KD-ART: Should we intensify or diversify tests to kill mutants?

Matthew Patrick[a,*], Yue Jia[b]

[a]*Department of Plant Sciences, University of Cambridge, CB2 3EA*
[b]*CREST, University College London, WC1E 6BT*

Abstract

Context: Adaptive Random Testing (ART) spreads test cases evenly over the input domain. Yet once a fault is found, decisions must be made to diversify or intensify subsequent inputs. Diversification employs a wide range of tests to increase the chances of finding new faults. Intensification selects test inputs similar to those previously shown to be successful.
Objective: Explore the trade-off between diversification and intensification to kill mutants.
Method: We augment Adaptive Random Testing (ART) to estimate the Kernel Density (KD–ART) of input values found to kill mutants. KD–ART was first proposed at the 10th International Workshop on Mutation Analysis. We now extend this work to handle real world non numeric applications. Specifically we incorporate a technique to support programs with input parameters that have composite data types (such as arrays and structs).
Results: Intensification is the most effective strategy for the numerical programs (it achieves 8.5% higher mutation score than ART). By contrast, diversification seems more effective for programs with composite inputs. KD–ART kills mutants 15.4 times faster than ART.
Conclusion: Intensify tests for numerical types, but diversify them for composite types.

Keywords: mutation analysis, adaptive random testing, intensification and diversification

1. Introduction

Kernel Density Adaptive Random Testing (KD–ART) is a new approach for improving the effectiveness of random testing by taking advantage of run-time test execution information. KD–ART has two stages: it starts by applying the traditional ART algorithm to generate test inputs that cover the input domain as evenly as possible, whilst collecting information about the run-time properties of these tests; then it applies Kernel Density Estimation (KDE) [1] to generate new test inputs according to the distribution of tests that were found to have useful run-time properties. Random Testing (RT) aims to reveal faults by executing a wide range of values from the input domain of the program under test [2]. Although this idea is simple, research and industry suggest RT can be effective and scalable [3, 4]. Its biggest drawback is the cost of evaluating all the test cases. We address this problem by using KD–ART to select test inputs which are more likely to find faults.

*Corresponding author
Email addresses: mtp33@cam.ac.uk (Matthew Patrick), yue.jia@ucl.ac.uk (Yue Jia)

Adaptive Random Testing (ART) is an improved form of RT, which maximises diversity by distributing the test cases evenly over the input domain [5, 6]. The original version of this algorithm repeatedly samples a small set of candidate tests, at each step selecting the one which is the farthest away from the test inputs executed so far. This is typically measured using the Euclidean distance metric (although other distance metrics are possible too). ART is based on the assumption that test inputs which cause program failures are often located in contiguous input regions [5, 7, 8]. As a result, a set of evenly selected test inputs are more likely to find the first fault faster than uniformly sampled test data. ART has been found to have a higher fault detection probability than RT [5] and it is up to 52% more effective (34% on average) at detecting the first failure [9]. In addition, Chen et al. [10] use structural and mutation analysis metrics to show that tests selected by ART achieve higher code coverage than RT. As well as being used to improve the efficiency of unit testing, ART has been applied to assist with the integration and regression testing of large software systems [11]. In particular, it can serve as an enhancement for Combinatorial Interaction Testing [12, 13].

Despite the advantages gained by ART, the technique still has some limitations. ART does not collect information from run time executions when deciding which test inputs to select. This limits the power of ART as its fault detection ability is based solely upon the size and shape of common failure patterns [6]. Chen and Merkel [14] showed that it is not possible to design a technique using just this information such that it finds the first failure after less than half the test cases as random testing. By contrast, when the source code or the specification of the program under test is available, additional information can be collected using coverage metrics that helps improve the efficiency of the test suite beyond this limit [10]. We designed two new strategies for the selection of test inputs according to the distribution of useful tests: intensification and diversification. The intensification strategy favours new test inputs that are close to the tests previously shown to be useful. It is based on the intuition that effective regions of the input domain for finding faults tend to overlap and/or be in close proximity to each other [15]. The diversification strategy favours test inputs that are far away from the tests previously shown to be useful. It follows the original intuition of ART that increasing the diversity of the test suite improves its effectiveness at detecting faults. Both strategies have been implemented as part of our KD–ART technique.

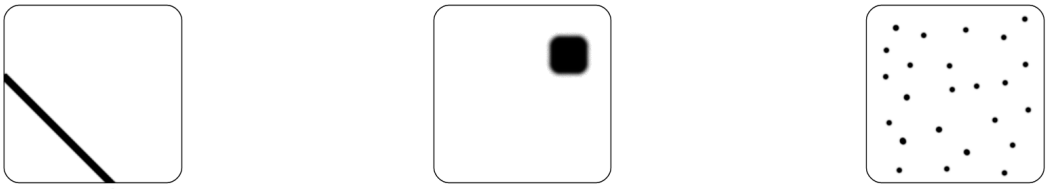
Previously, we evaluated the two KD–ART strategies by applying them to programs with input parameters that have simple numeric data types (integers and floats). Our work was published at the 10th International Workshop on Mutation Analysis [16]. In this new paper, we extend our work to consider whether KD–ART is also applicable to programs with input parameters that have composite data types (such as arrays and structs). Although there is substantial literature on the application of various ART techniques to numerical programs, there has been little work on programs with composite data types. In particular, this paper represents the first attempt to apply ART techniques to C programs with input parameters that have composite data types. We have extended our previous work to compare the results of the experiments for numerical programs with programs that have composite data types. We compared the effectiveness of KD–ART using mutation analysis, which has been applied in a number of other ART empirical studies [5, 8, 10]. Mutation analysis was previously used to evaluate the effectiveness of ART, but in this paper we take information about the

values found to kill mutants and use it to guide ART in the selection of new test inputs.

The paper is organised as follows: Section 2 provides some background, Section 3 explains the algorithm and Section 4 explains how it can be adapted for application to programs with composite data types; We introduce our research questions and experiments in Sections 5 and 6; The results are analysed in Section 7, the threats to their validity in Section 8; We describe related work in Section 9 and present our conclusions in Section 10.

2. Background

ART was first proposed in 2004 by Chen et al. [5]. It was inspired by research into the characteristics of fault exposing regions in the input domains of numerical programs [17, 7]. Studies suggest the regions from which test inputs can be selected to reveal faults are often formed in the shape of contiguous 'strips' or 'blocks' (see Figure 1). The strip pattern is typical when a domain error causes the input domain to shift so that values towards the edge of the correct range result in the program following the wrong path [17]. By contrast, the block pattern occurs when the correct path is followed, but a computation error causes an assignment to result in incorrect output for a closely related set of input values. Some failures occur in a point pattern, whereby contiguous input values do not reveal the same fault. For strip or block patterns, the likelihood of finding the first failure may be improved compared to uniform random testing by distributing the test cases as evenly as possible over the input domain. However, for point or other complex patterns, this might not be helpful.



Strip Pattern Block Pattern Point Pattern

Figure 1: Three Common Failure Patterns [18]

Algorithm 1 illustrates a simple implementation of ART. It selects the first test uniformly from the input domain, then the following test inputs based upon distance measurements. Let T_a and T_b be two test inputs which contain n input values. The Euclidean distance between two test inputs is measured using Equation 1. In each iteration, ART samples 10 candidate test inputs and measures their distance to the previous tests. Only the candidate with maximum distance to the previous test inputs is included in the test suite [5].

$$distance(T_a, T_b) = \sqrt{\sum_{i=1}^{n}(T_{ai} - T_{bi})^2} \quad (1)$$

Until now there have been no attempts to improve ART by adding additional information about the program execution. Previous research has aimed at achieving one of two research

Algorithm 1 Algorithm for ART

```
1: Generate test input t_0 uniformly at random
2: T = {t_0}
3: repeat
4:    maxDist = −∞
5:    for i ∈ 1...10 do
6:       Generate test input t_i at random
7:       if dist(t, T) > maxDist then
8:          maxDist = dist(t_i, T)
9:          t_max = t_i
10:      end if
11:   end for
12:   T = T ∪ {t_max}
13: until some termination condition is reached
14: return T
```

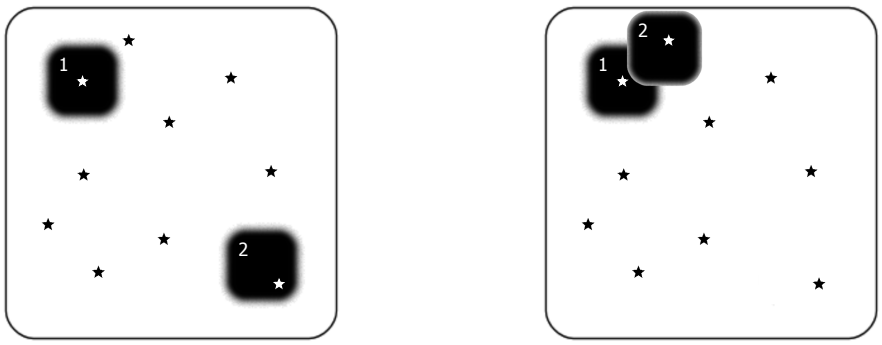
goals: improving the diversity of the test selection [19, 20, 21] and reducing the overhead computational cost of ART [22, 23, 24]. None of these research efforts used run-time information collected during the execution of the test inputs. Such information can be compared against a number of criteria (control/data flow, specification etc.) and it makes sense to use it to improve the effectiveness of the test suite.

3. Algorithm

In this paper, we propose a new algorithm called Kernel Density ART (KD–ART). KD–ART enhances the effectiveness of ART by taking advantage of additional white box information to guide the selection of test cases towards those that are likely to be more effective. KD–ART generates test inputs according to a probability distribution over the input domain, which is weighted to reflect test inputs previously been shown to be useful in finding faults. This information may be gathered using feedback from the people doing the testing or derived from test adequacy metrics such as branch coverage and mutation score. KD–ART uses this information to diversify or intensify the search for additional test inputs.

Diversification follows the same intuition as traditional ART. New test inputs are preferentially selected to be as far away from the previous successful test inputs as possible. The idea is that this reduces the risk of detecting the same faults as before, whilst increasing the probability of test inputs detecting new faults. The motivation behind this technique is supported by a number of researchers, including Chen et al. [6]. Intensification represents the opposing view point to diversification and ART. Rather than selecting test inputs with the aim of increasing diversity in the test suite, intensification generates new test inputs so that they are close to existing tests that have been shown to be useful in detecting faults. The idea is to take advantage of the effective input values discovered previously. The conditions required to reach one fault, cause a difference in the program state and propagate its effect to the output are likely to be relevant for other faults in the program.

Diversification and intensification make sense when used in different situations (see Figure 2). They both make assumptions about failure patterns in the input domain and have the potential to outperform random testing. However, it is not trivial to decide when each technique should be used and using wrong technique is likely to be less efficient than selecting test cases at random. Figure 2a shows 10 test cases (identified with stars) and two failure regions (block patterns) associated with faults. If the first test case detects fault 1, it is best to select the one farthest away next (diversification). By contrast in 2b, it is advantageous to select the one nearest the test case already selected. In both examples, random testing only has a 10% chance of detecting fault 2 on the next test case, whereas intensification or diversification is guaranteed to detect the fault. Diversification works best when failure regions are far apart (or spread evenly throughout the input domain), which may happen if they are unrelated. Intensification is more effective when regions overlap (or are near each other), perhaps because they share some of the same branch conditions.



(a) Diversification is the best choice (b) Intensification is the best choice

Figure 2: Intensification vs Diversification

At the beginning of the test data generation process, there is not enough information to successfully intensify or diversify the search, so KD–ART starts by applying traditional ART to select test inputs (see Algorithm 1). For every test input that is selected, KD–ART requires a measure of how useful it is for detecting faults. Although other metrics are possible, we have chosen to implement this using the mutation score, as it has been shown to be a realistic measure of fault finding effectiveness [25]. KD–ART only considers test inputs which kill mutants that have not been killed before. This reduces the number of mutants to which each test input must be applied and avoids biasing the distribution towards test inputs that target easy to kill mutants. After a certain mutation score is reached, ART is stopped and KD–ART starts to select test inputs. For each test input selected, KD–ART determines whether it kills any new mutants. If it does, then it is added to a list of useful mutant killing tests that are use in guiding the generation and selection of new test inputs.

Algorithm 2 Algorithm for KD–ART (intensify)

Require: T is the set of test inputs selected by ART,
M is the set of mutants for the program under test

1: $T_{mut} = \{\}$
2: $\forall m \in M : killed_m = false$
3: **for** $t_i \in T$ **do**
4: **if** $\exists m \in M : kills(t, m) \wedge \neg killed_m$ **then**
5: $T_{mut} = T_{mut} \cup \{t_i\}$
6: $killed_m = true$
7: **end if**
8: **end for**
9: **repeat**
10: $maxDensity = -\infty$
11: **for** $i \in 1 \ldots 10$ **do**
12: Generate test input t_i at random
13: **if** $density(t, T_{mut}) > maxDensity^*$ **then** $\#$ $< minDensity$ for diversify
14: $maxDensity = density(t_i, T_{mut})$
15: $t_{max} = t_i$
16: **end if**
17: **end for**
18: **if** $\exists m \in M : kills(t, m) \wedge \neg killed_m$ **then**
19: $T_{mut} = T_{mut} \cup \{t_{max}\}$
20: $killed_m = true$
21: **end if**
22: **until** some termination condition is reached
23: **return** T_{mut}

Algorithm 2 describes the process used by KD–ART to produce additional test inputs. As with ART, KD–ART uses rejection sampling. At each iteration, 10 test inputs are generated uniformly at random from the input domain and only one of the inputs is selected to be included in the test suite. KD–ART determines which test input to be included by applying KDE to discover how similar they are to the previous tests. The version of KD–ART described in Algorithm 2 applies intensification to select the test input that has the highest probability density estimation. This makes it more likely to select new test inputs that are closer to existing tests that have been shown to kill mutants. By contrast, diversification provides a higher probability of selection to test inputs that are farther away. It can be implemented through a simple change to line 13 of the algorithm, by selecting test inputs that have the minimum rather than maximum probability density estimation.

3.1. Kernel Density Estimation

The probability distribution from which test cases are selected in our algorithm is generated using Kernel Density Estimation (KDE) [26], a technique for estimating probability

density functions of continuous random variables, without making any assumptions about their underlying distribution. KDE is preferable to nearest neighbour distance because it takes into account the distance to multiple previous test cases. It is also more useful than the average distance to all previous test cases because it emphasises the closest test cases more strongly. In other words, it is an ideal compromise between these two metrics.

KDE determines the probability of a new event by summing up the densities of previous events. The densities are weighted according to a kernel function, centred on the event whose probability is to be estimated. We use a Gaussian kernel (see Equation 2) because it is unimodal and radially symmetric, i.e. the influence of known events is the same in every direction and there is only one location that has maximum weighting. The probabilities of nearby events have greater influence in estimating the density of the new event. This is sensible because information gained from these events is likely to be more relevant.

$$K(x) = \frac{1}{\sqrt{2\pi}} exp\left(-\frac{1}{2}x^2\right) \tag{2}$$

KDE is used by KD–ART to determine how similar a new test input is to those that have previously been found to be useful (see Equation 3). This technique is more effective than simpler metrics, such as nearest neighbour distance, because it takes into account the densities of all the test inputs that are used. Let T_1 to T_n be the set of test inputs, selected from those that were generated by ART, according to some evaluation criteria (e.g. mutation analysis). The distance between new and existing test inputs is calculated in terms of their input values, using the Euclidean metric described in Section 2. KDE determines the probability that a new test input ($T_{(n+1)}$) belongs to the same distribution as the previous set of tests by summing up the densities for all the previous test inputs and giving more weight to those that are similar to the new one.

$$density(T_{(n+1)}) = \frac{1}{n}\sum_{i=1}^{n} K\left(\frac{T_{(n+1)} - T_i}{h}\right) \tag{3}$$

As well as the choice of kernel, the success of KDE depends on a smoothing parameter (h), known as the bandwidth. The higher the bandwidth, the smoother the probability estimation (see Figure 3). However, the smaller the bandwidth, the more precisely the estimation fits the existing data. A number of techniques have been devised for calculating a suitable bandwidth that estimates the probability density distribution without over-fitting or over-generalisation. Yet, for our application, we are not interested in representing the underlying density distribution precisely, since typically our samples of the distribution will be very sparse. Instead, it is more important that the estimated density for all the potential new test inputs is sufficient for comparison within the precision allowed by the data type. For this reason, we have manually tuned the bandwidth to an unusually high value of 10.

We tuned the bandwidth in accordance with previous research in Adaptive Random Testing. In particular, Chan et al.'s work [27] on restricted random testing techniques assumes failure inducing regions cover 0.1% of the input domain. The bandwidth should be large enough to take into account the parameter values of mutant-killing test cases that

have predicted failure regions which include the candidate tests. However, the bandwidth should not be too large that the Kernel Density Estimation is imprecise and previous test cases which have no relation to the candidate tests are allowed too much influence.

The limiting factor on the distance with which test cases are compared is the exponential term in Equation 2. If the value of the term inside the exponential is sufficiently large, the result of the kernel evaluation becomes zero (due to the limits of numerical precision), thus eliminating that point from the density calculation. The smallest representable double in the IEEE 754 Standard for Floating-Point Arithmetic [28] is 2.2×10^{-308} and we can use this to set the bandwidth. Out of the programs featured in our experiments (see Table 1), the largest input domain extends between -300000 and 300000 (for bessj0). Putting these values into Equation 2 gives us a bandwidth of $0.001 * 600000/\sqrt{-2 * \ln(2.2 \times 10^{-308})} = 7.97$. We round this value up to give us the bandwidth of 10 that is used in our experiments.

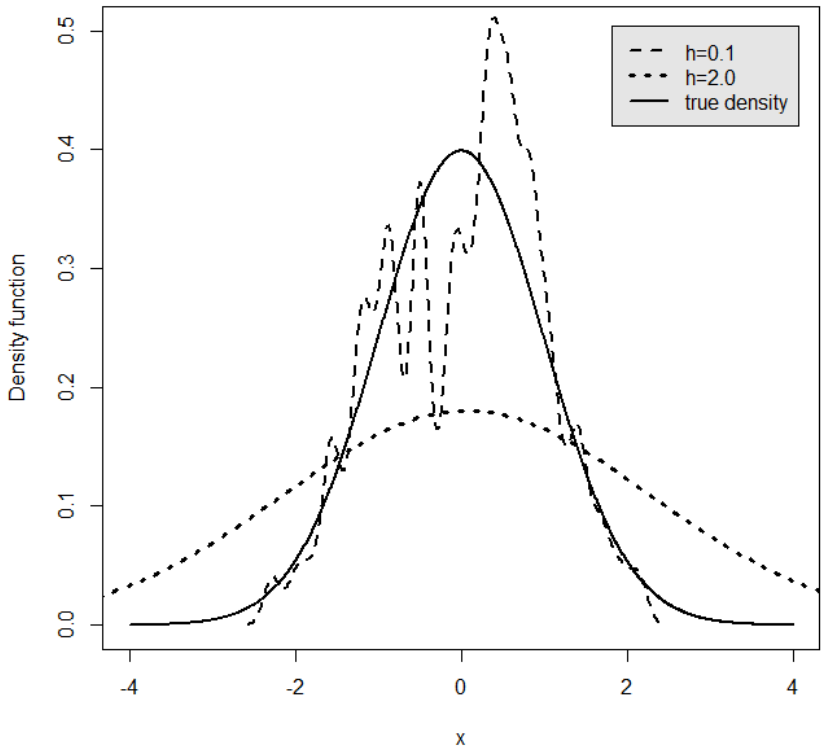


Figure 3: The effect of bandwidth on KDE for a standard Gaussian function

4. Testing using Composite Data Types

Composite data types differ from primitive types in that their instances are composed of multiple fields. Each field represents either a primitive value (integer, boolean etc.), or another instance of a composite data type. Unlike with primitive values, the distance between two composite input values is not trivial to calculate. We need to take into account differences in their types, the types of the values their composite fields refer to and the differences between each of the primitive fields the composite fields contain. Any non-zero value for each of these measurements should increase the distance value, i.e. the distance metric should be monotonically increasing. However, changes to some values may be more significant than others and, even if a direct comparison of two composite values suggests the differences between them are large, they could still be related by a subtle transformation (such as the reordering of values in an array). The challenge is to create a metric that is computationally tractable, yet represents the distances between values in a realistic way.

Ciupa et al. [29] proposed a recursive technique for measuring the distance between two composite objects (p and q). A monotonically increasing distance function (Equation 4) is produced by combining the distance measures from three different components: type distance measures the difference between the types of p and q, independent of their actual values; field distance measures the difference between the composite values stored within p and q; and elementary distance measures the difference between their primitive values.

$$\begin{aligned} p \longleftrightarrow q = combination(&elementaryDistance(p, q)), \\ &typeDistance(type(p), type(q)), \\ &fieldDistance(\{[p.f \longleftrightarrow q.f] \\ &\quad | f \in fields(type(p), type(q)\}) \end{aligned} \tag{4}$$

(where $fields(t1, t2)$ is the set of fields shared between type $t1$ and type $t2$)

The elementary distance between two numbers is typically calculated using the Euclidean metric (as is described in Section 2). Characters and Booleans can be compared exactly and a distance penalty introduced if they do not match [29]. There are many metrics available for comparing strings, but one of the most popular is the Levenshtein distance [30], which computes the minimum number of insertions, deletions or substitutions that would be required to convert one string into another. Ciupa et al. [29] introduced a simple metric for including references in the distance calculation: if the references are identical, the distance between them is zero; if the references a different, but none of them is void, a particular penalty value is applied; if exactly one of the references is void, another (potentially different) penalty value is applied; and if both references are void, they are considered equal and the distance between them is zero. Ciupa et al.'s reference distance metric is easy to calculate, but it does not take into account any differences in the data the references point to. Part of the reason for this is that Ciupa et al. target their metric at programs written in Java, where the use of references in composite data types is rare. For languages such as C, where pointers are commonplace (even necessary), a slightly different metric must be used.

Another difference between Ciupa et al.'s distance metric for composite data types and the one used in this paper is that in object-oriented programs it is possible to measure the differences between types in terms of their path distance along the inheritance hierarchy [29]. For non object-oriented programs, such as those written in C, there is no inheritance hierarchy available. Instead, we need to look inside the data that a struct's pointers are referring to, if we are to determine how their types will appear dynamically. In addition to the path distance, Ciupa et al. measure the distance between types by counting the number of non-shared fields (see Equation 5). We can still use this technique to measure dynamic type difference in C, by recursively analysing the code to determine the structure of the data being pointed to, but this would be challenging to do automatically. Instead, we identify an appropriate structure for the input data manually, using insight gathered from the specification. We can then compute the distances between the composite values logically, without having to worry about their pointer structure, by treating each reference to a pointer as if it contains all the information necessary to infer its type.

$$typeDistance(t,u) = \sum_{f \in nonShared(t,u)} weight_f \tag{5}$$

Finally, the field distance is determined by recursively calculating distances for all the fields that are contained within the composite value (see Equation 5). Comparing each field in turn, we compute its combined distance as before, using Equation 4. So as to avoid biasing the metric towards composite types that contain more fields, we take the arithmetic mean of these combined distances. For each distance metric (elementary, type and field distance), it is helpful to have the flexibility to make certain fields contribute differently to the distance measure. We do this by introducing the option to set a weight value for each field. There is also a coefficient for the contribution of each distance (elementary, type and field). The final distance value is calculated by adding up the sum of these distance metrics.

$$fieldDistance(p,q) = \overline{\sum_{f}} weight_f * p.f \longleftrightarrow q.f \tag{6}$$

4.1. Worked example

Let us take one of the programs used in our experiments as an example. The main method of chunkyBar takes as its input a struct containing a list of 'chunk' struct data and an integer value (see below). The integer value is used to store the maximum number of chunks allowed for the bar. Each 'struct chunk' data item contains two integer values for the offset and length of the bar, and a pointer link to the next chunk.

```
typedef struct {
    var_chunk_t * first_chunk;
    unsigned int max;
}chunkybar_t

typedef struct
{
    int offset;
    int len;
    var_chunk_t * next;
}var_chunk_t
```

To construct a valid input for the chunkybar_t data type, we randomly generate an integer number, assign it to the max attribute, and then generate a fixed number of var_chunk_t data items. We randomly generate the offset and length attribute of each var_chunk_t and then add it to our chunkybar_t through the construction method provided. For example, to generate a chunkybar with four chunks, we need 9 integer values from the input domain.

In our example, the distance between two chunkybar_t values is computed using the recursive *combination* formula. We determine the $fieldDistance$ for each var_chunk_t by calculating the *elementaryDistance* for the offset and len value in each data item, and then the mean of these values is taken and added to the means of every other data item. Finally, we take the mean of this sum and add it to the *elementaryDistance* for the max attribute of the chunkybar_t. We do not need to worry about the *typeDistance* in our experiments with this program, as all the composite values we use have the same dynamic type (i.e. they have the same number of attributes and the var_chunk_t array is the same size).

In our experiments, we require three more integer values to test two search functions that have additional input parameters. The first function, chunky_have, takes a chunkybar_t composite value, along with two numeric values (length and offset) as inputs. It searches for valid chunks within the search domain specified by the length and offset variable. The second function, chunky_get_incomplete, takes a chunkybar_t composite value, along with an additional numeric value (the maximum length of the search). It searches for any incomplete chunks within the valid search domain. We generate these values at random as usual, and incorporate them into the distance measure by adding up their *elementaryDistance* values.

5. Research Questions

This section presents the research questions concerning the effectiveness and the efficiency of the Kernel Density Testing approach, for which Section 7 provides the answers.

RQ1: Are KD–ART techniques (intensify and/or diversify) more effective at killing mutants than ART?

A test data generation technique is considered to be more effective if it kills more mutants with the same number of test inputs. Mutation analysis is used to determine

whether KD–ART techniques are more effective than ART for test suites of 5000 test inputs. It is important to note which KD–ART technique is the most effective, as this indicates whether intensification or diversification should be used. If the mutation score achieved by KD–ART (intensify) or KD–ART (diversify) is higher on average, then it can be considered to be more effective.

RQ2: Does it take longer to generate test suites using KD–ART (intensify and/or diversify) than ART?
To answer this research question, we report the average CPU time required to run 5000 test inputs. ART generates test suites by computing the distance between each test input. By contrast, KD–ART must also collect run-time information about the program under test. It seems that KD–ART is likely to be more expensive than ART. However, since KD–ART only computes distances for test inputs that have been used to kill mutants, the answer to this question is not immediately obvious.

RQ3: What is the best point to switch from the default ART to KD–ART (intensify and/or diversify)?
We investigate the optimal point for switching between ART and KD–ART. Both KD–ART (intensify) and KD–ART (diversify) rely on the test inputs selected by ART to construct a test suite. Allowing ART to continue until it identifies tests with a high mutation score provides a good starting point for KD–ART. However, since fewer mutants remain to be killed, the difference between KD–ART and ART may not be as apparent. We investigate the correlation between the number of lines-of-code and mutants of the program under test and the optimal switch point. If such a correlation exists, it may be used to choose a suitable switch point in advance.

RQ4: Do programs with input parameters that have composite data types have an effect on the most suitable choice of technique for testing?
Finally, we apply ART and KD–ART to four new programs which have input parameters that have composite data types. In order to show our techniques are robust and versatile, we need to demonstrate that they work on programs with a variety of types, both composite and simple. However, it will be interesting to find out whether the advantages gained by KD–ART are increased or decreased when applied to programs with composite data types compared with those that only have input parameters with simple numeric data types. Programs with composite data types can be very different from numerical programs: they are often written in a more modular way, with an increased number of branches but fewer numerical calculations. We will also consider whether ART and KD–ART take any longer on these kinds of programs.

6. Experiment Setup

This section explains the methodology we used to perform our empirical study in order to answer the four research questions set out in the previous section.

6.1. Programs Studied

For the numerical part of out study, we used eight benchmark C programs that have previously been used in empirical studies of ART [5, 8]: `erfcc`, `probks`, `bessj`, `plgndr`, `airy`

and `gammq` are numerical functions selected from a numerical recipes book [31]; `Triangle` is a classification program for isosceles and equilateral triangles; whilst `TCAS` is a traffic collision avoidance system downloaded from the SIR repository [32]. For the numerical recipes functions, we chose to use the input domain reported in a previous study [5]. For the remaining programs, the choice of input domains was not documented in the previous literature, so we examined the source code to identify input domains that cover all possible input conditions. The benefit of using programs from literature is that it allows comparisons to be made with the previous results, so that we can verify the results of our own experiments.

The second set of programs used in our experiments was chosen to represent the various issues related to testing programs with composite input parameters. All of the subjects are open source C programs, selected from GitHub. They have recently been used in a study on memory mutation [33]. The four subjects are from different areas of application: textureAtlas is designed to store and manipulate multiple textures efficiently for graphics libraries such as OpenGL; chunkybar implements multi-piece progress bars, which are used in bittorrent clients; pseudoLRU is a C implementation of the Pseudo-LRU cache algorithm created to improve the Least Recently Used (LRU) algorithm; and QPHashMap is a hashmap data structure that uses quadratic probing for managing collisions. Information about all the selected programs, including their name, size, mutants generated and input domain is shown in Table 1. The table is sorted by size of program, in lines of code.

6.2. Mutants Used

In this paper, we use artificially generated faults (mutants) to evaluate ART and KD–ART against a wide variety of faults and we use the output of the original program as an automated oracle. We apply the open source C mutation testing tool MILU [34] to generate mutants from the selective operators proposed by Offutt et. al. [35]. MILU uses a novel 'test harness' technique to embed mutants and their associated test suites into a single-invocation procedure. The selective mutation operators were found to generate mutants that provide good coverage of faults generated by other operators [35]. Although, these five operators operators (ABS, AOR, LCR, ROR, and UOI) were designed for Fortran programs, they were found to be sufficient to achieve almost full mutation coverage. As a result, many subsequent authors [36] have used only operators from these five classes.

6.3. Experiments

We apply ART and KD–ART to select 5000 test inputs in each experiment and do this 20 times to produce an average mutation score. At each step, ART and KD–ART select one test input from 10 candidates sampled uniformly at random from the input domain. This number of candidates has been used before in empirical studies of ART [5]. We also record the time for applying ART and KD–ART and running the mutation analysis. The mutants for the numerical programs were generated and run on the Microsoft Azure Cloud platform using an A9 Compute Intensive Instance (16 cores and 112GB of memory), whereas the mutants with composite data type inputs were generated and run and all the tests were selected using an Intel Core i7 desktop with 8GB of memory.

Table 1: Subject programs ordered by Size in Lines of Code ('#LoC'). The '#Muts' column shows the number of first order mutants generated. The '#Selected Muts' column shows the number of non trivial and non equivalent first order mutants generated.

Programs	#LoC	#Muts.	Selected Muts.	Input Type	Input Domain–From	Input Domain–To
erfcc	14	144	105	float	(-30000)	(30000)
probks	22	120	71	float	(-50000)	(50000)
bessj0	28	276	225	float	(-300000)	(300000)
plgndr	36	371	297	int, int, float	(10, 0, 0)	(500, 11, 1)
airy	43	305	201	float	(-5000)	(5000)
triangle	75	488	400	int, int, int	(0, 0, 0)	(100, 100, 100)
gammq	106	521	337	float, float	(0, 0)	(1700, 40)
tcas	182	271	205	int, int, int, int, int, int, int, int, int, int, int, int	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	(1000, 1, 1, 50000, 1000, 50000, 3, 1000, 1000, 2, 2, 1)
textureAtlas	405	282	126	struct(int, int) x 5, int	((0, 0), 0)	((512, 512), 512)
chunkybar	451	316	290	int, struct (int) x 4, int, int, int	(0, (0, 0), 0, 0, 0)	(100, (100, 100), 100, 100, 100)
pseudoLRU	472	192	134	struct(int,int) x 5, int, int	((0, 0), 0, 0)	((100, 100), 100, 100)
QPHashMap	568	421	257	int, struct(int,int), x 5, int	(0, (0, 0), 0)	(100, (100, 100), 100)

Table 2: The Mutant Operators used in our experiments

Names	Description	Specific mutation operator
ABS	Absolute Value Insertion	$\{(e,\texttt{abs}(e)), (e,\texttt{-abs}(e))\}$
AOR	Arithmetic Operator Replacement	$\{(x,y) \mid x,y \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\} \wedge x \neq y\}$
LCR	Logical Connector Replacement	$\{(x,y) \mid x,y \in \{\texttt{\&\&},\texttt{\|\|}\} \wedge x \neq y\}$
ROR	Relational Operator Replacement	$\{(x,y) \mid x,y \in \{\texttt{>},\texttt{>=},\texttt{<},\texttt{<=},\texttt{==},\texttt{!=}\} \wedge x \neq y\}$
UOI	Unary Operator Insertion	$\{(v,\texttt{--}v), (v,v\texttt{--}), (v,\texttt{++}v), (v,v\texttt{++})\}$

7. Results and Discussions

7.1. Answer to RQ1: Are KD–ART techniques (intensify and/or diversify) more effective at killing mutants than ART?

RQ1 is evaluated by applying mutation analysis to compare the effectiveness of the two KD–ART techniques (intensify and diversify) with ART. KD–ART starts generating test inputs in the same way as ART, but then it switches to take advantage of information gathered about test inputs which were found to be successful at killing new mutants. To avoid bias in our experiments, we set the switch point to be the same for each program (we switch from ART to KD–ART once 50% mutation score has been achieved). The effects of this decision are later evaluated in RQ3. Our results show that KD–ART is typically more effective at selecting test inputs than ART. In addition, KD–ART (intensify) achieves a higher mutation score for more programs than KD–ART (diversify).

Figure 4 presents the median mutation scores achieved by KD–ART and ART over 20 trials (we present the median rather than the mean, so as to reduce the effect of outliers). The number of mutants killed continues to increase as new test inputs are added to the test suite (with the total number ranging from 1 up to 5000 test inputs). However, the effectiveness of each technique varies significantly according to the program under test. For 'bessj0' and 'erfcc', KD–ART (intensify) outperforms both KD–ART (diversify) and ART. Whereas for 'plgndr', KD–ART (intensify) is the least effective technique. The differences between the median mutation scores achieved by these techniques seems small for the majority of programs. We therefore turn to statistical analysis to compare them.

Table 3 shows the results of a Friedman test applied to the three techniques after 5000 test inputs have been executed. The Friedman test is a non-parametric test that can be used to determine whether there are any statistically significant differences in the median mutation scores achieved for each program. At the 95% confidence interval, the Friedman test indicates significant differences in median mutation score for five out of the eight programs. We investigate the differences using a Wilcoxon Signed Rank Test with Bonferroni correction for three-way comparison: KD–ART (intensify) achieves a significantly higher mutation score

than ART for 'erfcc' (p=0.026), 'bessj0' (p=0.00042) and 'gammq' (p=0.00028); KD–ART (diversify) achieves a significantly higher mutation score than ART for 'airy' (p=0.0197); ART achieves a significantly higher mutation score than KD–ART (intensify) and KD–ART (diversify) for 'plgndr' (p=0.00032 and p=0.04414 respectively).

Table 3: Mutation scores achieved after 5000 test input executions

Program	**Median**			**Friedman Test**	
	ART	KD–ART (intensify)	KD–ART (diversify)	χ^2-value	p-value
erfcc	0.705	0.876	0.705	16.00	0.000
probks	0.289	0.289	0.289	4.000	0.135
bessj0	0.929	0.962	0.840	39.52	0.000
plgndr	0.990	0.902	0.990	23.13	0.000
airy	0.995	0.995	1.000	18.38	0.000
triangle	0.921	0.923	0.910	1.595	0.451
gammq	0.985	1.000	0.760	40.00	0.000
tcas	0.920	0.922	0.922	0.767	0.681

In addition to running Friedman tests at the end of the testing process, we also compared the areas under the curves, with the number of tests on the x-axis and the mutation score on the y-axis (see Table 4). The maximum value for the x-axis was set as the minimum number of tests ART or the KD-ART techniques need to reach to the maximum mutation score. This provides a measure independent from the number of tests. The results of the Friedman Test on the area under the curve confirm those after 5000 test cases and they show significant differences for 'probks' and 'tcas' at the 95% confidence interval and 'triangle' at the 90% confidence interval. However, Wilcoxon Signed Rank Tests with Bonferroni correction do not show any additional significant pairwise differences: 'probks' p=[1.000, 1.000, 1.000], 'tcas' p=[1.000, 0.904, 0.190] and 'triangle' p=[0.147, 0.168, 1.000] for ART vs. KD-ART (intensify), ART vs. KD-ART (diversify) and KD-ART (intensify) vs. KD-ART (diversify).

Table 4: Proportional area under mutation score curves (until maximum achieved)

Program	**Median**			**Friedman Test**	
	ART	KD–ART (intensify)	KD–ART (diversify)	χ^2-value	p-value
erfcc	0.201	0.202	0.201	20.83	0.000
probks	0.225	0.225	0.225	7.429	0.024
bessj0	0.868	0.910	0.797	40.00	0.000
plgndr	0.963	0.902	0.990	23.13	0.000
airy	0.976	0.935	0.968	19.60	0.000
triangle	0.891	0.881	0.882	4.900	0.086
gammq	0.974	0.990	0.756	40.00	0.000
tcas	0.896	0.890	0.898	6.400	0.041

KD–ART achieves a higher mutation score than ART on four out of the five programs for which a statistically significant difference was observed. The remaining program, 'plgndr' has a statistically higher mutation score with ART than KD–ART. However, the size of this difference is small to non-existent; the median mutation score achieved by ART on 'plgndr' is exactly the same as that of KD–ART (diversify). KD–ART (intensify) achieved a significantly higher mutation score than KD–ART (diversify) on three out of the five programs ('erfcc', 'bessj0' and 'gammq'). On average, the mutation score for KD–ART (intensify) was 8.58% higher than ART and 7.24% higher than KD–ART (diversify). We can therefore say with confidence that KD–ART was more effective at killing mutants than ART. Of the two KD–ART techniques, KD–ART (intensify) was the most effective.

It is interesting KD–ART (intensify) is more effective than KD–ART (diversify) and ART, as this goes against a common perception in research that it is advantageous to diversify the test suite [6]. Certainly diversification can be useful and KD–ART (intensify) achieves this by initially operating the same way as ART, but once test inputs have been found that are useful at killing mutants, it makes sense to take advantage of this information. This corresponds well with research in the area of metaheuristic optimisation [37] that intensification and diversification must be used side by side to achieve an optimal result. Ideally we should seek to find a balance between these two strategies for test data generation.

7.2. Answer to RQ2: Does it take longer to generate test suites using KD–ART (intensify and/or diversify) than ART?

We evaluate RQ2 by recording the time taken by KD–ART (intensify), KD–ART (diversify) and ART to select and run each test input. Table 5 presents the average time required by each technique to select 5000 test inputs, along with the average time taken to run this test suite against the entire set of mutants. KD–ART requires just 5.98% of the time taken by ART to select test inputs; it only has to calculate distances for test cases that have been shown to be successful at killing mutants, whereas ART has to take into account all the test cases that have so far been executed. The difference between KD–ART (diversify) and KD–ART (intensify) is negligible. This is most likely because they are essentially the same process - one minimises the distance to existing test inputs, whereas the other maximises it. The vast majority of time (23.2 times that taken by ART) is spent running mutants.

It is debatable whether or not to include the time taken to run the test suite against the mutants into the total time required by KD–ART. Although in our experiments we have implemented KD–ART to guide test suite selection using mutation analysis, it can also be made to take advantage of other metrics such as branch coverage. Coverage metrics are frequently used in test data generation to evaluate the effectiveness of testing. If we assume coverage metrics are already being used, or information exists from other sources about the usefulness of certain input values (e.g. as a result of regression analysis), we only need to consider the time required by KD–ART to select a test suite.

As well as the time taken to generate test data, we should also consider the human effort involved. Mutation analysis can be applied automatically without human interaction, so it may be performed whilst the programmers are not working. KD–ART can find faults with fewer test inputs than ART and this reduces the human effort required to evaluate the

Table 5: Average time taken to run KDT and ART (in seconds)

Program	**Selecting test inputs**			**Running mutants**
	KD–ART (intensify)	KD–ART (diversify)	ART	
erfcc	0.878	0.890	1.132	104.3
probks	0.954	0.970	1.079	74.48
bessj0	0.328	0.350	1.209	192.6
plgndr	0.350	0.345	1.593	260.0
airy	0.265	0.268	1.284	269.6
triangle	0.485	0.479	2.160	314.1
gammq	0.397	0.434	1.411	330.2
tcas	0.480	0.505	61.01	99.37

outputs of each test input. There is a human cost to consider as well as a computational one and manual labour is typically more expensive than computational effort. Ultimately, these results show KD–ART is computationally efficient compared to ART when excluding the coverage calculation in KD-ART.

7.3. Answer to RQ3: What is the best point to switch from the default ART to KD–ART (intensify and/or diversify)?

RQ3 is evaluated by applying mutation analysis to compare the effectiveness of the two KD–ART techniques (intensify and diversify) with ART when switching between them at different points in their execution. A high mutation score is achieved for some programs (e.g. 'airy') after the first few test inputs, but the mutation score stays low for others (e.g. 'probks') even after 5000 test inputs are executed. Switching between ART and KD–ART at a fixed mutation score eliminates the exploration period for some programs, whereas it prevents switching entirely for others. We would like to see whether tuning the switch point for each program has a significant impact on the effectiveness of KD–ART.

Figure 5 shows the mean difference in mutation scores achieved after each test input is selected for KD–ART (intensify), KD–ART (diversify) and ART. This calculation is made by adding up the differences between the mutation scores of KD–ART and ART after each test input is selected and dividing the result by the number of tests. When the switch point is set too early, not enough test inputs have been evaluated for KD–ART to learn from effectively, but when switch point is set too late, there are not enough test inputs remaining for KD–ART to achieve significant and observable improvement. Table 6 shows that tuning the switch point between ART and KD–ART has a significant effect on the mutation score achieved for the diversification strategy at the 90% confidence interval, and for the intensification strategy at the 95% confidence interval.

Tuning the switch point has a greater effect on the mutation score achieved by KD–ART (intensify) than KD–ART (diversify). The tuned version of KD–ART (diversify) achieves a

Table 6: Student's t-test comparing optimal with fixed switch point for KD–ART

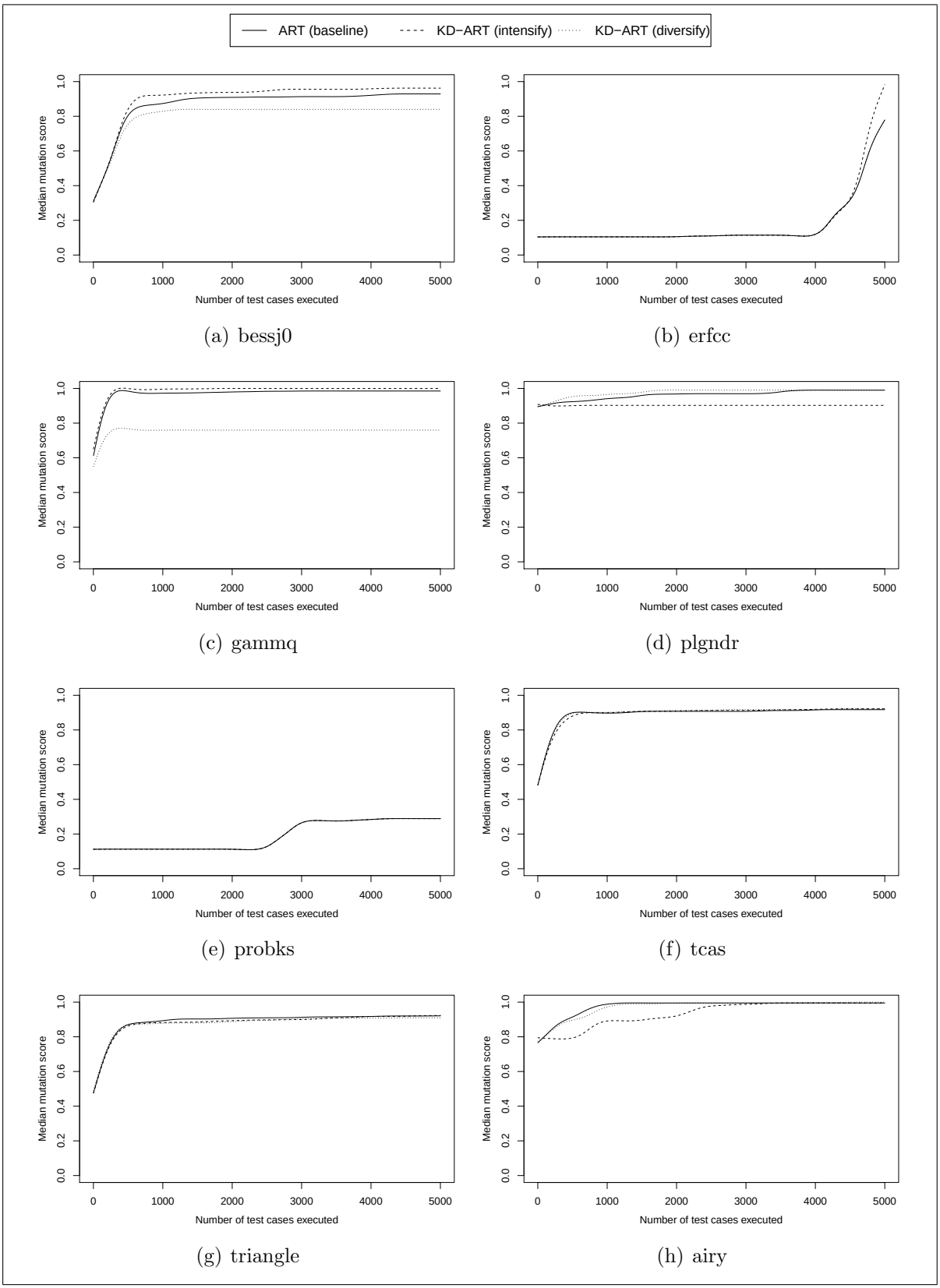
	Switch	Minimum	Maximum	Mean	p-value	Effect size
KD–ART (intensify)	Optimal	0.458	1.000	0.914	0.039	0.897
	Fixed	0.289	1.000	0.859		
KD–ART (diversify)	Optimal	0.404	1.000	0.864	0.074	0.743
	Fixed	0.289	1.000	0.801		

mutation score that is only on average slightly higher than that of the fixed (50% mutation score) version of KD–ART (intensify). If the switch point is tuned carefully, KD–ART (intensify) always outperforms KD–ART (diversify) and ART. The difficulty is in deciding when to switch. One way to address this problem is to switch between the techniques dynamically, but further work is required to achieve this. Until a technique is developed for dynamic switch point tuning, the tester can get some intuition as to a suitable switch point from their experience testing previous versions of their software or other similar programs.

Table 7 suggests that it may also be possible to predict the optimal switch point for KD–ART (intensify) using easily calculable properties of the programs under test. There is a 0.768 Pearson correlation coefficient between the number of mutants produced from a program and its optimal switch point (in terms of mutation score). The Pearson correlation coefficients between the optimal switch point and the number of lines of code (LOC) and the cyclomatic complexity (CC) is smaller at 0.529 and 0.569, but these values are still statistically significant at the 90% confidence interval. By contrast, the correlation coefficients for KD–ART (diversify) are much lower and they are not statistically significant. These three metrics (number of mutants, lines of code and cyclomatic complexity) are given merely as an example of how metrics can be used to help the tester identify suitable switch points for new programs they are testing. There are other metrics available that might give a stronger correlation, but these were chosen as they are easily calculable and commonly used. We can therefore have hope that it might be possible to select an optimal switch point in KD–ART (intensify) without computing the effectiveness of all the possibilities, but for the same reason we should probably avoid KD–ART (diversify).

Table 7: Program properties and optimal switch point

	KD–ART (intensify)		**KD–ART (diversify)**	
	Pearson's r	p-value	Pearson's r	p-value
Mutants	0.768	0.013	0.171	0.342
LOC	0.529	0.089	0.315	0.224
CC	0.569	0.071	0.202	0.316



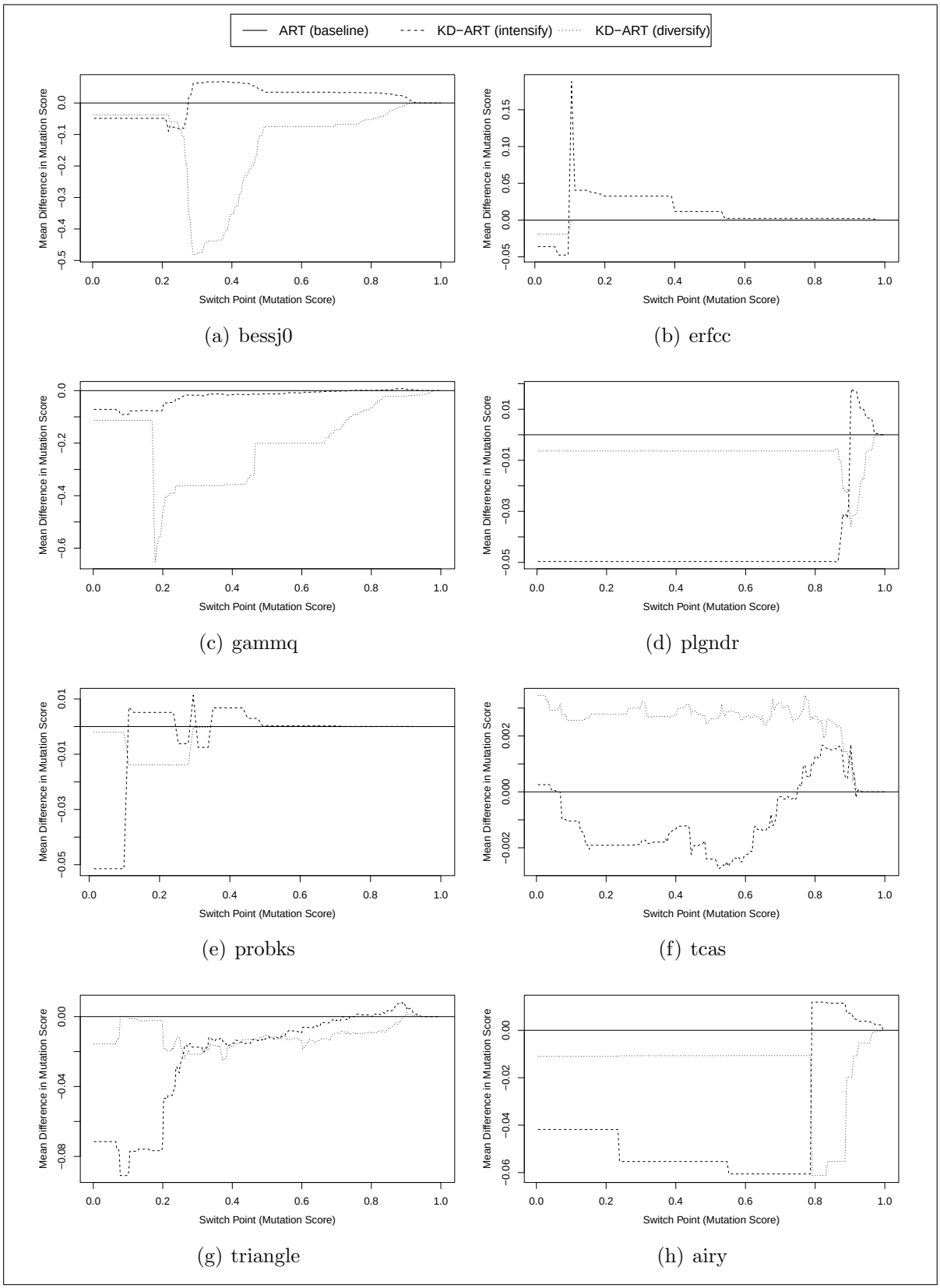
(a) bessj0 (b) erfcc

(c) gammq (d) plgndr

(e) probks (f) tcas

(g) triangle (h) airy

Figure 4: Mutation scores achieved with switching point at 50% mutation score (averaged over 20 trials)



(a) bessj0 (b) erfcc

(c) gammq (d) plgndr

(e) probks (f) tcas

(g) triangle (h) airy

Figure 5: Mean difference in mutation scores achieved for different switching points (averaged over 20 trials)

7.4. Answer to RQ4: Do programs with input parameters that have composite data types have an effect on the most suitable choice of technique for testing?

We evaluate RQ4 by running the experiments on a new set of programs that have composite input parameters. The aim is to determine if there are differences in the effectiveness of KD–ART on programs with composite and numerical data types. It was not necessary to run all 5000 test cases for each program, as the testing process had already reached its maximum mutation score long before this point (see Figure 6). We believe this is because the programs are implemented in an object oriented manner. Each method under test is smaller and simpler than the numerical functions used previously and hence their mutants are easier to kill. We therefore report the results of applying 100 test cases to each program.

Table 8 shows the mutation scores achieved by ART and KD–ART after 100 test cases have been executed for each of the programs with composite input parameters. This confirms the vast majority of mutants have been killed. A Friedman test indicates no significant difference in the mutation scores achieved by either of the KD–ART techniques or ART once 100 test cases have been executed. However, the mutation scores achieved for quadratic (0.809 with ART) are considerably lower than other programs (e.g. 0.984 for texture). This program seems harder to test than the others. We therefore extended our experiment to run 5000 tests on quadratic (the same as with the previous set of programs). ART and KD–ART (intensify) achieved a median mutation score of 1.000, whereas the median mutation score for KD–ART (diversify) was 0.954. A Friedman test on these results indicates a significant difference (p=0.009) and this is confirmed further by pairwise Wilcoxon Signed Rank Tests with Bonferroni correction (ART vs KD–ART (diversify) p=0.008, KD–ART (intensify) vs KD–ART (diversify) p=0.028), ART vs KD–ART (intensify) p=1.000).

Table 8: Mutation scores achieved after 100 test input executions

Program	**Median**			**Friedman Test**	
	ART	KD–ART (intensify)	KD–ART (diversify)	χ^2-value	p-value
chunkybar	0.960	0.964	0.966	0.560	0.756
pseudolru	0.993	0.993	0.993	2.80	0.247
quadratic	0.809	0.815	0.809	1.333	0.513
texture	0.984	0.984	0.984	1.182	0.554

In addition, Table 9 shows results for the areas under the mutation score curves (until the maximum mutation score is achieved). The Friedman test indicates significant differences for quadratic and the pairwise Wilcoxon tests confirm that the areas under the curves for KD–ART (diversify) are lower (ART vs KD–ART (diversify) p=0.000, KD–ART (intensify) vs KD–ART (diversify) p=0.000), ART vs KD–ART (intensify) p=1.000). This means KD–ART (diversify) increases the mutation score more slowly than the other two techniques and it may be important if for some reason, it is not possible to run all the test cases that are available within the time allowed. On the programs evaluated, this is unlikely to be a

problem, but for larger programs it might become significant. Both KD–ART techniques have a lower median area under curve than ART for pseudolru (as can be seen clearly in Figure 6c), but this difference is not consistent enough to be statistically significant.

Table 9: Proportional area under mutation score curves (until maximum achieved)

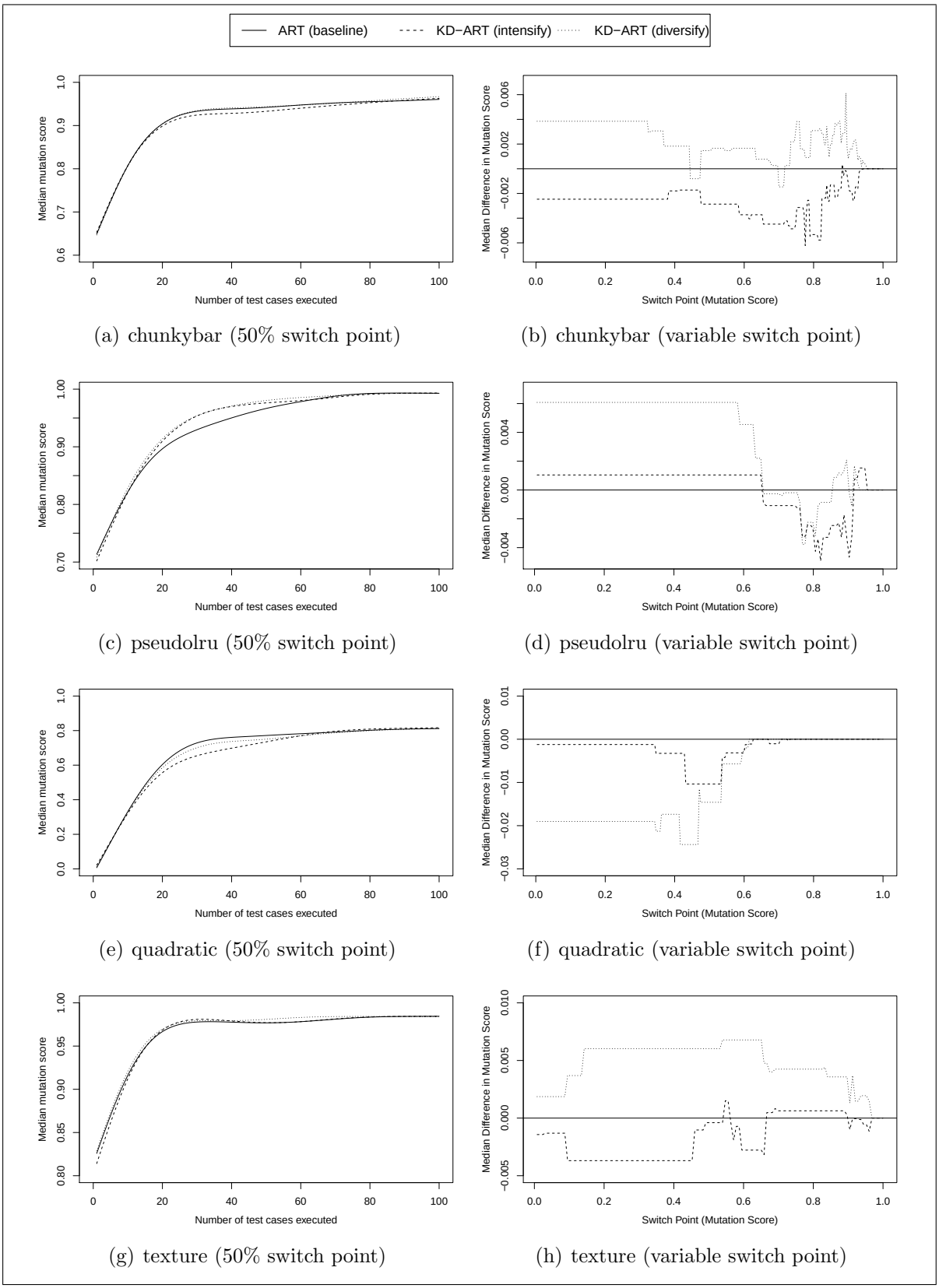
Program	**Median**			**Friedman Test**	
	ART	KD–ART (intensify)	KD–ART (diversify)	χ^2-value	p-value
chunkybar	0.914	0.910	0.915	1.600	0.449
pseudolru	0.936	0.942	0.950	0.900	0.638
quadratic	0.964	0.969	0.912	9.100	0.011
texture	0.959	0.964	0.966	2.100	0.350

Overall, our results suggest that KD–ART and ART can be used on real world C programs with complex data structures. However KD–ART was more effective at generating test data for the traditional numeric programs used in our experiments (the differences between KD–ART and ART for these programs were more significant).

Table 10 shows a Student's t-test applied to the results of the composite programs, excluding quadratic. There is no significant difference between the optimal and fixed switch points for KD–ART (intensify), but it is significant for KD–ART (diversify). This is the exact opposite of the result for the numerical programs. It suggests there is something characteristically different about programs with composite types. We can see a similar result in the experiments with Tcas, where diversify performs better than intensify. Tcas is similar to the composite programs in that it has a large number of input parameters and it uses them without performing many numerical calculations to select the execution path. It is often necessary to spread test cases broadly over the input domain to cover all the branch conditions in programs with composite types, whereas changing the input values slightly for numerical programs can have a large effect on the result of their numerical expressions.

Table 10: Student's t-test comparing optimal with fixed switch point for KD–ART

	Switch	Minimum	Maximum	Mean	p-value	Effect size
KD–ART (intensify)	Optimal	0.964	0.997	0.982	0.274	0.110
	Fixed	0.964	0.993	0.980		
KD–ART (diversify)	Optimal	0.972	0.999	0.987	0.001	0.460
	Fixed	0.966	0.993	0.981		



(a) chunkybar (50% switch point)

(b) chunkybar (variable switch point)

(c) pseudolru (50% switch point)

(d) pseudolru (variable switch point)

(e) quadratic (50% switch point)

(f) quadratic (variable switch point)

(g) texture (50% switch point)

(h) texture (variable switch point)

Figure 6: Mutation scores achieved for programs with composite input parameters (averaged over 20 trials)

Finally, we check that KD–ART and ART are fast enough to be feasible for the new programs. As in the previous experiments, KD–ART (intensify) and KD–ART (diversify) are faster than ART and they run in a similar amount of time. However, KD–ART also requires the mutants to be run. This is not a problem if mutation analysis (or some other coverage metric) is already being used, but it puts KD–ART at a disadvantage if mutants are generated solely for this reason. We should note the times reported here are for 100 test cases, whereas 5000 tests were used previously. However, KD–ART is still considerably faster than ART and it could be argued we should measure how long the techniques require to achieve a certain mutation score rather than to generate a certain number of test cases.

Table 11: Average time taken to run KDT and ART (in seconds)

Program	**Selecting test inputs**			**Running mutants**
	KD–ART (intensify)	KD–ART (diversify)	ART	
chunkybar	0.220	0.217	1.677	46.7
pseudolru	0.169	0.159	1.622	24.8
quadratic	0.304	0.305	1.668	44.3
texture	0.156	0.153	1.517	40.2

8. Threats to Validity

8.1. Internal Validity

We were careful to address any internal threats to validity by repeating our experiments over 20 trials and applying statistical tests to calculate p-values and effect sizes. Since we are comparing three different techniques (ART, KD–ART (intensify) and KD–ART (diversify)), we need to be aware of the multiple comparisons problem: as more comparisons are added, the probability increases that at least one of the differences will be statistically significant (by random chance). We addressed this threat by first applying the Friedman test across the techniques, then making pairwise post-hoc comparisons of each technique using the Bonferroni Correction. We also strengthened the internal validity of our results by presenting median values rather than means (to reduce the effect of outliers) and reported both the mutation scores achieved at the end of the optimisation and the areas under the mutation score curves (to provide results that are independent of the number of test cases).

8.2. Construct Validity

The choice of mutation operators impacts the types of faults represented by our experiments and may affect which strategy (intensification or diversification) is the most effective. If the mutants can be killed by values close to each other in the input domain, intensification should work best, otherwise, diversification may be the better choice. We need to address

the potential threat that our results may only be valid for the operators we used. To do this we selected mutation operators that are widely used by other researchers (see Section 6.3). The mutants these operators produce have been shown to be correlated with real faults [38] and provide a good indicator of the fault detection ability of test suites [25].

We reported the time taken to run the mutants separately from how long our technique took to select the test cases. Our reasoning was that if we already have some information (e.g. by running test cases against mutants), there is no additional cost to make use of it. If this can be accepted, our KD–ART is substantially faster than ART (e.g. 100 times faster for tcas). However, since our technique uses mutant evaluations, it could be argued this cost needs to be taken into account. The validity of our conclusion that KD–ART is faster than ART depends on this, but the validity of our results is unaffected, since we provide both sets of timing information. We envision our technique will be used by testers to utilise information that is available from previous tests - mutation testing was employed in our work to provide an example of using our technique against simulated faults. It would be entirely possible to use our technique against other cheaper metrics, such as branch coverage.

8.3. External Validity

Equivalent mutants pose a threat to our experiments. Since they cannot be killed by any tests, equivalent mutants reduce the highest mutation score that can be achieved [39, 40]. This makes it difficult to compare results between programs that have different numbers of equivalent mutants, but has no effect on comparisons between multiple techniques (e.g. KD–ART and ART). We treat mutants that are not killed by any test data generated during our experiments as equivalent and remove them from our results. This could have an impact on the mutation scores reported in our experiments, but not our comparison of the techniques.

The size of the studied programs may also be a threat. The largest program in our experiments (QPHashMap) has 568 lines of code. Whilst this is larger than the programs typically used in ART research (over 4 times the size of the largest program used by Arcuri and Briand for example [8]), it is still smaller than some of the programs our techniques could be applied to. We carefully chose our programs to address this threat by including both the numeric programs used before in ART research and more complex programs that have composite data type inputs. This has allowed us to show our technique can be used on a wide variety of programs.

The mutants for our experiments with numerical programs were generated and run using cloud computing. For this reason, we need to consider whether there are any scalability issues related to the mutation analysis part of our research - not everyone has access to cloud computing facilities. However, in contrast to this, we observed mutation generation and execution ran faster on a normal desktop than on the cloud. The reason for this is that mutation analysis requires intensive I/O access, which is a known bottleneck for cloud computing. This is why for the subsequent experiments, involving programs which have composite data type inputs, we generated and ran the mutants on a desktop computer.

9. Related Work

There is a long history of research into test data generation for mutation testing [36]. Many of the tools are based on the Reachability-Infection-Propagation model (RIP) [41]. RIP states that killing a mutant requires inputs that: 1) cause execution to reach the location of the mutation in the program code; 2) infect a difference in the state of the program once the mutation has been reached; and finally 3) propagate this difference to an observable output. These conditions are known respectively as reachability, infection and propagation.

Two main approaches have been used to select test data that satisfy the RIP conditions: Dynamic Symbolic Execution (DSE) [42] and Search Based Software Testing (SBST) [43]. DSE executes the program symbolically, but uses concrete values to simplify complex constraints. It has been widely used to achieve structural testing criteria such as branch coverage [44]. By transforming the requirements for infection in the program under test into additional branches to be reached, DSE can also generate weakly killing test data [45].

SBST is a more recent innovation that applies search based optimisation techniques to generate test inputs that satisfy weakly killing constraints. SBST for mutation analysis uses fitness functions to measure how close the current test data are to reaching and infecting a particular mutant. Ayari et al. [46] applied Ant Colony optimisation and Fraser and Zeller [47] applied genetic algorithms to generate test data for killing mutants in Java. More recently DSE and SBST have also been proposed in attempt to kill mutants strongly [48].

Although these white box techniques are effective at generating test data to kill mutants, their computational costs can be high. The techniques rely on program analysis to identify constraints for achieving the RIP conditions. For large and complex programs, there are too many paths to explore and it is difficult to generate all the constraints. In addition, it can be time consuming to solve complex non-linear constraints. This may help explain why DSE and SBST have so far not been widely used in practice. By contrast, the approach introduced in this paper has a low computational cost. As a form of black box test data generation, it does not require any program analysis or constraint solving processes to operate.

There has been much work on improving the efficiency of black box testing through ART. A central aim is to increase the diversity of a test suite, as this allows the first fault to be found more quickly. Discrepancy checks whether different regions of the input domain have the same density of test inputs, whereas dispersion records the maximum distance any point has from its nearest neighbour [19, 20]. Together they can be used to evaluate the extent to which a test suite is diverse. Lattice-based ART (L-ART), starts by placing test inputs in a systematic pattern, so as to maximise the distance between them [49]. This is followed by a series of random adjustments to further increase diversity. Other techniques for diversifying the test suite include evolutionary algorithms [21] and dynamic candidate distribution [20].

More generally, test suite diversification has been used as an objective function for optimisation, to test models [50], database applications [51] and large-scale real world programs [52], using various measures of diversity. In addition, Thomas et al. [53] employed a diversity metric for black-box test-case prioritisation. This later example is relevant to our work, as in our answer to RQ4, we showed that KD–ART (intensify) can be used to select test cases such that they achieve a higher mutation score more quickly.

All of the previous work focuses of diversification rather than intensification and they do not use kernel densities to calculate a weighting for comparing previous test cases. In contrast to previous work, we have shown intensification can sometimes be more effective than diversification. Just because failure causing values tend to be clustered in the input domain, this does not mean each cluster represents a single fault. We need to take a more balanced view, applying intensify and diversify appropriately for each situation.

10. Conclusions

In this paper we have presented a new ART-based technique for killing mutants (KD–ART) and two different strategies for selecting test inputs (intensification and diversification). We have evaluated the technique with programs that have simple (numeric) as well as composite input parameters. On the numeric programs, KD–ART (intensify) achieves an 8.58% higher mutation score than traditional ART and a 7.24% higher mutation score than KD–ART (diversify). The switch point between ART and KD–ART (intensify) has a significant effect on the mutation score achieved and there is a correlation between the optimal switch point and easily calculable properties such as the number of mutants. By contrast, on the programs with composite inputs, the difference between the techniques is much less and the switch point is only statistically significant for KD–ART (diversify). This may be because these programs are object oriented, with smaller individual functions that have mutants which are easier to kill. We conclude that although KD–ART works on programs with composite inputs, it is much more effective at testing numerical programs.

11. Future Work

Although the increases in mutation score achieved by KD–ART compared with ART are generally quite small, they are significant. This suggests it is possible to utilise information about previously successful test cases to improve random testing. Since information is typically available (through branch, mutant or other coverage metrics), we should use it to guide the selection of tests. It is hoped that other researchers will be inspired by our ideas and build upon our work to create techniques that are even more effective. In our future work, we will investigate how to use properties of the program under test to set a suitable switch point between ART and KD–ART. For example, when the switch point was finely tuned for the erfcc program, this resulted in a mean mutation score increase of 0.2, an improvement of 23% compared to ART. We will also look into dynamically switching between ART and KD–ART to achieve maximal efficiency throughout the testing process.

References

[1] E. Parzen, On estimation of a probability density function and mode, The Annals of Mathematical Statistics 33 (3) (1962) 1065–1076.

[2] R. Hamlet, Random testing, in: Encyclopedia of Software Engineering, Wiley, 1994, pp. 970–978.

[3] J. W. Duran, S. Ntafos, An evaluation of random testing, Software Engineering, IEEE Transactions on SE-10 (4) (1984) 438–444. doi:10.1109/TSE.1984.5010257.

[4] A. Arcuri, M. Iqbal, L. Briand, Random testing: Theoretical results and practical implications, Software Engineering, IEEE Transactions on 38 (2) (2012) 258–277. doi:10.1109/TSE.2011.121.
[5] T. Chen, H. Leung, I. Mak, Adaptive random testing, in: M. Maher (Ed.), Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making, Vol. 3321 of Lecture Notes in Computer Science, 2005, pp. 320–329.
[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, T. H. Tse, Adaptive random testing: The art of test case diversity, Journal of Systems Software 83 (1) (2010) 60–66.
[7] C. Schneckenburger, J. Mayer, Towards the determination of typical failure patterns, in: Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting, SOQUA '07, ACM, New York, NY, USA, 2007, pp. 90–93.
[8] A. Arcuri, L. Briand, Adaptive random testing: An illusion of effectiveness?, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, ACM, New York, NY, USA, 2011, pp. 265–275.
[9] H. Liu, F.-C. Kuo, T. Y. Chen, Comparison of adaptive random testing and random testing under various testing and debugging scenarios, Software: Practice and Experience 42 (8) (2012) 1055–1074.
[10] T. Y. Chen, F.-C. Kuo, H. Liu, W. Wong, Code coverage of adaptive random testing, Reliability, IEEE Transactions on 62 (1) (2013) 226–237. doi:10.1109/TR.2013.2240898.
[11] S.-H. Shin, S.-K. Park, K.-H. Choi, K.-H. Jung, Normalized adaptive random test for integration tests, in: Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual, 2010, pp. 335–340. doi:10.1109/COMPSACW.2010.65.
[12] R. Huang, X. Xie, T. Y. Chen, Y. Lu, Adaptive random test case generation for combinatorial testing, in: Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual, 2012, pp. 52–61. doi:10.1109/COMPSAC.2012.15.
[13] R. Huang, J. Chen, Z. Li, R. Wang, Y. Lu, Adaptive random prioritization for interaction test suites, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, New York, NY, USA, 2014, pp. 1058–1063.
[14] T. Y. Chen, R. G. Merkel, An upper bound on software testing effectiveness, ACM Trans. Softw. Eng. Methodology 17 (3) (2008) article no. 16.
[15] M. Patrick, R. Alexander, M. Oriol, J. A. Clark, Subdomain-based test data generation, Journal of Systems and Software.
[16] M. Patrick, Y. Jia, Kernel density adaptive random testing, in: 10th International Workshop on Mutation Analysis, 2015, pp. 1–10.
[17] L. White, E. Cohen, A domain strategy for computer program testing, IEEE Transactions on Software Engineering 6 (3) (1980) 247–257. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.1980.234486.
[18] F. Chan, T. Chen, I. Mak, Y. Yu, Proportional sampling strategy: Guidelines for software testing practitioners, Information and Software Technology 38 (12) (1996) 775–782.
[19] T. Y. Chen, F.-C. Kuo, H. Liu, Distributing test cases more evenly in adaptive random testing, Journal of Systems and Software 81 (12) (2008) 2146 – 2162, best papers from the 2007 Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, April 10-13, 2007 Australian Software Engineering Conference 2007.
[20] T. Y. Chen, F.-C. Kuo, H. Liu, Adaptive random testing based on distribution metrics, Journal of Systems and Software 82 (9) (2009) 1419 – 1433.
[21] A. Tappenden, J. Miller, A novel evolutionary approach for adaptive random testing, Reliability, IEEE Transactions on 58 (4) (2009) 619–633. doi:10.1109/TR.2009.2034288.
[22] F.-C. Kuo, An indepth study of mirror adaptive random testing, in: Quality Software, 2009. QSIC '09. 9th International Conference on, 2009, pp. 51–58. doi:10.1109/QSIC.2009.15.
[23] T. Y. Chen, D. H. Huang, F.-C. Kuo, R. G. Merkel, J. Mayer, Enhanced lattice-based adaptive random testing, in: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, ACM, New York, NY, USA, 2009, pp. 422–429.
[24] C. Chow, T. Y. Chen, T. Tse, The art of divide and conquer: An innovative approach to improving the efficiency of adaptive random testing, in: Quality Software (QSIC), 2013 13th International Conference

on, 2013, pp. 268–275. doi:10.1109/QSIC.2013.19.
[25] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: ICSE, 2005, pp. 402–411.
[26] M. Rosenblatt, Remarks on some nonparametric estimates of a density function, Ann. Mathematical Statistics 27 (3) (1956) 832–837.
[27] K. Chan, T. Chen, D. Towey, Restricted random testing: Adaptive random testing by exclusion, International Journal of Software Engineering and Knowledge 16 (4) (2006) 553–584.
[28] IEEE, Standard for Floating-Point Arithmetic, IEEE 754-2008, Institute of Electrical and Electronics Engineers, New York, NY (2008).
[29] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, Artoo: Adaptive random testing for object-oriented software, in: 30th International Conference on Software Engineering, 2008, pp. 71–80.
[30] V. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Doklady Akademii Nauk SSSR 163 (4) (1965) 845–848.
[31] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes in C. The Art of Scientific Computing., Cambridge University Press, 2002. doi:10.1016/0898-1221(90)90201-T.
[32] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, Empirical Softw. Engg. 10 (4) (2005) 405–435.
[33] J. Nanavati, F. Wu, M. Harman, Y. Jia, J. Krinke, Mutation testing of memory-related operators, in: 10th International Workshop on Mutation Analysis, 2015, pp. 1–10.
[34] and Mark Harman, MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language, in: Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08), IEEE Computer Society, Windsor, UK, 2008, pp. 94–98.
[35] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An Experimental Determination of Sufficient Mutant Operators, ACM Transactions on Software Engineering and Methodology 5 (2) (1996) 99–118.
[36] Y. Jia, M. Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Transactions of Software Engineering 37 (5) (2011) 649–678.
[37] X.-S. Yang, Metaheuristic optimization: Algorithm analysis and open problems, Lecture Notes Comp. Sci. 6630 (2011) 21–32.
[38] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of the Symposium on the Foundations of Software Engineering (FSE), 2014, pp. 654–665.
[39] X. Yao, M. Harman, Y. Jia, A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence, in: Proceedings of the 36th International Conference on Software Engineering (ICSE'2014), Hyderabad, India, 2014, pp. 919–930.
[40] M. Papadakis, Y. Jia, M. Harman, Y. L. Traon, Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple Fast and Effective Equivalent Mutant Detection Technique, in: Proceedings of the 37th International Conference on Software Engineering (ICSE'15), Florence, Italy, to appear.
[41] R. A. DeMillo, A. J. Offutt, Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.
[42] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, ACM, New York, NY, USA, 2005, pp. 213–223.
[43] P. McMinn, Search-based software testing: Past, present and future, in: Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, 2011, pp. 153–163. doi:10.1109/ICSTW.2011.100.
[44] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 263–272.
[45] M. Papadakis, N. Malevris, An Empirical Evaluation of the First and Second Order Mutation Testing Strategies, in: Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10), IEEE Computer Society, Paris, France, 2010, published with *Proceedings of the 3rd International Con-*

ference on Software Testing, Verification, and Validation Workshops.

[46] K. Ayari, S. Bouktif, G. Antoniol, Automatic Mutation Test Input Data Generation via Ant Colony, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), London, England, 2007, pp. 1074–1081.

[47] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, in: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10), ACM, Trento, Italy, ISSTA '10, pp. 147–158.

[48] M. Harman, Y. Jia, W. B. Langdon, Strong higher order mutation-based test data generation, in: SIGSOFT FSE, 2011, pp. 212–222.

[49] J. Mayer, Lattice-based adaptive random testing, in: Proceedings of the 20th International Conference on Automated Software Engineering, ASE '05, ACM, New York, NY, USA, 2005, pp. 333–336.

[50] H. Hemmati, A. Arcuri, L. Briand, Achieving scalable model-based testing through test case diversity, ACM Transactions on Software Engineering and Methodology 22 (1).

[51] E. Rogstad, L. Briand, R. Torkar, Test case selection for black-box regression testing of database applications, Information and Software Technology 55 (10) (2013) 1781–1795.

[52] D. Mondal, H. Hemmati, S. Durocher, Exploring test suite diversification and code coverage in multi-objective test case selection, in: Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation, 2015, pp. 1–10.

[53] S. Thomas, H. Hemmati, A. Hassan, D. Blostein, Static test case prioritization using topic models, Empirical Software Engineering 19 (1) (2012) 182–212.