# Analysis and Enhancements of Adaptive Random Testing

**Article** · January 2005

CITATIONS
21

READS
58

**1 author:**

Robert G. Merkel
Monash University (Australia)
**38** PUBLICATIONS **730** CITATIONS

SEE PROFILE

# Analysis and Enhancements of Adaptive Random Testing

Robert Merkel

June 22, 2005

## Abstract

Random testing is a standard software testing method. It is a popular method for reliability assessment, but its use for *debug testing* has been opposed by some authorities. Random testing does not use any information to guide test case selection, and so, it is argued, testing is less likely to be effective than other methods.

Based on the observation that failures often cluster in contiguous regions, *Adaptive Random Testing* (ART) is a more effective random testing method. While retaining random selection of test cases, selection is guided by the idea that tests should be widely spread throughout the input domain. A simple way to implement this concept, FSCS-ART, involves randomly generating a number of candidates, and choosing the candidate most widely spread from any already-executed test. This method has already shown to be up to 50% more effective than random testing. This thesis examines a number of theoretical and practical issues related to ART.

Firstly, an theoretical examination of the scope of adaptive methods to improve testing effectiveness is conducted. Our results show that the maximum improvement in failure detection effectiveness possible is only 50% - so ART performs close to this limit on many occasions. Secondly, the statistical validity of the previous empirical results is examined. A mathematical analysis of the sampling distribution of the various failure-detection effectiveness methods shows that the measure preferred in previous studies has a slightly unusual distribution known as the *geometric distribution*, and that that it and other measures are likely to show high variance, requiring very large sample sizes for accurate comparisons.

A potential limitation of current ART methods is the relatively high selection overhead. A number of methods to obtain lower overheads are proposed and evaluated, involving a less-strict randomness or wide-spreading criterion. Two methods use dynamic, as-needed partitioning to divide the input domain, spreading test cases throughout the partitions as required. Another involves using a class of numeric sequences called quasi-random sequences. Finally, a more efficient implementation of the existing FSCS-ART method is proposed using the mathematical structure known as the *Voronoi diagram*.

Finally, the use of ART on programs whose input is non-numeric is examined. While existing techniques can be used to generate random non-numeric candidates, a criterion for "wide spread" is required to perform ART effectively. It is proposed to use the notion of *category-partition* as such a criterion.

## Acknowledgements

I would like to thank the exceptional support and guidance of my supervisor, T.Y. Chen, to guide me to the completion of this thesis. His enthusiasm for our work has been a shining light to me.

I would also like to thank my research colleagues in the Software Testing Group, and especially my family.

## Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Signed:

Dated:

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background to software testing

### 1.1.1 Prehistory

Since the invention of the digital computer, ensuring that a program performs according to its specification has absorbed huge amounts of programmer time and effort. Maurice Wilkes[1] [73], a British computer pioneer, stated that:

> As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

During this era, however, it appears that most effort was placed into *debugging*, the location and elimination of programming mistakes once a failure had been revealed. Gill[30], a colleague of Wilkes, describes subroutines for debugging programs on the EDSAC, their digital computer. These subroutines were used to print information about the machine state onto a teleprinter, from which the programmer could make a diagnosis of the error. However, it seems that devising program inputs specifically for testing purposes was not considered necessary. Gill[30] wrote:

> The difficulty lies not in detecting the presence of a mistake, but in diagnosing it. In practice its presence is nearly always obvious, for the character of most programmes is such that even a slight error will usually have an extensive effect. If its presence is not immediately apparent, it will be detected by the arithmetical checks which must be incorporated in every calculation.

Such a view is quite understandable in the context of the unreliable hardware technology of the time, and the applications and use patterns of these early digital computers.

[1] Wilkes confirmed the quote in a personal communication, but could not recall exactly where he had stated it

Bowles[6] (p. 49), an operator of CSIRAC, Australia's first computer, describes elaborate hardware fault detection procedures such as a "valve tester" to check whether individual valves had failed, and the commonplace nature of memory faults. These were apparently as common, or more so, as software failures. According to Bowles, "Occasionally, after a user running his own program had reported a computer error, the test programs[2] would run without any indication of errors ... In many of these occasions it turned out to be the user misunderstanding what the result would be of a particular computer operation."

Early computer installations were primarily used for one-off computation tasks, and most of these were of a scientific nature [48]. Therefore, it seems that most computational results were checked for plausibility before they were used, with the aid of the intermediate results produced as described by Gill [30], and the further reliability of the program beyond the specific task it was created to solve was not of much interest.

### 1.1.2 First testing methods

However, as computer hardware became more reliable and were applied in the military, business, and industry, software reliability requirements grew, and users began to expect programs to be largely debugged before they were deployed. By 1963, Miller and Maloney[51], had begun to realise this this need, and the possibility that errors may remain undetected for some time after deployment: "Too often, however, mistakes show up as late as several months (or even years) after a program has been put into operation." Heard [37] stated in 1962 that testing deserved "as much thought and planning as the program itself".

Around this time, publications on testing tools and methodologies began to appear in the literature. Renfer [60] describes "typical" testing tools for the systems of the time. These tools mainly provided facilities for the automated execution of manually generated test data. Only brief mention of "data preparation" is made. However, others systems were focussed on test case generation. Sauder [61] describes a tool that parses the Data Division of a COBOL program (roughly analogous to the type definitions in a modern language) to determine the format of input data, and lets the tester specify the semantics of the desired tests using standard logical relationships. Sequences of test data could be defined in which the content of certain elements could be varied systematically, or, notably, in a random fashion. Sauder noted that exploring all possible relationships between data elements could lead to an exponentially large number of test cases. Sauder's paper proposed that a human tester could designate certain relations as a priority for testing; he gives no guidance as to how the human tester might choose these. Sauder also speculated that tests might be generated on the basis of the program's specification rather than the source code itself.

The first comprehensive description of a systematic approach to test case selection was by Miller and Maloney [51]. They proposed that testers select test cases such that all

[2]The "test programs" were programs designed to exercise CSIRAC's hardware to detect problems with it

paths in the module under test be executed at least once, and described a simple tree-based notation for doing so. Their notion, in the current context known as *path coverage*, is the basis for many code-based (or *white-box*) testing techniques, both in terms of a method to construct test cases and a basis for comparison of other testing techniques.

A large number of testing techniques were developed throughout the 1960's and 70's (see [53] for a survey), including ideas such as *boundary-value analysis* and *equivalence classes*, both based on program specifications (*black-box* testing), and a considerable number of coverage criteria were derived. Most were *debug testing* methods - the idea of discovering as many unique program errors as possible with the testing resources available.

### 1.1.3 Random testing approaches

Early examination of random testing seem to be mainly (but not exclusively) in the context of software reliability estimation. Girard and Rault [31] proposed two general methods of doing so. Firstly, they proposed the deliberate insertion of errors into the program to be tested, and then the use of random testing until a certain proportion of the inserted errors are detected. The number of other errors detected whilst this process occurs could then be used to estimate the total number of errors present.

Their alternative approach eschewed seeding errors. This model was simply testing using randomly selected inputs. Their paper described some simple statistical models indicating the number of tests required to detect buggy programs with a certain probability. Girard and Rault's analysis did not take into account what they described as "constraints specific to the problem dealt with by the program." Whilst not the primary focus of their paper, Girard and Rault also mentioned the use of "probabilistic testing" in error detection, claiming that it is possible to "obtain rapidly tests that detect a good percentage of the total possible faults."

Thayer *et. al* [66] took a somewhat more sophisticated approach to using random testing for software reliability estimation. Their approach, sometimes termed *operational testing*, requires the tester to determine a profile of the inputs the software under test will receive while in use (an *operational profile*). Once this is done, tests are generated according to the profile, and the number of program failures recorded. From this information, reliability estimations can be made in a straightforward manner.

Hanford [36] demonstrated how to apply random testing in a non-trivial problem domain: programs whose input was specified by a context-free grammar (in Backus-Naur form), though no information was provided on its effectiveness in actual testing.

In the wider testing community, there seemed to be some doubt as to whether random testing was useful at all, let. alone as a debug testing method. Myers[53], in his seminal textbook on software testing, looked unfavourably on random debug testing:

> Probably the poorest method of all is random-input testing – the process of testing a program by selecting, at random, some subset of all possible input value. In terms of the probability of detecting the most errors, a randomly

> selection collection of test cases has little chance of being an optimal, or close to optimal, subset...

However, not all researchers were so uncomplimentary. Duran and Wiorkowski [27], in the context of analysing software reliability, showed that, contrary to intuition, estimates of software reliability could sometimes be calculated more accurately by using random sampling than could be achieved by path-coverage style testing. On this basis, Duran and Ntafos[26] conducted an empirical examination of the fault-finding capabilities of random testing. They collected several example (small) programs with errors from the literature and recording how often a random test found the error. They found that for many of the errors they examined, randomly-selected test data would detect it a reasonably high proportion of the time. The one case they describe where random testing was likely to fail was an on a boundary case, which they argue was unlikely to significantly impact the program's operational reliability.

Random testing, as a debug testing method, was compared with *partition testing* by Duran and Ntafos [25]. Partition testing describes any testing scheme that divides the input into more than one disjoint subset (partition), and allocates test cases amongst them. Partition testing schemes are based around the assumption that failure behaviour within a partition is reasonably homogeneous - in other words, either most inputs within a partition will cause a failure, or most will not. Path coverage schemes can be viewed as a partition testing scheme, with the subset of inputs causing a particular path to be executed forming a partition. In this case, the assumption of relative homogeneous failure behaviour within a partition seems intuitively justified. Therefore, allocating a few tests to each partition would be a more effective test method than random testing.

Duran and Ntafos conducted simulations to determine whether this was the case. Their experiment simulated a faulty program, dividing the program's input domain into 25 partitions, each of which was randomly assigned a failure rate. In one experiment, the failure rate was allocated randomly using the bimodal distribution assumed above, in others it was allocated using a uniform distribution, with the maximum failure rate varied in different experiments. They supported the use of uniform failure rate distributions by citing Howden [38]. Howden examined the notion of failure rates of a program when executed on a particular path. Rather than the commonly assumed behaviour that a program would either always succeed or always fail when executed on a path, Howden's experiments indicated that many faults resulted in failures when executed on the relevant path only a proportion of the tim. Therefore, Duran and Ntafos concluded that a partition scheme based on paths would result in many paths having moderate failure rates rather than the exclusively very high or low rates assumed previously.

Even though this simulation was under conditions that may have been less favourable to partition testing, it was still more effective than random testing - in that the probability of detecting at least one failure with test case sets of the same size was higher with partition testing than with random testing. However, if double the number of tests were performed using random testing than with partition testing, random testing was more effective. They argued that given the relative simplicity of selecting tests using random testing that in

many cases it would be more cost effective to test using a greater number of randomly-selected test cases rather than fewer selected using a partition testing technique.

Weyuker and Jeng [71] considered under what conditions the probability of detecting at least one failure using a specified number of tests generated by partition testing ($P_p$) is better than that of the same number of tests generated by random testing with uniform sampling from the entire input domain($P_r$). They found that when all subdomains were of the same size, and equal numbers of test cases were allocated to each subdomain, that $P_p \geq P_r$. They also argued that even in the best case, $P_p$ was not much greater than $P_r$. However, their observation did show that relatively simple strategies could be use to improve on the failure-detection capability of random testing. Chen and Yu [18] generalised this condition to include any partition testing scheme where test cases were allocated in direct proportion to the size of each partition. Based on this, they advocated what they termed the Optimally Refined Proportional Sampling Strategy - that is, to perform $n$ tests divide the input domain into $n$ equally-sized partitions, and then place one test into each partition.

### 1.1.4 Random number generation

To perform random testing, one requires a source of random numbers. The generation of random number sequences, or more precisely "pseudorandom" number sequences that have many of the same properties as truly random sequences, has been of great interest to cryptographers and physicists since the very earliest days of computing. The development of Monte Carlo techniques [68] for numerical multidimensional integration focussed considerable attention on to the problem of generating such sequences.

Many cryptographic techniques are, at their core, methods of generating a pseudorandom number sequence, using a short "key" to set up the internal state of the generator. For cryptographic purposes, if it is computationally easy to determine the internal state of the generator (and thus its future output) by inspecting the recent output, any scrambled message will be compromised. Therefore, cryptographically secure random number generators are designed specifically to avoid this. Speed and memory usage of the generator is generally a secondary concern.

For other purposes, however, this is not an issue. For simulation, speed, memory usage, the ability to generate very long sequences without repetition, even distribution, and independence between closely-spaced sequences of numbers (so that, when short sequences are used as vectors, patterns do not appear). Early attempts at fast random number generation date from the earliest history of computing, with methods such as the middle square method, proposed by John von Neumann in 1946 ([39] pp. 3-5)), and methods based on the Fibonnaci numbers. These early approaches were demonstrated to be unsatisfactory, as their period was too short and the distribution of their numbers nonuniform.

A more effective approach was introduced by D.H. Lehmer in 1948 ([39], p. 9), the linear congruential random number generator. In this approach, four parameters are chosen: $X_0 \geq 0$, the starting value, $a \geq 0$, the multiplier, $c \geq 0$, the increment, and $m, m > a, m > c$, the modulus. The sequence is then determined using the following

relation:

$$X_{n+1} = (aX_n + c) \bmod m \qquad n \geq 0 \tag{1.1}$$

With some care in the selection of appropriate values for the parameters, linear congruential methods generated sequences that were used widely and proved satisfactory for many purposes for many years. A large number of tests for random number generators to check that the sequences they generated behaved similarly to truly random numbers were also devised - though some random number generators in widely-used system libraries failed these tests [58].

Marsaglia [46] pointed out a major problem with the linear congruential generator that is particularly relevant to testing applications: if groups of $n$ numbers taken, in sequence, from a linear congruential generator are treated as points in an $n$-dimensional space, the points will lie in a limited number of $n-1$ dimensional hyperplanes, with the number of hyperplanes decreasing as $n$ increases. This difference might lead to odd results if "random" testing was conducted using such a generator and the pattern of hyperplanes was similar to the pattern of failure-causing inputs; for instance, consider a situation where a "strip" failure pattern, as shown in Figure 1.1 (b) happens to be parallel to one of the hyperplanes, but does not contain it. In this case, the failure detection effectiveness would be much lower than that expected from truly randomly selected test cases.

For some time afterwards linear congruential generators remained the standard type deployed in system libraries, despite the development of a number of improved generators [43]. In practice. generators have been tried in a variety of applications, from simulations to Monte Carlo integration, in which most worked reasonably well most of the time.

With the increasing speed of computers, the consequent use of much larger random number sequences where this type of issue becomes more serious, and wider recognition of the problems caused by linear congruential generators, interest has risen in recent times. One of the most popular of the new generation of random number generators (RNGs) is the Mersenne Twister algorithm [47]. It has passed a wide variety of statistical tests, has been theoretically proven to have the enormous period of non-repetition of $2^{19937}$, and be equidistributed in up to 623 dimensions. It is also as fast as most linear congruential generators.

Virtually no work appears to have been done in the software testing community on the suitability of existing pseudorandom number generators for the purpose, though previous empirical studies of random testing have shown results consistent with that expected from truly random numbers and thus have shown no indication that the pseudorandom generators are *un*suitable.

### 1.1.5 Failure Patterns

Chan *et. al* [10] observed that certain common types of programming mistakes were likely to give raise to failures occurring in regular "patterns" throughout the input domain. They describe three particular types of patterns, as shown in figure 1.1. The first pattern they describe is the "block" pattern, where a single, contiguous, compact region of the program's

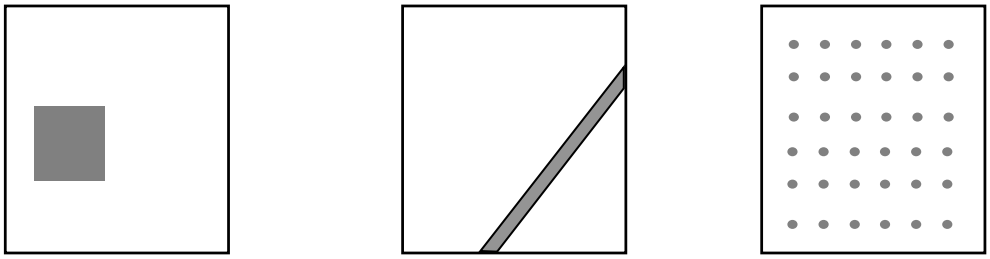



Figure 1.1: Typical program failure patterns for a two-dimensional input domain (a) "block" pattern, (b) "strip" pattern, (c) "point" pattern.

input space fails. The second, the "strip" failure pattern, involves a thin, elongated strip. Finally, the "point" pattern pattern, where a program fault causes failures in one or a number of very small regions, with each region possibly representing only a single test case. They noted that these points were sometimes spread in a regular pattern throughout the input domain. Chan *et. al* claimed that strip and block failure patterns were likely to be more common than point patterns.

Bishop [5], in an empirical study of two programs used as control code in a nuclear reactor system, found that the overwhelming majority of faults detected caused contiguous failure patterns with "sharp" edges - in the terms of Chan *et. al*, "strip" or "block" failures. He presented a model of the programs as a collection of sequential functions, of which one was faulty in some subset of its input domain. From this basis, he argued that if these functions were smooth and continuous, the failure regions were likely to be continuous. He furthermore argued that some of the edges were likely to be aligned with "contours of equal output value for the function upstream of the error". Ammann and Knight [2] found similar patterns in a small-scale study of a hypothetical missile control program.

## 1.2 Adaptive Random Testing

On the basis that contiguous failure regions are common, two test cases that are close to each other are more likely to have the same failure behaviour as two test cases that are widely spaced. Therefore, given a choice between a point that is close to other points already tested and that have not detected a failure, and a point further away, the point that is further away is more likely to reveal a fault. This observation leads directly to the notion of *Adaptive Random Testing*, in which Chen *et. al* [16] use this information to guide test case selection.

Mak [44] presented several algorithms that implemented the basic ART concept. The simplest, was termed the *Fixed Size Candidate Set*. The algorithm used is shown in detail in Figure 1.2. In essence, to generate another test case, a small number of *candidates* (they state that initial tests indicated 10 maximised performance) are randomly selected

```
do
    set M = 0
    generate next k candidates c_1, ..., c_k
    for each candidate c_n
        calculate minimum distance m from existing test cases
        If m > M
            set M = m
            b = c_n
    Add b to list of existing test cases
    Test program using input vector b
while (no failure found)
```

Figure 1.2: The FSCS ART algorithm

with all inputs within the input domain equally likely to be chosen. For each candidate, the distance to the closest (most similar) existing test case is calculated. The test cases for which this distance is greatest is selected, and the program is tested using this candidate. Tests are selected and performed until a fault is found, or sufficient tests have been performed.

Obviously, this raises the question of defining "distance" in a sensible manner. This is a very difficult problem to solve in the general case. However, for the case where the input to a program takes the form of a fixed-length vector of integer or real numbers, Mak used the Euclidean distance measure:

$$\text{dist}(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2} \tag{1.2}$$

Mak [44] examined whether FSCS-ART was more effective at detecting program failures than random testing. He obtained a sample of 12 small programs from the numerical methods textbook *Numerical Recipes* [59]. He then artificially "seeded" the programs with errors, such as changing conditions in loop predicates, keeping an unaltered copy for comparison of results. He could therefore detect "faults" by comparing the output of the two versions of the program and reporting a fault when the outputs differed.

Mak introduced a new effectivenesss metric for the comparison. The F-measure was defined as the expected number of test cases required to detect the first failure. He argues that this measure is more intuitively appealing, and a better match for testing practice when discovery of a bug causes the testing to be suspended whilst the fault is located. From a more pragmatic viewpoint, the F-measure also avoids the issue of how testing should continue after a fault is located, an issue not discussed in their paper. He found very substantial reductions in the F-measures for most of the 12 programs they examined. For nine of the 12, the F-measure was reduced by over 40%. At worst, the F-measures were similar to that of random testing.

As he noted, however, the selection overhead for these test cases is substantial. If $k$ candidates are generated, then the cost of candidate generation is increased by a factor of

$k$ over random testing. However, the total overhead is dominated by the cost of selecting the best candidate. In their implementation, for each candidate, it is required to calculate its distance from every existing test case, so for the $n$th test case this has cost proportional to $n$. Generating $i$ test cases using the method therefore takes $\mathrm{O}(ki^2)$ time.

Mak examined two other variations of the ART methodology, the *Universal Candidate Set* approach, and the *Growing Candidate Set*. The UCS method was an experiment, used to argue that that the success of FSCS was not entirely due to the greater number of candidate test cases considered. It was not intended to be a practical test case selection method. In the UCS method, a large, fixed set of candidate test cases was generated, of sufficient size so that the expected number of failure-causing inputs in that set was one. For the purpose of the experiment, the set was then checked to ensure that at least one failure-causing input was present. To generate the sequence of test cases, they initially allocated this entire initial set to be the candidate set. To choose the next test case, they searched the entire candidate set for the best case $T$ according to the minimax criterion. They then tested with $T$. If no failure was detected, it was added to the set of executed test cases, and removed from the candidate set. They compared the effectiveness of this method with random testing, but in this case the source of the random testing was the same candidate set as was used for UCS. Despite the fact that the exact same random test cases were used, UCS-ART had F-measures lower than that of RT. Improvements were generally smaller than with FSCS-ART.

They proposed one final variant, the *Growing Candidate Set* ART method. In this method, the "unused" candidates from a round of test case selections are not discarded. Instead, they are kept and evaluated, along with a small number of new candidates, for subsequent test cases. They found the effectiveness of this method to be a little inferior to FSCS-ART.

An alternative method for improving the effectiveness of random testing by spreading the test cases more evenly was proposed by Chan, *et. al* [11]. *Restricted Random Testing* achieves this by creating circular (or, in the N-dimensional case, hyperspherical) *exclusion zones* around previously executed test cases. When a new test case is selected, a candidate is randomly generated. If the candidate lies within an exclusion zone, it is discarded and another candidate generated, the process repeated until a candidate laying outside an exclusion zone is chosen. The radius of the exclusion zones is chosen so as to keep the sum of the areas of the exclusion zones constant, ignoring the effects of overlap and domain boundaries. Restricted Random Testing (R-ART), as implemented, has the same $\mathrm{O}(i^2)$ complexity for selecting $i$ test cases, though the constant factors are smaller.

They compared the effectiveness of R-ART with FSCS-ART and RT using seven of the 12 programs used in the previous evaluation of ART. They determined that increasing the size of the exclusion zone (up to the point where the exclusion zones covered too much of the input domain to make selecting test cases feasible) improved the performance, and, if the largest exclusion zone possible was used, the effectiveness of R-ART was similar to (or sometimes slightly greater) than ART.

## 1.3 Open Issues

There are several issues of both theoretical and practical concern relating to ART. These range from practical concerns that would affect the use of present ART techniques, to more abstract questions about the theoretical justification of the approach.

An obvious practical concern is the high test case selection overhead of ART compared to RT, as the implementations presented so far have required $O(n^2)$ time to select a sequence of $n$ test cases. If a large number of tests are to be performed, and the overhead of test execution and evaluation is low, this extra overhead would mean that that the per-test failure-detection effectiveness advantages of ART would be outweighed by the extra overhead. Therefore, methods that have similar effectiveness to ART but with reduced overheads would significantly broaden its applicability.

More fundamentally, while empirical justification for ART exists, there is no theoretical analysis of its performance. It would be of great interest to prove that ART, given a block failure pattern, performs better than random testing, for example. Arguably even more interesting is how close ART is to an "optimal" method - is there much room for improvements in effectiveness?

On a practical level, the range of programs that ART has been tested on has been very limited. For many programs, simply generating random test data is very challenging. However, there are quite a range of programs, such as SQL engines [63] and even complex graphical user interfaces, where this has been done successfully. It is not currently possible to apply ART to these programs, though. To perform ART requires a heuristic to measure how well-separated pairs of program inputs are; as yet, no such heuristics have been devised. Until such heuristics are devised, ART is of quite limited application.

## 1.4 Outline

It will undoubtedly come as a great shock to the reader that this thesis outlines methods to help overcome some of the problems with ART listed in the previous section![3].

Chapter 2 examines the question of whether ART is an "optimal" test case selection strategy, or how close it is to one. Rather than attempting a theoretical analysis of the effectiveness of ART, which has proven exceptionally difficult, much stronger assumptions are made about the distribution of failures through the input domain, and upon these assumptions an optimal testing strategy is devised. Despite these, much stronger assumptions, the optimal strategy's performance is only a little better than that of existing ART strategies under the same conditions. Therefore, we argue that that future work on ART strategies should be concentrated on lowering overheads and broadening the method's applicability, rather than vainly trying to seek further improvements in effectiveness.

Chapter 3 examines two key statistical properties of ART. The first property is how the test cases selected for ART are spatially distributed throughout the input domain - are they uniformly distributed, as is the case with uniform random testing? The second is the sampling distribution of the F-measure, which is analysed theoretically for random

[3] Apologies for the little joke

testing (for the first time), and empirically for ART. I show that the sampling distribution for RT follows a standard statistical distribution called the *geometric distribution*, and that the distribution for ART is similar, but with a much steeper "dropoff" for high F-measures, and examine the implications of this for previous studies and future analysis. As a consequence of these results, chapter 4 presents a small simulation study examining the potential for future gains in testing performance with the aid of more information about failure patterns, showing that, particularly for strip patterns, useful but not enormous improvements may be possible.

We then turn our attention to efficiency, with a number of different approaches to reducing the selection overhead of ART. There are several different ways this can be done. Existing ART approaches offer very explicit guarantees of randomness and even spreading of test cases. By relaxing one or both of these, selection overhead can be lowered.

In chapter 5, two methods that use a more relaxed standard for "even spread" are tried, using dynamic partitioning to spread test cases widely throughout the input space without explicitly checking for separateness. Experiments show that these methods have both theoretically and practically lower overheads, and are reasonably effective (though not as much so as earlier ART methods).

Instead of relaxing the even spread, in chapter 6, the use of low-discrepancy sequences for testing is proposed. Low-discrepancy sequences have but not all of the properties of random numbers - notably, they are explicitly spread evenly over the input domain and don't "clump" together. The use of such sequences, and "scrambled" quasi-random sequences, is explored. They are shown to be low-overhead and quite effective - in many cases almost as effective as random testing. As well as the considerable potential for testing uses, these experiments pose interesting questions about the discrepancy of the number sequences generated by ART and potential applications for this.

In chapter 7, a more efficient method of implementing FSCS-ART is proposed, through the dynamic construction of a data structure known as the *Voronoi diagram* and the associated *Delaunay Triangulation*. An experimental study confirms that a Voronoi diagram-based implementation of FSCS-ART can conduct a sequence of $n$ test cases in $O(n\sqrt{n})$ time, rather than the $O(n^2)$. The benefits and drawbacks of this improved implementation are discussed.

Finally, chapter 8 examines the other piece of the puzzle - how to apply ART to a broader range of programs. A number of syntactic approaches, based on the structure of the input, are briefly discussed. A more promising semantic method, based on the category-partition method, is then proposed. This flexible method,while preliminary, nevertheless demonstrates a promising way in which ART can be extended to a far wider range of problems than in the past.

In Appendix A, I examine whether existing pseudorandom number generation algorithms are "random enough" for the purposes of using them for ART. My study suggested that most existing pseudorandom number generators are perfectly adequate for random testing experimentation and practice, so there was relatively little to report from these experiments.

# Chapter 2

# An upper limit on testing effectiveness

## 2.1 Introduction

In the past, ART's performance has been analysed almost entirely through experiments. No mathematical analysis has been conducted as to when, and how much by, its performance exceeds that of random testing. Nor is there any indication of how close it is to an "optimal" testing strategy, and under what circumstances it is close to optimal. It may be possible to prove that, given certain assumptions about a program's failure patterns and using a specific effectiveness metric, a particular strategy is "optimal". That is, it might be possible to show that a particular test case selection strategy is at least as effective as any other proposed strategy. However, theoretical analysis of ART appears to be extremely difficult, and in any case making assumptions about failure patterns usually allows the construction of special-purpose test case selection schemes which will outperform the general method of which one is trying to prove optimality.

We have followed an alternative approach. Rather than trying to examine the optimality of a specific method, we allow the tester additional information about the failure behaviour of the program, and then calculate the effectiveness of a provably optimal strategy given this additional information. Any possible testing scheme with the same or less information about the failure patterns of the program will be at best as effective as the optimal scheme. We can also then compare the experimentally-measured effectiveness of ART schemes against our bounding value.

This chapter presents a definition and proof of an upper bound of test case effectiveness, and examine in detail its assumptions. We then examine some of the implications of our result.

## 2.2 Definitions and assumptions

First, assume that the input domain of the program to be tested can be completely represented as a convex polytope in $\Re^n$. Requiring a particular input to be integer does not affect the result.

For some of the succeeding discussion, we present results assuming that the input domain is a polytope in $\Re^2$. In this case generalising the result to $n$ dimensions is very straightforward.

A test case is therefore represented as a point in $\Re^n$ inside the input domain. The result of running a test case is either that the performance of the program is as specified (no failure detected), or it differs from the specification (a failure).

A *failure region* is a contiguous region of the input domain where any test case within that region will result in a failure.

Assume a failure region's size, shape, and orientation are completely known. If we then specify a point on the boundary of that failure region, which we term the *anchor point*, specifying the location of the anchor point unambiguously locates the failure region within the input domain.

If we assume the presence of a single failure region with known size, shape and orientation and a defined anchor point, we define the *anchor domain* as that section of the input domain in which the anchor point could be located. Note that the boundaries of the anchor domain will change if a different anchor point is chosen, but its size and shape will remain unchanged.

We define the *failure domain* as that section of the input domain which could be part of the failure region. Note that the failure domain contains the anchor domain.

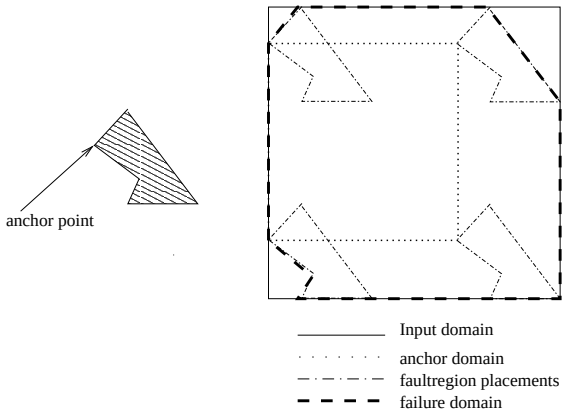



Figure 2.1: A hypothetical failure region and corresponding anchor and failure domains

Figure 2.1 shows a hypothetical failure region with a specified anchor point, and the corresponding anchor and failure domains, found by hypothesising that the failure region is at the extremities of the input domain (the "ghosted" outlines of the failure region). As can be seen, the anchor region is, in this case, significantly smaller than the input domain, whilst the failure domain is almost as large as the input domain.

Again assuming the failure region as above, we define the *exclusion region* of a test case $t$ as the area in which the anchor point of the failure region could be and still cause a failure when the program is tested with $t$.

### 2.2.1 Effectiveness metrics

There are several different ways in which the failure-detection effectiveness of testing methods can be measured.

Historically common metrics used to compare testing effectiveness have been the P-measure and E-measure. The P-measure is defined as the probability that at least one program failure is detected with a specified sequence of tests. The E-measure is defined as the expected *number* of failures detected by a sequence of tests. They have been used to compare *partition testing* to *random testing* analytically [18] [35] [65] [71]. Other relationships between these metrics have also been determined, one of the more notable [19] being that when the failure rate and $n$ are sufficiently small, the P-measure and E-measure closely approximate each other[1].

More recently, Chen *et al.* [17] have proposed a new effectiveness measure, the F-measure, using it to compare Adaptive Random Testing (ART) to random testing. The F-measure is defined as the number of test cases required to detect the first failure. They argue that this measure is more informative, and natural, than the earlier measures. This metric has been adopted for subsequent studies of related work, notably Chan *et al.*[11], which examined Restricted Random Testing, another ART method using the principle of exclusion to achieve an even spread of test cases, and Chen *et al.* [14]. It is the primary measure used throughout this thesis.

If the program under test's input domain is of size $D$, and of those, $d$ inputs fail to produce the correct output, the failure rate $\theta = \frac{d}{D}$. For random testing as described previously, if $n$ tests are conducted the E-measure is $n\theta$, and the P-measure is $1-(1-\theta)^n$. For the E-measure, it is trivial to use the Central Limit Theorem to show that the sampling distribution will be normal. For the F-measure, Chan *et al.* [11] have stated that the expected F-measure for random testing is $\frac{1}{\theta}$.

## 2.3 Theorem

**Lemma 1** *Assume the existence of a single, continuous, failure region $f$, of unknown location but with a known shape, size, orientation, and specified anchor point $p_f$ in input domain $I$. For any test case $t$, the exclusion region $E_t$ has an area smaller than or equal to the area of $f$.*

Lemma 1 is straightforward. As an illustration, consider the alternative situation where the fault region's location is fixed and the test case is mobile. In that case, it is clear that the test case can be moved to any position within the failure region to still trigger the failure. The "smaller than or equal to" caveat is required because of the possibility that a test case might be so placed close to the edge of the input domain and thus the exclusion region is not entirely located within it.

[1] [19] defines "sufficiently small" more precisely

**Lemma 2** *Assuming $f$ as above and the anchor domain $A$, but no knowledge about its location, the probability of detecting a failure with a single test case is smaller than or equal to $\frac{|f|}{|A|}$.*

The anchor point has an equal probability of being in any location of the anchor domain, and from Lemma 1 it is known that a single test case will detect failures if the anchor point is located in an area equal to that or smaller than $f$. Therefore, the probability of the anchor point being in the exclusion region of the test case is equal to the ratio of the area of the exclusion region and the anchor domain.

**Lemma 3** *The probability of detecting a failure with $n$ distinct test cases is smaller than or equal to $\frac{n|f|}{|A|}$. If we ignore the possibility of edge effects with individual test cases and assume the exclusion regions of the test cases do not overlap, it will be equal to $\frac{n|f|}{|A|}$.*

Lemma 3 is a direct consequence of lemma 2.

We now examine, given the above, the probability density function for the F-measure. $P(F = n)$ is the probability that the $n$th test case detects failure. In a single sequence of test cases, for $F = n$ to be true, the anchor point of the failure region must fall within the exclusion region of the $n$th test case, and not within the exclusion regions of any of the others. We can then state and state the following:

**Lemma 4** *$P(F = n) \leq \frac{|f|}{|A|}$. If there is no overlap between the exclusion zones of any test cases, $P(F = n) = \frac{|f|}{|A|}$, for $n \leq \frac{|A|}{|f|}$, and 0 for $n > \frac{|A|}{|f|}$ (because at that point exclusion regions will cover the entire anchor domain and the failure must already have been detected).*

**Lemma 5** *If there is no overlap between the exclusion regions, the expected F-measure will be*

$$\sum_{i=1}^{\frac{|A|}{|f|}} i \cdot \frac{|f|}{|A|} = \frac{|A|}{2|f|} + \frac{1}{2} \tag{2.1}$$

The question then arises as to what happens if the exclusion regions overlap. If they do, the effect on the sum calculating the F-measure will be to reduce the probability fraction of the existing terms and cause the probability of F-measures greater than $\frac{|A|}{|F|}$ to be non-zero, thus adding additional terms to the sum. In any case, the value of the sum must be increased. Therefore:

**Lemma 6** *Any overlap between exclusion regions will increase the expected F-measure. Therefore, the expected F-measure will be smaller than or equal to*

$$\frac{|A|}{2|f|} + \frac{1}{2} \approx \frac{|A|}{2|f|} \tag{2.2}$$

If the size of the failure region, compared to that of the input domain, is "small" in all dimensions, then the anchor domain will closely approximate the input domain. If we make this assumption, we can compare the performance of our optimal scheme with that of random testing:

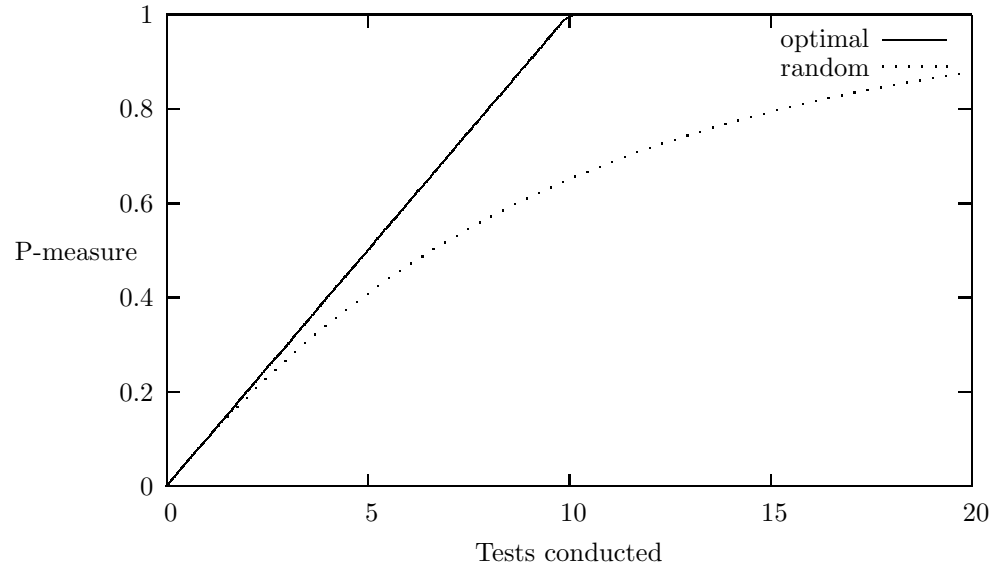



Figure 2.2: P-measure for optimal testing and random testing where $\theta = 0.1$

**Theorem 1** *Assuming that the area of the input domain $I \approx |A|$, the smallest possible expected F-measure is approximately $\frac{I}{2|f|}$. This is half the expected F-measure from random testing, $\frac{I}{|f|}$.*

In other words, given precise information about the shape and size of a failure region, but no information about its location, the best test case selection scheme we can devise will reduce the expected F-measure by no more than 50% compared to that of random testing.

## 2.4 Other Effectiveness Metrics

### 2.4.1 P-measure

Using the approach above, a similar analysis can be performed for the P-measure, an alternative effectiveness metric. Using the rationale described above, the P-measure, when conducting $n$ tests with a failure rate of $\theta$ using optimal testing is:

$$P_0 \leq \min(n\theta, 1) \tag{2.3}$$

The P-measure for random testing with replacement in similar circumstances is:

$$P_r = 1 - (1 - \theta)^n \tag{2.4}$$

Figure 2.2 shows that if only a few tests are performed, the relative effectiveness of random testing and optimal testing are quite similar. The difference in effectiveness between optimal and random increases as more testing is performed, up to the point where optimal testing is guaranteed to detect an error (when $n = 1/\theta$). The expected P-measure from random testing at this point is

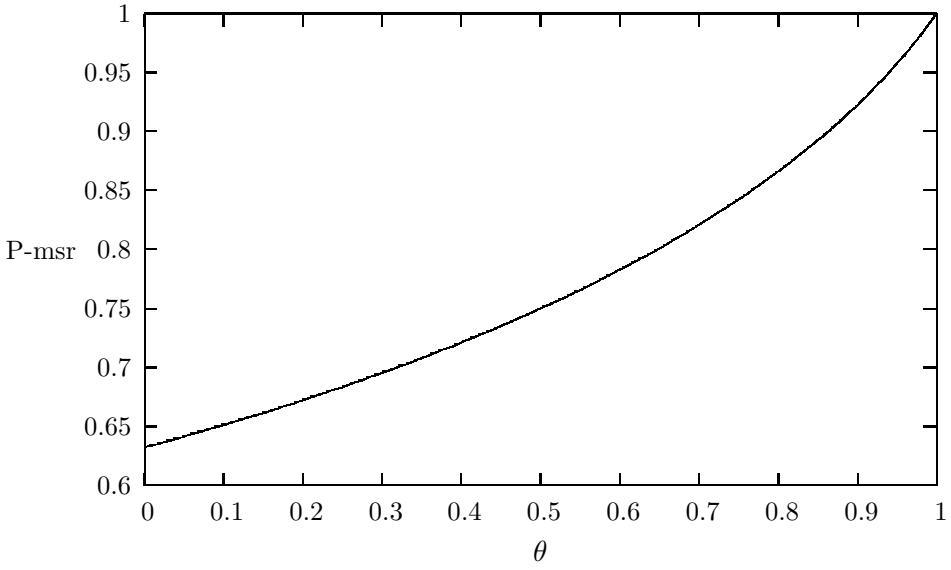



Figure 2.3: P-measure for random testing where P-measure for optimal testing is 1

$$P_r = 1 - (1 - \theta)^{\frac{1}{\theta}} \tag{2.5}$$

A plot of this equation for values of $\theta$ from 0 to 1 is shown in figure 2.3. As can be seen, the difference between optimal and random increases as $\theta$ becomes smaller. In the limit, the difference as $\theta \to 0$ is $1 - \frac{e-1}{e}$, or about 0.37.

### 2.4.2 E-measure

If an optimal strategy for maximising the P-measure is used, the E-measure is equal to $n\theta$. This is the same as the E-measure for random testing! This seemingly counterintuitive result is explained by the fact that with the optimal (for P-measure) strategy, at most one failure will be detected, whereas with the random strategy it is possible for multiple failures to be detected.

The trivially optimal strategy for maximising the E-measure is to select test cases optimally for maximising the P-measure until a failure is detected, and once that failure is detected, perform the same test case again repeatedly until the desired number of tests are conducted. However, the optimal E-measure of this strategy serves as a demonstration of the inadequacy of the E-measure as an effectiveness measure for adaptive test case selection methods. The implied rationale of the E-measure is that a greater number of failures detected is likely to mean that that a greater number of distinct program faults will be detected. The "optimal" strategy shows that an adaptive method designed to maximise failure detection may well do so in a manner that violates this assumption.

## 2.5 Applicability

The key assumption that was made to derive the theoretical results for the P-measure and F-measures in the previous sections was that the anchor region would be approximately

the same size as the input domain, which is true when the single failure region is small in all dimensions compared to the input domain. Whilst this is undoubtedly true for many faults in real-world programs, there are some common and obvious types of faults where it is not. What happens in these cases? In this section we examine this, and whether our bounding result - that the expected F-measure cannot be reduced by more than 50% - still holds in these circumstances.

### 2.5.1 Exceptional failure patterns

Given an input domain, it is easy to construct a failure pattern that is guaranteed to be detectable with a single test case. Such patterns are designed such that a bounding rectangle containing the entire pattern is of the same dimensions as the input domain, though the area of the pattern itself is small Figure 2.4 (a) shows one such pattern. Clearly, the area of the failure region is much smaller than the are of the input domain (and by constructing the example appropriately can be made arbitrarily small), and so for random testing the expected F-measure would be arbitrarily greater than 2. A similar argument applies for Figure 2.4 (b), which is perhaps a slightly more likely example.

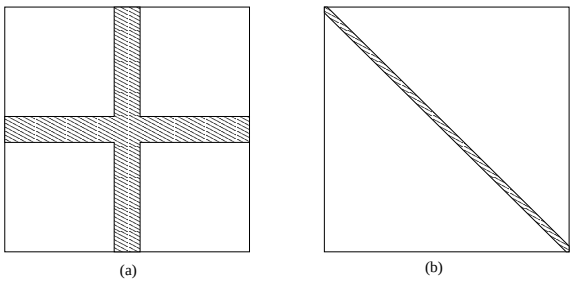



Figure 2.4: Hypothetical failure regions detectable in a single case

The above cases are relatively uninteresting, as the failure domain is equal in area to that of the failure pattern - most of the input domain has no chance of being part of a failure region, in which case there is no point in testing there. If random testing was performed, but selecting test cases only from within the failure region, the expected F-measure would be 1.

### 2.5.2 Strip failures

More relevant are cases involving strip failure patterns (see Section 3.4.2 for a description of strip failure patterns). These violate the assumption that the anchor domain is similar to the input domain, but (depending on the orientation of the strip), most or all of the input domain may be included in the failure domain.

Many common strip failure patterns do give rise to effectiveness bounds equivalent to Theorem 1. Figure 2.5 (a) shows a common such case - a "vertical" strip where the strip is as long as the input domain is in that dimension. As our failure pattern has area $f = lw$, and our input domain is of size $A = l^2$ the expected F-measure of random testing is

$$A/f = l^2/lw = l/w \tag{2.6}$$

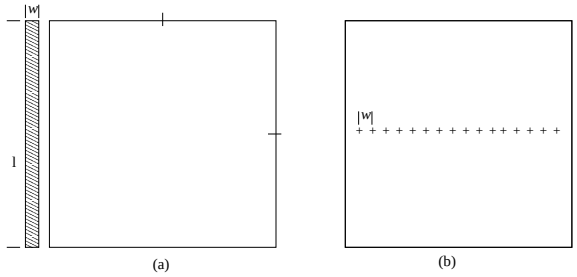



Figure 2.5: Strip failure patterns (a) indicating the shape of the failure region, and b) showing an optimal set of test cases

Given the above failure pattern, it should be clear that the optimal set of test cases involves placing test cases along the "short" dimension at distance $w$ apart, as shown in Figure 2.5 (b). Assuming the failure region is equally likely to be located anywhere possible, the $P(F = n) = w/l$ for $n \leq l/w$, and 0 for $n > l/w$. Given that, the expected F-measure is $\frac{l}{2w} + \frac{1}{2} \approx l/2w$ - in other words, half that of the expected F-measure for random testing.

However, it is possible to construct strip-like failure patterns in which the entire input domain is part of the failure domain and an optimal test case selection strategy can be devised which does better than the usual lower bound. Figure 2.6 a) gives a failure region for which this is the case - where the "vertical" extent of the strip is $l/2 + \epsilon, \epsilon \ll l$. The expected F-measure of random testing is $\frac{l^2}{(\frac{l}{2}+\epsilon)w} \approx \frac{2l}{w}$.

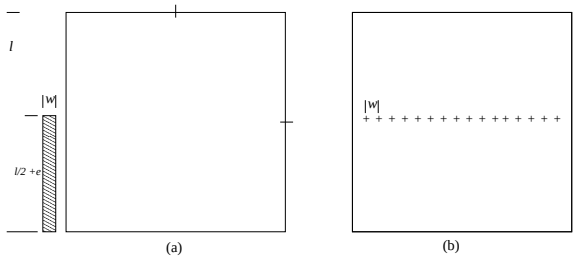



Figure 2.6: Strip failure patterns (a) indicating the shape of the failure region, and b) showing an optimal set of test cases

An optimal pattern of test cases to detect this failure pattern is shown in Figure 2.6 b) - in fact, it is the exact same pattern as demonstrated in Figure 2.5 (b). No matter where the failure pattern is placed vertically, the centrally located test cases will detect the fault over the same horizontal range as previously, and the expected F-measure using this sequence of test cases would be approximately $l/2w$, that is, *1/4 of that expected from random testing.*

However, none of the above should detract that in all situations where the dimensions of the failure pattern are small compared to the input space, and many others where this is not the case, the bounding limit holds. Many of the cases that exceed the limit are probably unlikely to occur in real programs.

### 2.5.3 Failure pattern shapes and optimality

In this section, we examine under what conditions test case selection strategies can be devised that are as effective as the limit demonstrated in Theorem 1 allows. It is assumed that the dimensions of the failure patterns are small relative to the input domain and thus the exceptions discussed in section 2.5 are not applicable.

The amount of overlap of the anchor regions of the test cases used determines how closely the expected F-measure approaches the limit. If there is no overlap at all until complete anchor region coverage is achieved (and thus a failure must have been detected) the effectiveness will be equal to the limit. As the shape of the anchor region is exactly the same as the failure region, the limit effectiveness is achieved if and only if the failure region can be "tiled" so as to cover the entire input domain completely with no overlap.

In two dimensions, there are two polygons that allow such coverage - two of the three *regular tessellations* [70], as shown in Figure 2.7. The other regular tessellation involves the use of equilateral triangles, some of which are inverted relative to others, and so does not meet our stricter requirements.

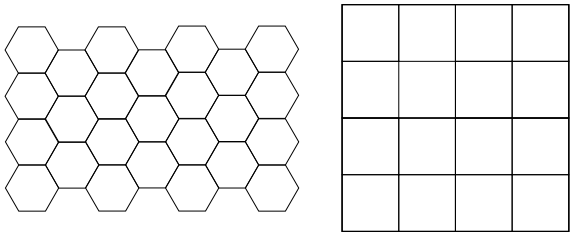



Figure 2.7: regular tessellations showing failure patterns that could be detected with good effectiveness

The effectiveness will be unaffected (except for border effects) by any failure pattern that can be transformed into one of the two patterns by a combination of rotation, skewing, and resizing in one dimension. Irregularly-shaped, non-convex failure patterns will have higher overlap than failure regions not exhibiting these properties, and thus the most effective sequences of test cases possible will be relatively less effective than for failure patterns not exhibiting these properties.

## 2.6 Conclusion

We have shown that for failure patterns that are small on all axes of the input domain, that with stronger assumptions about the failure behaviour than is often able to be made (specifically, that only a single failure pattern is present, and its size and shape are known), no testing strategy will have an expected F-measure less than half the expected F-measure of a random testing strategy.

Previous empirical studies of ART have shown failure-detection effectiveness improvements approaching our bounding value in some (but not all) of the faulty programs analysed [16]. This suggests that it will be somewhat difficult to improve on the effectiveness

of an ART strategy for the favourable case of a block failure pattern, without more information about the likely failure behaviour of the program (for instance, specific knowledge of areas which are likely to have homogeneous failure behaviour).

Given this, future research into testing strategies like ART would probably not be fruitfully spent trying to significantly increase its effectiveness on block-like failure patterns. More fruitful areas to explore may include lower-overhead approaches that have similar effectiveness to ART, and methods that take advantage of particular pieces of information available to the tester.

While it is not difficult to generate optimal sets of test cases if a rectangular failure pattern is assumed, we do not yet know much about optimal cases for other, more irregular failure pattern shapes. We know of no algorithm to determine the optimal test pattern for an arbitrary shape or, indeed, any class of shape other than discussed in the previous. There is no answer even to the simpler question of how much improvement is possible over random testing given an arbitrary failure region shape. While not directly of use for testing practitioners, some answers to the above questions should provide additional insight into the improvements possible over present testing techniques. Chapter 4 examines this further.

As discussed previously, analytical results about ART methods have been hard to come by. As well as the significant result in this paper, this general approach may provide the basis for further mathematical analysis of the failure-finding abilities of ART.

# Chapter 3

# Statistical Properties of Adaptive Random Testing

In this chapter, we examine two statistical properties of Adaptive Random Testing. Firstly, we examine the *spatial* distribution of tests throughout the testing domain, at different points in the test sequence. Previous studies have assumed that points generated by ART are distributed evenly throughout the input domain. In this chapter, actual data is collected to examine whether this is true.

Secondly, we examine the the sampling distribution of random testing and ART - what is the probability density function of the F, P, and E-measures? This is important for practitioners who wish to gain some insight into how varying the results of testing might be, but even more important for experimenters comparing test case selection methods on the basis of effectiveness. We show that the F-measure has a quite unusual sampling distribution, and that all three measures show much higher levels of statistical variability than might be expected.

## 3.1 Distribution of test cases throughout the input domain

In random testing where a uniform probability distribution is desired, all that is required is that the pseudo random number generator produces uniformly distributed output. This is a basic requirement for most random number generators, and one widely-used modern generator, the Mersenne Twister [47] has been proved to be equidistributed in up to 623 dimensions. This requirement is therefore trivially satisfied. Adaptive Random Testing is not proposed as a reliability assessment method and therefore requirements for a specified test distribution are not required for that purpose. Pragmatically, however, the desire to ensure that failure regions, wherever they are located in the input domain, are detected promptly makes it important to know whether tests generated by the various ART methods are located approximately uniformly.

For a new method presented later in this thesis, Bisection-ART, (see section 5.4 for more details) it is easy to show under certain easily satisfiable conditions that test cases will be uniformly distributed throughout the input domain. In the case of FSCS-ART, by contrast, a truly uniform distribution is unlikely. Kuo and Towey[42] suggested that

FSCS-ART generation is likely to be biased towards the edges of the input domain, for the simple reason that tests on the edges do not have neighbours outside the domain boundaries. This tenancy should be self-correcting, for if initial tests cluster at the edge, for subsequent tests the distances to the nearest already-executed test case for candidates at the edge will also decrease and thus ensure that they are not selected. However, how pronounced this effect is, and for how many test cases it persists, can only be determined by experiment.

### 3.1.1 Empirical Investigation of FSCS-ART spatial distribution

A simple simulation was conducted to examine the distribution of test cases throughout the input domain for FSCS-ART.

In this simulation, sequences of points were generated in a two-dimensional unit square input domain. The position of the $n$th-occurring test case, where $n = \{5, 10, 15, 20, 25, 30, 40, 50, 60, 80, 100\}$ were recorded separately. 10,000,000 such runs were performed, leaving us with a total of 11 "buckets" of 10,000,000 points each. For each "bucket", the domain was divided into a 128x128 grid and the number of points occurring in each grid area was calculated.

Figures 3.1 through 3.11 show the spatial distributions for the various test cases. Lighter areas indicate a higher probability of selection.

Figure 3.1 shows a somewhat skewed early distribution, with areas nearest the corners most likely to be chosen for testing, and the regions a little closer to the centre least likely, and with the centre reasonably like to get selected. The skewing is quite strong, with the less-likely regions having approximately 1/10th the probability of being selected than the most-likely regions in the corners. However, this dramatic skewing is quite short-lived. Figure 3.2 shows by the 10th test in the sequence, the pattern has become more complex, but less dramatic, with only a small region in from the corners showing a dramatic skew and, a number of regions spaced regularly throughout the input domain with moderately higher selection probability. This pattern appears more faintly for the 15th test (Figure 3.3), but by the 20th test (Figure 3.4) this becomes indistinguishable. For this and subsequent tests, the major deviation from uniformity is located at the edge of the input domain, which has a significantly higher probability of selection, and a strip immediately interior to the edge, which has a lower probability of selection. Inside this the majority of the input domain has near-equal probability of selection. These strips get narrower for later test cases.

This simple study shows that the spatial distribution of test cases generated by FSCS-ART are nonuniform. In a sequence of tests generated by FSCS-ART, early test cases show a particularly skewed distribution, whereas later test cases show less. The major trend in the skew is a higher probability of test cases being selected very close to the edges of the input domain, and a lower probability in a strip adjacent to those edge regions. Later test cases have a much more uniform distribution.

This nonuniform distribution of test cases is a potential concern; if the failure pattern is present entirely in a region that remains less likely to be selected, it is possible that FSCS-

ART would perform worse than uniform random testing in detecting that failure pattern - an undesirable possibility, as other studies have suggested that the worst-case performance of FSCS-ART is approximately equivalent to random testing. It must be remembered, however, that after the first few tests are performed, the probability of selection is near-uniform over most of the input domain, and that the regions of nonuniform selection shrink as more tests are generated. If the failure rate is large, failures will be detected quickly in any case, and any relative lack of performance is less of a concern.

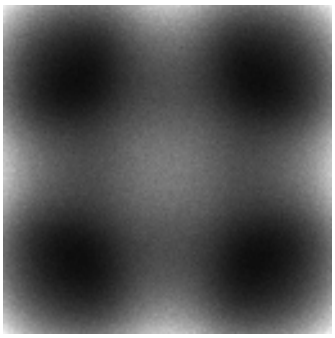


Figure 3.1: Spatial distribution of 5th generated points in FSCS-ART Sequences

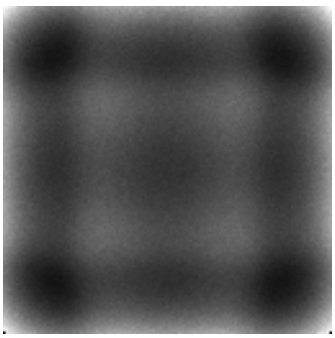


Figure 3.2: Spatial distribution of 10th generated points in FSCS-ART Sequences

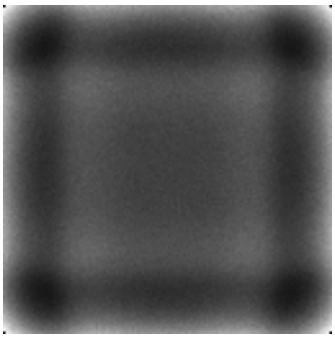


Figure 3.3: Spatial distribution of 15th generated points in FSCS-ART Sequences

### 3.1.2 Spatial Distribution of R-ART test sequences

The study in section 3.1.1 showed that FSCS-ART test sequences had nonuniform spatial distribution characteristics. Intuitively, similar arguments suggest that R-ART may have

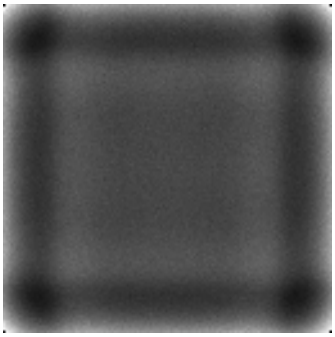


Figure 3.4: Spatial distribution of 20th generated points in FSCS-ART Sequences

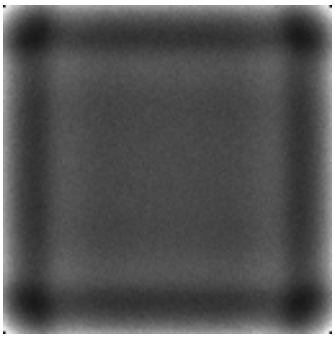


Figure 3.5: Spatial distribution of 25th generated points in FSCS-ART Sequences

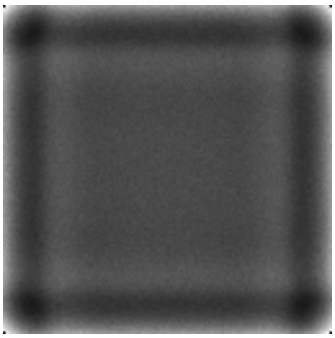


Figure 3.6: Spatial distribution of 30th generated points in FSCS-ART Sequences

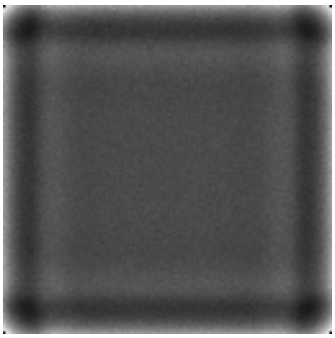


Figure 3.7: Spatial distribution of 40th generated points in FSCS-ART Sequences

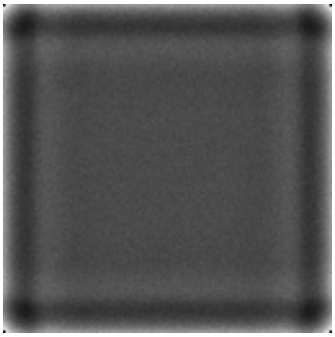


Figure 3.8: Spatial distribution of 50th generated points in FSCS-ART Sequences

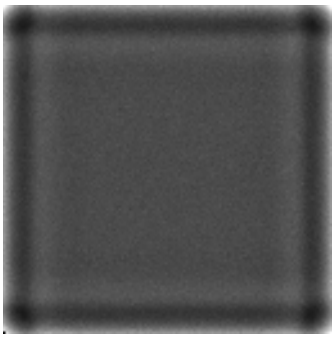


Figure 3.9: Spatial distribution of 60th generated points in FSCS-ART Sequences

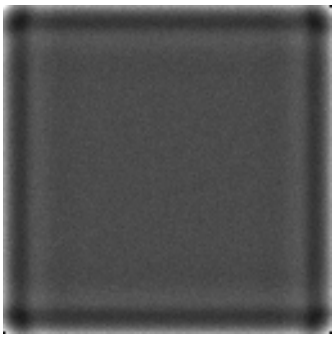


Figure 3.10: Spatial distribution of 80th generated points in FSCS-ART Sequences

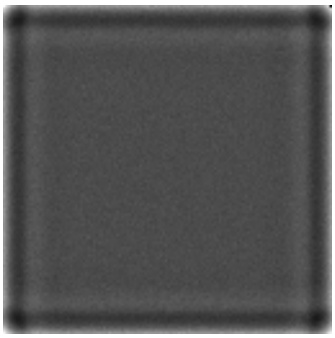


Figure 3.11: Spatial distribution of 100th generated points in FSCS-ART Sequences

the similar characteristics. It was therefore natural to conduct a similar simulation for it.

The simulation was conducted as described in section 3.1.1, except that tests were generated using R-ART, with an R parameter of 1.3. The patterns appear to be extremely similar to the those from FSCS-ART in section 3.1.1.

Overall, it seems that both FSCS-ART and R-ART generate test sequences where early tests, in particular, are unequally distributed throughout the input domain. There seems to be little difference in this regard between R-ART and FSCS-ART. This is undesirable, as it raises the possibility that if the failure region of a program under test is in a region which is less likely to be selected for testing the performance of these methods may be inferior to random testing.

There are several factors that combine to mitigate the potential for this to occur. It is a generally accepted tenet in software testing that errors often occur on boundary cases, including input domain boundaries. By happy coincidence, the areas with the highest probability of being tested are those boundary cases. Secondly, as more tests are performed, the distribution of test cases across the input domain becomes much more uniform. The effectiveness of a test case generation method is most significant when the failure rate is low and a high number of tests need to be performed; on these cases the possible effects of the non-uniformity are minimised. Even in the most uneven cases, "holes" do not appear in the coverage, even the least likely regions have a reasonable, if somewhat smaller than elsewhere, chance of being selected.

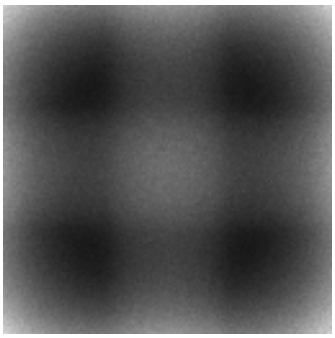


Figure 3.12: Spatial distribution of 5th generated points in R-ART Sequences

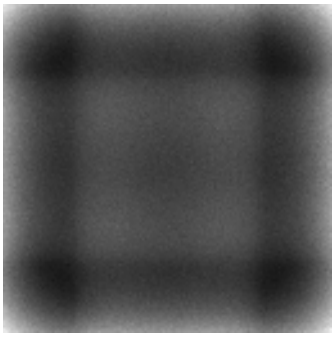


Figure 3.13: Spatial distribution of 10th generated points in R-ART Sequences

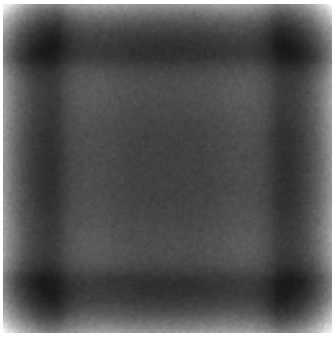


Figure 3.14: Spatial distribution of 15th generated points in R-ART Sequences

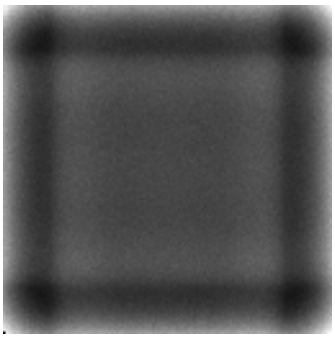


Figure 3.15: Spatial distribution of 20th generated points in R-ART Sequences

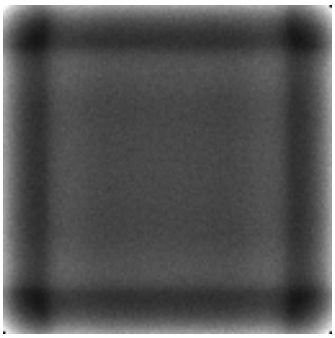


Figure 3.16: Spatial distribution of 25th generated points in R-ART Sequences

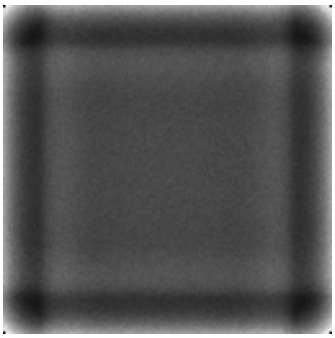


Figure 3.17: Spatial distribution of 30th generated points in R-ART Sequences

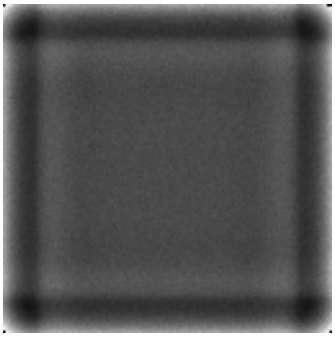


Figure 3.18: Spatial distribution of 40th generated points in R-ART Sequences

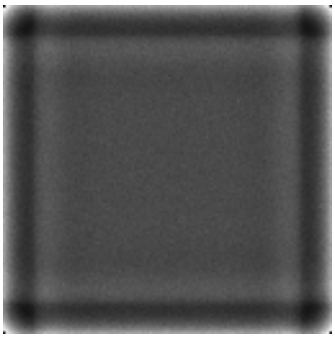


Figure 3.19: Spatial distribution of 50th generated points in R-ART Sequences

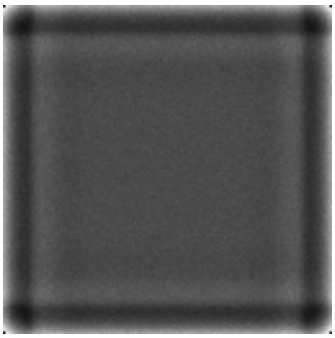


Figure 3.20: Spatial distribution of 60th generated points in R-ART Sequences

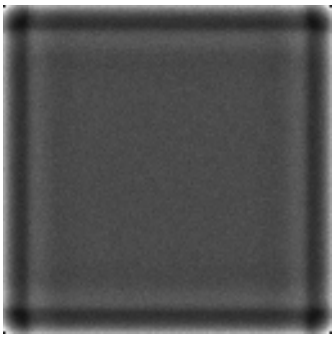


Figure 3.21: Spatial distribution of 80th generated points in R-ART Sequences

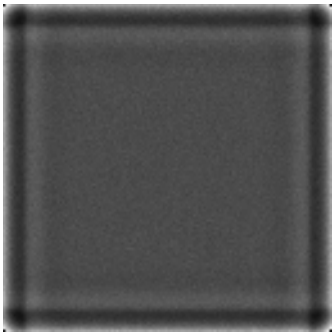


Figure 3.22: Spatial distribution of 100th generated points in R-ART Sequences

## 3.2 Sampling distribution of effectiveness measures

We now turn our attention to another statistical property of debug testing that has received scant attention - the statistical properties of the effectiveness measures used to compare different testing methods.

As mentioned in section 2.2.1, according to Mak [44] the expected mean F-measure for random testing with failure rate $\theta$ was $1/\theta$. No indication of the expected sampling distribution was provided. In the absence of other evidence, a normal sampling distribution is typically assumed. However, the high sample standard deviations reported suggested that further investigation was warranted.

In this section, we examine the sampling distribution of the F-measure both from an analytic and experimental perspective. Section 3.3 examines analytically the statistical properties of the F-measure for random testing, showing that the sampling distribution of the F-measure for random testing follows a well-known distribution, the *geometric distribution*. In this context random testing describes a strategy where test cases are randomly selected from across the entire input domain, with a uniform probability of selection. In section 3.4 we conduct a simulation study to examine whether the same sampling distributions are present when testing using FSCS-ART and R-ART.

In sections 3.5 and 3.6 we examine the sampling distributions of the P-measure and E-measure. They are normally distributed, but our analysis shows that they have very high statistical variability.

Finally, in section 3.7 we discuss the implications of our results with regards to previous related studies using the F-measure, and recommendations for future comparison studies.

## 3.3 Sample distribution of the F-measure in random testing

If we test a program with failure rate $\theta$ using a randomly selected test case, the probability of detecting a failure is precisely $\theta$. Given this, if we test a program with a set of randomly selected test cases, the probability that the first test case in this set will detect a failure is also $\theta$. In other word

$$\mathrm{P}(F = 1) = \theta \tag{3.1}$$

| $\theta$ | Mean | Median |
|---|---|---|
| 0.1 | 10 | 7 |
| 0.05 | 20 | 14 |
| 0.01 | 100 | 69 |
| 0.005 | 200 | 139 |
| 0.001 | 1000 | 693 |

Table 3.1: Median and mean F-measure for random testing

If failure is detected with the first test case, for the purposes of calculating the F-measure the test sequence is complete. Therefore, the chances of a second test case being selected is $(1-\theta)$. If that second test case, selected with replacement, (that is, the previously executed test case is not excluded from the pool of possible selections) is executed, the chances of it detecting a failure is still $\theta$. Therefore,

$$\mathrm{P}(F=2) = (1-\theta)\,\theta \tag{3.2}$$

.

In general, the probability of the $n$th trial detecting failure is the probability of preceding trials not detecting failure, multiplied by the probability of this specific trial detecting failure:

$$\mathrm{P}(F=n) = (1-\theta)^{n-1}\,\theta \tag{3.3}$$

This probability distribution turns out to be an instance of the *geometric distribution* ([52], p. 176). Mosteller *et. al* ([52] p. 189), describe a proof that the expected number of trials before first failure (denoted by $F$ in our specific case), is $\frac{1}{\theta}$, and also ([52] p. 219), outlines a proof that the variance of $F$ is $\frac{1-\theta}{\theta^2}$. If $\theta$ is small, this simplifies to approximately $\frac{1}{\theta^2}$, and the standard deviation is approximately $\frac{1}{\theta}$.

The median F-measure can be approximated using the solution to the following summation:

$$\sum_{n=1}^{x} (1-\theta)^{n-1}\,\theta = \frac{1}{2} \tag{3.4}$$

In general, there is no exact integral solution for $x$, however, the sampling median will be the smallest whole number greater than the analytical solution to the summation:

$$\text{median} = \left\lceil \frac{\ln \frac{1}{2}}{\ln (1-\theta)} \right\rceil \tag{3.5}$$

Table 3.1 shows that in most cases the ratio $\frac{\text{median}}{\text{mean}}$ is approximately 0.7. As $\theta$ approaches 0, the ratio $\frac{\text{median}}{\text{mean}}$ decreases, but the effect is marginal over typical values of $\theta$. In the limit as $\theta \to 0$, the ratio $\frac{\text{median}}{\text{mean}}$ approaches $ln2 \approx 0.693$. Consequently, for typical values of $\theta$, well over 50% of test runs will require fewer than the mean number of test cases to detect the first failure.

## 3.4 Simulation study of distribution in random and adaptive random testing

### 3.4.1 Introduction

While the previous section shows analytically the distribution of the F-measure for random testing, the analysis is not applicable to adaptive random testing as the probability of detecting failure is not constant, but should increase from trial to trial. However, it was not clear whether different versions of ART would have markedly different distributions, or whether other factors such as the distribution of inputs that fail, or the failure rate would affect the overall shape of the distributions. We therefore conducted [13] a simulation study to examine the sampling distribution of the F-measure under these conditions. In our simulations, three different parameters were varied: the test case selection strategy, the failure pattern, and the failure rate.

### 3.4.2 Method

For our simulation, rather than using actual program failures we simulated program failures using patterns representing the three types of failure patterns - block, strip and point - shown in figure 1.1. For our simulations, block patterns were simulated by square-shaped regions of the input domain. Strip patterns were simulated by randomly choosing a point on a vertical boundary of the input domain, and a point on a horizontal boundary of the input domain, and plotting the boundaries of a strip centred on them of width chosen so as to ensure the failure rate is as specified. Point patterns were created by randomly placing 10 circular regions without overlapping, whose total size was chosen so as to have the appropriate failure rate, in the input domain.

To conduct a simulation run, the failure pattern was chosen according to the particular experimental condition. Test cases were then generated using the chosen test case selection method until a "failure" was detected (the test case was within the simulated failure region), and the number of test cases required for the detection was recorded. For each experimental condition, 2000 simulation runs were performed.

The test case selection strategies compared were random testing, with replacement, and with a uniform probability of all points in the input domain to be selected; Fixed Size Candidate Set-Adaptive Random Testing (FSCS-ART), using 10 candidates per test; and Restricted Random Testing [11] (R-ART), using an exclusion ratio of 150%. Failure rates used in our trials were 0.02, 0.01, 0.002, and 0.001. With 4 failure rates, 3 different types of failure patterns, and 3 testing strategies, there were 36 experimental conditions in total.

### 3.4.3 Results

As our simulations generated a considerable amount of data, we discuss here only results for the failure rate 0.002. Varying the failure rate had little qualitative effect on the shape of the distribution except the obvious scaling factors.

Figure 3.23 shows a histogram (smoothed by placing the data in buckets) for the three

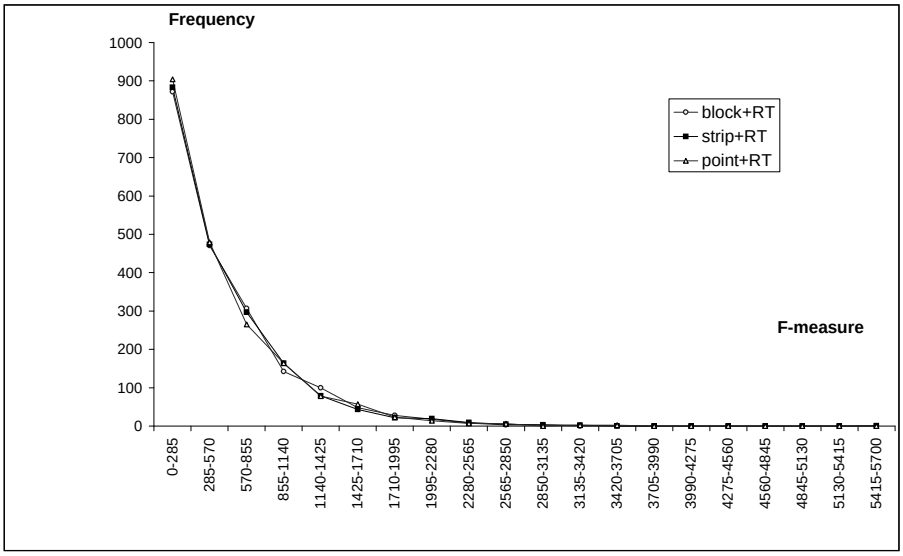



Figure 3.23: Histogram of block, strip and point patterns when performing random testing with failure rate 0.002

different types of failure patterns using random testing. As can be seen, the resulting distributions are identical and a good match for the geometric distribution predicted by our analysis in the previous section.

For the ART methods, Figures 3.27 and 3.28 show that the distributions for the point and strip patterns were close to those for RT. It is anticipated that if the failure rates were higher, there would be a more significant difference between the distributions of ART and RT for strip patterns. Our experimental data for $\theta = 0.01$ and 0.02 are consistent with this expectation. For the block pattern (Figure 3.26), however, the distribution is quite different, with a much steeper drop-off in frequency for high F-measures. Furthermore, at small F-measures, the frequency for ART is significantly higher than that for RT.

The same data, grouped by testing method, clearly shows the effect of the failure pattern on the F-measure distribution for ART methods. Figures 3.24 and 3.25 show the steep drop-off for block patterns, the close-to-geometric distribution for point patterns, and the strip patterns much closer to the point pattern in this case.

### 3.4.4 Quantile-quantile analysis

A followup analysis was conducted, using quantile-quantile plots, to show more clearly that the F-measure distribution for block patterns for ART and RRT more closely resembled the geometric distribution than a normal distribution.

A quantile-quantile plot is a graphical technique "for determining if two data sets come from populations with a common distribution" [54]. It simply plots the quantiles of the first data set against the quantiles of the second data set. To determine whether

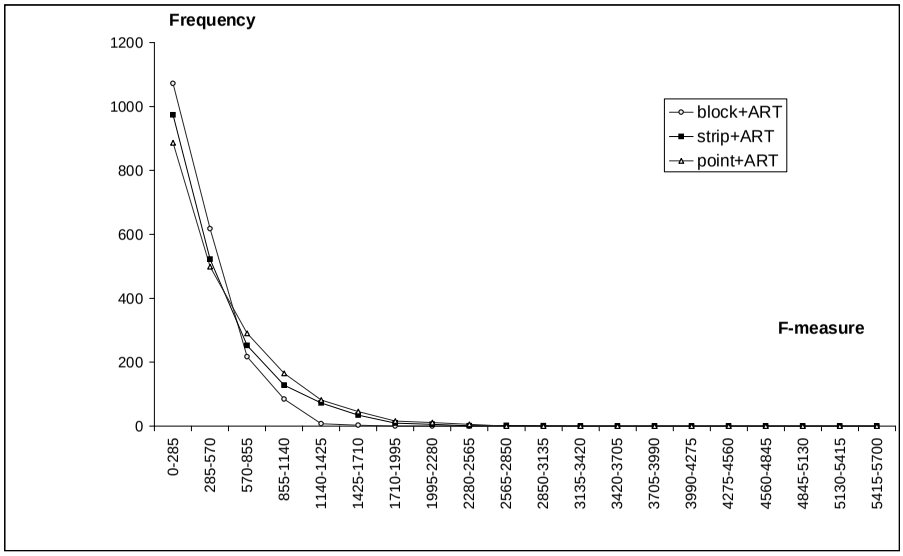



Figure 3.24: Histogram of block, strip and point patterns when performing FSCS-ART with failure rate 0.002

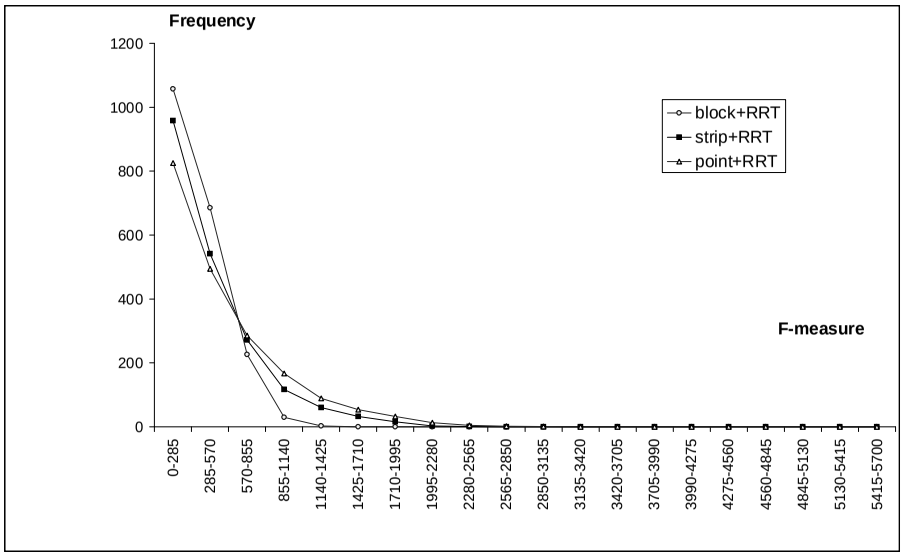



Figure 3.25: Histogram of block, strip and point patterns when performing restricted random testing with failure rate 0.002

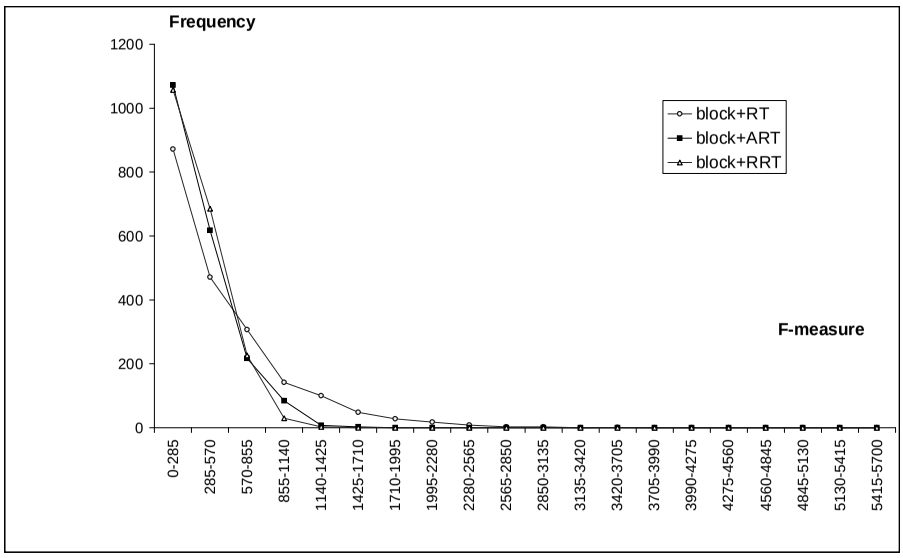



Figure 3.26: Histogram for block patterns with failure rate 0.002

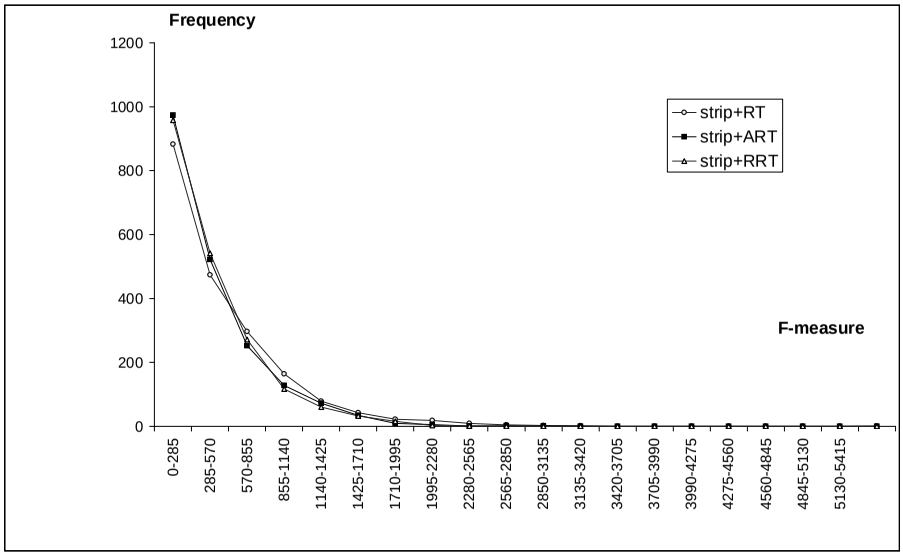



Figure 3.27: Histogram for strip patterns with failure rate 0.002

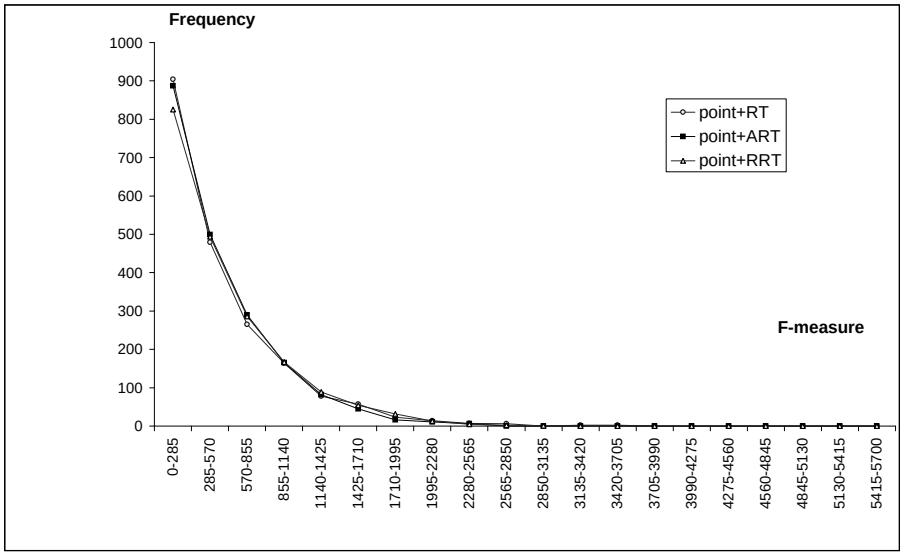



Figure 3.28: Histogram for point pattern with failure rate 0.002

an experimentally-generated data set matches a standard distribution, the experimenter simply generates a synthetic dataset of the required distribution. If the two distributions are the same, the plotted quantiles will form a straight line. The bigger the deviation from a straight line, the less similar the distributions are.

For this analysis, data collected in an identical manner to the previous study was compared to synthetically-generated normally-distributed and geometrically-distributed data. Block failure patterns were used, with a failure rate of 0.01. Data was collected for ART and RRT.

Figure 3.29 compares the F-measure distribution of ART with block patterns with a known geometrically-distributed dataset. It is clear that the quantiles do not form a straight line, and thus the ART F-measure distribution is not geometric. However, the ART data is much more similar to the geometric dataset than the normally-distributed dataset to which it is compared in Figure 3.30. The pattern for RRT is very similar, as shown in Figures 3.31 and 3.32.

## 3.5 The P-measure

With the issues regarding the F-measure's sampling distribution, it may seem tempting to consider using an alternative effectiveness metric such as the P-measure. However, accurate estimation of the P-measure is not as straightforward as it might seem.

First, we consider the experimental design for estimating the P-measure for a arbitrary test case selection strategy. Assume a single program under test, and that the selection strategy is not deterministic (so that multiple, different test sets can be obtained). The number of test cases in a test set to be performed using the strategy must then be decided;

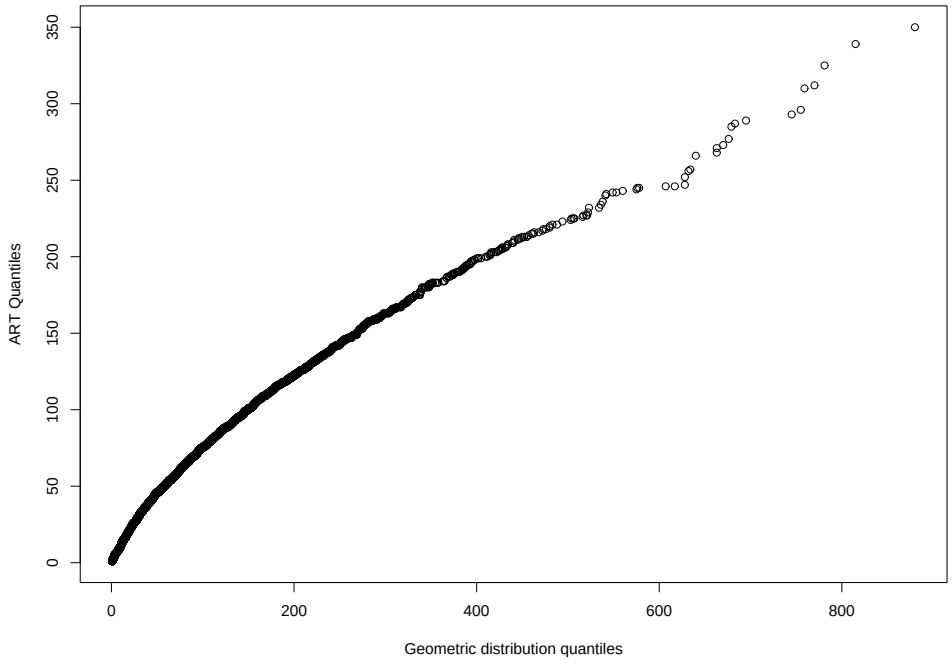



Figure 3.29: Quantile-quantile plot of ART F-measure distribution and known geometrically-distributed data

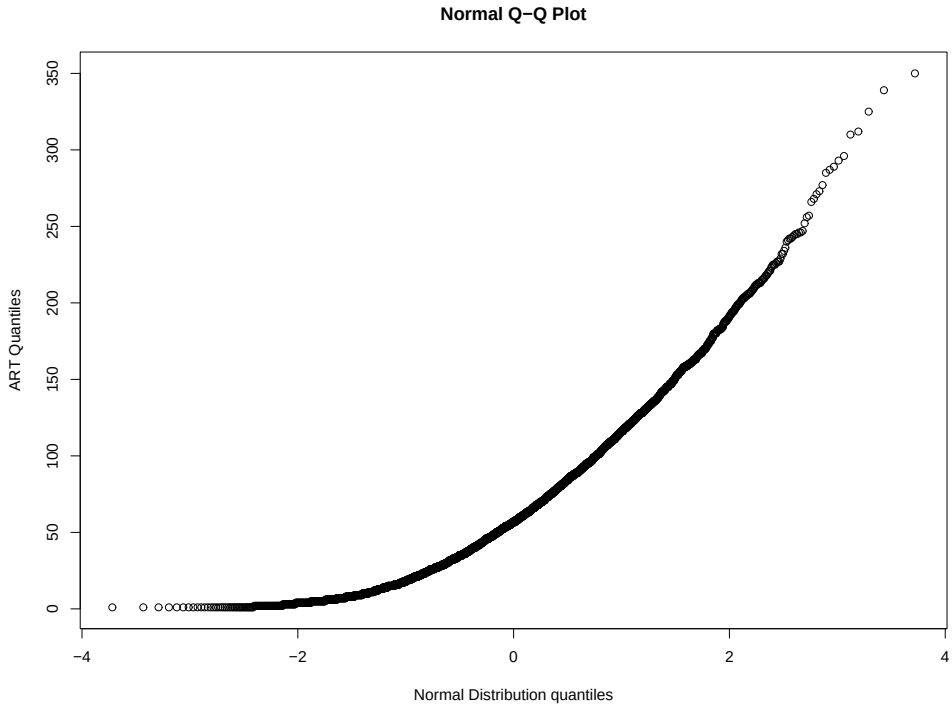



Figure 3.30: Quantile-quantile plot of ART F-measure distribution and known normally-distributed data

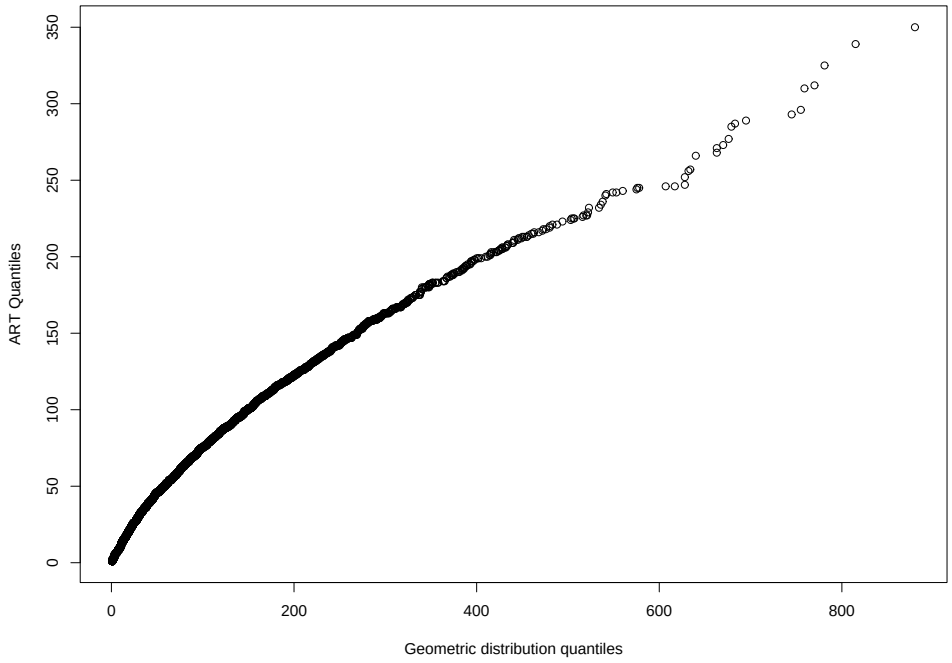



Figure 3.31: Quantile-quantile plot of RRT F-measure distribution and known geometrically-distributed data

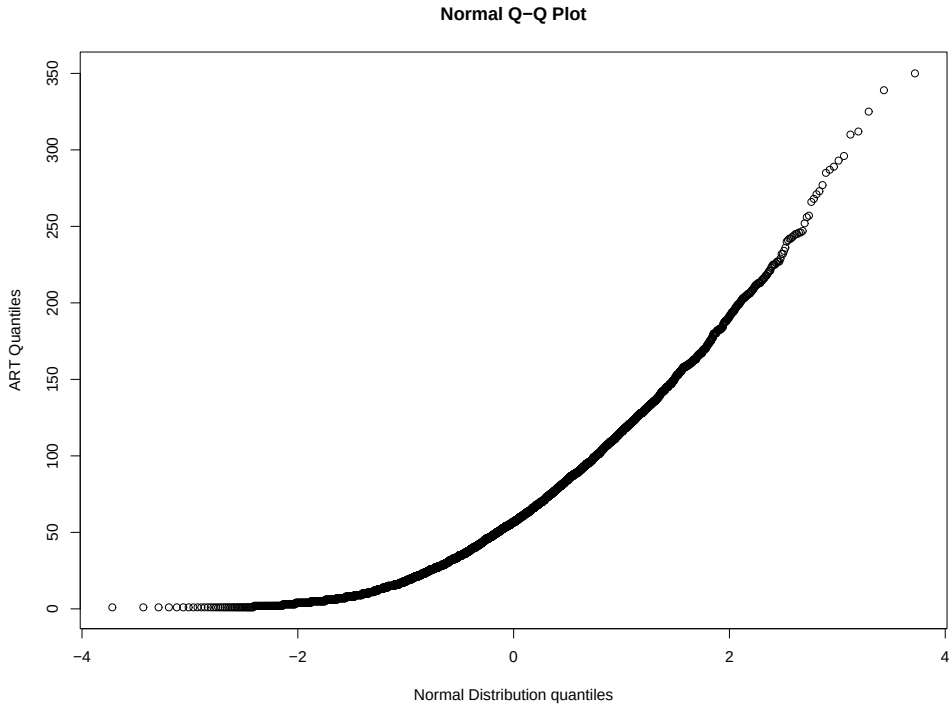



Figure 3.32: Quantile-quantile plot of RRT F-measure distribution and known normally-distributed data

as the P-measure is an estimate of the probability of a *test set* detecting failure, the test set size must be determined beforehand. We can then obtain an estimate of the P-measure for that strategy, on that program, for that test set size. We simply repeatedly apply the strategy to obtain test sets of the desired size, run the program using the tests, and record the number of times a test set detects at least one fault. If the number of test sets is $n$, and the number of test sets in which at least one test case detected a failure is $f$, the P-measure can be estimated as $\frac{f}{n}$. But how accurate is this estimate?

Consider the result of testing with a test set as an independent random variable $t$ which has the value 1 if a failure was detected in that test set, and 0 otherwise. Then the P-measure is simply an estimate of $\mathrm{E}(t)$, and standard measures of statistical spread can be used to determine how accurate an estimate might be. The sample variance of a random variable can be calculated using the following formula:

$$\mathrm{var}(t) = \frac{n \sum_{i=1}^{n} t_i^2 - \left(\sum_{i=1}^{n} t_i\right)^2}{n\,(n-1)} \tag{3.6}$$

where $n$ is the number of test sets.

Consider the simple case where the probability of detection by a test case is a constant $\theta$. This is true for random testing, where $\theta$ is simply the failure rate, but not for ART. In any case, if we make that assumption the expected values for $\sum_{i=1}^{n} t_i^2$ and $\sum_{i=1}^{n} t_i$ can be calculated. Indeed, because the only two values that instances of $t$ can take are 0 and 1, the two summations will be identical, and their expected value is simply $nP$. For random testing, the expected value of $P = 1 - (1-\theta)^l$, where $l$ is the size of the test set.

To simplify further manipulation, we make the further approximation that $n\,(n-1) = n^2$. Therefore, the expected variance is approximately:

$$\mathrm{var}(t) \approx \frac{n\left\{n\left[1-(1-\theta)^l\right]\right\} - \left\{n\left[1-(1-\theta)^l\right]\right\}^2}{n^2} \tag{3.7}$$

This relatively complex equation simplifies to:

$$\mathrm{var}(t) \approx (1-\theta)^l - (1-\theta)^{2l} \tag{3.8}$$

The standard deviation $s$, is simply the square root of the variance. As the variance is between 0 and 1, the standard deviation will be greater than the variance. The standard error of $t$ can then be estimated as $\frac{s}{\sqrt{n}}$.

When typical values of $\theta$ and $l$ are chosen, the standard deviation can be quite high. Figure 3.33 shows the predicted standard deviation for various values of $\theta$ when $l = 100$. As can be seen, the standard deviation (and thus the standard error) is maximised when $\theta$ is about 0.007. By finding the derivative of the equation for the standard error, and then finding the root of this derivative, the exact value for the maximum can be determined: $\theta = \frac{2^{\frac{1}{100}}-1}{2^{\frac{1}{100}}}$. For this value of $\theta$, the expected P-measure for random testing is exactly $\frac{1}{2}$. This is unsurprising; when the P-measure is near 0 or 1 the amount of variability would be expected to be lower.

In this worst-case scenario, the expected standard deviation would be $\frac{1}{2}$. Fig 3.34

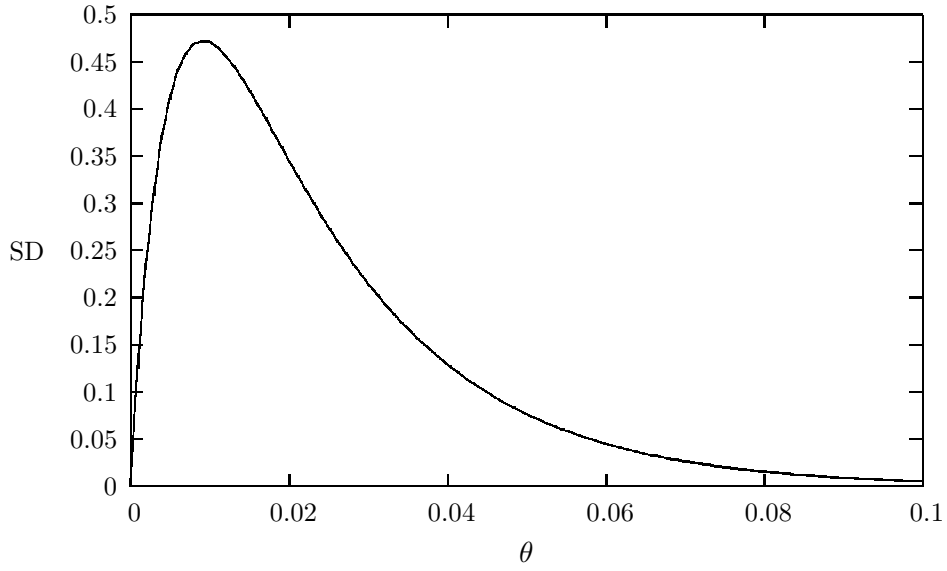



Figure 3.33: Standard deviation for $t$, $l = 100$

shows the expected standard error in this case. Even when $n = 1000$, the standard error is about 0.0158, giving an expected 95% confidence interval for P as wide as $0.47 \leq P \leq 0.53$. Such a broad confidence interval would make it impossible to detect subtle improvements in fault detection effectiveness.

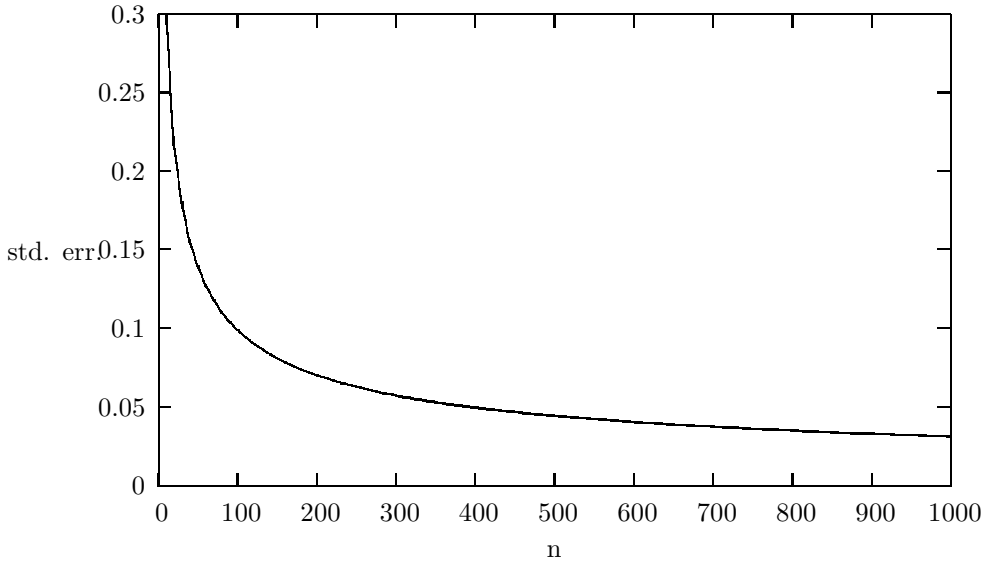



Figure 3.34: Standard error for P-measure estimates when expected P=0.5

## 3.6 The E-measure

The process for estimating the P-measure of a testing procedure, as described in the previous section discards a lot of information. The only information retained from running a test set is whether at least one failure was detected. With the E-measure, the more precise

measure of the number of failures detected can be used. Therefore, it might be expected that fewer test sets are required to calculate accurate estimates for the E-measure than for the P-measure.

We consider an experimental design to calculate the E-measure for a given testing strategy for a specific program under test. Like the P-measure, the E-measure depends on the number of tests in a test set $t$, so to make an estimate this must again be fixed. A very rough measurement could be made by simply creating a single test set of the required size. The number of failures detected would be an estimate of the E-measure.

If we assume that the probability of detecting a failure with a single test case is a constant $\theta$ (as in random testing), the expected value of the estimate $\mathrm{E}(e_1) = t \cdot \theta$. The variance of this figure will be rather high, though. Mosteller *et. al* [52], (p. 350) show that the variance will be $t\theta\,(1-\theta)$, and therefore the standard deviation $s_1 = \sqrt{t\theta\,(1-\theta)}$. To give some idea of the typical accuracy likely to be attained, Figure 3.35 plots the ratio $\frac{s_1}{\mathrm{E}(e_1)}$ for values of $\theta$ between 0.001 and 0.5, with $t = 100$ fixed. As can be seen, when $\theta$ is small the ratio is much too high to give an accurate estimate over this combination of test set size and failure rates.

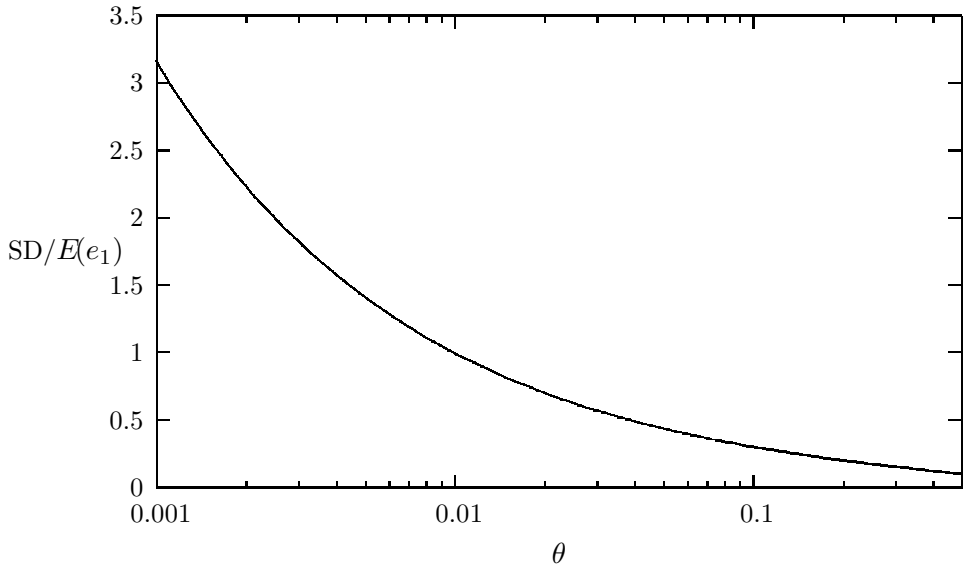



Figure 3.35: standard deviation/E-measure ratio for one-sample E-measure when t=100 ($\theta$ plotted log-scale)

Instead of trying to compute an estimate with a single test set, then, the experimenter will need to use multiple test sets and compute a more accurate estimate of the E-measure, $E$ by taking the mean of the number of failures detected in each set. If all other conditions of testing remain the same, and data is collected from $n$ test sets, the standard deviation of $E$ - the standard error f estimate, is $s_1/\sqrt{n}$. Figure 3.36 shows the standard error for an estimate made using $n$ test sets, with $t = 100$ and $\theta = 0.01$. In this case, the expected E-measure is 1.

For the case where $n = 1000, t = 100, \theta = 0.01$ the standard error is approximately 0.031. This would give a confidence interval for E of $0.94 \leq E \leq 1.06$ - making the estimate approximately as accurate as the confidence interval for the P-measure case discussed in the

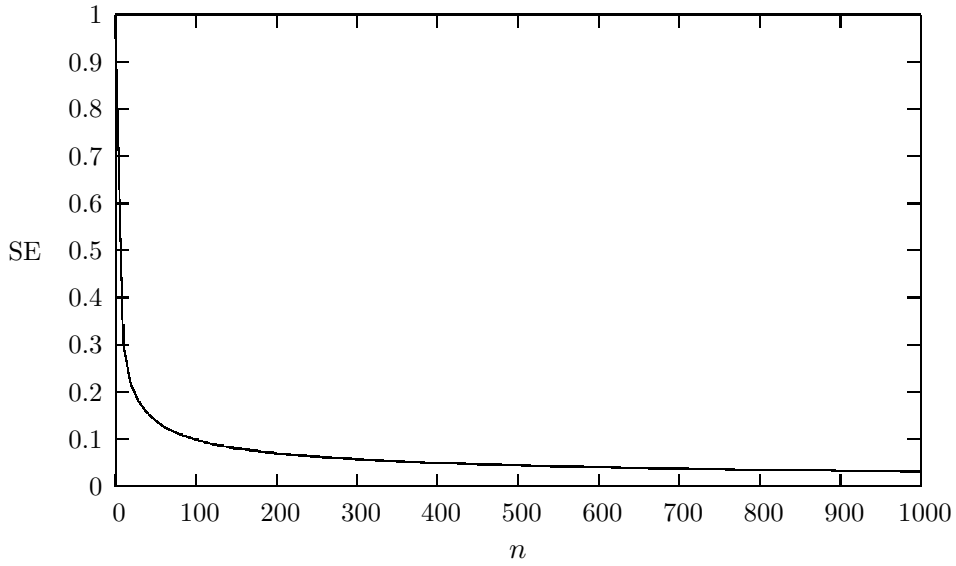



Figure 3.36: Standard error of estimate for E-measure for random testing, $t = 100$, $\theta = 0.01$

previous section. Again, the wide confidence intervals show that statistical variation would render undetectable small differences in effectiveness between different testing methods.

## 3.7 Discussion

In this chapter, we have shown that the F-measure will have a geometric sampling distribution, and examined the properties of this distribution in the light of testing practice. We have also shown that while the P-measure and E-measure have normal sampling distributions, large sample sizes are required to get accurate estimates of these metrics in typical cases.

In previous studies, effectiveness has been examined purely by quoting mean F-measures. Given the distributions we have shown here, we believe that this does not provide a complete basis for comparison. With the default assumption of normality, there is an expectation that the mean, median, and mode of the quoted statistic will be the same, and that data will be clustered around the mean in the usual pattern. These assumptions are false, in this case, with the median substantially smaller than the mean.

Our simulation study showed that for block patterns, FSCS-ART and R-ART produce characteristic sampling distributions that resemble a geometric distribution much more closely than the normal distribution. The distinctive feature of both distributions is a much steeper drop-off than in random testing. A much less pronounced drop-off is evidenced for strip patterns. This is consistent with previous studies that have shown a large reduction in mean F-measure using FSCS-ART and R-ART on block patterns, and a much more modest reduction for strip patterns. The sampling distributions of FSCS-ART and R-ART are very similar to each other. Although they are based on different intuitions to evenly spread test cases, their performance is shown here to be very similar. This characteristic drop-off indicates that for ART methods, there will be a quite large proportion of runs that detect failures very quickly, and a quite high percentage of runs below the mean.

### 3.7.1 Robustness of ART methods

One interesting observation can be made of the effectiveness of FSCS-ART and R-ART in the most unfavourable case - the point failure pattern. Previous studies have shown that in these cases, the mean F-measure is very similar to random testing. Our results show that not only are the means similar, the sampling distributions are very similar. This provides further evidence of the robustness of the ART methods, in that, at worst, their effectiveness is similar to that of random testing.

### 3.7.2 Implications for previous related studies

Given these non-normal and somewhat divergent distributions, the reader may question whether previous reported statistical comparisons of adaptive random testing methods have provided a full and complete picture. Consider first the descriptive statistics. One of the consequences of the well-known Central Limit Theorem is that the distribution of a sample mean will tend towards normality for sufficiently large sample sizes, regardless of the distribution of the population being sampled. Therefore, confidence intervals for population means can be accurately estimated regardless of the distribution of the sampled population. The only caveat outstanding is whether the sample standard deviation is an accurate estimator of the population standard deviation; Mendenhall and Sincich ([49], p. 307) state that the approximation is generally "quite satisfactory" if the sample size is over 30. The sample sizes used in these earlier studies were over an order of magnitude greater than 30, therefore the estimates were quite justified. The studies also quoted sample standard deviations, and used them to dynamically decide whether the sample size was large enough (when the confidence interval for the mean was small enough). This also appears to be acceptable. None of the studies quoted confidence intervals for the standard deviation, however, a naive approach to doing so would have been statistically erroneous, as the standard methods for doing so assume that the sample is approximately normally distributed regardless of the sample size ([49], p. 333).

As far as inferential statistics are concerned, the main such statistics were comparisons of two sample means. Given the assumption of "large" ($> 30$) sample sizes (and most sample sizes were at least an order of magnitude greater than this), testing whether the difference between two sample means is statistically significant does not require any assumptions about the sampling distributions of the two populations ([49], p. 374). However, while the inferential statistics actually performed in the past are valid, other, more elaborate types of inferential statistics often have more stringent assumptions, and great care should be taken when applying them to non-normal situations.

It is important here to differentiate between the statistical *validity* of descriptive or influential statistics, and their actual *usefulness*. Any "large" sample could be used to calculate sample means, or compare them. However, as we have shown at length, just because a sample size is large enough for the results to be statistically valid, it doesn't mean that they are accurate enough to be useful. With a sample size of 100, for instance, would likely generate confidence intervals so wide as to make the sample mean near useless, and inferential statistics comparing means would lack the power to show that differences

were significant. It should also be noted that most statistically significant results quoted in past studies were highly statistically significant and the sample sizes were extremely large. The distributions of the various test case selection methods are also quite similar. All of these properties are, in general, good indicators that statistically significant results are genuine.

As well as being statistically valid, the results quoted in previous studies of ART are meaningful. While the sampling distributions of random testing and ART methods are different, they are not radically different - a lower mean F-measure from ART also corresponds to a lower median and less likelihood of a very high F-measure on a single test set.

As discussed in section 3.3, when failure rates ($\theta$) are small, the standard deviation of the F-measure is about $\frac{1}{\theta}$, which is the same as the mean. Hence, our theoretical result has explained why in previous simulations, the mean and standard deviation are very close to each other.

### 3.7.3 Guidelines for testing

Given the distinctive, and slightly unusual sampling distributions shown by FSCS-ART and R-ART, it would seem that, while statistically valid, simply quoting means and standard deviations to characterise the F-measures of the methods is less than optimally illustrative. The automated nature of simulation and experimental studies in this area makes it quite straightforward to obtain very large sample sizes, and if sample sizes are large enough, confidence intervals become so small that any measurable difference is statistically significant. If this is done, the entire sampling distributions of methods can be accurately characterised and compared using graphical methods. We believe that, for the purposes of comparing effectiveness, a cumulative histogram is a particularly useful comparison tool.

The complications revealed by the F-measure simulations might tempt the reader to discard the F-measure and choose the E-measure or P-measure to compare testing methods. The results in sections 3.5 and 3.6 do not provide any support for such a change. All of the measures are characterised by high statistical variability, and a consequent need for large sample sizes for proper estimation. It seems to be the nature of the raw data itself, rather than the particular summary statistic used.

The difficulty of performing statistical analysis on testing data is, on reflection, quite predictable. The entropy of the raw data - the results of running tests - is very low. Each test gives a single bit of data, and in most cases the probability of detecting a failure is extremely low. In this sense, therefore, the density of *information* in the data is extremely low, and therefore, intuitively, a great deal of it is required for meaningful, accurate analysis.

The large amount of data that will be necessary for comparing testing methods is not particularly problematic for cases where test data can be generated, executed, and evaluated automatically and speedily. However, it poses a very considerable challenge in those cases where manual intervention is required for one or more of those steps. These are

precisely the cases where small, incremental improvements in testing effectiveness would be most useful! Unfortunately, it seems that mathematics is not going to be terribly cooperative. Decisions on testing methods may have to be made on the basis of results obtained in problem domains where the large amount of data required can be collected.

# Chapter 4

# Pixel-ART

## 4.1 Introduction

The results in chapter 2 show that if the shape, size, and orientation of the failure patterns are known but the location within the input domain is not, the problem of generating the optimal failure pattern is equivalent to covering the input domain with the minimum number of "stamps" of the same size and shape as the failure pattern.

This problem, hence referred to as the *stamping problem*, appears to be novel. It is superficially similar to *packing* [69], which covers a class of mathematical problems involving filling as much of a shape with as many non-overlapping copies of other, smaller shapes as possible. It is not, however, similar enough to suggest that techniques used for packing are directly applicable to this new problem. Obviously, packing could be used in the case where the number of tests was sufficiently small as to not require any overlap, but it is unknown whether a complete, optimally dense packing is the optimal starting point for a complete stamping. In any case, much of the packing literature appears focussed on optimal solutions for specific cases, rather than algorithms for arbitrary shapes. No straightforward transformation of the stamping problem into well-known optimisation problems like the knapsack or bin packing problems [22] suggested itself. However, given that even an approximate solution to the problem could give a lower bound for possible gains in effectiveness, a heuristic, similar to the ART heuristic was examined.

## 4.2 Greedy heuristic for stamping

An initial study examined a straightforward "greedy heuristic" for the stamping problem.

The problem domain was changed somewhat to make the problem more tractable. Instead of a subdomain of $\mathcal{R}^n$, the simulated input domain was a 2 dimensional integer space. In this case, the failure pattern shapes, and the input domain, can be represented on a grid, just as in a bitmap image representation. In this way, "stamps" could be placed incrementally on the input domain, and the extra coverage gained by placing a stamp in a specific location very easily calculated.

A greedy heuristic similar to FSCS-ART was used to place the stamps to cover the input domain as efficiently as possible:

1. Randomly generate $k$ candidate stamp placements, by randomly generating anchor points. All $k$ candidates should have anchor points chosen such that the entire failure pattern lies within the input domain.

2. Choose the candidate which covers the greatest number of previously uncovered pixels in the input domain.

The efficiency of this stamping method was examined by comparing it to random stampings. A variety of shapes to generate stampings for were chosen, as shown in figure 4.2. Stamping was performed in a 256x256 square domain. After each stamping, the proportion of that domain covered by the stamping was recorded. For each shape, two stamping runs were performed, one using the greedy heuristic, and one using pure random placement. Figure 4.2 clearly shows that the greedy heuristic technique achieved significantly greater domain coverage than random for the same number of stampings.

## 4.3 Greedy heuristic for fault finding

On the basis of this, and the results from the previous chapter, it appeared reasonable to use this "stamping" method to generate test cases optimised for detecting a specific failure pattern. However, it was unclear whether tests selected in this manner would be more effective than ART. We therefore ran another simulation study to examine the effectiveness of the method for failure detection.

To actually select test cases, an anchor point on the boundary of each pattern was defined. When a pattern was selected for stamping, the test case was selected according to the absolute position of the anchor point in the domain. As a basis for comparison, FSCS-ART was used to place test cases. The same bitmaps as the previous study were used as failure patterns, and a 256x256 integer domain was used for the experiment.

Preliminary experimentation suggested a potential problem with this scheme: poor effectiveness if a failure pattern is close to certain edges of the input domain. A modified procedure was devised: instead of discarding all candidates whose corresponding coverage zone lies partly outside the input domain, the candidate was moved towards the interior of the test region such that the outermost portions of the corresponding pattern were on the input domain boundary.

For an experimental run, the bitmap was randomly placed in the input domain, and test cases selected using the method until a failure was detected (the test case was within the area covered by the bitmap in the input domain), and the number of required test cases recorded. 2000 runs were performed for each stamping.

## 4.4 Results

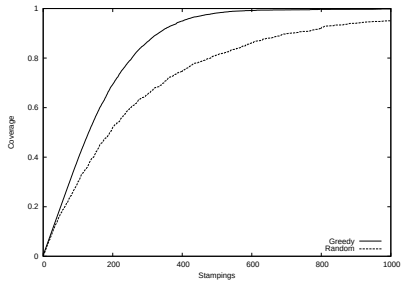
The graphs (Figure 4.3–4.11) show the cumulative probability distribution for the F-measure for random testing, FSCS-ART, and the original and modified greedy schemes. The histograms show that, regardless of the shape, in a large majority of runs the greedy

(a) Square (b) Diamond (c) Star

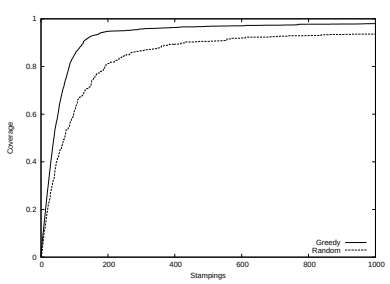
(d) Equilateral Triangle (e) Exponential Curve (f) Irregular Shape

(g) Right-Angled Triangle (h) "T" shape (i) Thin Strip

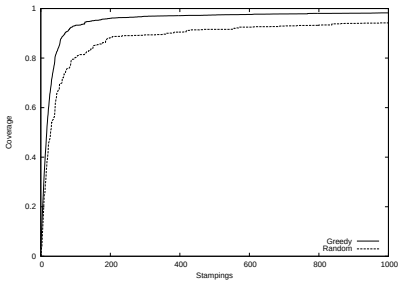
Figure 4.1: Coverage using random and greedy stamping methods of various shapes in a 256x256 square domain



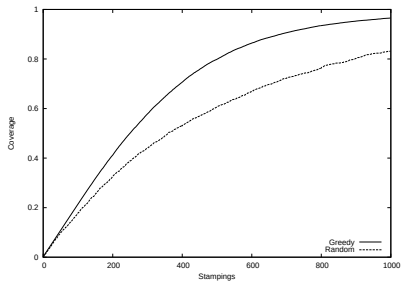

(a) Square



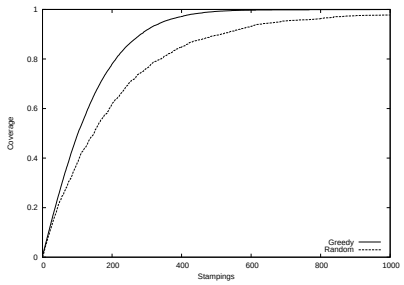

(b) Diamond



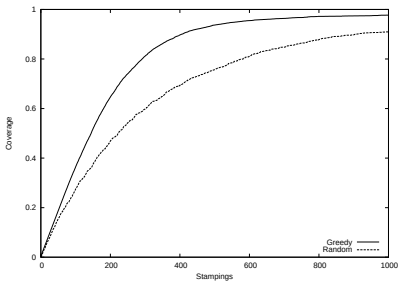

(c) Star



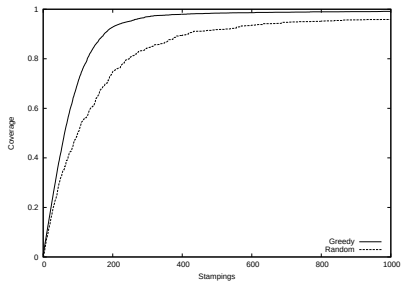

(d) Equilateral Triangle



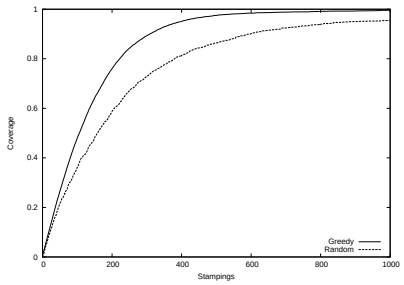

(e) Exponential Curve



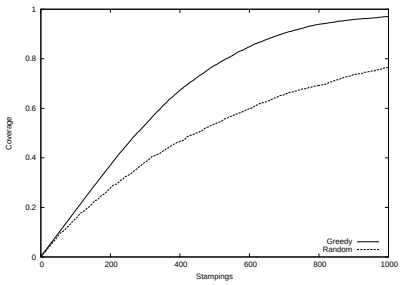

(f) Irregular Shape




(g) Right-Angled Triangle




(h) "T" shape




(i) Thin Strip

Figure 4.2: Coverage using random and greedy stamping methods of various shapes in a 256x256 square domain

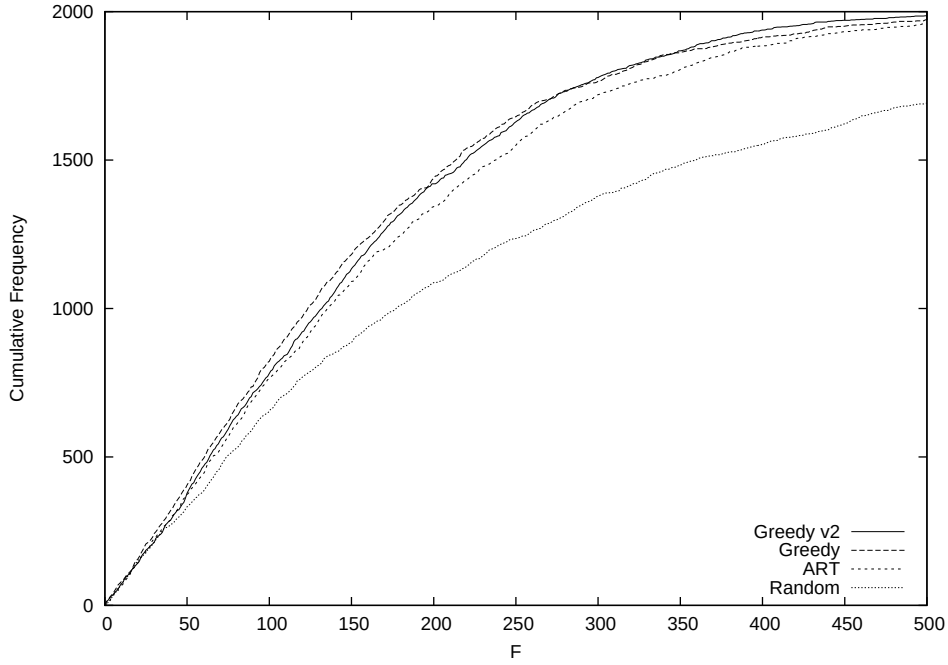



Figure 4.3: F-measure distribution: Square ($\theta = 0.0038$)

schemes are more effective at detecting the simulated patterns than either FSCS-ART or random testing. Unsurprisingly, the advantage is greater on shapes that are less block like, such as the very thin strip (Figure 4.11) and the irregular shape (Figure 4.8, and much smaller on the square pattern 4.3. In most cases, the modified greedy method outperforms the greedy one.

In a few cases, such as the diamond pattern (Figure 4.4), while the greedy methods generally perform better than FSCS-ART, the histogram shows that ART has slightly better worst case performance. Examination of the raw data suggests that the worst-case performance of the greedy method usually occurs when the failure pattern is close to the edge of the input domain. This is the opposite of FSCS-ART, where the distribution of test cases, as shown in section 3.1.1 suggests that the best-case performance will occur when the failure is located at the edge of the input domain.

## 4.5 Discussion

In this chapter, we have examined whether the "stamping" approach can be used to improve on the failure-detection effectiveness of adaptive random testing. We have shown that a straightforward greedy heuristic can be used to achieve much more efficient stampings than random placement, and, assuming a known failure region shape, can be used to detect that failure region shape more quickly than FSCS-ART.

This method is clearly not, as it stands, a practical software testing method. It is hard to conceive of a situation where a tester would have such specific knowledge about the shape and size of failure pattern without corresponding insights into the likely location of the failure pattern in the input domain. The method will work very poorly if the failure

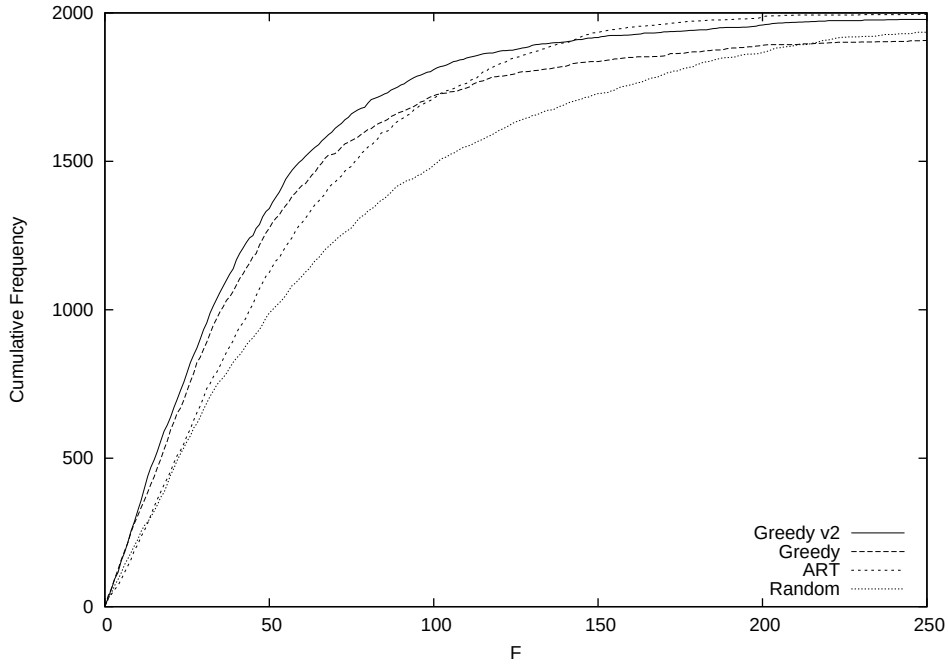



Figure 4.4: F-Measure distribution: Diamond ($\theta = 0.014$)

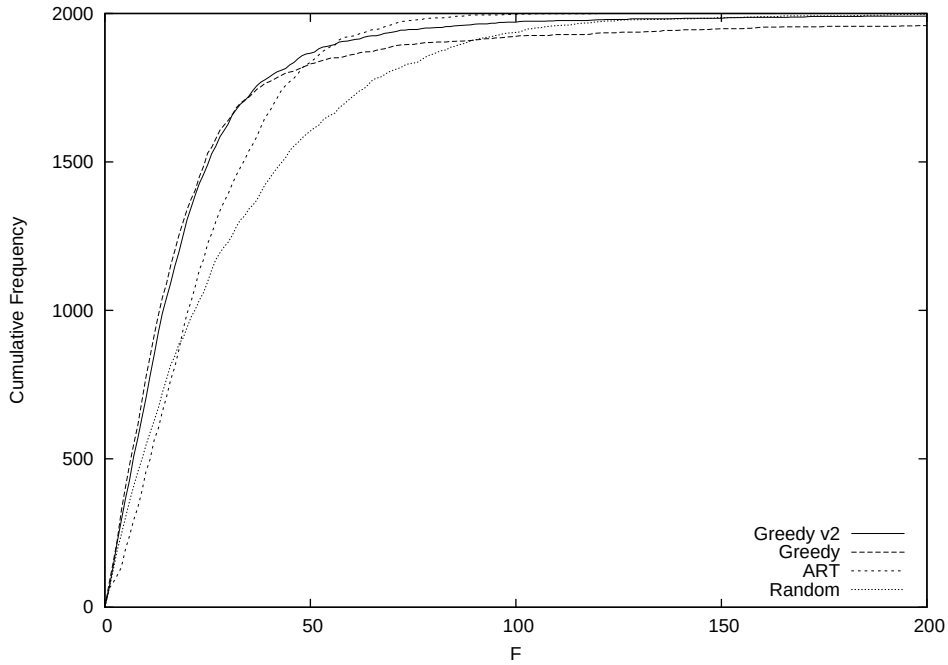



Figure 4.5: F-Measure distribution: Star ($\theta = 0.032$)

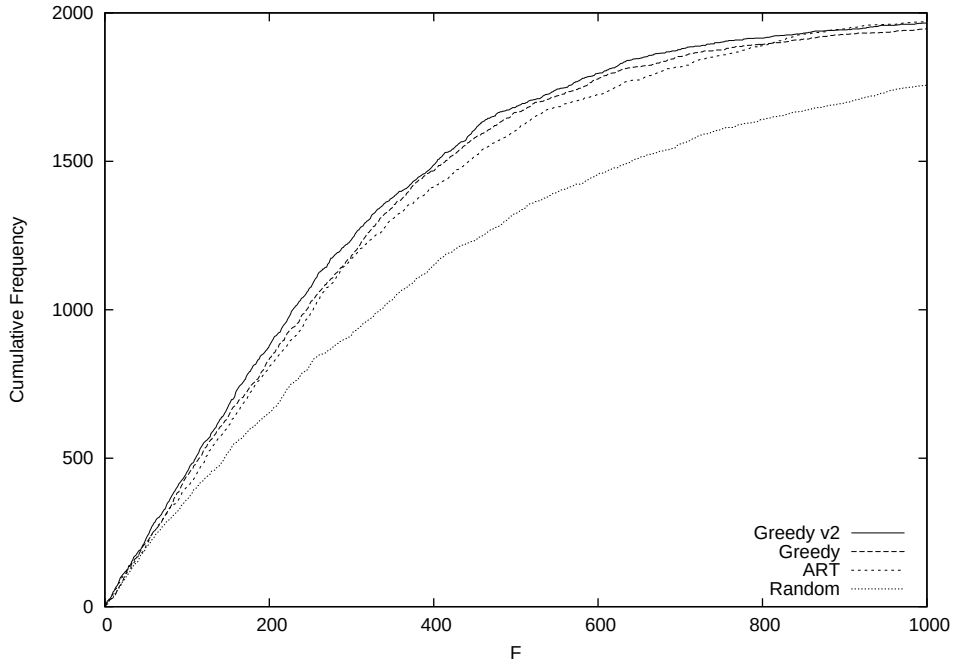



Figure 4.6: F-Measure distribution:Equilateral Triangle ($\theta = 0.0021$

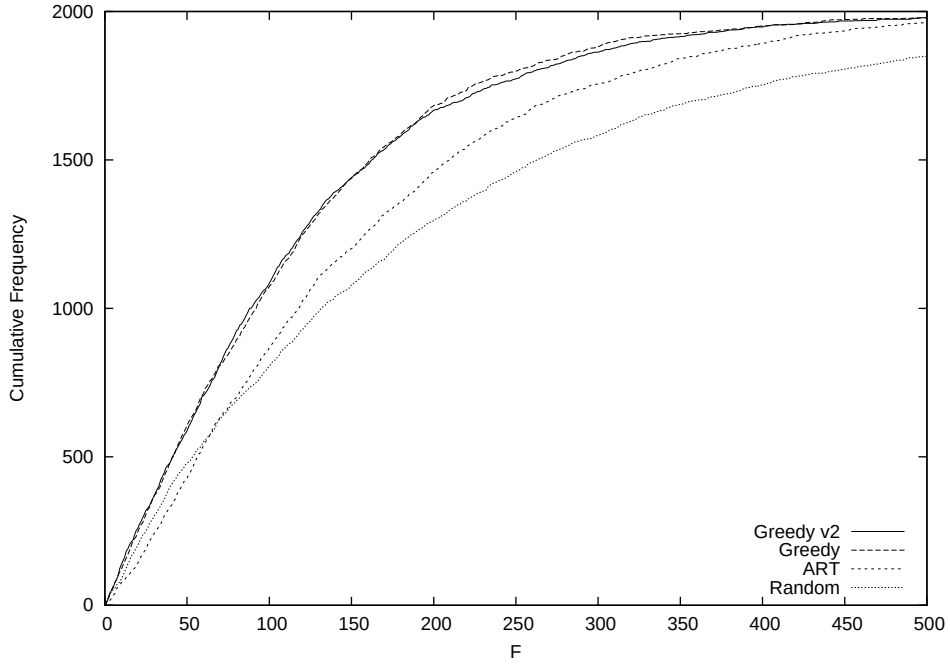



Figure 4.7: F-Measure distribution:Exponential Curve ($\theta = 0.0052$)

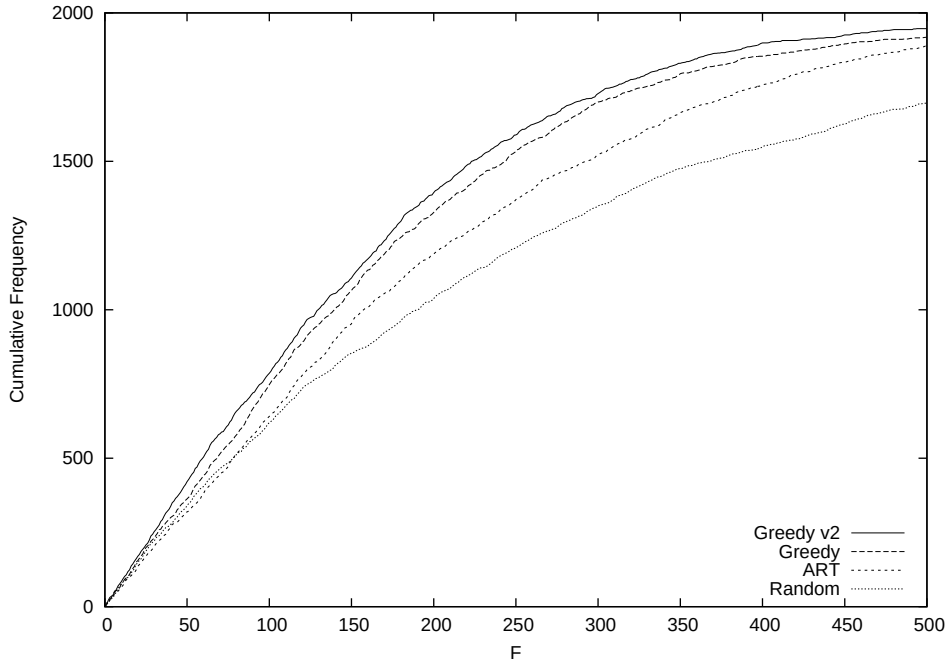



Figure 4.8: F-measure distribution:irregular shape ($\theta = 0.0037$)

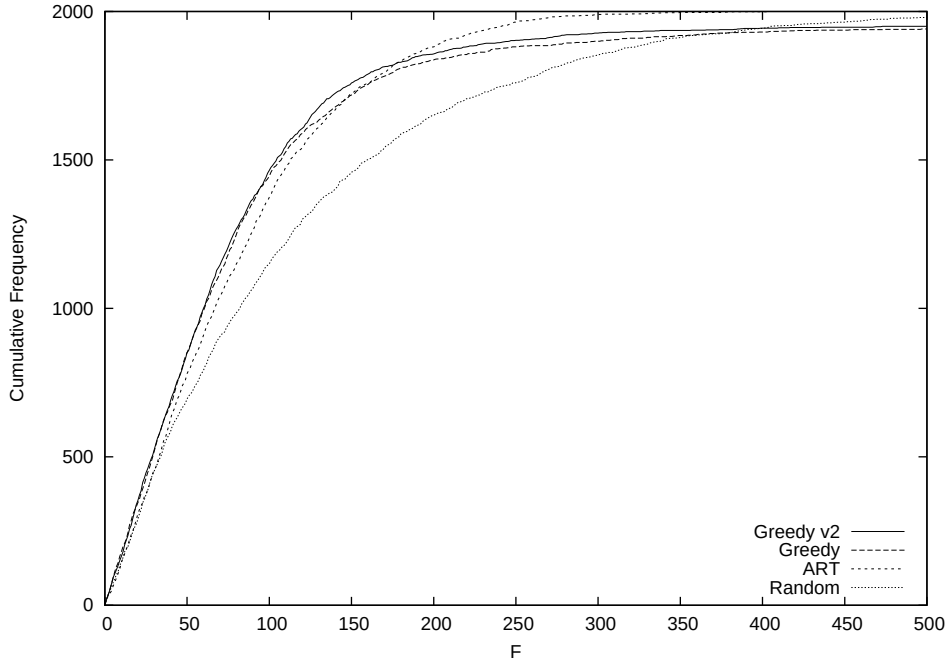



Figure 4.9: F-measure distribution:Right-angle triangle ($\theta = 0.0087$)

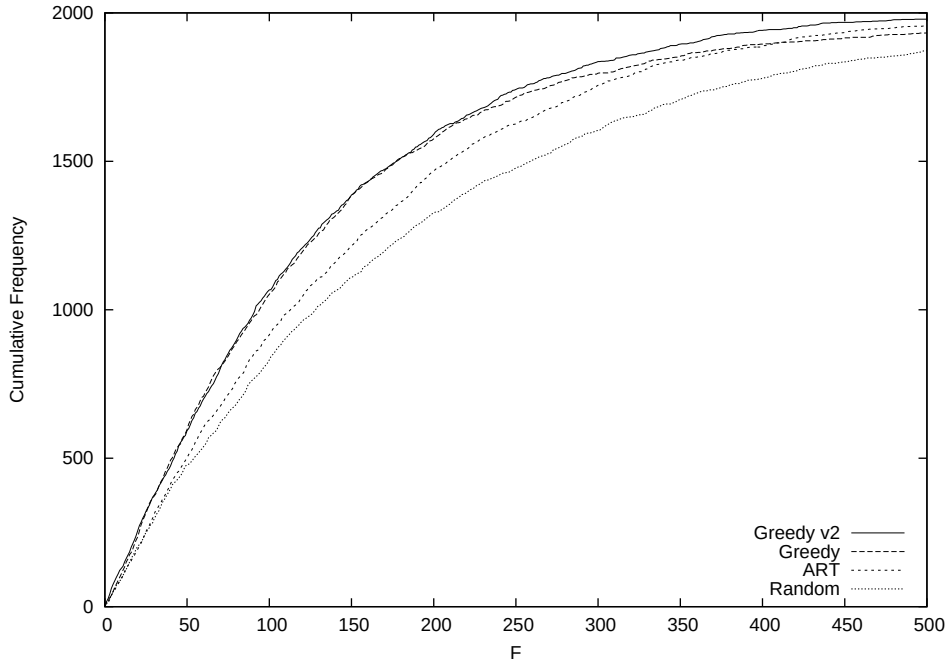



Figure 4.10: F-measure distribution:Tee shape ($\theta = 0.0054$)

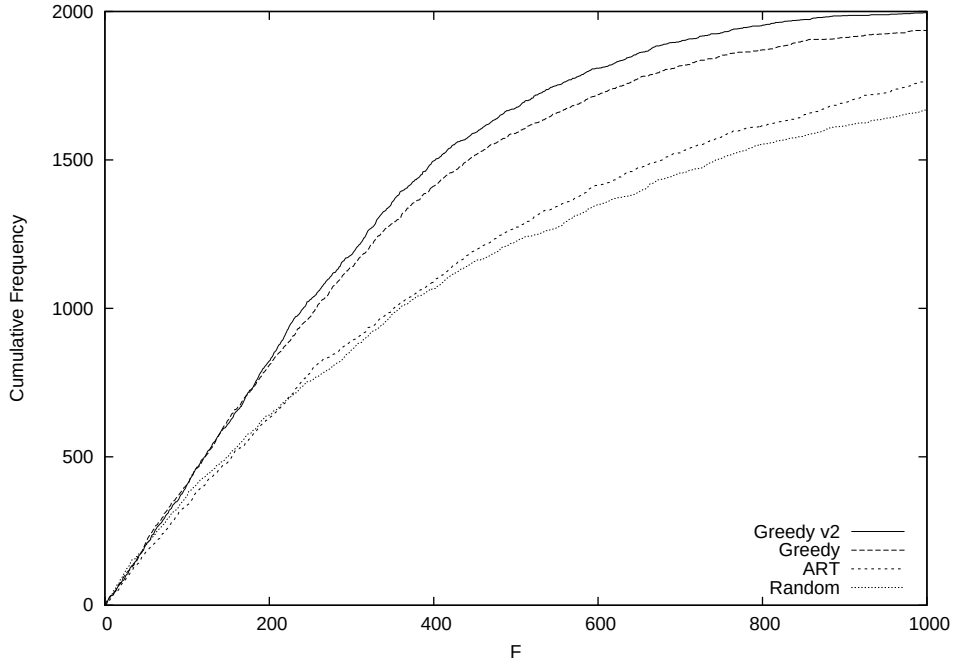



Figure 4.11: F-measure distribution:thin strip ($\theta = 0.0018$)

pattern is different from the one being searched for; for instance, if the failure pattern is smaller than the search pattern the method may simply cover the entire domain and from that point degenerate to random testing.

One pattern where the method showed particularly striking improvement over ART was the "thin strip" pattern. Previous experiments have shown that ART does not improve much over random testing in this situation. This can be explained in terms of the optimal patterns described in the previous chapter. An optimal test set for detecting strip patterns is one in which tests are very widely spaced parallel to the long part of the strip, and more closely spaced along an axis perpendicular to it. By contrast, ART test sets are evenly spaced along all axes equally.

A version of ART designed to detect strip patterns would be highly desirable. However, the present method doesn't appear to be useful as the basis of a method to do so, as the present method assumes a particular strip orientation, information is generally not available to the tester. However, given that domain errors are highly likely to result in strip failure patterns, more thought to how best to search for these failure patterns may be very productive.

# Chapter 5

# Adaptive Random Testing Through Dynamic Partitioning

## 5.1 Introduction

The next three chapters examine the second major shortcoming of ART - the overhead in selecting test cases. Three ways of doing so - using a less strict "wide spread" criterion, relaxing the randomness requirement somewhat, or implementing the existing method more efficiently. In this chapter we examine the first of these by using a less strict, easier to compute way of ensuring wide spread [12]. Rather than explicitly measuring distances between existing test cases and candidates, the new methods divide the input space into partitions and allocate test cases amongst them. Unlike previous partition-based testing techniques, however, partitions are generated dynamically as testing proceeds, rather than fixed partitions being decided before testing begins.

Chen *et. al* [14] have already explored one method which relaxes both the randomness and the wide spread conditions slightly to achieve lower overheads - "Mirror Adaptive Random Testing". In this method, the input domain is statically split up into a small number (in tests, between 2 and 9) of identically-sized partitions. ART is performed, essentially unchanged, within one of these partitions. Once a test is generated and executed within the "source" partition, the test is then mirrored across the other partitions using a low-cost transformation such as translation. The dominant cost of FSCS-ART (as discussed in more detail in chapter 7), is the cost of comparing the distance between candidates and existing test cases. Assuming the standard FSCS-ART implementation is used (rather than the more advanced, complex version discussion in that chapter), if $m$ partitions are used the number of comparison is $\frac{1}{m^2}$ of that required by simple FSCS-ART. Mirror-ART was tested empirically, and found to have failure-detection effectiveness very similar to ART.

MART is quite effective, but the overhead of the method remains fundamentally quadratic. It is also unclear how many partitions to divide the input space in to. Therefore, there still seems room for other methods that use different techniques, relaxing the "wide spread" requirement for lower overheads while hopefully retaining the effectiveness of ART.

In this chapter, two such methods are examined; one where the input domain is partitioned on the basis of executed test cases, and one where it is successively bisected. Both methods are low-overhead, and perform significantly better than random testing. As well as an empirical examination of their performance, one method has an extremely interesting connection to some of the known theory behind partition testing, with possible implications for future research.

### 5.1.1 Partition Testing

In essence, partition testing involves dividing the input domain up into a fixed number of disjoint partitions, and choosing test cases from within each partition. A typical example is the domain testing strategy proposed by White and Cohen [72].

Partition testing performs at its best if such partitions are *homogeneous*; that is, either all elements of the partition will reveal a failure when tested, or no elements will. In practice, the only partitioning strategy which guarantees this treats every single possible input to the program as an individual partition, trivially degrading the partition testing strategy to that of exhaustive testing. Therefore, we must usually settle for strategies which approximate this ideal. For instance, some strategies partition the input domain according to program execution paths. In these strategies, termed *path-coverage partitioning* inputs from each different partition trigger different "paths" through the program's source code. These partitions, however, are not necessarily homogeneous.

Partition testing has powerful intuitive appeal, and analytical results show that even simple partitioning schemes may be more effective at fault detection than random testing. It does however, have several drawbacks that may counterbalance its advantages over random testing. Chief among these is the initial overhead of partitioning. Complex partitioning schemes such as the path-coverage partitioning model require a significant amount of preprocessing before testing can begin. Even simpler schemes require some overhead, all of which is expended at the very start of the testing procedure - which can be wasteful, particularly in the preliminary phases of testing where faults may be detected after only a few test cases. Therefore, it is important to develop partitioning schemes that have acceptable overheads.

## 5.2 Background

### 5.2.1 Overview of Algorithms

As discussed in the previous section, Adaptive Random Testing is essentially a random testing method, but with a mechanism which attempts to widely spread test cases over the input domain. There are obviously different methods available to enforce the spreading of test cases.

In this paper, we propose two new ways to implement the notion of spreading. One way is to use the selected test case itself to partition the input domain. This method is termed *ART by Random Partitioning*. This method is outlined in section 5.3.

The second method we present in section 5.4 is to repeatedly bisect the input partition into smaller partitions, selecting test cases from those partitions which do not contain previously-performed tests.

### 5.2.2 Notation

We have conducted a simulation study to evaluate our two proposed methods. In our simulation, we assume the input domain is of two dimensions, and use the notation $\{(a,b)(c,d)\}$ to describe a rectangle with vertices at $(a,b)$, $(a,d)$, $(c,b)$, and $(c,d)$.

## 5.3 ART by Random Partitioning

### 5.3.1 Algorithm description

This method uses the most recently performed test to subdivide the input domain, and selects test cases randomly from within the largest remaining subdomain. As the partition process is performed on the basis of a randomly selected test case, we call this method *ART by random partitioning*.

This method is based upon the intuition that, given that the boundaries of the subdivisions are located according to the already executed test cases, choosing a test case randomly from within the largest subdivision is more likely to provide a test case distant from existing test cases. By putting all the regions thus generated into a global queue, and selecting the biggest region for the next test, eventually all regions are covered.

As previously mentioned, the algorithm is presented here in a form suitable for testing a program with two real inputs, $i, j$, where $i_{\min} \leq i \leq i_{\max}, j_{\min} \leq j \leq j_{\max}$. It is trivially extensible to any program with a fixed number of bounded real or integer inputs.

Pseudo-code for the algorithm is as follows:

**Algorithm 1 Adaptive random testing by random partitioning**

1. Set test region candidate set, $C$, to be a set containing one element, a pair of tuples representing the entire input domain. Formally, $C = \{\{(i_{\min}, j_{\min})(i_{\max}, j_{\max})\}\}$.
2. Select a new test region, $T\{(i,j)(s,t)\}$, to be the element in $C$ with the largest area and remove it from $C$. If there is more than one test region with the largest area, choose one randomly.
3. Randomly select a test point, $(x,y)$, from within T; that is, $i \leq x \leq s$ and $j \leq y \leq t$.
4. If the point $(x,y)$ is a failure-causing input, report fault detection and terminate.
5. Otherwise, divide the current test region, $\{(i,j)(s,t)\}$, into four test regions, $\{(i,j)(x,y)\}$, $\{(i,y)(x,t)\}$, $\{(x,j)(s,y)\}$, and $\{(x,y)(s,t)\}$ and add them to $C$.
6. Goto Step 2.

The operation of the algorithm as presented above can be seen in Figure 5.1. Initially, the candidate set contains one domain, the entire input domain. A test case is selected,

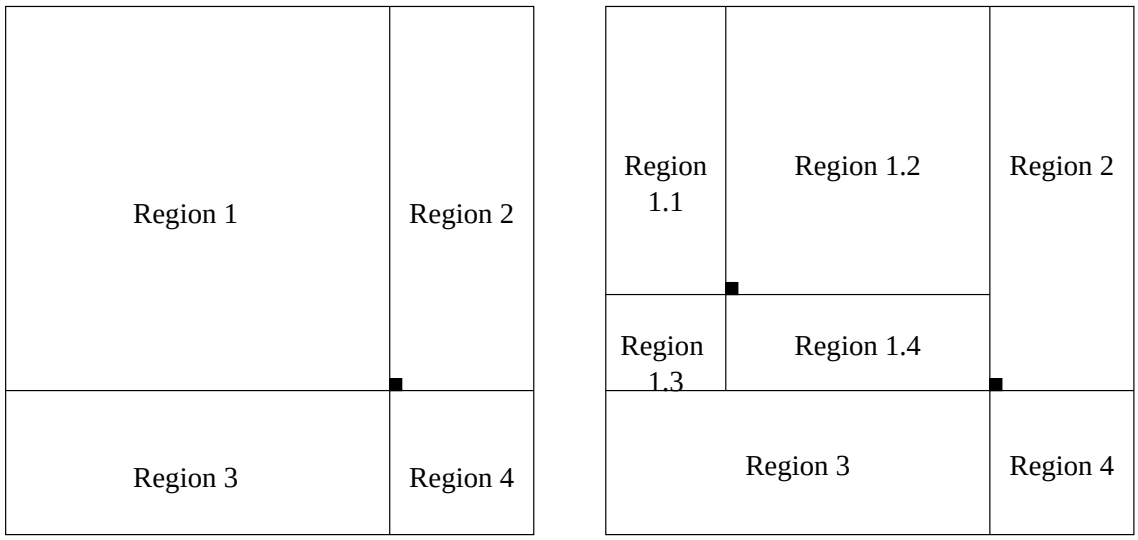



Figure 5.1: The operation of Algorithm 1 (ART by random partitioning). Each generated test case is used to further partition the region it was generated from.

which does not detect failure. Hence, the domain is divided into four subdomains (regions 1, 2, 3, and 4). In this case, region 1 is the largest region, and so the next case is selected from within it. Again, no failure is detected, so region 1 is partitioned into subregions 1.1, 1.2, 1.3 and 1.4. The algorithm will continue by selecting the largest domain from the 7 candidates.

While some mathematical analysis of the effectiveness of static partitioning schemes has been performed, we believe it is extremely difficult to develop general results for the effectiveness of dynamic partitioning schemes, due to the interaction of the shape of the failure pattern, and the input domain. Therefore we used an experimental approach to measure its effectiveness on simulated failure patterns of the types described in the introduction.

### 5.3.2 Experimental evaluation

The algorithm's effectiveness was measured experimentally. A square test domain was used. For each test run, a single, randomly located, failure region of the required size to produce the desired failure rate, and of the desired pattern was created. Block patterns were simulated by randomly designating a square region of the input domain, of size such as to give the desired failure rate as the failure region. For strip failures, two points on adjacent sides were chosen, and a strip joining the two points was designated the failure region. The width of the strip was then chosen to achieve the desired failure region, this width of course varied from run to run. Points close to the corners were not chosen to avoid having overly wide strips. To simulate the point pattern, 50 circular regions were randomly chosen from within the input domain. Regions overlapping already-chosen regions were rejected and another candidate chosen until 50 non-overlapping regions were created. For each type of failure pattern, statistics were collected for several different failure rates. For each combination of failure pattern and rate, 5000 test runs were performed, and the

average F-measure for each trial recorded.

| Failure Rate | Expected F of RT ($F_{rt}$) | **Block Pattern** Mean F of ART ($F_{art}$) | $\frac{F_{art}}{F_{rt}}$ (%) | **Strip Pattern** Mean F of ART ($F_{art}$) | $\frac{F_{art}}{F_{rt}}$ (%) | **Point Pattern** Mean F of ART ($F_{art}$) | $\frac{F_{art}}{F_{rt}}$ (%) |
|---|---|---|---|---|---|---|---|
| 0.01 | 100 | 77.4 | 77.4% | 92.8 | 92.8 % | 99.5 | 99.5 % |
| 0.005 | 200 | 154.9 | 77.4% | 194.4 | 97.1 % | 199.1 | 99.6 % |
| 0.002 | 500 | 390.0 | 78.0 % | 477.5 | 95.3 % | 498.5 | 99.7 % |
| 0.001 | 1000 | 796.5 | 79.6 % | 965.4 | 96.5 % | 980.1 | 98.0 % |

Table 5.1: F-measures for ART with random partitioning compared with random testing .

From Table 5.1, we can see that for block patterns, ART by random division has an average F-measure between 20-25% less than random testing. For strip patterns, the F-measure for ART by random division is generally around 5% lower than for random testing, whereas for point patterns the difference is negligible.

## 5.4 ART by Bisection

### 5.4.1 Algorithm description

This method again relies on partitioning the input domain, but rather than partitioning on the current test case, the process divides the input domain up into equally-sized partitions. As the testing progresses, and a test case from each partition is selected, the partitions are bisected, and test cases are chosen from those subdomains not containing test cases. In this way, we have ensured that test cases are widely spread out as they reside in different partitions.

We call this method *ART by bisection*, as it effectively divides the input domain into equal sized partitions, bisecting each existing partition as more test cases are generated. It ensures that test cases continue to be widely spread by only selecting new cases from partitions which contain no previous test case.

We again present the algorithm in a form suitable testing a program with two real inputs, $i, j$, where $i_{\min} \leq i \leq i_{\max}, j_{\min} \leq j \leq j_{\max}$, which is trivially extensible to the n-dimensional or integer cases.

Pseudo-code for the algorithm is as follows:

**Algorithm 2 Adaptive random testing by bisection**

1. Set the untested region list, $L_{untested}\{(i_{\min}, j_{\min})(i_{\max}, j_{\max})\}$, the entire test domain.
2. Set the tested region list, $L_{tested}$, to be null.
3. If $L_{untested}$ is not empty, randomly select a test region $T = \{(i, j)(s, t)\}$ from $L_{untested}$, and delete $T$ from $L_{untested}$. Otherwise goto Step 7.

4. Randomly select a point, $(x, y)$, within $T$ (such that $i \leq x \leq s$ and $j \leq y \leq t$).

5. If the point $(x, y)$ is a failure-causing input, report failure and terminate.

6. Otherwise, append the test region $T$ to $L_{tested}$, and goto Step 3.

7. Set a temporary list $L_{temp}$ to be null.

8. For each test region in $L_{tested}\{(i,j)(s,t)\}$, if $s-i \geq t-j$, set $p := \frac{s-i}{2}$, and divide the test region into two test regions, $\{(i,j)(p,t)\}$ and $\{(p,j)(s,t)\}$. Otherwise, set $q := \frac{t-j}{2}$, and divide the test region into two test regions, $\{(i,j)(s,q)\}$ and $\{(i,q)(s,t)\}$.

9. For each of the two test regions, if there is a test case in this test region, append the test region to $L_{temp}$, else append it to $L_{untested}$.

10. Rename $L_{temp}$ to $L_{tested}$, and goto step 3.

Figure 5.2 shows Algorithm 2 in operation. Initially, in 5.2(a), the first test case is selected from the input domain. In Figure 5.2(b), another test case is selected randomly from the "empty" domain. In 5.2(c), a further subdivision has taken place, with the third and fourth tests selected from the empty subdomains. The process continues in 5.2(d).

### 5.4.2 Experimental analysis

Algorithm 2 was tested in the same manner as Algorithm 1.

| Failure Rate | Expected F-msr of RT ($F_{rt}$) | **Block Pattern** F-msr of ART ($F_{art}$) | ($F_{art}/F_{rt}$) (%) | **Strip Pattern** F-msr of ART ($F_{art}$) | ($F_{art}/F_{rt}$) (%) | **Point Pattern** F-msr of ART ($F_{art}$) | ($F_{art}/F_{rt}$) (%) |
|---|---|---|---|---|---|---|---|
| 0.01 | 100 | 72.1 | 72.1 % | 91.9 | 91.9 % | 97.7 | 97.7 % |
| 0.005 | 200 | 148.8 | 74.3 % | 191.0 | 95.5 % | 178.7 | 98.1 % |
| 0.002 | 500 | 368.0 | 73.6 % | 475.3 | 95.1 % | 466.4 | 98.7 % |
| 0.001 | 1000 | 741.0 | 74.1 % | 966.2 | 96.6 % | 967.5 | 96.8 % |

Table 5.2: F-measures for ART with bisection compared with random testing .

Table 5.2 shows that for block patterns, ART by bisection outperforms random testing by approximately 25%. For strip patterns, there is 5-8% improvement on random testing. The method is marginally more effective than random testing for point patterns.

## 5.5 Discussion

In all cases, the experimental results show that both new methods significantly outperform pure random testing on block patterns, make a slight but useful improvement over random testing for strip patterns, and are about the same as random testing on point patterns. This is consistent with the performance of other ART-based methods [11], but the methods are significantly lower in overhead than other ART methods proposed so far.

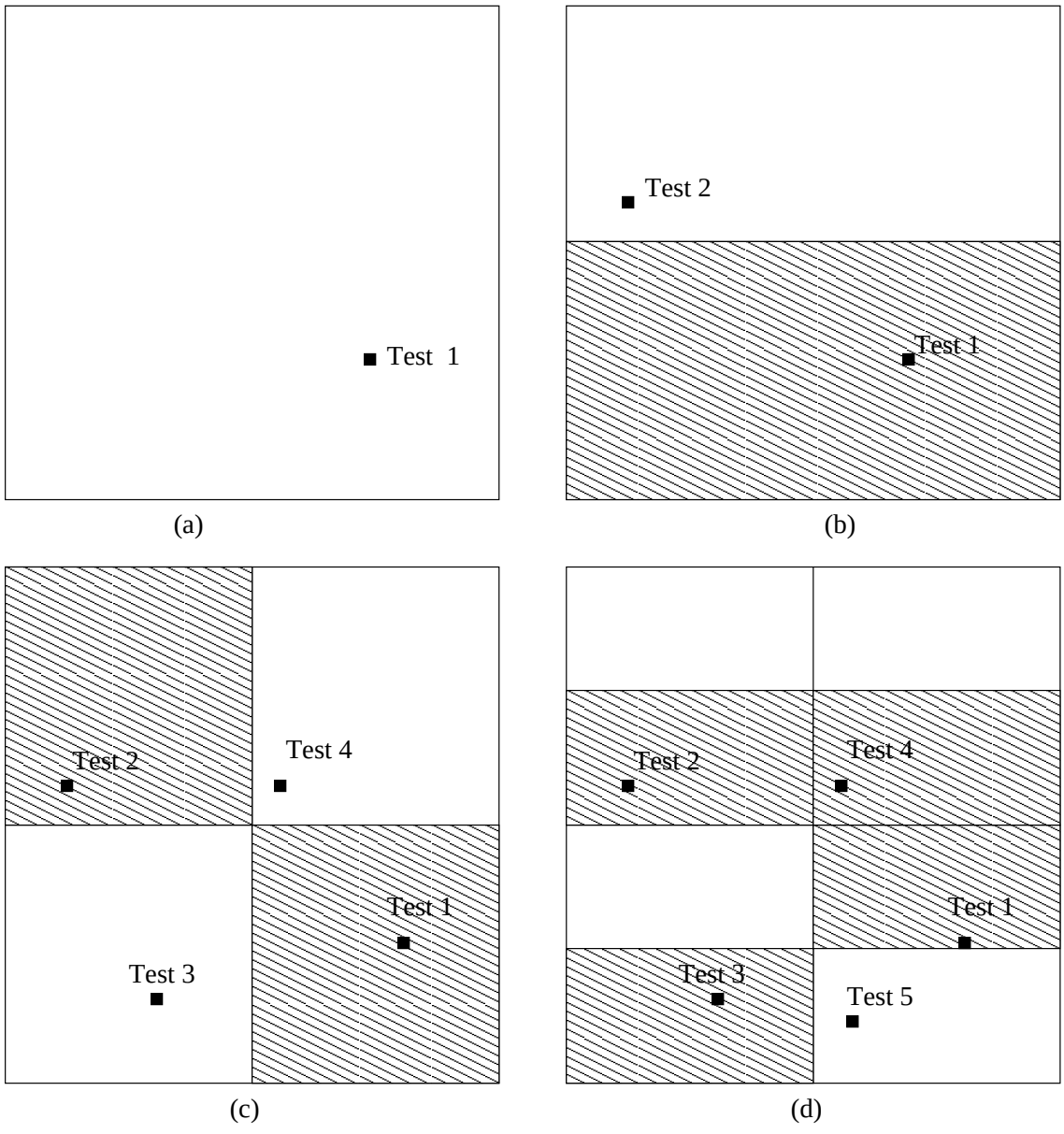



Figure 5.2: The operation of Algorithm 2 (ART by bisection)

Since partition testing is generally considered to impose greater processing overhead than random testing, much research has been aimed at identifying conditions under which partition testing will perform better than random testing. Two conditions that have been identified are the Equal-Size-Equal-Number strategy (Weyuker and Jeng [71]), and the Proportional Sampling (PS) strategy (Chen and Yu [19]). The equal-size-equal-number strategy divides the input domain into partitions of equal size and selects an equal number of test cases from each. The proportional sampling strategy requires a uniform sampling rate of test cases in all partitions (that is, the number of test cases selected from a partition is proportional to its size). Both of these strategies have been shown to have an equal or greater probability of detecting at least one failure than random testing.

The new algorithm, ART by bisection, exhibits properties of both the Equal-Size-Equal-Number and Proportional Sampling strategies. At any time, all partitions are of the same size and at most one test case is drawn from any partition. In other words, in addition to using fewer test cases to detect the first failure than random testing, ART by bisection also has an equal or greater probability of detecting at least one failure than random testing. Such an additional advantage has not yet been guaranteed by other ART methods.

Recently, Chen et al. [17] have reported interesting similarities between ART and PS. First, "there is still an element of randomness associated with the PS strategy". Secondly, they state that "intuitively, enforcing a uniform sampling rate somehow attempts, via the process of partitioning, to evenly spread out the test cases". They also point out that the motivation and intuition behind ART and PS are substantially different. ART was proposed with the goal of performing better for block and strip failure patterns, in terms of the number of test cases to detect the first failure. PS was proposed to perform better than random testing in terms of the probability of detecting at least one failure, in the absence of any information, albeit the failure pattern. ART by bisection clearly provides an example of a testing method which is both an efficient implementation of ART and meets the conditions of a PS strategy. It is of great interest to understand why such striking similarities exist despite the fact that ART and PS were developed from very different intuitions.

Chan et al. [11] have used the principle of exclusion to achieve wide spread of random test cases in their development of alternative ART implementations. ART by exclusion selects test cases only from parts of the input domain which are distant from all previously executed test cases. This is done by defining an exclusion region around each previously executed but non failure-causing test case. In ART by bisection, new test cases are not selected from partitions that contain already executed test cases. Therefore, ART by bisection can be viewed as using the "principle of exclusion" to achieve wide spread.

When compared against other ART methods, the two presented incur considerably lower overheads. Calculations of "distance" are not required, nor are candidates generated and discarded without use - the two most computationally expensive operations in previous ART implementations ([44], [11]). The methods presented clearly demonstrate that the overheads of implementing ART can be significantly reduced, whilst retaining its

much improved effectiveness over pure random testing. We believe they offer a significant practical improvement over pure random testing.

Though our methods involve the notion of partitioning, they are very different from conventional partition testing strategies where partitioning is normally performed prior to the selection of any test cases. So, while our methods can be viewed as a kind of partition testing, all partitioning is done "on the fly" so that the cost of processing is proportional to the number of test cases that have already been executed. In addition, there is no need for the number of test cases to be determined in advance. This can be a major advantage in practice, since it is common practice to stop testing after the first failure is detected.

Our method does have some limitations, some of which are shared by all random testing methods. Obviously, our method has the same limitations as random testing with regards to the requirement for a testing oracle. A program fault that causes a very low failure rate may not be effectively detected by random testing, and the improvement made by our methods is unlikely to provide much assistance in this case. It is also not straightforward to generate random tests for programs which take complex data structures as input. If some of the program's input parameters are suitable for dynamic partitioning, but others are not, a hybrid approach may be practical where the non-partitionable parameters are simply randomly generated.

An interesting unresolved question is how to apply these algorithms where the user wishes to continue testing after the first failure. In fact, this question is relevant not only to these new methods, but to most ART methods thus far described. The answer, however, depends closely on the goal of the tester. As section 2.4.2 describes, some previously-used effectiveness metrics are inappropriate in the context of adaptive testing methods. Some researchers have advocated the idea of *failure-pursuit sampling* [24] - once a failure is detected, trying similar test cases to assist in debugging. Alternatively, if the tester wishes to maximise the number of distinct faults detected, it may be better, for the purposes of selecting subsequent test cases, to simply continue selecting test cases using the algorithms as before, taking no notice of whether a failure was detected with a particular test case, and use some resource exhaustion criterion to determine when to stop testing.

In conclusion, I have proposed two innovative test case generation methods. They have been demonstrated to use significantly fewer test cases to detect the first failure than pure random testing, and have lower selection overheads than previous ART methods. Furthermore, the two methods integrate the concepts of random testing and partition testing. As far as we know, this is the first attempt to integrate the two concepts. My experimental results suggest to testers that if random testing is suitable for their purpose, it is worthwhile to seriously consider using these two new methods instead.

# Chapter 6

# Quasi-random testing

In this chapter, we explore the second of the three potential ways to reduce the overhead of ART - relaxing the randomness requirements somewhat. If randomness requirements are relaxed, it turns out that wide spread can be achieved very effectively and efficiently. To achieve this, we use a type of numerical sequence - *quasi-random sequences*.

Quasi-random sequences are a widely-used technique in numerical integration. Quasi-random sequences are *not* random, but they have many of the desirable properties of random sequences for our purposes, and they can be generated in a highly efficient, low overhead manner. Despite the much lowered overheads, they appear to be highly effective as test sequences. The properties of these sequences, and techniques for their generation, are discussed. We then compare their fault detection effectiveness and selection overhead to existing techniques in both a simulation study and with a set of error-seeded programs, which shows the methods' excellent effectiveness.

## 6.1 Quasi-Random Sequences

Quasi-random sequences are a somewhat imprecise term used to describe sequences of numbers with two different properties: a small *discrepancy* and low *dispersion*. Sequences generally used for one property, however, tend to display the other, hence the use of a general term.

Intuitively, a low-discrepancy sequence is a sequence of points in an $s$-dimensional half-open unit cube with the property that in any given subinterval points will be reasonably evenly distributed. The formal definition (see [55]) is as follows: For $N$ points $x_1, \ldots, x_n$ in the $s$-dimensional half-open unit cube $I^s = [0,1)^s, s \geq 1$, and a subinterval $J$ of $I^s$ we define the local discrepancy as:

$$\mathrm{D}(J;N) = \left| \frac{\mathrm{A}(J)}{N} - \mathrm{V}(J) \right| \tag{6.1}$$

where $\mathrm{A}(J)$ is the number of $x_i$ in $J$ and $\mathrm{V}(J)$ is the volume of $J$.

In other words, the local discrepancy of a point set $P$ and a subregion $S$ in the unit $n$-dimensional cube is the difference between the proportion of points in the subregion, and the proportion of the unit cube the subregion inhabits.

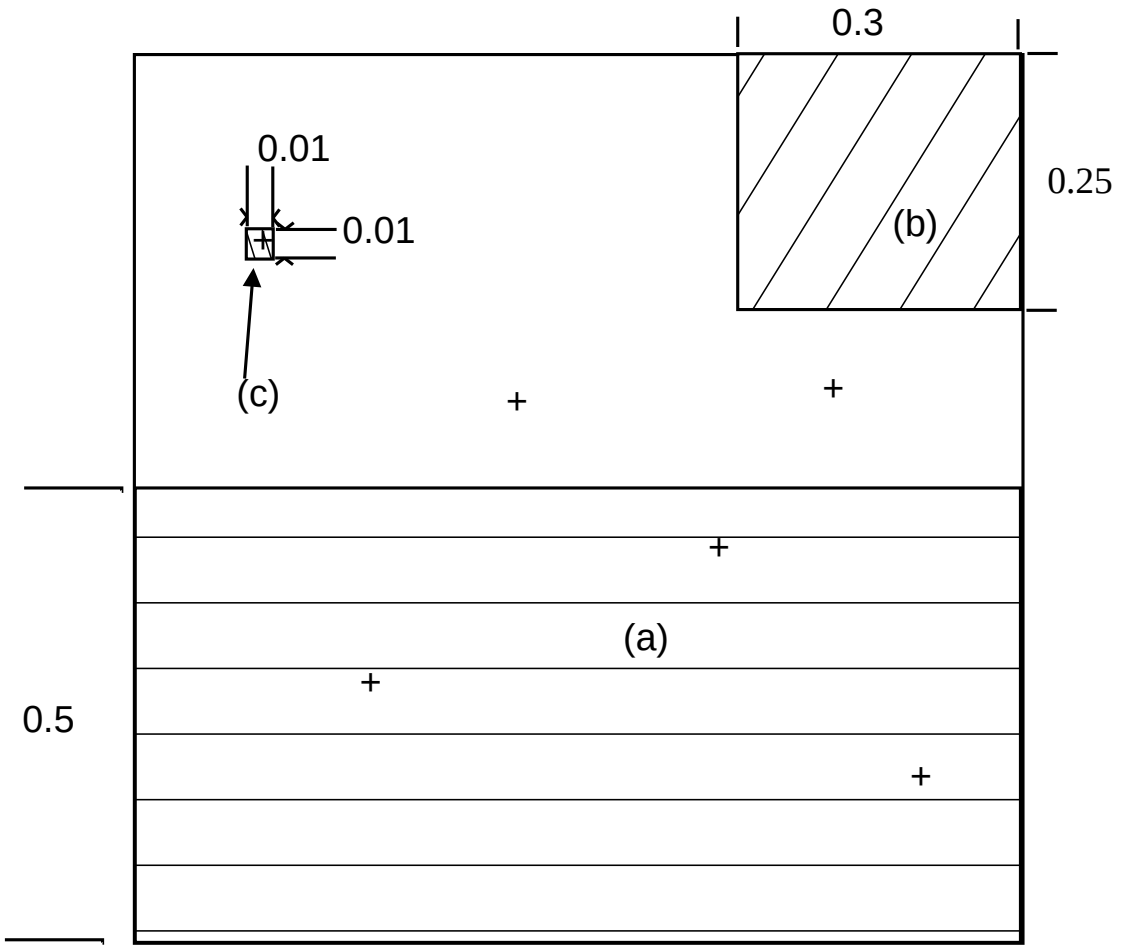



Figure 6.1: Discrepancy: a unit square containing six points and three subregions

Figure 6.1 shows a unit square (a 2-dimensional unit hypercube) containing a set of six points, and three regions. Region (a), the bottom half of the unit square, containing three of the six points, has zero discrepancy; half the area containing half of the points. Region (b) contains zero points, but has an area of $0.3 \times 0.25 = 0.075$, so the discrepancy would be 0.075. Region (c) contains $1/6 \approx 0.1666$ of the points and has an area of 0.00001, so the discrepancy is nearly 1/6; the region could be made arbitrarily small and thus the discrepancy would asymptotically approach 1/6.

The star discrepancy, $\Delta(N)$ of the points $x_1, \ldots, x_n$ is then defined as follows:

$$\Delta(N) = \sup_J |\mathrm{D}(J; N)| \tag{6.2}$$

where the supremum is extended over all half-open subintervals $J = \prod_{i=1}^{s} [0, u_i)$ of $I^s$.

In other words, given a set of points, the star discrepancy is the biggest possible local discrepancy for a rectangular subregion (or its $n$-dimensional equivalent) containing the origin.

The star discrepancy of a point set can be measured numerically; however, doing so for a large point set can be very computationally expensive as many different possible subregions need to be considered (not only subregions just including points to try and

maximise the proportion of points in the subregion, but subregions marginally excluding points to try and minimise that proportion).

Low-discrepancy sequences are sequences of points in $I^s$ where for all $N \geq 2$

$$\Delta(N) \leq B_s (\log N)^{s-1} + \mathrm{O}\Big((\log N)^{s-2}\Big) \tag{6.3}$$

and

$$\Delta(N) \leq C_s (\log N)^{s} + \mathrm{O}\Big((\log N)^{s-1}\Big) \tag{6.4}$$

where the constants $B_s$ and $C_s$ are as small as possible.

It is not difficult to construct sequences so that for *some* sequence lengths the discrepancy is low. However, constructing sequences such that the discrepancy remains low for all $N$ is much more challenging.

## 6.2 Construction of low-discrepancy sequences

Several methods for constructing low-discrepancy sequences have been proposed ([55] provides a good reference). One of the more modern is by Niederreiter [55]. The details of its construction, and the proof of the discrepancy bound, are very complex and hence are omitted in this paper; Niederreiter's method is only fully explained over two lengthy papers using advanced number theory.

All of these methods can be described in terms of what Niederreiter calls $(t, s)$-sequences. First, he defines an *elementary interval in base b* as an interval of the form

$$E = \prod_{i=1}^{s} \Big[a_i b^{-d_i}, (a_i + 1)\, b^{-d_i}\Big) \tag{6.5}$$

with integers $d_i \geq 0$ and integers $0 \leq a_i < b^{d_i}$ for $1 \leq i \leq s$.

Intuitively, an elementary interval is a rectangular (or $n$-dimensional equivalent) subregion precisely expressible in that base-$b$ number system (so for base 10, the elementary intervals would be those precisely expressible in decimal-point notation).

Next, Niederreiter defines the concept of a $(t, m, s)$-net in base $b$. Let $0 \leq t \leq m$ be integers. A *$(t, m, s)$-net in base $b$* is a point set of $b^m$ points in $I^s$ such that $\mathrm{A}(E; b^m) = b^t$ for every elementary interval $E$ in base $b$ with $\mathrm{V}(E) = b^{t-m}$.

In other words, $s$ indicates the dimensionality of the point set we are concerned with. For every elementary interval (aligning on the digit boundaries), which can be any proportions but must have a specified area, must contain the same number of points. So if $b = 2, t = 1, s = 2$ and $m = 2$, the requisite net would have 4 points. For every elementary interval of size $2^{1-2} = 2^{-1} = 1/2$, there must be precisely $2^1 = 2$ points in it. Note that the local discrepancy of any such elementary interval will therefore be zero.

A $(t, s)$-sequence is defined as follows.:

Let $t \geq 0$ be an integer. A sequence $x_1, x_2, \ldots$ in $I^s$ is called a $(t, s)$-sequence in base $b$ if for all integers $k \geq 0$ and $m > t$ the point set consisting of the $x_n$ with $kb^m < n \leq (k + 1)\, b^m$ is a $(t, m, s)$-net in base $b$.

So, for example, if $t = 1, s = 2, b = 2$, any sequence meeting the definition would require the point set $\{x_1, x_2, x_3, x_4\}$ to be a (1,2,2)-net, as would $\{x_5, x_6, x_7, x_8\}$, $\{x_9, \cdots, x_{12}\}$, and so on. $\{x_1, x_2, \cdots, x_8\}$, $\{x_9, \cdots, x_{16}\}$, and so on would have to form (1,3,2)-nets. Successively larger groups would have to form nets with a bigger value for $m$.

The discrepancy for $(t, s)$-sequences is bounded according to formulae in Niederreiter ([55], p. 53), which is lower than any other sequence yet known.

Niederreiter then provides a method for efficiently generating a $(t, s)$-sequence. While a complete method for any prime base is provided by Bratley *et al.* [9], they further show that base-2 sequences can be generated much more quickly (using the fact that base-2 arithmetic is the native form used by computers). For their purpose of quasi Monte Carlo integration, they showed that the base-2 sequences perform more effectively than other bases.

While the theory behind Niederreiter's method is very complex, its actual implementation is reasonably simple and efficient. Generating the next $N$-dimensional point in a sequence takes constant time and negligible storage. An implementation of Niederreiter's method for base 2 sequences is available as part of the GNU Scientific Library [33]; the implementation in this library is a direct C translation of original FORTRAN code written by Bratley *et al.*[9].

## 6.3 Simulation Study

An experiment was conducted to determine whether tests specified by a quasi-random sequence could detect failures more effectively than randomly selected tests.

With previous examinations of the performance of new testing methods, both simulation studies, where a simulated program failure is artificially created, and empirical studies, where real programs with failures in them, have been used. While empirical studies have clear advantages, the nature of the quasi-random sequence generator poses a considerable problem for the experimental design used in the past.

In other studies in this thesis, a mean F-measure for a testing method, for a particular program under test, have been estimated by repeatedly running the method, but obtaining a different sequence each time by using a different random seed in the generation process on each occasion. This will not work for testing based on quasi-random sequences. Bratley, Fox, and Niederreiter's implementation can only generate one unique sequence for a particular base, and given that only the base-2 generator was available only one unique sequence could be generated.

Therefore, as a preliminary investigation, the effectiveness of quasi-random sequences was tested by using a simulated failure pattern - a block failure pattern, that could be placed in random locations over multiple runs. While this is undoubtedly less realistic than using real program failures, it at least lets some statistically meaningful measurements be made.

| failure pattern size | F-measure for random testing ($F_{rt}$) | F-measure for quasi-random ($F_{qrt}$) | $F_{qrt}/F_rt$ (%) |
|---|---|---|---|
| 0.01 | 100 | 67.0 | 67.0% |
| 0.002 | 500 | 358.0 | 71.6% |
| 0.001 | 1000 | 788.8 | 78.9% |
| 0.0002 | 5000 | 3847 | 76.9% |

Table 6.1: F-measures for testing using quasi-random sequences

### 6.3.1 Method

For each experimental trial, a simulated program input domain of a unit square was created. A square "failure region" of a specified size was randomly placed inside the input domain. The quasi-random sequence generator was then used to generate to generate "tests" within the input domain, until a point was generated within the simulated failure region, indicating the detection of the "failure". The number of "tests" required to detect the failure region was recorded.

Four different experimental conditions were tried, with the parameter varied being the size of the simulated failure regions. They were of size 0.01, 0.002, 0.001, and 0.0002. For random testing with replacement, the expected mean F-measure for those experimental conditions was 100, 500, 1000, and 5000 respectively.

### 6.3.2 Results

It is clear from these results that testing using quasi-random sequences offers the possibility of considerable improvements in effectiveness over random testing. However, without an ability to somehow "randomise" the sequences there is no way to evaluate the effectiveness of the methods empirically with a small number of test programs; and, practically, the availability of a large number of sequences to test with is very convenient. Hence, I propose an enhanced approach using *randomized low-discrepancy sequences*.

## 6.4 Randomised low-discrepancy sequences

The restriction to small numbers of sequences is a significant restriction in the major application of low-discrepancy sequences: numerical multidimensional integration. While they are often more accurate than methods using random sequences, it is difficult to assess the accuracy of the results obtained [56]. Owen [56] therefore proposed to "scramble" quasi-random sequences in a randomised fashion.

### 6.4.1 Definition

Owen [56] defines randomised low-discrepancy sequences as follows:

Let $(A_i)$ be a $(t, m, s)$-net or a $(t, s)$-sequence in base $b$. The $i$th term in the sequence is written $A_i = (A_i^1, \ldots, A_i^s)$. We may then write the components of $A_i$ in their base $b$ expansion

$$A_i^j = \sum_{k=1}^{\inf} a_{ijk} b^{-k} \tag{6.6}$$

where $0 \leq a_{ijk} < b$ for all $i, j, k$.

A randomised version of $(A_i)$ is a sequence $(X_i)$ where elements $X_i = (X_i^1, \ldots, X_i^s)$ written as

$$X_i^j = \sum_{k=1}^{\inf} x_{ijk} b^{-k} \tag{6.7}$$

with $x_{ijk}$ defined in terms of random permutations of the $a_{ijk}$

$$\begin{aligned}
x_{ij1} &= \pi_{\mathrm{j}}(a_{ij1}) && (6.8)\\
x_{ij2} &= \pi_{\mathrm{j a_{ij1}}}(a_{ij2}) && (6.9)\\
x_{ij3} &= \pi_{\mathrm{j a_{ij1} a_{ij2}}}(a_{ij3}) && (6.10)\\
&\vdots && (6.11)\\
x_{ijk} &= \pi_{\mathrm{j a_{ij1} a_{ij2} \cdots a_{ijk}}}(a_{ijk}) && (6.12)\\
& && (6.13)
\end{aligned}$$

Each permutation $\pi$ is uniformly distributed over the $b!$ permutations of $\{0, 1, \ldots, b-1\}$ and the permutations are mutually independent. $\pi_j$ permutes the first digit in the base $b$ expansion of $A_i^j$ for all $j$. The second digit is permuted by $\pi_{ja_{ij1}}$, so the permutation applied to the second digit depends on the value of the first digit. The permutation applied to the $k$th digit depends on the values of the $k-1$ digits before it.

Owen then shows that if a $(t, m, s)$-net or $(t, s)$-sequence is randomised using the procedure described above, that it retains the defining properties of these sequences.

### 6.4.2 Implementation

While implementation of the method described appears straightforward, particularly where the base the number of permutations required by a naive implementation makes the storage requirements extremely high. In practice, the number of digits in the base $b$ representation is restricted to a limited number. In base 2, a number of other tricks can be employed, particularly as there are only two permutations in base 2 and thus a permutation can be represented as a single bit, and applied using a bitwise exclusive-or.

A number of implementations of this scrambling are freely downloadable for non-commercial use [41] [28] [57]. SamplePack [41], a C++ implementation was chosen for our experiments as it was readily available and easily compatible with our existing work.

### 6.4.3 Empirical study

An empirical study was conducted to compare the effectiveness of testing using the "scrambled" low-discrepancy sequences with ART and random testing. We follow the basic ex-

perimental procedure of Chen *et al.* [16].

In their study of ART [16], they used as a study basis twelve small numerical programs drawn from the book "Numerical Recipes" [59], and the "Collected Algorithms of the ACM" [3], varying in length from 30 to 200 lines. These programs each take a fixed length vector (varying in length between 1 and 4 for each program), with each vector element within specified bounds, as input, and output a number. The original programs (most originally written in FORTRAN) were translated line by line into C++ for convenience; as the programs were numerical routines that used scalar data structures, the translated versions are all but identical to the originals, differing only in the details of syntax, and produce the same output.

These programs were seeded with a number of errors. The errors are were of a number of different types, including arithmetic operator replacement (AOR), relational operator replacement (ROR), scalar variable replacement, (SVR), constant replacement (CR) and other (OTH). As a test oracle, the results from running the error-seeded program were compared with that of the unaltered program, with an error reported if the results differ. We use the same test programs so as to be able to directly compare the results with our previous experiments. Details of the program input domains, types of errors, and failure rate (from [16]) are presented in Table 6.2.

In an experimental run, the randomised low-discrepancy generator that generates vectors of the appropriate dimensionality is initialised using a random number generator as a source. As elements $X_i = (X_i^1, \ldots, X_i^n)$ in the low-discrepancy sequence are points in the half-open unit cube in $n$ dimensions, they are transformed into test points $T_i = (T_i^1, \ldots T_i^n)$ using the formula

$$T_i^j = X_i^j \times range_j + min_j \tag{6.14}$$

where $min_j$ is the lower permitted bound for the $j$th dimension of the input vector and $range_j$ is the permitted range.

The program under test is then executed with the test point until a failure is detected. The number of tests required to detect a failure (F-measure) is then recorded. 5000 runs were conducted for each program under test. Chen *et al.* used a statistical procedure to decide how many runs to conduct, stopping when the confidence interval for the mean F-measure was within 5% of the estimate. Typically, this resulted in around 1500 runs being conducted for each program under test. The larger, fixed number of 5000 was chosen because with the much faster computers available today there seemed no reason not to perform additional runs and get a more accurate estimation.

In terms of performance, the time taken to generate a randomized quasi-random sequence of of the lengths required was negligible on a 1.3GHz Pentium-M laptop.

### 6.4.4 Results

Table 6.3 shows the F-measure for the 12 test programs for randomized quasi-random testing, ART, and random testing. Note that the ART and random testing data is from Chen *et al.* [16].

| Prog Name | D | Input Domain Min | Input Domain Max | Error Type AOR | Error Type ROR | Error Type SVR | Error Type CR | Error Type OTH | Failure Rate |
|---|---|---|---|---|---|---|---|---|---|
| AIRY | 1 | (-5000.0) | (5000.0) | | | | 1 | 3 | 0.000716 |
| BESSJ | 2 | (2.0, -1000.0) | (300.0, 15000.0) | 2 | 1 | | 1 | | 0.001298 |
| BESSJ0 | 1 | (-300000) | (300000) | 2 | 1 | 1 | 1 | | 0.001373 |
| CEL | 4 | (0.001, 0.001, 0.001, 0.001) | (1.0, 300.0, 10000.0, 1000.0) | 1 | 1 | | 1 | | 0.000332 |
| EL2 | 4 | (0.0, 0.0, | (250.0, 250.0, | 1 | 3 | 2 | 3 | | 0.000690 |
| | | 0.0, | 250.0, | | | | | | |
| | | 0.0, | 250.0) | | | | | | |
| ERFCC | 1 | (-30000.0) | (30000.0) | 1 | 1 | 1 | 1 | | 0.000574 |
| GAMMQ<br>GAMMQ | 2<br>2 | (0.0,<br>0.0) | (1700.0,)<br>40.0) | | 3 | | 1 | | 0.000830 |
| GOLDEN | 3 | (-100.0, -100.0, -100.0) | (60.0, 60.0, 60.0) | | 3 | 1 | 1 | | 0.000550 |
| PLGNDR | 3 | (10.0, 0.0, | (500.0, 11.0) | 1 | 2 | | 2 | | 0.000368 |
| | | 0.0) | 1.0) | | | | | | |
| PROBKS | 1 | (-50000) | (50000) | 1 | 1 | 1 | 1 | | 0.000387 |
| SNCNDN | 2 | (-5000.0, 5000.0) | (5000.0, 5000.0) | | | 4 | 1 | | 0.001623 |
| TANH | 1 | (-500.0) | (500.0) | 1 | 1 | 1 | 1 | | 0.001817 |

Table 6.2: Details of programs used in empirical study (Note: D=dimensionality of program input domain)

| Program | $F_{quasi}$ | 95% CI ($F_{quasi}$) | $F_{art}$ | 95% CI $F_{art}$ | $F_r$ | 95% CI ($F_r$) |
|---|---|---|---|---|---|---|
| AIRY | 799 | (784, 814) | 799 | (779, 819) | 1381 | (1333, 1430) |
| BESSJ | 603 | (588, 617) | 467 | (452, 481) | 802 | (723, 831) |
| BESSJ0 | 522 | (511,533) | 424 | (413, 435) | 734 | (707, 760) |
| CEL | 1697 | (1666, 1728) | 1608 | (1552, 1663) | 3065 | (2955, 3175) |
| EL2 | 1019 | (996, 1041) | 686 | (662, 711) | 1431 | (1378, 1484) |
| ERFCC | 1092 | (1070, 1113) | 1005 | (980, 1029) | 1804 | (1739, 1868) |
| GAMMQ | 1138 | (1106, 1169) | 1081 | (1044, 1119) | 1220 | (1176, 1264) |
| GOLDEN | 1832 | (1781, 1883) | 1830 | (1766, 1894) | 1861 | (1794, 1928) |
| PLGNDR | 2082 | (2033, 2130) | 1807 | (1754, 1859) | 2742 | (2647, 2837) |
| PROBKS | 1801 | (1765, 1840) | 1443 | (1407, 1478) | 2635 | (2542, 2727) |
| SNCNDN | 636 | (618, 653) | 629 | (606, 651) | 636 | (613, 660) |
| TANH | 379 | (372, 387) | 307 | (299, 315) | 558 | (538, 577) |

Table 6.3: Mean F-measure and confidence intervals (CI) for scrambled quasi-random, ART, and random testing

In 9 out of the 12 test programs, the average F-measure was clearly much lower than that previously measured for random testing. However, the improvement was smaller than that reported for ART. In the remaining three cases, the programs "GAMMQ", "GOLDEN", and "SNCNDN", the improvement is insignificant, which was also the case for ART.

## 6.5 Discussion

In this chapter, we have shown that low-discrepancy sequences are a promising method for generating sequences of test cases. We have also shown that the use of "scrambling" allows the generation of many different sequences, allowing us to obtain useful estimates of the improvement over random testing on real programs, and allows different sequences to be tried - an important point for industrial use.

The preliminary nature of this investigation saw that only one of the low-discrepancy sequences described in the literature was tried. While it would be surprising if different low-discrepancy sequences produced much different overall results, it would be straightforward, and worthwhile, to check.

More importantly, the implementation of scrambling used for this preliminary study is unsatisfactory, as it requires the number of points in the sequence to be specified in advance, and then allocates and seeds the scrambler (with the permutations) based on the size. It should be possible to implement an incremental version that allocates and generates permutations as needed, but the implementation details will take some effort. Alternatively, there are a number of other scrambling techniques. They are considerably less sophisticated than Owen's technique, but they may be adequate for testing purposes; experimenting with this alternative techniques should be very straightforward, as implementations are already freely available.

Testing using quasi-random sequences seems particularly well-suited to cases where a very large number of tests are required, as the cost of generating $n$ tests is $O(n)$. The improvement in testing effectiveness is not as large as in FSCS-ART, so it does not make that method obsolete. It is also not nearly as flexible as FSCS-ART; if Euclidean distance between points is not a good indicator of similarity in failure behaviour, a different similarity measure can be chosen for FSCS-ART by the tester. Changing a low-discrepancy sequence to have some other property is likely to be just as difficult as coming up with the sequences in the first instance.

While I have not conducted experiments, it seems likely that the points generated by FSCS-ART would, in themselves, form a sequence with considerably lower discrepancy than random points. It would be interesting to quantify this, and examine whether some of our previous work on software testing is useful in other domains.

# Chapter 7

# Efficient Implementations of Adaptive Random Testing

## 7.1 Introduction

This chapter examines the third proposed method of reducing the selection overhead of ART. Here, rather than modifying the method, the existing method is implemented more efficiently. A more efficient data structure - the Voronoi diagram - is used to locate the closest existing test case to the current candidate. This is the most expensive operation in FSCS-ART, taking $O(n)$ time for the $n$th test case in a naive implementation. The improved version can reduce this to $\sqrt{n}$ (as implemented) or even $\lg n$ (possible, but considerably more complex to implement). This method is a considerable theoretical and not insubstantial practical improvement.

A number of ways to solve the "find-the-nearest" problem, which turns out to be a classic problem in computer science called the post-office problem, are examined. The Voronoi diagram, the standard solution, is then implemented and the performance compared empirically with the naive linear search approach. A significant theoretical and practical improvement is found. However, the Voronoi method is not without problems, particularly if the input domain has more than 3 or 4 dimensions. These issues are discussed extensively.

### 7.1.1 The post-office problem

If, as previously discussed, we assume it to be the case that the test can be represented in terms of a fixed-length vector, the "find the nearest" problem is essentially the simplest case of the "post-office problem" described by Knuth ([40], p. 555). Knuth describes the problem as follows:

> Suppose, for example, that we wish to handle queries like "What is the nearest city to point $x$", given the value of $x$.

At the time, Knuth noted that "no really nice data structures seem to be available" for this problem, and described a simple approach using a tree representation by McNutt

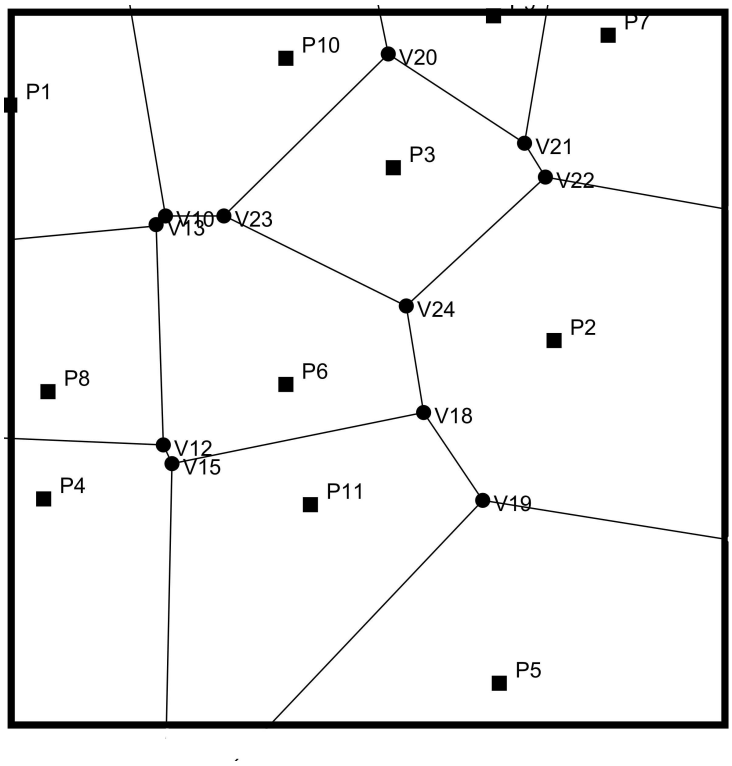



Figure 7.1: A Voronoi diagram for random points in the plane

and Pring. Shamos [62] was the first to identify that the *Voronoi diagram* was an efficient method for solving the post-office problem. The Voronoi diagram is a fundamental data structure in computational geometry. It has been developed independently in various areas of the natural sciences; see [4] for an extensive survey. The following description of the Voronoi diagram is from Bowyer [8]: Consider a set of distinct points $P = \{P_1 \dots P_m\}$ in the plane. For each point, we define the territory of that point as the area of the plane which is closer to that point than any other point in $P$. The set of resulting territories will form a pattern of convex packed polygons covering the whole plane. This definition trivially extends to arbitrary numbers of dimensions. Figure 7.1 shows a simple Voronoi diagram in the plane for a small number of random points. If all point pairs which share a boundary segment are joined, the result is a triangulation of the convex hull of the points, termed the *Delaunay triangulation*. The Voronoi and the Delaunay are duals, and one can very straightforwardly be computed from the other.

Given the existence of a Delaunay triangulation of such a point set, Green and Sibson [32] point out one simple way to locate the closest neighbour of a point is to simply perform a walk along the points of the triangulation. They further suggest a point close to the centroid of the point set would be a good place to start. For a point set in $\mathcal{R}^d$, such a walk should take $\mathrm{O}\left(n^{\frac{1}{d}}\right)$ time. There are several ways to post-process a Voronoi diagram to allow a post-office query to be answered in $\mathrm{O}(\log n)$ time.

Shamos [62] published an $\mathrm{O}(n \log n)$ algorithm for constructing a Voronoi diagram in $\mathcal{R}^2$. This algorithm uses a classical divide-and-conquer approach, sorting the points in one dimension, constructing a number of small Voronoi diagrams for subsets of points in

sections of the range and then describing a procedure for merging the resulting diagrams. This algorithm is totally unsuitable for the task at hand, however, as it does not allow for the incremental addition of points to the diagram.

The first incremental algorithm for constructing a Voronoi diagram was described by Green and Sibson [32]. To add a point $P_a$ to an existing diagram, they first identify the point $P_c$ nearest to the new point, then make local modifications to the diagram (a constant-time operation). They use the triangulation walk described above to identify $P_c$. Therefore, their version takes $O(\sqrt{n})$ time to add a point to the structure. To reduce this to $O(\log n)$ for random points, they suggest a simple modification - at regular intervals, duplicate the Delaunay triangulation, and after each duplication retain the older versions unmodified. Then, when performing the walk, start with the version containing the fewest nodes, and when the closest node in the earliest version is found, switch to searching the next version using the closest node found so far as the starting point. In Green and Sibson's tests, they found that this optimisation was unnecessary because the dominant cost was the modifications to the data structure. However, Green and Sibson were only interested in constructing the tree, rather than performing nearest-neighbour queries. For our application, there are many more queries than additions to the diagram.

Green and Sibson's algorithm only works for Voronoi diagrams in the plane. Bowyer [8] and Watson [67] were the first to describe algorithms suitable for use in $d \geq 3$ dimensions. Bowyer's algorithm, in particular, is a reasonably straightforward conceptual extension of the ideas used in Green and Sibson. Using it, it is possible to add the $n$th point to the diagram in $O\left(n^{\frac{1}{d}}\right)$ time, and find the nearest neighbour amongst $n$ points in the same order of magnitude time. Again, the non-constant cost comes from the walk through the points, so the same optimisation described for Green and Sibson's method should reduce the time complexity of searching or adding a random point to $O(\log n)$.

In this paper, we examine whether the improved asymptotic performance of the Voronoi-diagram based search over a naive linear search translates into a empirical reduction in ART overhead.

## 7.2 Method

### 7.2.1 Implementing ART using the Voronoi diagram

An implementation of Bowyer's algorithm is available in the freely downloadable Svlis solid modelling library [64]. Amongst many other functions, this library enables the incremental construction of a Voronoi diagram for points in $\mathcal{R}^3$, and the use of such for post-office queries. Svlis's implementation does not feature the optimised walk described in section 7.1.1, and was clearly not optimised for speed. Several simple implementation detail modifications (for instance, speeding up array initialisations in inner loops) were made and produced substantial linear speed improvements. The library is implemented in C++.

The naive linear search method was also implemented straightforwardly in C++, using the Svlis's point class and the C++ list implementation to store the data. Svlis's point manipulation and comparison functions were used here, also, to localise the differences

between the linear search and the Voronoi method.

All experiments were performed on a machine with an Intel Pentium III processor running at 733 MHz with 512 megabytes of RAM. It was the development snapshot of Debian GNU/Linux ([23] , including the Linux kernel version 2.4.18. The Gnu Compiler Collection[29], version 3.2.3, was used to compile both Svlis and both ART implementations. Timing measurements were performed by using the system's `time` command.

The candidate test cases were generated using the Mersenne Twister random number generator, as implemented in the GNU Scientific Library [33]. The same random number generator and seed were used for all experiments.

### 7.2.2 Experiment 1 - Effectiveness

As the linear search method and the Voronoi diagram method implement the same operation, the sequences of test cases produced by each should be identical. Preliminary testing showed this to indeed be the case.

### 7.2.3 Experiment 2 - Efficiency

In testing, the time taken to run automated tests is the sum of:

- the time taken to generate the tests,
- the time to execute the program using the generated tests as input, and
- the time to evaluate the results of testing.

Of these, only the generation time is affected by the use of the Voronoi method or the linear search method. Therefore, any comparison of the relative efficiency of the methods should be restricted to the time to generate sequences of test cases.

We therefore compared the execution time that our two implementations took to generate specified numbers of test cases. As the Voronoi diagram implementation available worked in $\mathcal{R}^3$ without modification, the test cases generated were vectors in $\mathcal{R}^3$. All elements in the vectors in the range $[0, 1)$.

Figure 7.2 shows the computation time taken to generate the specified number of test cases using the two methods, and normalised plots of $n^{\frac{4}{3}}$ and $n^2$ for comparison.

## 7.3 Discussion and Conclusion

As shown by Figure 7.2, the FSCS-ART implementation using the Voronoi diagram is clearly faster than the linear search implementation under the testing conditions given, both in terms of the absolute time taken and the asymptotic behaviour displayed. The time increase over the size range trialled was very close to that predicted by the theoretical asymptotic analysis. Even the suboptimal $\mathrm{O}\left(n^{1+\frac{1}{d}}\right)$ complexity of the implementation tested is enough of an improvement to make generating very large test sequences practical that would have been infeasible using a quadratic algorithm.

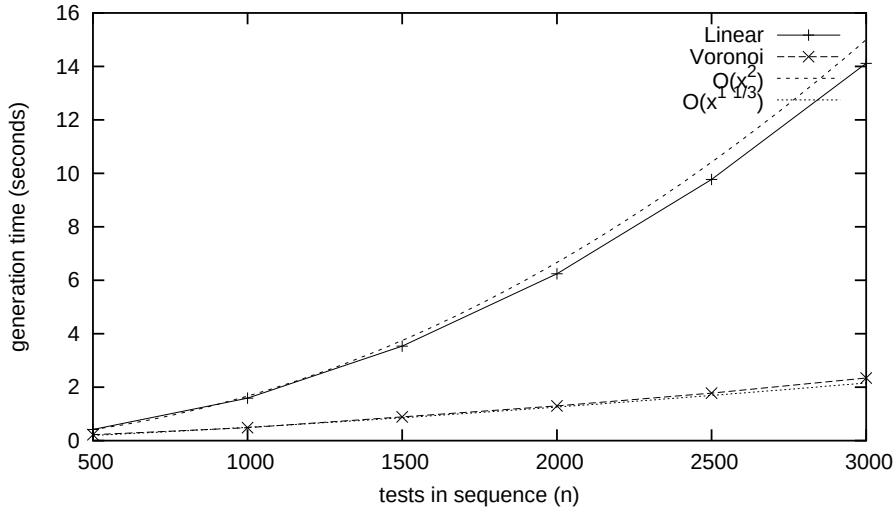



Figure 7.2: generation time for test sequences of length $n$ in $\mathcal{R}^3$

This study did not cover the effect of the number of dimensions of the test vectors on the the relative overheads of the method. The constant factors in the algorithm depend mainly on the number of vertices per point in the Voronoi diagram, which clearly increases swiftly in higher dimensions. In fact, the most expensive operation in the current implementation takes time proportional to the square of the number of vertices per node. Bowyer [8], states that for the two-dimensional case, the Euler-Poincare formula shows that the expected number of vertices per point is exactly six. Bowyer also quotes an upper bound from Miles for the three-dimensional case of 27.07, and notes that "Unfortunately there is, as yet, no general expression for the number in $k$ dimensions." Clearly, however, the effect of increased dimensionality will limit the range of problems to which the enhancements may be profitably applied.

As previously stated, the implementation tested here was not particularly optimised. As the overheads of test-case generation are still non-trivial, experimentation with the optimised Voronoi walk, and general tuning of the algorithm. The optimisation of the Voronoi walk would be particularly advantageous for sequences in $\mathcal{R}^2$, as the walk is actually more computationally expensive for those sequences compared with higher-dimensional ones. General tuning may increase the number of dimensions that a sequence may have, and decrease the minimum number of points in the sequence, for which the Voronoi method empirically outperforms the linear search method.

Bowyer's method is not the only method for generating the necessary Delaunay triangulations available. Watson [67] describes another equivalent method, and it would be worthwhile to compare the performance of the two in our application. Green and Sibson's method is actually recommended by Bowyer as more suitable for applications in $\mathcal{R}^2$. Additionally, there exists another major method for solving the post-office problem using a data structure termed a *Randomised Post Office tree* (usually abbreviated to simply RPO tree) [20]. Whilst having worse asymptotic performance than the Voronoi algorithms in construction of the data structure, Clarkson claims that for many distributions of points their performance is likely considerably better than the worst case bound. Clarkson's

algorithm, as described in the reference, is not designed for incremental insertion of new data, requiring the insertion of data in randomised order. However, in our application, the data will be inserted in an order than may be sufficiently close to random to enable good performance.

In any case, we have demonstrated that, with the aid of the Voronoi diagram data structure, FSCS ART can be used to generate sequences of test cases more efficiently than previously known. We recommend its use if the overhead of test case generation presently poses a barrier to applying ART.

# Chapter 8

# ART for programs with non-numeric input data

To apply most of the ART methods discussed in this thesis to a particular problem domain, there are two prerequisites. The first is a method to generate test cases randomly, with a uniform probability of selection, from within the input domain. The second is a useful similarity measure that can examine pairs of test cases and indicate whether one pair of test cases is more likely to exhibit the same failure behaviour, than another pair of test cases. For the case where the input form of the program under test is a fixed-length numerical vector, generating such a vector is trivial using random number generators provided in most programming environments. This thesis has demonstrated experimentally that the Euclidean distance is an effective measure of distance for this type of problem. For programs whose input takes different forms, however, finding methods for these two prerequisite tasks is considerably more difficult.

This chapter reviews some typical types of program input, and consider how they might be randomly generated, and some potential methods for calculating distance measures for these types of input.

## 8.1 String Generation

A very large class of programs take strings of text as input, so methods to apply ART to them would be of great interest. Generating "unstructured" text is very straightforward. However, trying to test most programs with unstructured text is not particularly productive, as a very large proportion of the generated tests will be "invalid" input and thus primarily test the handling of this type of input, which is likely to be a relatively small part of the functionality. [1] To test other aspects of program operation, it will be necessary to generate random test cases from the set of strings that represents the program's input domain, or, more likely, some subset of these. The standard method for specifying the structure for text to an input program are context-free grammars. It is therefore unsurprising that researchers have experimented with using this notation to specify the

[1] In fact, testing this type of programs in this manner is akin to composing sonnets by giving monkeys typewriters.

set from which test cases can be randomly drawn. Such approaches date at least back to Howden [38], who demonstrated an algorithm for randomly generating strings specified by a context-free language, and proposed its use for testing. The algorithm essentially involves the top-down rewriting of a context-free grammar, with the random replacement of nonterminals with their sentential forms.

Whilst this method does generate random strings that are words in the language specified, they are not "uniform". Consider a grammar with a single nonterminal $S$: $S \rightarrow aS|a$. Howden's method would generate the string $a$ with probability 0.5, the string $aa$ with probability 0.25, and the string $a^n$ with probability $1/2^n$. This is potentially a significant practical difficulty as well as a theoretical objection. This problem is mediated, to some degree, by the ability to annotate the grammar with nonuniform probability of selection, but the non-uniformity remains.

This approach was adopted by Slutz [63], who used it to generate statements in the SQL database query language [7]. Slutz provided extensive control over tree depths and other properties of the generated string by allowing extensive configurability of maximum tree depths, and probabilities of various types of SQL elements. As the tree derivation proceeded, a state variable recording the use of different elements was updated dynamically an considered whenever a choice in derivation path was considered. Configurability was used to change the emphasis of testing; he notes that the configuration file of the system "contains almost 100 items".

A method that can guarantee uniform sampling, in a sense, is by placing an additional restriction on the domain to be sampled from, making it finite by setting a maximum length. Given that length $l$, if the number of strings $n$ in the language are then determined, a unique mapping between the positive integers $\{1 \ldots n\}$ and the strings can be defined. Strings from this language can be randomly selected, with a uniform probability of such, by using a numerical random number generator to choose a number in the range $\{1 \ldots n\}$, and then using the string selected by the mapping. Merkel [50] developed algorithms for this purpose in his honours thesis. Unbeknownst to the author at the time, more efficient algorithms, with $\mathrm{O}(n \log n)$ time complexity for the mapping step, have been developed. [45]

In practice, for many types of testing it is likely that testers will need to be able to further control the set of tests to sample from. In most common programming languages for instance, the grammar does not completely specify the set of compilable programs [1](p. 179). For example, context-free languages are insufficiently powerful to specify the ANSI C language requirement that variables must be declared prior to use. Therefore, to express the types of test cases desired precisely enough, more expressive power might be required.

Howden's paper [38] describes one such extension, termed "not rules", which indicate that a nonterminal "cannot be" a member of the class defined by the right-hand side of the rule. However, this kind of restriction does not appear to be powerful enough to enforce rules of the type mentioned above, for instance. These kinds of limitations suggest that a generalised grammar-based approach may not be practical, and ad-hoc, programmatic,

customised approaches may be required. Yoshikawa *et. al* [74], when building a tool to generate random Java bytecode sequences to test Java Virtual Machines, used such a tailored approach.

## 8.2 Similarity Measures

In previous chapters, the use of the Euclidean distance measure has been justified in terms of "failure patterns", which represented likely patterns for the distribution of failure-causing inputs through the input domain of numerical programs. Two of the three patterns, the *strip* and *block* failure patterns, meant that the failure-causing inputs form a continuous region within the input domain. Therefore, if one test, $t_1$, has been conducted, and candidate $c_1$ is very close to $t_1$ (as measured by the Euclidean distance) and candidate $c_2$ is more distant, $c_1$ is likely to have the same failure behaviour as $t_1$ so if no failure was detected when testing $t_1$ it would be more beneficial to test $c_2$.

While the approach clearly works reasonably well for numerical programs, there is no obvious way to apply the method to the non-numeric input domain.

A naive approach to similarity measures is based on the existence of the ranking methods in the previous section. In essence, the distance between two members of the input domain is computed by finding their corresponding integer using the ranking algorithm, and computing the numerical difference. The author experimented with this approach in his honours thesis [50], and found the results to be poor, with little improvement on random selection in most cases.

In fact, this approach is fundamentally flawed, and in most cases no ranking algorithm could serve as a useful similarity measure. Consider a simple command language for, say, a robot, with three commands: "move", "turn", and "sit", each with some form of parameters. Given the wildly divergent nature of each action, any pair of inputs that are of the same class should presumably be treated as more similar than any pair from different classes. A ranking to implement this would therefore have all possible "move" commands consecutively ranked, all possible "turn" commands consecutively ranked, and all the "sit" commands consecutively ranked. However, our goal for the similarity measure will be violated at the boundaries between the strings for each command: for instance, assume the ranking method allocated the "turn" commands to the rankings $\{1 \dots k_1\}$, and "move" to $\{k_1 + 1, k_1 + k_2 + 1\}$. Therefore, the strings corresponding to rankings 1 and $k$ will have a distance of $k - 1$ despite having the same command, whilst the strings corresponding to rankings $k$ and $k+1$ will have a distance of 1 and thus be treated as much more similar. This appears to be a fundamental flaw with any such ranking approach. Zhao and Lyu [75] describe a distance measure for strings for testing. When stripped to its essentials, the process involves treating each ASCII character as a base-128 number and calculating distance between two strings by calculating differences between the numbers. In essence, therefore, their approach can be treated as a special case as the mapping approach described in this thesis, and suffers from all of the same problems as it.

However, we observe that the "failure patterns" are a consequence of a more fundamental phenomenon; as a heuristic, inputs that are close to each other using the Euclidean

distance tend to result in similar computation patterns. Using computation patterns as a heuristic for predicting failure behaviour dates from the earliest days of software engineering, and is the basis of most white-box testing technique. In fact, one very simple distance measure on this principle would be to compare inputs by executing them, and using a test coverage tool such as gcov [29] to record the percentage of statements the two inputs have in common, and use this as a similarity measure. However, if such a measure is used, a lower-overhead and probably more effective method would be to keep track of all statements covered by existing test cases and selecting a candidate based on the one that covers the most previously uncovered statements.

However, the overhead of evaluating a candidate is almost the same as doing the test in the first place. In fact, it could potentially be higher, as the instrumentation required to determine the coverage may cause the execution to be much slower than it would otherwise be. However, if the majority of testing cost is in evaluating the results of testing, then this method may be highly effective. This would however retain many of the problems of code coverage measures - the distance measure would not detect missing elements of the specification, for instance.

### 8.2.1 Grammar-based approaches

Where a string is generated by a context-free grammar, one obvious place to look for difference measures is the tree of productions used to derive the string from the start symbol, and the most straightforward distance metric is probably the maximum depth of the subtree rooted at the common ancestor of the two strings being compared. However, like the ranking methods, this method suffers from some fairly obvious drawbacks. Consider two strings. Consider a grammar with start symbol $S$ and the following productions:

$$S \rightarrow A|B \tag{8.1}$$

$$A \rightarrow a|aA \tag{8.2}$$

$$B \rightarrow b|bB \tag{8.3}$$

The strings $a$ and $b$ would have a distance of 1, whereas the strings $aaaaaaaaaa$ and $aaaaaaaaaaaaaaa$ would have a distance of 6, not reflecting common sense or likely failure behaviour of a program using strings in the language described by the above grammar as input.

A more useful approach might be to determine the productions applied to derive the strings, and compare the level of commonality; the simplest metric might be the number of productions in a candidate string not present in the existing string it is compared to. This, in essence, would be a "grammar coverage" approach. In common with the code coverage approach, rather than comparing candidates with particular existing test cases, it may sometimes be more effective, and would certainly be more efficient, to simply rank candidates by the number of thus-far untested productions they contain. However, such a method would quickly fail to be of use if a large number of test cases was desired, as the

entire grammar could be covered by a number of test cases not larger than the number of productions in the grammar.

### 8.2.2 Record data structures - syntax based approaches

While strings are an important data format to consider for applying ART methods to, they are by no means the only one of interest. Equally of interest are aggregate data types, which allow the grouping of a number of different variables of different types into a single variable. One example of an aggregate data type system is C's `struct`. A `struct` declaration gives the new data type a name, and lists the component members of the new data type, indicating their data type and giving each member a distinctive name so that they can be accessed. Figure 8.1 shows an example of a `struct`.

As an aggregation of variables, distance measures for struct variables can be calculated simply by treating each member of the struct as distinct variables and calculating a total distance using the same Cartesian methods as with distinct variables. This procedure is not suitable for all `structs`, however. An example of a `struct` that is unsuitable can be seen in Figure 8.2. This structure, which would be used to implement the linked list data type, is an example of a self-referential data structure, containing pointers to data structure as the same type as itself.

If distance measures are unavailable for a particular `struct` member, it is still possible to calculate a distance measure based on the other member of the struct.

```
struct foo
{
  int bar;
  double baz;
  struct burfle qux;
  char *quux
};
```

Figure 8.1: A C data structure

```
typedef struct lnode *Lnode;
typedef int data;

struct lnode {
  data dt;
  Lnode next;
};
```

Figure 8.2: A C data structure for a linked list

### 8.2.3 Enumerated types

Another data type in common use are enumerated data types like C's `enum` types. In C's version, users can create new `enum` types. When a new `enum` is declared, users define a

discrete number of symbolic values. The declaration may also specify a numerical value for each symbolic value, if this is not defined they are assigned numerical values according to the order in which they are declared. The numerical value is used when an enumeration is cast to an numeric type, or when relational operators are used to compare two members of that type.

Generating values of an enumerated type is clearly trivial with the use of a pseudorandom number generator, but appropriate distance measures may need a little more care. Obviously, a distance measure can be computed for enumerated types by simply converting them to a numerical value. However, that may not always be appropriate, as it implies that some pairs of values are more similar to each other than others pairs. If the enumeration specifies categorical information, the only sensible distance measure might be that identical values have a distance of 0, and all non-identical pairs have some pre-specified value.

It is quite feasible to define a complete distance function over all pairs of values in an enumeration using a lookup table. For an enumeration with $n$ distinct values, the table will need only $\frac{n^2}{2}$ entries.

One point to consider when dealing with enumerations is what kind of scaling factor should be applied to the distance measure.

### 8.2.4 Arrays and lists

Sequential data structures like arrays and lists pose a considerable challenge when trying to generate distance measures. It is doubtful, in fact, that *general* guidelines for the purpose can be drawn up, as there are so many different ways as to how the different arrangements of members of the list could affect the running of the program. As with strings, code coverage measurements could well be used for this purpose, particularly path coverage measures. The question then arises whether other methods, such as genetic algorithms, would be a more effective way of achieving path coverage than an ART-style approach.

This type of data, with so little syntax to analyse, is unlikely to be amenable to a purely syntactic approach. More generally, it seems from this quick survey of such methods that they are likely to be severely limited in applicability.

## 8.3 A semantic approach: adapting the category-choice method

Aside from the code coverage approach, the methods proposed above are tied quite intimately with the structure of the input data. While this makes them relatively easy to implement where applicable, they are rather inflexible and ill-suited to many programs. So, clearly, a more powerful framework is required. For this, we return to our key observation in the application of code coverage measures: the Euclidean distance measure is effective *because it reflects the fact that numerically similar inputs tend to result in similar computation.* Generally, the program's specification will provide strong indications of how different types of input are to be handled. Therefore, it should be possible for programmers to use the specification to come up with a similarity measure.

We propose to adapt the category-choice method for just this purpose [15]. The category-choice method is a specification-based testing method that allows the tester to specify a number of "categories", a major aspect of the input parameters or operating environment. The set of possible values in a category is divided into disjoint sets known as choices. Choices are such defined on the intuition that elements of different choices are to be processed differently. Hence, choices form a natural basis to provide measurement for "similarity" or "difference" between various input values.

Let us denote the set of categories by $C = \{C_1, \ldots, C_k\}$. For each $C_i$, let us denote its choices by $p_1^i, \ldots, p_{i_k}^i$.

It should be noted that any input is a combination of input values chosen from a non-empty subset of $C$. For input $x$, let us denote the corresponding non-empty subset by $\mathrm{C}(x) = \{C_{x1}, \ldots, C_{xk}\}$. Since categories are distinct and their choices are disjoint, input x in fact consists of values chosen from a non-empty subsets of choices, denoted as $\mathrm{P}(x) = \left\{p_{x1_i}^{x1}, \ldots, p_{xk_i}^{xk}\right\}$.

For any 2 inputs $x$ and $y$, we can define $\mathrm{DP}(x, y)$ as the set that contains elements in either $\mathrm{P}(x)$ or $\mathrm{P}(y)$ but not in both. That is, $\mathrm{DP}(x, y) = \mathrm{P}(x) \bigcup P(y) \backslash (\mathrm{P}(x) \bigcap \mathrm{P}(y))$. Now, we define $\mathrm{DC}(x, y) = \left\{C_m | C_i \text{if} \exists p_j^i \text{ in } \mathrm{DP}(x, y)\right\}$. Then, the difference measure between x and y is defined as the size of $\mathrm{DC}(x, y)$. Intuitively speaking, this difference measure is a measure of the difference of choices from which the values of $x$ and $y$ are derived.

## 8.4 Example: International postal rate calculator

In this section, we demonstrate how the category-partition approach can be used to devise a similarity measure to use adaptive random testing on a hypothetical international postal rate calculator.

The input to this postal rate calculator is a quintuple I = {country, weight, delivery, value, confirmation} where:

**country** : the destination country. Countries are divided into five zones.

**weight** : the mass of the parcel, up to a specified maximum weight.

**delivery system** : The means by which the parcel is to be delivered; there are five categorical types: air, economy air, sea, EMS document, and EMS merchandise.

**value** : the insurance value of the item, up to a specified maximum. Insurance costs $5 + $2 per $100 of value.

**registration** : whether the mail is unregistered, registered, or registered with a delivery confirmation service.

The base charge depends on the delivery system, weight, and zone, according to table 8.1:

### 8.4.1 Categories and partitions

We now describe a set of categories and partitions for this pricing function:

| Zone | Weight | Air | Econ | Sea | EMS (doc) | EMS (mer) |
|---|---|---|---|---|---|---|
| 1 | < 250g | 5.50 | 5.00 | - | 28.00 | 35.00 |
| | 250-500g | 8.50 | 7.50 | - | 28.00 | 35.00 |
| | 500-750g | 11.50 | 10.00 | - | 32.00 | 39.00 |
| | 750g-1kg | 14.50 | 12.50 | - | 32.00 | 39.00 |
| | Each extra 500 g | 3.00 | 2.50 | | 4.00 | 4.00 |
| 2,3,4 | < 100 g | 6.50 | 6.00 | - | 30.00 | 37.00 |
| | 100 - 250g | 10.50 | 9.00 | - | 30.00 | 37.00 |
| | 250- 500g | 16.50 | 13.00 | - | 38.00 | 43.00 |
| | 500-750g | 20.50 | 17.00 | - | 38.00 | 43.00 |
| | Each extra 250 g | 5.00 | 4.00 | - | 8.00 | 6.00 |

Table 8.1: Excerpt of pricing table for postal rate calculator application

**Categories**: Countries, Delivery System, Registration, Weights-for-Zone-1, Weights-for-zones-234, value.

**Partitions for category "Countries"**:

Zone-1 = {NZ}

Zone-2(Asian Countries) = {China, Japan, Korea, ...}

Zone-3(European Countries) = {UK, Germany, France, Spain, ...}

Zone-4(American Countries) = {USA, Canada, Brazil, Argentina, ...}

**Partitions for category "Delivery System"**

Air = {air}

Economy = {economy}

Surface = {sea, train}

EMS = document, merchandise

**Partitions for category Registration:**

No = {not registered}

Registered = {Registered without delivery confirmation}

Confirm = {Registration with delivery confirmation}

**Partitions for Weights-for-Zone-1:**

Zone-1-weight-1 = { $\leq$250g}

Zone-1-weight-2 = {250g $< x \leq$ 500g}

Zone-1-weight-3 = {500g $< x \leq$ 1kg}

Zone-1-weight-4 = {>1kg}

**Partitions for category Weights-for-Zones-234:**

Zone-2-weight-1 = {$\leq$ 100g}

Zone-2-weight-2 = {100g $< x \leq$ 250g}

Zone-2-weight-3 = {250g $< x \leq$500g }

Zone-2-weight-4 = {500g $< x \leq$750 g}

Zone-2-weight-5 = {750 g $< x \leq$ 1kg }

Zone-2-weight-6 = {> 1kg }

**Partitions for category value:**

specified = {a value was specified and insurance claimed}

unspecified = {no insurance claimed, so no value set}

**Difference measurement examples**

Assume we are deriving a test set for the postal rate article, and one test, $t_1$ has already been chosen with the following characteristics: country = "New Zealand", delivery system = "Economy", registration="Registered", weight = 150 grams, placing it in the Zone-1-weight-1 partition of the Weights-for-zone-1 category.

We then consider two candidate test cases: $c_1$ and $c_2$. For $c_1$, country = "New Zealand", delivery system = "EMS document", registration = "no", weight = 125 grams. For $c_2$, country = "USA", delivery system = "Economy", registration = "confirm", value = \$50, weight = 150 grams.

To calculate the differences, we need to first need to determine P($t_1$), P($c_1$), and P($c_2$):

P($t_1$) = {country = "Zone 1",
delivery system = "Economy",
registration = "confirm",
Weights-for- Zone-1 = "Zone-1-weight-1" }

P($c_1$) = {country = "Zone 1",
delivery system = "EMS document",
registration = "no",
weights-for-zone-1 = "Zone-1-weight-1" }

P($c_2$) = {country = "Zone 2",
delivery system = "Economy",
registration = "confirm",
value = \$50,
weights-for-zone-2 = "Zone-2-weight-1" }

From this, we calculate DP($t_1, c_1$) and DP($t_1, c_2$). DP($t_1, c_1$) = {delivery system = "Economy", delivery system = "EMS document", registration = "no", registration = "confirm"}. Therefore, DC($t_1, c_1$) = {registration, delivery system} and $|\text{DC}(t_1, c_1)| = 2$. DP($t_1, c_2$) = { country = "Zone 1", country = "Zone 2", value = \$50, weights-for-zone-1="Zone-1-weight-1", weights-for-zone-2 = "Zone-2-weight-1"}. Therefore DC($t_1, c_2$) = {country, value, weights-for-zone-1, weights-for-zone-2}. $|\text{DC}(t_1, c2)| = 4$. Therefore, according to our distance measure $c_2$ is more different to $t_1$ than $c_1$ is, and would be chosen for testing in the ART paradigm.

### 8.4.2 Example: "ls" system command

In a Unix system shell, file listings are provided by the system `ls` command. Over the years, considerable flexibility as to which files are listed, and what information about the files is provided, has been made available through command-line options. Specifying all the possible options of the modern `ls` in terms of the category-partition method impracticable here for space reasons: we present a characterisation of a subset of its functionality:

**Categories:**

**sorting** the sort criterion for file listing; there are a number of different criteria.

**reverse** sort order

**view-hidden** whether hidden files should be displayed

**columns** how the data should be displayed in columns

**screenwidth** the width of the screen on which the data is to be displayed.

**Partitions for category "sorting":**

| | |
|---|---|
| by-name: | sort by name |
| by-extension: | sort by file extension |
| by-size: | sort by size |
| by-time: | by modification time |
| by-none: | do not sort |

**Partitions for category "reverse"**

| | |
|---|---|
| original | |
| reversed | |

**Partitions for category "view-hidden"**

| | |
|---|---|
| almost-all | display most hidden files |
| all | display everything |

**Partitions for category "format"**

| | |
|---|---|
| onecolumn: | 1 column |
| colsep: | separate files by commas |

**Partitions for category "screenwidth"**

| | | |
|---|---|---|
| screenwidth-1 | = | $\{\leq 40\}$ |
| screenwidth-2 | = | $\{40 < x \leq 80\}$ |
| screenwidth-3 | = | $\{80 < x \leq 132\}$ |
| screenwidth-4 | = | $\{> 132\}$ |

### 8.4.3 Difference measures

We again consider a single test case $t_1$ and candidates $c_1$ and $c_2$. Each test case is a shell command line, as follows:

| | | |
|---|---|---|
| $t_1$ | = | "ls –sort=size -r -a –screenwidth=40" |
| $c_1$ | = | "ls –sort=size -1" |
| $c_2$ | = | "ls –sort=extension -r -A –screenwidth=80" |

Again, we determine P($t_1$), P($c_1$), and P($c_2$):

$$
\begin{aligned}
\mathrm{P}(t_1) &= \{\text{sorting} = \text{“by-size”},\\
&\quad \text{reverse} = \text{“reversed”},\\
&\quad \text{view-hidden} = \text{“almost-all”}\\
&\quad \text{screenwidth} = \text{“screenwidth-1”}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{P}(c_1) &= \{\text{sorting} = \text{“by-size”},\\
&\quad \text{format} = \text{“onecolumn”}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{P}(c_2) &= \{\text{sorting} = \text{“by-extension”},\\
&\quad \text{reverse} = \text{“reversed”},\\
&\quad \text{view-hidden} = \text{“all”},\\
&\quad \text{screenwidth} = \text{“screenwidth-2”}\ \}
\end{aligned}
$$

We then calculate $|\mathrm{DC}(t_1, c_1)|$ and $|\mathrm{DC}(t_1, c_2)|$:

$$
\begin{aligned}
\mathrm{DP}(t_1, c_1) &= \{\text{reverse} = \text{“reversed”},\\
&\quad \text{view-hidden} = \text{“almost-all”},\\
&\quad \text{screenwidth} = \text{“screenwidth-1”}\\
&\quad \text{format} = \text{“onecolumn”}\ \}\\
\mathrm{DC}(t_1, c_1) &= \{\text{reverse, view-hidden, screenwidth, format}\}\\
|\mathrm{DC}(t_1, c_1)| &= 4
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{DP}(t_1, c_2) &= \{\text{sorting= “by-size”},\\
&\quad \text{sorting} = \text{“by-extension”},\\
&\quad \text{view-hidden} = \text{“almost-all”},\\
&\quad \text{view-hidden} = \text{“all”},\\
&\quad \text{screenwidth} = \text{“screenwidth-1”},\\
&\quad \text{screenwidth} = \text{“screenwidth-2”}\}\\
\mathrm{DP}(t_1, c_2) &= \{\text{sorting, view-hidden, screenwidth}\}\\
|\mathrm{D}(t_1, c_q)| &= 3
\end{aligned}
$$

$c_1$ is more different and thus would be preferred over $c_2$.

This work is obviously preliminary and we would certainly not claim that the concepts presented here lead immediately to the straightforward application of ART to all programs where it has not previously been applicable. It does however demonstrate that ART techniques can indeed be applied to at least some programs with non-numeric input, and will hopefully act as a starting-off point for further investigations in the area.

Our adaptation of the category-partition method is far more powerful and more broadly applicable than the original version. Firstly, our proposal is designed around the concept of full automation; previous methods have involved at best the partial automation of test case generation. Additionally, the original category-partition method proposed the exhaustive testing of all feasible category combinations, thus rendering testing infeasible if the number of categories and partitions is too large. In our method, as many categories and partitions as is natural for the problem domain can be specified, and the use of ART with the derived distance measure will ensure that the most different possibilities are explored first.

At this point, while intuitively we expect our method to improve error-detection performance, we have not collected any experimental data to support that. Obviously, such empirical data is a high priority. The second example, using the "ls" command, should be able to be modified to test an actual existing implementation, appropriately seeded with bugs.

There are any number of extensions to the framework which may allow for a more accurate difference heuristic. For instance, differences in some categories seem more likely than others to reveal different failure behaviour. Therefore, it would seem reasonable to extend the difference measure to allow variable weightings to be attached to certain categories, so that if partition elements from these categories appear in $\mathrm{DP}(x, y)$, the difference can be counted as greater or lesser than a simple numerical count we have proposed. Similarly, if two elements from the same category appear in $\mathrm{DP}(x, y)$ it may be that they represent some kind of categorical data; some categorical pairings may be more different than others, and this could be reflected in some kind of weighting factor also.

## 8.5 Discussion

In this chapter, we have examined how ART techniques might be applied to programs whose input takes forms other than the simple fixed-dimension vectors used in previous experiments. There are two requirements for ART to be applied to test a program: the ability to randomly generate candidate inputs, and the ability to look at two inputs and estimate the relative likelihood of common failure behaviour.

As is clear from the relatively brief survey here, both problems are quite challenging. This is not entirely surprising. In general, the problem of whether two inputs have the same failure behaviour is undecidable[2], and research into partitioning methods, for instance, has examined the problem of identifying areas of homogeneous failure behaviour for many years. This should not be seen, however, as casting doubt on the utility of ART for a wider class of problems. Instead, ART can be used as a better, more versatile, more efficient, and more powerful way to apply existing heuristics.

With our adaptation of the category-choice method, we have shown a concrete example of this - an existing test case selection heuristic made much more powerful through the ART methodology. We have, by combining category-choice with ART, solved the complexity explosion that limits the category-choice method, and opened up the future to much more subtle use of the category-choice method through differential weightings. We believe that this is only a beginning, and that other test case selection heuristics will also prove suitable for integration with ART.

It is worth noting that methods of applying ART to non-numeric problems tend to be most readily applicable to the earlier distance-based ART methods, rather than the lower-overhead of ART by bisection or by random partitioning as described in chapter 5, or the quasi-random approach in chapter 6. The counterexample in section 8.2 will probably make adaptation of these methods, which rely on a continuous mapping of the

[2]It's not hard to show that a method to do this would also be able to decide the halting problem

input space, somewhat difficult.

It is also clear that the random generation of members of a program's input space will pose problems. In some respects, the difficulty is really that for many programs, their input space is not really clearly defined, with many implicit assumptions present in textual descriptions. Due to these problems, it seems that automated generation based purely on the program's specification will often not be possible, and testers will have to build customised generation tools.

# Chapter 9

# Conclusion

We are confident that this thesis has significantly advanced the state of adaptive random testing. It has made contributions in several, quite different areas:

- Proved mathematical bounds of the potential improvement in failure detection effectiveness of ART, or indeed any method that relies on assumptions about the shapes of failure patterns, rather their likely location.
- Conducted an investigation of further potential improvements in ART effectiveness likely from targetting specific failure pattern shapes, such as strips.
- Provided a much more extensive statistical characterization of the behaviour of ART algorithms, in terms of both the distribution of tests within the input domain, and the sampling distribution of the commonly-used effectiveness measures.
- Described two new ART methods that use the concept of dynamic partitioning to achieve the even spreading of test cases. One method, bisection-ART, has shown an interesting connection between the ART concept and the proportional sampling strategy (PSS) for partition testing.
- Described the use of low-discrepancy sequences as a testing method, offering a very low-overhead way of spreading test cases evenly through the input domain.
- Demonstrated how the existing FSCS-ART algorithm can be implemented much more efficiently using the Voronoi diagram data structure.
- Investigated how ART can be applied to programs which take non-numeric input, and described heuristics ehow to use the notion of category-partition as the distance measure required for ART to be effective.

Of these contributions, the most significant is probably the proof of the effectiveness bound in chapter 2. This proof has showed that, at least for block failure patterns, existing ART methods are close to optimally effective in detecting failures. This shows that further experimentation into improving ART's effectiveness is not going to provide big improvements in effectiveness. Effectiveness improvements are only likely to come

from methods that use additional information about failure behaviour. Such methods would, in essence, make use of knowledge about what parts of the input domain are more likely to contain failures, rather than just the knowledge that failure patterns are likely to be contiguous. Improvements in effectiveness might also come from a testing method specifically designed to look for strip failure patterns, as indicated by the experiments in Chapter 4.

More than that, however, it shows a relatively simple mathematical analysis can provide a useful insight into how effective a testing process - *any possible* testing process - can be. This kind of result is the software testing analogue to computational complexity theory. Results in complexity theory have been enormously helpful to algorithm designers; the bound in this thesis has been very useful in setting the direction of the subsequent research presented and will likely continue to guide ART research.

The development of a number of new ART methods was a major focus of this thesis. Probably the method of most immediate practical interest was ART by bisection, presented in chapter 5. Simple to implement, with far lower overheads than FSCS-ART, it also has a neat theoretical advantage. Its probability of detecting at least one failure guaranteed to be, at a minimum, as good as random testing due to it being an incremental implementation of the "equal-size-equal-number" partition testing strategy. ART by random partitioning does not have this advantage, and empirical analysis showed it to be slightly less effective. The other methods presented are promising targets for further work, but they are more difficult to apply. The use of quasi-random sequences looks highly promising, particularly in the case where very large test sets are required. However, the implementation of an incremental scrambled quasi-random sequence generator, while clearly possible, looks to be a considerable programming task. The Voronoi diagram-based implementation of FSCS-ART, to be practical, requires a faster and more flexible Voronoi diagram implementation, which, again, is a nontrivial programming effort. It is also only really useful for problems where the number of dimensions is small.

The statistical analysis in chapter 3, shows that statistics aren't going to make life easy for for testing practitioner, even less the testing researcher! The very high variance of the F-measure, P-measure, and E-measure indicate that results from testing can vary greatly from run to run, leaving the practitioner needing to do many more tests to ensure failure detection than they might assume. For the researcher, our results indicate that a very large number of experimental trials will be required to accurately characterise the effectiveness of testing methods in typical conditions. However, all is not lost. The simulations show that ART methods not only have better average-case performance than random methods, they have better worst-case performance as well. The chances of a large number of tests failing to detect an error is much lower with ART than with random testing - a powerful argument for their adoption.

Finally, chapter 8 tackles a subject of pivotal importance for the wide adoption of ART - how to apply it to real programs whose inputs are not trivially modelled as a fixed-length numerical vector. While the methods presented in this chapter are promising and workable, it is obviously compromised by its preliminary nature. Experimental validation

for these ideas is essential; case studies to provide validating evidence is easily the work of a thesis on its own.

It is hard to overstate the importance of work in this area if ART is to be of future research interest. Software engineering is a practical discipline; ART, up to and including this thesis, has largely been a theoretical undertaking. It is time for it to be "taken out of the laboratory", so to speak, and show that it can be applied to a broader range of programs. We are confident that this can indeed be done.

More broadly though, while software testing may well be a practical discipline, a rigorous treatment of it as a research area requires a better theoretical underpinning. The general approach used to obtain the lower bound may well be extensible to provide more results about the potential maximum effectiveness of testing - not only in terms of failures detected, but in terms of actual software reliability improvement. As an example of the kinds of questions that might be examined, an improved model might allow the study of the effects of providing probabilistic information about failure locations. Through this kind of work, the performance of many existing testing methods may be able to be compared with theoretical bounds, and potential targets for improvement identified. Better analysis may also allow testers to calculate the likely number of failures detected, and the reliability improvement gained, by a particular testing plan.

Sir Maurice Wilkes, the computer pioneer and source of the famous quote in the introduction to this thesis, is enjoying a comfortable retirement[1] as his successors still labour under the burden he first articulated - that is, finding mistakes in computer software. While that burden will never be completely lifted as long as software continues to be written, it is hoped that the ideas in this thesis are another small step along the road to easing it.

[1]To the best of the author's knowledge - his Cambridge webpage is updated occasionally with new writings

# Bibliography


[1] A. V. Aho, R. Seti, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in Computer Science. Addison-Wesley, 1986.

[2] P. E. Ammann and J. C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, April 1988.

[3] Association for Computing Machinery, *Collected Algorithms from ACM, Vol I, II, III.* Association for Computing Machinery, 1980.

[4] F. Aurenhammer, "Voronoi diagrams – a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.

[5] P. G. Bishop, "The variation of software survival times for different operational input profiles," in *FTSC-23. Digitest of Papers, the Twenty-Third International Symposium on Fault-Tolerant Computing.* IEEE Computer Society Press, June 1993, pp. 98–107. [Online]. Available: http://www.adelard.co.uk/resources/papers/pdf/ftsc23.pdf

[6] R. Bowles, *The Last of the First; CSIRAC: Australia's First Computer.* Department of Computer Science, University of Melbourne, 2000, ch. 10, pp. 43–53.

[7] J. S. Bowman, S. L. Emerson, and M. Darnovsky, *The Practical SQL Handbook*, 3rd ed. Reading, Massachusetts: Addison-Wesley, 1996.

[8] A. Bowyer, "Computing dirichlet tesselations," *The Computer Journal*, vol. 24, pp. 162–166, 1981.

[9] P. Bratley, B. L. Fox, and H. Niederreiter, "Implementation and tests of low-discrepancy sequences," *ACM Transactions on Modeling and Computer Simulation*, vol. 2, no. 3, pp. 195–213, July 1992.

[10] F. Chan, T. Chen, I. Mak, and Y. Yu, "Proportional sampling strategy:guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, 1996.

[11] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Proceedings of the 7th European Conference on Software Quality*, ser. Lecture Notes in Computer Science, vol. 2349, 2002, pp. 321–330.

[12] T. Y. Chen, G. Eddy, R. G. Merkel, and P. K. Wong, "Adaptive random testing through dynamic partitioning," in *Proceedings of the Fourth International Conference*

*on Quality Software (QSIC 2004)*. Braunschweig, Germany: IEEE, September 2004, pp. 79–86.

[13] T. Y. Chen, F.-C. Kuo, and R. Merkel, "On the statistical properties of the f-measure," in *Proceedings of the Fourth International Conference on Quality Software (QSIC 2004)*. Braunschweig, Germany: IEEE, September 2004, pp. 146–153.

[14] T. Y. Chen, F.-C. Kuo, R. Merkel, and S. Ng, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, pp. 1001–1010, 2004.

[15] T. Y. Chen, F.-C. Kuo, R. Merkel, and T. H. Tse, "Adaptive random testing for programs with non-numeric inputs," in preparation.

[16] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proceedings of the 9th Asian Computing Science Conference*, ser. Lecture Notes in Computer Science, vol. 3321, 2004, pp. 320–329.

[17] T. Y. Chen, T. H. Tse, and Y. T. Yu, "Proportional sampling strategy: A compendium and some insights," *Journal of Systems and Software*, vol. 58, no. 1, pp. 65–81, 2001.

[18] T. Y. Chen and Y. T. Yu, "On the relationship between partition and random testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 977–980, December 1994.

[19] ——, "On the expected number of failure detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, 1996.

[20] K. L. Clarkson, "A randomized algorithm for closest-point queries," *SIAM Journal on Computing*, pp. 830–847, 1988.

[21] *The Computing and Data Processing Society of Canada, Third Conference*, vol. 3. University of Toronto Press, 1962.

[22] T. A. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, Massachusetts, USA: The MIT Press, 2001, pp. 382–384.

[23] "Debian GNU/Linux," website, August 2003, a product of the Debian project. [Online]. Available: http://www.debian.org

[24] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing failure: the distribution of program failures in a profile space," in *ESEC / SIGSOFT FSE*, September 2001, pp. 246–255.

[25] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 438–444, July 1978.

[26] ——, "A report on random testing," in *Proceedings of the 5th International Conference on Software Engineering*, 1981, pp. 179–183.

[27] J. W. Duran and J. J. Wiorkowski, "Quantifying software reliability by sampling," *IEEE Transactions on Reliability*, vol. 29, pp. 141–144, 1980.

[28] I. Friedel and A. Keller, "Fast generation of randomized low-discrepancy point sequences," in *Monte Carlo and Quasi-Monte Carlo Methods 2000*, K. T. Fang, F. J. Hickernell, and H. Niederreiter, Eds. Springer-Verlag, 2002, pp. 257–273.

[29] "Gnu compiler collection," website, August 2003, a product of the GNU Project. [Online]. Available: http://gcc.gnu.org

[30] S. Gill, "The diagnosis of mistakes in programmes on the edsac," *Proceedings of the Royal Society of London, Series A - Mathematical and Physical Science*, vol. 206, no. 1087, pp. 538–551, 1951.

[31] E. Girard and J. Rault, "A programming technique for software reliability," in *IEEE Symposium on Computer Software Reliability.* IEEE, 4 1973, pp. 44–50, iEEE Catalog No. 73 CH0741-9 CSR.

[32] P. Green and R. Sibson, "Computing dirichlet tesselations in the plane," *The Computer Journal*, vol. 21, pp. 168–173, 1978.

[33] (2004, July) The GNU Scientific Library. Produced by the GNU Project. [Online]. Available: http://www.gnu.org/software/gsl

[34] (2004, July) The GNU Scientific Library Manual. A product of the GSL team, part of the GNU project. [Online]. Available: http://www.gnu.org/software/gsl/manual/gsl-ref_toc.html

[35] W. J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, 1999.

[36] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, no. 4, pp. 242–257, 1970.

[37] J. B. Heard, "The adequacy and efficiency of program testing," in *The Computing and Data Processing Society of Canada, Third Conference*, vol. 3. University of Toronto Press, 1962.

[38] W. E. Howden, "Functional program testing," *IEEE Transactions on Software Engineering*, vol. 6, pp. 162–169, 1980.

[39] D. E. Knuth, *Seminumerical Algorithms*, 1st ed., ser. The Art of Computer Programming. Addison-Wesley, 1973, vol. two.

[40] ——, *Sorting and Searching*, 1st ed., ser. The Art of Computer Programming. Addison-Wesley, 1973, vol. three.

[41] T. Kollig and A. Keller. (2004, July) Samplepack. [Online]. Available: http://www.uni-kl.de/AG-Heinrich/SamplePack.html

[42] F.-C. Kuo and D. Towey, personal communication, 2003.

[43] P. L'Ecuyer, "Random numbers for simulation," *Communications of the ACM*, October 1990.

[44] I. K. Mak, "On the effectiveness of random testing," Master's thesis, University of Melbourne, 1997.

[45] E. Makinen, "Ranking and unranking left szilard languages," University of Tampere, Tech. Rep. A-1997-2, 1997, department of Computer Science Series of publications A. [Online]. Available: http://www.cs.uta.fi/reports/pdf/A-1997-2.pdf

[46] G. Marsaglia, "Random numbers fall mainly in the plains," *Proceedings of the National Academy of Sciences*, vol. 61, pp. 25–28, 1968.

[47] M. Matsumoto and T. Nishimura, "Mersenne tiwster: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1 1998.

[48] D. McCann and P. Thorne, Eds., *The Last of the First; CSIRAC: Australia's First Computer*. Department of Computer Science, University of Melbourne, 2000.

[49] W. Mendenhall and T. Sincich, *Statistics for the Engineering and Computer Sciences*, 2nd ed. San Francisco, California, USA: Dellen Publishing Company, 1988.

[50] R. Merkel, "Generating test strings described by regular expressions," April 1999, honours Thesis, University of Melbourne.

[51] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of computer programs," *Communications of the ACM*, vol. 6, no. 2, pp. 58–63, 1963.

[52] F. Mosteller, R. E. K. Rourke, and G. B. Thomas, *Probability with Statistical Applications*, 2nd ed. Addison-Wesley, 1973.

[53] G. J. Myers, *The Art of Software Testing*. John Wiley and Sons, Inc, 1979, iSBN 0-471-04328-1.

[54] National Institute of Standards and Technology, "Quantile-quantile plot," In NIST/SEMATECH e-handbook of statisticfs methods. [Online]. Available: http://www.itl.nist.gov/div898/handbook/eda/section3/qqplot.htm

[55] H. Niederreiter, "Low discrepancy and low-dispersion sequences," *Journal of Number Theory*, vol. 30, pp. 51–70, 1988.

[56] A. B. Owen, "Randomly permuted (t, m, s)-nets and (t, s)-sequences," in *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, ser. Lecture Notes in Statistics, H. Niederreiter and P.-S. Shiue, Eds., vol. 127. Springer-Verlag, 1995.

[57] ——. (2004, July) Scrambled nets for accurate multivariate inegration. [Online]. Available: http://www-stat.stanford.edu/~owen/scramblednets

[58] S. K. Park and K. W. Miller, "Random number generators:good ones are hard to find," *Communications of the ACM*, vol. 31, no. 10, pp. 1192–1201, October 1988.

[59] W. H. Press, *Numerical Recipes in FORTRAN: the art of scientiric computing*, 2nd ed. Cambridge University Press, 1992.

[60] G. F. Renfer, "Automatic program testing," in *The Computing and Data Processing Society of Canada, Third Conference*, vol. 3. University of Toronto Press, 1962.

[61] L. R. L. Sauder, "A general test data generator for cobol," in *Proceedings of the AFIPS Spring Joint Computer Conference*, vol. 21. American Federation of Information Processing Societies, May 1962, pp. 317–323.

[62] M. I. Shamos, "Geometric complexity," in *Proceedings of the 7th annual ACM Symposium on STOC*, 1975, pp. 224–233.

[63] D. Slutz, "Massive stochastic testing of SQL," in *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB98)*, August 1998, pp. 618–622. [Online]. Available: http://citeseer.ist.psu.edu/slutz98massive.html

[64] "Svlis solid modelling library homepage," August 2003. [Online]. Available: http://www.bath.ac.uk/~ensab/G_mod/Svlis

[65] R. Taylor and R. Hamlet, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402–1411, December 1990.

[66] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software reliability*, ser. TRW Series of Software Technology. North-Holland Publishing Company, 1978, iSBN 0-444-85217-4.

[67] D. Watson, "Computing the n-dimensional delaunay tessellation with application to voronoi polytopes," *The Computer Journal*, vol. 24, pp. 167–172, 1981.

[68] E. W. Weisstein. (2004, December) Monte carlo method. From MathWorld, a Wolfram Web Resource. Http://mathworld.wolfram/com/MonteCarloMethod.html.

[69] ——. (2004, February) Packing. From Mathworld - a Wolfram Web Resource. [Online]. Available: http://mathworld.wolfram.com/Packing.html

[70] ——. (2004, February) Tessellation. From Mathworld - a Wolfram Web Resource. [Online]. Available: http://mathworld.wolfram.com/Tessellation.html

[71] E. J. Weyuker and B. Jeng, "Analysing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, pp. 703–711, July 1991.

[72] L. White and E. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, vol. 6, pp. 247–257, 1980.

[73] M. Wilkes, personal communication, September 2003.

[74] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *Third International Conference On Quality Software (QSIC 2003)*. IEEE Computer Society, November 2003, pp. 20–24, dallas, TX, USA.

[75] R. Zhao and M. R. Lyu, "Character string predicate based automatic software test data generation," in *Third International Conference on Quality Software (QSIC 2003)*. IEEE Computer Society Press, November 2003, pp. 255–262.

# Appendix A

# Suitability of Random Number Generators for Random Testing

## A.1 Introduction

As discussed in section 1.1.4, while a great deal of theoretical and practical investigation into the properties of pseudorandom number generators has been conducted, no specific investigation of the suitability of standard random number generators for conducting random testing. While the periodicity of most random number generators is much larger than the sequence lengths used in previous studies, [44], some of the undesirable properties of some generators may cause them to give anomalous results when used for this purpose.

A specific concern is that the "plains problem" described by Marsaglia [46] might seriously influence failure-detection performance, particularly for strip failure patterns. Marsaglia's classic result applies to a a type of PRNG known as a multiplicative congruential generator, one of the most historically popular and widely used types of pseudorandom number generators. Such a generator has the general form:

$$I_x = K \cdot I_{x-1} \bmod M \tag{A.1}$$

where K and M are constants.

Marsaglia's theorem states that for such generators, if points in an n-dimensional space are selected by choosing $n$ successive outputs of the generator, the points generated will lie in fewer than $(n!M)^{\frac{1}{n}}$ hyperplanes. This is quite a small number - for the case that $M = 2^{32}$ and $n = 4$, $(n!M)^{\frac{1}{n}} = 566$.

The combination of this property of multiplicative congruential generations, and strip failure patterns, could potentially lead to failure-detection effectiveness considerably different to random testing. If a strip failure pattern happened to be congruent to many of the hyperplanes, the effectiveness might be considerably worse than random. Conversely, if we were lucky enough to have some of these hyperplanes not only congruent, but largely located within the failure pattern, performance might be significantly better than random. In either case, the generator's non-random properties would cause anomalous results, which in an experimental context would be problematic.

The GNU Scientific Library [33] (GSL) implements a large number of random number generators, including some of the most modern generators that are currently recommended for general purpose use by researchers in the area, random number generators as implemented in system libraries on a number of common platforms, and some historical generators, many of which are multiplicative congruential generators and some of which are known to have serious problems. Conveniently, all the generators are accessible using a common API. With the availability of 12 error-seeded programs previously used to test

the effectiveness of ART, it was straightforward to investigate whether the use of different random number generators for random testing produced statistically different results, and whether they were characteristic of what would be expected from true random numbers.

## A.2 Method

The twelve-error seeded programs used by Mak [44] were used for this study. They are a variety of numerical computation routines described in the well-known textbook *Numerical Recipes* [59]. For convenience, the original FORTRAN subroutines used by Mak were ported (line-by-line with the logic unmodified) to the C language by Chen *et. al* [16]. An unmodified and modified version were run with the same input parameters, and faults are detected by the two versions producing different outputs. Each program takes between one and four input parameters in specified ranges. A few input parameters are of C native type `int`, most are of type `double` (a double-precision floating-point type). For each of the 12 programs, the failure detection likelihood of the 35 pseudorandom number generators supported by the GSL were examined - each combination of a program under test and a PRNG representing an experimental condition. There were a total of 420 experimental conditions.

To evaluate the performance of an experimental condition, 1000 sequences of tests were generated, with the random number generator seeded using the initial values of 0 through 999. In each sequence, 100000 tests were generated, and the number of failures detected was recorded. It is trivial to show that the E-measure for random testing should have a normal sampling distribution; thus a non-normal distribution is an instant indicator of aberrant behaviour.

## A.3 Results

For each program under test, boxplots were constructed of the results for each pseudorandom number generator. Figures A.3 to A.14 show that, for each of the programs under test, the vast majority of the random number generators used give very similar results. Two generators did not, however: the "slatec" generator, and the "lecuyer21" generator. Not only was the median E-measure recorded in the trials noticeably different to the other generators, but the distribution of the E-measures appears to be non-normal. The "slatec" generator produces these odd results for virtually all of the programs under test, but the "lecuyer21" is sometimes close to the other generators. For instance, for the BESSJ program, the L'ecuyer generator was close to the other generators, but not for AIRY.

Quartile-quartile plots are a standard graphical technique used to assess the distribution of experimental data. In a "qqplot", the original dataset is sorted into ascending order. A dataset of the same size, with a known distribution, is also sorted into ascending order. The points in the two sets are then paired and plotted. If the distributions are the same, the points will form something close to a straight line. A common case is to compare with a synthetic dataset following the normal distribution; such a plot is a "qqnorm" plot.

A full presentation of the qqnorm plots for all 420 experimental conditions is impractical and redundant, as the overwhelming majority showed the distribution of E-measures through the experimental runs was close to normal. There was usually some small divergence from normality in the outliers, however: the qqnorm plot for Mersenne Twister generator for the program AIRY (Figure A.1) shows the straight line of the data near the mean, with the moderate divergence of the data a long way from it. By contrast, the qqnorm plot for the same program using the "slatec" generator (Figure A.2)looks quite odd, with highly stratified data.

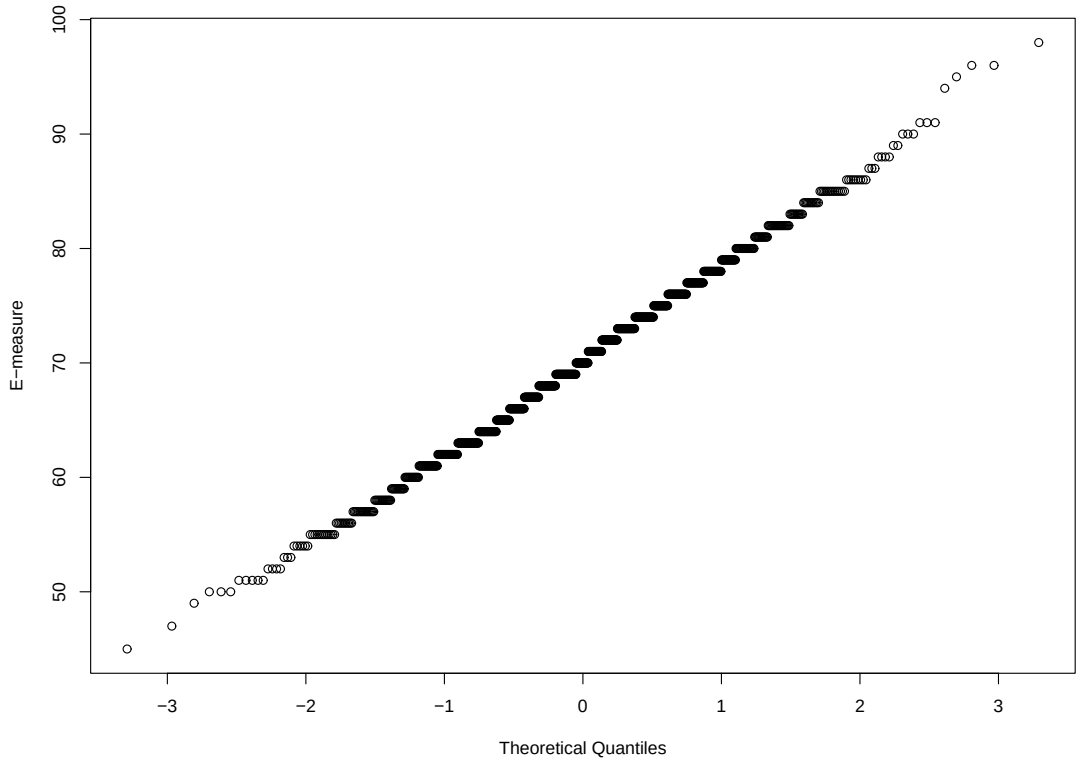



Figure A.1: QQNorm plot of E-measure distribution for the Mersenne Twister PRNG on program AIRY

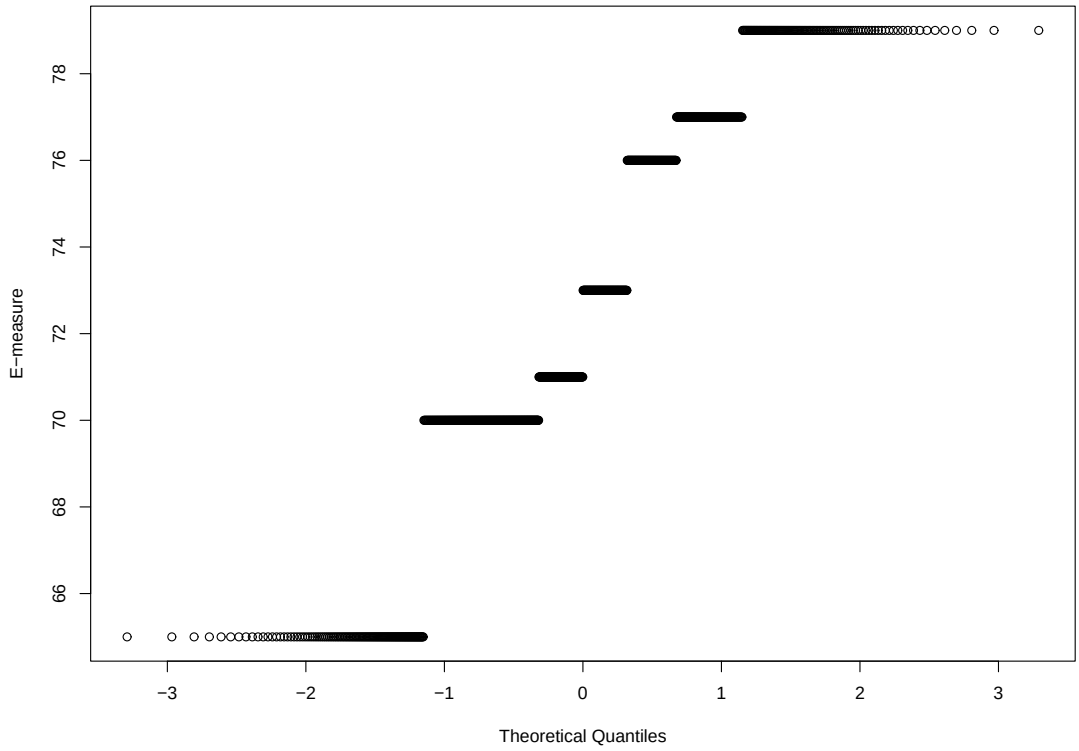



Figure A.2: QQNorm plot of E-measure distribution for the SLATEC PRNG on program AIRY

Because of the very large sample size, inferential statistics were able to detect the divergence from normality of the samples in many cases. For instance, a Shapiro-Wilk normality test on the data for the "randluxd2" generator, testing the program "el2", showed that the data was not normal, W=0.9965, p=0.023.

## A.4 Conclusion

In this simple study, I examined the effect of different number generators on the effectiveness of random testing by studying the distribution of estimated E-measures on a variety of sample programs, using a large set of different random number generators. In most cases, the results with the random number generators were about the same, and for each individual result approximately normally distributed, though the divergence from normality was detectable with statistical testing.

While the quantity of numbers in any particular sequence was insufficient to trigger problems with the periodicity of most generators, the "plains problem" [46] might have been thought to affect the results, particularly for those programs which take four-dimensional input. Our study showed no evidence for this, with the results from older, linear congruential generators approximately the same as from newer generators without known problems.

The two generators which did produce anomalous results are not used in system libraries on modern computers. The "slatec" random number generator is described in the GSL documentation as "ancient"[34], and the "lecuyer21" is taken from Knuth's *The Art of Computer Programming* [39], published in 1973. L'Ecuyer has, in fact, been active in the pseudorandom number generator community for many years and has designed a number of newer algorithms.

The slight divergence from normality present in a large number of experimental conditions is a little surprising. It may have something to do with the fact that a true normal distribution is for a continuous-valued statistic, whereas the number of failures detected in a test run is a discrete statistic. With the large sample sizes, most commonly-used statistical tests are reasonably robust to small deviations from normality, and, indeed, many, such as T-tests, have no such requirement.

In any case, it seems that most existing random number generators are satisfactory for the purposes of conducting random testing. There was no evidence to suggest that the known problems with older linear congruential generators affect simulation results. However, given the possibility that higher-dimensional test vectors may be susceptible to such issues, and given the free availability of similarly speedy generators which do not suffer from these issues, practitioners would be wise to use one of the more modern generators in this situation.

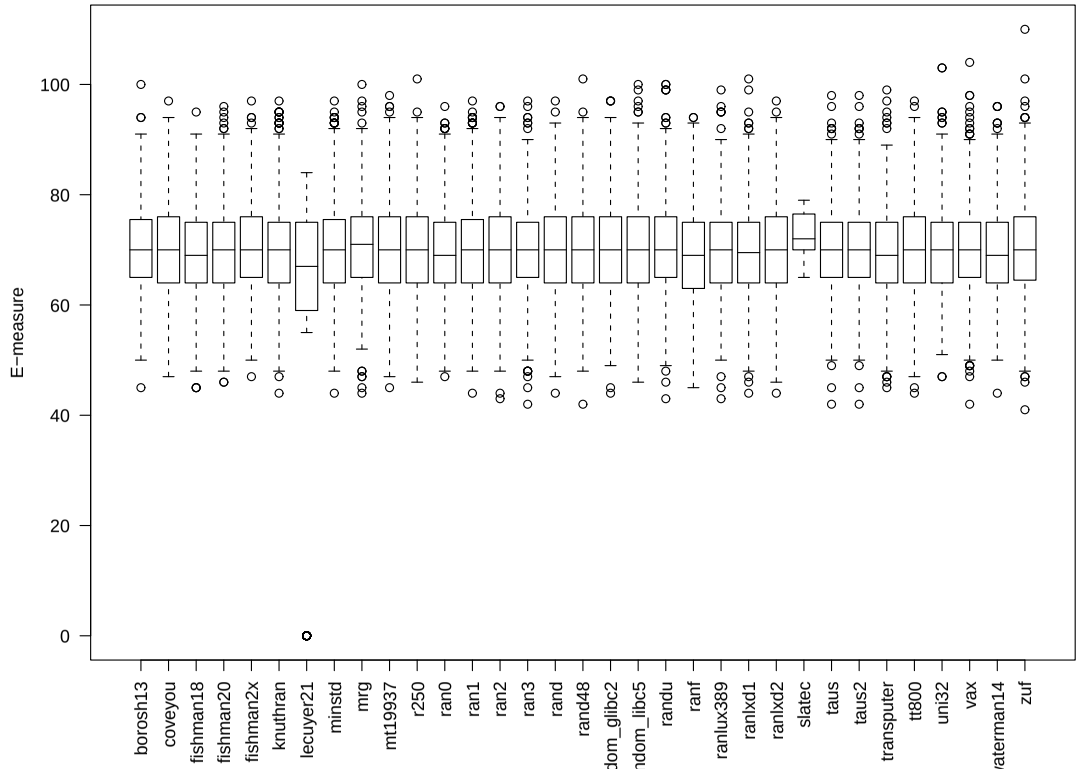



Figure A.3: E-measure for AIRY program with different random number generators

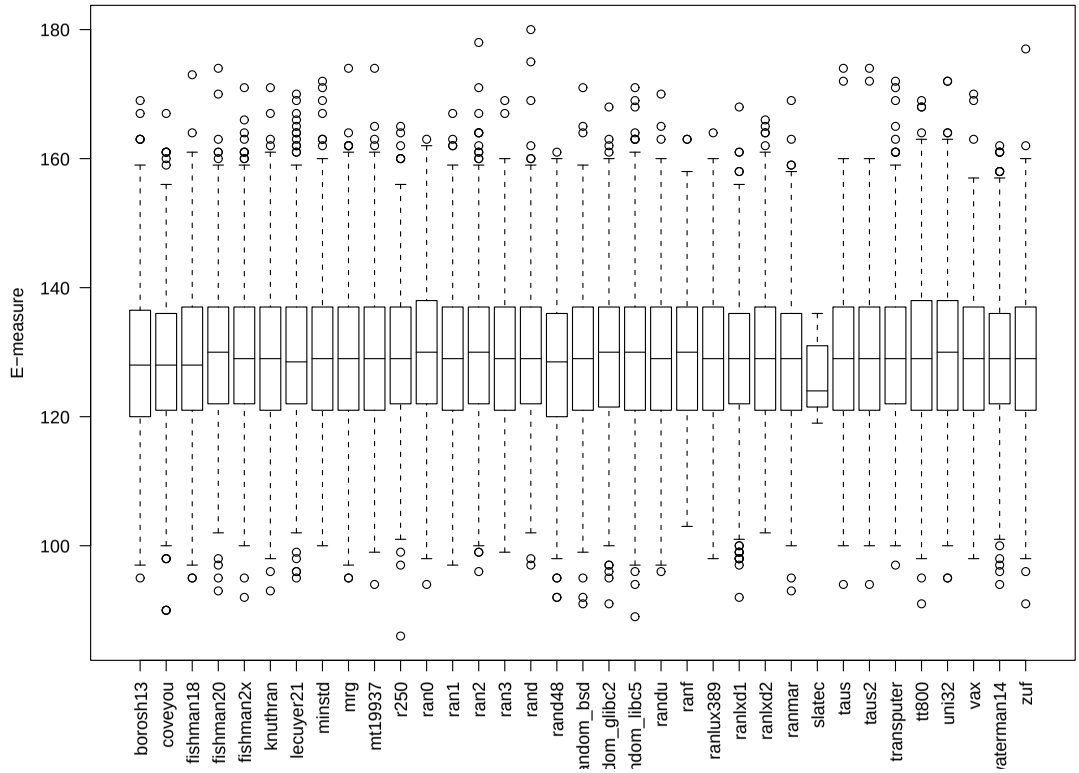



Figure A.4: E-measure for BESSJ program with different random number generators

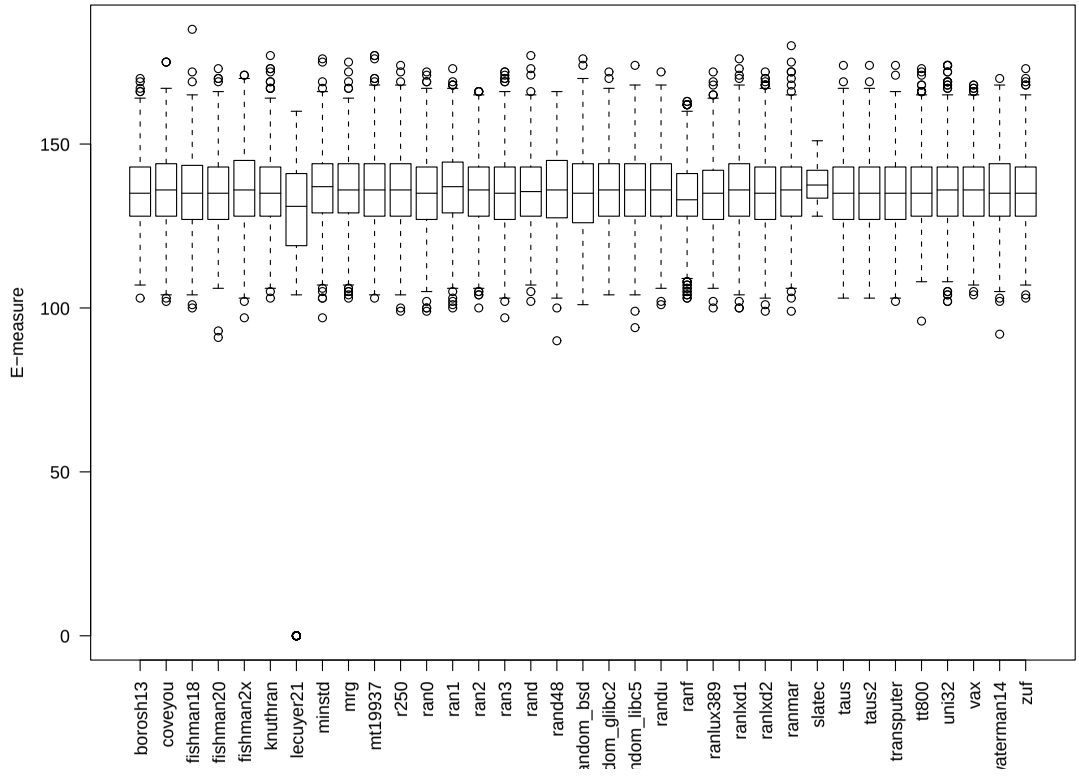



Figure A.5: E-measure for BESSJ0 program with different random number generators

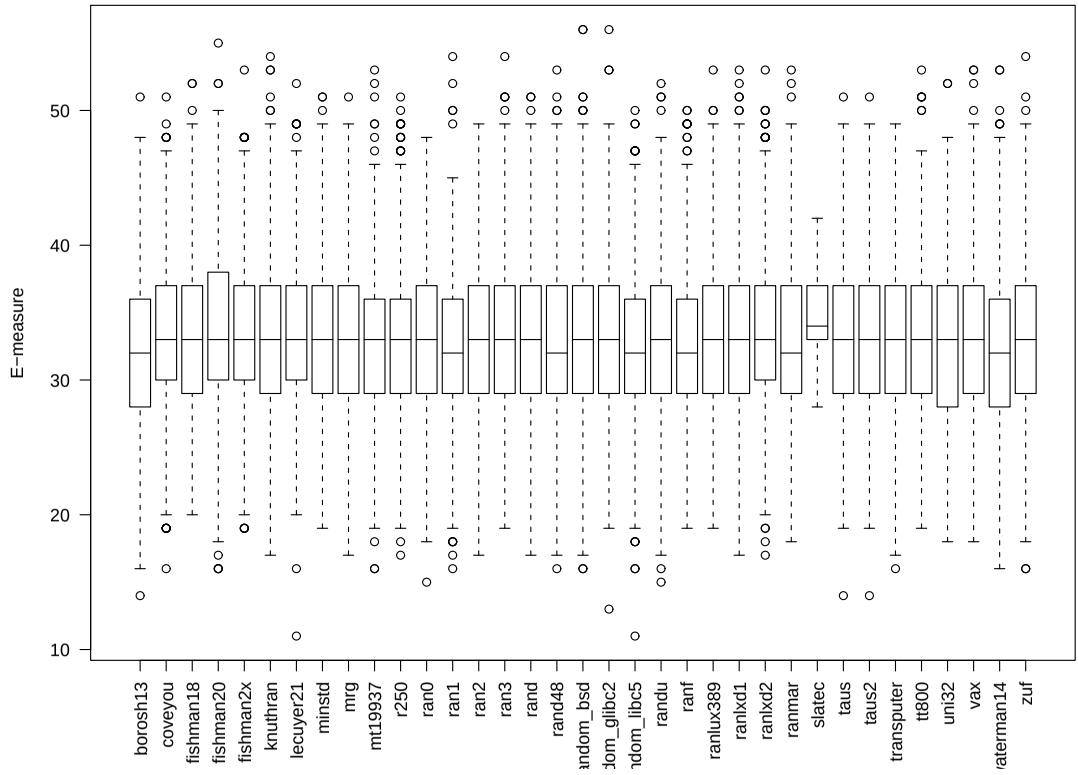



Figure A.6: E-measure for BESSJ0 program with different random number generators

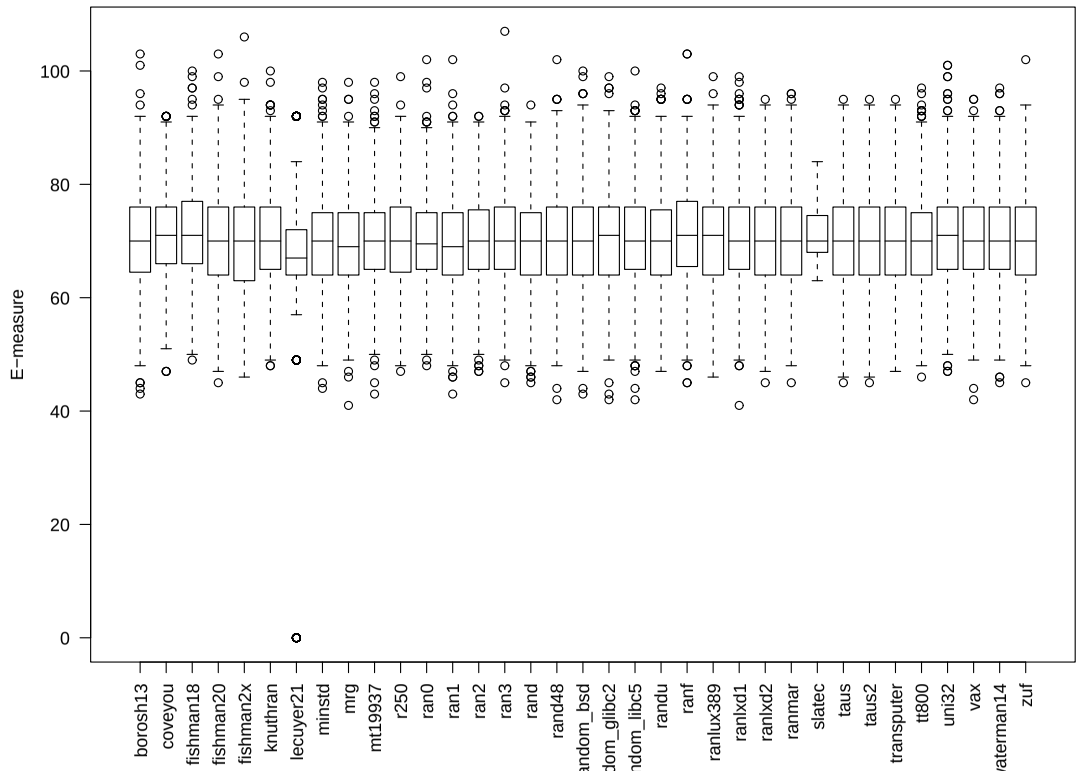



Figure A.7: E-measure for EL2 program with different random number generators

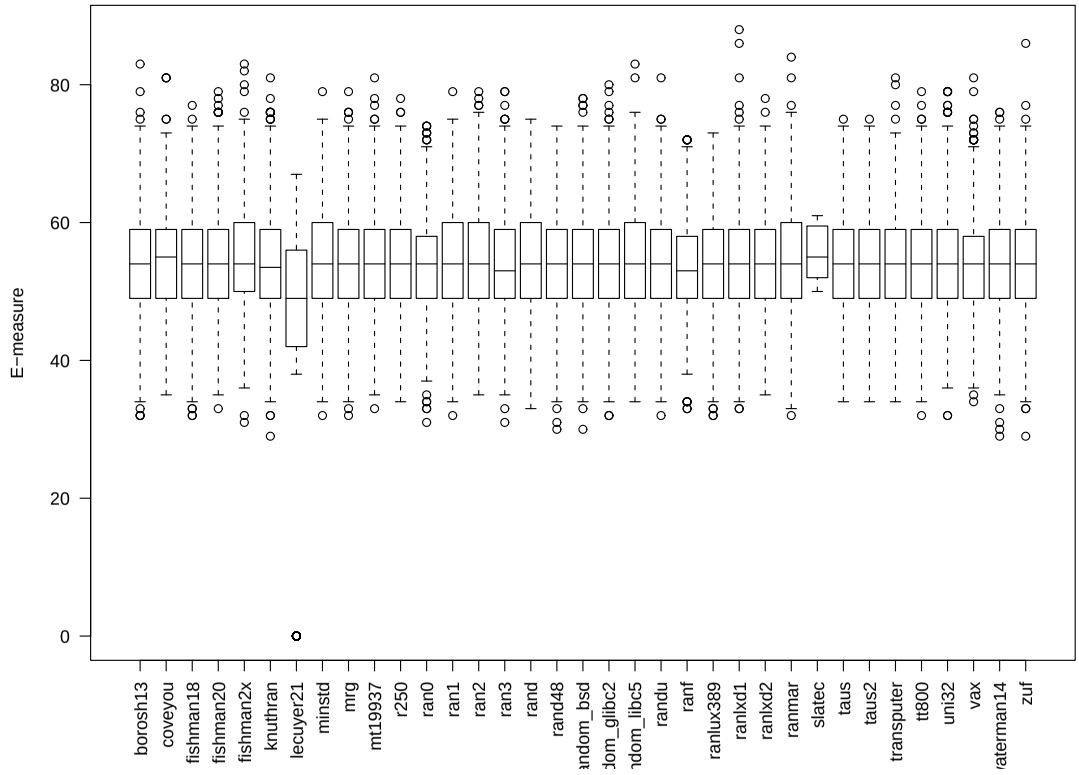



Figure A.8: E-measure for ERFCC program with different random number generators

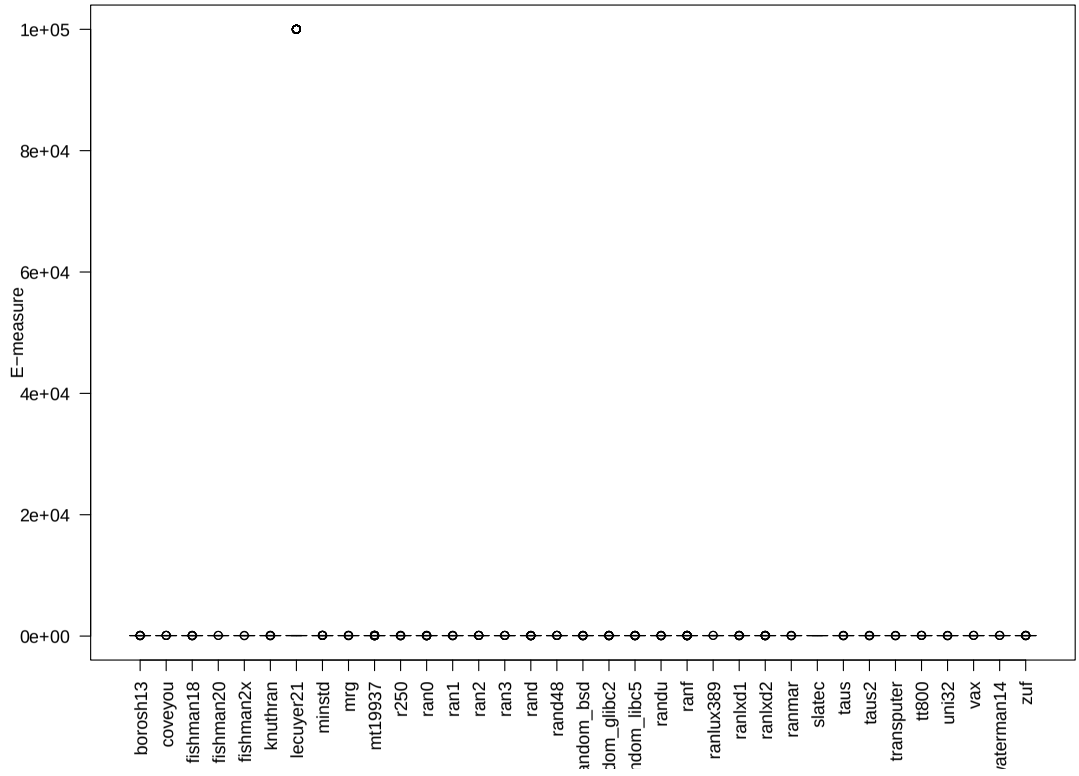



Figure A.9: E-measure for GAMMQ program with different random number generators

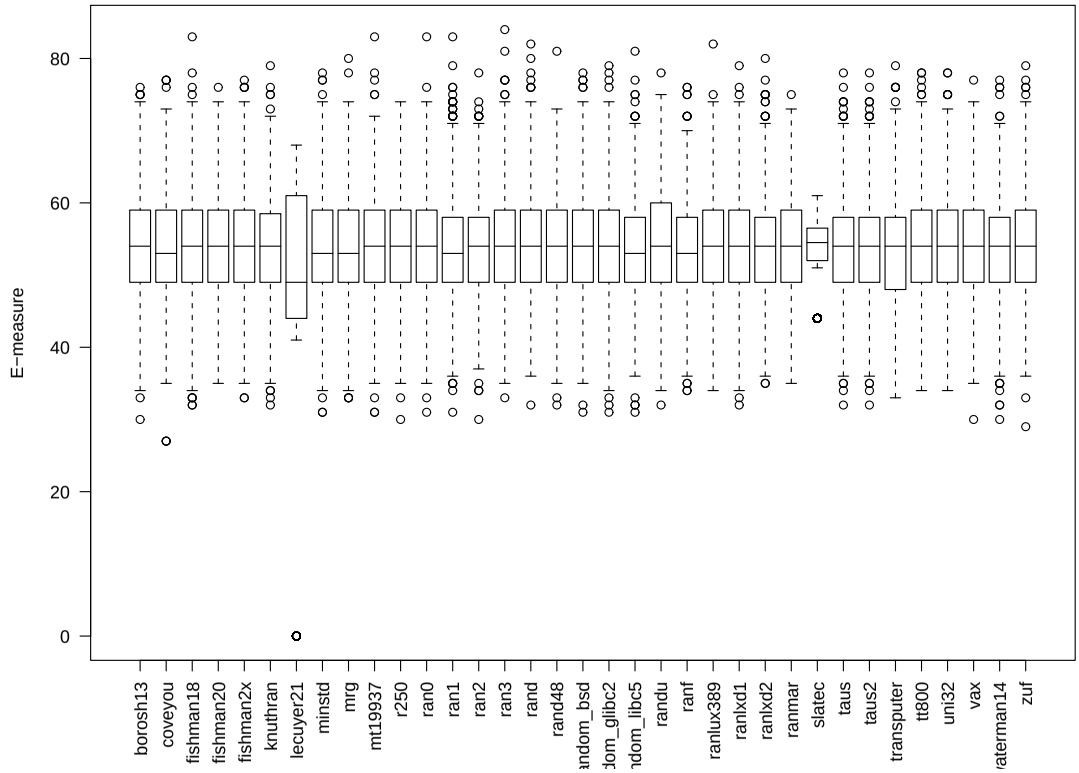



Figure A.10: E-measure for GOLDEN program with different random number generators

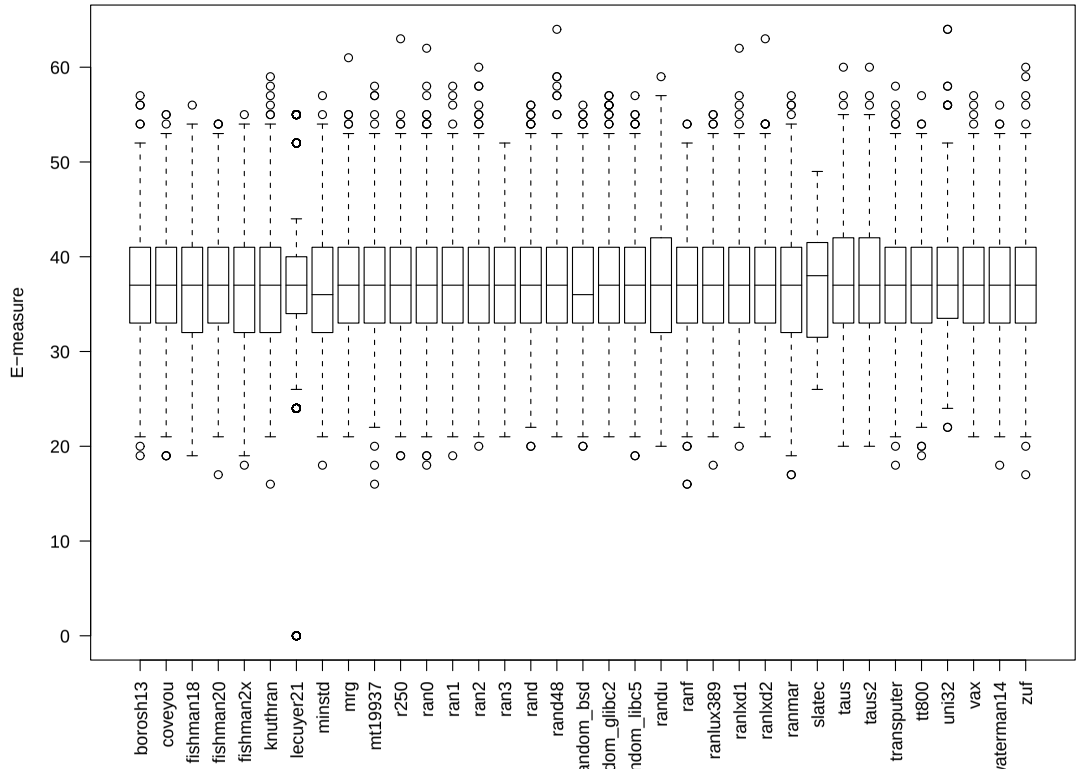



Figure A.11: E-measure for PLGNDR program with different random number generators

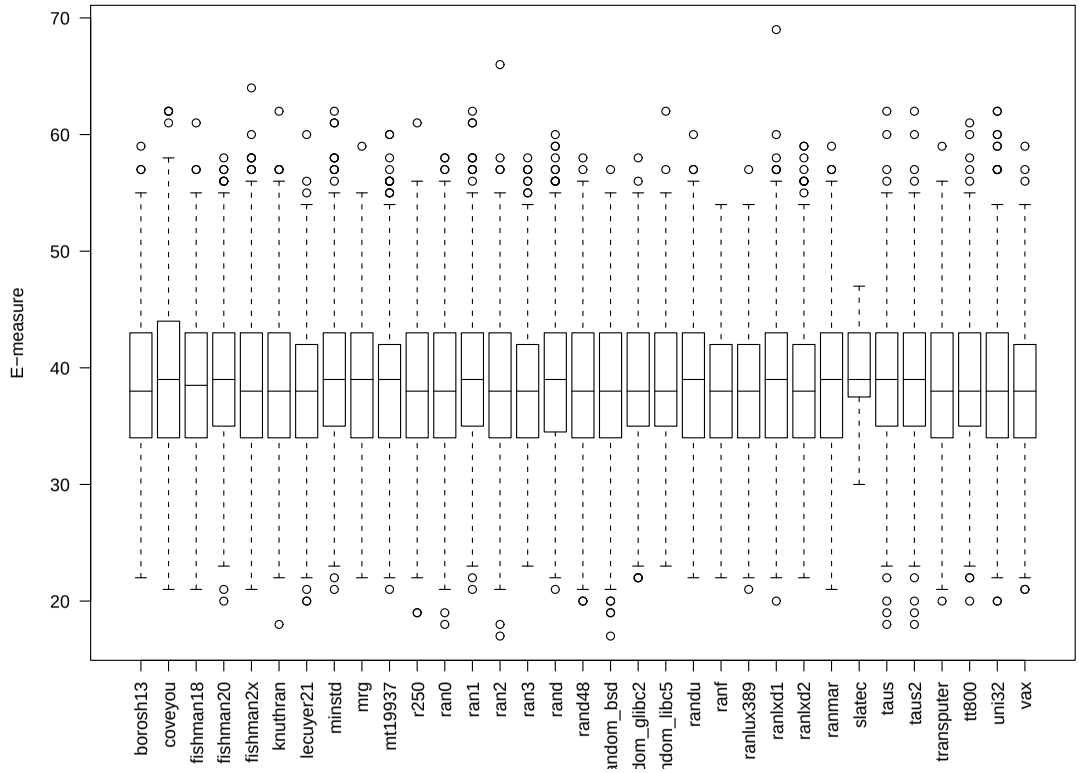



Figure A.12: E-measure for PROBKS program with different random number generators

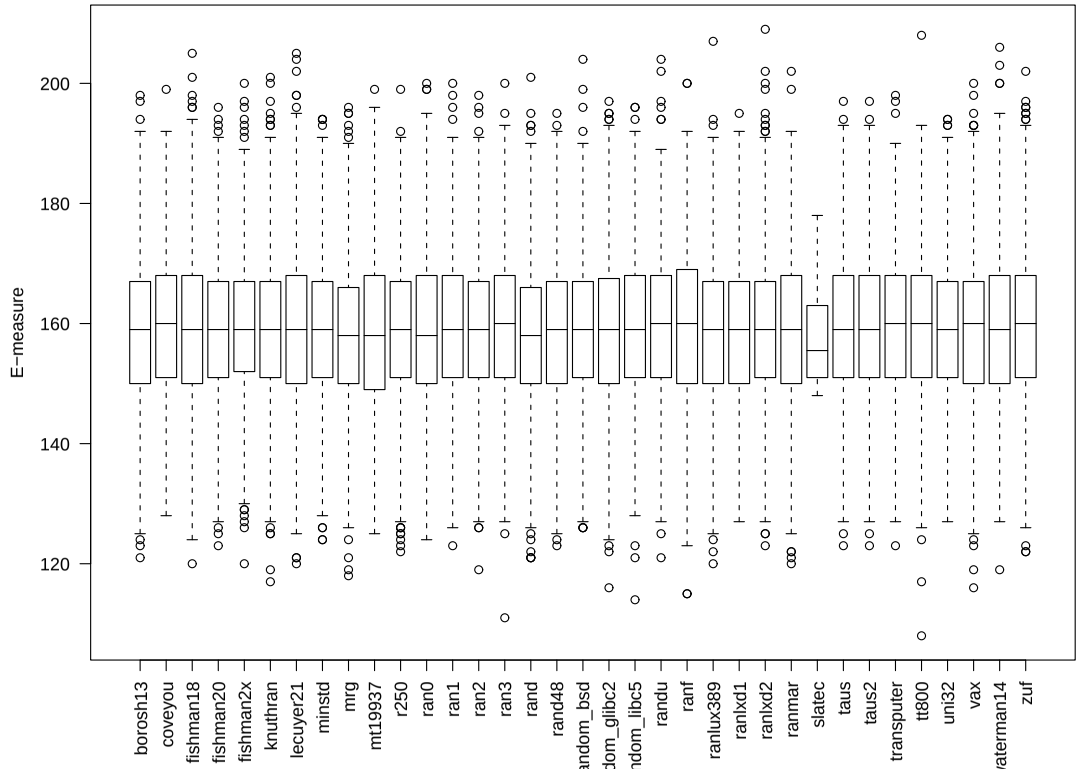



Figure A.13: E-measure for SNCNDN program with different random number generators

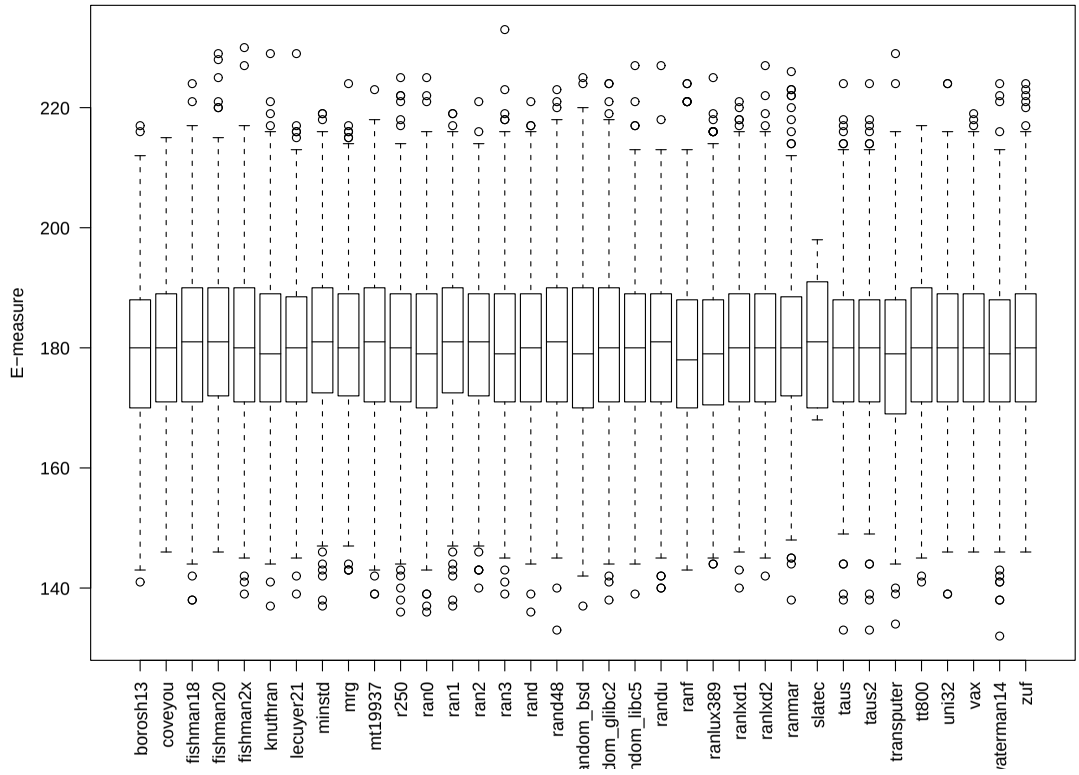



Figure A.14: E-measure for TANH program with different random number generators

# List of Publications

K.P. Chan, T.Y. Chen, F.-C. Kuo, R. Merkel, D.P. Towey, "Using the information: incorporating positive feedback information into the testing process", in *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP '03)*. Amsterdam, Netherlands: IEEE, September 2003, pp. 19-21.

T.Y. Chen, G. Eddy, R.G. Merkel and P.K. Wong, "Adaptive random testing through dynamic partitioning", in *Proceedings of the Fourth International Conference on Quality Software (QSIC 2004)*. Braunschweig, Germany. IEEE Press, September 2004. pp. 79-86

T.Y. Chen, F.-C. Kuo, and R. Merkel, "On the statistical properties of the f-measure", in *Proceedings of the Fourth International Conference on Quality Software (QSIC 2004)*. Braunschweig, Germany: IEEE, September 2004, pp. 146-153.

T.Y. Chen, F.-C. Kuo, R. Merkel and S. Ng, "Mirror adaptive random testing", *Information and Software Technology*, Vol 46, no. 15, pp 1001-1010, 2004.

T.Y. Chen and R. Merkel, "An upper bound of software testing effectiveness", submitted for publication.

T.Y. Chen and R. Merkel, "Quasi-random testing", submitted for publication.

T.Y. Chen, F.-C. Kuo, R. Merkel, and T.H. Tse, "Adaptive random testing for programs with non-numeric inputs", in preparation.