorithm initially selects a test case that has the highest statement coverage. Subsequently, it tries to find the next test case that will cover the largest number of statements that are yet to be covered. This procedure is repeated until the highest possible coverage is reached, at which point the algorithm is recursively applied to the remaining test cases until all test cases are ordered.

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 [35]. 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 $O(lmnp)$.

We propose 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 traces data from structured programs often show highly repetitive patterns, because often some statements are executed only by the same subset of test cases, regardless of their location.

2.6.1 Coverage Compaction Algorithm

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) $\quad cc \leftarrow M_{(,i)}$
(4) $\quad$ **for** $j = i + 1$ **to** $M.width - 1$
(5) $\quad\quad$ **if** $cc == M_{(,j)}$
(6) $\quad\quad\quad M_{(,i)} \leftarrow M_{(,j)} + cc$
(7) $\quad\quad\quad$ delete $M_{(,j)}$ from M
(8) $\quad i \leftarrow i + 1$
(9) **return** M

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.

The coverage achieved by any subset of rows (i.e. test cases) can be calculated from the compact matrix. Suppose we want to calculate the coverage of the subset of the second and the third test case in Figure 1. With binary coverage matrix, the aggregation of rows is performed by bitwise OR; with the compact matrix, the aggregation is performed by addition, but we only add the value of a column from a newly added row when the corresponding column in the accumulated coverage row is still 0. For example, Aggregating

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

Figure 1: Example of Coverage Compaction

the 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: $\frac{5}{8} \cdot 100 = 62.5\%$.

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). In this section, we 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 ith column of the compacted coverage matrix, and $c_i (1 \leq i \leq m')$ is the non-zero entry in ith 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. APDC$_c$ and, if the number of historical faults is large enough, APFC$_c$ too, can be calculated in a similar way.

3 Empirical Study

3.1 Research Questions

The empirical study compares the performance of the two multi objective evolutionary algorithms (NSGA-II and TAEA), as well as the additional greedy algorithm, for the multi objective test case prioritisation problem. The first research question, **RQ1**, concerns the quality of multi objective optimisation.

- **RQ1. Optimisation Quality:** Do MOEAs produce better quality solutions when compared to the additional greedy algorithm? If so, how much better?

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. We therefore also ask **RQ2**:

- **RQ2. Testing Effectiveness:** How effective are the prioritized test suites in terms of early fault detection?

RQ2 is answered by calculating and comparing the standard effectiveness measure for test case prioritisation, the APFD$_c$ metric. Our 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 three algorithms, with and without coverage compaction.

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

RQ3 is answered by comparing the average wall-clock execution time from repeated runs of all three algorithms, both with and without coverage compaction.

4 Experimental Setup

4.1 Subjects

We evaluate multi objective test case prioritisation using six `C/C++` open source programs as subjects. Five subject programs are Unix utilities taken from the widely-used Software-artefact Infrastructure Repository (SIR) [5]: `flex`, `grep`, `gzip`, `make`, and `sed`. The sixth subject program is `mysql`, one of the most popular open source relational database management systems [22].

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 studied subject programs in Lines of Code (LOC), measured with the `cloc` tool[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. we cannot prioritise `V1`s). In addition, `mysql-v0` was used as the baseline to collect the regression faults in `mysql-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 (a full description of the fault seeding is available from the SIR[2]).

The source code of `mysql` comes with a testing framework that provides a set of test cases and the infrastructure required for their execution. We use the test suite for `V0` as the regression test suite for all

[1] `http://cloc.sourceforge.net`
[2] `http://sir.unl.edu/`

subsequent versions[3]. We manually collected 20 real faults from the online bug-tracking system used by `mysql` community[4]. 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. We use widely used genetic operators for permutation type representation: Partially Matched Crossover (PMX) and swap mutation, as well as binary tournament selection [11,12]. 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, both MOEAs are executed 30 times.

4.3 Measurements & Environment

Statement coverage data is obtained using the GNU `gcov` profiling tool. The Δ-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 algorithm nor MOEAs due to insufficient amount of memory; as a result, we measured the execution time of all algorithms 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, we use a *reference* Pareto front as a surrogate. A reference Pareto front consists of the best non-dominated solutions found by all evaluated algorithms. Formally, the reference Pareto front, P_{ref}, can be defined as follows:

Definition 4.1. Reference Pareto Front: Let us assume that we have N different Pareto fronts, $P_i (i = 1, 2, \ldots, N)$, and the union of all P_i, P_{U}. The reference Pareto front, P_{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 **RQ 1**, we use three widely studied MOEA quality indicators: 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:

[3] A detailed documentation of the `mysql` Test Framework and its components can be found in: `http://dev.mysql.com/doc/mysqltest/2.0/en/`

[4] `http://bugs.mysql.com/`

- **EPSILON** measures the shortest distance that is required to transform every solution in A so that it dominates the reference Pareto front [18].
- **IGD** is the average distance from solutions in the reference Pareto front to the closest solution in A [30].
- **HV** is the volume of objective space dominated by solutions in A [38].

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

To assess the statistical significance of differences in performance, measured by the quality indicators, we use non-parametric Wilcoxon-signed rank test [1,13] (the Shapiro-Wilk normality test [27] does not report 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 one algorithm produces either lower (or higher, in case of HV) median quality indicator values. We use 0.05 significance level. In addition, to address the problem of the higher probability of Type I errors in multiple comparisons, we have applied the standard Bonferroni adjustment [1] and report the adjusted p-value, p_{Bonf}. This is conservative, but safer in the sense that it avoids Type I errors.

Furthermore, to assess the magnitude of an algorithm's performance improvement, we use a non-parametric effect size measure, called Vargha and Delaney's $\hat{A}_{12}$ statistic [29]. 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 [29], 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.

Table 3: Descriptive statistics and statistical significance analysis on the performance of the optimization algorithms in terms of the EPSILON, HV and IGD indicators. For each subject version, we present mean (μ) and standart deviation (σ) of the quality indicators for the NSGA-II (N), the TAEA (T) and the additional greedy (G) algorithm. We statistically compare each pair of algorithms on each quality indicator and report p-values for the Wilcoxon test (p-value) and the Bonferroni correction (p_{Bonf}), as well as the $\hat{A}_{12}$ effect size. NSGA-II and TAEA outperform the additional greedy algorithm in the majority of the subject versions.

		EPSILON		HV		IGD			EPSILON			HV			IGD		
	Alg.	μ	σ	μ	σ	μ	σ	Alg.	p-value	p_{Bonf}	$\hat{A}_{12}$	p-value	p_{Bonf}	$\hat{A}_{12}$	p-value	p_{Bonf}	$\hat{A}_{12}$
flex-v2	N	0.005	0.001	0.768	0.051	0.006	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.004**	0.002	**0.857**	0.049	**0.005**	0.001	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.225	0.000	0.000	0.000	3.290	0.000	T vs N	**0.009**	**0.027**	0.29	**<0.001**	**<0.001**	0.12	**0.003**	**0.010**	0.27
flex-v3	N	0.003	0.001	0.574	0.057	**0.019**	0.004	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.001**	0.001	**0.671**	0.090	0.020	0.006	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.191	0.000	0.000	0.000	9.509	0.000	T vs N	**<0.001**	**<0.001**	0.05	**<0.001**	**<0.001**	0.18	0.334	1.000	0.54
flex-v4	N	**0.003**	0.001	0.200	0.148	**0.019**	0.008	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.92**	**<0.001**	**<0.001**	**1.00**
	T	**0.003**	0.003	**0.224**	0.274	0.031	0.033	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.80**	**<0.001**	**<0.001**	**1.00**
	G	0.181	0.000	0.000	0.000	8.923	0.000	T vs N	0.128	0.384	0.39	0.846	1.000	0.55	0.324	0.971	0.48
flex-v5	N	**0.001**	0.001	**0.526**	0.158	**0.022**	0.006	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	0.003	0.002	0.246	0.243	0.040	0.021	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.85**	**<0.001**	**<0.001**	**1.00**
	G	0.189	0.000	0.000	0.000	46.182	0.000	T vs N	**<0.001**	**<0.001**	**0.85**	**<0.001**	**<0.001**	**0.82**	**<0.001**	**<0.001**	**0.84**
gzip-v2	N	**0.000**	0.000	**0.792**	0.033	**0.003**	0.003	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.000**	0.000	0.752	0.101	0.005	0.005	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.001	0.000	0.000	0.000	0.167	0.000	T vs N	**<0.001**	**<0.001**	0.08	**0.031**	0.092	0.63	0.118	0.354	0.60
gzip-v3	N	**0.000**	0.000	0.144	0.145	0.029	0.010	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.95**	**<0.001**	**<0.001**	**1.00**
	T	**0.000**	0.000	**0.602**	0.171	**0.010**	0.006	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**

	G	0.011	0.000	0.000	0.000	0.561	0.000	T vs N	**<0.001**	**<0.001**	0.03	**<0.001**	**<0.001**	0.05	**<0.001**	**<0.001**	0.04
`gzip-v4`	N	**0.000**	0.000	0.533	0.120	0.015	0.004	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.000**	0.000	**0.705**	0.174	**0.009**	0.009	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.98**	**<0.001**	**<0.001**	**1.00**
	G	0.032	0.000	0.000	0.000	2.665	0.000	T vs N	**<0.001**	**<0.001**	0.16	**<0.001**	**<0.001**	0.14	**<0.001**	**<0.001**	0.12
`gzip-v5`	N	**0.000**	0.000	**0.986**	0.009	**0.004**	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.000**	0.000	0.979	0.025	0.005	0.001	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.116	0.000	0.059	0.000	0.036	0.000	T vs N	0.773	1.000	0.35	0.258	0.774	0.56	**<0.001**	**<0.001**	**0.82**
`grep-v2`	N	0.004	0.001	0.702	0.051	**0.009**	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.003**	0.001	**0.773**	0.122	0.010	0.002	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.096	0.000	0.000	0.000	0.041	0.000	T vs N	**<0.001**	**<0.001**	0.05	**0.014**	**0.041**	0.24	**0.002**	**0.007**	**0.68**
`grep-v3`	N	0.005	0.001	0.590	0.038	**0.014**	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.004**	0.001	**0.698**	0.042	**0.014**	0.001	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.043	0.000	0.000	0.000	0.060	0.000	T vs N	**<0.001**	**<0.001**	0.04	**<0.001**	**<0.001**	0.02	0.934	1.000	0.51
`grep-v4`	N	0.005	0.001	0.714	0.033	**0.005**	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.004**	0.001	**0.778**	0.029	**0.005**	0.001	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.027	0.000	0.000	0.000	0.035	0.000	T vs N	**<0.001**	**<0.001**	0.10	**<0.001**	**<0.001**	0.06	0.696	1.000	0.51
`grep-v5`	N	**0.003**	0.001	0.015	0.035	0.026	0.005	N vs G	**<0.001**	**<0.001**	**1.00**	**0.004**	**0.012**	**0.68**	**<0.001**	**<0.001**	**1.00**
	T	**0.003**	0.001	**0.118**	0.158	**0.022**	0.010	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.85**	**<0.001**	**<0.001**	**1.00**
	G	0.007	0.000	0.000	0.000	2.489	0.000	T vs N	0.376	1.000	0.42	**<0.001**	**<0.001**	0.26	0.080	0.241	0.36
`make-v2`	N	**0.003**	0.001	0.727	0.042	**0.009**	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.90**	**<0.001**	**<0.001**	**1.00**
	T	**0.003**	0.001	**0.783**	0.057	0.010	0.002	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.97**	**<0.001**	**<0.001**	**1.00**
	G	0.015	0.000	0.672	0.000	0.022	0.000	T vs N	**<0.001**	**0.001**	0.22	**<0.001**	**<0.001**	0.21	0.113	0.340	0.58
`make-v3`	N	0.003	0.001	0.458	0.098	**0.037**	0.006	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.002**	0.001	**0.526**	0.235	0.040	0.017	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.93**
	G	0.016	0.000	0.000	0.000	0.073	0.000	T vs N	0.113	0.340	0.34	0.123	0.369	0.35	0.789	1.000	0.47
`make-v4`	N	**0.000**	0.000	**0.059**	0.125	0.359	0.220	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	0.00	**<0.001**	**<0.001**	0.00
	T	0.002	0.001	0.000	0.000	4.745	2.470	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	0.00	**<0.001**	**<0.001**	0.00
	G	0.008	0.000	0.000	0.000	**0.097**	0.000	T vs N	**<0.001**	**<0.001**	**0.99**	**<0.001**	**<0.001**	0.00	**<0.001**	**<0.001**	**0.99**
`sed-v2`	N	**0.002**	0.000	0.656	0.071	0.021	0.005	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.002**	0.001	**0.792**	0.068	**0.012**	0.005	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.028	0.000	0.000	0.000	0.093	0.000	T vs N	**0.002**	**0.006**	0.28	**<0.001**	**<0.001**	0.09	**<0.001**	**<0.001**	0.04
`sed-v3`	N	0.003	0.001	0.740	0.031	**0.006**	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.002**	0.001	**0.797**	0.035	**0.006**	0.001	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.143	0.000	0.000	0.000	0.137	0.000	T vs N	**0.004**	**0.012**	0.29	**<0.001**	**<0.001**	0.11	0.681	1.000	0.54
`sed-v4`	N	**0.001**	0.000	0.492	0.105	0.030	0.008	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.001**	0.001	**0.568**	0.265	**0.028**	0.022	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**0.97**	**<0.001**	**<0.001**	**1.00**
	G	0.007	0.000	0.000	0.000	0.151	0.000	T vs N	0.422	1.000	0.35	0.168	0.505	0.26	0.210	0.629	0.28
`sed-v5`	N	**0.008**	0.004	0.719	0.041	**0.009**	0.003	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.008**	0.004	**0.772**	0.042	**0.009**	0.003	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.062	0.000	0.000	0.000	0.034	0.000	T vs N	1.000	1.000	0.51	**<0.001**	**<0.001**	0.18	0.651	1.000	0.54
`sed-v6`	N	0.002	0.000	0.684	0.067	0.010	0.003	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.001**	0.000	**0.839**	0.063	**0.005**	0.002	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.018	0.000	0.000	0.000	0.285	0.000	T vs N	**<0.001**	**<0.001**	0.04	**<0.001**	**<0.001**	0.05	**<0.001**	**<0.001**	0.10
`sed-v7`	N	**0.002**	0.000	0.833	0.025	0.004	0.001	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.002**	0.000	**0.859**	0.031	**0.003**	0.001	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.036	0.000	0.000	0.000	0.046	0.000	T vs N	**<0.001**	**<0.001**	0.16	**0.005**	**0.015**	0.27	0.100	0.300	0.35
`mysql-v2`	N	0.007	0.003	0.424	0.079	**0.025**	0.005	N vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	T	**0.006**	0.004	**0.438**	0.145	0.026	0.008	T vs G	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**	**<0.001**	**<0.001**	**1.00**
	G	0.207	0.000	0.000	0.000	0.076	0.000	T vs N	0.821	1.000	0.46	0.294	0.883	0.39	0.472	1.000	0.55

5 Results & Discussion

5.1 Optimisation Quality

We first consider the quality of solutions produced by the multi objective and the additional greedy algorithm. Table 3 presents the complete quantitative results from the NSGA-II (denoted 'N'), the TAEA ('T') and the additional greedy algorithm ('G') applied to 22 versions of subject programs. The left side of Table 3 shows the quality indicators (EPSILON, IGD, and HV) applied to three algorithms; the right side presents the results of statistical hypothesis testing between three pairs of algorithms. To indicate the best performing cases, we highlight, in the left side, the cases where an algorithm exhibits better performance values with **boldface** font[5]. On the right side, we highlight the cases with statistically significant differences (for the $\alpha = 0.05$ level of significance) with **boldface** font. In addition, for the $\hat{A}_{12}$ effect size metric, we underline the medium effect size (there is only one such case) and highlight the large effect size with **boldface** font.

[5]Note that lower values indicate better performance for EPSILON and the IGD, while the opposite holds for the HV.

Table 4: APFD$_c$ values from all three algorithms: MOEAs outperform the additional greedy in 20 out of 22 versions with statistical significance.

	Alg.	μ	σ	Alg.	p-value	p_{Bonf}	$\hat{A}_{12}$		Alg.	μ	σ	Alg.	p-value	p_{Bonf}	$\hat{A}_{12}$
`flex-v2`	N	**0.999**	0.000	N vs G	**<0.001**	**<0.001**	**1.00**	`grep-v5`	N	**1.000**	0.000	N vs G	**<0.001**	**<0.001**	**1.00**
	T	**0.999**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**		T	**1.000**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**
	G	0.809	0.000	T vs N	**0.014**	**0.043**	0.32		G	0.999	0.000	T vs N	0.600	1.000	0.57
`flex-v3`	N	0.987	0.006	N vs G	**<0.001**	**<0.001**	**1.00**	`make-v2`	N	0.993	0.002	N vs G	**<0.001**	**<0.001**	**1.00**
	T	**0.993**	0.003	T vs G	**<0.001**	**<0.001**	**1.00**		T	**0.994**	0.003	T vs G	**<0.001**	**<0.001**	**1.00**
	G	0.818	0.000	T vs N	**<0.001**	**<0.001**	0.18		G	0.982	0.000	T vs N	**0.013**	**0.038**	0.35
`flex-v4`	N	**0.999**	0.000	N vs G	**<0.001**	**<0.001**	**1.00**	`make-v3`	N	0.996	0.002	N vs G	**<0.001**	**<0.001**	**0.97**
	T	**0.999**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**		T	**0.997**	0.002	T vs G	**<0.001**	**<0.001**	**1.00**
	G	0.810	0.000	T vs N	**0.028**	0.083	0.28		G	0.991	0.000	T vs N	**0.006**	**0.018**	0.28
`flex-v5`	N	**1.000**	0.000	N vs G	**<0.001**	**<0.001**	**1.00**	`make-v4`	N	0.998	0.002	N vs G	**<0.001**	**<0.001**	**1.00**
	T	**1.000**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**		T	**0.999**	0.002	T vs G	**<0.001**	**<0.001**	**1.00**
	G	0.856	0.000	T vs N	**0.015**	**0.046**	**0.70**		G	0.989	0.000	T vs N	**0.049**	0.147	0.26
`gzip-v2`	N	**1.000**	0.001	N vs G	**<0.001**	**<0.001**	**0.97**	`sed-v2`	N	**0.890**	0.026	N vs G	**<0.001**	**<0.001**	**0.93**
	T	**1.000**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**		T	0.886	0.045	T vs G	**0.002**	**0.005**	**0.77**
	G	0.999	0.000	T vs N	0.118	0.354	0.36		G	0.854	0.000	T vs N	0.484	1.000	0.54
`gzip-v3`	N	**1.000**	0.000	N vs G	**<0.001**	**<0.001**	**1.00**	`sed-v3`	N	0.975	0.004	N vs G	**0.002**	**0.007**	0.27
	T	**1.000**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**		T	**0.980**	0.004	T vs G	**0.005**	**0.014**	**0.73**
	G	**1.000**	0.000	T vs N	**0.008**	**0.024**	0.28		G	0.977	0.000	T vs N	**<0.001**	**<0.001**	0.18
`gzip-v4`	N	**1.000**	0.000	N vs G	**<0.001**	**<0.001**	**1.00**	`sed-v4`	N	**0.996**	0.004	N vs G	0.524	1.000	0.63
	T	**1.000**	0.000	T vs G	**<0.001**	**<0.001**	**1.00**		T	**0.996**	0.002	T vs G	0.217	0.651	0.67
	G	0.999	0.000	T vs N	**<0.001**	**<0.001**	0.13		G	0.988	0.046	T vs N	0.434	1.000	0.64
`gzip-v5`	N	**0.992**	0.005	N vs G	**<0.001**	**<0.001**	**1.00**	`sed-v5`	N	0.965	0.006	N vs G	**<0.001**	**<0.001**	0.01
	T	**0.992**	0.008	T vs G	**<0.001**	**<0.001**	**1.00**		T	0.974	0.006	T vs G	**<0.001**	**<0.001**	0.17
	G	0.857	0.000	T vs N	0.334	1.000	0.40		G	**0.981**	0.005	T vs N	**<0.001**	**<0.001**	0.16
`grep-v2`	N	0.980	0.004	N vs G	**<0.001**	**<0.001**	**1.00**	`sed-v6`	N	0.966	0.006	N vs G	**<0.001**	**<0.001**	0.07
	T	**0.986**	0.005	T vs G	**<0.001**	**<0.001**	**1.00**		T	**0.976**	0.006	T vs G	**0.013**	**0.038**	**0.73**
	G	0.932	0.000	T vs N	**<0.001**	**<0.001**	0.17		G	0.973	0.000	T vs N	**<0.001**	**<0.001**	0.11
`grep-v3`	N	**0.982**	0.004	N vs G	**<0.001**	**<0.001**	**1.00**	`sed-v7`	N	**0.859**	0.018	N vs G	**<0.001**	**<0.001**	**1.00**
	T	**0.982**	0.004	T vs G	**<0.001**	**<0.001**	**1.00**		T	0.855	0.019	T vs G	**<0.001**	**<0.001**	**1.00**
	G	0.968	0.000	T vs N	0.886	1.000	0.50		G	0.735	0.000	T vs N	0.742	1.000	0.54
`grep-v4`	N	0.994	0.001	N vs G	**<0.001**	**<0.001**	**0.83**	`mysql-v2`	N	0.706	0.020	N vs G	**<0.001**	**<0.001**	**1.00**
	T	**0.995**	0.002	T vs G	**<0.001**	**<0.001**	**0.87**		T	**0.719**	0.036	T vs G	**<0.001**	**<0.001**	**1.00**
	G	0.993	0.000	T vs N	**0.042**	0.125	0.35		G	0.534	0.000	T vs N	0.139	0.416	0.40

The results provide a positive answer to **RQ1**. Both NSGA-II and TAEA outperform the additional greedy algorithm, for all subject programs and versions, with their quality indicator metrics. The additional greedy algorithm is unable to produce as many solutions on or close to the reference front set as MOEAs: this is indicated by the high values of the EPSILON and IGD indicators. HV indicator values of the additional greedy algorithm are mostly nearly zero, which is expected because, being a single objective algorithm, it does not try to optimise for all three objectives.

The performance difference between the additional greedy and the MOEAs is statistically significant with large effect size in all but one case. that is, the observed p-values (both the Wilcoxon-signed rank test p-values and its adjusted p_{Bonf}) are significant at the $\alpha = 0.05$ significance level, confirming the alternative hypothesis. Note that NSGA-II and TAEA significantly outperform the additional greedy algorithm in all cases in EPSILON and IGD measures, while they exhibit significant improvement on only 21 out of 22 versions in the HV measure.

In terms of p-values, TAEA shows statistically significant improvements over NSGA-II in 14 out of 22 subject versions. However the effect size in the majority of these cases is not substantial. Furthermore, in the largest subject `mysql-v2`, the difference between the two MOEAs is inconclusive; TAEA contributes slightly more to the diversity of the reference front (indicated by the higher HV measure). Further study is needed to make a conclusive comparison of TAEA and NSGA-II.

Figure 2 plots Pareto fronts for `make-v2` and `mysql-v2`. In both cases, the additional greedy algorithm produces a single solution that is on the reference Pareto front. However, the trade-offs between objectives identified by MOEAs suggests that there often exist other solutions that can significantly outperform the solution produced by the additional greedy algorithm.

5.2 Effectiveness of Prioritisation

We thus turn to **RQ2** to investigate the early fault detection capability. Table 4 reports the APFD$_c$ metric values achieved by all three algorithms for 22 subject versions. It also presents the statistical hypothesis testing between three pairs of algorithms.

We observe that, in the majority of the subject versions, MOEAs outperform the additional greedy algorithm according to APFD$_c$. That is, in 20 out of 22 versions, both NSGA-II and TAEA exhibit statistically significant improvement in APFD$_c$ over the additional greedy algorithm, with large effect sizes according to the $\hat{A}_{12}$ statistic. TAEA shows the best average performance in 19 out of the 22 versions, while NSGA-II does so in 12 versions. The additional greedy algorithm only performs as well as MOEAs in two cases: `gzip-v2` and `gzip-v3`. In `sed-v2`, NSGA-II

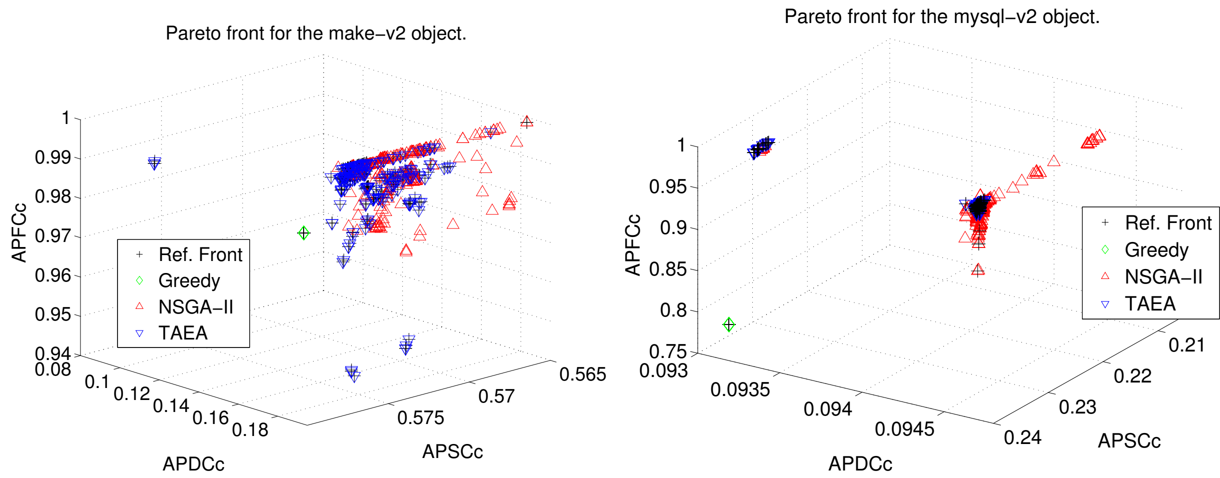


Figure 2: 3D Plots of Pareto fronts for `make-v2` and `mysql-v2`

outperforms the additional greedy algorithm with statistic significance, but the effect size is small. Between NSGA-II and TAEA, TAEA produces better mean performance, with statistically significant improvements in 10 out of the 22 subject versions. However, note that the effect size is large only in one case (`flex-v5`). With `mysql-v2`, both NSGA-II and TAEA outperform the additional greedy algorithm with statistical significance and large effect sizes. While TAEA exhibits the best average performance for `mysql-v2`, the difference to NSGA-II is not significant.

To answer **RQ2**, both MOEAs can outperform the additional greedy algorithm in terms of APFD_c: if the decision maker chooses a solution on the reference Pareto front produced by an MOEA, there is a high chance that the ordering will produce a higher APFD_c value than that of the additional greedy algorithm.

5.3 *Efficiency*

Finally, we consider the efficiency of the studied algorithms. To answer **RQ3**, we measure the size of coverage trace data, as well as the execution time of each algorithm. Table 5 shows the average size of coverage trace data across all versions of studied subject programs, both before (S_O) and after (S_C) the application of our 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 5: Average coverage trace size per subject program 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. We have measured the wall clock execution time of all three algorithms for 30 times[6], using the same configuration described in Section 4.2. Table 6 shows the mean (μ) and the standard deviation (σ) of the average wall clock execution time that each algorithm takes, with and without compact coverage, along with the speed-up.

Being a single objective heuristic, it is expected that the additional greedy algorithm will be the fastest algorithm on average. Without coverage compaction, it takes less than 30 seconds for all SIR subjects, and less than 1.5 hours for `mysql`. 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 coverage compaction, as they take almost 9 days.

Fortunately, the application of the coverage compaction algorithm makes a dramatic change to the performance of all three 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 algorithm can also prioritise `mysql` test cases under 7 seconds after coverage compaction (a speed-up of 2 orders of magnitude).

It is clear that the application of the compaction algorithm can introduce critical efficiency improvement when employing meta-heuristic search algorithms and, in particular, MOEAs, to regression testing optimisation that involves structural coverage. This may prove to be a pivotal contribution for future work. Without coverage compaction, a large real world systems such as `mysql` will not be able to take advantage of MOEA-based

[6]Note that the execution time of MOEAs for `mysql` could not be measured for 30 times, as each run took over a week. We could only repeat it twice.

Table 6: Average wall clock execution time in seconds without and with Compaction, along with Speed-up

	Alg.	No Compaction μ	No Compaction σ	Compaction μ	Compaction σ	Speed-up
flex	N	627.276	80.413	16.472	13.547	38.082
	T	628.520	81.659	19.375	19.222	32.440
	G	**4.538**	0.398	**0.452**	0.053	10.034
gzip	N	79.841	9.093	8.069	14.928	9.895
	T	81.243	10.012	6.143	17.593	13.226
	G	**0.343**	0.032	**0.015**	0.008	23.641
grep	N	926.386	145.629	34.479	13.754	26.868
	T	905.816	157.088	38.903	35.802	23.284
	G	**8.693**	0.641	**1.256**	2.021	6.921
make	N	2117.282	222.362	18.498	21.619	114.460
	T	2100.426	220.702	22.290	28.970	94.232
	G	**27.062**	1.781	**0.398**	0.206	68.064
sed	N	280.284	132.083	8.588	2.592	32.639
	T	279.579	125.271	8.785	4.150	31.824
	G	**4.781**	10.168	**0.126**	0.040	38.067
mysql	N	758170.247	562.460	73.624	4.411	10297.896
	T	756153.493	982.511	72.253	5.533	10465.357
	G	**5013.982**	66.392	**6.899**	0.386	726.790

regression testing optimisation. The reduced execution time also enables the use of MOEA-based techniques in time-limited testing scenarios, such as smoke testing, for which GPGPU based speed-up has been introduced [35].

6 Threats to Validity

The main threat to internal validity derives from potential instrumentation inaccuracy. To alleviate this, we employed widely used and tested open source tools, such as `gcov`, `diff`, and the `valgrind` profiling tool. To avoid any bias, we choose the majority of our subject programs from a well-managed software repository [5], where not only the subject programs and their test suites, but also the fault seeding process is documented. With `mysql`, we choose a mature source code branch (5.5.X) that available well-documented and tested; all artefacts 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. We choose two well-known MOEAs that have been successfully applied to software engineering problems [10, 35]. 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. We choose not only the standard benchmark in the literature but also a large and complex real world system to avoid over-fitting our results to a small set of programs. However, wider generalisation will require further empirical studies with a disjoint set of subject programs.

7 Related Work

Recent trend in the regression literature suggests that test case prioritisation is receiving increasing attention [32]: 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 [6, 8, 16]. Coverage of differences between two versions of program has been considered as an isolated single objective for prioritisation [7], but ours is the first paper to study the trade-off between coverage of differences and coverage of the entire system. Evolutionary algorithms have been applied to test case prioritisation [20], but only with single objective formation, apart from Li et al. [19], which is concerned with parallelisation speed-ups and not fault detection and solution quality.

Other prioritisation criteria in the literature include time of last test execution [17], coverage augmented by human knowledge [33], and interaction coverage [2]. The multi objective approach used in this paper is agnostic to the objective functions, and can be applied to any combination of these objective in addition to structural coverage.

The size of coverage trace data from large systems can significantly increase the execution time of evolutionary algorithms, limiting the scalability of the technique. Generic Purpose computation on Graphics Processing Units (GPGPU) has been suggested to parallelise and, therefore, improve the scalability [19, 35]. This paper proposes compact coverage, which provides 2 to 5 orders of magnitude speed-up without any parallelism. Moreover, the compact coverage is compatible with the existing GPGPU parallelism, so could provide further scalability over-and-above that available through parallelisation.

8 Conclusions

In this paper we have introduced and empirically evaluated two multi objective test suite prioritisation techniques, based on the multi objective evolutionary algorithms NSGA-II and TAEA. We evaluate them against the state-of-the-art on a set of five utility programs from the Software

Infrastructure Repository (SIR), together with a larger program, `mysql`, from which we extracted fault data for 20 of its faults (all of which have status "closed").

Even when the tester has only a single objective in mind, it is useful to use a multi objective formulation to improve early fault revelation. Since fault-revealing tests are unidentifiable at prioritisation time, the tester is forced to use a surrogate. We introduce a three-objective formulation of the problem, in which all three objectives are coverage-related surrogates. Our results demonstrate that this approach is highly effective; we find faults significantly faster than the state-of-the-art and with large effect size in 19 out of the 22 cases studied.

We also introduce coverage compaction algorithm which dramatically reduces coverage data size, and thereby algorithm execution time. On the larger program, `mysql`, the additional greedy algorithm takes 1.4 hours to prioritise without compaction, but only 7 seconds after compaction. The performance improvement is even more dramatic for the multi objective algorithms. 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, we believe that these performance improvements may make an important contribution to the practical application of repression test optimisation in future work.

Acknowledgements

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

References

[1] 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.

[2] 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.

[3] 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* 2nd *International Symposium on Search based Software Engineering (SSBSE 2010)*, pages 143–152, Benevento, Italy, 2010. IEEE Computer Society Press.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] P. G. Sapna and M. Hrushikesha. Automated test scenario selection based on levenshtein distance. In T. Janowski and H. Mohanty, editors, 6th *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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In 8th *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.

[37] 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.

[38] 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.