Using Hybrid Algorithm For Pareto Efficient Multi-Objective Test Suite Minimisation

Shin Yoo & Mark Harman

King's College London, Strand, London, WC2R 2LS, UK

Abstract

Test suite minimisation techniques seek to reduce the effort required for regression testing by selecting a subset of test suites. In previous work, the problem has been considered as a single-objective optimisation problem. However, real world regression testing can be a complex process in which multiple testing criteria and constraints are involved. This paper presents the concept of Pareto efficiency for the test suite minimisation problem. The Pareto efficient approach is inherently capable of dealing with multiple objectives, providing the decision maker with a group of solutions that are not dominated by each other. The paper illustrates the benefits of Pareto efficient multi-objective test suite minimisation with empirical studies of two and three objective formulations, in which multiple objectives such as coverage and past fault-detection history are considered. The paper utilises a hybrid, multi-objective genetic algorithm that combines the efficient approximation of the greedy approach with the capability of population based genetic algorithm to produce higher-quality Pareto fronts.

Key words: Regression testing, Test suite minimisation

1. Introduction

Regression testing is performed in order to guarantee that the recent changes in a program do not interfere with the functionality of the unchanged parts. The most straightforward approach to guarantee this is to execute all of the existing test cases to ensure that the new changes are harmless. However, this *retest-all* approach is often infeasible because, over time, the size of the test suite grows. Thus it may become prohibitively expensive to execute the entire test suite. Furthermore, as the development cycle of

software moves to shorter iterations, the regression testing often needs to be performed within a severely restricted time frame using limited resources.

A group of 'test case management' techniques have emerged to cope with these limitations, one of which is *test suite minimisation* (sometimes called *reduction*). A test suite minimisation technique identifies the subset of the original test suite that is deemed to be redundant, and either removes the subset from the original test suite permanently or reject it for the current iteration of regression testing.

However, testing often involves multiple test criteria and conflicting constraints. For example, different types of testing, such as functional testing and structural testing, may require different testing criteria [1]. The tester may also benefit from considering multiple testing criteria simply because there is no single most ideal testing criterion. For example, real fault detection rate may be the most ideal testing criterion but it cannot be known until testing finishes. Code coverage is a widely used surrogate, but there does not exist a guaranteed correlation between code coverage and fault detection capability [2]. Therefore, it may be valuable to complement code coverage with fault detection *history*. There may also exist other types of domain knowledge that may contribute to the accuracy and efficiency of testing process.

Apart from testing criteria, there also may exist multiple constraints on the testing process. For example, cost is one of the essential constraints because the whole purpose of test suite minimisation and prioritisation is to reduce testing cost. One important cost measure, considered by other researchers [3, 4, 5], is the execution time of the test suites. With emerging trends like the agile software development paradigm [6], regression testing often needs to be completed in even more limited time than in previous paradigms. Execution time, as well as other resources like human efforts and hardware equipments, can become critical constraints to testing process.

Existing approaches to regression test suite minimisation (and prioritisation) have been single-objective approaches that have sought to optimise a single-objective function. Even where there exists more than one objective, these multiple objectives have been combined into a single-objective fitness function. For example, the recent work on test case prioritisation [3] concerns both code coverage and cost, which is essentially a two-objective formulation of the test case prioritisation problem. However, it is dealt with by conflation to a single-objective of coverage per unit cost. Where there are multiple competing and conflicting objectives the optimisation literature recommends the consideration of a Pareto optimal optimisation approach [7, 8]. Such a

Pareto optimal approach is able to take account of the need to balance the conflicting objectives, all of which the software engineer seeks to optimise.

This paper presents the first multi-objective formulation of the test suite minimisation problem and applies a hybrid algorithm that combines existing greedy approach with a well known Multi-Objective Evolutionary Algorithm, NSGA-II. It is likely that a tester would want to optimise several, possibly conflicting constraints, for which this approach will be well suited.

The primary contributions of this paper are as follows:

1. The paper introduces a multi-objective formulation of the regression test suite minimisation problem and instantiates this with two versions: A two-objective formulation that caters for coverage and cost and a three objective formulation that caters for coverage, cost and fault history. The formulations facilitate a theoretical treatment of the optimality of the greedy algorithm and makes it possible to establish a relationship between the multi-objective problems of test case prioritisation and test suite minimisation.
2. The paper presents two algorithms for solving the two and three objective instances of the test suite minimisation problem: a re-formulation of the single–objective greedy algorithm, and a hybrid variant of NSGA-II of Deb et al. [9], which we call HNSGA-II. The hybrid nature of HNSGA-II is based on the known fact that the greedy algorithm produces a good approximation to the set cover problem, which forms the basis of the test suite minimisation problem.
3. The paper presents the results for these algorithms, when applied to the two-objective version of the problem using, as subjects, five non-trivial real world programs from Software-architecture Infrastructure Repository, SIR [10]. The results confirm the theoretical analysis, revealing cases where the search based algorithms out–perform the greedy approach. More importantly, the results show that the hybrid approach is capable of filling in large gaps in the Pareto fronts approximated by the greedy algorithm.
4. The paper also presents results from an empirical study of the algorithms applied to the three objective formulation of the problem. These results also show that the hybrid approaches can out–perform the greedy approach.

The rest of this paper is organised as follows. Section 2 describes related work. Section 3 introduces the multi-objective formulation of test suite

minimisation, giving theoretical results and connections between the minimisation and prioritisation problems. Section 4 shows that the additional greedy algorithm produces a good approximation, which forms the basis of the hybrid approach used in the present paper. Section 5 presents two empirical studies of multi-objective test suite minimisation for two and three objective versions of the multi-objective formulation. The results of the empirical studies are analysed in Section 6. Finally, Section 7 concludes with directions for future work.

2. Background

The existing literature on test case management can be categorised into three different areas of investigation; *test suite minimisation* (or *reduction*), *test case selection*, and *test case prioritisation*.

Test suite minimisation shares many similarities with test case selection. It selects a subset of the test suite that satisfies all the test requirements.

Test Case Minimisation

Given: a test suite $T = \{t_1, t_2, \ldots, t_n\}$ and a set of test requirements $R = \{r_1, r_2, \ldots, t_m\}$

Problem: to find the smallest T' such that $T' \subset T$, $\forall r \in R(T'$ satisfies $r)$.

One major difference between test suite minimisation and test case selection is that test case selection chooses a temporary subset of test cases based on the modifications made to a specific version of System Under Test, whereas test suite minimisation reduces the test suite based on some external criterion such as structural coverage. Harrold et al. formulated test suite minimisation as a minimal hitting set problem, and applied a heuristic approach [1]. This paper considers test suite minimisation as a non-weighted version of the set-cover problem, which is equivalent to the hitting set problem.

It is known that test suite minimisation can be efficient provided that the cost of the reduction is smaller than the gain in the cost of the reduced test suite [11]. However, a weakness of test suite reduction is that the removal of some test cases from the test suite may potentially reduce the fault detecting capability of the test suite too. Some studies have shown that the fault-detection capability of the test suite was indeed damaged [12], while others have shown that the reduced test suite still preserved its fault-detection

capability [13]. The reduction technique studied by Harrold et al. is of particular interest in the context of this paper because the technique considers multiple criteria when deciding whether to preserve a test case or not [1]. Their technique converts this multi-objective problem into a series of single-objective problems, by solving the problem for the first objective then uses this intermediate solution as the starting point of the solution for the next objective. While this *sequential* approach is one of the classical techniques to solve multi-objective optimisation problems, it may produce less optimal results compared to Pareto-efficient approach because the earlier objectives may restrict the possibility of finding potential solutions to the objectives solved later.

Test case selection focuses on selecting a subset of the test suite in order to test software modifications. The selection is typically made in terms of the structure of the program P and the test suite T. Several techniques have been considered, including symbolic execution [14], flow graph based [15] and dependence graph based approaches [16, 17].

Test case prioritisation is a closely related topic, in which the goal is to find an optimal order in which to execute test cases. The ideal ordering of test cases would be the one that maximises the rate of fault detection. However, since the fault information is not known to the tester in advance, prioritisation techniques have to depend on surrogates. Rothermel et al. defined test case prioritisation problem as follows:

Test Case Prioritisation

Given: a test suite, T, the set of permutations of T, PT; a function from PT to real numbers, f.

Problem: to find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

The function f acts as a surrogate for the unknown rate of fault detection. One of the most widely used metrics for $f : PT \rightarrow \mathbb{R}$ is APFD (Average Percentage of Fault Detected), which rewards orderings with earlier fault detection abilities [18]. The additional greedy algorithm is known to produce good results for the test case prioritisation problem [19, 20, 21]. Rothermel et al. introduced APFDc, the cost cognizant version of APFD [3], which inspired our formulation of the weighted objective greedy algorithm. Elbaum et al. [4] expanded the problem to incorporate not just the cost of test cases, but the severity of the detected faults.

Walcott et al. also take time into account in their work on the test case prioritisation problem [5]. Their approach to prioritisation combines both selection and prioritisation problems into a single-objective, which is the weighted sum of the selection fitness and prioritisation fitness. The coefficients used for weights are defined to ensure that selection fitness is the primary objective, while ordering is secondary.

This paper is an extension of a conference paper by the same authors [22]. This previous work applied NSGA-II and an island genetic algorithm variant of NSGA-II to both two and three-objective formulations for the programs from Siemens suite and `space`. The results showed that the search-based techniques produce wider and more efficient Pareto-frontier for smaller programs, while the additional greedy algorithm produces a good approximation for `space`. This observation provided the basis for the formulation of the hybrid approach in the present paper. The empirical results in the present paper shows that adopting the hybrid approach indeed produces better results; the approximation produced by the additional greedy algorithm is complemented by intermediate solutions that are found by NSGA-II. These intermediate solutions add valuable information to the trade-offs between testing criteria and resources.

3. Multi-Objective Paradigm

This section introduces the multi-objective formulation of test suite minimisation. Section 3.1 introduces the Pareto optimal formulation of the test suite minimisation problem. Section 3.2 explores the theoretical properties of the two-objective greedy algorithm, while Section 3.3 shows the relationship between multi-objective selection and prioritisation.

3.1. Pareto Optimality

Pareto optimality is a notion from economics with broad range of applications in game theory and engineering. The original presentation of the Pareto optimality is that, given a set of alternative allocations and a set of individuals, allocation A is an improvement over allocation B only if A can make at least one person better off than B, without making any other worse off [23].

Based on this, the multi-objective optimisation problem can be defined as the problem of finding a vector of decision variables x, which optimises

a vector of M objective functions $f_i(x)$ where $i = 1, \ldots, M$. The objective functions are the mathematical description of the optimisation criteria.

Without the loss of generality, it is assumed that the goal is to maximise f_i where $i = 1, \ldots, M$. A decision vector x is said to dominate a decision vector y (also written $x \succ y$) if and only if their objective vectors $f_i(x)$ and $f_i(y)$ satisfies:

$$\forall i \in \{1, \ldots, M\}.f_i(x) \geq f_i(y) \text{ and}$$

$$\exists i \in \{1, \ldots, M\}.f_i(x) > f_i(y)$$

All decision vectors that are not dominated by any other decision vector are said to form the *Pareto optimal set*, while the corresponding objective vectors are said to form the *Pareto frontier*. Now the multi-objective optimisation problem can be defined as follows:

Given: a vector of decision variables, x, and a set of objective functions, $f_i(x)$ where $i = 1, \ldots, M$

Definition: maximise $\{f_1(x), \ldots, f_M(x)\}$ by finding the Pareto optimal set over the feasible set of solutions.

Identifying the Pareto frontier is particularly useful in engineering because the decision maker can use the frontier to make a well-informed decision that balances the trade-offs between the objectives. The knowledge of these trade-offs proved to be useful in various other engineering domains such as architecture [24], chemical engineering [25], and aerodynamics [26].

The multi-objective test suite minimisation problem is to select a Pareto efficient subset of the test suite, based on multiple test criteria. It can be defined as follows:

Multi-Objective Test Suite Minimisation

Given: a test suite, T, a vector of M objective functions, f_i, $i = 1, \ldots, M$.

Problem: to find a subset of T, T', such that T' is a Pareto optimal set with respect to the set of objective functions, f_i, $i = 1, \ldots, M$.

The objective functions are the mathematical descriptions of test criteria concerned. A subset t_1 is said to dominate t_2 when $(\{f_1(t_1), \ldots, f_M(t_1)\})$, the decision vector for t_1 dominates that of t_2. The resulting subset of the test

suite, T', has several benefits in regards to the regression testing, as shown in Section 3.2.

3.2. Properties of 2-Objective Coverage Based Selection

Here we instantiate the two objective formulation with code coverage as a measure of test adequacy and execution time as a measure of cost. Thus, code coverage becomes one of the two objectives, and it should be maximised for a given cost. Time is the other objective, which should be minimised for a given code coverage.

In this instantiation of the problem, should there exist a subset of test suite S with coverage C and execution time T on the Pareto frontier, it means that:

- **T1.** No other subset of S can achieve more coverage than C without spending more time than T.
- **T2.** No other subset of S can finish in less time than T while achieving a coverage that is equal to or greater than C.

This is the implication of Pareto optimality. Rather than obtaining a single answer that approximates the global optimum in the search space for a single objective, we obtain a set of points, each of which denotes one possible way of balancing the two objectives in a globally optimal way. Each member of the Pareto frontier is therefore a candidate solution to the problem, upon which it is not possible to improve.

In the single objective formulation of test suite minimisation, greedy algorithms have been used to maximise coverage. The greedy approach starts with an empty test set as the 'current solution' and iteratively adds a test case which gives the most coverage of those that remain. A variant, additional greedy, improves on this by adding to the current solution the test case that gives the best *additional* coverage to the current solution. Each addition by the greedy algorithm of a new test case to the 'current solution' denotes a candidate element of the Pareto frontier.

Greedy algorithms have proved effective for the single objective formulation, so they make a sensible starting point for the consideration of the multi-objective formulation. In order to optimise both coverage and cost, the additional greedy algorithm will need to be formulated to measure not coverage, but coverage per unit time. This produces a single objective cost

cognizant variant of the greedy algorithm, similar to that used by Malishevsky et al. for the single objective prioritisation problem [3].

Suppose that the additional greedy algorithm has chosen a test case t that covers a set of structural elements, s. Let $Cov(s)$ be the coverage of test case t and let $Time(t)$ be the execution time of t. Assume that the selection of t increases the coverage by $\Delta Cov(s)$. By definition, there is no single test case t' (which would cover s') that the algorithm could have chosen, such that $\Delta Cov(s') > \Delta Cov(s)$ and $Time(t') \leq Time(t)$ (otherwise the algorithm would have picked t'). Therefore, the selection of a test case made by the two objective cost cognizant additional greedy algorithm cannot be improved upon by the addition of another single case. However, this leaves open the possibility that there may be a *set* of test cases that, taken together, could have produced a better approximation to the Pareto front.

Let us consider the case of the basic greedy algorithm that selects one test case at a time. It turns out that any selection of a test case made by the additional greedy algorithm can only be improved with respect to **T2**. It is not possible to improve on the selection made by the additional greedy algorithm with respect to **T1**. This observation is stated and proved more formally below.

Proposition 1 (Partial Optimality).
The selection of a test case made by the additional greedy algorithm cannot be improved upon with respect to **T1**.

Proof 1. *Suppose the contrary. That is, let t_1 be a test case that covers a set of structural elements, s_1. Suppose there also exists a pair of test cases, t_2 and t_3, covering s_2 and s_3 respectively, that together improve upon t_1 by achieving more coverage without spending more time. By definition, we have*

$$\frac{\Delta Cov(s_2)}{Time(t_2)} < \frac{\Delta Cov(s_1)}{Time(t_1)} \text{ and } \frac{\Delta Cov(s_3)}{Time(t_3)} < \frac{\Delta Cov(s_1)}{Time(t_1)} \quad (1)$$

because, otherwise, the additional greedy algorithm would not have selected t_1. From this, it follows that

$$Time(t_1) \cdot (\Delta Cov(s_2) + \Delta Cov(s_3)) < \Delta Cov(s_1) \cdot (Time(t_2) + Time(t_3)) \quad (2)$$

However, in order for t_2 and t_3 to be collectively a better choice than t_1 we require t_2 and t_3 to achieve higher increase in coverage, taking no longer than t_1. That is,

$$\Delta Cov(s_2 \cup s_3) > \Delta Cov(s_1) \tag{3}$$

and

$$Time(t_2) + Time(t_3) \leq Time(t_1) \tag{4}$$

Combining step 2 and step 4, we get: $\Delta Cov(s_2) + \Delta Cov(s_3) < \Delta Cov(s_1)$. *Now, because code coverage is a set theoretic concept, it is not possible for the coverage of the union to be greater than the sum of the coverage of the parts. Therefore we have:* $\Delta Cov(s_2 \cup s_3) \leq \Delta Cov(s_2) + \Delta Cov(s_3)$. *By transitivity,* $\Delta Cov(s_2 \cup s_3) < \Delta Cov(s_1)$, *which contradicts step 3, so we must conclude that it is not possible to dominate the selection made by the additional greedy algorithm by breaking* ***T1****.*

	Program Points										Exec. Time
t_1	X	X	X	X	X	X	X	X			4
t_2	X	X		X	X	X	X	X	X	X	5
t_3	X	X	X						X		3
t_4	X	X	X	X	X						3

Table 1: An example of a test suite where the additional greedy algorithm produces suboptimal minimisation of test cases

However it is possible to construct an example that shows that the additional greedy algorithm does not produce solutions that are Pareto efficient with respect to **T2**. Such an example is shown in Table 1. The first choice of the additional greedy algorithm will be t_1, which has the additional coverage per unit time value of $\frac{0.8}{4} = 0.2$ (T_2, T_3, T_4 each has 0.18, $0.1\tilde{3}$, and $0.1\tilde{6}$). The second choice will be t_2 with the additional coverage per unit time value of $\frac{0.2}{5} = 0.04$, whereas t_3 and t_4 each has $0.0\tilde{3}$ and 0. At this point, the algorithm achieves 100% coverage in 9 units of time. However, the same amount of coverage is also achievable in 8 units of time by selecting t_2 and t_3, so the subset $\{t_2, t_3\}$ dominates the subset $\{t_1, t_2\}$.

It is indeed possible to extend the greedy approach to consider a pair of test cases, rather than a single test case, at a time, to overcome this problem. This formulation of the greedy approach is often called a *2-way*

greedy algorithm. Then, however, it would be possible to construct another counter-exmple that consists of a set of 3 test cases. Eventually, for n test cases, an n-way greedy approach is required to ensure its Pareto-optimality with respect to **T2**. However, the n-way greedy approach would be identical to an exhaustive search, which is not practical.

Furthermore, though the additional greedy algorithm may produce points that are Pareto efficient with respect to **T1**, it does not produce a complete Pareto frontier. The existence of t_4 in the above example demonstrates this. According to the additional greedy algorithm, the first decision point chosen for this example would be the subset of $\{t_1\}$, which achieves 80% coverage in 4 units of time. The subset $\{t_1\}$ is on the Pareto frontier because no other test case can achieve 80% coverage in 4 units of time. However, the subset of $\{t_4\}$ is *also* on the Pareto frontier, because no other test case can achieve 50% coverage in 3 units of time. This point $\{t_4\}$ on the Pareto frontier is ignored by the additional greedy algorithm. As we will see in the next subsection, this issue is important, because it is necessary to produce the most complete approximation to the Pareto frontier possible in order to exploit the relationship between multi-objective selection and prioritisation.

3.3. The Relationship Between Multi Objective Selection and Prioritisation

While they are formally different concepts, test suite minimisation and test case prioritisation problems are closely related to each other. Test case prioritisation concerns the ideal ordering of a given test suite. Since it only changes the order of a given test suite, it is not capable of producing an efficient test case scheduling when the available time is shorter than the total time required by the test suite, assuming that the test suite can be executed in its entirety.

Figure 1 shows the result that the additional greedy algorithm produces with the test data shown in Table 1, along with the real Pareto frontier of the test data. If the tester applies the existing test case prioritisation techniques based on the additional greedy algorithm, the algorithm will produce an ordering of $t_1 - t_2$ for the test cases in Table 1.

If the tester is allowed 9 units of time for the testing, it is possible to follow the ordering produced by the additional greedy algorithm; t_1 is executed first, followed by t_2, at which point a final coverage of 100% is achieved in 9 units of time. However, let us suppose that the testing environment allows only 6 units of time for the testing. According to the additional greedy algorithm, the tester should fall back to the first data point, $\{t_1\}$, since it is not possible

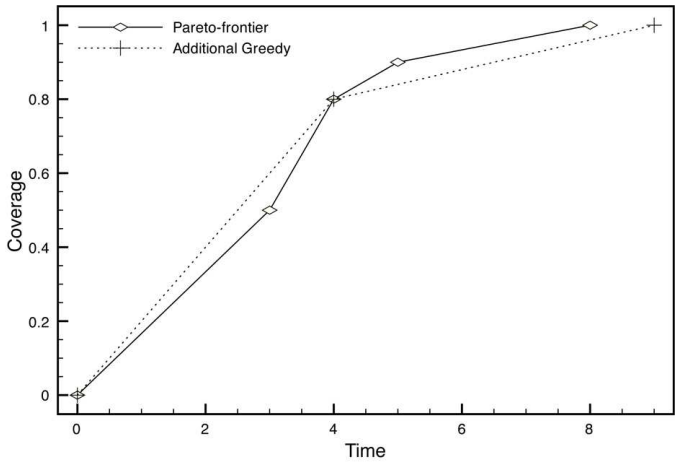

Figure 1: Comparison between the Pareto frontier and the results of the additional greedy algorithm from the test data shown in Table 1

to execute both t_1 and t_2 in the given time. This achieves 80% coverage in 4 units of time, but leaves 2 units of time unused. However, the Pareto frontier tells us that the subset of $\{t_2\}$ can achieve 90% coverage in 5 units of time, making more efficient use of the given time. Similarly, it also reveals that, should the budget allow only 3 units of time, it is still possible to achieve 50% coverage by executing T_4. Furthermore, the Pareto frontier also shows us that a coverage of 100% is achievable in only 8 units of time, which is shorter than the 9 units of time predicted by the additional greedy algorithm.

Concerning the limitations of the testing environment, it is possible to consider a few different scenarios. First, it may be the case that the entire regression can be executed, regardless of the amount of time it takes. In this scenario, only prioritisation matters as the complete test suite can be executed. Second, there may be a case where the tester is given an exact amount of time available for the regression testing. The benefits of knowing the existence of $\{t_2, t_3\}$ and $\{t_4\}$ as candidate selections of test cases becomes clear under this scenario, where there is a cost constraint, i.e., testing budget. Prioritisation alone cannot optimise the testing process in such a situation, because it is not capable of *selecting* test cases. Fortunately, the Pareto efficient approach enables the tester to make an informed decision about the regression testing under such constrained scenarios. Finally, there may exist

cases where the tester does not know the exact amount of time available for the regression testing but has to cater for the possibility that testing may be stopped abruptly. In this scenario, only prioritisation matters because it seeks to maximise the early fault detection. However, the knowledge of trade-offs between testing criteria and testing budget can still provide a valuable frame of reference when measuring the progress of testing procedure.

The second scenario discussed here is increasingly relevant because of the trend towards shorter development cycles. For example, nightly build is widely adopted in open source software development [27]. This process usually involves regression testing, which naturally needs to be completed within tight, inflexible time constraints. In order to construct an efficient test sequence under such constraints, an appropriate subset of test cases should be selected first. This subset can subsequently be prioritised in order to achieve the ideal ordering among the selected test cases. This way, test suite minimisation and test case prioritisation techniques can be used in combination in order to achieve more efficient regression testing.

4. Greedy Algorithm

Code coverage is a discrete and bounded concept, in a sense that there exist only a finite number of entities (i.e. statements or blocks) to *cover*. Therefore, the problem of maximising code coverage can be reformulated as a weighted set-cover problem as follows:

Weighted Set Cover Problem
Given: a universe $\mathcal{U}$ with n elements, a set $\mathcal{S}$ of m subsets of $\mathcal{U}$ with $cost_1, \ldots, cost_m$
Problem: to find $\mathcal{C}$ such that $\mathcal{C} \subseteq \mathcal{S}$, $\bigcup_{S_i \in \mathcal{C}} S_i = \mathcal{U}$, and $\forall S' \subseteq \mathcal{S}[\bigcup_{S_i \in \mathcal{S}'} S_i = \mathcal{U} \rightarrow \sum_{S_i \in \mathcal{C}} cost_i \leq \sum_{S_i \in S'} cost_i]$

The additional greedy algorithm is illustrated in Algorithm 1. Let $\mathcal{U}$ be the universe, $\{e_1, \ldots, e_n\}$; $\mathcal{S}$ the set containing $S_1, \ldots, S_m$, the subsets of $\mathcal{U}$ such that $\bigcup_i S_i = \mathcal{U}$; $cost_1, \ldots, cost_m$ the cost of each subset in $\mathcal{S}$. Without loss of generality, it is assumed that there exists a subset $S' \subset \mathcal{S}$ that covers $\mathcal{U}$ completely. Through line (4) of Algorithm 1, the additional greedy algorithm picks $S_j \in \mathcal{S}$ according to the density of the set, $cost_j/|S_j - C|$. The minimum density corresponds to the maximum *increase in coverage* per *cost* in each iteration.

Algorithm 1: Outline of additional greedy algorithm
AdditionalGreedy($\mathcal{U}$, $\mathcal{S}$)
(1) $C \leftarrow \phi$ // covered elements in $\mathcal{U}$
(2) **repeat**
(3) $j \leftarrow min_k(cost_k/|S_k - C|)$
(4) add S_j to solution
(5) $C = C \bigcup S_j$
(6) **until** $C = \mathcal{U}$

The set-covering problem is known to be NP-hard [28]. Fortunately, the additional greedy algorithm is also empirically known to be efficient for solving test case prioritisation problems [18]. The weighted set-cover problem is essentially a selection problem. However, the requirement in test case prioritisation, i.e. to maximise the test adequacy as early as possible, combined with the inherent nature of the algorithm, allows the greedy algorithm to simultaneously produce an efficient solution for the prioritisation problem, which is a sequencing problem.

With respect to **T2** in Section 3.2, it is known that the additional greedy algorithm gives an approximation ratio of $\ln n$ [29], with n being the input size, i.e. the size of test suite in the context of the present paper. However, the theoretical analysis in Section 3.2 suggests that there exist solutions that are better than those approximated by the greedy algorithm. This paper introduces a hybrid approach that utilises both the deterministic additional greedy algorithm and a population-based multi-objective genetic algorithm in order to produce better solutions more efficiently.

5. Empirical Studies

This section explains the experiments conducted to explore the two and three objective formulations of the multi-objective selection problem. Section 5.1 describes subjects studied and Section 5.2 describes the objectives to be optimised. Section 5.3 describes the algorithms studied, while Section 5.4 explains the mechanisms by which these algorithms will be evaluated in the two empirical studies. Finally Section 5.5 sets out research questions.

5.1. Subjects

A total of 5 programs are studied in this paper: the program `space` from the European Space Agency, and the GNU open-source programs `flex`, `grep`, `gzip`, and `sed`. These programs range from 6,199 to 19,737 lines of code, and they are all real world applications with non-trivial test suites. The software artifacts and their test suites were available from Software-artifact Infrastructure Repository (SIR) [10].

For GNU open-source programs, the largest test suite available from the archive was selected. The program `space` has multiple test suites; the test suite that provides the highest code coverage was selected among the available test suites. The size of the programs and their test suites are shown in Table 2.

Program	Lines of code	Test suite size	Description
`flex`	15,297	567	Lexical analyser
`grep`	15,633	806	Regular expression utility
`gzip`	8,889	213	Compression tool
`sed`	19,737	370	Non-iterative text editor
`space`	6,199	156	European Space Agency program

Table 2: Size of test suites and studied programs

5.2. Objectives

It is not the aim of this paper to enter into a discussion concerning which objectives are more important for regression testing. We simply note that, irrespective of arguments about their suitability, coverage and fault histories are likely candidate objectives for assessing test adequacy and that execution time is one realistic measure of effort. It also should be noted that it is not the intention of the present paper to confine the test criteria to coverage-based measurements. The formulations for which the paper reports results serve to illustrate the possibilities created by a multi-objective approach.

For the two-objective formulation, statement coverage and computational cost of test cases will be used as objectives. The additional objective used in the three objective formulation is the past fault detection history. Each software artifact used in this paper has several seeded faults (taken from the

data available on the SIR [10]). SIR records the test cases that reveal these faults. Using this information, it is possible to assign past fault coverage to each test case subset, corresponding to how many of the known past faults in the previous version this subset would have revealed (for the evaluation of the three-objective formulation, `grep` was excluded because no historical fault information was available for it).

The statement level coverage information used in this paper was measured using the GNU compiler, `gcc`, and its profiling tool, `gcov`. The instrumentation is performed by `gcc` during compilation. The instrumented subject program produces execution-trace information with each execution, which is converted to statement level coverage information using `gcov`.

Physical execution time of test cases is hard to measure accurately. Measurement is confounded by many external factors; different hardware, application software and operating system. In particular, any measurement of execution time is likely to be affected by aspects of the environment unconnected to the choice of test cases. Such factors include concurrent execution, caching and other low-level processor optimisations.

In this paper we circumvent these issues by using the software profiling tool, Valgrind, which executes the program in an emulated, virtual CPU [30]. The computational cost of each test case was measured by counting the number of virtual instruction codes executed by the emulated environment. Valgrind was created to allow just this sort of precise and unequivocal assessment of computational effort; it allows us to argue that these counts are directly proportional to the cost of the test case execution.

5.3. Algorithms

5.3.1. Hybrid Multi-Objective Algorithm

Population-based genetic algorithms are inherently well-suited to multi-objective optimisation problems because of their ability to retain multiple solutions. In this paper NSGA-II is used. NSGA-II is a multi-objective genetic algorithm developed by Deb et al. [9]. It produces a group of solutions that collectively form the final state of the Pareto-frontier when the algorithm terminates. NSGA-II is based on elitism, which means that, unless better solutions are found, the *best-so-far* solutions are always retained.

NSGA-II differs from normal genetic algorithms in two major aspects. First, selection of individual solutions for the next generation is based on Pareto optimality. NSGA-II uses an algorithm called fast-nondominated sorting, which classifies individual solutions into different dominance levels.

Algorithm 2: Outline of the main loop for NSGA-II
NSGA-II-Main-Loop(Generation counter, t, Parent population, P_t, Children population, Q_t, Population size, N)

(1) $R_t \leftarrow P_t \cup Q_t$
(2) $\mathcal{F} \leftarrow$ fast-nondominated-sort(R_t)
(3) $P_{t+1} \leftarrow \emptyset$ and $i \leftarrow 1$
(4) **repeat**
(5) Crowding-distance-assignment$(\mathcal{F}_i)$
(6) $P_{t+1} \leftarrow P_{t+1} \cup \mathcal{F}_i$
(7) $i \leftarrow i + 1$
(8) **until** $|P_{t+1}| + |\mathcal{F}_i| \leq N$
(9) $Sort(\mathcal{F}_i, \prec_n)$
(10) $P_{t+1} \leftarrow P_{t+1} \cup \mathcal{F}_i[1 : (N - |P_{t+1}|)]$
(11) $Q_{t+1} \leftarrow$ make-new-population(P_{t+1})
(12) $t \leftarrow t + 1$

For example, solutions on the current Pareto-frontier get assigned dominance level of 0. Then, after taking these solutions out, fast-nondominated sorting calculates the Pareto-frontier of the remaining population; solutions on this second frontier get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation.

The second difference concerns the concept of crowding distance. When NSGA-II has to select one out of two individual solutions with the same dominance level, it relies on the crowding distance to make the selection. Intuitively, the crowding distance of an individual solution is the normalised sum of distances from other individuals with respect to each objective. An individual with higher crowding distance is located in the part of the solution space that is sparsely populated. In order to obtain wider Pareto-frontiers, NSGA-II rewards individuals with higher crowding distance when the dominance level is the same. In the context of test suite minimisation under cost-constraints, it means that the algorithm seeks to produce a higher variety of decision points that correspond to different testing budgets.

Algorithm 2 presents a top-level outline of the main loop of NSGA-II. In line (2), Fast-Nondominated-Sort() assigns dominance level to individuals. In the loop that spans from line (4) to (8), the algorithm adds as many non-dominated frontiers (i.e. sets of solutions with the same domi-

nance level) as possible to the next generation. Any remaining slots in the next generation are filled in according to the descending order of crowding distance in line (9) and (10).

The present paper utilises a variant of the basic NSGA-II, called Hybrid NSGA-II (HNSGA-II). The additional greedy algorithm is known to produce an efficient approximation to the set cover problem [29], but Section 3.2 also shows that there can exist solutions that are better than those found by the greedy algorithm. Therefore, it is natural to complement the additional greedy algorithm with NSGA-II by taking the results from the additional greedy algorithm as the initial population, which is how HNSGA-II works. Thanks to elitism, the approximation produced by the additional greedy algorithm will be discarded only when more efficient solutions are found. Additionally, HNSGA-II also includes the same number of random solutions as the number of solutions produced by the additional greedy algorithm in order to guarantee a certain level of diversity in the initial population.

HNSGA-II algorithm was executed 20 times for each subject program in order to account for their inherent randomness. HNSGA-II uses single-point crossover and bit-flip mutation. Crossover rate and mutation rate were set to 0.1. Because of the hybrid approach to the initial population, HNSGA-II uses different population size and maximum fitness evaluation count for each subject programs, which is shown in Table 3.

Program	Population size	Max. fitness evaluation
`flex`	92	9,200
`grep`	154	15,400
`gzip`	42	4,200
`sed`	64	6,400
`space`	232	23,200

Table 3: HNSGA-II Configurations for Subject Programs

5.3.2. Greedy Algorithms

Two Greedy Algorithms were also implemented. For the two-objective formulation, a cost cognizant version of the additional greedy algorithm was implemented. For the three objective formulation, code coverage, fault coverage and execution time were combined by taking the weighted sum of [code

coverage per unit time] and [fault coverage per unit time] using the classical weighted-sum approach. With M different objectives, f_i with $i = 1, 2, \ldots, M$, the weighted-sum approach calculates the single-objective, f', as follows:

$$f' = \sum_{i=1}^{M} (w_i \cdot f_i), \sum_{i=1}^{M} w_i = 1$$

Both the additional code coverage per unit time and additional past fault coverage per unit time were combined using coefficients of 0.5 and 0.5, thereby giving equal weighting to each objective.

5.4. Evaluation Mechanisms

The difficulty of evaluating Pareto frontiers lies in the fact that the absolute frame of reference is the *real* Pareto frontier, which, by definition, is impossible to know *a priori*. Instead, a reference Pareto frontier can be constructed and used when comparing different algorithms with respect to the Pareto frontiers that they produce. The reference frontier represents the hybrid of all approaches, combining the best of each. It is one of the advantages of Pareto optimality that results for various approaches can be combined in this way.

More formally, let us assume that we have N different Pareto frontiers, P_i with $i = 1, 2, \ldots, N$. A reference Pareto frontier, P_{ref}, can be formulated as follows. Let P' be the union of all P_i with $i = 1, 2, \ldots, N$. Then:

$$P_{ref} \subset P', (\forall p \in P_{ref})(\nexists q \in P')(q \succ p)$$

For all the programs, the reference Pareto-frontier was produced by combining the result of the additional greedy algorithm and the multiple executions of HNSGA-II.

One of the methods to compare Pareto frontiers is to look at the number of solutions that are not dominated by the reference Pareto frontier. By definition, P_{ref} is not dominated by any of the N different Pareto frontiers, because it consists of the best parts of the different Pareto frontiers. However, each of N different Pareto frontiers may be partly dominated by P_{ref}. Therefore, these N different Pareto frontiers can be compared with each other by counting the number of solutions that are not dominated by P_{ref} in each Pareto frontier.

Another meaningful measurement is the size of each Pareto frontier. Achieving larger Pareto frontiers is one of the important goals of Pareto

optimisation. This is particularly of concern in engineering application, because a larger Pareto frontier means a larger number of alternatives available to the decision maker.

Both the number of non-dominated solutions and the size of Pareto frontiers were measured and statistically analysed in this paper using an one-sided Wilcoxon signed-rank test. The Wilcoxon signed-rank test is a non-parametric hypothesis test that does not require any assumption on the parametric distribution of the samples. It tests the null hypothesis that the means of two normally distributed groups are equal. In the context of this paper, the null hypothesis is that with two different algorithms, the mean values of the size of Pareto frontiers and the number of solutions that are not dominated by the reference Pareto frontier are equivalent. For these tests the α level was set to 0.95. Significant $p - values$ suggest that the null hypothesis should be rejected in favour of the alternative hypothesis, which states that one of the algorithm produces a larger Pareto frontier or a larger number of non-dominated solutions.

5.5. Research Questions

This paper aims to provide empirical evidence to answer the four research questions stated below. **RQ1** concerns whether the use of the hybrid multi-objective optimisation technique can be validated by identifying solutions that cannot be found by single-objective greedy approach.

RQ1: Do the situations theoretically predicted in Section 3.2 arise in practice? That is, does there exist a situation in which the Pareto efficient approach can provide the tester with additional information, either by finding solutions that achieve identical coverage in less time or by producing additional points on the Pareto-frontier?

If the answer to **RQ1** is positive, i.e. if solutions formally identified as theoretically possible in Section 3.2 actually exist in practice, by definition of the elitism in the hybrid algorithm guarantees improved results. **RQ2** and **RQ3** concern how much improvement can be observed and at what cost. Improvements are measured by the contribution to the reference frontier as described in Section 5.4. The costs of improvement are evaluated by the execution time of optimisation algorithms.

RQ2: How much improvement, in terms of the number of non-dominated solutions, do the greedy and the hybrid algorithm produce for the 2-objective formulation? What is the cost of the improvement?

RQ3: How much improvement, in terms of the number of non-dominated solutions, do the greedy and the hybrid algorithm produce for the 3-objective formulation? What is the cost of the improvement?

Finally, **RQ4** concerns the potential insights that can be obtained by performing the multi-objective optimisation of test suite minimisation.

RQ4: What can be said about the shape of the Pareto frontiers, both approximated and optimal? What insights do they reveal concerning the tester's dilemma as to how to balance the trade-offs between objectives?

The first three research questions will be answered quantitatively using the approaches described in Section 5.4. The last research question is more qualitative in nature.

6. Results and Analysis

The results for the 2-objective formulation for the five different subjects are shown in Figure 2. The figures are provided for illustration and qualitative evaluation only. For complete quantitative data, see Table 4. Plotted data-points for the reference Pareto-frontier and the additional greedy algorithm represent the entire respective results. With HNSGA-II, a single execution out of 20 executions was chosen for each subject program, in order to increase the readability. The variance in its complete results over 20 runs can be seen in Table 4.

The results provide a positive answer to **RQ1**; the results for `flex` and `gzip` show that HNSGA-II is capable of finding solutions that are not found by the additional greedy algorithm. For both programs, the groups of data-points on the right end, produced by the additional greedy algorithm, indicate that after a certain point a large amount of computational resource is required in order to cover the remaining small amount of the program. However, the intermediate data-points produced by HNSGA-II suggest that it is still possible to increase the coverage without requiring the same amount of

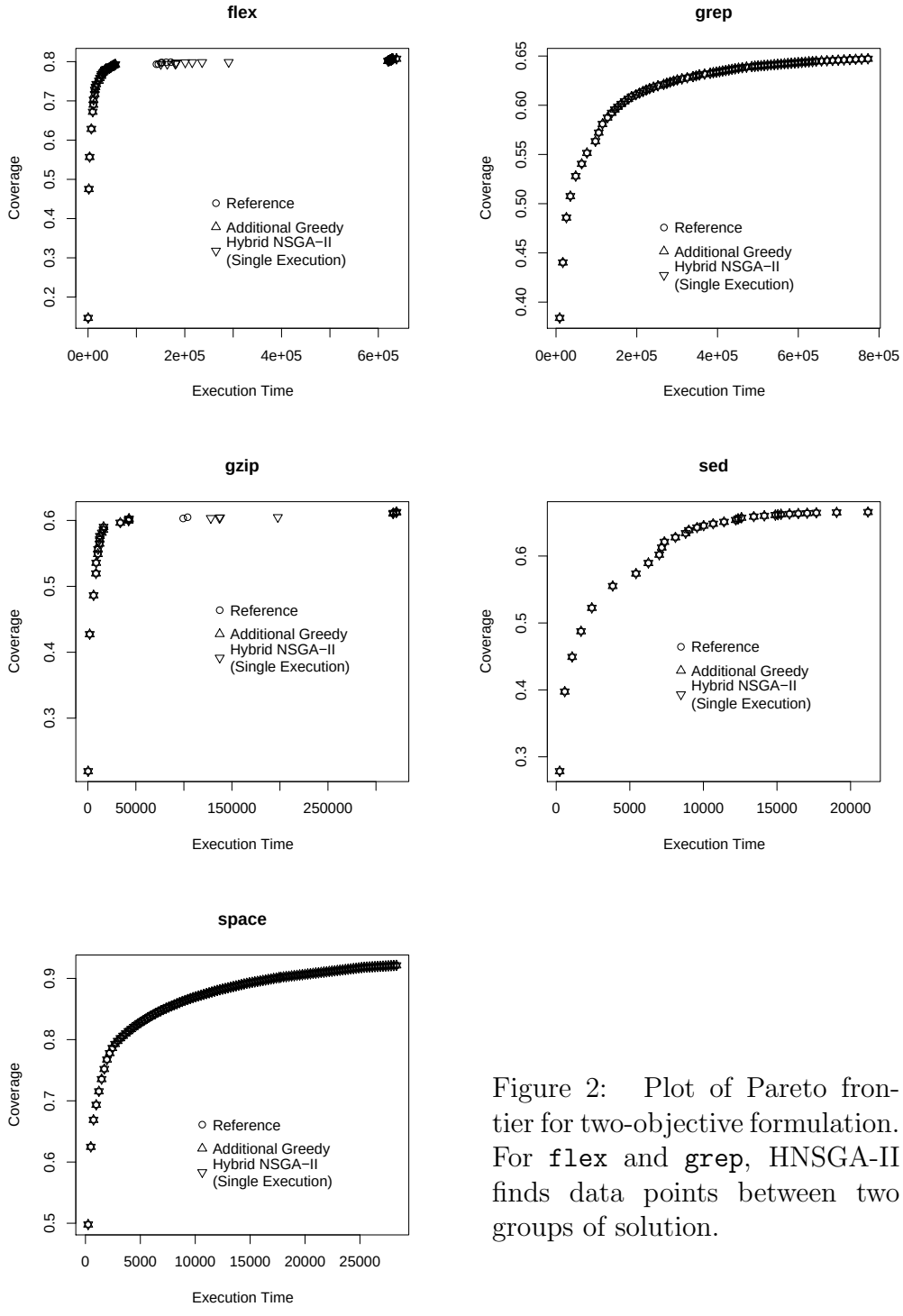

Figure 2: Plot of Pareto frontier for two-objective formulation. For `flex` and `grep`, HNSGA-II finds data points between two groups of solution.

resource. In a testing environment with limited resource, these data-points may provide important alternatives to the tester.

While it is difficult to read from the plotted graphs, Table 4 shows that some of the solutions produced by the additional greedy algorithm for `flex` and `gzip` are dominated by HNSGA-II with statistical significance. Note that, for HNSGA-II, all solutions found by the algorithm are by definition non-dominated because of the elitism; therefore, we only present the number of non-dominated solutions, n_{ND}, and do not record the size of the Pareto-frontier separately. For these programs, the size of the reference frontier is larger than the number of non-dominated solutions produced by the additional greedy algorithm, implying that HNSGA-II produced and contributed solutions that dominate some of those produced by the additional greedy.

Apart from the additional solutions found for `flex` and `gzip`, the Pareto-frontiers produced by HNSGA-II are largely non-dominated by the reference frontier, suggesting that HNSGA-II succeeds at finding the solutions on the reference frontier. It is noticeable that, with `grep`, `sed` and `space`, none of the solutions produced by the additional greedy algorithm is dominated by the reference frontier, meaning that HNSGA-II was not able to improve on these solutions. This suggests that the results may indeed be close to the optimal Pareto-frontier. Combined with **RQ1**, this answers **RQ2** by showing the existence of additional solutions on Pareto-frontier. The approximations produced by the additional greedy algorithm are close to the optimal solution. However, the hybrid approach complements the approximations by finding intermediate solutions that cannot be found by the additional greedy algorithm.

Figure 3 shows the results for the three objective formulation. The 3D plots display the solutions produced by the weighted-sum additional greedy algorithm and HNSGA-II. The intermediate points that were found by HNSGA-II in two-objective formulation are still present in the 3D plot, showing that HNSGA-II is capable of improving upon the solutions produced by the additional greedy algorithm in three-objective formulation as well. Table 5 also shows that HNSGA-II contributes solutions that dominate those produced by the additional greedy algorithm in programs `flex`, `gzip` and `space` with statistical significance. Let us also note that, while it fails to gain statistical significance, the result for `sed` consistently produce higher $\bar{n}_{ND}$ values than n_{ND} values by the additional greedy. These results provide a positive answer to **RQ3**.

In order to provide a more concrete quantitative analysis of the answers

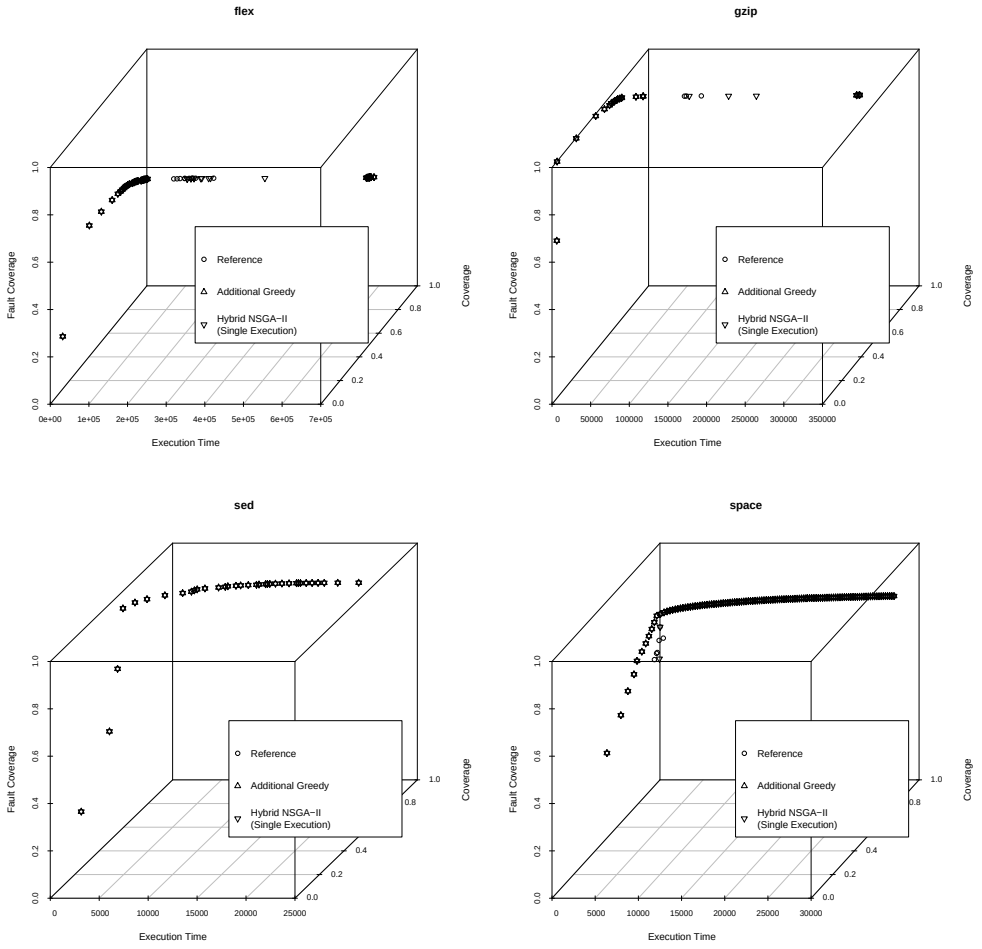

Figure 3: Plots of Pareto frontier for three objective formulation. Programs with fault-history show an elbow-point where the rates of both coverages change.

2 Objectives						
Program	Reference	Add.Greedy	HNSGA-II			
	Size	Size	n_{ND}	$\bar{n}_{ND}$	σ_{ND}	p
`flex`	52	46	46	51.80	1.58	$3.6\text{x}10^{-9}$
`grep`	77	77	77	77	0	1.0
`gzip`	23	21	20	23.60	1.05	$3.2\text{x}10^{-9}$
`sed`	32	32	32	32	0	1.0
`space`	116	116	116	116	0	1.0

Table 4: The statistical analysis of the results for two-objective formulation. The average size of Pareto-frontiers for `flex` and `gzip`, which is larger than those of the additional greedy algorithm, reflects the findings of the intermediate solutions for those programs shown in Figure 2. Also, only 108 of 116 solutions provided by the additional greedy for `space` are non-dominated by the reference frontier, implying that 8 of its solutions were dominated by the solutions produced by HNSGA-II.

to **RQ2** and **RQ3**, we compare the results obtained using tests for statistical significance. In two-objective formulation, HNSGA-II consistently produces Pareto-frontiers of the same size as the additional greedy algorithm for `grep`, `sed` and `space`. It also produces larger Pareto-frontier than the additional greedy algorithm for `flex` and `gzip`; the observed $p - values$ for the comparison of the frontier size are significant at the 95% level. The additional solutions produced by HNSGA-II for `flex` and `gzip` are the intermediate solutions we have observed in the plotted graphs.

For `grep`, `sed` and `space`, HNSGA-II consistently produces the same number of non-dominated solutions ($\bar{n}_{ND}$) as the additional greedy algorithm. The observed $p - value$ for `flex` and `gzip`, however, *accepts* the alternative hypothesis at 95% significance level, which means HNSGA-II has produced larger numbers of non-dominated solutions for these subject programs.

In three-objective formulation, HNSGA-II constantly produces Pareto-frontiers of a size larger than that of the additional greedy algorithm for `sed` (i.e. $\bar{n}_{ND} = 35$ compared to $n_{ND} = 33$). While the result fails to gain statistical significance, it shows that HNSGA-II improved upon the result from the additional greedy algorithm. For `flex`, `gzip` and `space`, the observed p-values are significant at the 95% level, resulting in the acceptance of the alternative hypothesis (i.e. HNSGA-II produces larger number of non-

dominated solutions than the additional greedy).

Turning to the last research question, **RQ4**, a more qualitative analysis is required. This is made possible by the visualisations of the solutions plotted in Figures 2 and 3, as well as the two-dimensional projections in Figure 4 and 5. In two-objective formulation, it is noticeable that all the programs share similar shape of Pareto-frontier with an *elbow point* where the rate of increase in the coverage changes abruptly. Such *elbow points* are considered important in the study of Pareto optimality. They indicate points of particular interest where the balance of trade offs inherent in the objectives changes.

With `flex` and `gzip`, the changes are even more extreme in a sense that the rightmost end of the plot is separated from the rest of the solutions. This change of the increase rate and the resulting flat tail may suggest a general relation between the code coverage and the computational cost, implying that there only exist a certain part of the software that is relatively inexpensive to cover. It is also interesting that the similarity in the shape of Pareto-frontier is shared across programs with incomplete test suites with less-than-optimal total coverage (`flex`, `grep`, `gzip`, `sed`) and the program with a rather complete test suite (`space`).

In three-objective formulation, there still exist elbow points in all program. More interestingly, the change in the increase rate does not only involve coverage vs. execution time, but fault coverage vs. code coverage as well, which is particularly evident in the case with `sed`. It is observed that once past the elbow point, the past fault detection rate rarely increases while the code coverage still increaes. These results provide evidence to suggest that the faults seeded into these programs under controlled environment is rather concentrated within parts of the software that is relatively inexpensive to cover. However more data is required to see whether this phenomenon is specific to the set of programs we have chosen to study in the present paper, or whether it is generic to test case.

Another important observation is that the weighted-sum greedy algorithm performs very well, despite the known fact that it does not necessarily cope well with multi-objective optimisation problems. This provide evidence to suggest that, within the program studies in the paper, there may exist a strong correlation between the code coverage and the past fault coverage. However, it is subject to debate whether the code coverage subsumes the fault coverage, and more data is required to see whether this is a generic tendency between two coverage concepts.

Table 6 contains the average execution time of each algorithm. The ex-

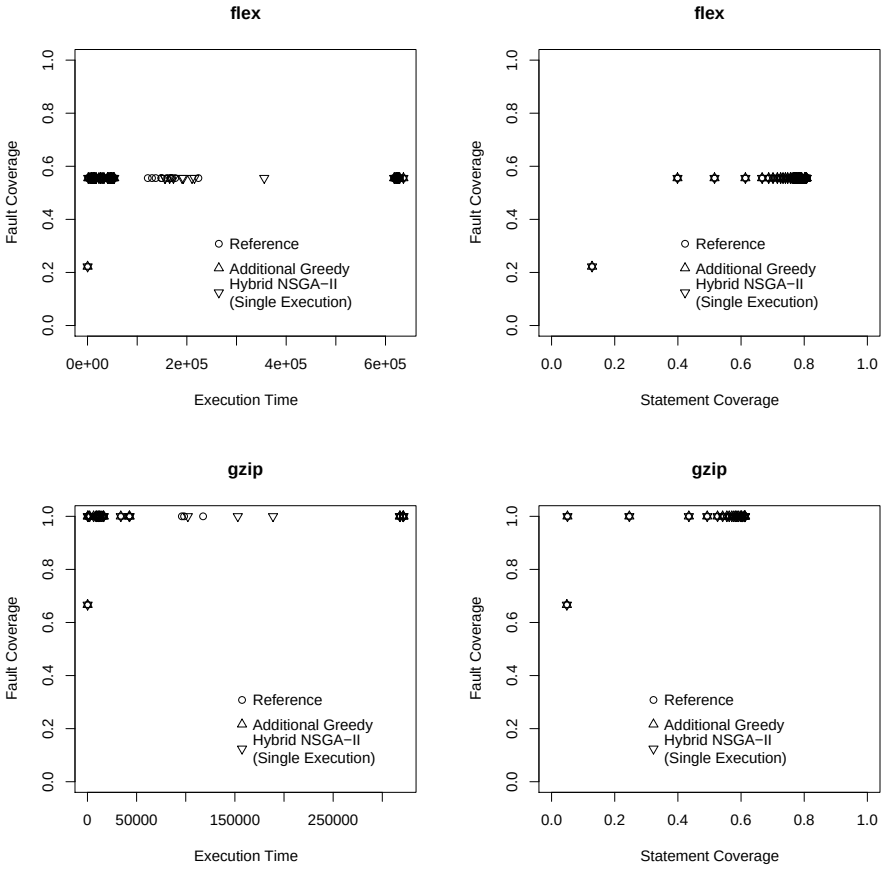

Figure 4: Two-dimensional projections of Figure 3 for `flex` and `gzip`. It can be observed that the fault coverage is often maximised before the statement coverage is fully maximised.

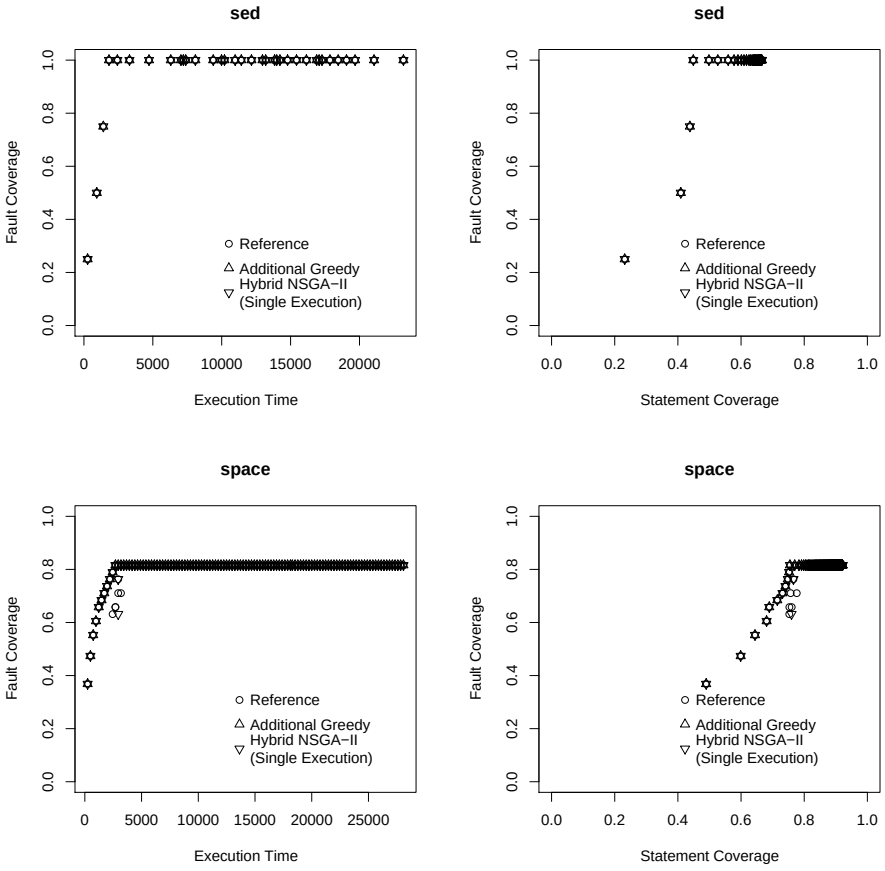

Figure 5: Two-dimensional projections of Figure 3 for `sed` and `space`. It can be observed that the fault coverage is often maximised before the statement coverage is fully maximised.

3 Objectives						
Program	Reference	Add.Greedy	HNSGA-II			
	Size	Size	n_{ND}	$\bar{n}_{ND}$	σ_{ND}	p
`flex`	52	46	45	53.55	1.79	$3.6\text{x}10^{-9}$
`gzip`	25	22	22	25.35	0.87	$2.9\text{x}10^{-9}$
`sed`	35	35	33	35	0	1.0
`space`	121	115	99	115.35	0.81	0.042

Table 5: The statistical analysis of the results for three-objective formulation. For `flex` and `gzip`, HNSGA-II still retain the intermediate solutions found in two-objective formulation, resulting in wider Pareto-frontier than the additional greedy algorithm.

Program	Greedy(2Obj)	HNSGA-II(2Obj)	Greedy(3Obj)	HNSGA-II(3Obj)
`flex`	4.383s	19.646s	6.461s	4m 3.980s
`grep`	11.774s	20.068s	18.335s	10m 1.719s
`gzip`	0.481s	1.120s	-	-
`sed`	1.245s	2.776s	1.978s	2m 4.777s
`space`	2.117s	10.892s	3.300s	21m 57.205s

Table 6: Average Execution Time for Algorithms

ecution time was measured using a machine with Intel Core Duo processor running at 2.16GHz with 2GB RAM. The algorithms were implemented in Java with no explicit attempt to optimise it for speed. For two-objective formulation, the execution time of HNSGA-II algorithm remains under 20 seconds for all programs. For three-objective formulation, the additional objective has a significant impact, increasing the execution time up to over 20 minutes.

6.1. Summary of Answers to the Research Questions

RQ1 is positively answered by the solutions found by HNSGA-II for `flex`,`gzip` and `space`. For these programs, HNSGA-II produces solutions that dominate some of the solutions produced by the greedy approach. Particularly for `flex` and `gzip`, HNSGA-II finds solutions that fill the large gap that existed in the Pareto-frontier estimated by the greedy approach. These solutions make it possible to increase coverage with reduced resource require-

ments. The solutions also provide important alternatives in cost-constrained testing environments.

RQ2 and **RQ3** are answered by the statistical analysis shown in Table 4 and Table 5. The Pareto-frontier approximated by the greedy approach is mostly not dominated by the hybrid approach, suggesting that the greedy approach is an efficient approximation technique. However, HNSGA-II complements the greedy approach by finding additional solutions that provide important alternatives. The overhead of using HNSGA-II can vary. For a large software product, the retest-all regression testing approach can be inhibitive and the overhead of using HNSGA-II for more precise test suite minimisation may be justified. For smaller software products, the results of the empirical study suggest that greedy approach may provide a good approximation.

In answering **RQ4**, the shapes of the Pareto-frontiers provide interesting insights into correlations between code coverage, fault coverage and cost. The universal existence of 'elbow points' suggests that there exist points where the balance of trade-offs between coverage and cost changes dramatically. The Pareto-frontiers of three-objective formulation also reveal how widely the faults are located in the program code. This suggests that Pareto analysis is able to provide useful insights into the structure of the solution space.

6.2. Threats to Validity

Threats to internal validity concern the factors that might have affected the multi-objective optimisation techniques used in the paper. One potential concern involves the accuracy of the instrumentation of the subject software, e.g. the correctness of the coverage information. To address this, a widely used and well tested open-source profiler/compiler tool (GNU `gcc` and `gcov`) was used to collect code coverage information. The fault coverage information was extracted from SIR - a well-managed software archive [10]. Precisely determined computational cost was used in place of the physical execution time in order to raise the precision of the cost information using the Valgrind profiling tool [30].

Another potential internal threat comes from the selection, optimisation and parameterisation of the meta-heuristic techniques themselves. No particular algorithm is known to be effective for the multi-objective test suite minimisation problem. However the genetic algorithm used in this paper is known to be effective for a wide range of multi-objective problems [31, 32], and the previous work strongly suggests that a hybrid approach between the

greedy algorithm and the genetic algorithm will be well-suited for the problem. The algorithms used in the present paper can serve as a basis for the future research. Systematic parameterisation of meta-heuristic optimisation lies beyond the scope of this paper. This is a current topic in the optimisation community, which seeks hyper-heuristics [33].

Usually, evaluation of a hybrid algorithm involves a comparison to its original components. In this paper, the elitism in NSGA-II guaranteed that any solution obtained by HNSGA-II will not be dominated by those obtained by NSGA-II and, therefore, a direct comparision to NSGA-II was not performed.

Threats to external validity concern the conditions that limit generalisation from the result. The primary concern for this paper is the representativeness of the subjects that were studied. This threat can be addressed only by additional research using a wider range of software artifacts and optimisation techniques.

7. Conclusion and Future Work

This paper introduced the concept of Pareto efficient multi-objective optimisation to the problem of test suite minimisation. It described the benefits of Pareto efficient multi-objective optimisation, and presented an empirical study that investigated the relative effectiveness of two algorithms for Pareto efficient multi-objective test suite minimisation. The two-objective formulation of the existing test case prioritisation problem, in particular, shows that multi-objective approach can lead to more efficient testing decisions. The empirical results obtained reveal that greedy algorithms (which perform well for single-objective formulations) are not always Pareto efficient in the multi-objective paradigm, motivating the study of meta-heuristic search techniques. Future work will consider a wider range of software artifacts with different meta-heuristic multi-objective optimisation techniques.

References

[1] M. J. Harrold, R. Gupta, M. L. Soffa, A methodology for controlling the size of a test suite, ACM Transactions on Software Engineering and Methodology 2 (3) (1993) 270–285.

[2] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria,

in: Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), IEEE Computer Society Press, 1994, pp. 191–200.

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

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

[5] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, R. S. Roos, Time aware test suite prioritization, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006), ACM Press, 2006, pp. 1–12.

[6] J. Highsmith, A. Cockburn, Agile software development: the business of innovation, Computer 34 (9) (2001) 120–127.

[7] Y. Collette, P. Siarry, Multiobjective Optimization: Principles and Case Studies, Springer, Oxford, UK, 2004.

[8] F. Szidarovsky, M. E. Gershon, L. Dukstein, Techniques for multiobjective decision making in systems management, Elsevier, New York, 1986.

[9] K. Deb, S. Agrawal, A. Pratab, 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 VI Conference, Springer. Lecture Notes in Computer Science No. 1917, Paris, France, 2000, pp. 849–858.

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

[11] H. K. N. Leung, L. White, Insight into regression testing, in: Proceedings of Interntional Conference on Software Maintenance (ICSM 1989), IEEE Computer Society Press, 1989, pp. 60–69.

[12] G. Rothermel, M. J. Harrold, J. Ostrin, C. Hong, An empirical study of the effects of minimization on the fault detection capabilities of test suites, in: Proceedings of International Conference on Software Maintenance (ICSM 1998), IEEE Computer Society Press, 1998, pp. 34–43.

[13] W. E. Wong, J. R. Horgan, S. London, A. P. Mathur, Effect of test set minimization on fault detection effectiveness, Software Practice and Experience 28 (4) (1998) 347–369.

[14] S. S. Yau, Z. Kishimoto, A method for revalidating modified programs in the maintenance phase, in: Proceedings of International Computer Software and Applications Conference (COMPSAC 1987), IEEE Computer Society Press, 1987, pp. 272–277.

[15] G. Rothermel, M. J. Harrold, Analyzing regression test selection techniques, IEEE Transactions on Software Engineering 22 (8) (1996) 529–551.

[16] S. Bates, S. Horwitz, Incremental program testing using program dependence graphs, in: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1993, pp. 384–396.

[17] D. Binkley, Reducing the cost of regression testing by semantics guided test case selection, in: Proceedings of the International Conference on Software Maintenance (ICSM 1995), IEEE Computer Society, 1995, pp. 251–260.

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

[19] S. G. Elbaum, A. G. Malishevsky, G. Rothermel, Prioritizing test cases for regression testing, in: Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2000), ACM Press, 2000, pp. 102–112.

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

[21] H. Do, G. Rothermel, A. Kinneer, Empirical studies of test case prioritization in a junit testing environment, in: Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE 2004), IEEE Computer Society Press, 2004, pp. 113–124.

[22] S. Yoo, M. Harman, Pareto efficient multi-objective test case selection, in: Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2007), ACM Press, 2007, pp. 140–150.

[23] D. Fudenberg, J. Tirole, Game Theory, MIT Press, 1983.

[24] R. Kicinger, T. Arciszewski, Multiobjective evolutionary design of steel structures in tall buildings, in: Proceedings of the AIAA 1st Intelligent Systems Technical Conference, Chicago, IL, September 20-23, 2004, American Institute of Aeronautics and Astronautics Press, Reston, VA, 2004, pp. AIAA 2004–6438.

[25] C. Fonteix, S. Massebeuf, F. Pla, L. N. Kiss, Multicriteria optimization of an emulsion polymerization process, European Journal of Operational Research 153 (2) (2004) 350–359.

[26] S. Obayashi, Multidisciplinary Design Optimization of Aircraft Wing Planform Based on Evolutionary Algorithms, in: Proceedings of the 1998 IEEE International Conference on Systems, Man, and Cybernetics, Vol. 4, IEEE Computer Society Press, La Jolla, California, 1998, pp. 3148–3153.

[27] T. J. Halloran, W. L. Scherlis, High quality and open source software practices, in: Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering, 2002.

[28] M. R. Garey, D. S. Johnson, Computers and Intractability: A guide to the theory of NP-Completeness, W. H. Freeman and Company, 1979.

[29] D. S. Johnson, Approximation algorithms for combinatorial problems, in: Proceedings of the 5th annual ACM Symposium on Theory of Computing (STOC 1973), ACM Press, 1973, pp. 38–49.

[30] N. Nethercote, J. Seward, Valgrind: A program supervision framework, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007), ACM Press, 2007, pp. 89–100.

[31] C. A. Coello Coello, D. A. Van Veldhuizen, G. B. Lamont, Evolutionary Algorithms for Solving Multi-Objective Problems, Kluwer Academic Publishers, New York, 2002.

[32] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms, Wiley, Chichester, UK, 2001.

[33] E.K.Burke, G.Kendall, J.Newall, E.Hart, P.Ross, S.Schulenburg, Hyper-heuristics: An Emerging Direction in Modern Search Technology, Kluwer, 2003, Ch. 16, pp. 457–474.