IEEE International Conference on Software Testing Verification and Validation Workshops

Measuring and Improving Latency to Avoid Test Suite Wear Out

Shin Yoo & Mark Harman
King's College London
Centre for Research on Evolution, Search & Testing (CREST)
London, UK
{*shin.yoo, mark.harman*}*@kcl.ac.uk*

Shmuel Ur
IBM, Haifa Research Lab
Haifa University Campus
Haifa, Israel
UR@il.ibm.com

Abstract

This paper introduces the concept of test suite latency. The more latent a test suite, the more it is possible to repeatedly select subsets that achieve a test goal (such as coverage) without re-applying test cases. Where a test case is re-applied it cannot reveal new information. The more a test suite is forced to re-apply already applied test cases in order to achieve the test goal, the more it has become 'worn out'. Test suite latency is the flipside of wear out; the more latent a test suite, the less prone it is to wear out. The paper introduces a theory of test suite latency. It presents results from the empirical study of latency, highlighting the need for latency enhancement. The paper also introduces a strategy and algorithms for improving latency and an empirical study of their effectiveness. The results show that local search is effective at improving the latency of a test suite.

1. Introduction

Test suite minimization aims to reduce the number of test cases that need to be applied, while retaining identical or near-identical satisfaction of a chosen test adequacy criterion. Test suite minimization is increasingly important, because of the tendency for test suite size to grow over time [7]. This growth in test suite size comes from a variety of sources, including the use of capture replay tools, improvements in test data generation techniques and procedures for capturing customer–created test cases arising from their use of deployed software. Test suite minimization provides a mechanism for managing the size of the test suite when there are insufficient resources available to apply all available test cases during a period of testing activity.

For example, a large IBM middleware product has a test suite of 20,000 test cases which take 10 days to run. IBM has an automatic regression test selection mechanism that chooses, for each regression test, a subset of the test cases to run. The algorithm first determines the relevant tests given the code that was changed. However, it then needs to further minimize within these selected test cases in order to achieve a test set that can be executed within the time available (which is normally eight to twelve hours).

Typically, the test adequacy criterion that minimization seeks to maintain is code coverage. That is, the minimized test suite aims to achieve the same, or nearly the same coverage as the original test suite, but with a reduced number of test cases. This paper also adopts coverage as the test adequacy criterion, but the techniques and results presented here can be adapted to apply to other test adequacy criteria.

IBM's experience with repeated application of test suite minimization for multiple iterations revealed a problem: minimized test suites have a tendency to 'wear out'. That is, some of the test cases execute parts of the code that have not changed (as well as parts that have). However, for deterministic software systems, repeated execution of unchanged sections of code with the same input cannot, by definition, reveal any new faults provided that the rest of the environment do not change. As the number of previously used test cases increases, the test suite 'wears out' its ability to reveal additional faults. The more a test suite contains test cases that have already been executed on previous versions of the system, the less the test suite is likely to reveal.

In order to avoid wear out, minimization strategies can be adapted to select different subsets of the test pool to form a new test suite to be applied at each regression cycle. It is trivial to adapt test minimization algorithms to avoid re-selection of previously used test cases, thereby yielding a novel minimized test suite on each iteration. However, at each iteration, the algorithm may achieve lower coverage because it cannot re-use test cases that have already been executed. Some parts of the system may have very few available test cases that are able to cover them. This raises an important question: How many times can a test pool allow repeated reduction that results in a novel set of test cases?

This paper introduces the term *latency* to capture this property. The more a test suite allows for selection of novel minimized test suites, the higher is its latency. The use of word 'latency' is overloaded in computer science, imbuing it with possible assiciations that are unwanted in the context of this work. The dictionary definition of the word is the one intended in this paper:

Latent(ˈleitənt) *a.* Hidden, concealed; present or existing, but not manifest, exhibited, or developed [1].

978-0-7695-3671-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSTW.2009.10

This paper addresses the issue of test suite latency. The paper introduces a theoretical foundation that forms the basis of an empirical investigation of real world test suites. Some of the test suites for the programs studied show surprisingly low latency. In order to *repair* test suites with low latency, the paper introduces a novel strategy for enhancing the latency of test suites. The strategy combines use of test suite reduction techniques and test data generation techniques. Three algorithms are presented in order to implement the strategy. The three algorithms are studied for their performance and efficiency.

The primary contributions of the paper are as follows:
1) The paper introduces a theoretical formulation of the concept of latency of a test suite. The theory supports investigation of formal properties of test suite reduction. For example, it is shown that repeated application of greedy test suite reduction results in monotonically decreasing statement coverage.
2) The paper presents an empirical study of latency for real world test suites. The results show that the coverage decrease is surprisingly severe for some programs, for example, the widely studied program: `space`.
3) The paper reports on the effect of allowing overlapping test suite reduction on the test suite latency. The empirical results show that even allowing 70% of the selected test cases to be reused (i.e. 70% overlap) fails to noticeably enhance the test suite's latency.
4) The paper shows how test suite enhancement can improve latency in a test suite. The paper uses our existing hill climbing algorithm [28] and compares its results to those obtained from an estimation of distribution algorithm. The paper presents an empirical study of the proposed latency enhancement strategy on four simple benchmark programs. The statistical analysis of the results shows that the hill climbing algorithm performs significantly better than both the estimation of distribution algorithm and random test data generation.

The rest of the paper is organised as follows. Section 2 presents formal definitions and theoretical results that underpin the subsequent empirical studies. Section 3 presents an empirical study of test suite latency for open source programs. Section 4 introduces the strategy and algorithms for combining test suite reduction and generation, which is evaluated in Section 5. Section 6 discusses related work and Section 7 concludes.

2. Problem Statement

Test suite reduction is the problem of selecting a subset of a given test suite in order to reduce the effort required to execute the test cases [15]. In this paper, the remaining unselected test cases are referred to as the 'retained' set, since they may be retained for subsequent selections. Some test suite quality metrics, for example structural code coverage, monotonically decrease as the number of retained test cases decreases. A test suite reduction technique is said to be monotonically decreasing with respect to a quality metric, if the repeated application of the reduction technique results in monotonically decreasing values of the quality metric. The latency of a test suite is defined as the maximum number of times a reduction technique can be applied to the retained portion of the test suite before the quality metric falls below a predefined threshold level. The definitions below formalise these concepts, facilitating theoretical study of monotonicity and latency.

Let $\mathcal{S}$ be the initial test suite available for testing. Let S be the set of all subsets of $\mathcal{S}$.

Definition 1 ***Test Suite Reduction Procedure***
A test suite reduction procedure τ is a relation in $S \leftrightarrow (S \times S)$ such that $s\ \tau\ (a, r) \Rightarrow a \cup r = s$. If additionally $\forall s.\ s\ \tau\ (a, r) \Rightarrow a \cap r = \emptyset$, then τ is said to be non-overlapping.

Thus, a reduction procedure τ is a relation, where $s\ \tau\ (a, r)$ means that when procedure τ is applied to a test suite s, the subset a of s is applied to the program under test, while the set r is retained for subsequent regression testing. If $a \cap r \neq \emptyset$ then some of the selected test cases will be reused in subsequent selections.

Definition 2 ***Applied Test Cases***
App is a function in $(S \leftrightarrow S \times S) \rightarrow (S \rightarrow S)$ such that $s\ App(\tau)\ a \Leftrightarrow s\ \tau\ (a, r)$

Definition 3 ***Retained Test Cases***
Ret is a function in $(S \leftrightarrow S \times S) \rightarrow (S \rightarrow S)$ such that $s\ Ret(\tau)\ r \Leftrightarrow s\ \tau\ (a, r)$

$App(\tau)$ is the projection of a procedure τ onto the used portion of the test suite; $Ret(\tau)$ is the projection of a procedure τ onto the retained portion of the test suite.

Definition 4 ***Repeated Reduction***
Repeated reduction from s using τ over n times $(n \geq 1)$ is denoted by τ^n and defined as follows:

$$\tau^1 = \tau$$

$$s\ \tau^{n+1}\ (a, r) \text{ iff } s\ \tau^n\ (a', r') \text{ and } r'\ \tau\ (a, r)$$

Repeated reduction is an iterative procedure of applying a reduction technique τ to the retained portion of the test suite from the previous reduction.

Definition 5 ***Quality Metric***
A quality metric μ is a function in $S \rightarrow \mathbb{R}$ that measures the quality of a set of test cases. If additionally $\forall s' \subseteq s.\ \mu(s') \leq \mu(s)$, then μ is said to be monotonically decreasing.

Definition 6 ***μ-Optimality***
A test suite reduction τ is μ-optimal if and only if $\forall s.\ \mu(App(\tau(s))) = max\{\mu(s')|s' \subseteq s\}$.

Proposition 1 *If μ is monotonically decreasing then a non-overlapping μ-optimal τ is guaranteed to be monotonically decreasing with respect to μ, that is,*

$$\mu(App(\tau^{n+1}(s))) \leq \mu(App(\tau^n(s))).$$

proof Let a and r be sets of test cases such that $s\ \tau^n\ (a, r)$. By definition of τ, $App(\tau^{n+1}(s)) = App(\tau(r))$ and $App(\tau^n(s)) = App(\tau(a \cup r))$. By definition of monotonically decreasing μ, $r \subseteq a \cup r \Rightarrow \mu(r) \leq \mu(a \cup r)$. By definition of μ-optimality, $\mu(App(\tau(r))) = \mu(r)$ and $\mu(App(\tau(a \cup r))) = \mu(a \cup r)$. Therefore $\mu(App(\tau^{n+1}(s))) \leq \mu(App(\tau^n(s)))$.

For a test suite reduction procedure τ that is monotonically decreasing with respect to a quality metric μ, the latency of a given test suite, s, against τ is defined as follows.

Definition 7 ***Latency***
A latency measure, λ, is a function in

$$(S \to \mathbb{R}) \to \mathbb{R} \to (S \leftrightarrow (S \times S)) \to S \to \mathbb{N}$$
$$\lambda\mu\alpha\tau s = \textit{largest } n \textit{ such that } \mu(Ret(\tau)^n s) \geq \alpha$$

That is, the latency $\lambda\mu\alpha\tau s$ of a test suite s with respect to a quality metric μ and threshold quality for adequacy α and a test suite reduction procedure τ is the largest number of times the procedure τ can be repeatedly applied to the retained portion of s before the quality (as measured by μ) falls below α. Higher λ value of a test suite means that the test suite is capable of providing multiple disjoint subsets of test cases with a quality metric value above the threshold, which makes the test suite more latent. If a test suite satisfies its given testing requirements (such as full branch coverage), then its measured latency should be at least 1. However, it would be advantageous to achieve λ value of 2 or higher so that the tester can have multiple test case subsets that achieve the same testing requirements.

3. Monotonicity & Overlap Study

The research questions for the monotonicity and overlap empirical study are: **RQ1. Monotonicity Measurement:** Are there test suites with poor latency? How does the monotonically decreasing property of quality metrics manifest itself in real world programs and their test suites? **RQ2. Overlap Effect:** How much increase in latency can be achieved by loosening the non-overlapping constraint so that some test cases are allowed to be reused across consecutive test suite reductions?

The ideal criterion for test suite reduction is the highest possible fault detection capability. However, the fault detection information is not available before the testing finishes. As a result, structural code coverage is often used as a readily available surrogate. Each test case is said to *cover* different parts of the program, e.g., statements, branches or blocks, that it executes.

Proposition 2 *Structural code coverage is a monotonically decreasing quality metric. The proof is trivial in this case: it is not possible to increase the code coverage achieved by a test suite by removing a test case.*

From Section 2, coverage is a monotonically decreasing quality metric. This means that any non-overlapping coverage-optimal test suite reduction technique also yields a monotonically decreasing coverage quality metric.

The paper uses the widely studied additional greedy algorithm as the test suite reduction technique [5], [6], [11], [17]. Maximising coverage achieved by testing is a set cover problem. The goal of test suite reduction is to have the smallest set of test cases that covers the entire program.

Greedy algorithms are known to be efficient, producing solutions to set cover problems of size n that are within $\ln n$ of the global optimum [9].

Proposition 3 *The additional greedy algorithm is a non-overlapping μ-optimal test suite reduction procedure when $\mu(s)$ is the coverage achieved by s.*
proof: The non-overlapping property of the additional greedy algorithm is obvious by the definition of the algorithm. For the μ-optimality, assume the contrary, that is, $\exists s' \subset s.\mu(s') > \mu(App(\tau(s)))$. Since code coverage is monotonically decreasing, $s' \subseteq s \Rightarrow \mu(s') \leq \mu(s)$. Therefore $\mu(App(\tau(s))) < \mu(s') \leq \mu(s)$, which means that there exists a test case, t, such that $t \in s, t \notin App(\tau(s)), \mu(App(\tau(s))) < \mu(\{t\} \cup App(\tau(s)))$. However, such t cannot exist if τ is the additional greedy algorithm; otherwise the algorithm would have chosen t after choosing the subset $App(\tau(s))$. Therefore, the additional greedy algorithm must be μ-optimal when μ measures code coverage.

Note that other non-greedy reduction techniques may not be monotonically decreasing. While repeated application of any reduction approach to a finite test suite must ultimately result in zero coverage, this does not mean that coverage will necessarily decrease monotonically.

3.1. Experimental Design

Six program test suites have been analysed for their level of latency. The programs were retrieved from Software-artifact Infrastructure Repository(SIR) along with test suites for each program [4]. The size of programs and test suites are shown in Table 1.

The programs are analysed for levels of latency against the greedy test suite reduction. The quality metric μ of the selected subsets is the statement coverage that each selected subset achieves, since this is one of the weakest

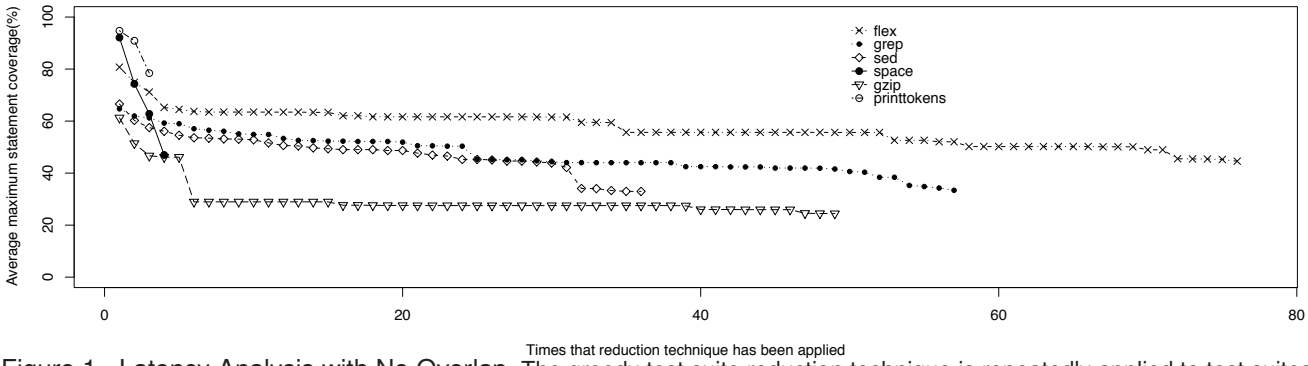

Figure 1. Latency Analysis with No Overlap. The greedy test suite reduction technique is repeatedly applied to test suites, with no overlap between repeated reductions. Notice that only 4 reductions are possible for the program `space` and that coverage drops dramatically for all test suites considered.

Program	Lines of code	Test suite size
`printtokens`	726	17
`flex`	15,297	567
`grep`	15,633	806
`gzip`	8,889	213
`sed`	19,737	370
`space`	6,199	156

Table 1. Test suite size for programs studied

coverage criteria; if even statement coverage latency cannot be achieved then clearly there is a need for latency improvement. Coverage information was obtained using `gcov` profiler tool from `gcc`.

One potential method addressing coverage degradation is to allow a certain level of test case overlap between reductions. This allows consecutive reductions to share test cases that may contribute to high quality metric value of the selected subset. Test suites which can retain coverage potency by allowing a degree of overlap can only be said to be 'weakly' latent, because some test case re-use is allowed, as determined by the overlap level. To explore the degree to which overlap allows increased coverage, four different levels of overlap are analysed, each allowing $\{10\%, 30\%, 50\%, 70\%\}$ reuse between test suite reductions. The reusable test cases are determined randomly. The latency analysis is performed 30 times for each test suite with average results presented.

3.2. Results and Analysis

Figure 1 shows the result of the latency analysis. As the additional greedy algorithm is repeatedly applied to the test suite, the maximum coverage achieved monotonically decreases, providing an answer to **RQ1**. The rate of decrease is generally higher in the region of early iterations, showing that each test suite contains a small number of test cases that cover the *hard-to-reach* regions of the programs.

Two programs in particular, `printtokens` and `space`, show an interesting contrast to other programs. The very first iteration on `printtokens` and `space` achieves over 94% and 92% of coverage separately, which is higher than that achieved for any of the other programs. This means that, as long as the tester is concerned with obtaining only a *single* set of test cases to execute, `printtokens` and `space` have very satisfactory test suites. However, the coverage of both programs deteriorates very quickly with repeated reductions. Should the tester want more than a single 'once and for all' test set, then each consecutive reduction from the test suite will necessarily lead to the re-execution of a great many test cases. This shows that even a test suite that achieves high level of test adequacy can be vulnerable to low latency.

Figure 1 showed that coverage drops dramatically as repeated reductions are made. In this figure no overlap was permitted, ensuring that each new reduction enjoys an entirely fresh set of test cases. This requirement for 100% 'freshness' could be the cause of the dramatic drop in coverage, suggesting that a more relaxed approach, allowing *some* overlap, may improve coverage. This is the motivation for **RQ2** which concerns the effects (on coverage achievable) of various levels of overlap allowance.

Figure 2 shows how the latency of test suites changes when different percentages of the selected test cases are allowed to be reused after each application of the reduction technique. The results plotted in Figure 2 are average values of coverage achieved by each of 30 executions. The allowance of overlap affects the test suites of `printtokens` and `space` positively, resulting in slower degradation of coverage. However, it is interesting to observe that even allowing 70% of the selected test cases to overlap between reductions does not affect the latency of test suites for other programs in any noticeable way. Overall, the results from the overlapping formulation of the analysis show that coverage decreases dramatically even when overlaps are allowed, which provides an answer to **RQ2**.

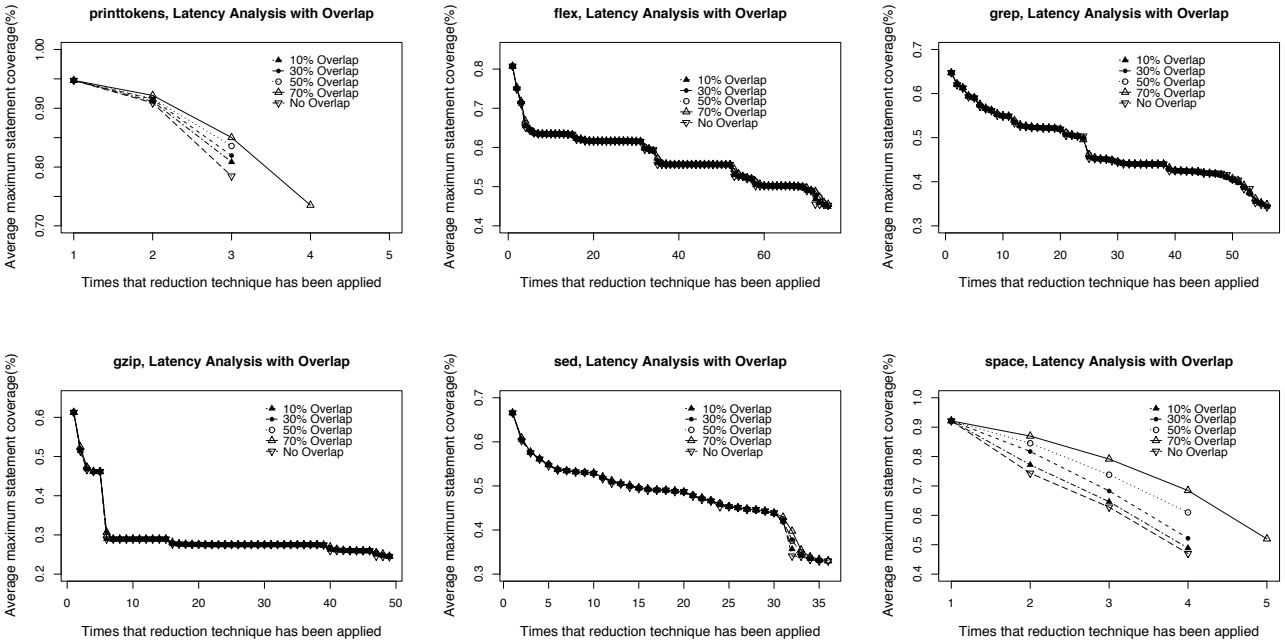

Figure 2. Weak Latency Analysis (With Overlapping Test Subsets Allowed). As more test cases are re-used by allowing a higher level of overlapping, test suites of `printtokens` and `space` show some achievement of weak latency. However, even allowing overlapping does not noticeably affect other programs. This can be seen because coverage profiles are almost identical regardless of overlap allowance.

4. Latency Enhancement Strategy

The results in Section 3.2 show that even allowing high levels of overlapping does not improve test suite latency noticeably. This motivates the consideration of ways of enhancing latency. In order to propose a simple and efficient method of enhancing low latency of test suites, the present paper raises the following question: if there are known test cases that are effective at achieving coverage, are there different but similar test cases that achieve the same coverage? More formally, given a set of test cases, a, such that $s\ \tau\ (a, r)$ and $\mu(a)$ satisfies the testing criteria, is it realistic to assume that there may exist a' such that a' is similar to a and $\mu(a') = \mu(a)$?

If so, then the enhancement of low latency of a test suite can benefit from knowledge of a. That is, the existing test cases can be used to seed an automated search for additional test cases that also satisfy the testing criteria. This approach to latency enhancement uses the 'test data augmentation' technique of Yoo and Harman [28]. The paper considers two search algorithms: a hill climbing algorithm, adapted from Yoo and Harman [28] and an estimation of distribution algorithm: and compares them to unguided random test data generation. Hill climbing is a local search technique that will seek solutions near to the existing test cases in the space of possible test inputs. EDA (Estimation of Distribution Algorithm) is a global search technique that will consider solutions that are sampled from probabilistic distributions centered around the existing test cases.

4.1. Combined Reduction & Generation Strategy

Figure 3 shows how test data generation techniques work in conjunction with test suite reduction techniques in order to enhance low latency of a test suite. A test suite reduction technique selects the best subset according to a quality metric μ, which is set to branch coverage in the empirical study present in the paper. This forces the enhancement strategy to 'work harder' because branch coverage subsumes statement coverage. These selected *model* test cases are fed into a test data generation technique, which seeks to generate new test cases that are different from the model test cases but still achieve the same quality metric value. The generated test cases are then added to the test suite for the next iteration.

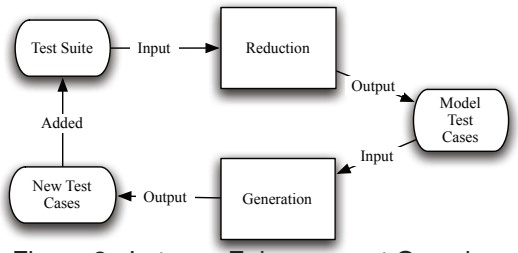

Figure 3. Latency Enhancement Overview

A single fitness function is used to guide both search algorithms, facilitating comparison. Let t be the individual test case the fitness of which is being measured, and t' be the original test case for which t seeks to mimic the behaviour. Let $\Delta_\mu(t, t')$ be the difference in quality metrics between

two test cases, and $\Delta_d(t, t')$ be the distance between the input vectors of two test cases.

$$\Delta_\mu(t, t') = |\mu(\{t\}) - \mu(\{t'\})|$$
$$\Delta_d(t, t') = distance(t, t')$$

The fitness value of the test case t, $f(t)$, is defined as follows [28]:

$$f(t) = \begin{cases} \Delta_d(t, t') & \text{if } \Delta_\mu(t, t') = 0 \wedge \Delta_d(t, t') > 0 \\ 0 & \text{if } \Delta_\mu(t, t') = 0 \wedge \Delta_d(t, t') = 0 \\ -\Delta_\mu(t, t') & \text{if } \Delta_\mu(t, t') > 0 \wedge \Delta_d(t, t') > 0 \end{cases}$$

If t is the same as t', the fitness function returns 0. However, if t is different from t' yet still achieves the same quality metric ($\Delta_\mu(t, t') = 0$), then t is guaranteed to receive a fitness value higher than 0, thereby encouraging the solution to move farther away from t'. Finally, if t is different from t' but has lower quality metric, t is guaranteed to receive a fitness value lower than 0, but encouraged to reduce the difference in quality metric.

It should be noted that, with this particular search problem, finding the global optimum is not as important as finding as many qualifying solutions as possible. From the definition of the fitness function, qualifying solutions will be the test cases with fitness values higher than 3.

4.2. Hill Climbing

The hill climbing algorithm is one of the simplest local search algorithms [18], which is shown in Algorithm 1. The algorithm requires a definition of 'neighbouring solutions' for a given problem. The algorithm starts the search from a random solution, and considers neighbouring solutions. If a neighbour has higher fitness than the current position, the algorithm *climbs* to the fitter neighbour. This is repeated until there is no fitter neighbour, at which point the algorithm has reached one of the local optima in the search space. The *steepest ascent* hill climbing algorithm moves to the neighbour with the highest fitness, whereas the *first ascent* hill climbing algorithm moves to the first neighbour it considers with higher fitness than the current solution.

Algorithm 1: Hill Climbing Algorithm

```
(1)  x ← a random solution
(2)  while true
(3)      N ← neighbours of x
(4)      if ∃x' s.t. x' ∈ N ∧ fitness(x') > fitness(x)
(5)          x ← x'
(6)      else
(7)          break
(8)  return x
```

For the latency enhancement study, the algorithm is modified to start from the known test case. Neighbouring solutions are defined as test cases that contain a single input variable that differs from the known test cases. These are created by either adding or subtracting a predefined amount to each input variable of the known test case, which results in $2n$ neighbouring solutions for a test case with an input vector of length n.

One way of escaping local optima is to restart the algorithm with a different starting solution whenever the algorithm reaches a local optimum. However, in the version used in the paper, the starting solution is set to the known test case in order to explore the input space near the known test case. To avoid deterministic behaviour, the algorithm used in the paper adopts a *random first ascent*; the algorithm considers the neighbouring solutions in random order and moves to the first neighbour with higher fitness, which introduces randomness to the algorithm. Since the goal of the search is to explore the search space around the known test cases rather than finding the global optimum, the inability to escape local optima is thought to be less critical so long as the algorithm retains the ability to explore the neighbouring solutions.

4.3. Estimation of Distribution Algorithm

EDAs (Estimation of Distribution Algorithms) were first introduced in the field by Mühlenbein and Paaß [14]. They have been previously applied to software test data generation [20], [19] by Sagarna et al. The biggest difference between EDA and other evolutionary computation heuristics is that EDA does not rely on cross-over and mutation operators in order to generate new individual solutions. Instead, the individual solutions forming the population of the next generation are generated from a sampling of probability distribution, estimated from the previous generation.

Algorithm 2: Estimation of Distribution Algorithm

```
(1)  P_0 ← generate initial population randomly
(2)  while stopping criterion is not met
(3)      S_{i-1} ← select fitter individuals from P_{i-1}
(4)      E_{i-1} ← probability distribution of S_{i-1}
(5)      P_i ← sample E_{i-1}
```

Algorithm 2 shows the top level view of EDA. The algorithm starts by randomly generating the individual solutions that form the initial population. During the following iterations, the algorithm first selects fitter individual solutions, guided by a predefined fitness function. Based on the selected individual solutions, the algorithm then estimates the probability distribution of the individuals. The next generation of individual solutions are sampled from the estimated probability distribution.

As in the case with the hill climbing algorithm, the estimation of distribution algorithm is modified in order to benefit from the knowledge of existing test cases. The initial population is not generated randomly; instead, Gaussian distributions are formed around the input variables of the known test cases (with mean values equal to the known values) and the initial population is sampled from these Gaussian distributions. By controlling the variance in the Gaussian distributions, it is possible to set the range of exploration around the known test cases.

5. Enhancement & Efficiency Study

The research questions for the enhancement and efficiency empirical study are: **RQ3. Enhancing Latency:** Can latency be improved by algorithms that implement the proposed combination of reduction and generation? **RQ4. Assessing Efficiency:** How efficient is the proposed approach to enhancing latency of test suites? Which search algorithm performs best for the proposed approach?

5.1. Subject Programs

A set of well-known benchmark programs for structural test data generation techniques is used. These are described in Table 2.

Program	Branches	Search Space
`triangle1`	20	2^{96}
`triangle2`	26	2^{96}
`remainder`	18	2^{64}
`complexbranch`	22	2^{192}

Table 2. Subject programs for enhancement

`Triangle1` is an implementation of the widely used program that determines whether the given three numeric values, each representing the length of a segment, can form a triangle. `Triangle1` is used by Michael and McGraw in their study of test data generation [13]. `Triangle2` is an alternative implementation of the same program by Sthamer who also studied test data generation for `remainder`, a program that calculates the remainder of the division of two integer input [21]. Finally, `complexbranch` is a program specifically created as a challenge for test data generation techniques [24]. It contains several branches that are known to be hard to cover. For all programs, the search space is both large and non-trivial.

The initial test suites for the subject programs are generated such that 100% branch coverage is achieved. The initial test suites for the studied programs were generated by branch-by-branch approach. Programs were instrumented for the measurement of branch coverage. For each branch in the program, a single test case was generated to make the predicates both `true` and `false`. This is standard practice in search based test data generation [12].

5.2. Experimental Design

The comparison between the hill climbing algorithm and EDA is performed by allowing both algorithms 3,000 fitness evaluations. Both algorithms are also compared to random test data generation technique, which generates 3,000 random test cases. All three algorithms are executed 30 times to factor out their inherent stochastic properties.

The fitness function described in Section 4.1 is used for both algorithms in order to facilitate comparison. The distance in quality metric for a single test case is measured by the Hamming distance between two binary strings that correspond to the code coverage of each test case. The distance between two test cases is measured by simply taking the Euclidean distance between two input vectors, which are numeric for all programs studied.

Following Korel [10], the neighbours in the hill climbing algorithm are generated by adding and subtracting 1 to each input variable in the current test case. Since the fitness function encourages the solution to move farther away from the original test case, it is possible that the algorithm exhausts the given fitness evaluations while moving arbitrarily farther away from the original test case. Therefore, the search is restricted to 10 ascents from the original test case.

For the initial Gaussian distributions that are used for the initialisation of EDA, the standard deviation values are set to 3 for all input variables. The population size is set to 30. The algorithm terminates under two conclusions: 1) the population converges; or 2) the total number of generations processed exceeds 100.

5.3. Results and Analysis

Figure 4 shows the result of the attempts to enhance low latency of test suites by generating additional test cases with the algorithms described above. Each algorithm is executed 30 times, resulting in 30 different enhanced test suites. For each enhanced test suite, the additional greedy algorithm is applied repeatedly in order to measure the quality metric (the branch coverage), with the average plotted in Figure 4. It should be noted that the latency rates, $\lambda(coverage)(0)(greedy)$, are different at each execution of EDA and the hill climbing algorithm due to their inherent stochastic property. The results plotted in Figure 4 include only up to $min(\lambda(coverage)(0)(greedy))$ reductions indicating worst case performance. On the other hand, the results for the random test data generation are plotted only up to 50 reductions. The solid line with ▲ represents the latency of the original test suites.

The results from the hill climbing algorithm show that the enhancement strategy has improved the latency for all four programs, which answers **RQ3**. For `remainder` and `complexbranch`, the hill climbing algorithm manages to enhance the latency of test suites so that 100% branch coverage is maintained across consecutive reductions. For `triangle1` and `triangle2`, the enhanced test suites fail to maintain 100% branch coverage but their coverage drops much more slowly than both that of the original test suite and those enhanced by other algorithms.

The reason why the hill climbing algorithm cannot maintain 100% branch coverage for `triangle1` and `triangle2` can be found in the semantics of the programs. The program `triangle1` contains a branch that determines whether the given 3 integers form an equilateral triangle. The definition of the neighbours used in the hill climbing algorithm prevents any generation of fitter neighbour for this branch. This is because the algorithm changes only a single input at a time to generate neighbouring solutions, preventing itself from reproducing a test case that corresponds to an equilateral triangle from another. However, from the original test case that forms an equilateral triangle,

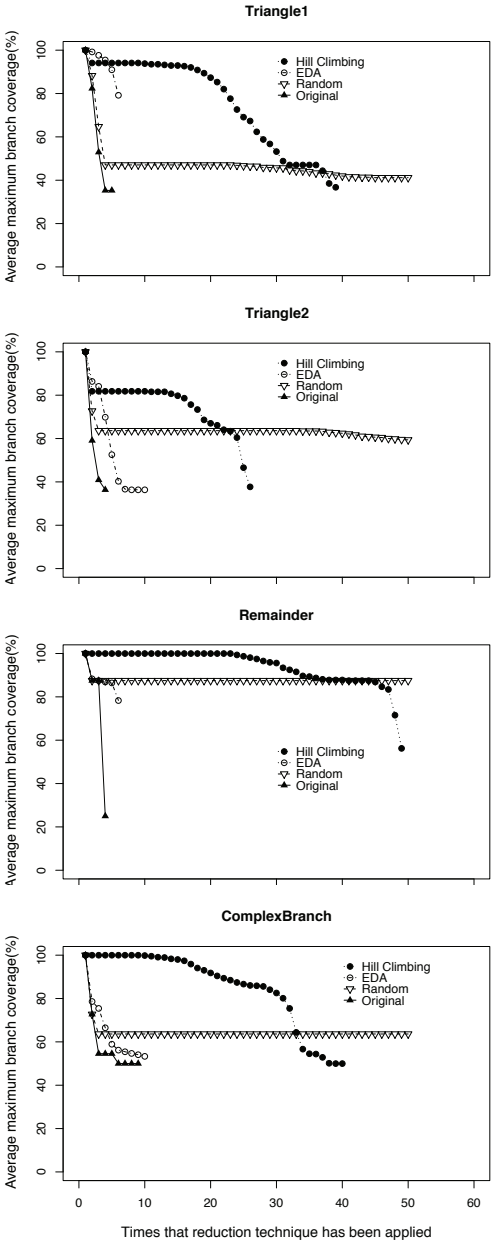

Figure 4. Results of latency enhancement. Each algorithm is given an identical budget of fitness evaluations for fair comparison of results. This budget allows the random search to produce suites that allow more reductions to be made for a given budget. However, as can be seen, these suites are not as latent as those produced by the Hill Climber. The EDA can sometime outperform both Hill Climber and EDA in the coverage achieved by the first few reductions, but it is the most expensive of the approaches, and the latency of its suites diminishes most rapidly.

EDA generates three Gaussian distributions with the same mean value, which results in high probability of generating another test case that forms an equilateral triangle.

The program `triangle2` contains, apart from the branch that determines an equilateral triangle, branches that determine whether the triangle is a right-angled triangle or not. Both algorithms will explore around the original test case (a, b, c) such that $a^2 + b^2 = c^2$. Neither the Gaussian distributions nor the neighbourhood definition used with the hill climbing algorithm is appropriate for finding an alternative test case that satisfies the condition.

The results for the program `remainder` are interesting because the random test data generation manages to maintain quality metric of 87.5% branch coverage. The remaining 12.5% branch coverage is controlled by two different predicates in the program that require either of the two input variable (a, b) to be equal to 0. The hill climbing algorithm shows the most successful performance because the neighbours are generated by changing a single input variable at a time, retaining the critical input value. On the other hand, the results from EDA show that the algorithm fails to retain the critical input value.

It is interesting to observe that the result of the random test data generation technique forms a flat plateau at different levels for different programs. This confirms findings of previous studies that show random test data generation can achieve a certain level of coverage, but cannot exceed this [12].

It should be also noted that, being a population-based evolutionary algorithm, EDA spends much larger amount of fitness evaluations in order to produce a single test case. This explains why the plotted lines for EDA are much shorter than those of other algorithms.

Table 3 shows the statistical analysis of the results shown in Figure 4. For each enhanced test suite s, the latency of enhanced test suites are evaluated by measuring the latency level $\lambda(coverage)(0.90)(greedy)s$, which means the largest number of times the additional greedy algorithm can be applied to the enhanced test suites before the branch coverage of the outcome of the additional greedy algorithm falls below 90%.

The λ levels of three algorithms are compared pair-wise using one-tailed Welch's t-test with the significance level of 95%. Welch's t-test is an adaptation of t-test for two samples having different variance values. It tests the null hypothesis that the means of two normally distributed groups are equal. In the context of the paper, the null hypothesis is that the λ levels achieved by different algorithms are are equal to each other. Three alternative hypothesis are formed as follows: H1. λ levels of EDA are greater than those of the random test data generation; H2. λ levels of the hill climbing algorithm are greater than those of the random test data generation; and H3. λ levels of the hill climbing algorithm are greater than those of EDA.

Program	$\bar{\lambda}_{random}$	σ_{random}	$\bar{\lambda}_{EDA}$	σ_{EDA}	$\bar{\lambda}_{HC}$	σ_{HC}	p_{H1}	p_{H2}	p_{H3}
triangle1	1.00	0.00	4.57	1.01	18.37	3.56	$<$ 2.2e-16	$<$ 2.2e-16	$<$ 2.2e-16
triangle2	1.00	0.00	1.00	0.00	1.00	0.00	N/A	N/A	N/A
remainder	1.00	0.00	1.13	0.35	32.60	3.89	$<$ 2.2e-16	0.0217	$<$ 2.2e-16
complexbranch	1.00	0.00	1.00	0.00	20.67	2.66	$<$ 2.2e-16	$<$ 2.2e-16	$<$ 2.2e-16

Table 3. Statistical analysis of latency enhancement strategy. The results of this analysis confirm statistically what can be seen visually in Figure 4; that the Hill Climbing is the most effective at enhancing the latency of test suites to which it was applied.

Apart from `triangle2` for which all three algorithms fail to increase λ above 1.0, the observed p-values for both H2 and H3 are significant at the 95% confidence level, confirming the alternative hypothesis that the λ levels of the hill climbing algorithm are higher than those of the random test data generation. The observed p-values for H1 for `triangle1` and `remainder` are also significant at the 95% confidence level, confirming the alternative hypothesis that the λ levels of EDA are higher than those of the random test data generation. Overall, the statistical analysis of the results suggests that the hill climbing algorithm is the most effective algorithm among the three algorithms studied. This answers **RQ4**.

6. Related Work

The existing literature on test case management for regression testing primarily concerns three major different techniques: test suite reduction, test case selection and test case prioritisation. Test case reduction (also called test suite minimisation) techniques aim to reduce the size of a given test suite by permanently eliminating some test cases from the test suite. There are mixed observations on whether the permanent reduction damages the fault-detecting capability of the test suite [16], [25]. In the context of the present paper, it is obvious that any permanent reduction of a given test suite will result in reduced latency.

Test case selection techniques focus on selecting a subset of the test suite that executes the modified part of the program. Naturally these techniques rely on structural information about the program under test such as symbolic execution [26], flow graph based [15] and dependence graph based approaches [2], [3]. A test case selection technique is said to be *safe* if the resulting subset of the test suite includes *all* test cases that are modification-traversing (i.e. execute the changed part of the program). Naturally, any *safe* test case selection technique leaves the test suite vulnerable to low latency because it exhausts the modification-traversing test cases with its first application to the test suite.

Test case prioritisation techniques prioritise test cases in an order that maximises earlier fault-detection. Since the fault-detection information is not known at the time of testing, it is often replaced with code coverage. Greedy approaches are known to produce efficient prioritisation results [5], [6], [11], [17] but other techniques including meta-heuristic search have also been applied [22], [23], [27].

The present paper combines test suite reduction with test data generation. Various meta-heuristic search techniques have been applied to test data generation, including local search [8], genetic algorithms [24] and estimation of distribution algorithms [20], [19]. However the existing literature on test data generation does not concern the use to which the generated test data is put. The strategy introduced in the present paper suggests that test data generation can be improved to consider not only the structural coverage, but other concerns in regression testing such as latency of test suites.

7. Conclusion And Future Work

The paper introduces the concept of latency in test suites, providing theoretical formulations and empirical results for latency measurements and enhancement. In order to enhance low latency of test suites, the paper introduces test data generation technique based on the exploration of the search space around existing selected test cases. The strategy combines test data generation techniques with test suite reduction techniques. The enhancement strategy study performed on benchmark programs shows that the proposed approach is capable of improving the latency of test suites. Future work will consider wider range of subject programs and other possible algorithms for test data generation.

Acknowledgements

The examples used in the empirical study in this paper are available from the Software-artifact Infrastructure Repository [4]. Shin Yoo is supported by the EPSRC SEBASE project (EP/D050863). Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest).

References

[1] *The Oxford English Dictionary*, 2nd ed. Oxford University Press, April 2000. [Online]. Available: http://dictionary.oed.com/cgi/entry/50130201

[2] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina: ACM Press, Jan.10–13, 1993, pp. 384–396.

[3] D. Binkley, "Reducing the cost of regression testing by semantics guided test case selection," in *ICSM '95: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 251–260.

[4] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[5] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *International Symposium on Software Testing and Analysis*. ACM Press, 2000, pp. 102–112.

[6] S. G. Elbaum, A. G. Malishevsky and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *International Conference on Software Engineering*, 2001, pp. 329–338.

[7] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *Proceedings of the 20th International Conference on Software Engineering*. IEEE Computer Society Press, Apr. 1998, pp. 188–197.

[8] M. Harman and P. McMinn, "A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press, 2007, pp. pp. 73–83.

[9] D. S. Johnson, "Approximation algorithms for combinatorial problems," in *Journal of Computer and System Sciences*, vol. 9, 1974, pp. 256 –278.

[10] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[11] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Department of Computer Science and Engineering, University of Nebraska-Lincoln, Techical Report, March 2006.

[12] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[13] C. Michael, G. McGraw, and M. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, no. 12, pp. 1085–1110, Dec. 2001.

[14] H. Mühlenbein and G. Paaß, "From recombination of genes to the estimation of distributions i. binary parameters," in *PPSN IV: Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*. London, UK: Springer-Verlag, 1996, pp. 178–187.

[15] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[16] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in in *Proceedings of International Conference on Software Maintenance*, 1998, pp. 34–43.

[17] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings; IEEE International Conference on Software Maintenance*. Los Alamitos, California, USA: IEEE Computer Society Press, 1999, pp. 179–188.

[18] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[19] R. Sagarna and J. A. Lozano, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms," *European Journal of Operational Research*, vol. 169, no. 2, pp. 392–412, 2006.

[20] R. Sagarna, A. Arcuri, and X. Yao, "Estimation of distribution algorithms for testing object oriented software," in *CEC'07: Proceedings of the IEEE Congress on Evolutionary Computation*, 2007, to appear.

[21] H. Sthamer, "The automatic generation of software test data using genetic algorithms," PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[22] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 123–133.

[23] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM Press, 2006, pp. 1–12.

[24] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, vol. 43, no. 14, pp. 841–854, 2001.

[25] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Software — Practice and Experience*, vol. 28, no. 4, pp. 347–369, 1998.

[26] S. S. Yau and Z. Kishimoto, "A method for revalidating modified programs in the maintenance phase," in *Proceedings of 11th International Computer Software and Applications Conference (COMPSAC '87)*, October 1987, pp. pp. 272–277.

[27] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007, pp. 140–150.

[28] S. Yoo and M. Harman, "Test data augmentation : generating new test data from existing test data (a version of which has been submitted to Software Testing, Verficiation and Reliability (STVR))," Department of Computer Science, King's College London, Technical Report TR-08-04, July 2008.