Multi-Objective Optimisation for Regression Testing

Wei Zheng[a], Robert M. Hierons[b], Miqing Li[b], XiaoHui Liu[b], Veronica Vinciotti[b]

[a]*Northwestern Polytechnical University, China*
[b]*School of Information Systems, Computing and Mathematics, Brunel University*
Uxbridge, Middlesex, UB7 7NU United Kingdom

Abstract

Regression testing is the process of retesting a system after it or its environment has changed. Many techniques aim to find the cheapest subset of the regression test suite that achieves full coverage. More recently, it has been observed that the tester might want to have a range of solutions providing different trade-offs between cost and one or more forms of coverage, this being a multi-objective optimisation problem. This paper further develops the multi-objective agenda by adapting a decomposition-based multi-objective evolutionary algorithm (MOEA/D). Experiments evaluated four approaches: a classic greedy algorithm; non-dominated sorting genetic algorithm II (NSGA-II); MOEA/D with a fixed value for a parameter c; and MOEA/D in which tuning was used to choose the value of c. These used six programs from the SIR repository and one larger program, VoidAuth. In all of the experiments MOEA/D with tuning was the most effective technique. The relative performance of the other techniques varied, although MOEA/D with fixed c outperformed NSGA-II on the larger programs (Space and VoidAuth).

Keywords: Software engineering, regression testing, test suite minimisation, multi-objective search.

1. Introduction

Software testing is one of the main verification and validation methods used in software development. Regression testing occurs whenever the *system under test (SUT)* or its environment changes, with the regression testing process involving the SUT being tested with the current test suite T or some subset of this (possibly with additional test cases for new code). Although there are many tools that reduce the cost of regression testing by re-applying the test suite T, regression testing can be a time consuming and expensive process. This is the case if the test suite is large, there is some manual element in testing, or test execution takes up significant resources. It has been found that there are situations in which regression testing takes weeks to run [7]. In addition, many systems are now developed in a continuous manner, with builds being frequently tested, and in such situations the time taken by regression testing can be a significant issue. There has thus been considerable interest in methods that reduce the cost of regression testing (see, for example, [1, 2, 3, 7, 10, 11, 13, 17, 29, 32, 33, 37, 41]). A good overview of this work can be found in a recent survey [42].

A natural approach, to reducing the cost of regression testing, is to select a subset T' of the test suite T, with methods that do this being called *test suite minimisation* methods [42]. The aim of such methods is to choose an effective but small subset and many approaches have been devised [1, 2, 3, 10, 11, 13, 17, 41]. In the absence of fault information, it is normal to use some form of coverage as a proxy for effectiveness. For example, we might want a small subset T' where the percentage of statements executed in testing (statement coverage) is high. The problem is then usually seen as either maximising coverage for a given test budget or minimising cost for a given coverage, with initial work on this topic devising greedy algorithms (see, for example, [2, 3, 11]).

The previously mentioned approaches fix one property (cost or coverage) and then aim to optimise the other property. It has been observed that instead we might treat this problem as a multi-objective optimisation problem: we attempt to optimise two or more properties (such as cost and coverage) at the same time [10]. In multi-objective optimisation it is normal to place a partial order on solutions by saying that solution

Email address: zhengweizr@gmail.com (Wei Zheng)

x Pareto dominates solution *y* if *x* is superior to *y* for at least one objective and is at least as good as *y* for all objectives. For a given problem, the *Pareto set* is the set of solutions that are not Pareto dominated. This Pareto set provides a range of trade-offs between the properties optimised: the tester can then choose from these solutions. While typically it is not feasible to produce the Pareto set for a problem, optimisation algorithms might return an approximation to the Pareto set. Yoo and Harman implemented a multi-objective approach, applying a greedy algorithm and two versions of *non-dominated sorting genetic algorithm II (NSGA-II)* [41]. The approaches were evaluated on Space and four programs from the Siemens suite. Interestingly, while the versions of NSGA-II outperformed the greedy algorithm on the programs from the Siemens suite, the greedy approach was superior for Space [41]. Harman has also identified many different factors that might be considered in the test suite minimisation problem [10].

Yoo and Harman [41] appear to be the first researchers to apply a multi-objective optimisation approach to the test suite minimisation problem. This was an important step that led to valuable results and insights, but the work (naturally) had a number of limitations. One limitation was that while the multi-objective evolutionary algorithm used (NSGA-II) is known to perform well on many problems, there is the potential to apply more specialised approaches. In the evaluation, the main approach used to compare algorithms operated as follows: take the union of the non-dominated solutions produced by the algorithms, form a new set *P* of non-dominated solutions from these, and for an algorithm *A* determine how many of its solutions are not Pareto dominated by solutions in *P*. The authors also compared the sizes of the sets of non-dominated solutions returned by the algorithms, arguing that an algorithm that returns many solutions provides the tester with more options. While this approach is sensible, the evolutionary optimisation community has developed other approaches. Finally, the authors considered two and three objective problems.

In this paper we aim to significantly develop the multi-objective agenda pioneered by Yoo and Harman in three major ways, with the following contributions. Firstly a 'modern' evolutionary algorithm *decomposition-based multi-objective evolutionary algorithm (MOEA/D)* [43], not previously used, has been utilised to overcome some of the limitations as stated above. Secondly, the *Hypervolume (HV)* [49] of a solution has been used in the evaluation as it has a number of important benefits and is widely used in comparing the performance of multi-objective and many-objective algorithms. Thirdly, we have extended a multi-objective solution as proposed in Yoo and Harman's paper to a many-objective one as this would enable one to consider the testing of other important types of software. It transpired that it was necessary to adapt MOEA/D and in doing so we introduced a parameter c used in the normalisation. Some initial runs were performed and in these a value of $c = 0.3$ was found to be reasonably effective. This led to two versions of MOEA/D being used in the experiments: one in which we used this fixed value of c and one in which the value of c was chosen for each experiment ('variable c'). We also explored problems with two to four objectives; Yoo and Harman considered two and three objective problems. One might expect the nature of the optimisation problem to change as the number of objectives increases since, for example, fewer pairs of solutions will be related (under the relation that one dominates the other). In addition, our seven experimental subjects included an additional real-world program; one that implements a web-service.

Yoo and Harman considered two and three objective problems, but we have chosen several combinations of objectives in this paper. The most basic included two objectives only (cost and statement coverage) and might be seen as corresponding to the development of 'normal' software. The three objective case added in branch coverage. Branch coverage is required in a number of areas including automotive software (see, for example, [39]). For four objectives we used cost, statement coverage, branch coverage, and modified condition/decision coverage (required for safety critical components in avionics software [34]) so this might be seen as corresponding to the testing of critical software. In this way we have moved from a multi-objective optimisation problem to a many-objective one, allowing the testing of additional types of software.

In this paper, we compare four optimisation algorithms: a greedy algorithm, NSGA-II, and MOEA/D (with a fixed value of c or variable c). We used five smaller experimental subjects plus Space from the software-artifact infrastructure repository (SIR) and one larger real world system, VoidAuth, that implements the main processes of the Universal Payment Gateway (UPG) service. In all cases the experiments found MOEA/D with variable c to be the most effective technique, with the differences being statistically significant. The relative performance of the other methods varied, although the greedy algorithm was worst for the smaller SIR programs. Similar to Yoo and Harman, we found that the greedy algorithm outperformed NSGA-II on Space but, interestingly, both variants of MOEA/D performed better than the greedy algo-

rithm. For VoidAuth the greedy algorithm outperformed NSGA-II for the two and three objective problems and outperformed MOEA/D with fixed c for the three objective problem. The relative performance of NSGA-II and MOEA/D with fixed c varied for the SIR programs, with neither being consistently better. However, in all cases MOEA/D with fixed c outperformed NSGA-II for VoidAuth. We also considered a specific scenario, where the tester decides to use a test suite that provides full coverage for every metric in the optimisation process. In every experiment MOEA/D with variable c returned the smallest test suite.

The paper is structured as follows. Section 2 provides an overview of previous work on test suite minimisation for software regression testing and Section 3 reviews approaches to evolutionary multi-objective optimisation. Section 4 explains how we used evolutionary multi-objective optimisation algorithms to address the test suite minimisation problem. Section 5 explains the experimental design and Section 6 analyses the results of the experiments. Section 7 explores threats to validity and, finally, Section 8 draws conclusion and discusses possible lines of future work.

2. Related Work

The testing carried out during software development is often divided into phases such as unit testing and system testing. However, the delivery of a system does not mark the end of testing: there is a need to re-test the system whenever it is changed or its environment changes. This process is called regression testing and typically the original test suite T produced during development is re-run, with this process often being automated. If features are added or removed or the interface of the SUT is changed then it may be necessary to fix test cases from T, add new test cases, and remove those that are no longer valid.

As previously noted, while it is often possible to automatically re-run a test suite T, this process can still be expensive. The cost of regression testing has led to the development of methods that increase the efficiency of regression testing, either through *prioritising* test cases (in the hope of finding any faults early in regression testing) [7, 29, 33, 37] or by *selecting* some subset of the regression test suite to use [1, 2, 3, 10, 11, 13, 17, 41]. In this section we focus on multi-objective approaches. A recent survey provides an excellent general overview of the work that aims to reduce the cost of regression testing [42].

The aim of prioritisation techniques is to order (prioritise) the test cases so that it is likely that faults are found early in regression testing. This is motivated by the desire to reduce the cost and time taken by regression testing: if faults are found in regression testing then the earlier these are found the sooner they can be fixed, potentially reducing the time to market.

A major issue when considering prioritisation is that we do not know whether the SUT will pass the regression test suite T and, if it does not, which test cases will lead to failure. Thus, we cannot know whether one ordering of the test cases is better than another. As a result, the focus has been on the use of a proxy, with coverage being the main proxy. This is based on the assumption that a test suite that achieves coverage quickly is likely to find faults earlier in testing. An alternative is to use historical fault data, prioritising test cases that execute parts of the code that have previously been found to be faulty.

Early work on prioritisation used greedy algorithms. Let us suppose, for example, that we are to prioritise on the basis of statement coverage. Then the test cases would be ranked on the basis of the number of program statements, with this providing the ordering (see, for example, [33]). Now let us suppose that we have three test cases t_1, t_2, t_3, test case t_1 covers statements s_1, s_2, s_4, t_2 covers s_1, s_3, and t_3 covers s_1, s_2, s_4. If we prioritise on statement coverage, as described above, then we might choose either order t_1, t_3, t_2 or order t_3, t_1, t_2 and in both cases the second test case does not increase the total coverage. If, instead, we choose an ordering such as t_1, t_2, t_3 then 100% statement coverage would be achieved using just the first two test cases. The problem is that approaches such as that described above optimise on the total statement coverage of test cases, not the additional coverage provided. This led to the development of additional greedy algorithms in which, at each stage in the process, we add the test case that maximally increases the total statement coverage [33]. Note that although prioritisation techniques typically use the source code, they have also been developed for binary code [36].

The focus of this paper is on test suite minimisation, in which we choose a subset of a regression test suite T, with the aim of reducing the cost of regression testing and ideally not reducing its effectiveness. This differs slightly from a third problem, called test case selection, in which we also use information regarding the new version of the SUT. Test case selection thus aims to test the changed parts of the SUT and to minimise the costs of doing so. More information about test case selection can be found in the survey by Yoo and Harman [42].

Since we cannot know in advance the effectiveness of a test suite, it is instead normal to describe test suite

minimisation in terms of a set of test requirements [11]. The requirements discussed in the literature typically correspond to coverage of parts of the SUT. For example, if the SUT has statements $s_1, \ldots, s_n$ then we might have n requirements $r_1, \ldots, r_n$, where requirement r_i is that statement s_i is executed during testing. If we have test suite T and set R of requirements that are satisfied by T, we might then aim to find the smallest subset of T that satisfies the requirements in R [11].

Given a test case t we can find the subset R_t of requirements from R that are satisfied by t. The test suite minimisation problem can then be seen as that of choosing the smallest set of R_t that, between them, include all elements of R. This is similar to the set cover problem, in which we have a set S, a set $S' = \{S_1, \ldots, S_k\}$ of subsets of S, and we wish to find the smallest set $S'' \subseteq S'$ whose union is S. It is not hard to show that any instance of the set cover problem can be represented as a test suite minimisation problem. Thus, since the set cover problem is NP-hard [22], we know that the test suite minimisation problem is NP-hard. This has led to interest in the development of heuristics for the test suite minimisation problem with much of the initial focus being on greedy algorithms [2, 3, 11]. It has also been noted that sometimes the tester is interested in two or more types of coverage. Jeffrey and Gupta [17] approached this situation by applying a greedy algorithm in which a test case is only deemed to be redundant if it is redundant with respect to all of the types of coverage. A number of authors have also represented the test suite minimisation problem as an Integer Linear Programming problem [1, 13]. If there is only one objective then the function being optimised is the coverage of requirements. Where there are multiple objectives, such as different types of coverage, a (possibly weighted) sum is used.

The test suite minimisation problem has typically been represented as that of finding the smallest (or cheapest) subset of test suite T that satisfies all of the requirements or that maximises some weighted sum of scores. However, in practice the tester might be willing to use a test suite that does not satisfy all of the requirements if this test suite is sufficiently small. Thus, it is natural to see the problem as a multi-objective optimisation problem. Yoo and Harman applied this approach, using two objectives (test execution cost and statement coverage) and also three objectives (test execution cost; statement coverage; and fault history) [41]. This work used a greedy algorithm and two versions of NSGA-II. The approaches were evaluated on part of the Siemens suite of programs (printtokens, printtokens2, schedule, schedule2) and Space and the results were mixed, with the differences being relatively small. In particular, while the variants of NSGA-II outperformed the greedy algorithm on the programs from the Siemens suite, the greedy approach was superior for the larger program (Space) [41]. This aim of this paper is to further develop the multi-objective agenda through using a more recently developed evolutionary algorithm MOEA/D, using Hypervolume values in the evaluation, and using a new real-world experimental subject.

3. Evolutionary Multi-Objective Optimisation

Multi-objective optimisation involves simultaneous optimisation of two or more conflicting objectives. In multi-objective optimisation problems (MOPs), there is usually no single optimal solution but rather a set of Pareto optimal solutions (called the Pareto set in the objective space). An MOP, without loss of generality, can be defined as follows:

$$\begin{array}{ll} \text{Minimise} & \boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_m(\boldsymbol{x})) \\ \text{Subject to} & \boldsymbol{x} \in \boldsymbol{\Omega} \end{array} \quad (1)$$

where $\boldsymbol{x}$ denotes a solution vector in the feasible solution space $\boldsymbol{\Omega}$, and f_i $(i = 1, 2, ..., m)$ is the ith objective to be minimised.

The concept of optimality in multi-objective optimisation is defined by the Pareto dominance relation. Given two decision vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, $\boldsymbol{x}$ is said to *Pareto dominate* (or *dominate*) $\boldsymbol{y}$ (denoted $\boldsymbol{x} \prec \boldsymbol{y}$), if and only if $\boldsymbol{x}$ is at least as good as $\boldsymbol{y}$ in all objectives and better in at least one objective. Accordingly, those decision vectors that are not dominated by any other vector are denoted as *Pareto optimal solutions*. In general, the set of Pareto optimal solutions in the decision space is called the *Pareto set*, and the set of the corresponding objective vectors the *Pareto front*. Unfortunately, it is often infeasible to obtain the whole Pareto set of an MOP, and we only hope to find a good approximation to the set. Usually, we consider the *non-dominated set* of the obtained solutions as the approximation.

An MOP with more than three objectives is often called a many-objective optimisation problem [8, 16, 30]. Many-objective optimisation is an important but challenging topic since the Pareto dominance-based criterion generally loses its effectiveness in dealing with this kind of problems. For the Pareto dominance criterion, the portion of any two solutions being comparable in the objective space decreases exponentially with the increase of the number of objectives. This leads to the appearance of new dominance criteria [23, 35, 40], which loosen the dominance condition in comparing two solutions.

Evolutionary algorithms (EAs) are a class of stochastic optimisation methods that simulate the process of natural evolution. In recent years, there has been significant interest in the use of EAs to solve MOPs, with success of various EAs, such as genetic algorithms, particle swarm optimisation, ant colony optimisation, and artificial bee colony, in diverse application domains [4, 18, 19, 20, 21, 25, 46]. There are two main advantages of EAs in the context of MOPs (usually called EMO algorithms). One is that they have low requirements on the problem characteristics, and the considered objectives can be easily added, removed, or modified. The other is that their population-based search can achieve an approximation of the problem's Pareto front, with each solution representing a unique trade-off amongst the objectives.

Over the past two decades, a number of effective EMO algorithms have been proposed, such as the non-dominated sorting genetic algorithm II (NSGA-II) [6], strength Pareto evolutionary algorithm 2 (SPEA2) [48], indicator-based evolutionary algorithm (IBEA) [47], and decomposition-based multi-objective evolutionary algorithm (MOEA/D) [43]. These algorithms, based on their selection mechanism, can be divided into two groups: Pareto-based algorithms and non-Pareto-based algorithms.

As their name suggests, Pareto-based algorithms compare individuals using the Pareto dominance relation, which reflects individuals' behaviour in terms of convergence. When Pareto dominance fails (e.g., the individuals concerned are non-dominated with respect to each other), these algorithms often introduce individuals' density information to distinguish between them, serving the purpose of maintaining the diversity of the evolutionary population. Many popular EMO algorithms belong to this group. NSGA-II and SPEA2 are two representative examples.

Recently, there has been increasing interest in the use of EMO algorithms that do not consider Pareto dominance as a selection criterion, with IBEA and MOEA/D being important examples of such algorithms. The former defines an optimisation goal with respect to a specified performance indicator and uses this goal to guide the search of the population; the latter decomposes an MOP into a number of scalar subproblems and then optimises them with the aid of the information from their neighbours. Non-Pareto-based algorithms have been found to be promising in many challenging MOPs [24, 44], especially in a problem with a high-dimensional objective space where the Pareto dominance relation fails to provide sufficient selection pressure towards the problem's Pareto front [9, 16, 26].

We now explain how the test suite minimisation problem can be expressed as a multi-objective optimisation problem to which the above approaches can be applied.

4. Regression Testing as a Multi-Objective Optimisation problem

4.1. Introduction

Test suite minimisation techniques find one or more subsets of the regression test suite T with the aim of providing a good trade-off between the various objectives. In this section we explain how we adapted three algorithms to the multi-objective test suite minimisation problem.

4.2. The objectives used

We chose several combinations of objectives (two, three or four objectives). Since the primary aim of test suite minimisation is to reduce the cost of regression testing, all combinations included cost as an objective. For cost we adopted the approach of Yoo and Harman [42], which is to use a tool (Valgrind) to estimate the execution time for a test case t and to use this time as the cost of executing t; we say more about this in Section 5.

The three additional objectives all corresponded to standard coverage metrics. The most basic of these was statement coverage: the percentage of program statements covered in testing. The two-objective case thus used cost and statement coverage. The three-objective case added in the branch coverage of the test suite: the percentage of program branches executed in testing. Finally, the four-objective case used cost, statement coverage, branch coverage, and Modified Condition/Decision Coverage (MC/DC) coverage. To satisfy the MC/DC coverage criterion, all of the below must be achieved at least once during testing.

1. Each decision takes on the values true and false during testing.
2. Each condition in a decision takes on the values true and false during testing.
3. Each entry and exit point is invoked.
4. Each condition in a decision is shown to independently influence the outcome of the decision.

If a test suite has the fourth property then it must also have the first two properties. In addition, all subject programs used in this work had only one entry and one exit. As a result, it was sufficient to focus on the fourth property, which we now explain in greater detail.

In a program a decision is a predicate whose evaluation determines the next piece of code to be executed.

Thus, for example, a while loop has a decision d and the result of evaluating d determines whether the program enters the body of the loop. A decision d in a program may be composed of multiple conditions. For example, the following has three conditions.

```
if (x>0 || x>z) && y>x then
```

For a condition *cond* in a decision d to be shown to independently influence the outcome of the decision we require two tests that produce different outcomes for d (one evaluates to true, the other to false) and where the values of *cond* differs but all other conditions take the same value. For the above example and the first condition $x > 0$ we might, for example, have two test cases such as the following.

1. A test case t where d is executed when we have that $x = 10$, $y = 25$, and $z = 20$. Here the decision evaluates to true, the value of the first condition is true, the value of the second condition is false, and the value of the third condition is true.
2. A test case t' where d is executed when we have that $x = 0$, $y = 5$, and $z = 2$. Here the decision evaluates to false, the value of the first condition is false, the value of the second condition is false, and the value of the third condition is true.

Between them, these satisfy the MC/DC requirements with regards to the first condition since the decision takes different values but the values of the second and third conditions do not change.

Statement coverage, branch coverage, and MC/DC coverage are standard well-known coverage criteria. Statement coverage is often seen as being a basic minimum, while branch coverage is mandated in a number of areas, including automotive software (see, for example, [39]). MC/DC coverage is often seen as being desirable for safety-critical software and is mandated for safety-critical avionics software [34]. The aim therefore was for these combinations of objectives to correspond to different classes of software of varying criticality, from 'normal' software (two objectives) through to safety-critical software (four objectives).

It may be observed that these forms of coverage are related under the subsumptions relation: if a test suite achieves 100% MC/DC coverage then it is guaranteed to achieve 100% branch coverage, and if a test suite achieves 100% branch coverage then it is guaranteed to achieve 100% statement coverage. However, the subsumption relation concerns full coverage and so, for example, if $1 \leq x < 100$ then x% branch coverage does not imply x% statement coverage. To demonstrate this it is sufficient to use a program p that starts with an `if` statement with `then` case S_1 and `else` case S_2 and construct S_1 so that it has many statement and no branches and construct S_2 so that it has many branches and relatively few statements. We can then produce a test suite T that executes S_2 only, covering all but one of the branches but missing many statements. Thus, these three forms of coverage are competing despite being related under subsumption.

In order to represent test suite minimisation in the usual way (in terms of a set of requirements), for a coverage metric such as statement coverage we needed a set of items that have to be covered in testing. Once we have such a representation, we can find the set of requirements (items that need to be covered) that are covered when using a test case t_i. For statement coverage, this set is simply the set of program statements executed when testing with t_i and for branch coverage it is the set of branches. For MC/DC things are rather more complicated, since we require a pair of test cases to satisfy the fourth condition above and there is potential for there to be different ways in which we can show a condition independently influencing the outcome of a decision. However, we found a relatively straightforward way of overcoming this issue. Consider, first, the case where we have a simple conjunction or disjunction of two conditions A and B. Then we require the following.

1. For conjunction ($A\&\&B$) we require one test case where both A and B are true; one where A is true and B is false; and one where A is false and B is true.
2. For disjunction ($A||B$) we require one test case where both are false; one where A is true and B is false; and one where A is false and B is true.

For each of these cases we have three corresponding test requirements.

We found that most decisions in our subject programs either had only one condition or were in one of the above forms. We were able to remove the few decisions not of one of these forms by applying simple transformations. Let us suppose, for example, that we have a decision of the form.

```
if (A || B) && (C || D) then
```

This would be rewritten to the following.

```
if (A || B) then if (C || D)
```

Having applied such simple transformations, each decision that is a conjunction ($A\&\&B$) or disjunction ($A||B$) defines three requirements and each decision that

has only one condition defines two requirements (as with branch coverage). For the experimental subjects, 10 decisions had to be rewritten for gzip while for the other programs we required only one or two rewrites.

Let us suppose that we have test suite $T = \{t_1, \ldots, t_n\}$ for program p and for a test case t_i, $1 \leq i \leq n$, we have that: $\tau(t_i)$ is the (estimated) execution time of t_i; $s(t_i)$ is the set of statements of p covered by t_i; $b(t_i)$ is the set of branches of p covered by t_i; and $m(t_i)$ is the set of MC/DC coverage items that are covered by t_i. Then, given $T' \subseteq T$, the four metrics used to drive optimisation were defined as follows.

1. Total execution time:
$$\tau(T') = \sum_{t_i \in T'} \tau(t_i) \tag{2}$$
2. Total statement coverage:
$$s(T') = \frac{|\cup_{t_i \in T'} s(t_i)|}{|\cup_{t_i \in T} s(t_i)|} \tag{3}$$
3. Total branch coverage:
$$b(T') = \frac{|\cup_{t_i \in T'} b(t_i)|}{|\cup_{t_i \in T} b(t_i)|} \tag{4}$$
4. Total MC/DC coverage
$$m(T') = \frac{|\cup_{t_i \in T'} m(t_i)|}{|\cup_{t_i \in T} m(t_i)|} \tag{5}$$

Naturally, we wish to minimise the first measure and maximise the others.

4.3. Using a Greedy algorithm

It is straightforward to develop a greedy algorithm for the two-objective problem. For example, if we wish to find a subset of T that achieves the same statement coverage as T then the greedy algorithm starts with the empty set and at each iteration it adds the test case that covers the most (so far uncovered) statements. The algorithm terminates once the subset covers all of the statements covered by T. We implemented a greedy algorithm for the general case, using the approach of Yoo and Harman [42], that operates as follows: at each iteration, if we have test suite T' then we choose the test case t_i that maximises the following: the sum, over the coverage objectives being considered, of the improvements in coverage, with this sum being divided by the cost of executing t_i. For example, for the three objective case the greedy algorithm chooses the test case t_i that maximises the following.

$$\frac{(s(T' \cup \{t_i\}) - s(T')) + (b(T' \cup \{t_i\}) - b(T'))}{\tau(t_i)} \tag{6}$$

The algorithm iterates until it obtains a test suite that has the same coverage as T (for the forms of coverage being considered).

4.4. Using NSGA-II

NSGA-II [6] is one of the most popular algorithms in the EMO field. Its remarkable characteristics, such as low computing complexity, independence of badly-scaled problems, the usage of elitism, and parameterless search, make it well-suited to a significant number of applications in real-world scenarios.

In order to guide the search in the evolutionary process, NSGA-II introduces two effective selection criteria: Pareto non-dominated sorting and crowding distance. The Pareto non-dominated sorting divides the individuals in a population into a number of fronts (ranks) according to their Pareto dominance relation. For a population, rank 1 corresponds to the non-dominated individuals with respect to the whole individual set, and rank 2 corresponds to the non-dominated individuals with respect to the set formed by removing the rank-1 individuals, and so on. The crowding distance criterion in the algorithm is used to estimate the density of individuals in a population. An individual's crowding distance is defined as the distance of its two neighbours on either side of the individual along each of the objectives. NSGA-II then defines a partial order relation using the two criteria, and prefers 1) individuals with lower rank and 2) individuals with larger crowding distance when they have the same rank.

It is possible to apply NSGA-II directly to our problems: it is sufficient to connect an implementation of this algorithm with functions that compute the cost and coverage scores of a test suite. There is no need for an additional normalisation process since the Pareto dominance relation is independent of the scaling of the objectives and also the calculation of the crowding distance is based on individuals' objective value that is already normalised according to the boundary of the current population.

4.5. Using MOEA/D

As a representative algorithm developed recently, MOEA/D [43] is based on conventional aggregation approaches from mathematical programming. MOEA/D decomposes an MOP into a number of scalar optimisation subproblems and deals with them simultaneously. Neighbourhood relations among these subproblems are defined based on the Euclidean distance between their aggregation weight vectors. When optimising a subproblem, the information from its neighbouring subproblems is adopted.

In MOEA/D, each subproblem keeps one individual in its memory, which could be the best individual found

so far for the subproblem. For each subproblem, the algorithm generates a new individual by performing variation operators on some of its neighbouring individuals (i.e., the individuals of its neighbouring subproblems). The memory of both the considered subproblem and its neighbouring subproblems will be updated if the new individual is better than their current one.

In MOEA/D each individual in the population has a unique weight vector, and all these weight vectors are controlled by a parameter h. More precisely, for a weight vector, each weight member takes a value from $\{0/h, 1/h, ..., h/h\}$, and also it holds that the sum of all the members of the weight vector is equal to 1.

MOEA/D has many good characteristics, including a well-organised memory mechanism, the utility of the neighbouring information in variation, and the non-Pareto-based criterion in environmental selection. It has been found that these become particularly beneficial when dealing with some challenging multi-objective optimisation problems (MOPs) [24, 28], like variable-linkage problems [45], many-objective problems [31], and combinatorial problems [14].

We initially implemented MOEA/D in a similar way to NSGA-II: we integrated an implementation of MOEA/D with functions that return the cost and coverage scores of a test suite and chose values for the parameters. Here, h is set to make the size of the population of MOEA/D be approximately the same as that used with NSGA-II. We performed some initial trials without any normalisation like in NSGA-II, and found that the performance was poor. This occurrence can be attributed to the fact that the range of values of test suite cost is much larger than that for the other objectives (coverage), which makes the converted scalar optimisation subproblems pay less attention to the coverage objective(s) during the evolutionary process. We therefore normalised the cost of a test suite. This normalisation was based on the maximum possible test suite cost *max* and the minimum test suite cost *min*. Thus, *min* was set to zero (the empty test suite) and *max* was set to the sum of the costs of the test cases. For each experiment we also introduced a parameter c: given a test suite T', we set the normalised overall cost of T' to be the following.

$$\frac{(\sum_{t_i \in T'} \tau(t_i)) - min}{c \times (max - min)} \tag{7}$$

It could be argued that this leads to a version of MOEA/D using a different fitness function and thus the need to evaluate both NSGA-II and the greedy algorithm using this alternative fitness function. However, the above introduces a constant scaling and so does not affect whether one candidate solution dominates another. Since NSGA-II and the greedy algorithm operate on the basis of dominance, the normalisation will not affect the performance of these approaches and so there is no need to separately evaluate NSGA-II and the greedy algorithm using the normalised fitness value.

Later we report on the results of experiments that investigate two approaches, the first using a fixed value of c suggested by initial experiments. In the second approach we allowed the value of c for an experiment (i.e. for one subject and number of objectives) to be based on some initial tuning and thus for the value of c to vary between experiments. The aim was for the first more restrictive scenario, in which the value of c is fixed, to explore the potential performance in the situation where a software engineer does not wish to tune the algorithm for a particular problem. The aim of the second set of experiments was to explore the potential for improvement provided by tuning.

5. Experimental Design

5.1. Introduction

In this section we explain how the experiments, that compared the alternative optimisation approaches, were carried out. We initially describe the experimental subjects, then give the values of the parameters used, and finally outline the tools used in the experiments.

5.2. Experimental subjects

The experiments used seven subjects, with six being obtained from the Software-artifact Infrastructure Repository (SIR)[1] along with test suites. We used five smaller SIR subjects, which were: two versions of gzip (a Compression tool that is a GNU open-source program); Schedule (a priority scheduler); tcas (an aircraft collision avoidance system); and tot_info (that takes input data and computes some statistics). We also used Space for the two objective problem since it is larger than the other SIR subjects used by Yoo and Harman and Yoo and Harman obtained quite different results with Space (but very similar results with their other experimental subjects). The seventh, larger, program was the VoidAuth process within the Universal Payment Gateway (UPG) service. This process is responsible for cancelling an approved authorisation when a customer has decided not to continue with a payment. This piece of software implements a number of features, including the following:

[1] At `sir.unl.edu/`

1. Checking that the data in an input request is valid.
2. Choosing the correct gateway for an input request.
3. Converting an input request into the gateway request format.
4. Sending a gateway request to the corresponding gateway service.
5. Accepting a response message from a gateway service and sending this to the client.

The VoidAuth process module also contains about 10,000 lines of '.cs' code.

The test cases for VoidAuth were manually generated and these are publicly available along with the code[2]. Table 1 provides details regarding the experimental subjects (name; lines of code; and test suite size).

name	lines of code	test suite size
gzip-v3	7259	214
gzip-v4	7329	214
Schedule-v2	413	2650
tcas-v1	174	1608
tot_info-v1	407	1052
Space	6199	300
VoidAuth	12600	530

Table 1: Experimental Subjects

5.3. Parameters

The greedy algorithm is deterministic so it was run once for each experiment (combination of subject and set of objectives). Since NSGA-II and MOEA/D are stochastic they were executed 30 times for each experiment to account for their inherent randomness. The termination criterion of the two algorithms was a predefined number of evaluations, $100,000$ for all 2-, 3- and 4-objective instances.

In NSGA-II and MOEA/D, all individuals were coded in a binary string format. An individual is a test suite, which can be represented by a 'choice' for each test case. A test case t_i was denoted as a bit (0 or 1) in the string, with 1 indicating t_i being chosen. Two commonly-used crossover and mutation operations, uniform crossover and bit-flip mutation, were used. A crossover probability $p_c = 1.0$ and a mutation probability $p_m = 1/n$ (where n is the number of decision variables) were set according to [5].

The size of the population in NSGA-II was set to 100. Note that the size of the population in MOEA/D is the same as the number of weight vectors, which depends on the parameter h. Due to the combinatorial nature of uniformly distributed weight vectors, the population size cannot be arbitrarily specified. Here, we set h to make the population size of MOEA/D the closest integer to 100 among the possible values (i.e., 100, 105, and 120 for 2-, 3-, and 4-objective problems, respectively). The above configurations of NSGA-II and MOEA/D are summarised in Table 2 and Table 3, respectively. Note that while we could not use exactly the same population size for the three and four objective problems, we did give all of the stochastic methods the same number of evaluations of the fitness function.

objective number	population size	evaluations
2	100	$100,000$
3	100	$100,000$
4	100	$100,000$

Table 2: Parameters for NSGA-II

objective number	h	population size	evaluations
2	99	100	$100,000$
3	13	105	$100,000$
4	7	120	$100,000$

Table 3: Parameters for MOEA/D

Finally, the implementation of MOEA/D for the considered problem also required the parameter c. In a first set of experiments we used a fixed value of $c = 0.3$ chosen on the basis of an initial tuning process. We then had a separate tuning process for each experiment and used the values given in Table 4, which gives the value of c used for each number of objectives.

program	Two Objectives	Three Objectives	Four Objectives
gzip-v3	0.61	0.28	0.19
gzip-v4	0.30	0.46	0.19
schedule-v2	0.26	0.10	0.18
tcas-v1	0.30	0.20	0.30
tot_info-v1	0.25	0.43	0.30
VoidAuth	0.40	0.20	0.60
Space	0.30		

Table 4: Value of c for MOEA/D

5.4. Tool support

After running a test case t_i, a tool would report the items covered, with this being represented by a vector

[2]These can be found at `http://pan.baidu.com/s/1mgBNEVQ`

$\langle r_1, \ldots, r_m \rangle$ in which $r_j = 1$ if t_i covered the jth item and otherwise $r_j = 0$. For each form of coverage, the coverage provided by a test case was therefore represented by a vector of 0s and 1s.

The statement coverage information for the SIR programs was derived using the GNU compiler, gcc, and its profiling tool, gcov. The instrumentation was performed by gcc during compilation. An instrumented subject program produces execution-trace information for each execution, which is converted to statement level coverage information using gcov.

For the SIR programs, we wrote a tool that instruments the code to provide branch coverage and MC/DC coverage information. For branch coverage, the instrumented code simply reported the result of a decision executed. For MC/DC coverage, instrumentation also included instrumenting conditions within any decision that had more than one condition.

While the SIR programs were written in C, VoidAuth was written in C#. We obtained test execution costs for VoidAuth by using Microsoft Visual Studio 2010, and instrumenting the program to provide information regarding statement coverage, branch coverage and MC/DC coverage.

Finally, in order to estimate execution cost we used a GPL system, Valgrind, that is designed for debugging and profiling Linux programs. We used the Valgrind profiling tool Callgrind, which creates a log that contains information about the execution of the SUT. The execution time (and so cost) was given by the Timerange field from the log.

5.5. Performance Comparison Metric

It is possible to visually compare the performance of alternative methods by plotting the points on a graph. However, this approach becomes more difficult as the number of objectives increases and does not provide any statistical evidence. We therefore used the Hypervolume (HV) [49] of a solution, a measure that is commonly used in multi-objective optimisation. HV calculates the volume of the objective space between the obtained solution set and a reference point, and a larger value is preferable. HV has three main benefits as an evaluation mechanism. First, HV has some good theoretical properties and, in particular, is Pareto compliant: if a solution set A Pareto dominates a solution set B (for any solution in B, there always exists a solution in A that Pareto dominates it), then A will obtain a better HV value than B. Second, the HV result of a solution set reflects both its convergence to the problem's Pareto set and the diversity of the information: both properties considered by Yoo and Harman are captured by one measure. In addition, unlike some performance metrics like Generational Distance (GD) [38] and Inverted Generational Distance (IGD) [43], it is possible to use HV without knowing the considered problem's Pareto set. This is an important property for real-world problems, where we typically do not know the Pareto set. For clarity, we provide a normalised HV value of each algorithm with respect to the proportion of the optimal HV result achieved. This normalisation makes all of the obtained results within the range $[0, 1]$, with 1 representing the optimal value. Since the optimal HV of the problems cannot be obtained by calculation, we, following the practice in [27], estimated the optimal value by the result of the mixed set consisting of all the obtained solutions on a given problem.

In the calculation of HV, two crucial issues are the scaling of the objective space and the choice of the reference point. Since the objectives in the considered problems take different ranges of values, we standardise the objective value of the obtained solutions according to the range of the approximation of the Pareto front. The Pareto front approximation is the nondominated set of all the obtained solutions. Following the recommendation in [15], the reference point was set to 1.1 times the upper bound of the Pareto front approximation to emphasise the balance between proximity and diversity of the tested algorithms.

6. Results and Discussion

6.1. Introduction

As previously explained, the stochastic methods were each run 30 times. We performed some initial experiments and found that the value $c = 0.3$ (for MOEA/D) appeared to work well for many of the subjects and then performed experiments for both MOEA/D with this value of c and MOEA/D with the value of c varying as shown in Table 4. We first describe the HV results obtained with the smaller SIR programs, then the results with Space and finally the results with the (larger) Void Auth subject. For each subject, we performed all six pairwise comparisons across the four methods and report the p-values of a Welch two-sample t-test for the difference in the average HVs between the methods. The same conclusions were obtained using a non-parametric Wilcoxon test due to the low variability in the HV values across the 30 runs. The p-values were adjusted using the Holm-Bonferroni method [12] to take into account the number of tests performed. Finally, we explore the impact of these differences on test suite size.

6.2. The smaller SIR Programs

Table 5 gives the mean normalised HV for the two-objective methods on the SIR programs, averaged across 30 runs for the stochastic methods. The results show that NSGA-II and MOEA/D are significantly more effective than the greedy algorithm in all cases considered (p-values ≈ 0). For tcas-v1 the results are identical for all of the stochastic algorithms as a result of the problem being small: all runs of these algorithms returned the same set of solutions. For the other subjects, we find that MOEA/D with fixed c is significantly more effective than NSGA-II on gzip-v4 and tot_info-v1, but significantly less effective on gzip-v3 and schedule-v2 (p-values ≈ 0). In contrast, MOEA/D with varying c has a higher mean HV value than NSGA-II for all subjects: the comparisons are all highly significant (p-value ≈ 0) with the exception of schedule-v2, where the difference is not statistically significant (p-value 0.21). For a visual understanding of the solutions' distribution, Figure 1 plots the final solutions of the greedy algorithm, NSGA-II and MOEA/D ($c = 0.3$) for one run on the gzip-v4 problem.

	Greedy	NSGA-II	MOEA/D fixed c	MOEA/D varying c
gzip-v3	0.968756	0.985749	0.982132	0.986071
gzip-v4	0.984493	0.991833	0.996564	0.996564
schedule-v2	0.905562	0.999606	0.986347	1.000000
tcas-v1	0.949391	0.991906	0.991906	0.991906
tot_info-v1	0.929316	0.999152	0.999732	1.000000

Table 5: Normalised HV for SIR Programs and Two Objectives

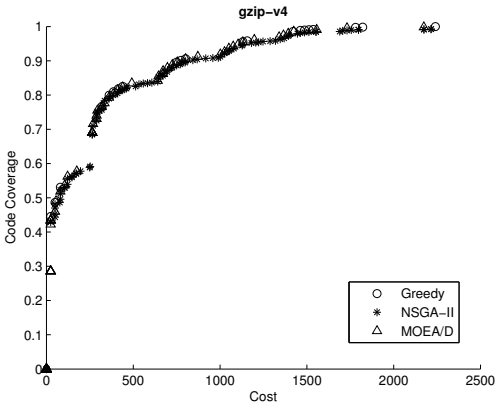

Figure 1: Solution sets obtained by the greedy algorithm, NSGA-II, and MOEA/D ($c = 0.3$) on the 2-objective gzip-v4 problem.

Table 6 shows similar results for three-objective methods. Greedy always has the lowest HV (p-values ≈ 0 for all comparisons). Also, there is no significant difference between NSGA-II and MOEA/D (fixed and varying c) on tcas-v1 (p-values larger than 0.3 for all comparisons). For the other subjects the relative performance of NSGA-II and MOEA/D with fixed c varies, whereas MOEA/D with varying c is more efficient than NSGA-II in all cases (p-values ≈ 0), with the exception of tot_inf-v1, where the difference is not significant (p-value 0.95).

	Greedy	NSGA-II	MOEA/D fixed c	MOEA/D varying c
gzip-v3	0.96796	0.98191	0.99118	0.989303
gzip-v4	0.96837	0.99621	0.98778	0.990970
schedule-v2	0.85269	0.95825	0.95865	0.995087
tcas-v1	0.97858	0.99988	0.99991	0.999929
tot_info-v1	0.89965	0.99915	0.99384	0.999157

Table 6: Normalised HV for SIR Programs and Three Objectives

Table 7 shows the results for four-objectives. The greedy algorithm always has the lowest HV (p-values ≈ 0 for all comparisons), and MOEA/D (with fixed and varying c) and NSGA-II have the same efficiency on tcas-v1, tot_info-v1 and Voidauth. For the remaining three cases, MOEA/D with fixed c is always less effective than both NSGA-II and MOEA/D with varying c (p-values ≈ 0); MOEA/D with varying c always has the highest mean HV, but the difference with NSGA-II is not significant for schedule-v2 (p-value 0.85), whereas it is significant in the other two cases (p-value 0.0071 for gzip-v3 and 0.018 for gzip-v4).

	Greedy	NSGA-II	MOEA/D fixed c	MOEA/D varying c
gzip-v3	0.958261	0.978181	0.963971	0.980888
gzip-v4	0.960178	0.978630	0.967824	0.980589
schedule-v2	0.615664	0.995886	0.911465	0.996157
tcas-v1	0.983472	0.999989	0.999989	0.999989
tot_info-v1	0.914306	0.982623	0.982623	0.982623

Table 7: Normalised HV for SIR Programs and Four Objectives

For all 15 experiments reported above we found that the greedy algorithm was worst and MOEA/D with varying c was always best (though in a few cases the result was not statistically significant). The relatively performance of NSGA-II and MOEA/D with fixed c varied. There appear to be no strong patterns regarding the relative performance of NSGA-II and MOEA/D with fixed c, although in most cases MOEA/D was best with three objectives (all but tot_info-v1) and in all case NSGA-II was best for four objectives.

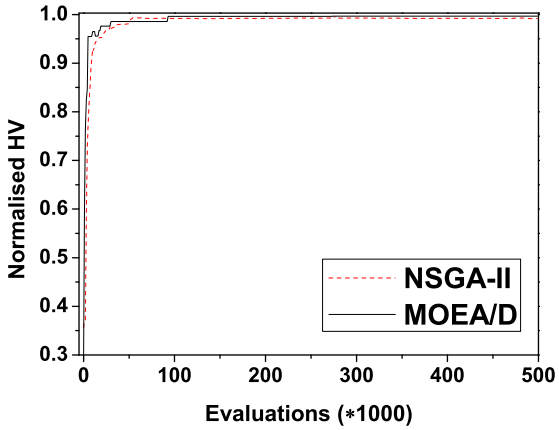

Figure 2: Evolutionary trajectories of the normalised HV for NSGA-II and MOEA/D ($c = 0.3$) on the 2-objective gzip-v4 problem.

In order to investigate how the performance changes over the course of the optimisation, Figure 2 plots the normalised HV trajectories during 500,000 evaluations for a typical run of NSGA-II and MOEA/D ($c = 0.3$) on the 2-objective gzip-v4 problem. Similar results have also been observed on the other problems. As seen, the HV trajectory of both algorithms rapidly increases in the initial stage of evolution, and approaches the optimal value at around 100,000 evaluations. This means that the evolutionary algorithms are able to converge well within 100,000 evaluations.

6.3. Space

Yoo and Harman found that a greedy algorithm outperformed NSGA-II for Space and so we carried out experiments with Space and two objectives. Table 8 gives the mean normalised HV for the two-objective methods. Similar to Yoo and Harman we found that NSGA-II was outperformed by the greedy algorithm (p-values ≈ 0). Interestingly, both variants of MOEA/D outperformed the greedy algorithm (p-value 0.042).

	Greedy	NSGA-II	MOEA/D fixed c	MOEA/D varying c
Space	0.999947	0.901329	0.9999624	0.9999624

Table 8: Normalised HV for Space and Two Objectives

6.4. VoidAuth

Table 9 gives the normalised HV values for VoidAuth, averaged across the 30 runs for the stochastic methods. As before, MOEA/D with varying c has the highest mean HV for all experiments, with all comparisons being highly significant (p-values ≈ 0). In contrast to earlier results, MOEA/D with fixed c outperforms NSGA-II in all cases (p-values ≈ 0). Surprisingly, however, for the three objective experiments the greedy algorithm outperformed both NSGA-II and MOEA/D with fixed c. In addition, the greedy algorithm outperformed NSGA-II for two objectives (p-value ≈ 0).

# of Objectives	Greedy	NSGA-II	MOEA/D fixed c	MOEA/D varying c
2	0.953961	0.948569	0.954703	0.955090
3	0.907430	0.888960	0.890862	0.908200
4	0.877899	0.995943	0.999746	0.999747

Table 9: Normalised HV for VoidAuth

Figure 3 shows a boxplot of the HV values for the 2-objective methods. The plot shows little variability in the HV values for the 30 runs of the stochastic methods (standard deviations 0.0003, 0.0002 and 0.0004 for NSGA-II, MOEA/D with fixed c and MOEA/D with varying c, respectively). This has been observed also in the other cases considered and is the reason for the very small p-values associated with most differences.

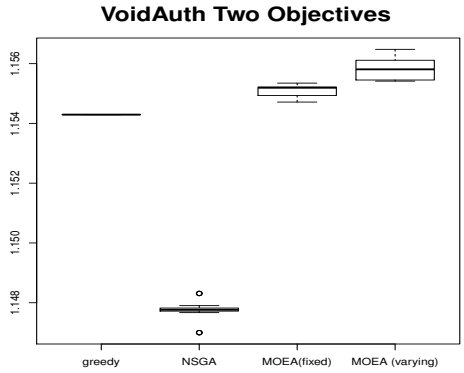

Figure 3: Boxplot of HV values for VoidAuth and 2-objective methods.

6.5. Test suite size for full coverage

The HV values give us valuable information about the relative effectiveness of the alternative optimisation methods and we have seen that many of the differences are statistically significant. In this section we explore the potential consequences of these differences by focussing on one special case, which is when the tester wants to achieve full coverage in each coverage criterion

considered. While this is only one possible scenario, it is realistic and much of the work on optimising regression testing has concerned this scenario. For each run, with a particular combination of experimental subject, number of objectives, and optimisation technique, we examined the final population and determined whether any of the test suites provided 100% coverage for all of the coverage metrics being considered. For example, for a run using two objectives we would examine the final population to determine whether any of the test suites provided 100% statement coverage. If there were such test suites then we recorded the size of the smallest such test suite across the 30 runs for the given combination (of experimental subject, number of objectives, and optimisation technique). In this section we compare the values found for the different techniques.

	Greedy	NSGA-II	MOEA/D fixed-c	MOEA/D varying-c
gzip-v3	2237.734[4]	2213.202[3]	1987.625*	1987.625*
gzip-v4	2240.176[3]	*nil*[4]	2173.935*	2173.935*
schedule-v2	632.625[3]	490.141[2]	655.456[4]	487.996*
tcas-v1	324.544[4]	324.476*	324.476*	324.476*
tot_info-v1	833.099[3]	833.099[3]	832.113[2]	828.045*

Table 10: Test suites providing full coverage: the two objective case

Table 10 shows the results for the two objective case and the SIR programs. Here, a value represents the cost of the test suite found that achieves 100% coverage and *nil* represents the case where there is no such test suite. A star is used to show that the value is the (possibly equally) best, a superscript of 2 denotes the value being the second best, 3 for the third best, and 4 for the fourth. It is immediately clear that in all cases MOEA/D with varying c produces the best (cheapest) test suite. All other techniques were worst in at least one case, with NSGA-II failing to produce a test suite with 100% coverage when applied to gzip-v4. Tables 11 and 12 shows the results for the three and four objectives case respectively.

	Greedy	NSGA-II	MOEA/D fixed-c	MOEA/D varying-c
gzip-v3	2376.816*	2377.386[4]	2376.816*	2376.816*
gzip-v4	2425.851[2]	2426.218[4]	2425.851[2]	2421.949*
schedule-v2	657.067[4]	491.013[2]	642.212[2]	488.334*
tcas-v1	433.561[3]	433.499[3]	433.478*	433.478*
tot_info-v1	944.338[4]	870.106[2]	882.134[3]	857.579*

Table 11: Test suites providing full coverage: the three objective case

In all cases we see that MOEA/D with varying c produces the smallest test suite. We also observe that NSGA-II failed to return a test suite with full coverage in one case. The scale of the differences varies considerably. For example, for tcas-v1 there is very little difference in test suite size. In contrast, for schedule-v2 and three objectives the greedy algorithm produces a test suite of cost of over 650, while MOEA/D with varying c produced a test suite with cost of under 500. Another example is gzip-v3 with two objectives, where the greedy algorithm and NSGA-II both produce test suites with costs of over 2,200 while both versions of MOEA/D produced test suites with costs of under 2,000.

	Greedy	NSGA-II	MOEA/D fixed-c	MOEA/D varying-c
gzip-v3	2442.768[3]	2401.356*	2496.154[4]	2401.356*
gzip-v4	2425.85[3]	2425.849*	2425.85[3]	2425.849*
schedule-v2	670.522*	678.26[3]	908.795[4]	670.522*
tcas-v1	433.565[4]	433.494[3]	433.528[3]	433.493*
tot_info-v1	1524.81[4]	1433.58*	1482.231[3]	1433.58*

Table 12: Test suites providing full coverage: the four objective case

Finally, Table 13 gives the values for VoidAuth. Again, MOEA/D with varying c is best in all cases but the greedy algorithm is equal best for the three and four objective problems. The relative performance of NSGA-II and MOEA/D with fixed c varies, with each failing to return a test suite with full coverage in one case.

	Greedy	NSGA-II	MOEA/D fixed-c	MOEA/D varying-c
2obj	44913.272[3]	*nil*[4]	44173.724[2]	41636.924*
3obj	46730.349*	47427.297[3]	*nil*[4]	46730.349*
4obj	59222.27*	59726.542[4]	59222.27*	59222.27*

Table 13: Test suites providing full coverage: VoidAuth

6.6. Computational cost of EMO algorithms

Table 14 gives the computational time of one run of the two EMO algorithms NSGA-II and MOEA/D on the six test suites[3]. Since the time difference of two versions of MOEA/D is negligible, we only show the results of the algorithm with fixed c. As can be seen from the table, the computational cost of the algorithms generally increases with the number of objectives. For the four

[3]The hardware used in the comparison experiment is a PC with 2.8 GHz Pentium 4 CPU with a memory of 1.00 GB.

programs gzip-v3, gzip-v4, tcas-v1, and tot_info-v1, the two algorithms can be executed within 6 minutes for all the three cases. For schedule-v2 and VoidAuth, the algorithms take more time, with nearly 15 and 30 minutes being required at most, respectively.

	2-objective case		3-objective case		4-objective case	
	NSGA-II	MOEA/D	NSGA-II	MOEA/D	NSGA-II	MOEA/D
gzip-v3	161.91	176.12	226.09	224.01	311.45	335.38
gzip-v4	161.31	179.38	237.52	254.06	294.55	319.70
schedule-v2	650.77	716.48	671.38	764.61	663.71	844.12
tcas-v1	197.27	204.75	204.51	217.88	217.78	243.72
tot_info-v1	279.70	294.38	274.94	297.70	291.70	335.00
VoidAuth	971.58	782.67	1167.09	1178.91	1633.62	1579.95

Table 14: Computational time (s) of NSGA-II and MOEA/D on the six programs

6.7. Summary of results

This section reported on the results of experiments that compared four different techniques: a classic greedy algorithm, NSGA-II (the previously used multi-objective optimisation algorithm) and two varieties of MOEA/D (fixed c and variable c). The primary method for comparing the techniques was through the HV values since this measure has a number of important properties and effectively combines information about the quality of the solutions and their diversity. We also looked at the cost of the test suites returned that provided full coverage since this corresponds to one scenario often considered.

MOEA/D with varying c was clearly the most effective technique. In all cases it produced the lowest HV values and the differences were highly significant in almost all cases. It also produced the cheapest test suite that achieved all of the objectives. The relative performance of the other methods varied. The greedy algorithm was consistently outperformed by NSGA-II and MOEA/D with fixed c for the smaller SIR programs. Similar to Yoo and Harman, we found that the greedy algorithm outperformed NSGA-II for Space. However, both variants of MOEA/D had superior performance to both NSGA-II and the greedy algorithm for Space. For VoidAuth, MOEA/D with fixed c always outperformed NSGA-II but the greedy algorithm was surprisingly effective (it outperformed MOEA/D with fixed c for two objectives and outperformed NSGA-II for both two and three objectives). Interestingly, Yoo and Harman found that NSGA-II outperformed a greedy algorithm for the Siemens programs used but not for Space, which again is a larger program (but smaller than VoidAuth). The relatively performance of NSGA-II and MOEA/D with fixed c varied, with neither being consistently preferable. However, MOEA/D with fixed c outperformed NSGA-II on the two larger programs (Space and VoidAuth).

7. Threats to Validity

In this section we briefly review the threats to validity in the experimental study and how these were addressed.

Threats to external validity relate to the degree to which we can generalise from the experiments. Since we can never know the true population of problems, and certainly cannot randomly sample from this, our ability to generalise from the results of software engineering experiments is inevitable limited. We attempted to reduce the impact of this problem through our choice of experimental subjects: as well as using well-known programs from the SIR suite (providing the potential for our results to be compared with those of others), we used an additional program from a different source. However, this is still far from being a representative sample and there is a need to perform additional experiments despite the statistically highly significant results obtained.

Threats to internal validity concern any factors that might introduce bias. In the experiments the techniques were all applied using the same experimental subjects. The nature of the evolutionary algorithms used did not allow us to use identical population sizes but we chose h values for MOEA/D to limit the differences. Importantly, the evolutionary algorithms were given the same resources (number of times the fitness function was evaluated). Three of the algorithms are stochastic and to reduce the impact of this in the comparison we ran each experiment 30 times and used statistical techniques to compare the resulting HV values. We also adjusted the p values used in order to reflect the number of comparison used. Since all factors other than the technique used were kept fixed, there is relatively little threat of internal validity.

Finally, threats to construct validity reflect the possibility that the measurements did not accurately reflect the properties of interest. Such threats to validity might arise when the tools are faulty. We limited these threats by using tools that are widely used. For the SIR programs, which were written in C, we used gcov to compute statement coverage and we used Valgrind to estimate the cost of executing a test case. We instrumented the code to return branch and MC/DC coverage information. VoidAuth was written in C# and for this we obtained coverage information through Microsoft Visual

Studio 2010 and we instrumented the program to provide information regarding statement coverage, branch coverage and MC/DC coverage. Where programs were instrumented, the instrumented version was manually reviewed and we checked the output of several test runs to confirm that this was operating correctly.

8. Conclusion

Regression testing should be carried out whenever a system or its environment is changed but this can be an expensive and time consuming process. There has thus been significant interest in test suite minimisation techniques for regression testing in which some subset of the regression test suite is chosen for use. Such techniques have traditionally been seen as optimisation techniques in which one aims to minimise the cost of testing subject to retaining full coverage for some coverage metric used. More recently, however, Yoo and Harman observed that there may be several coverage metrics of interest and also that there is value in returning to the tester a set of subsets of the regression test suite that provide alternative points in the trade-off between test suite execution cost and a vector of coverage values. This leads to the problem being represented as a multi-objective optimisation problem and the NSGA-II algorithm has previously been proposed [41].

This paper developed further the multi-objective approach of Yoo and Harman [41]. As well as the greedy algorithm and the multi-objective evolutionary algorithm NSGA-II used by Yoo and Harman, we also applied a more recently developed multi-objective evolutionary algorithm MOEA/D [43]. A second development relates to how algorithms were compared. The main approach previously used operated as follows: take the union of the non-dominated solutions produced by the algorithms, form a new set P of non-dominated solutions from these, and for an algorithm A determine how many of its solutions are not Pareto dominated by solutions in P [41]. Yoo and Harman also compared the sizes of the non-dominated solutions returned by the algorithms, arguing that an algorithm that returns many solutions provides the tester with more options. While this approach to evaluation is sensible, both aspects (convergence to the problem's Pareto front and the diversity of the information) are captured by the Hypervolume (HV) [49] of a solution set so we used this in our evaluation.

Interestingly, we found that initially MOEA/D performed poorly since the range of values for test suite cost is much larger than the range of values for coverage. This led us to adapt MOEA/D by introducing normalisation, with the normalisation including a constant c. Some initial trials suggested that a value of $c = 0.3$ works well so this led to two versions of MOEA/D: one with fixed c ($c = 0.3$) and another where the value of c was separately chosen for each experiment (subject and number of objectives). The experiments therefore compared four techniques: a greedy algorithm, NSGA-II, MOEA/D with fixed c, and MOEA/D with varying c.

We used five smaller experimental subjects from the SIR repository plus Space and one larger real-world system, VoidAuth. VoidAuth implements processes of the Universal Payment Gateway (UPG) service. MOEA/D with variable c had the best performance in all of the experiments, with the differences being statistically significant in almost all cases. The relative performance of the other methods varied. For the smaller SIR programs the greedy algorithm was always worst and the relative performance of NSGA-II and MOEA/D with fixed c varied. Similar to Yoo and Harman, we found that the greedy algorithm outperformed NSGA-II on Space. Interestingly, for Space both versions of MOEA/D performed better than both NSGA-II and the greedy algorithm. For VoidAuth we found that the greedy algorithm outperformed NSGA-II for the two and three objective problems and outperformed MOEA/D with fixed c for the three objective problem. In all cases MOEA/D with fixed c outperformed NSGA-II for the larger program, VoidAuth. Overall, although NSGA-II performed well in some experiments, the performance of MOEA/D was more robust.

While HV is widely used in evaluating multi-objective algorithms, it is not immediately clear what these differences in HV values mean to the software engineer. We therefore also considered a specific scenario, where the tester decides to use a test suite that provides full coverage for each metric considered. MOEA/D with variable c returned the smallest test suite in every experiment. We also looked at how the HV values changed as we let the number of evaluations of NSGA-II and MOEA/D increase and found that for both algorithms the HV values rapidly increased and converged at around 100,000 evaluations.

There are several lines of future work. First, there is a need to perform additional experiments in which the methods are applied to a wider range of experimental subjects. The use of $c = 0.3$ was based on some initial tuning and additional experiments are required in order to determine whether this really is a good choice. It would also be interesting to explore the sensitivity of MOEA/D with regards to changes in the value of c. Finally, there is the challenge of developing a version of MOEA/D that does not require the parameter c in the

normalisation process.

References

[1] J. Black, E. Melachrinoudis, D. Kaeli, Bi-criteria models for all-uses test suite reduction, in: 26th International Conference on Software Engineering (ICSE 2004), 2004, pp. 106–115.

[2] T. Y. Chen, M. F. Lau, Dividing strategies for the optimization of a test suite, Information Processing Letters 60 (3) (1996) 135–141.

[3] T. Y. Chen, M. F. Lau, A new heuristic for test suite reduction, Information and Software Technology 40 (5) (1998) 347–354.

[4] C. A. C. Coello, D. A. V. Veldhuizen, G. B. Lamont, Evolutionary Algorithms for Solving Multi-Objective Problems, 2nd Edition, Springer, Heidelberg, 2007.

[5] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms, John Wiley, New York, 2001.

[6] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.

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

[8] P. Fleming, R. Purshouse, R. Lygoe, Many-objective optimization: An engineering design perspective, in: Proceedings of the 3rd International Conference on Evolutionary Multi-Criterion Optimization (EMO), 2005, pp. 14–32.

[9] I. Giagkiozis, R. C. Purshouse, P. J. Fleming, Generalized decomposition and cross entropy methods for many-objective optimization, Information Sciences 282 (2014) 363–387.

[10] M. Harman, Making the case for MORTO: Multi objective regression test optimization, in: ICST Workshops, IEEE Computer Society, 2011, pp. 111–114.

[11] 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.

[12] S. Holm, A simple sequentially rejective multiple test procedure, Scandinavian Journal of Statistics 6 (2) (1979) 65–70.

[13] H.-Y. Hsu, A. Orso, MINTS: A general framework and tool for supporting test-suite minimization, in: 31st International Conference on Software Engineering (ICSE 2009), IEEE, 2009, pp. 419–429.

[14] H. Ishibuchi, N. Akedo, Y. Nojima, Behavior of multi-objective evolutionary algorithms on many-objective knapsack problems, IEEE Transactions on Evolutionary Computation 19 (2) (2015) 264–283.

[15] H. Ishibuchi, Y. Hitotsuyanagi, N. Tsukamoto, Y. Nojima, Many-objective test problems to visually examine the behavior of multiobjective evolution in a decision space, in: Proc. Parallel Problem Solv. Nature, 2010, pp. 91–100.

[16] H. Ishibuchi, N. Tsukamoto, Y. Nojima, Evolutionary many-objective optimization: A short review, in: Proc. IEEE Congr. Evol. Comput., 2008, pp. 2419–2426.

[17] D. Jeffrey, N. Gupta, Improving fault detection capability by selectively retaining test cases during test suite reduction, IEEE Transactions on Software Engineering 33 (2) (2007) 108–123.

[18] F. Kang, J. Li, H. Li, Artificial bee colony algorithm and pattern search hybridized for global optimization, Applied Soft Computing 13 (4) (2013) 1781–1791.

[19] F. Kang, J. Li, Z. Ma, Rosenbrock artificial bee colony algorithm for accurate global optimization of numerical functions, Information Sciences 181 (16) (2011) 3508–3531.

[20] F. Kang, J. Li, Q. Xu, Structural inverse analysis by hybrid simplex artificial bee colony algorithms, Computers & Structures 87 (13) (2009) 861–870.

[21] F. Kang, J. Li, Q. Xu, Damage detection based on improved particle swarm optimization using vibration data, Applied Soft Computing 12 (8) (2012) 2329–2335.

[22] R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York-London, 1972, pp. 85–103.

[23] M. Laumanns, L. Thiele, K. Deb, E. Zitzler, Combining convergence and diversity in evolutionary multiobjective optimization, Evolutionary Computation 10 (3) (2002) 263–282.

[24] H. Li, Q. Zhang, Multiobjective optimization problems with complicated Pareto sets, MOEA/D and NSGA-II, IEEE Transactions on Evolutionary Computation 13 (2) (2009) 284–302.

[25] K. Li, S. Kwong, J. Cao, M. Li, J. Zheng, R. Shen, Achieving balance between proximity and diversity in multi-objective evolutionary algorithm, Information Sciences 182 (1) (2012) 220–242.

[26] M. Li, S. Yang, X. Liu, Shift-based density estimation for Pareto-based algorithms in many-objective optimization, IEEE Transactions on Evolutionary Computation 18 (3) (2014) 348–365.

[27] M. Li, S. Yang, X. Liu, Bi-goal evolution for many-objective optimization problems, Artificial Intelligence 228 (2015) 45–65.

[28] M. Li, S. Yang, X. Liu, R. Shen, A comparative study on evolutionary algorithms for many-objective optimization, in: Evolutionary Multi-Criterion Optimization, 2013, pp. 261–275.

[29] Z. Li, M. Harman, R. M. Hierons, Search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (4) (2007) 225–237.

[30] C. von Lücken, B. Barán, C. Brizuela, A survey on multi-objective evolutionary algorithms for many-objective problems, Computational Optimization and Applications 58 (3) (2014) 707–756.

[31] R. C. Purshouse, P. J. Fleming, On the evolutionary optimization of many conflicting objectives, IEEE Transactions on Evolutionary Computation 11 (6) (2007) 770–784.

[32] G. Rothermel, M. J. Harrold, Empirical studies of a safe regression test selection technique, IEEE Transactions on Software Engineering 24 (6) (1998) 401–419.

[33] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (10) (2001) 929–948.

[34] Requirements, T. C. for Aviation, RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification, RTA, Washington, DC, 1992.

[35] H. Sato, H. Aguirre, K. Tanaka, Controlling dominance area of solutions and its impact on the performance of MOEAs, in: Proceedings of the 4th International Conference on Evolutionary Multi-Criterion Optimization (EMO), 2007, pp. 5–20.

[36] A. Srivastava, J. Thiagarajan, Effectively prioritizing tests in development environment, in: ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), 2002, pp. 97–106. doi:10.1145/566172.566187.

[37] L. H. Tahat, B. Korel, M. Harman, H. Ural, Regression test suite prioritization using system models, Software Testing, Verification and Reliability 22 (7) (2012) 481–506.

[38] D. A. V. Veldhuizen, G. B. Lamont, Evolutionary computation and convergence to a Pareto front, in: Late Breaking Papers at the Genetic Programming Conference, 1998, pp. 221–228.

[39] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information & Software Technology 43 (14) (2001) 841–854.

[40] S. Yang, M. Li, X. Liu, J. Zheng, A grid-based evolutionary algorithm for many-objective optimization, IEEE Transactions on Evolutionary Computation 17 (5) (2013) 721–736.

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

[42] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification and Reliability 22 (2) (2012) 67–120.

[43] Q. Zhang, H. Li, MOEA/D: A multiobjective evolutionary algorithm based on decomposition, IEEE Transactions on Evolutionary Computation 11 (6) (2007) 712–731.

[44] Q. Zhang, W. Liu, E. Tsang, B. Virginas, Expensive multiobjective optimization by MOEA/D with gaussian process model, IEEE Transactions on Evolutionary Computation 14 (3) (2010) 456–474.

[45] Q. Zhang, A. Zhou, S. Zhao, P. N. Suganthan, W. Liu, S. Tiwari, Multiobjective optimization test instances for the CEC 2009 special session and competition, Working Report CES-487, School of CS & EE, University of Essex (2009).

[46] A. Zhou, B. Qu, H. Li, S. Zhao, P. Suganthan, Q. Zhang, Multiobjective evolutionary algorithms: A survey of the state of the art, Swarm and Evolutionary Computation 1 (1) (2011) 32–49.

[47] E. Zitzler, S. Künzli, Indicator-based selection in multiobjective search, in: Proc. Parallel Problem Solv. Nature, Springer, 2004, pp. 832–842.

[48] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization, in: Evolutionary Methods for Design, Optimisation and Control, Barcelona, Spain, 2002, pp. 95–100.

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