Test Data Regeneration : Generating New Test Data from Existing Test Data

Shin Yoo, Mark Harman

King's College London, Centre for Research on Evolution, Search & Testing, Strand, London, WC2R 2LS, UK

ABSTRACT

Existing automated test data generation techniques tend to start from scratch, implicitly assuming no pre-existing test data are available. However, this assumption may not always hold, and where it does not, there may be a missed opportunity; perhaps the pre-existing test cases could be used to assist the automated generation of additional test cases. This paper introduces search-based test data regeneration, a technique that can generate additional test data from existing test data using a meta-heuristic search algorithm. The proposed technique is compared to a widely studied test data generation approach in terms of both efficiency and effectiveness. The empirical evaluation shows that test data regeneration can be up to two orders of magnitude more efficient than existing test data generation techniques, while achieving comparable effectiveness in terms of structural coverage and mutation score.

1. INTRODUCTION

This paper introduces an optimising approach to automated software test data generation for structural testing. Unlike previous work, our approach assumes that *some* test data already exist. Where this assumption is reasonable, our approach is able to efficiently and effectively generate additional test data, using the pre-existing test data as a starting point. It can be argued that there are many scenarios in which there may be pre-existing test data, and that where there is, it makes good engineering sense to exploit it.

The idea of using existing test data in order to generate additional test data renders itself very well to search-based software testing, because most meta-heuristic algorithms that have been used for test data generation require one or more initial solutions from which to start the search. While the idea of test data regeneration may apply to other forms of automated test data generation, this paper focuses on search-based test data regeneration in order to introduce concrete algorithms for regeneration and to evaluate the proposed regeneration approach against existing approaches to automated 'from scratch' test data generation. The paper also focuses on structural test adequacy, though, once again, this focus serves merely to allow the introduction of concrete instantiations of the idea of test data regeneration; existing structurally adequate test data will be augmented with additional structurally adequate test data.

Like other forms of automated test data generation, current approaches to search-based test data generation start from a blank sheet of paper. That is, they assume that no test cases exist. Based on the execution of an instrumented version of the program under test, structural search-based testing seeks to find test cases that cover certain paths of interest (as measured by the instrumentation). The measurement of the difference between a path traversed and a desired path provide the fitness function value that is used to guide the search. As well as structural testing [11, 17, 25], search-based approaches have been applied to functional testing [16, 40, 43], and verification of non-functional properties [31, 45, 46]. However, in every case, the search starts from scratch, assuming no previous test data exist. Where there are already test data, these might be exploited with the potential to both reduce subsequent test data generation effort and to improve its effectiveness. It is this potential of existing test data that is explored in this paper.

There are many testing scenarios in which a tester may have available to them, pre–existing test cases, and where they may seek additional benefit from a subsequent automated test data generation activity. For example

1. The tester may have, perhaps somewhat painstakingly, generated some test data by hand, based on their experience, expertise and domain knowledge. No matter how good an automated technique might be, a human tester is likely to want to contribute some of their favourite 'bug revealing' test cases to the overall process. Furthermore, where there does exist such a pool of human generated test cases, with all the insight and experience they denote, it would surely make sense to make them available to any subsequent automated test data generation effort.

2. The tester may have some test cases available from the developer, who will typically have, at least, experimented with a few rudimentary 'sanity check' executions of the program before passing it to a more rigorous testing phase.

3. The tester could simply create test cases at random, choosing arbitrary inputs to provide an initial exploration of the system. Though such an approach, on its own, might be rather unsatisfactory, it could provide an initial pool of test data to be used by a subsequent and more intelligent test data generation technique.

4. The tester may have some regression test cases available from testing of a previous version of the program under test, or from testing of a similar component or system.

5. The tester may have a traditional 'from scratch' tool for automated test data generation. Such a tool may seek to meet some pre–defined adequacy criterion. No matter how well the tool performs, there is still the possibility that a subsequent phase might be applied, in which a different automated approach is adopted and which takes the pool of test data so–created as a starting point.

This paper introduces test data regeneration, a technique that generates test data by exploiting the existing test data. It differs from the existing work called test data *augmentation* which refers to the problem of generating additional test cases that will test the newly added features of the System Under Test (SUT) [14]. Test data regeneration, however, represents the process of creating additional test data using the knowledge of existing test data (i.e. *regenerate*), in order to improve the fault detection capability of a test suite, without assuming any change to SUT. But why does one want to generate additional test cases if there already exists a test suite? There can be various reasons. Since testing can only detect existence of faults, not lack of them, executing additional test cases can only increase the confidence in the program under test. Empirical research in test case management techniques has also shown that it is desirable to have certain level of redundancy in testing [33–35]. Additional test suites can be also beneficial for verification of fault fixes or statistical estimation of program reliability.

The proposed test data regeneration technique uses meta-heuristic search to search for a test case that behaves in the same way as, but with different test input from, the original test case. This is achieved by applying a series of a pre-defined set of modification operators to the original test case. Each modification operator applies some changes to the original test case while preserving a certain aspect of the original test case. The new test case is included in the new test suite if it shows the same behaviour as the original test case. Here the same behaviour means the same contribution to test objectives. For example, for structural testing the new and modified test case should also achieve the same coverage as the original test case.

This paper introduces a novel search-based test data regeneration algorithm and presents the results of an empirical comparison with one of the state-of-art test data generation techniques. Both the efficiency and the effectiveness of two techniques are evaluated. The results are promising; savings in the cost as large as two orders of magnitude can be observed. With a suitable configuration, test data regeneration can generate an additional test suite that achieves the same structural coverage as the original test suite. Furthermore, having the new test suites generated by test data regeneration technique proves to be beneficial both quantitatively and qualitatively. First, it tends to allow for higher mutation score, showing the benefits of having additional test suites. More interestingly, the mutation faults detected by test data regeneration technique are different from those that are detected by the original test suite and the ones generated by a state-of-art search-based test data generation technique; the details of the results from mutation testing are discussed in Secton 6.3.

The primary contributions of the paper can be summarised as follows:

1. The paper introduces a novel approach to test data generation, which is based on the idea of using existing test data as a starting point for further automatic test data generation.

2. The paper presents an empirical evaluation of the proposed approach by comparing it to an established search-based test data generation tool.

    (a) The proposed approach is compared to search-based testing (SBT), an existing test data generation technique that has recently received a lot of attention. Efficiency is measured by counting the number of fitness evaluations required for the generation of a test suite. The empirical results show that the proposed approach can be up to two orders of magnitude more efficient than SBT.

    (b) The proposed approach is compared to SBT in terms of effectiveness, measured by structural coverage. The empirical results show that, with an appropriate configuration, the proposed approach can achieve higher structural coverage compared to SBT.

    (c) The proposed approach is also compared to SBT in terms of effectiveness, measured by structural mutation score. The empirical results show that the proposed approach can achieve competent level of mutation score compared to SBT at much lower cost.

3. The paper also presents a case study of real world subjects that require more sophisticated test input. The results show that the proposed approach can still be more effective and efficient for real world subjects when compared to SBT.

The rest of the paper is organised as follows. Section 2 discusses background material on test data generation. Section 3 presents the motivation for test data regeneration and the research questions. Section 4 illustrates the proposed search-based test data regeneration technique in detail. Section 5 describes experimental setup used for the empirical study. Section 6 presents the empirical evaluation of the proposed technique. Section 7 interprets the empirical results. Section 8.6 presents a case study with real world subjects. Section 9 discusses the potential threats to validity. Finally, Section 10 concludes and lists directions for future work.

2. BACKGROUND

Automatic test data generation using meta-heuristic search techniques has been a productive area of research in recent years. Manual test data generation, still widely used in the industry, is very costly and laborious. The application of meta-heuristic search techniques to test data generation presents a promising possibility of automating the process. Various meta-heuristics including hill climbing, simulated annealing and genetic algorithms have been applied to testing problems with objectives such as structural coverage, specification-based testing, and non-functional properties [23].

```
(1)    read(x)
(2)    read(y)
(3)    if x == y then print x
(4)               else print x - y
```

Search-based test data generation is a form of dynamic testing. The program under test has to be instrumented according to the test objectives, e.g. structural coverage or execution time. The meta-heuristic search aims to find test data that meets the testing objectives by executing the instrumented program for the evaluation of the fitness of a candidate solution. The combination of program execution and search algorithm dates back to Miller and Spooner [27]. They produced a *straight* version of the program under test for each path, converting predicates to constraints. Solving these constraints produces a test input that executes the chosen path. Later, Korel extended the idea for Pascal programs using a hill climbing local search algorithm [18].

Korel introduced a concept called *branch distance*. Branch distance of a predicate measures how close it is to being evaluated as `true`. Once instrumented for branch distance, a branch can be resolved in a way that is desirable for the test objective by optimising the distance. For example, suppose that the tester wants to execute the `true` branch of the predicate in line (3) of the following program. The branch distance for the predicate `x == y` is calculated as $|x - y|$. This forms the fitness function for a test input. That is, any test input to the program, $t = (x, y)$, is evaluated using $f(t) = |x - y|$ in order to reach the `true` branch in line (3). It is then possible to apply a variety of meta-heuristic optimisation techniques to minimise or maximise the fitness function. In the example program, minimising the fitness function will produce a test input $t = (x, y)$ such that $|x - y| = 0$, i.e., the one that will execute the `true` branch.

The branch distance concept is a generic approach to structural testing and can be used with other meta-heuristic search techniques such as simulated annealing [41,42], genetic algorithms [1,5,26,30,44], Estimation of Distribution Algorithm (EDA) and scatter search [36,37]. Search-based test data generation has been applied to testing of non-functional properties such as worst-case execution time [31,45] and dynamic memory consumption [20].

Several techniques were developed in order to overcome problems in search-based test data generation, e.g., large search space and challenging search landscape. Input domain reduction tries to reduce the size of the search space by putting constraints to test input using symbolic execution [28], or slicing the program under test so that irrelevant test inputs are removed [12]. Testability transformation tries to transform search landscape into one that is friendlier to automated test data generation [15,19]. For example, removing a flag variable can transform a large plateau into a landscape with gradient. Test data are generated from the transformed program, but applied to the original program to test it.

The idea of using existing test cases for the generation of new ones is not entirely new. Similar ideas have appeared previously in the literature. Baudry et al. presented the bacteriologic algorithm for test case optimisation [2–4]. The bacteriologic algorithm applies a series of mutations on an initial test suite, and incrementally evolves a test suite that is deemed to be superior to the original one in terms of mutation score. The algorithm is initialised with an existing test suite, in a manner similar to the test data regeneration technique proposed in this paper. Tlili et al. improved Evolutionary Real-Time Testing (ERTT) by seeding the ERTT algorithm for the Worst-Case Execution Time analysis with a test suite that achieves high structural coverage [39]. Without the seeding approach, some parts of the source code were never executed during the search. The seeding approach helped the ERTT algorithm to generate more reliable WCET solutions that execute the larger parts of the program under test. The differences between these algorithms and the technique proposed in this paper will be discussed in Section 4.5.

3. PROBLEM STATEMENT

3.1 Motivations for Search-based Test Data Regeneration

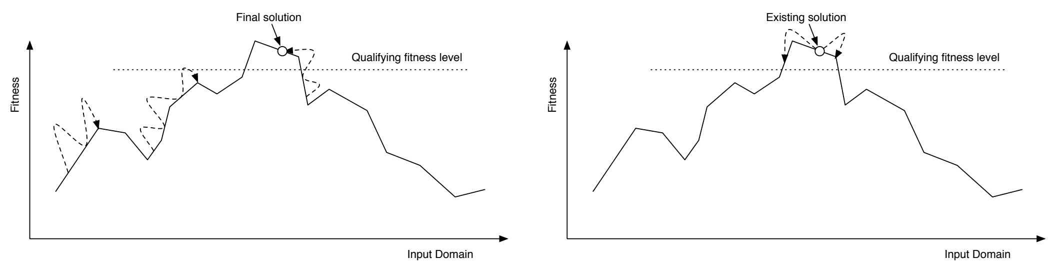


Figure 1: The left image shows how the traditional search-based test data generation techniques work. Meta-heuristic search techniques start from one or more random solutions, and try to obtain a qualifying solution by optimising the fitness function. The right image shows how the test data regeneration works. Since a qualifying solution is already known, meta-heuristic search is applied to explore regions of the search space that are close to the qualifying solution.

This paper presents a search-based test data generation that uses the knowledge of existing test data. The proposed approach is called test data *regeneration*; it aims to improve an existing test suite with alternative test data derived from those already available. In many cases, it is not unrealistic to assume that a tester in a software organisation already has some test data. Why does one want more test data when there already exist some? Since testing can only reveal the existence of a fault, not the lack of it, repeated testing can only raise the confidence in the correctness of the program under test. Having a low-cost method of generating additional test data from existing data can be beneficial in many ways. It may prevent the program under test becoming over-fitted to existing set of test data. It may be helpful when the tester wants to gain statistical confidence in the correctness of the program behaviour.

The knowledge of existing test data can make the generation of the new test data less expensive. Consider the simplified visualisation of a fitness landscape in Figure 1. The x-axis represents all the possible test data in the input domain; the y-axis depicts the possible fitness values for some unspecified test objective. The image in the left shows how the traditional search-based test data generation techniques work; they start with a random test data and try to find a qualifying solution by optimising on the fitness function. The image on the right shows how one can exploit the knowledge of existing test data to generate additional test data. Assuming that there exist a small *window* of qualifying test data around the one that has already been found, it can be expected to find additional test data, near the existing one, that also qualify.

A meta-heuristic search similar to the one depicted on the right side of Figure 1 has two major advantages compared to other meta-heuristic optimisation. The shape of search landscape becomes less problematic since the search starts from a point with sufficient fitness value. Similarly, the size of search landscape becomes less problematic since the search only needs to explore the regions of search space that are close to the existing solution.

Depending on the program under test and the relevant test objective, there may or may not exist a 'window of qualification' as described here. The definition of being "near" to the existing test data also plays an important role in deciding whether such a window of qualification exists. In general, there is no particular reason to believe that it always exists. For instance, a counter example can be found by considering the predicate `x == 0`. Only the test data that has 0 as the value of `x` can qualify if one is trying to make the predicate `true`. Therefore, the paper presents the proposed approach as a complimentary alternative to existing techniques, not as a replacement, and aims to observe the trade-off between two different approaches. Indeed, the proposed approach to test data generation assumes the existence of a known set of test data, which has to be generated using some other techniques including manual test data generation.

This paper presents a search-based test data regeneration technique, which formulates test data regeneration as a search *around* the existing test data. In order to perform search-based test data regeneration, the following elements need to be defined:

- **Neighbourhood:** in order to search *near* the existing test data, it has to be defined what constitute the near-neighbours that the algorithm will search through.

- **Search Radius:** given a definition of near-neighbours, how far from the existing test data should the algorithm search?

Once the definition of near-neighbours and the search distance is fixed, the proposed search-based test data regeneration proceeds following the algorithm outlined in Algorithm 1. While Algorithm 1 is based on a local search technique, the same idea can be applied to other meta-heuristic search techniques.

Algorithm 1: Outline of search-based test data regeneration

```
(1)    $currentSol = existingSol$
(2)    while within the search radius
(3)        if near neighbours of $currentSol$ contains a qualifying solution $newSol$
(4)            currentSol = newSol
(5)        else
(6)            break
(7)    return currentSol
```

3.2 Problem Statement

This paper evaluates test data regeneration in terms of efficiency and effectiveness across different input domain, with comparison to Iguana, a state-of-the-art tool for search-based test data generation based on a variation of the hill climbing algorithm [24]. The first research question concerns the efficiency of the proposed technique. Efficiency is measured by counting the number of times that the program under test has to be executed in order to generate a set of test data that meets test objectives.

RQ1. Efficiency : How efficient is test data regeneration, compared to search-based test data generation, across different input domains?

The second and third research questions concern the effectiveness of test data regeneration, in terms of branch coverage and mutation score respectively.

RQ2. Coverage: How much branch coverage does test data regeneration technique achieve?

RQ3. Mutation Faults: How high a mutation score does test data regeneration technique achieve?

Finally, the definition of neighbouring solutions, the search radius, and the size of the input domain are parameterised and their impact on efficiency and effectiveness of the proposed approach is observed.

RQ4. Settings: What variance in effectiveness and efficiency is observed for different set-up of neighbourhood and the search radius?

4. SEARCH-BASED TEST DATA REGENERATION

For the introduction of the proposed test data regeneration technique and the first empirical study, the discussion will be confined to the test data regeneration of numerical test input composed as a vector of integers. This allows clearer and more intuitive definitions. However, the proposed approach can be applied to other types of test input provided that suitable definitions of 'nearness' and 'search radius' can be constructed. The case study in Section 8 presents how search-based test data regeneration technique can be applied to programs that require more sophisticated input type such as strings and arrays.

The proposed search-based test data regeneration is based on the hill climbing algorithm, but there is one important difference between the proposed technique and the 'normal' hill climbing algorithm (and test data generation techniques that are based on hill climbing). The hill climbing algorithms adopt random restart in order to escape local optima; test data regeneration assumes that the existing test data belong to global optima, and, therefore, always starts from a global optimum that corresponds to the existing test data. Note that there may exist more than one globally optimal solution in a search-based test data generation problem. For example, suppose the goal of a given search-based test data generation problem is to cover the `true` branch of the predicate `x > 3`. Any value of `x` that is greater than 3 will cover the branch, and therefore will qualify as a globally optimal solution.

Since the lack of randomness can inhibit the variety of resulting solutions, the second difference is introduced. In hill climbing algorithms, the move towards the next candidate solution (climb) can be made in a few different ways including *first-ascent*, where the algorithm moves to the first neighbouring solution that has higher fitness than the current solution, and *steepest-ascent*, where the algorithm moves to the neighbouring solution with highest fitness. However, both approaches behave deterministically when the starting point is fixed. Therefore, the proposed test data regeneration adopts *random-first-ascent*, where the algorithm examines the neighbouring solutions in a random order and moves to the first neighbouring solution with a higher fitness than the current solution.

4.1 Neighbouring Solutions And Interaction Level

In search-based test data generation, neighbouring solutions of a given solution are defined as the solutions that have the minimal difference from the original. For example, for integer input domain, neighbouring solution of an integer i is often defined to be $\{i-1, i+1\}$. This paper formulates the process of obtaining neighbouring solutions as an application of a set of modification operators, which is a natural extension of existing methods.

This paper utilises four modification operators. Following Korel, the first and second modification operators are defined as

$\lambda x.x+1$ and $\lambda x.x-1$ [18]. When applied to integer values, these two operators generate test inputs that are *shifted* from the original. The third and the fourth modification operators are $\lambda x.x*2$ and $\lambda x.\lceil x/2 \rceil$. These operators generate test inputs that are *scaled* from the original.

Finally, *interaction level* is defined as the number of variables that are modified at the same time. The concept of 'interaction' is borrowed from interaction testing, which is a test for interaction of different configurations [7,32]. If a modification operator is applied to a test input vector of size m with interaction level of i, it means all $\begin{pmatrix} m \\ i \end{pmatrix}$ possible combinations of variables will be modified by the operator at the same time, resulting in a neighbouring solution set of size no bigger than $\begin{pmatrix} m \\ i \end{pmatrix}$.

For example, suppose that an integer input vector (2, 3, 1) is modified with the described modification operators using interaction level of 2. With $\lambda x.x+1$ the resulting input vectors will be $\{(3,4,1),(3,3,2),(2,4,2)\}$. With $\lambda x.x-1$ they will be $\{(1,2,1),(1,3,0),(2,2,0)\}$. Similarly, $\lambda x.x*2$ and $\lambda x.\lceil x/2 \rceil$ result in $\{(4,6,1),(4,3,2),(2,6,2)\}$ and $\{(1,1,1),(1,3,0),(2,1,0)\}$ respectively. The final neighbouring solution set is the union of four sets shown below.

$$\{(3,4,1),(3,3,2),(2,4,3),(1,2,1),(1,3,0),(2,2,0),(4,6,1),(4,3,2),(2,6,2),(1,1,1),(2,1,0)\}$$

Modification operators and interaction level allows flexible approach towards the generation of neighbouring solutions. In existing search-based test data generation, the neighbouring solutions are explored to improve the fitness value. In test data regeneration, neighbouring solutions are generated from an existing solutions that already qualifies for the given objective. Therefore, the neighbouring solutions should aim to *preserve* certain properties found in the original solution. For example, consider a predicate `x * 2 = y` and a test input $(x,y)=(3,6)$. Applying $\lambda x.x+1$ and $\lambda x.x-1$ with any interaction level results in different resolution of the predicate. However, applying $\lambda x.x*2$ with interaction level of 2 results in $(x,y)=(6,12)$, which still resolves the predicate as `true`.

From the combined use of modification operators and interaction level, an analogy can be drawn between the test data regeneration technique and mutation testing. In mutation testing, the program under test is mutated to simulate actual faults. In test data regeneration, the processed is reversed by modifying the test cases rather than the program. The constraint for mutation testing is that the mutants should be executable, whereas the constraint for test data regeneration is that the new test case remains an eligible test case for the program.

4.2 Search Radius

Search radius is defined as the upper limit to the number of modifications that can be applied to the original test case. The number of modifications has to have an upper limit so that the cost of the search process can be controlled. For example, suppose one is modifying a test input vector $(x,y)=(2,1)$ in order to resolve the predicate $x>y$ to `true`, with interaction level of 1. A successful chain of modifications might, for example, consist of applying $\lambda x.x+1$ repeatedly to x until the value of x reaches the maximum possible value within the input domain, MAX. The algorithm will evaluate every candidate solution in $\{(x,1)|2 \le x \le MAX\}$ by executing the program under test $MAX-1$ times. This may be too costly. By having an upper limit to the number of modifications, it is possible to control the size of the search area, and thereby the cost of the search.

However, within the predefined the search radius, the proposed test data regeneration technique is encouraged to move away from the original test data as far as possible. The intuitive underlying assumption is that, when the program under test has already been tested with the existing set of test data, the set of additional test data is most valuable when it is as different from the original set as possible, while still fulfilling the criteria. This is called the *distance-value* assumption.

4.3 Fitness Function

The proposed search-based test data regeneration technique uses a novel fitness function for test data generation. Since it assumes that an existing test input that meets the test objective is available, the new fitness function is designed to be maximised so as not to decrease the fitness below the qualifying level, rather than to increase the fitness above the qualifying level.

Let t be the individual test input, the fitness of which is being measured, and t' be the original test input known to meet the test objective. Let μ be the measurement of quality of testing in terms of meeting the test objective. Let $\Delta_\mu(t,t')$ be the difference in quality metric between two test inputs, and $\Delta_d(t,t')$ be the distance between the input vectors of two test cases.

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

$$\Delta_d(t,t') = \text{the distance between } t \text{ and } t'$$

This paper utilises the Euclidian distance between numerical vectors for Δ_d. It can be difficult to measure the difference in μ quantitatively; this paper utilises the hamming distance between two binary strings that represent the branch coverage record of t and t' for Δ_μ. Hamming distance is the number of bits that need to be flipped in order to transform one binary string into another. Here, lower hamming distance means that t and t' covers similar branches in the program under test. Without losing generality, the fitness function can be defined to be maximised, i.e. higher fitness values are better. Based on these, the fitness value of the test case t, $f(t)$, is defined as follows:

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

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 positive fitness value, guiding the search towards t. Because of $\Delta_d(t,t')$, the search is encouraged to move farther away from t'. Finally, if t is different from t' but has lower quality metric, t is guaranteed to receive a negative fitness value, thereby ensuring that the search never escapes the window of qualification.

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 inputs with positive fitness values.

Algorithm 2: Test data regeneration algorithm
Input: A test suite containing existing test inputs, S, the fitness function, f, search radius, r, and a set of modification operators, M, and the interaction level, i
Output: A new set of test inputs, S'

```
(1)   $S' \leftarrow \{\}$
(2)   foreach $t \in S$
(3)       $finalSol \leftarrow null$
(4)       $currentSol \leftarrow t$
(5)       $count \leftarrow 0$
(6)       while $count < r$
(7)           $nextSol \leftarrow null$
(8)           $N \leftarrow$ GENERATENEIGHBOURS($M, currentSol$)
(9)           while true
(10)              remove a randomly selected neighbour, $n$, from $N$
(11)              if $f(n) > f(currentSol)$
(12)                  $nextSol \leftarrow n$
(13)                  break
(14)              if $size(N) == 0$
(15)                  $nextSol \leftarrow null$
(16)                  break
(17)          if $nextSol == null$
(18)              break
(19)          else
(20)              $currentSol \leftarrow nextSol$
(21)              $count \leftarrow count + 1$
(22)      if $finalSol \neq t$
(23)          $S' \leftarrow S' \cup \{finalSol\}$
(24)  return $S'$
```

4.4 Algorithm

The pseudo-code of test data regeneration algorithm used in this paper is shown in Algorithm 2. The main loop in line (2) iterates over each existing test input in the given test suite, S. After initialisation in lines from (3) to (5), the inner loop in line (6) initiates the search process. In line (8), the algorithm generates the set of neighbouring solutions, N, by calling GenerateNeighbours($M, currentSol$). The loop in line (9) repeats until there is no neighbouring solution in N. If the algorithm finds a neighbouring solution with higher fitness value than *currentSol*, *finalSol* is updated and the algorithm exits the loop. If N runs out of solutions, *nextSol* is marked as *null* and the loop in line (8) exits. If *nextSol* equals *null* in line (17), it means that none of the neighbouring solutions in N had higher fitness value than *currentSol*, in which case the loop in line (6) exits even if $count \geq r$. Otherwise, *currentSol* is updated with the non-*null* *nextSol* and next move begins. The loop in line (6) will eventually exit when *count* becomes equal to r. Finally, if the algorithm has found *finalSol* that is not *null*, it is added to the new test suite, S'.

```
Algorithm 3: GenerateNeighbours() subroutine
Input: An existing test input vector, $t$, and a set of modification operators, $M$, and the interaction level, $i$
Output: A set of neighbouring solutions, $N$
(1)  $N \leftarrow \{\}$
(2)  foreach operator $op$ in $M$
(3)      foreach each combination $C$ of $i$ variables out of $t$
(4)          $newNeighbour \leftarrow copy(t)$
(5)          foreach variable $v$ in $C$
(6)              update $newNeighbour$ by replacing the value of $v$ with $op(v)$
(7)          $N \leftarrow N \cup \{newNeighbour\}$
(8)  return N
```

The psuedo-code of the subroutine that generates neighbouring solutions is shown in Algorithm 3. The loop in line (2) iterates over all modification operators available. Each of these operators is applied to $\binom{m}{i}$ combinations of input variables, which are stored in *newNeighbour*. The variables not in the given combination remain the same as t since *newNeighbour* is initialised with t in line (4).

A precise formulation of average computational complexity for the algorithm is problematic, due to the inherent probabilistic nature of the algorithm. For example, it is not always possible to determine the probability of a neighbouring solution having a higher fitness value than the current solution. However, it is possible to obtain the exact upper bound for the worst-case execution time. Let S be the existing test suite, which contains test cases that are input vectors of size m. Let M be the set of modification operators, i the interaction level, and r the search radius. For each test case in S, the algorithm considers up to $\binom{m}{i}$ neighbouring solutions, which can be repeated for r times at maximum. For each consideration, $|M|$ operators are applied. Therefore, the upper bound for the worst-case execution time is calculated as following:

$$O(|S| \cdot \binom{m}{i} \cdot |M| \cdot r)$$

4.5 Differences to Existing Techniques

The idea of initialising a meta-heuristic algorithm with a set of known solutions is shared between the search-based test data regeneration technique and other existing techniques such as the bacteriologic algorithm by Baudry et al. [2–4] or the seeded Evolutionary Real-Time Testing [39]. However, these algorithms differ from the proposed technique in one important aspect, which is preservation of the behaviour of the original test data. The bacteriologic algorithm does not consider the behaviour of individual test case, because its fitness function evaluates test cases only collectively. With the proposed test data regeneration technique, it is possible to generate a specific test cases that follows the exactly same execution trace of the original test case while still being different from the original. The seeded ERTT algorithm does not preserve the behaviour of individual test cases, because it uses one property (structural coverage) as a starting point to evolve and improve another (WCET). Unlike the test data regeneration technique, therefore, the resulting test data are not guaranteed to achieve the same level of structural coverage.

Another difference is how these two algorithms generate candidate solutions. The bacteriologic algorithm only uses a syntax tree based mutation, and the seeded ERTT uses selection, crossover, and mutation operators. On the other hand, the test data regeneration technique uses a set of modification operators, which allows for more flexibility to tailor the regeneration process for the semantics of the program under test.

The overall approach advocated in this paper is also very general, and is not constrained to any particular algorithm. That is, the paper introduces the idea that testing does not start, *ab initio*, with no existing test data. There will often be some existing test inputs and it may make sense to start the test data generation process from any such initial cases.

5. EXPERIMENTAL DESIGN

This section sets out the experimental design for the empirical study that evaluates the proposed test data regeneration technique.

5.1 Iguana : Hill Climbing Test Data Generation

The proposed test data regeneration technique is compared to a well known search-based test data generation tool called Iguana (hereby referred to IG) [24]. IG uses an advanced version of hill climbing called *alternating variables method.* Each input variable in the test input is taken and adjusted, while other variables are kept constant. First the algorithm performs what is called an *exploratory* phase, in which the algorithm makes a probe movement to neighbouring solutions. If one of the probe movements turns out to be successful, i.e. produces higher fitness, then the algorithm enters what is called a *pattern* phase. In the pattern phase, the previous move is repeated in the same direction (adding or subtracting to the same variable) while doubling the distance in each iteration. If such a move produces a solution with lower fitness than current solution, then the algorithm switches back to exploratory phase with the next input variable. This method ensures that the algorithm reaches local or global optima quickly.

Mutation Operator	Description
AORB	Replace basic binary arithmetic ops.
AORS	Replace short-cut binary arithmetic ops.
AOIU	Insert basic arithmetic ops.
AOIS	Insert short-cut arithmetic ops.
AODU	Delete basic unary arithmetic ops.
AODS	Delete short-cut arithmetic ops.
ROR	Replace relational ops.
COR	Replace conditional ops.
COD	Delete unary conditional ops.
COI	Insert unary conditional ops.
LOR	Replace unary logic ops.
LOI	Insert unary logic ops.
LOD	Delete unary logic ops.
ASRS	Replace short-cut assignment ops.

Table 1: Mutation operators used in the paper

IG produces test suites that achieve branch coverage, i.e. every branch in the program under test is evaluated to both `true` and `false`. Given a program under test, it targets each branch in the program sequentially and tries to generate a single test input that executes the branch. If it cannot generate a test input within set number of fitness evaluations, the search is terminated. This paper utilise the default maximum fitness evaluation setting, which is to spend 50,000 fitness evaluations maximum per branch.

It should be noted that IG also contains test data generation toolkits that are based on genetic algorithms. However, since the proposed test data regeneration is based on a hill climb algorithm, the comparison is made only to the hill climb based test data generation toolkit of IG.

5.2 Subject Programs

A set of well-known benchmark programs for structural test data generation techniques is used: two versions of the `triangle` program, `remainder`, and `complexbranch`. Each program contains from 18 to 26 branches, which, though small, provides non-trivial branch coverage possibilities. These programs take 2 to 6 input variables. `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. It is also the example program shown in Algorithm 4. `Triangle1` is used by Michael and McGraw in their study of test data generation [26]. `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 the two integers input [38]. Finally, `complexbranch` is a program specifically created as a challenge for test data generation techniques [44]. It contains several branches that are known to be hard to cover.

One of the subject programs, `remainder`, constantly caused IG to consume large enough memory to halt with 32bit integer input domain. This is not a weakness of search-based test data generation, but rather an outcome of internal design decisions within IG. As a result, experiments regarding `remainder` do not use 32bit integer input domain and only use 8bit and 16bit integer domain.

5.3 Input Domain

Since one of the restrictions inherent in existing test data generation techniques is the restriction in size of the input domain, the proposed test data regeneration technique is tested against different input domain sizes to see if it can cope better with a significantly large input domain than traditional techniques. Both IG and the proposed technique were executed for three different input domain sizes : 8bit integers, [-128, 127], 16bit integers, [-32768, 32767], and 32bit integers, [-2147483648, 2147483647]. Combined with the number of input variables for the subject programs, this results in search spaces ranging in sizes from 2^{24} to 2^{192}. Both techniques are evaluated in terms of efficiency and effectiveness against different input domains.

5.4 Original Test Suites and Mutation Faults

The initial test suites used by the test data regeneration technique have been manually generated for each subject program and each input domain so that 100% branch coverage is achieved. For each branch in the subject programs, a single test input was generated to make the predicates both `true` and `false`. This is standard practice in search-based test data generation [23].

Apart from branch coverage, mutation score was used as a measure of effectiveness. Mutation faults are based on the notion of mutation testing, where the adequacy of test cases is evaluated by introducing simple syntactic modifications to the program [6]. If a test case reveals this modification, it *kills* the mutant program. The mutation score of a test case is the total number of mutation faults that the test case has killed. A test case with higher mutation score is assumed to have a higher chance of detecting real faults, which is observed in several empirical studies of mutation testing for procedural languages [10, 29].

A well known mutation testing tool, `muJava`, was applied to the subject programs [22]. The types of mutation operators used in the paper are described in Table 1. Application of these mutation operators produced 203, 241, 324, and 499 mutants for `triangle1`, `triangle2`, `remainder`, and `complexbranch` respectively. Sets of test data generated by different techniques were analysed against these mutants.

Subject Programs	Input Domain	$\bar{n}_{IG}$	$\sigma^2_{n_{IG}}$	$\bar{n}_{TR}$	$\sigma^2_{n_{TR}}$	p-value
`triangle1`	8bit	16030.00	110408634.00	531.30	1618.54	$< 10^{-6}$
`triangle1`	16bit	373900.00	2043159037.00	525.30	766.85	$< 10^{-6}$
`triangle1`	32bit	452000.00	10559180.00	585.40	2132.99	$< 10^{-6}$
`triangle2`	8bit	9008.00	25917844.00	497.35	731.71	$< 10^{-6}$
`triangle2`	16bit	129200.00	695019728.00	769.10	769.10	$< 10^{-6}$
`triangle2`	32bit	158700.00	10490236.00	679.85	755.61	$< 10^{-6}$
`remainder`	8bit	196.30	1833.06	302.50	197.00	1.0
`remainder`	16bit	342.80	1119.36	295.55	163.94	0.0002
`complexbranch`	8bit	894.40	72117.63	600.60	1918.15	0.0006
`complexbranch`	16bit	2053.00	380509.10	584.30	1219.27	$< 10^{-6}$
`complexbranch`	32bit	6564.00	10241196.00	886.40	5452.04	$< 10^{-6}$

Table 2: Average number of fitness evaluations ($\bar{n}$) and variance (σ_n) for TR and IG.

It should be noted that the mutation faults studied in the paper may contain equivalent mutants, i.e. mutants that are semantically identical to the original program. The equivalent mutants raise serious problems for mutation testing. However, since equivalent mutants cannot be killed by the original test suite nor enhanced test suites, they do not affect the validity of the findings of the current experiment; their existence can only strengthen the null hypothesis.

5.5 Evaluations

To cater for the inherent randomness in both techniques, each individual experiment was repeated 20 times. By default, the interaction level was set to 1, similar to the alternating variable method, while the search radius was restricted to 10. **RQ1** is answered by comparing the average number of fitness evaluations required for the generation of a new set of test data between the two techniques across different input domains. **RQ2** and **RQ3** are answered by comparing the average branch coverage and average mutation score between the two techniques across different input domains.

The inherent difference between the nature of two techniques, combined with the need for parameter tuning, presents a challenge to formulate a framework of comparison based on a single metric for efficiency and effectiveness. To perform an unbiased comparison, both techniques are applied to the subject programs without any additional parameter tuning that would be required to make per-effort effectiveness or per-goal efficiency measurement. Instead, the results from the basic configurations are presented with detailed analysis of when and why a technique outperforms the other.

RQ4 is answered by comparing efficiency and effectiveness measurements of the test data regeneration technique against itself using different settings. In the first set of experiments for **RQ4**, the interaction level is changed from 1 to the size of the input vector, while keeping other experimental factors constant, and observe effectiveness and efficiency. In the second set of experiments for **RQ4**, the search radius is changed from 1 to 10, while keeping other experimental factors constant, and make the same observation.

6. RESULTS AND ANALYSIS

6.1 Efficiency Evaluation

Figure 2 shows the efficiency measurement of the IG test data generation technique and test data regeneration technique (hereby denoted as TR) against the subject programs. The x-axis represents different input domains. The y-axis, which is in logarithmic scale, represents average number of fitness evaluations required for the generation of a new set of test data for each program. In general, test data regeneration is not much worse than IG (`remainder`), or much more efficient than IG `triangle1`, `triangle2`, and `complexbranch`). With `triangle1` and `triangle2`, the gain in efficiency is more than two orders of magnitude. More importantly, the gain increases as the input domain grows larger. While the test data regeneration technique shows relatively constant level of efficiency, test data generation technique does not cope well with larger input domains.

Table 2 presents the statistics observed in Figure 2 in more detail with statistical testing. Since there is no evidence to believe that the number of fitness evaluations required has a normal distribution, one-sided sign-test is used, which does not make assumptions about the probabilistic distribution of the population. The null hypothesis is that $\bar{n}_{IG}$ and $\bar{n}_{TR}$ has the same value. The alternative hypothesis is that $\bar{n}_{IG}$ is greater than $\bar{n}_{TR}$. The confidence level is 95%. For all cases except `remainder` with 8bit input domain, the resulting p-values indicates significant results. Therefore, the alternative hypotheses is accepted for these cases. However, the alternative hypothesis is rejected for `remainder` with 8bit input domain. Therefore, the null hypothesis is accepted. Overall, this answers **RQ1** positively. The statistical analysis shows that there is a significant gain in efficiency for most cases when test data regeneration is used. The amount of gain in efficiency can be as large as two orders of magnitude.

It should also be noted that the cost of TR tends to be consistent across different input domains, whereas the cost of IG tends to increase as the input domain grows larger. This is due to the fact that IG always has to cope with the whole input domain, starting with a random solution, whereas TR can focus on the small region around the original solution defined by

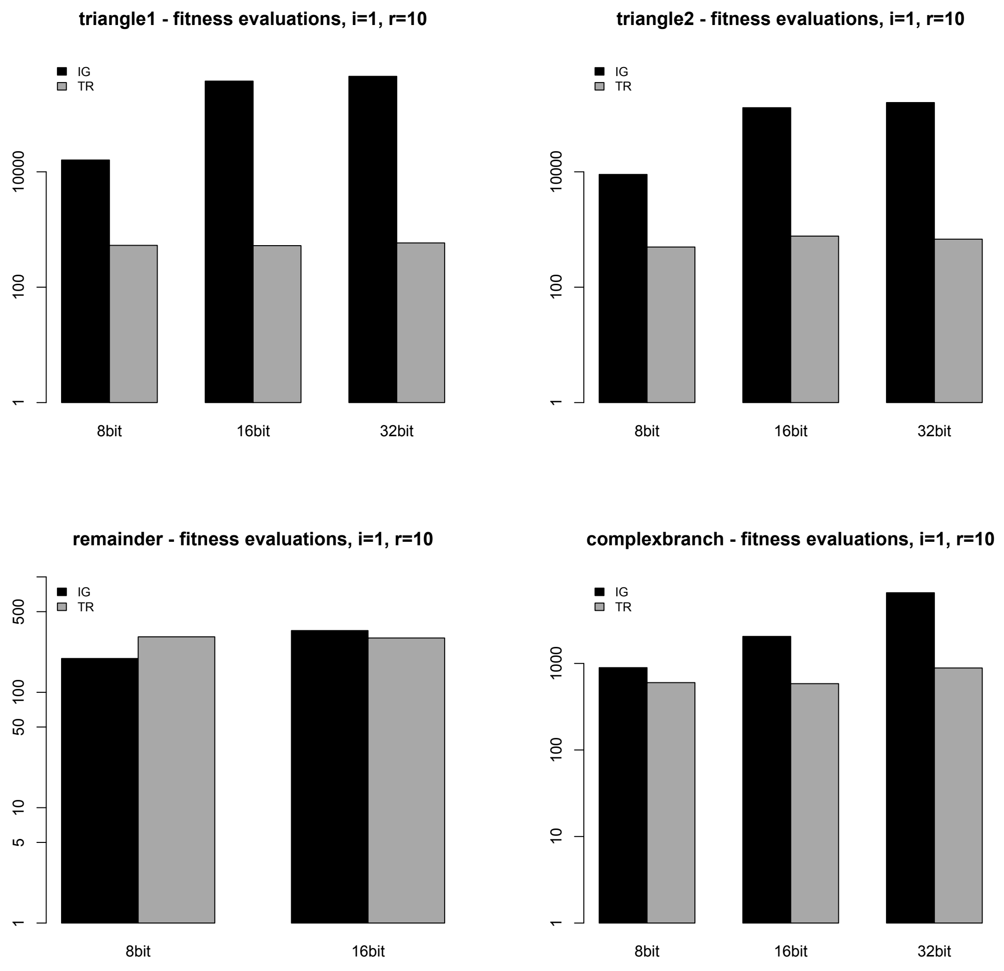


Figure 2: Comparisons of efficiency between test data regeneration and test data generation tool, Iguana, across different input domains. The vertical axis is in logarithmic scale. Plots represent average number of fitness evaluations required for the generation of a new set of test data. Apart from `remainder`, test data regeneration always requires a smaller number of fitness evaluations in order to generate a set of test data, which is statistically confirmed at 95% significance level. With `triangle1` and `triangle2`, the differences are more than two orders of magnitude.

Subject Programs	Input Domain	$\bar{c}_{IG}$	$\sigma^2_{c_{IG}}$	$\bar{c}_{TR}$	$\sigma^2_{c_{TR}}$
`triangle1`	8bit	1.00	0.00	0.94	0.00
`triangle1`	16bit	0.91	0.00	0.94	0.00
`triangle1`	32bit	0.83	0.00	0.94	0.00
`triangle2`	8bit	1.00	0.00	0.82	0.00
`triangle2`	16bit	0.90	0.00	0.82	0.00
`triangle2`	32bit	0.86	0.00	0.82	0.00
`remainder`	8bit	1.00	0.00	1.00	0.00
`remainder`	16bit	1.00	0.00	1.00	0.00
`complexbranch`	8bit	1.00	0.00	1.00	0.00
`complexbranch`	16bit	1.00	0.00	1.00	0.00
`complexbranch`	32bit	0.98	0.00	1.00	0.00

Table 3: Average branch coverage ($\bar{c}$) and variance (σ_c) for TR and IG.

the radius parameter.

6.2 Effectiveness Evaluation: Coverage

Figure 3 shows average branch coverage achieved by both techniques. The x-axis represents the different input domains. The y-axis represents average branch coverage achieved by both techniques across 20 executions. TR achieves 100% branch coverage for `remainder` and `complexbranch` for all input domains. However, it fails to achieve 100% branch coverage for `triangle1` and `triangle2`, although it achieves a constant level of coverage. IG achieves 100% branch coverage for all subject programs in 8bit input domain, but the coverage decreases as the input domain grows larger.

Table 3 presents the statistics observed in Figure 3 in more detail. Note that variances are all 0, meaning that all 20 executions achieved the same branch coverage with both techniques. Therefore, statistical hypothesis testing is not performed. For `triangle1` and `triangle2`, TR shows constant branch coverage for all input domains, whereas IG cannot maintain constant branch coverage as input domain grows. In fact, with the appropriate setting, TR can achieve 100% branch coverage (this will be discussed in Section 6.4). Both techniques achieve 100% branch coverage for `remainder` with all 20 executions. Both techniques achieve 100% branch coverage for `complexbranch` with the exception of IG with 32bit input domain. This provides a partially positive answer for **RQ2**. When the input domain is sufficiently large, TR can be as effective as, or more effective than IG, in terms of branch coverage. For smaller input domains, it is still possible to observe an attractive trade-off between efficiency and effectiveness.

6.3 Effectiveness Evaluation : Mutation Score

Figure 4 shows average mutation score achieved by both techniques. In order to justify the need for additional test suites, the mutation score of original test suites is also included. If executing the additional test suite increases the mutation score significantly, the cost of generating and executing the additional test suites may be justified. For `remainder` and `complexbranch`, both IG and TR mostly achieve either mutation score similar to the original, or higher than the original. The results for `triangle1` and `triangle2` form an interesting contrast. Note that for both programs, IG and TR failed to achieve full coverage as input domain grows larger. This has a significant impact on the mutation score for `triangle2`. Lower coverage leads to lower mutation score because certain mutants are never covered. With `triangle1`, IG is also similarly affected by the lack of full coverage. However, TR shows relatively constant mutation score with `triangle1` across input domain even though it fails to achieve full coverage.

Table 4 presents the statistics observed in Figure 4. In order to compare the mutation score of IG and TR, hypothesis testing is performed. Without any assumption about the population distribution, the one-sided sign test is performed with 95% significance level. The p-value represents the results of sign test between m_{IG} and $\bar{m}_{TR}$. The null hypothesis is that $\bar{m}_{IG}$ is equal to $\bar{m}_{TR}$. The alternative hypothesis is that $\bar{m}_{TR}$ is greater than $\bar{m}_{IG}$. The result is mixed; none of the two techniques dominates the other across all experiments. The alternative hypothesis is accepted for `triangle1` with 32bit input domain, `remainder` for all input domains, and `complexbranch` with 32bit input domain. For other experiments, the results from two techniques either show no statistically significant difference or dominance by IG. However, considering the significantly lower cost of TR, this still provides an attractive trade-off between efficiency and effectiveness.

While mutation score is one possible measure of testing effectiveness, it does not consider the fact that different test suites kill different sets of mutation faults. In order to make a detailed comparison between the original test suite and the test suites generated by IG and TR, the mutation faults are classified according to the test suite that killed them. If, during the 20 executions, a mutation fault is killed by a test suite at least once, the fault is classified as being killed by the test suite. This results in Venn diagrams shown in Figure 5. The set OR, TR, and IG represents the mutation faults that were killed by each technique respectively. Note that both $|TR - OR|$ and $|IG - OR|$ are greater than 0 for all experiments. Considering that the original test suites did achieve 100% branch coverage, this signifies that achieving structural coverage is not always a sufficient testing requirement. It therefore justifies the generation of an additional test suite, which TR can perform very efficiently. For `triangle2` with 8bit and 16bit input domains, TR fails to kill 96 mutation faults that are killed by OR and IG. This is due to the lack of complete coverage observed in Section 6.2. For all other experiments, TR kills at least more than

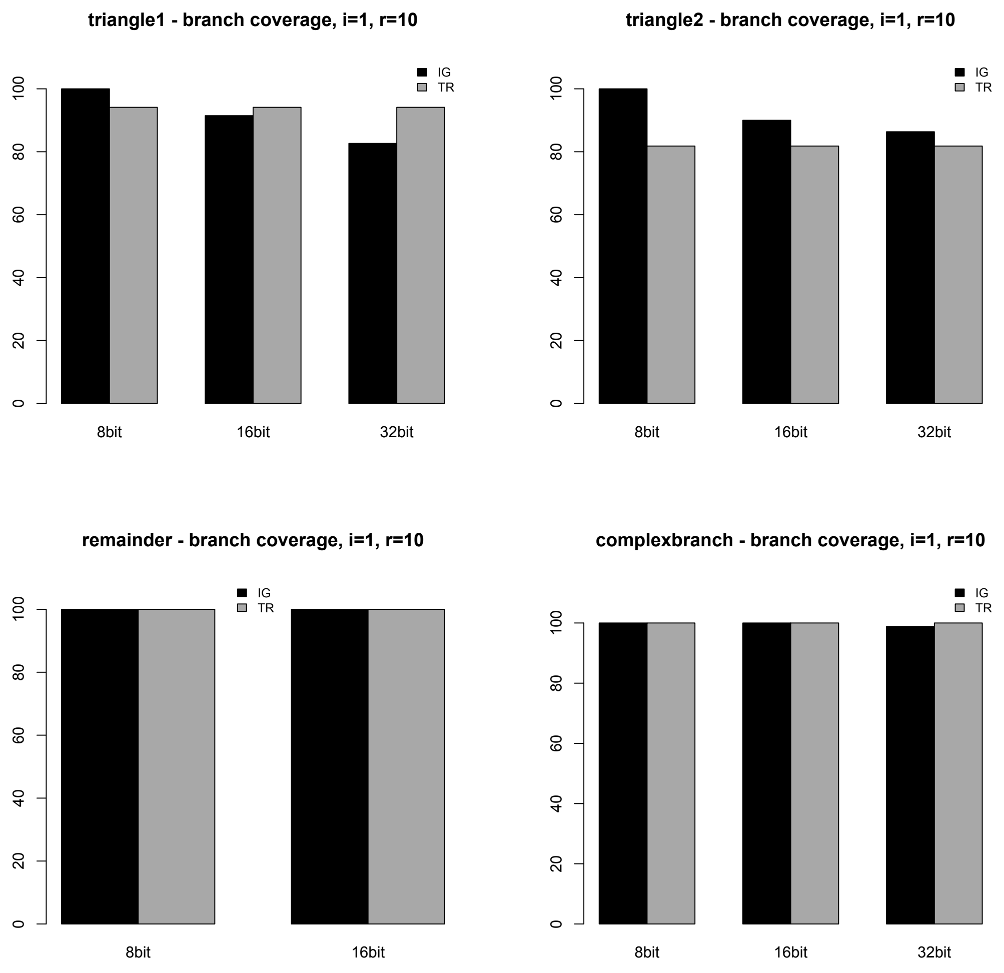


Figure 3: Comparisons of effectiveness between IG and TR, across different input domains, in terms of branch coverage. Plots represent average branch coverage achieved by both techniques. For `triangle1` and `triangle2`, IG shows decreasing branch coverage as the input domain grows larger, suggesting a possible lack of scalability. TR does not achieve 100% for these programs, but maintains the same branch coverage across different input domains. Both techniques achieve 100% branch coverage for `remainder` with all 20 executions. IG fails to achieve 100% branch coverage for `complexbranch` in 32bit integer domain. Note that the plotted coverage is the coverage achieved by the newly generated test suite alone; the original test suites were not included.

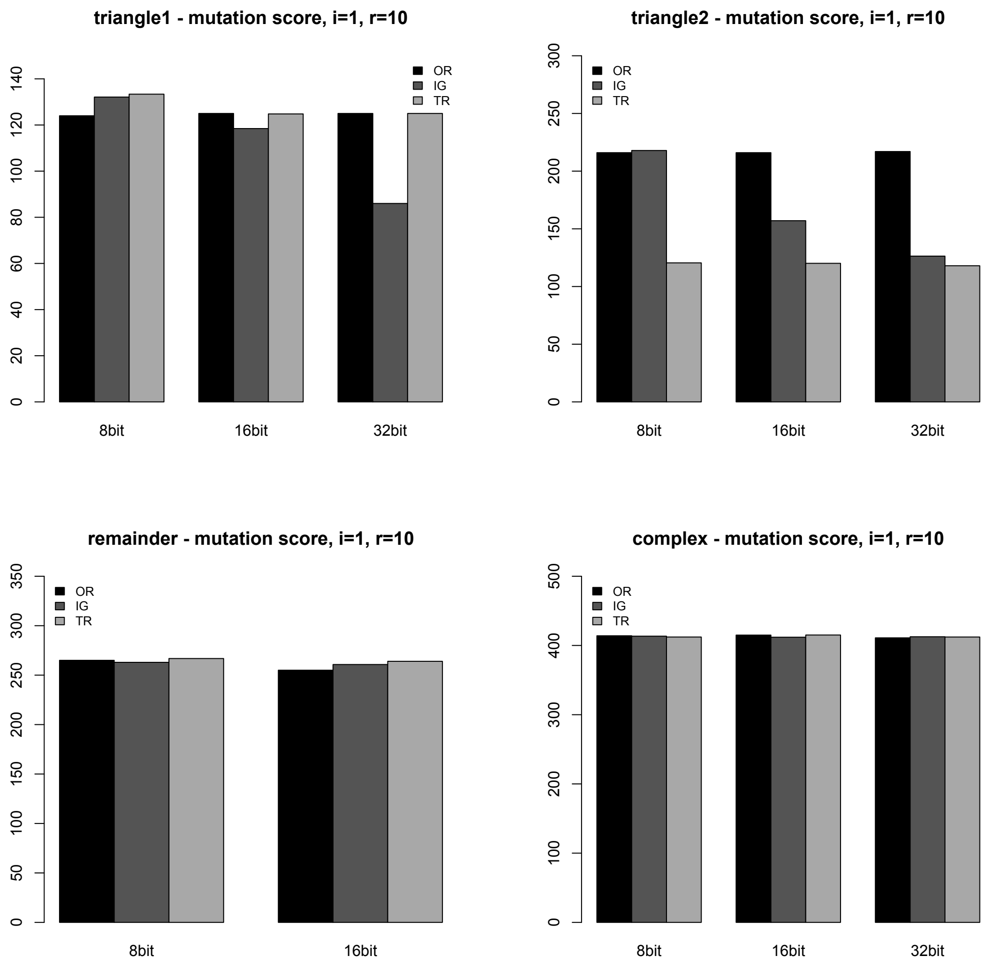


Figure 4: Comparisons of effectiveness between IG and TR, across different input domains, in terms of mutation score. In order to test whether the additional set of test data improves the quality of testing, the mutation score of original test suites is included (OR). Plots represent either the mutation score (OR) or average mutation score (IG and TR). With `triangle2`, the lack of full coverage significantly affects both IG and TR. For other programs, TR is as effective as, or more effective than OR and IG in terms of mutation score.

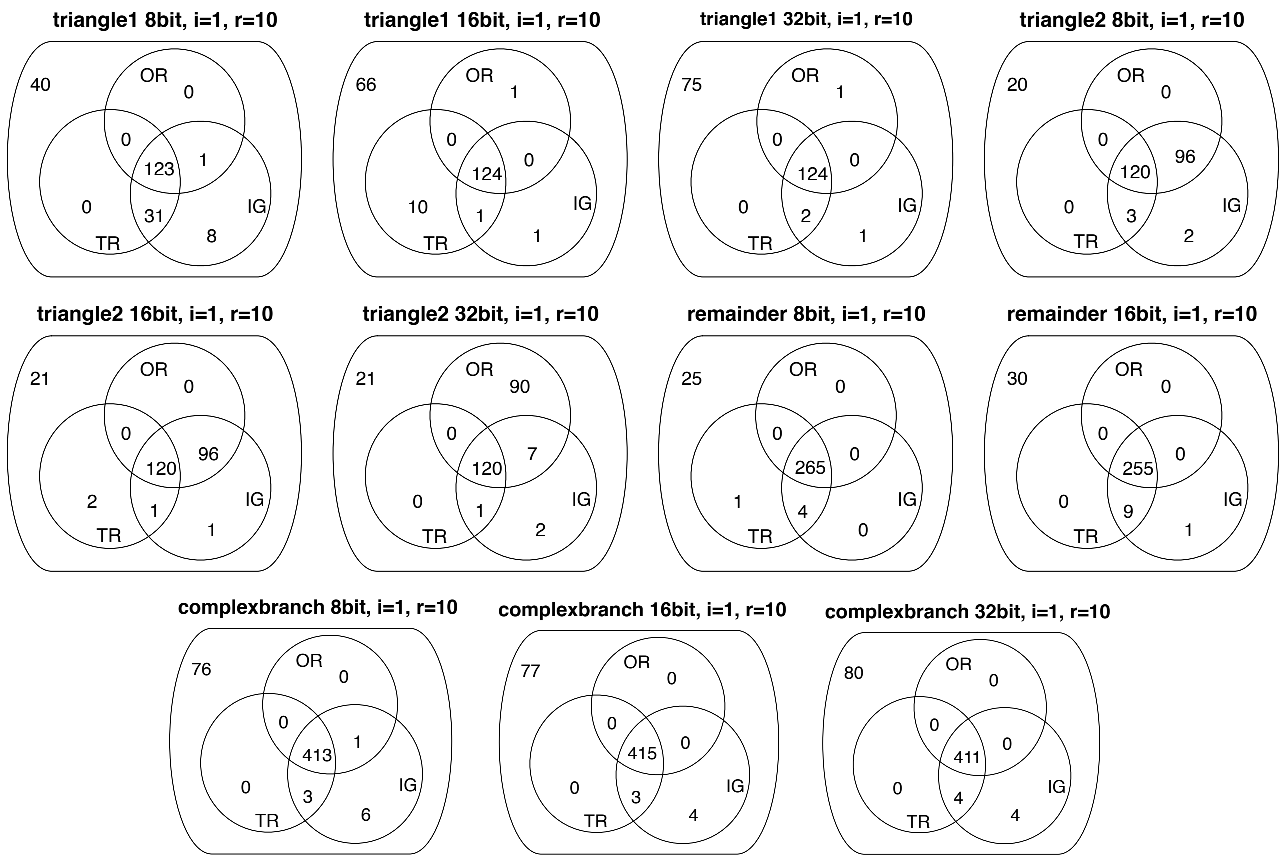

triangle1 8bit, i=1, r=10
40 OR 0 0 123 1 0 31 8 TR IG
triangle1 16bit, i=1, r=10
66 OR 1 0 124 0 10 1 1 TR IG
triangle1 32bit, i=1, r=10
75 OR 1 0 124 0 0 2 1 TR IG
triangle2 8bit, i=1, r=10
20 OR 0 0 120 96 0 3 2 TR IG
triangle2 16bit, i=1, r=10
21 OR 0 0 120 96 2 1 1 TR IG
triangle2 32bit, i=1, r=10
21 OR 90 0 120 7 0 1 2 TR IG
remainder 8bit, i=1, r=10
25 OR 0 0 265 0 1 4 0 TR IG
remainder 16bit, i=1, r=10
30 OR 0 0 255 0 0 9 1 TR IG
complexbranch 8bit, i=1, r=10
76 OR 0 0 413 1 0 3 6 TR IG
complexbranch 16bit, i=1, r=10
77 OR 0 0 415 0 0 3 4 TR IG
complexbranch 32bit, i=1, r=10
80 OR 0 0 411 0 0 4 4 TR IG

Figure 5: Venn diagrams classifying mutation faults according to the test suites that killed them. Note that for all experiments, $|TR - OR| > 0$, i.e. test data regeneration has made an improvement in mutation score for all experiments. IG has also made an improvement that is as good as or better than TR for all experiments. However, considering Section 6.1, TR still provides an attractive trade-off between efficiency and effectiveness.

Subject Programs	Input Domain	m_{OR}	$\bar{m}_{IG}$	$\sigma^2_{m_{IG}}$	$\bar{m}_{TR}$	$\sigma^2_{m_{TR}}$	p-value
`triangle1`	8bit	124	132.10	51.67	133.35	36.34	0.5000
`triangle1`	16bit	125	118.46	75.31	124.80	10.06	0.0898
`triangle1`	32bit	125	86.00	370.32	125.00	0.42	$< 10^{-6}$
`triangle2`	8bit	216	217.90	2.20	120.50	2.79	1.0000
`triangle2`	16bit	216	157.05	1158.79	120.10	3.67	1.0000
`triangle2`	32bit	217	126.35	0.56	118.00	1.58	1.0000
`remainder`	8bit	265	262.95	6.68	266.80	3.01	0.0037
`remainder`	16bit	255	260.75	5.99	264.00	0.00	0.0002
`complexbranch`	8bit	414	413.45	8.16	412.20	2.69	0.9283
`complexbranch`	16bit	415	411.95	7.63	415.10	0.62	$< 10^{-4}$
`complexbranch`	32bit	411	412.60	2.04	412.20	0.69	0.8950

Table 4: Average mutation score ($\bar{m}$) and variance (σ_m) for TR and IG. The column m_{OR} represents the mutation score achieved by the original test suite used by TR. The p-value represents the results of the one-sided sign test between m_{IG} and $\bar{m}_{TR}$. The null hypothesis is that $\bar{m}_{IG}$ is equal to $\bar{m}_{TR}$. The alternative hypothesis is that $\bar{m}_{TR}$ is greater than $\bar{m}_{IG}$. The alternative hypothesis is accepted for `triangle1` with 32bit input domain, `remainder` for all input domains, and `complexbranch` with 32bit input domain.

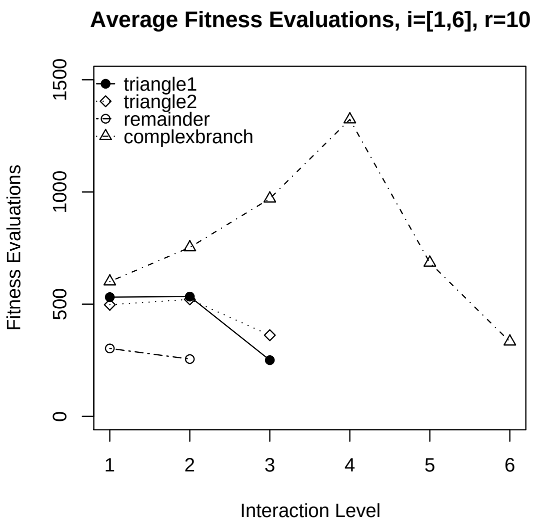


Figure 6: Plot of average number of fitness evaluations against interaction level. The number of neighbouring solutions is bounded depending on the size of test input vector and interaction level. Higher number of neighbouring solutions generally results in higher number of fitness evaluations, i.e. more candidate solutions to evaluate. For `triangle1` and `triangle2`, $i = [1, 2]$ yields the most neighbouring solutions. For `complexbranch`, $i = 3$ yields the most neighbouring solutions, but $i = 4$ results in most fitness evaluations.

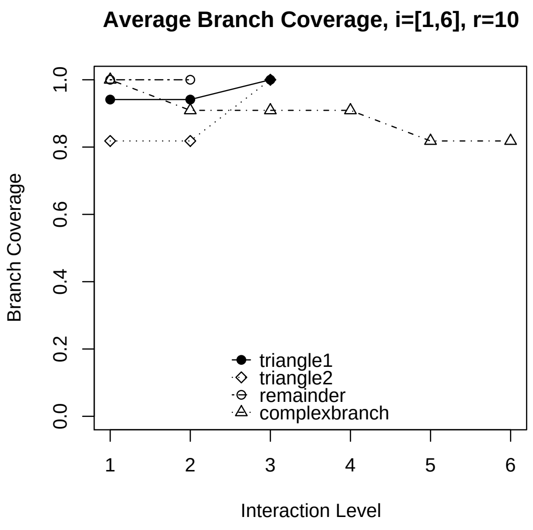


Figure 7: Plot of average branch coverage against interaction level. Both `triangle1` and `triangle2` achieve full coverage when $i = 3$ due to certain branches that require the interaction level of 3 in order to generate a qualifying solution. The branch coverage for `remainder` is not affected by increasing interaction level. For `complexbranch`, increasing interaction level actually reduces the average branch coverage achieved.

half of the mutants in the set $IG - OR$. Indeed, $|TR - OR|$ is greater than $|IG - OR|$ for `triangle1` with 16bit integer input domain and `remainder` with 8bit integer input domain. Combining this with the gain in efficiency observed in Section 6.1, it provides an evidence that there exists an attractive trade-off between efficiency and effectiveness for TR. Overall this provides a positive answer to **RQ3**.

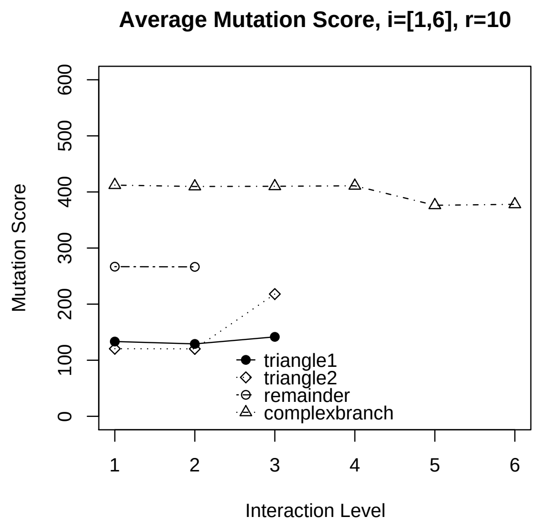
6.4 Settings : Impact of Interaction Level

This section concerns the first part of **RQ4** and observe the impact of different interaction levels on efficiency and effectiveness. The minimum possible interaction level is 1; with 0 interaction level, it is not possible to generate a new test input. The maximum possible interaction level for `triangle1`, `triangle2`, `remainder` and `complexbranch` is 3, 3, 2, and 6 respectively, i.e. the size of test input vector for these programs. The input domain is fixed at 8bit integers. The search radius is fixed at 10.

Figure 6 shows the change in the average number of fitness evaluations against different interaction levels. The interaction level determines the number of neighbouring solutions that TR considers in a single iteration. With a test input vector of size m and interaction level i, the number of neighbouring solutions is bounded by $\binom{m}{i}$. The more neighbouring solution TR considers, the more fitness evaluations TR is likely to spend. Based on this, it is expected that each plot to peak at the point that corresponds to the pair of m and i that yields the maximum value $\binom{m}{i}$:

- `triangle1` : interaction level $i = \{1, 2\}$ is expected to yields the most fitness evaluations.
- `triangle2` : interaction level $i = \{1, 2\}$ is expected to yields the most fitness evaluations.
- `remainder` : interaction level $i = 1$ is expected to yield more fitness evaluations than $i = 2$.
- `complexbranch` : interaction level $i = 3$ is expected to yield the most fitness evaluations.

The result shown in Figure 6 mostly confirms the expectations. For `triangle1`, `trioangle2`, and `remainder`, the expectations are confirmed. However, for `complexbranch`, the peak occurs at $i = 4$, not $i = 3$ as expected. This suggests that TR considered more candidate solutions when $i = 4$, even though at each iteration it generates more neighbouring solutions when



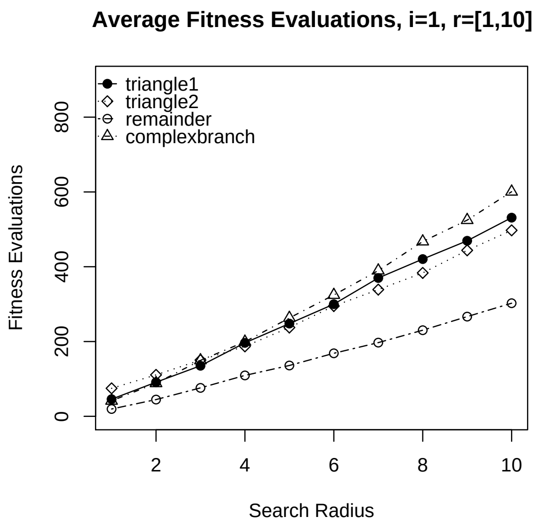


Figure 8: Plot of average mutation score against interaction level. Both `triangle1` and `triangle2` benefit from the increased branch coverage observed in Figure 7 when $i = 3$. On the other hand, `complexbranch` suffers from the decreased coverage as the interaction level increases. The mutation score of `remainder` is not affected.

Figure 9: Plots of the average number of fitness evaluations against the search radius. The average number of fitness evaluation shows a very strong correlation to the search radius; TR spends more fitness evaluations while trying to make more modifications.

$i = 3$. Therefore, the real impact comes not only from the number of maximum possible neighbours, but also from the number of those that are within the window of qualification. If the program under test allows sufficient number of neighbouring solutions that qualify, the number of fitness evaluation of TR is more likely to be limited by the search radius. On the other hand, if the program under test does not allow sufficient neighbouring solutions that qualify, TR will spend large number of fitness evaluations for disqualifying neighbouring solutions.

Figure 7 and Figure 8 observe effectiveness measures against interaction level in terms of branch coverage and mutation score respectively. In Figure 7, both `triangle1` and `triangle2` achieves full coverage when $i = 3$, complimenting the lack of full coverage observed in Section 6.2. More importantly, TR achieves full coverage with $i = 3$, while still remaining significantly more efficient than IG. Both programs contain branches that require the modification of all input variables in order to generate another qualifying solution. For `triangle1`, it is a branch that determines whether the given numbers form an equilateral triangle. From a set of three numbers that form an equilateral triangle, it is possible to generate alternative qualifying set of numbers when all three numbers are modified at the same time, by one of the modification operators $\{\lambda x.x + 1, \lambda x.x - 1, \lambda x.x * 2, \lambda x.\lceil x/2 \rceil\}$. Similarly, `triangle2` contains a branch that determines whether the given numbers form a right angle triangle. The generation of an alternative qualifying solutions is *guaranteed* only when $i = 3$ and only with the modification operator $\lambda x.x * 2$. This provides a justification for utilising modification operators that are more complex than $\{\lambda x.x+1, \lambda x.x-1\}$. With domain knowledge, it may be possible to generate more complex but potentially more effective set of modification operators.

In Figure 8, it is possible to observe the positive impact of the increased branch coverage. Both `triangle1` and `triangle2` show improved mutation score with $i = 3$. The plot for `complexbranch` corresponds to the branch coverage of the program observed in Figure 7. With less coverage, TR kills fewer mutants as interaction level increases. For `remainder`, the increased interaction level has no significant impact on mutation score.

6.5 Settings : Impact of Search Radius

Finally, this section concerns the second part of **RQ4** by observing the efficiency and effectiveness of TR while changing the search radius. The interaction level is fixed at 1, and the input domain is fixed at 8bit integers. TR is executed 20 times for each search radius value from 1 to 10.

Figure 9 plots the average number of fitness evaluations and average branch coverage against the search radius. For all subject programs, the average number of fitness evaluations shows a very strong correlation to the search radius, which is expected. As the search radius increases, TR is allowed to make more modifications from the original test input, thereby spending more fitness evaluations. However, the observed coverage values remain constant at the maximum level that can be achieved for each program under the given configuration. TR achieved full branch coverage for `remainder` and `complexbranch`; it also covered all the branches in `triangle1` and `triangle2` except for the ones discussed in Section 7.

Figure 10 shows plots of average mutation score against the search radius. With the exception of `complexbranch`, the

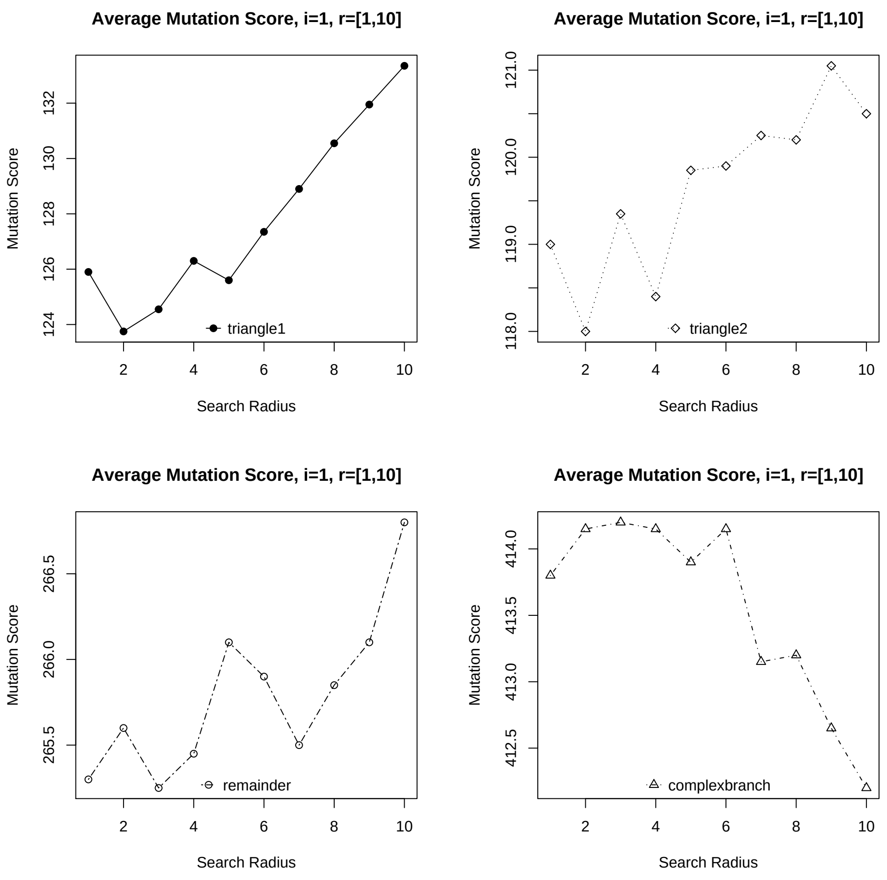


Figure 10: Plots of average mutation score against the search radius. Except for `complexbranch`, average mutation score shows an increasing trend, providing a partial justification for TR to make as many modifications as possible. That is, the more an additional test case is different from the original, the more valuable it is for these programs.

Program	ρ_1	ρ_2
`triangle1`	0.9991983	0.9279003
`triangle2`	0.9977082	0.8569495
`remainder`	0.9992023	0.7774765
`complexbranch`	0.9983878	-0.8378015

Table 5: The column of ρ_1 shows the linear correlation coefficient between the search radius and the average number of fitness evaluation. All four programs show a very strong correlation. The column of ρ_2 shows the linear correlation coefficient between the search radius and average mutation score. It shows a significant correlation for `triangle1`, `triangle2`, and `remainder`. On the other hand, `complexbranch` shows a negative correlation.

observed trend is that average mutation score increases as the search radius increases. This provides partial evidence to confirm the distance-value assumption described in Section 4.2; that is, the more different an additional test input is from the original test input, the more valuable it is. Table 5 shows linear correlation coefficients between the search radius and the average fitness evaluations/averate mutation score. The coefficients confirm the visual trends observed in Figure 9 and Figure 10.

7. DISCUSSION

This section discusses factors that prevent IG from achieving full coverage for larger input domains. For `triangle1` and `triangle2`, IG fails to achieve full branch coverage when the input domain is set to 16bit or 32bit integers. This is due to the data dependency observed in both programs. Algorithm 4 is an excerpt from `triangle1`.

The program takes three integers as input, and classifies the input according to the type of triangle that can be formed with lengths of sides equal to the three integers. Suppose that the tester is trying to make the predicate in line (6) `false`, which means `tri != 0`. According to Korel's definition, the branch distance is calculated as $-|tri - 0|$ [18], which should be minimised. The value of the variable `tri` is assigned between line (3) and (5) based on the values of the program input, `(i, j, k)`. Since the predicate in line (6) is control-independent from the predicates in line (3), (4), and (5), the branch distance of $-|tri - 0|$ cannot guide the input vector. That is, changes made to the input vector `(i, j, k)` do not correlate with the branch distance of the predicate in (6), except for the very rare cases where the algorithm starts with a random input vector that is very close to satisfying either one of the predicates in line (3), (4), and (5). This results in a flat fitness landscape, thereby significantly weakening performance of any meta-heuristic search technique. As a result, IG often fails to cover the `false` edge of the predicate in line (6).

Algorithm 4: An excerpt from the `triangle` program

```
(1)     read(i, j, k)
(2)     tri = 0
(3)     if i == j then tri += 1
(4)     if i == k then tri += 2
(5)     if j == j then tri += 3
(6)     if tri == 0
(7)         if i + j ≤ k or j + k ≤ i or i +k ≤ j
(8)             tri = 4
(9)         else
(10)            tri = 1
(11)        return tri
(12)    ...
```

Compared to `triangle1`, `triangle2` contains additional branches that determine whether the given input forms a right-angle triangle. IG tends to fail to cover these branches as the input domain grows larger. There is no data dependency involved as in `triangle1`; the difficulty lies in the behaviour of alternating variable method called *over-shooting.* In the exploratory phase, AVM doubles the amount of change with every successful iteration. Eventually it over-shoots, i.e. the fitness value decreases due to an excessive change. The excessiveness is exaggerated in this case because the input values are squared in the predicate that determines the formation of a right-angle triangle. Once it over-shoots, AVM switches to pattern phase and starts to change the next input variable. If the desired solution is sufficiently hard to find, there is a chance that AVM will keep over-shooting with oscillating fitness values. IG fails to achieve full branch coverage for these predicates because it spends the allowed maximum fitness evaluations while over-shooting.

Figure 6-8 and Figure 10 show that the reaction of `complexbranch` to changes in interaction level and the search radius is different from that of other subject programs. The number of fitness evaluations does not peak at the predicted interaction level. Increasing interaction level does not seem to have a positive impact on either coverage or mutation score. Finally, increasing the search radius does seem to have a negative impact on mutation score. Unlike `triangle1` and `triangle2`, it was not possible to identify a clear reason why it behaved so differently. It does not contain the type of data dependency observed in `triangle1`; the only notable difference between `complexbranch` and other subject programs was the fact that `complexbranch` uses a `switch` statement, which is essentially equivalent to nested `if` statements and should not present any particular challenge. The differences may be caused by the fact that `complexbranch` has the largest input domain among the subject programs, with complex branch predicates that are specifically designed to make it hard to achieve full branch coverage.

8. CASE STUDY

This section applies the test data regeneration technique to unit level testing of real world examples that use more complex input data types.

8.1 Subject Programs

The case study considers two Java methods from real world Java libraries. The first library is `colt`, which is an open source Java library for High Performance Scientific and Technical Computing developed at the European Organisation for Nuclear Research (CERN). From the library, `int binarySearch(int[] list, int key, int from, int to)` was chosen, which is an implementation of binary search algorithm. It searches the integer array `list` for the integer `key`, starting from the index denoted by the formal parameter `from` up to the index denoted by the parameter `to`. The method contains 5 branches. The second library is `SIENA` (Scalable Internet Event Notification Architecture), which is an open source event notification framework. From the Java API implementation, `int read_number(byte[] buffer)` was chosen, which parses a number from the string contained in the byte array `buffer` and determines the type of the number, i.e. integer, double, or unknown. The method contains 22 branches including one unreachable branch. Since both methods take an array as input, the size of the search space for potential input is essentially unbounded, making this a challenging search space in which to seek solutions.

8.2 Test Data Regeneration Technique

Test input for a dynamic data structure requires more sophisticated modification operators than those used in Section 4.4. For `binarySearch`, the following modification operators were considered. Given a test input of <`int[] list`, `int key`, `int from`, `int to`>, the operators are defined as follows:

- **Insert into list**: inserts a random integer into the integer array. After insertion, the array is sorted in order to meet the precondition of binary search.
- **Replace key**: replaces the value of the parameter `key` with a randomly chosen number from `list`
- **Increase/decrease from**: increases or decreases the value of the `from` variable by 1
- **Increase/decrease to**: increases or decreases the value of the `to` variable by 1

For `read_number`, the following modification operators were considered. Given a test input of <`byte[] buffer`>, the operators are defined as follows:

- **Insert into buffer**: inserts a random ASCII character at a random position
- **Delete from buffer** : selects a random position in `buffer` and removes the value at the position, which results in a shorter array
- **Increase/decrease byte**: selects a random byte in `buffer` and increases or decreases the stored value by 1

In order to apply a set of test data generation tools to real world examples, it is not uncommon to find that some small modifications are required relating to implementation details. The case studies upon which are reported here are no exception. Specifically, two minor extensions are required in order for the test data regeneration technique described in Section 4.4 to cater for the dynamic data structure.

First, Euclidean distance needs to be replaced with a more generic distance metric. There is some existing work on generic distance measures between test inputs based on Information Theory [9]. While these might be promising for future work, it can be computationally expensive. For this reason, the 'distance' between two test inputs was measured using the weighted Levenshtein distance between the two minimal string representations of the test inputs [21]. Levenshtein distance is used to measure the distance between two arbitrary strings. It is computed as the minimum number of operations needed to transform one string into the other. The operations are insertion, removal, and replacement of a single character. Algorithm 5 describes the dynamic programming approach used to compute Levenshtein distance.

Here, the basic definition of Levenshtein distance is extended to obtain a weighted version, by replacing `cost:=1` in line 11 of Algorithm 5 with `cost:=|s[i-1]-t[j-1]|`, i.e. the difference between byte representation of two characters at index `i-1`. This enables ranking the results obtained from the replacement operator, according to the number of incremental/decremental changes required. For example, the distance between "1,3" and "1,4" is shorter than the distance between "1,3" and "1,7".

The second modification actually simplifies the algorithm described in Algorithm 2. The modification operators used in the empirical study are deterministic, and so is Algorithm 3, the algorithm that generates the neighbourhood solutions. However, some of the new modification operators introduced in this section are inherently stochastic. This means that the test data regeneration algorithm is likely to consider a different set of neighbourhood solutions, with each restart of the hill climbing. Therefore, the algorithm used in the case study adopts steepest ascent hill climbing.

Algorithm 5: Pseudo-code of basic Levenshtein Distance calculation algorithm
Input: Two strings, s[1..m] and t[1..n]
Output: Distance d in integer

```
(1)   declare int d[0..m, 0..n]
(2)   for i = 0 to m
(3)       d[i, 0] := i
(4)       for j = 0 to n
(5)           d[0, j] := j
(6)   for i = 1 to m
(7)       for j = 1 to n
(8)           if s[i - 1] = t[j - 1]
(9)               cost := 0
(10)          else
(11)              cost := 1
(12)          d[i, j] := $min(d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1]+cost)$
(13)
(14)  return d[m, n]
```

8.3 Iguana : Hill Climbing Test Data Generation

The results of the test data regeneration technique are compared to the Hill Climbing algorithm based test data generated using Iguana tool. The specifications of the subject programs require that special care is taken to use Iguana to generate test data for the programs. For example, `binarySearch` takes <`int[] list`, `int key`, `int from`, `int to`> as input. It is implied in the specification that variables `from` and `to` should contain integers that can be index values of the array `int[] list`. However, this is not reflected in the form of branch distance, preventing the Hill Climbing algorithm from receiving any guidance for the values of `from` and `to`. Therefore, the length of a list has been limited to 10 and the input domain for variable `from` and `to` has been adjusted to $[0 \ldots 9]$. The variable `key`, on the other hand, is used in a branch that determines whether the binary search has found the `key` variable in the `list`. This enables the Hill Climbing algorithm to receive guidance for the value of `key`.

Another interesting issue is raised when applying Iguana to `binarySearch`, which is that of satisfying input preconditions. Because the function is an implementation of the binary search algorithm, the integer array is expected to be sorted. However, there is no way to force Iguana to form a sorted array within the normal usage of the tool. In order to make a completely unbiased comparison, it was decided that Iguana will be used without any modification. In fact, the intentional decision not to use any domain knowledge makes an ideal comparison to the TR approach which can be thought of as being dependent on a form of domain knowledge (i.e. the existing test input). This decision results in an interesting observation on branch coverage, which is discussed in Section 8.6.

In the case of `read_number`, the only constraint implied by the input specification is that the byte array contains characters. Therefore, the input domain for each character in the byte array has been limited to that of an ASCII code, in the range $[0 \ldots 255]$.

8.4 Original Test Suite and Mutation Faults

The original test suites were generated manually so that there exists one test case per branch in the program that is covered by the test case. Therefore, the test suite for `binarySearch` contains 5 test cases, which collectively achieve 100% branch coverage. The test suite for `read_number` contains 21 test cases (because `read_number` has 22 branches, one of which is unreachable). Due to the unreachable branch in the code, the maximum branch coverage that can be achieved from `read_number` is 95.4%.

Mutation faults are generated by the `muJava` mutation tool using the same set of mutation operators shown in Table 1. This resulted in 61 executable mutants in case of `binarySearch` and 190 executable mutants in case of `read_number`. No effort was made to exclude equivalent mutants.

8.5 Evaluations

In order to cater for the potential impact of the inherent randomness of search-based optimisation algorithms, the experiment for each subject is repeated 20 times. For the case study, the interaction level of test data regeneration algorithm is set to 1; only one input variable is modified for each neighbouring solution. The search radius is set to 20. As in the empirical study, the efficiency of the technique is measured by counting the number of fitness evaluations, whereas the effectiveness of the technique is measured both by mutation score and by branch coverage. The Iguana tool is configured to spend the maximum of 3,000 fitness evaluations for a single branch before moving on to the next branch. Following the discussion in Section 5.5, these numbers are measured and presented without forcing per-effort or per-goal normalisation.

8.6 Results and Analysis

Table 6 shows the statistical analysis of the average number of fitness evaluations required by IG and TR for the generation of the additional test suite. For `binarySearch`, IG turns out to be more efficient than TR, being contrary to the trend observed

Subject	$\bar{n}_{IG}$	$\sigma^2_{n_{IG}}$	$\bar{n}_{TR}$	$\sigma^2_{n_{TR}}$	p-value
`binarySearch`	475.70	5136.85	556.20	940.84	1.0
`read_number`	23,370.30	14,631,258.00	574.00	1,367.37	$< 10^{-15}$

Table 6: Average number of fitness evaluations ($\bar{n}$) and variance (σ^2_n) for TR and IG.

Subject	c_{OR}	$\bar{c}_{IG}$	$\sigma^2_{c_{IG}}$	$\bar{c}_{TR}$	$\sigma^2_{c_{TR}}$	p-value
`binarySearch`	100.00%	100.00%	0.00	100.00%	0.00	n/a
`read_number`	95.54%	76.36%	75.25	95.23%	1.03	$< 10^{-8}$

Table 7: Average branch coverage ($\bar{c}$) and variance (σ^2_c) for TR and IG.

in smaller programs. This is due to the fact that Iguana ignores the requirements that the input array is sorted. When an unsorted array is given to a binary search algorithm, it is very difficult for the binary search to be successful but it remains relatively easy to cover some branches before terminating.

For example, the first step of a binary search is to compare the middle element of the array to the key variable in order to determine which one is bigger. A random assignment of the array and the key variable will have a relatively high chance of satisfying either branch of this step, which is an advantage for IG as it assigned the initial values randomly. However, while they succeed in covering branches, very few of the test cases generated by IG will conform to the *normal* behaviour of a binary search (because they may not satisfy the precondition). On the other hand, TR starts with a legitimate test suite, i.e. the array elements are sorted. The definition of the fitness function in Section 4.3 requires that, when generating an additional test case, TR should preserve the execution trace of the original test case; that is, TR can only produce test cases that conform to the normal behaviour of a binary search (which satisfies the precondition), because the manually generated original test suite does. This limits the scope of acceptable modifications to the original test case, resulting in suboptimal performance with respect to efficiency. However, this arguably produces more useful results, because the new test can achieve coverage while simultaneously meeting the precondition.

By contrast, the results for `read_number` follow the trend observed in the smaller programs. This is because `read_number` shares a similar type of data dependency that was observed in `triangle1`. A few branches in `read_number` contain predicates that depend on either a variable that was assigned earlier or a return value of an external function. For the same reason discussed in Section 7, IG is vulnerable to this type of data dependency in branch predicates. As a result, TR is more efficient than IG for `read_number`.

The average number of fitness evaluations required by IG and TR are compared using one-sided t-test with the significance level of 95%. The null hypothesis is that there is no difference between mean values of n_{IG} and n_{TR}. The alternative hypothesis is that $n_{IG} > n_{TR}$. For `binarySearch`, the alternative hypothesis is rejected for the reason described above. For `read_number`, the alternative hypothesis is accepted.

Table 7 shows the average branch coverage achieved by the original test suite, IG and TR respectively. For `binarySearch`, both techniques reproduce the 100% branch coverage of the original test suite with variance of 0. In `read_number`, there exists a single branch that is unreachable, making the highest achievable branch coverage 95.54%. The average branch coverage achieved by TR almost reaches the highest possible value; in fact, TR successfully reaches 95.54% branch coverage 18 times out of the 20 runs, as can be observed from the relatively small variance. On the other hand, the average branch coverage achieved by IG is 76.36%, with higher variance than TR.

The average branch coverage achieved by IG and TR are compared using one-sided t-test with the significance level of 95%. The null hypothesis is that there is no difference between mean values of c_{IG} and c_{TR}. The alternative hypothesis is that $c_{TR} > c_{IG}$. The results from `binarySearch` do not require the t-test as both samples are essentially uniform and equal to each other. For `read_number`, the alternative hypothesis is accepted at 95% significance level.

Figure 11 shows the average mutation score achieved by IG and TR, compared to the original mutation score. Plots for IG and TR show the average mutation score from 20 runs. While both IG and TR achieve a higher mutation score than the original test suite, TR shows the highest mutation score in both programs. Table 8 shows the statistical analysis of the mutation score results in detail. The mutation scores of IG and TR are compared using one-sided t-test. The null hypothesis is that there is no difference between mean values of m_{IG} and m_{TR}. The alternative hypothesis is that $m_{IG} < m_{TR}$. While $\bar{m}_{IG}$ is less than $\bar{m}_{IG}$ for `binarySearch`, the observed p-value narrowly rejects the alternative hypothesis, i.e. there is no statistically significant difference between $\bar{m}_{IG}$ and $\bar{m}_{IG}$. The observed p-value for `read_number` confirms the alternative hypothesis, i.e., TR achieves a higher mutation score than IG with statistical significance.

In order to make a detailed comparison between the original test suite and the test suites generated by IG and TR, the mutation faults are classified according to the test suite that killed it. Figure 12 shows the resulting Venn diagrams for the results of mutation testing. If, during the 20 executions, a mutation fault is killed by a test suite at least once, the fault is classified as being killed by the test suite. The diagram for `binarySearch` shows that, although there is no statistically significant difference between mutation scores of IG and TR, the two techniques kill different set of mutation faults. The

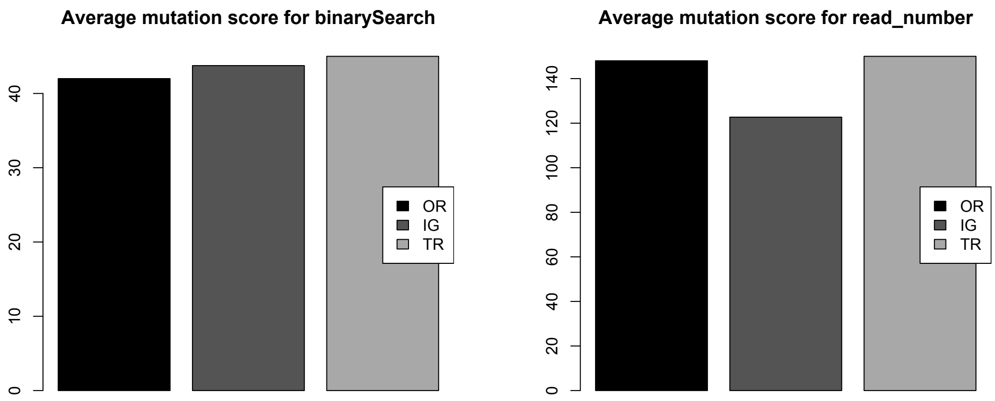


Figure 11: Average mutation score for OR, TR and IG. For both `binarySearch` and `read_number`, TR scores the highest mutation score on average.

Subject	m_{OR}	$\bar{m}_{IG}$	$\sigma^2_{m_{IG}}$	$\bar{m}_{TR}$	$\sigma^2_{m_{TR}}$	p-value
`binarySearch`	42.00	43.65	7.78	45.00	6.63	0.07
`read_number`	148.00	122.70	418.64	150.00	5.68	$< 10^{-5}$

Table 8: Average mutation score ($\bar{m}$) and variance (σ^2_m) for TR and IG.

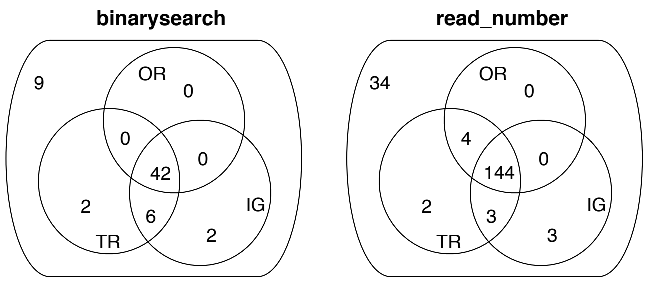


Figure 12: Venn diagrams classifying mutation faults according to the test suites that killed them.

diagram for `read_number` also shows that TR is capable of detecting a set of mutation faults that are not detected by other techniques. Finally, it can be observed that, as with the empirical studies in Section 6, both $|TR - OR|$ and $|IG - OR|$ are greater than 0, confirming the added value of executing additional test cases.

9. THREATS TO VALIDITY

Threats to internal validity concern the factors that might have affected the comparison of IG and TR. The two techniques are so inherently different from each other that the comparison at algorithmic level may not be sound. However, should a tester want a new and additional test suite, the only available state-of-the-art solution is to adopt existing test data generation technique and hope that it will generate a completely new test suite. One alternative is to modify existing search-based test data generation technique to be existing test suite aware. The modified fitness function will measure not only branch distance based fitness, but also the distance from the existing test data. However, this can only add more complexity to traditional search-based test data generation techniques.

Both techniques are inherently stochastic, which may affect the comparison. The experiments are repeated 20 times to cater for the stochastic nature, and the results are verified with statistical testing. For some experiments, the observed measurements are consistent and show little variance.

With TR, the quality of the original test suite can affect the quality of the new test suite generated by TR. The original test suites are manually generated for each input domain with the sole purpose of achieving full branch coverage. That is, other testing concerns such as boundary values or mutation score are not considered. However, there still exists a chance that the original test suites are biased.

Threats to external validity concerns the factors that limit generalisation of the results. One issue is the representativeness of the subject programs, which are relatively small-scale examples. It cannot be guaranteed that the observed results of this paper will extrapolate to larger-scale real world programs. This can be addressed only by further study with larger programs. However, the subject programs have been utilised as benchmarks for test data generation techniques [26, 38, 44]. They also provide non-trivial search space in combination with different input domains. Another issue concerns the selection of the test data generation technique for the comparison. This paper compares the proposed search-based test data regeneration technique to a hill climbing based AVM (Alternating Variable Method). Other test data generation techniques may produce different results. However, AVM has been known to be as much as, or even better than other search techniques such as genetic algorithms, for certain classes of test data generation problems [13].

Threats to construct validity arise when the measurements in the experiments do not capture the concepts that they are supposed to represent. The efficiency measure is a count of fitness evaluations, which equals the number of times the program under test is executed. It does not consider other costs of testing such as generating and checking test oracles. However, as test data regeneration is applied to larger programs, the execution time of the program under test is at least one of the major elements that account for the cost of testing for any type of dynamic testing. Similarly, the effectiveness measure used in this paper may not capture the real effectiveness of a test suite; its fault detection capability. However, structural coverage and mutation score has been widely used as successful surrogate of fault detection capability in software testing literature [8, 26, 29, 38, 44]. Also, the testing process can benefit from the existence of an alternative test suite even when it does not detect any additional faults. For example, the estimation of program reliability when no fault is detected can benefit from alternative test suites.

10. CONCLUSION AND FUTURE WORK

10.1 Conclusions

This paper introduces a search-based test data regeneration, which is a novel method of generating test data from existing test data. Test data regeneration is based on two observations. First, it is beneficial to generate additional test data even if test data are already available. Second, if there are existing test data, generating additional test data can be much more efficient when it utilises the knowledge of existing test data.

The paper introduces a search-based test data regeneration algorithm, and empirically compares its efficiency and effectiveness to a state-of-art test data generation technique. The results show that there exists an attractive trade-off between efficiency and effectiveness of test data regeneration. The cost can be reduced by two orders of magnitude for some cases, while achieving competitive structural coverage and mutation score. Test data regeneration is less affected by the size of input domain compared to existing test data generation techniques that suffer with significantly large input domain.

10.2 Directions of Future Work

The future work will consider the use of more sophisticated modification operators and the characteristics of input domains for which this approach is particularly suitable. It may be possible to design new modification operators that are more effective against spiky search landscape. This also relates to search space smoothing studied in meta-heuristic optimisation. The paper only aims to generate an alternative test suite from the one that is already available and, more importantly, achieves certain test objectives. However, it may be possible to utilise test data regeneration in order to improve an existing test suite so that it achieves the known test objective, or even additional test objectives.

Acknowledgements:
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) and also by the kind support of DaimlerChrysler

Berlin and Vizuri Ltd., London. Simon Poulding provided important insights to the discussion of the distance-value hypothesis and test data regeneration. Tao Xie provided valuable comments on the way the empirical evaluation was presented and evaluated. Authors also thank the referees for their insightful comments, which helped greatly to improve the paper.

11. REFERENCES

[1] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. Automatic test case optimization: A bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[3] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Software Testing, Verification, and Reliability*, 15(2):73–96, 2005.

[4] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 82–91, New York, NY, USA, 2006. ACM.

[5] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[6] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.

[8] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[9] R. Feldt., R. Torkar, T. Gorschek, and W. Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *ICSTW '08. IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 178–186, April 2008.

[10] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems Software*, 38:235–253, 1997.

[11] M. J. Gallagher and V. Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

[12] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 155–164, New York, NY, USA, 2007. ACM.

[13] 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*, pages pp. 73–83. ACM Press, 2007.

[14] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, and S. Spoon. Regression test selection for java software. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 312–326, Tampa Bay, FL, October 2001.

[15] R. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

[16] B. Jones, H. Sthamer, X. Yang, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on Software Quality Management*, pages 435–444, Seville, Spain, 1995.

[17] B. Jones, H.-H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996.

[18] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[19] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, and R. Gupta. Data dependence based testability transformation in automated test generation. In 16^{th} *International Symposium on Software Reliability Engineering (ISSRE 05)*, pages 245–254, Washington, DC, USA, Nov. 2005. IEEE Computer Society.

[20] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *GECCO '07: Proceedings of the 9th annual onference on Genetic and Evolutionary Computation*, pages 1098–1105, New York, NY, USA, 2007. ACM.

[21] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707+, February 1966.

[22] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system: Research articles. *Software Testing, Verification, and Reliability*, 15(2):97–133, 2005.

[23] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[24] P. McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Technical Report CS-07-14, Department of Computer Science, University of Sheffield, 2007.

[25] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24, Portland, Maine, USA., 2006.

[26] C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, Dec. 2001.

[27] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[28] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, January 1999.

[29] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, 1996.

[30] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[31] H. Pohlheim and J. Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1795, San Francisco, CA 94104, USA, 13-17July 1999. Morgan Kaufmann.

[32] X. Qu, M. Cohen, and K. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 255–264, Oct. 2007.

[33] G. Rothermel, M. Harrold, J. Ronne, and C. Hong. Empirical studies of test suite reduction. *Journal of Software Testing, Verification, and Reliability*, 4(2), December 2002.

[34] G. Rothermel, M. Harrold, J. von Ronne, C. Hong, and J. Ostrin. Experiments to assess the costbenefits of test-suite reduction. Technical Report GIT-99-29, College of Computing, Georgia Institute of Technology, 1999.

[35] 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 *Proceedings of International Conference on Software Maintenance 1998 (ICSM1998)*, pages 34–43, 1998.

[36] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *Proceedings of IEEE Congress on Evolutionary Computation, 2007*, pages 438–444, 2007.

[37] R. Sagarna and J. A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412, 2006.

[38] H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[39] M. Tlili, S. Wappler, and H. Sthamer. Improving evolutionary real-time testing. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1917–1924, New York, NY, USA, 2006. ACM.

[40] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, March 1998.

[41] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.

[42] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of 13th IEEE International Conference on Automated Software Engineering 1998*, pages 285–288, Oct 1998.

[43] N. Tracey, J. Clark, J. McDermid, and K. Mander. *A search-based automated test-data generation framework for safety-critical systems*, pages 174–213. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[44] 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*, 43(14):841–854, 2001.

[45] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275 – 298, 1998.

[46] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality*, 6:127–135, 1997.