Diversity-Based Automated Test Case Generation

by

Ali Shahbazi

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Software testing is an expensive task that consumes around half of a project's effort. To reduce the cost of testing and improve the software quality, test cases can be produced automatically. Random Testing (RT) is a low cost and straightforward automated test generation approach. However, its effectiveness is not satisfactory. To increase the effectiveness of RT, researchers have developed more effective test generation approaches such as Adaptive Random Testing (ART) which improves the testing by increasing the test case coverage of the input domain.

This research proposes new test case generation methods that improve the effectiveness of the test cases by increasing the diversity of the test cases. Numerical, string, and tree test case structures are investigated. For numerical test generation, the use of Centroidal Voronoi Tessellations (CVT) is proposed. Accordingly, a test case generation method, namely Random Border CVT (RBCVT), is introduced which can enhance the previous RT methods to improve their coverage of the input space. The generated numerical test cases by the other methods act as the input to the RBCVT algorithm and the output is an improved set of test cases. An extensive simulation study and a mutant based software testing investigation have been performed demonstrating that RBCVT outperforms previous methods.

For string test cases, two objective functions are introduced to produce effective test cases. The diversity of the test cases is the first objective, where it can be measured through string distance functions. The second objective is guiding the string length distribution into a Benford distribution which implies shorter strings have, in general, a higher chance of failure detection. When both objectives are enforced via a multi-objective optimization algorithm, superior string test sets are produced. An empirical study is performed with several real-world programs indicating that the generated string

test cases outperform test cases generated by other methods.

Prior to tree test generation study, a new tree distance function is proposed. Although several distance or similarity functions for trees have been introduced, their failure detection performance is not always satisfactory. This research proposes a new similarity function for trees, namely Extended Subtree (EST), where a new subtree mapping is proposed. EST generalizes the edit base distances by providing new rules for subtree mapping. Further, the new approach seeks to resolve the problems and limitations of previous approaches. Extensive evaluation frameworks are developed to evaluate the performance of the new approach against previous methods. Clustering and classification case studies are performed to provide an evaluation against different tree distance functions. The experimental results demonstrate the superior performance of the proposed distance function. In addition, an empirical runtime analysis demonstrates that the new approach is one of the best tree distance functions in terms of runtime efficiency.

Finally, the study on the string test case generation is extended to tree test case generation. An abstract tree model is defined by a user based on a program under the test. Then, tree test cases are produced according to the model where diversity is maximized through an evolutionary optimization technique. Real world programs are used to investigate the performance of generated test cases where superior performance of the introduced method is demonstrated compared to the previous methods. Further, the proposed tree distance function is compared against the previous functions in the tree test case generation context. The proposed tree distance function outperforms other functions in tree test generation.

Preface

Chapter 2 of this thesis has been published as A. Shahbazi; A.F. Tappenden; J. Miller, "Centroidal Voronoi Tessellations -- A New Approach to Random Testing," *IEEE Transactions on Software Engineering*, vol.39, no.2, pp.163-183, Feb. 2013. I was responsible for developing the idea, the data collection and analysis, and the manuscript composition. J. Miller was the supervisory author and was involved with concept formation and manuscript composition. A.F. Tappenden also contributed in manuscript composition.

Chapter 3 of this thesis is submitted as A. Shahbazi; J. Miller, "Black-Box String Test Case Generation through a Multi-Objective Optimization," *Under revision in IEEE Transactions on Software Engineering*, 2015. I was responsible for developing the idea, the data collection and analysis, and the manuscript composition. J. Miller was the supervisory author and was involved with concept formation and manuscript composition.

Chapter 4 of this thesis has been published as A. Shahbazi; J. Miller, "Extended Subtree: A New Similarity Function for Tree Structured Data," *IEEE Transactions on Knowledge and Data Engineering*, vol.26, no.4, pp.864-877, 2014. I was responsible for developing the idea, the data collection and analysis, and the manuscript composition. J. Miller was the supervisory author and was involved with concept formation and manuscript composition.

Chapter 5 of this thesis is submitted as A. Shahbazi; J. Miller, "Black-Box Tree Test Case Generation through Diversity," *Submitted to IEEE Transactions on Software Engineering*, 2015. I was responsible for developing the idea, the data collection and analysis, and the manuscript composition. J. Miller was the supervisory author and was involved with concept formation and manuscript composition.

Acknowledgments

I would like to express my deepest respect and gratitude to my supervisor, Professor James Miller, for his patience, innovations, enthusiasm, and supports during the years of my Ph.D. study. It has been an honor to be his Ph.D. student.

I also would like to acknowledge the financial support provided by the Alberta Innovates (iCORE Graduate Recruitment Scholarship in ICT).

Finally, I would like to thank my mother for her support and encouragement.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview of Automated Software Testing

Software testing is any activity aimed at evaluating an attribute or capability of a software and determining software bugs. Software testing is an important step in the software development lifecycle due to the high cost associated with software bugs found after deployment. This cost can be reduced by optimizing the input test cases of the automated test case generation system. Considering the fact that software plays an important role in many aspects of human life, software failures can produce significant financial losses as well as endangering human lives. Although software testing cannot assure bug free software, its role is critical in software development. According to a study commissioned by the Department of Commerce's National Institute of Standards and Technology (NIST), software errors cost the U.S. economy 59.5 billion dollars annually [1]. Further, Jones [2] reported that due to poor software quality 500 billion dollars are lost worldwide, per year.

Accordingly, software testing consumes a significant portion of the software development budget. Studies have shown that often testing accounts for half of total project costs [3]. Since manual software testing is a labour-intensive task, the cost of testing is enormous, mostly because of the high cost of human resources. Manual testing is slow, leading to long time-to-market period, which increases the cost of software production. Further, human errors may be another drawback of manual testing. In addition, market pressures for the delivery of new functionality and applications have also never been stronger. The only practical solution to these difficulties is to automate the software testing process. Automated software testing has been introduced as an approach to reduce the cost and speed up the testing process. Further, it enhances the manual testing effort by increasing the testing coverage leading to higher software quality.

As indicated in Figure 1.1, a testing framework has three major components including test case generation, test case harness (execution), and a test oracle. Test case generation is the first step of the testing process. This component generates test cases where the objective is generating test cases with maximum coverage of the input space. In other words, the objective is generating minimum number of test cases that detects maximum

number of failures. The second step of the testing process that needs to be automated is test case harness which, in general, has two responsibilities. First, it executes the test cases generated in the previous step; and second, it captures the results that will be used in the oracle.

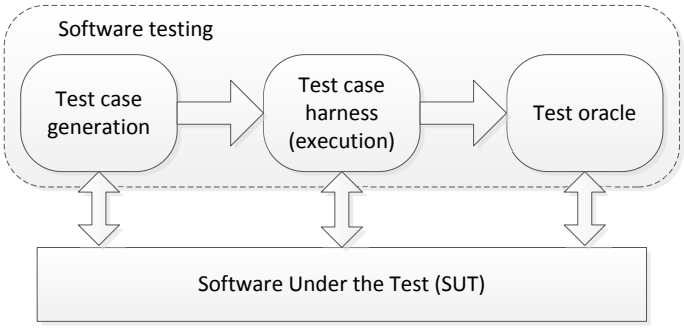


Figure 1.1. Software testing steps.

An oracle is a mechanism used in the testing process for determining whether a test has passed or failed [4]. To achieve this objective which is usually the most complicated part of the automated testing, the following two tasks need to be performed by the oracle:

1. The oracle must generate the expected results. The expected results are the outputs that the oracle determines that software should generate for the given input.

2. The second task is comparing the captured output(s) to the expected output(s) and then determining whether a test has passed or failed.

The testing can be automated in part of the process, for example, the test case generation and harness can be automated, while analyzing the results are performed manually (human oracle). Many industrial tools [5] for automated testing, that are sold for very high prices, only automate the test case harness component. That is, the user still needs to define test cases as well as the expected results. From an academic perspective, this level of automation is not considered automated testing. In fact, test case harness is the easy part, whereas automated test oracle and an effective automated test case generator are the difficult parts. For small systems, manual test case generation is easy to write and maintain. However, as systems become more complicated and the number of bugs increases, manual test generation is not effective and cost sensitive.

Automated test case generation can be divided into black-box testing and white-box testing. In black-box test generation, the automated test generation tool has no access to

the source code. Therefore, these methods are independent from the language of the source code. As a result, black-box testing methods are very general and independent of the programming language; all that is needed is the structure of the inputs and the outputs of the program under test. However, white-box test generation tools read and analyze the source code to generate test cases.

1.2 Random Testing and Input Coverage

Random Testing (RT) [6] is a straightforward black-box testing approach. RT's application in industry includes Dot NET error detection [7], security assessment [8], [9], Java Just-In-Time (JIT) compilers [10], and Windows NT robustness assessment [11]. Many companies use RT to detect security bugs; e.g. the Trustworthy Computing Security Development Lifecycle document (SDL) [12] states that fuzzing, a form of RT, is a key tool for security vulnerability detection.

RT is interesting since it has a low computational cost and is easy to implement. However, RT is not very effective regarding fault detection. According to various empirical studies, e.g. [13]–[17], faults usually occur in continuous regions within the input domain. This is referred to as error crystals by Finelli [14]. This means that faults are often clustered in the input space [18]. Accordingly, a diverse set of test cases that has a better coverage of the input domain has a greater chance of detecting a fault. As a result, RT's failure detection performance can be improved if test cases are distributed more diversely in the input space. RT test cases for a 2-dimensional space are presented in Figure 1.2, where RT's failure to evenly distributed test cases is demonstrated. That is, there is no test case in region one, while there are 14 test cases in region two.

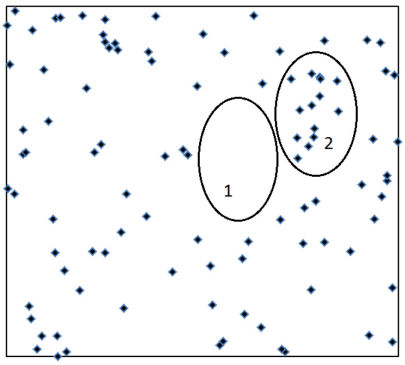


Figure 1.2. RT fails to evenly distribute the test cases throughout the input domain. No test cases is produced in region one by the RT generator, whereas we have 14 test cases in region two with a same size.

Adaptive Random Testing (ART) approaches [19]–[21] were developed to enhance the performance of RT. ART approaches generate more effective test cases by producing more diverse test cases across the input domain. Therefore, the probability of fault detection is improved [19].

1.3 The Focus of This Research

In this research, we limit our scope to black-box automated test case generation and hence, we introduce approaches to generate more effective test cases. Since black-box testing is a common testing strategy, any improvement in this domain could produce a significant improvement. Accordingly, the objective is to generate a diverse set of test cases. As explained in the previous section, failure usually occur in failure crystals or failure regions according to several empirical studies, e.g. [13]–[17]. Hence, it is believed that a diverse set of test cases is more likely to produce more effective test cases in the context of black-box testing.

To achieve this, we develop new test case generation methods for three data structures as test cases; numerical, string, and tree test cases. Hence, any program that accepts one of these types as input or the input that can be modeled by one of these data structures can be tested.

Accordingly, in chapter 2, numerical test case generation is studied where we introduced a new test generation method which is compared against the previous black-box numerical test case generators. We investigate the numerical test generation for higher dimensions than two. Further, the runtime of the new method is optimized and compared against the previous methods.

Following that, string test cases are investigated in chapter 3. A few string test case generation methods are investigated and compared. We indicate that with multi-objective optimization where diversity and string size distribution are the objectives, more effective test cases can be generated. We also investigate the performance of a few string distance functions which are part of string test generation.

In chapter 4, we propose a new tree similarity and/or distance function which works based on tree mappings. We empirically investigate the performance of the new function compared to other tree distance functions in clustering and classification applications. We introduce this tree distance function to later use it in a tree test generation framework in

the next chapter.

Finally, we study tree test case generation in chapter 5. Test case generation methods from the string generation chapter are ported to generate trees based on an abstract tree model. Again test generation methods are evaluated in an empirical framework. Furthermore, the proposed tree distance function is compared against the other tree distance functions in the context of test cases generation.

2 Numerical Test Data Generation Using Centroidal Voronoi Tessellation

Although Random Testing (RT) is low cost and straightforward, its effectiveness is not satisfactory. To increase the effectiveness of RT for numerical test case generation, researchers have developed Adaptive Random Testing (ART) and Quasi-Random Testing (QRT) methods which attempt to maximize the test case coverage of the input domain. This chapter proposes the use of Centroidal Voronoi Tessellations (CVT) to address this problem. Accordingly, a test case generation method, namely Random Border CVT (RBCVT), is proposed which can enhance the previous RT methods to improve their coverage of the input space. The generated test cases by the other methods act as the input to the RBCVT algorithm and the output is an improved set of test cases. Therefore, RBCVT is not an independent method and is considered as an add-on to the previous methods. An extensive simulation study and a mutant based software testing investigation have been performed to demonstrate the effectiveness of RBCVT against the ART and QRT methods. Results from the experimental frameworks demonstrate that RBCVT outperforms previous methods. In addition, a novel search algorithm has been incorporated into RBCVT reducing the order of computational complexity of the new approach. To further analyze the RBCVT method, randomness analysis was undertaken demonstrating that RBCVT has the same characteristics as ART methods in this regard.

2.1 The Focus of This Chapter

In this chapter, we propose a new test case generation approach, namely Random Border Centroidal Voronoi Tessellations (RBCVT) which utilizes Centroidal Voronoi Tessellations (CVT). The proposed RBCVT approach enhances the existing state-of-the-art test case generation techniques. Specifically, we will demonstrate that RBCVT:

1. Is able to produce a superiorly distributed set of test cases when compared to RT, ART, and QRT;
2. Still retains the random nature of RT; and,
3. Can be optimized to have linear execution characteristics across a wide set of situations.

RBCVT is not an independent method to generate input test cases. It considers other test case generation methods as an input and increases software testing effectiveness by spreading the test cases more diversely throughout the input domain. In addition, a novel search algorithm is proposed to enhance the computational complexity of the RBCVT test case generation from a quadratic to linear runtime order.

In addition to the even distribution of test cases over the input space, the degree of randomness 1) within a set of test cases; and 2) between multiple sequences of test sets, is an important aspect. The test cases' randomness is critical in avoiding systematic poor-performance in certain situations (that is, where a non-random sequence could significantly (negatively) correlate with a current set of defects). Similarly, in regression-type testing, we can prevent inefficient testing if test cases are uncorrelated with respect to each other, meaning a high degree of randomness. The proposed RBCVT approach seeks to generate a more effective sequence of test cases with respect to software testing practice, while retaining the degree of randomness possessed by RT and ARTs methods. This, randomness requirement, is investigated using Kolmogorov complexity which provides a new class of distances appropriate for measuring similarity relations between sequences [22], [23].

2.2 Notations Used in This Chapter

The following notations and assumptions are provided to simplify the discussion in the rest of this chapter.

- I denotes the input space which is considered a two-dimensional unit hypercube ($I = [0,1]^2$).
- H denotes the area outside I which is defined as $H = [0-h, 1+h]^2 - I$ where the width of H is indicated by h.
- d denotes the dimension of a test case or input space.
- $|\bullet|$ denotes the size of a set.
- T denotes selected test cases on I generating a test set ($T = \{t_i\}_{i=1}^{|T|}$).
- B denotes a random background point set on $H \cup I$ regarding the RBCVT calculation algorithm ($B = \{\mathrm{b}_j\}_{j=1}^{|B|}$).
- R denotes a random border point set on H which simulates random borders in

RBCVT approach ($R = \{\mathrm{r_n}\}_{n=1}^{|R|}$).

- TR denotes the combination of T and R which is defined as $TR = T \cup R = \{tr_m\}_{m=1}^{|TR|}$ where $|TR| = |T| + |R|$.
- V_i denotes a Voronoi region (a cell in Voronoi tessellation).
- $dist(p,q)$ denotes the Euclidian distance between points p and q.
- $\beta(p,T)$ denotes nearest point of T to the point p.
- $O(.)$ represents the runtime order of an approach.
- argmax$(.)$ returns the index of an element with maximum value.
- θ denotes the failure rate.
- std denotes the standard deviation.
- $\oplus$ is the bit-by-bit exclusive-or operator.
- $\underset{j=1,\ldots,k}{\mathrm{XOR}}(.)$ denotes the bit-by-bit exclusive-or for the specified range.
- G_N represents the number of cells in each dimension of the grid with respect to RBCVT-Fast algorithm.
- C_{avg} denotes the average number of points in each cell.
- $Round(.)$ returns the nearest integer value to the input data.
- C_l is a set which contains all the cells in layer *l* where each cell in C_l is denoted by c_{lm}.
- $dist_c(b_j, c_{lm})$ indicates the minimum Euclidian distance between the point b_j and the cell c_{lm}.
- $dist_l(b_j, l)$ represents the minimum Euclidian distance between point b_j and cells in layer *l*.
- $\beta_c(b_j, c_{lm})$ denotes nearest child of c_{lm} to the point b_j.
- tr_{winner} denotes a point of *TR* with minimum Euclidian distance from b_j.
- $RTime(.)$ denotes a runtime of an algorithm or a method.
- $\varphi(T)$ indicates the preprocessing function which preforms the required processing on T regarding randomness analysis.

- $CR(T)$ represents the compression ratio of T .
- $\delta(.)$ denotes the Kolmogorov complexity of the input data.
- $NCD(T_i, T_j)$ represents the normalized compression distance between T_i and T_j .

2.3 Current Approaches

2.3.1 Adaptive Random Testing (ART)

Adaptive Random testing methods seek to resolve the deficiencies of RT demonstrated in Figure 1.2. These methods seek to retain the random nature of RT, while providing a more "even distribution" of the sequence of test cases across the input domain. Since the introduction of ART by Chen et al. [18] a variety of different ART methods have been proposed, including Fixed Size Candidate Set (FSCS) [18], [24], [25], Restricted Random Testing (RRT) [26], Mirror Adaptive Random Testing (M-ART) [27], Adaptive Random Testing by Bisection (ART-B) [28], Adaptive Random Testing by Random Partitioning (ART-RP) [29], ART through Iterative Partitioning (IP-ART) [30], ART based on distribution metrics [31], and Evolutionary Adaptive Random Testing (EAR) [19].

The ART methods are developed based on the observation that failures occur in failure regions which are clustered within the input domain. Each of these methods possesses strengths and weaknesses regarding efficient test case generation and computational complexity. Via empirical investigations, Mayer et al. [32] concluded that FSCS [18], [24], [25] and RRT [26] were the best ART methods. Subsequently, Tappenden and Miller [19] introduced EAR and demonstrated that this method has superior performance than FSCS and RRT. Hence, we compare RBCVT's performance against these methods. In each of these ART techniques, the first test case is generated randomly and subsequent test cases are based on each method's specific algorithm.

2.3.1.1 Fixed Size Candidate Set (FSCS)

FSCS uses a distance based algorithm to generate test cases [18]. In this method, a fixed size candidate set is used to produce test cases. A set of k randomly generated candidates, cd , are evaluated against all previously selected test cases and a candidate with largest distance from previously executed test cases is selected as

$$J = \arg\max_{j=1,\ldots,k} \left(dist\left(cd_j, \beta\left(cd_j, T\right)\right)\right), \tag{2.1}$$

where cd_j denotes the jth candidate; and J represents the index of the selected candidate as a next test case. The computational requirement for this method is $FSCS(|T|) \in O(|T|^2)$ due to the computation of the distance between candidates and each previously generated test case [19], [32].

2.3.1.2 Restricted Random Testing (RRT)

RRT [26] also uses a distance based algorithm to generate test cases via a circular exclusion zone [32] centered around each previously generated test case. The radius of each exclusion zone is determined using a constant coverage ratio (γ), which is the sum of the areas of all the existing exclusion zones divided by the total area of the input domain. A candidate test case (cd_j) is generated randomly, and disregarded, if it is within the exclusion zone of any other test case, i.e. if the following inequality is true.

$$dist\left(cd_j, \beta\left(cd_j, T\right)\right) < \sqrt{\frac{\gamma}{\pi |T|}}. \tag{2.2}$$

This process is repeated until an appropriate candidate is found [26]. Calculation of the algorithm's computational efficiency is not straight forward, given the stochastic nature of the technique. However, it has been demonstrated empirically that the average runtime order is within $RRT(|T|) \in O(|T|^2 \log(|T|))$ [32].

2.3.1.3 Evolutionary Adaptive Random Testing (EAR)

EAR uses an evolutionary approach to find an approximation for the test case that has the maximum distance from all the previous test cases [19]. For each test case, a pool of k (population size) random candidates is generated. This population is evolved until a stopping criterion is met. This approach is encoded using two genes in each chromosome. Each gene is a number representing the value for one of the two dimensions. The evolution is based upon a Euclidean distance-based fitness function [19]

$$Fitness\left(ch_j, T\right) = dist\left(ch_j, \beta\left(ch_j, T\right)\right), \tag{2.3}$$

where ch_j represents a chromosome. Single-point crossover was applied to the two chromosomes to generate an offspring evolving the population. When the stopping criterion is met, the best chromosome is selected as the next test point according to the fitness function. The runtime of this algorithm [19] is in the order of quadratic time

$(EAR(|T|) \in O(|T|^2))$.

It is worthwhile to note that there are two sub-optimum techniques, introduced in previous ART studies, to reduce the ART computational complexity, namely mirroring [27] and forgetting [33]. Both techniques can be applied to all the studied ART methods. Producing the next test case gets more time consuming as the number of test cases grows. Accordingly, the technique of forgetting only considers a constant number of previous test cases when designing a new test case, not all of them. It makes the new test case design independent of $|T|$, leading to a one order reduction in the overall time complexity. In mirroring, ART is only applied to a part of the input domain and then the designed test cases are mirrored to other parts. Obviously, there is a trade-off between effectiveness and computational complexity, if the techniques of mirroring and forgetting are applied.

2.3.2 Quasi-Random testing (QRT)

In addition to ART, the use of quasi-random sequences in software testing has been recently proposed [34], [35] for numerical test case generation. Quasi-random sequences are mathematically developed sequences which are rigorously designed to produce low-discrepant sample points in a d-dimensional hypercube. They fill the space more uniformly than uncorrelated random points. It has been observed [34], [35] that using these sequences as input test case generators produces better results than RT in software testing. However, it has not been shown that their results are better than ART methods. Until now various quasi-random sequences have been constructed including Sobol [36], Halton [37], Niederreiter [38], Faure [39], and Hammersley [40]. In this chapter, we consider the following quasi-random sequences.

2.3.2.1 The Halton Sequence

The Halton sequence has been derived from Van der Corput sequence [35] which is defined as

$$\Phi_b(n) = \sum_{j=0}^{k} n_j b^{-j-1}, \tag{2.4}$$

where n_j is the jth digit of n in the base b; and k denotes the lowest integer that makes $n_j = 0$, for all $j > k$. The Halton sequence can be seen as the natural d-dimensional extension of the Van der Corput sequence. The Halton sequence generates

values deterministically using prime numbers as its base. The standard Halton sequence performance is good in low dimensions, whereas in large dimensions a correlation problem between sequences generated in different dimensions appears [41]. As a remedy, several scrambling and randomization methods have been introduced [41].

2.3.2.2 The Sobol Sequence

The Sobol sequence [36] has been proposed for software testing by Chi and Jones [35]. The Sobol sequence can be considered as a permutation of the binary Van der Corput sequence in each dimension [35] and is defined by the following equations.

$$Sobol(n) = \underset{j=1,\ldots,k}{\text{XOR}}\left(n_j w_j\right), \tag{2.5}$$

$$w_j = \underset{i=1,\ldots,r}{\text{XOR}}\left(\frac{\alpha_i w_{j-i}}{2^j}\right) \oplus \frac{w_{j-r}}{2^{j+r}}, \tag{2.6}$$

where n_j is the jth digit of n in binary, k represents the number of digits of n in binary, and $Sobol(n)$ denotes nth element of the Sobol sequence. To construct a Sobol sequence, we need to choose a primitive polynomial of degree r with $\alpha_i \in \{0,1\}$ coefficients. The required computational overhead for the Sobol generator is within the order of $Sobol(|T|) \in O(\log(|T|)^2)$ [42]. This low computational cost is the primary advantage of QRT compared to ART approaches.

2.3.2.3 The Niederreiter Sequence

The Niederreiter sequence was introduced in 1988 [38] and provides a general form for quasi-random sequences. This sequence has provided a good reference for other quasi-random sequences, as all of these methods can be described in terms of what Niederreiter called (t,s)-sequence. The discrepancy of this sequence is lower than any other known sequence [34]. Chen et al. [34] has proposed this sequence for test case generation where a large number of test cases are required.

2.4 Centroidal Voronoi Tessellation (CVT)

In this section, we introduce the concept of CVT and discuss approaches to its calculation as well as its application to software testing. A Voronoi diagram (Voronoi tessellation) is a decomposition of a space, in our case a unit hypercube, into a set of cells (Voronoi

regions) such that $V_i \cap V_j = \varnothing$ for $i \neq j$; and $\bigcup_{i=1}^{k} V_i = 11$, where V_i is a Voronoi region and *k* is the number of Voronoi regions. Each Voronoi region is associated with an object and consists of all the areas that are closer to that object than any other object. These objects are disjoint [43] and are referred to as the generators of the Voronoi diagram. In this chapter, an object is a point (t_i) and Euclidian distance is considered as a distance measure. The Voronoi region corresponding to the point t_i is defined as

$$V_i = \left\{ x \in I \mid \forall j = 1,...,|T|, j \neq i : dist(x, t_i) < dist(x, t_j) \right\}. \tag{2.7}$$

Centers of mass, centroids, of a Voronoi region (V_i) is defined as

$$t_i^* = \frac{\int_{V_i} x\rho(x)\,dx}{\int_{V_i} \rho(x)\,dx}, \tag{2.8}$$

where ρ is a density function defined in *I*. Centroids in the decomposed cells of a Voronoi tessellation possess characteristics that seem to have some advantages with respect to software testing. In Figure 2.1, adapted from [44], 10 randomly (RT or alternatively by using ART or QRT techniques) generated points are used as the generators or inputs to the system. Accordingly, the Voronoi regions have been formed corresponding to the generators and the centroid of each Voronoi region is indicated by a circle. As shown in this figure, the resulting circles are "more evenly distributed" compared to the input points making them more appropriate for software testing.

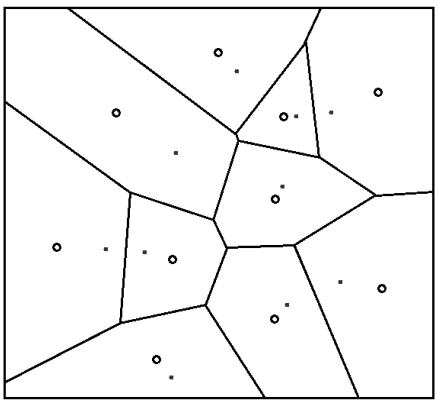


Figure 2.1. The lines specify Voronoi regions corresponding to 10 randomly generated points. The points are Voronoi generators and the circles are the centroids of the Voronoi regions.

A CVT is a collection of Voronoi regions where their generator points are the centroids of the corresponding Voronoi regions [44]. This case is a special case; and the probability of a set of random generators having the same positions as the centroids is quite low. In general, the generators of Voronoi tessellations will not be at the same places as the centroids. An important property of CVT is that these special generators producing a CVT are not unique and we can have distinct CVTs within a *d*-dimensional unit hypercube [44], [45].

A CVT can be produced either deterministically or probabilistically [44]–[46]. A deterministic approach, such as Lloyd's method [44], produces a consistent output for every input. Whereas, a probabilistic approach, such as MacQueen's method [45], uses a random mechanism to generate a CVT leading to distinct outputs, for the same input set, in different runs allowing additional exploration of the input space. Since this is beneficial during testing scenarios (e.g. regression situations), we develop a probabilistic calculation approach in this study for the RBCVT test case generation method, which is introduced in Section 2.5.

2.4.1 CVT and Software Testing

In this section, we introduce the application of CVT in software testing as well as its desirable and undesirable features in this regard. CVT has been applied within the wide array of applications [44]. However, the use of this technique for improving RT, ART and QRT techniques is novel. The CVT methodology requires a set of initial points named generators. The use of the output from other test case generation methods (RT, ART, and QRT) is proposed as inputs (generators) to the CVT algorithm leading to an improved set of test cases. Chen and Merkel [47] presented a new calculation method for FSCS using Voronoi diagrams; they utilized Voronoi diagrams to develop a search algorithm with the ability to calculate $\beta\left(c_j, T\right)$ with a reduced computational complexity. This work is significantly different from our proposed use of Voronoi diagrams in test case generation, since they use Voronoi diagrams to speed up finding the nearest point in FSCS test case generation approach, whereas we use the centroids of Voronoi regions to improve the effectiveness of the test case generation.

To indicate CVT's effect on test cases, Figure 2.2 is presented. This figure indicates the generator (input) points for CVT (Figure 2.2a), points generated by RT, as well as the resultant points generated by CVT (Figure 2.2b). According to this figure, one can

observe that CVT points possess the following desirable properties:

- The CVT points are more "evenly distributed" than their generators in the space. Since faults often occur in failure regions or error crystals, the CVT points are likely to detect a failure region more efficiently.
- As discussed in the previous section, as CVT generates its (output) points by a probabilistic approach, the displayed points are not unique as the CVT process is stochastic. Furthermore, the input generators are generated using a random procedure, except for quasi-random points. Therefore, the output CVT points seem to possess "randomness" (the randomness will be investigated in Section 2.8).

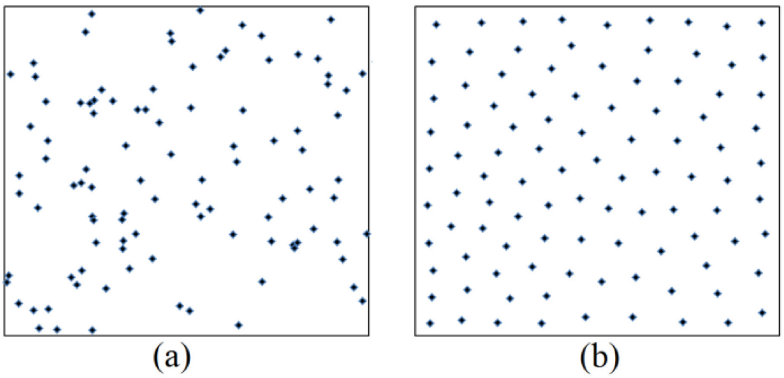


Figure 2.2. The (a) RT and (b) corresponding CVT points generated using a probabilistic approach.

Further, the application of CVT to software testing requires a unique solution to the "boundary conditions" introduced by this domain. It is a well-established principle that the probability of a software defect is higher near the boundaries. In this regard, CVT needs to be extended to explicitly consider defect behavior near these boundaries. As indicated in Figure 2.2b, all the test cases near the borders have a relatively constant distance with the border. Accordingly, CVT is unable to generate test cases near or on the border. This undesirable feature is due to the traditional CVT definition. To solve this problem, we propose the novel RBCVT approach, which is presented in the next section.

2.5 Proposed Test Case Generation Approach: Random Border CVT (RBCVT)

In this section, we propose the novel RBCVT test case generation approach, which removes the undesirable feature of the CVT discussed in the previous section. In this

regard, we propose a RBCVT calculation approach and investigate its associated runtime order. In addition, we propose a novel search algorithm to reduce the computational complexity of RBCVT. Finally, we investigate the generalization of the RBCVT beyond two dimensions.

RBCVT is based on defining an imaginary random border outside the real borders of *I*. In this regard, we introduce a set of random points (*R*) in *H,* which simulate an imaginary random border as discussed in the next section. In Figure 2.3, a set of RBCVT test cases is demonstrated as well as the random border points in *H*. As indicated in this figure, RBCVT effectively removes the aforementioned undesirable feature of the CVT. Accordingly, Figure 2.4 indicates the generator points of RBCVT (one for each of the seven test generation methods studied) in the left-hand side; and the resultant RBCVT points on the right-hand side.

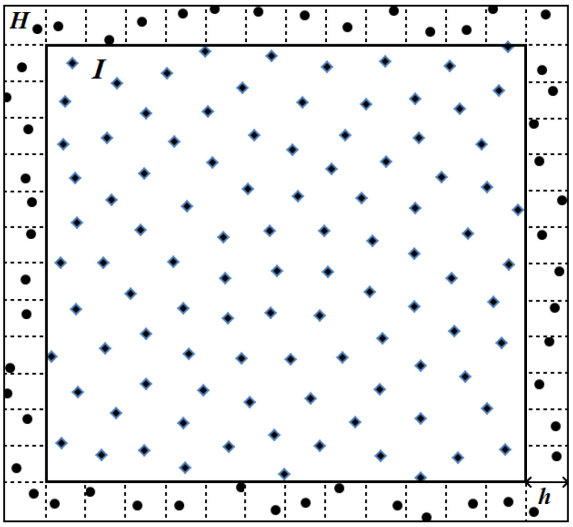


Figure 2.3. RBCVT test cases in I and the random border points (R) in H.

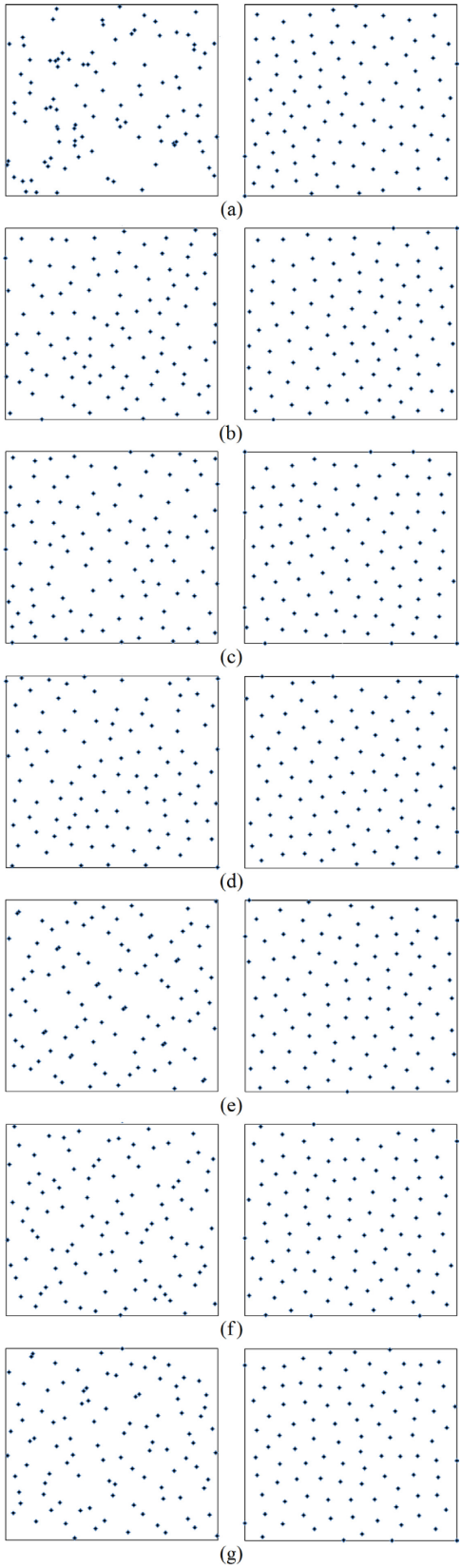


Figure 2.4. The (a) RT, (b) FSCS, (c) RRT, (d) EAR, (e) Sobol, (f) Halton, and (g) Niederreiter on the left and corresponding RBCVT points on the right.

2.5.1 RBCVT Calculation Method

To calculate the RBCVT test cases using a set of generator points, we propose a probabilistic method as follows:

Step1. Determine the initial set of $T = \{t_i\}_{i=1}^{|T|}$ as generators, $t_i \in I$ where $i = 1,...,|T|$.

Step2. Initialize a random border point set of $R = \{r_n\}_{n=1}^{|R|}$ in which $r_n \in H$, where $n = 1,...,|R|$. In addition, the combination of T and R is defined as $TR = T \cup R = \{tr_m\}_{m=1}^{|TR|}$ where $|TR| = |T| + |R|$. Each tr_m has an associated Voronoi cell named V_m.

Step3. Initialize a random background point set of $B = \{b_j\}_{j=1}^{|B|}$ in which $b_j \in (I \cup H)$ where $j = 1,...,|B|$.

Step4. Cluster the B into $|TR|$ cells such that $b_j \in V_m, tr_m = \beta(b_j, TR)$.

Step5. Calculate the centroids of Voronoi regions only for those V_m where the generator belongs to T, denoted by V_i (We do not need to update border points). For the probabilistic approach, (2.8) is simplified to $t_i^* = \frac{\sum_{b_j \in V_i} b_j}{\sum_{b_j \in V_i} 1}$ where ρ is set to a unit value in this application.

Step6. Update the generators, t_i, where $i = 1,...,|T|$ are replaced with the corresponding t_i^*.

Step7. Go to step3 until the stopping criterion is met.

A stopping criterion can be 1) the distortion value between t_i and $t_i^*, i = 1,2,\ldots,|T|$ in each iteration, is reduced to less than a threshold; or 2) a constant number of iterations. Within this study, a constant number of 10 iterations has been selected. This stopping criterion was selected due to its perceived convergence amongst all trial runs of the algorithm. The parameter $|B|$ was set relative to the value of $|T|$, $100 \times |T|$. It has been observed that with 10 iterations and considering $|B| = 100 \times |T|$, the produced RBCVT test cases are in a stable situation and no further iterations were required to more uniformly distribute the generators. Finally, we need to specify how to generate random border

points of *R*. As indicated in Figure 2.3, we considered a set of square cells around *I* as *H* and a random point is inserted in each cell. The number of cells in each side of *I* is selected in accordance with the $|T|$ as $\alpha \times \sqrt{|T|}$ where α is a coefficient which is selected as $\alpha = 2$ based upon an initial empirical exploration. Accordingly, $|\mathrm{R}| = 4 \times \alpha \times \sqrt{|T|}$. Finally, the *h* which is defined as the width of *H*, indicated in Figure 2.3, is equal to a side of a square cell.

2.5.1.1 RBCVT Runtime Analysis

In this section, we discuss the order of computational complexity of the RBCVT algorithm. In each RBCVT iteration, the main computational load is associated with clustering the set B (Step4). Since each b_j is clustered by comparing it to the all members of TR, each b_j clustering complexity grows linearly with $|TR|$ given by $RBCVT(|TR|)_{b_j} \in O(|TR|)$. Obviously, the runtime order of RBCVT is also dependent on $|B|$ and the number of iterations (held constant in this study), hence $RBCVT(|TR|,|B|) \in O(10 \times |TR| \times |B|)$. Since the number of $|B| = 100 \times |T|$ grows linearly with $|T|$; and $|TR| = |T| + |R|$, the previous equation can be simplified as $RBCVT(|T|, |\mathrm{R}|) \in O\left(1000 \times |T| \times \left(|\mathrm{T}| + |\mathrm{R}|\right)\right)$. However, the constant number of 1000 becomes insignificant as $|T|$ grows. As a result, $RBCVT(|T|) \in O(|T|^2 + 4\alpha |T|^{1.5})$. Finally, we need to keep the term with highest order. Therefore, the runtime complexity of RBCVT grows within the order of quadratic time as $RBCVT(|T|) \in O(|T|^2)$.

2.5.2 RBCVT's Runtime Order Reduction (RBCVT-Fast)

The runtime of $O(|T|^2)$ which was calculated for the RBCVT method in the previous section, is the basic calculation method without any algorithmic optimizations. Hence, in this section, we propose an optimized RBCVT calculation method (RBCVT-Fast) using a novel search algorithm to generate test cases with a linear runtime given by $\mathrm{RBCVTFast}\left(|T|\right) \in O\left(|T|\right)$. Although there are some special search algorithms like R*-tree [48], none of them are appropriate for our application. The steps of the new algorithm are similar to the previous section with an additional preprocessing step after Step3 that we call Step3B to prevent the renumbering of steps. Furthermore, Step4's

calculation procedure is updated with a new algorithm.

Each b_j in Step4 is clustered by comparing it to the all members of TR given by $\beta(b_j, TR)$. This process produces a linearly growing runtime with $|TR|$ for clustering b_j, given by $RBCVT(|TR|)_{b_j} \in O(|TR|)$. In contrast, we propose a novel search algorithm, specifically designed for RBCVT, which results in a constant runtime for clustering each b_j. In other words, b_j clustering runtime is independent from the size of TR or T; and we will find the nearest tr_m to the b_j by comparing b_j to a constant number of points in TR.

2.5.2.1 Preprocessing Step

This section explains Step3B of the RBCVT-Fast algorithm that is intended to prepare tr_m, $m = 1,...,|TR|$ for the search algorithm (proposed in the next section). As indicated in Figure 2.5, the preprocessing step involves defining a grid on $H \cup I$, which divides $H \cup I$ into a set of cells, called grid cells. Consequently, each tr_m is placed in one of the cells, which is referred to as the parent cell for that tr_m. All the tr_m points that are in a cell are called child points of that cell.

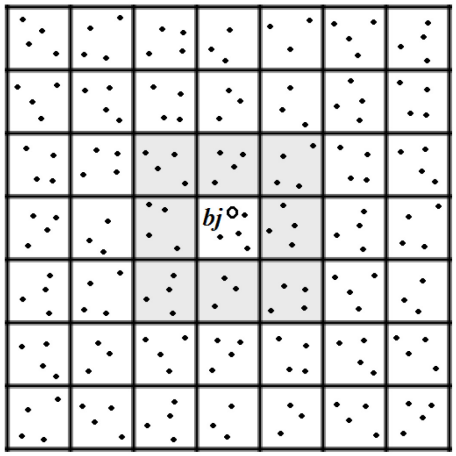


Figure 2.5. A grid divides $H \cup I$ into a set of cells. The points are tr_m, $m = 1,...,|TR|$ and the circle is b_j. Cells in layer one regarding b_j are highlighted, as an example.

In the preprocessing step, we determine each cell's child points and store them in an array. The parent cell of each point is simply determined form the point's coordinates.

The critical parameter in the preprocessing step that affects the runtime of RBCVT-Fast, is C_{avg} which must be a constant for any size of *TR*. We have informally (empirically) observed that C_{avg} = 20, produces the most efficient algorithm with respect to runtime. Having the C_{avg} value, we can calculate the number of cells in each dimension, G_N, given by

$$G_N = Round\left(\sqrt{\frac{|T|}{C_{avg}}}\right) \tag{2.9}$$

Consequently, the total number of cells in a two-dimensional space is $G_N \times G_N$.

2.5.2.2 A Novel Search Algorithm

In this section, a novel search algorithm is discussed which reduces the linear runtime order of clustering b_j to a constant runtime. The main idea behind this search algorithm is that we do not need to compare the b_j with all of the tr_m. As indicated in Figure 2.5, to find the nearest point to b_j, we need to calculate the distance between b_j and the children of the adjacent cells, not all the cells. That is, we need to compare b_j with the children of C_l (a set which contains all the cells in layer *l*), where *l* starts from zero. Layer *l* includes all the cells that have a similar Chebychev distance from the cell with b_j as a child. The highlighted cells in Figure 2.5 are in layer one. This algorithm starts by calculating $tr_{winner} \leftarrow \beta_c\left(b_j, c_{lm}\right)$ for layer zero where each cell of C_l is denoted by c_{lm} (c_{lm} for layer zero is only one cell which is the cell parent of b_j). Then, we check that tr_{winner} is the nearest point to b_j by comparing $dist\left(b_j, tr_{winner}\right)$ with $dist_l\left(b_j, 1\right)$. If $dist\left(b_j, tr_{winner}\right) < dist_l\left(b_j, 1\right)$ then the process is finished and tr_{winner} is the nearest point of *TR* to b_j. Otherwise, we have to compare b_j with the children of layer one's cells and update tr_{winner}, in case we found a closer point to b_j. To reduce the runtime complexity, b_j is only compared with the children of those cells in layer one that $dist_c\left(b_j, c_{lm}\right) < dist\left(b_j, tr_{winner}\right)$. This process will continue until we find the nearest point to b_j. Pseudo code for the proposed search algorithm is indicated in Figure 2.6.

```
begin
  l ← 0    // l denotes the layer number
  MD ← 1   // MD indicates minimum distance
  while dist_l(b_j, l) < MD do
    for each cell in C_l do
      if dist_c(b_j, c_lm) < MD then
        if dist(b_j, β_c(b_j, c_lm)) < MD then
          tr_winner ← β_c(b_j, c_lm)
          MD ← dist(b_j, tr_winner)
        end if
      end if
    end for
    l ← l + 1.
  end while
end
```

Figure 2.6. Pseudo code for the proposed search algorithm utilized in the RBCVT-Fast algorithm.

2.5.2.3 RBCVT-Fast Runtime Analysis

Although the proposed search algorithm does not guarantee that finding the nearest point to b_j is accomplished by comparing b_j with a constant number of points, empirical investigations have indicated that the average number of comparisons stays constant independent from the size of *TR*. Similarly, since |*TR*| is only dependent to |*T*|, the average number of comparisons is independent from |*T*|. Figure 2.7 represents the average number of points and cells compared to b_j in order to find tr_{winner} in a RBCVT-Fast calculation, where a RT test set is utilized as generator points. This graph is presented for different sizes of *T* with respect to the optimized C_{avg} = 20. Since considering other ART and QRT approaches as initial generator points revealed similar results with RT as initial generator points, we only included RBCVT with RT as generator points to avoid duplication.

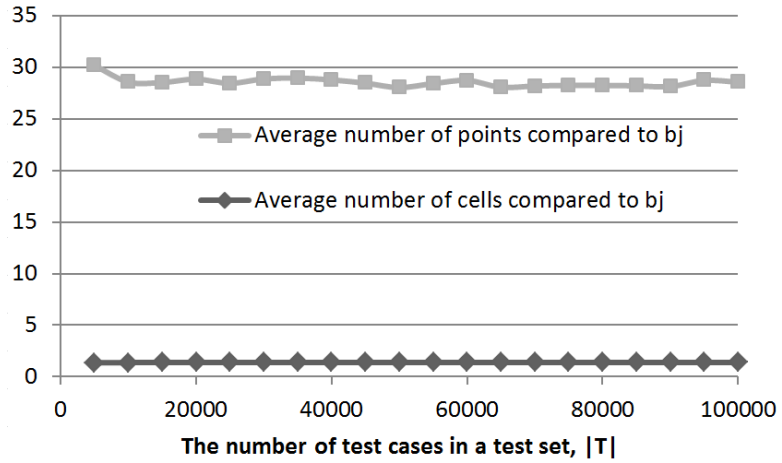


Figure 2.7. Average number of points/cells that are compared to b_j calculating the nearest point of *TR* to b_j in a RBCVT-Fast calculation, where a RT test set is utilized as generators.

As indicated in Figure 2.7, we have produced a search algorithm that, on average, requires a constant number of comparisons to calculate $\beta\left(b_j, TR\right)$ leading to $RBCVTFast(|TR|)_{b_j} \in O(1)$. Another distinction between RBCVT and RBCVT-Fast regarding runtime is the preprocessing step that is included in the RBCVT-Fast. Obviously, the $Preprocessing\left(|TR|\right) \in O\left(10 \times |\mathrm{TR}|\right)$ where 10 indicates the number of iterations. Accordingly, the total RBCVT-Fast runtime order is $O\left(10 \times 1 \times |B| + 10 \times |\mathrm{TR}|\right)$. Similar to the discussion in Section 2.5.1.1, this runtime order can be simplified as $\mathrm{O}\left(1000|\mathrm{T}| + 10|\mathrm{T}| + 10|R|\right) = \mathrm{O}\left(1010|\mathrm{T}| + 40\alpha\sqrt{|T|}\right)$. Since we need to keep the term with highest order, the final runtime of the RBCVT-Fast algorithm is linear given by $RBCVTFast\left(|T|\right) \in O\left(|T|\right)$. The linear runtime is also investigated in empirical runtime analysis section.

2.5.3 Generalization of the RBCVT beyond two dimensions

The concept of the RBCVT is not limited to a two-dimensional hypercube. As defined in Section 2.4 in (2.7), the Voronoi region related to t_i is all the areas that are closer to t_i than any other point. Obviously, we can observe from the definition that the Voronoi region can be of any dimension, having an appropriate *d*-dimensional distance function. The distance function used in this study is Euclidian (l^2-norm) which can be used in any dimension. To analyze the calculation of RBCVT for higher dimensions, we go through

the steps presented in Section 2.5.1 as well as the RBCVT-Fast calculation method as follows:

- The initial generator set (*T*) in Step1 which are the result of other test case generation approaches (RT, ARTs, and QRTs), can be of any dimension since RT, ARTs, and QRTs can produce test cases beyond two dimensions.

- To generate the random border points (*R*) in Step2, we define a set of cells around the *d*-dimensional input space hypercube and then we insert a random point in each cell which is straightforward. The number of cells in each dimension of the input space is selected as $\alpha \times \sqrt[d]{|T|}$. Accordingly, each side of the input space hypercube has $\left(\alpha \times \sqrt[d]{|T|}\right)^{d-1}$ cells, since the dimension of each side of a *d*-dimensional unit hypercube is *d-1*. Finally, a *d*-dimensional unit hypercube has $2 \times d$ sides leading to the following equation for the number of cells which covers all borders of the input space.

$$|R| = 2 \times d \times \left(\alpha \times \sqrt[d]{|T|}\right)^{d-1}. \tag{2.10}$$

- The background points (*B*) in Step3, are easy to generalize to higher dimensions, since we only need *d*-dimensional random numbers.

- In Step3B regarding the preprocessing step of the RBCVT-Fast, we can define the grid on *d* dimensions rather than a two-dimensional hypercube. Then each *d*-dimensional tr_m can be assigned to a cell of the grid. In addition, G_N for the *d*-dimensional hypercube can be calculated by

$$G_N = Round\left(\sqrt[d]{\frac{|T|}{C_{avg}}}\right). \tag{2.11}$$

- In the non-optimized RBCVT approach, Step4 is easy to calculate in any dimension as we compute the distance of each b_j with all $tr_m, m = 1,2,\ldots,|TR|$ with the *d*-dimensional Euclidian distance function. The algorithm of this step in the RBCVT-Fast is exactly equal to the pseudo code presented in Figure 2.6. The only changes are the generalization of $dist_l\left(b_j, l\right)$, $dist_c\left(b_j, c_{lm}\right)$, and $\beta_c\left(b_j, c_{lm}\right)$ into *d*

dimensions. All of these functions require a *d*-dimensional Euclidian distance function which is available.

- Finally, Steps 5-7, including the calculation and updating of the centroids (t_i^*) can be calculated for any dimension.

2.5.3.1 Runtime Analysis of *d*-dimensional RBCVT

Looking precisely to the non-optimized RBCVT algorithm, one can observe that the only process dependent to the dimension is the distance function, and its runtime changes linearly with *d*. The number of comparisons is independent from *d* leading to $RBCVT\left(\mathrm{d},|T|\right)\in O\left(d\times|T|^2\right)$. This indicates a linear increase in $RTime\left(RBCVT\right)$ as *d* grows.

In the contrary, the order of $RTime\left(RBCVTFast\right)$ is not linear with *d* since the number of required comparisons grows as *d* increases. The increasing number of cells in layer *l* as *d* increases is the cause of this issue. The number of cells in layer *l* increases exponentially as *d* grows leading to exponential increase in the number of distance comparisons. In addition, each distance comparison runtime grows linearly with *d*. As a result, the order of $RTime\left(RBCVTFast\right)$ is given by $RBCVTFast\left(\mathrm{d},|T|\right)\in O\left(d\times E^d\times|T|\right)$ where E is a constant. Note that in a given *d,* $RTime\left(RBCVTFast\right)$ is still linear regarding $|T|$.

Although the runtime complexity of RBCVT-Fast with respect to *d* is higher than the non-optimized RBCVT, the runtime complexity for RBCVT-Fast is lower with respect to $|T|$ than the non-optimized RBCVT. Combining these two observations results in $RTime\left(RBCVTFast\right)\le RTime\left(RBCVT\right)$ for any $|T|$ and *d.* That is, the number of comparisons in the RBCVT-Fast is less than or equal to the non-optimized RBCVT algorithm. According to (2.11), G_N reduces when *d* increases with constant C_{avg} and $|T|$, leading to an increasing $\frac{RTime\left(RBCVTFast\right)}{RTime\left(RBCVT\right)}$. With G_N=1, the RBCVT and RBCVT-Fast are exactly equal since there is only one cell in the hypercube. In G_N=3, the runtime of both approaches are similar, since the RBCVT-Fast uses layers 0 and 1 on average to find the nearest point. As G_N increases, the runtime effectiveness of the

RBCVT-Fast grows compared to the non-optimized RBCVT algorithm. To summarize, $RTime(RBCVTFast) \ll RTime(RBCVT)$ when $G_N \gg 3$ leading to $\sqrt[d]{\frac{|T|}{C_{avg}}} \gg 3$ which is concluded from (2.11). Therefore, when the number of test cases is large enough, RBCVT-Fast algorithm is more efficient than non-optimized RBCVT algorithm regarding time complexity.

2.6 Experimental Frameworks

The conducted study to investigate the effectiveness of RBCVT against the ART and QRT methods is described in this section. We have designed two experimental frameworks: a simulation based and a mutant based software testing framework. The simulation framework utilizes three failure patterns derived from empirical studies [13]–[17] investigating defect types. The mutant based software testing framework simulates defects in software by producing mutants within the code in a systematic fashion [49]. For the mutant based software testing framework, we utilize the Briand and Arcuri [49] framework; this framework has been accepted via publication as a valuable mechanism for empirically exploring such mechanisms. This framework is based on 11 short mathematical programs that appear in the ART literature [17]. Both frameworks require an effectiveness measure to evaluate the results which is discussed in the following section.

2.6.1 Testing Effectiveness Measure

There are three well-known testing effectiveness measures, E-measure, P-measure, and F-measure. The E-measure is defined as the expected number of detected failures in a series of tests. Assuming the probability of a test case to detect a failure is θ, similar to a random test case, then the E-measure and its standard deviation are [50]

$$Emeasure = |T| \times \theta, \tag{2.12}$$

$$std = \sqrt{\theta |T| (1-\theta)}. \tag{2.13}$$

The P-measure is defined as the probability of at least one failure being detected within a test set. Considering the number of test sets as M_t and the number of test sets that detect at least one failure as M_{fault}, the P-measure can be estimated as M_{fault} / M_t. In addition,

in RT, the P-measure is equal to [50]

$$Pmeasure = 1 - (1-\theta)^{|T|}. \tag{2.14}$$

The standard deviation associated with the calculation of a P-measure for RT can be approximated by [50]

$$std \approx \sqrt{(1-\theta)^{|T|} - (1-\theta)^{2|T|}}. \tag{2.15}$$

The last testing effectiveness measure is the F-measure which is defined as the number of test cases required to detect the first failure within the input domain. Chan et al. [26] have indicated that for RT the expected value of the F-measure is equal to θ^{-1}. The sampling distribution of the P-measure and the E-measure can be approximated by the normal distribution [50], whereas the probability distribution of the F-measure is geometric [50].

The main question that should be answered is: which of these measures best characterizes software testing? Since the software testing trend is toward automating the process, selecting a measure that best represents the operation of an Automated Testing System (ATS) is essential. When we consider the "desirable" aspects of automated software testing with respect to RT, ART, QRT, or RBCVT, it does impose certain constraints on the measurement process that must be adhered to:

- ATS is intrinsically an automated technique at least on the test case generation side. This implies that the traditional incremental cost of manual production of a new, additional test case is minimized. ATS is characterized by: 1) a tester selecting an arbitrary large number of test cases to be produced; and 2) the ATS system producing the required volume of test cases.

- Test case generation often seeks to generate values with a specific purpose, while we can generate truly random values and exercise them against the entire system. The huge dimension of the input space for modern software systems tends to imply that this "scatter gun" approach is ineffective. Instead, the tester will often have a specific testing objective and will attempt to generate a specific set of test cases under specific circumstances that answer this question. That is, the tester tends to test aspects of the system or sub-components of the system rather than blindly "attacking" the entire system. For example, automated security testing investigates an aspect of the system, and automated unit testing explores a sub-component.

Accordingly, the tester will require a large volume of test cases, possessing limited dimensions, which are cost effective for an automated testing process.

- These large volumes of test sets are automatically applied to the system under test and the “outputs” from the system are automatically captured. The system under test is normally placed into a known state before each execution commences. The large volume of test cases implies that manual application of the test data is not a realistic option.

- This input process results in large volumes of test results, again implying that the manual examination of every test result is prohibited by cost. Instead, two options are commonly deployed: 1) A Test Oracle is constructed. The test oracle typically has a simplified description of a defect. Does the system crash or not is an example of such a description. Here each crash is considered a "defect". The oracle either stops after finding the first crash or collects all of the crashes. Data about the crashes is presented to the tester for analysis. If the oracle collects multiple crashes, the system has no mechanism to understand if these crashes have the same root cause or are in fact independent. The tester may select to only investigate a subset of these multiple crashes to avoid excessive, potentially redundant (when crashes are in fact dependent) costs. 2) The output is investigated manually as a single integrated entity. Here the test results, or shorter proxies of the results, are sent to a log file or other recording mechanism. The tester inspects this mechanism after all the test runs are finished. Here the tester is looking for output values that look anomalous. Again, the tester may select one or more test results to explore more closely, However, the number of test results explored is always small to ensure a cost effective process.

The above description of ATS is in correspondence with many ATS systems reported in the literature, including [7], [51], [52]. Accordingly, it is believed that this process is well characterized by the E- or the P-measure rather than F-measure. That is, the incremental viewpoint of the F-measure is not supported by the operation of these automated testing systems [7], [51], [52] in the operational profile discussed. Since in software testing, failure areas tend to be clustered [13], [14], [24], [53], detecting multiple failures are often redundant as it is indicative of multiple test cases discovering the same defect. This argument strongly suggests the use of the P-measure over the E-measure. Therefore, the P-measure is utilized in this study as an appropriate effectiveness measure for automated

software testing.

Chen et al. [50] demonstrate that the F-measure has better statistical power than the P-measure. However, this "performance difference" tends to zero as the number of measurements tends to infinity. It is believed that the above analysis effectively implies that this difference is essentially zero at the number of measurements utilized within this chapter.

2.6.2 Parameters of Test Case Generation Methods

A number of parameters are associated with each ART algorithm which are considered constant through all the experiments. We selected the value of these parameters as recommended in their respective works. The k in FSCS method, representing the number of randomly selected candidates, is held constant at $k = 10$ based on the recommendation of Chen et al. [25]. Similarly, the coverage ratio in RRT method is considered constant at 1.5 due to recommendation of Chan et al. [53]. The EAR method [19] has several parameters regarding the evolutionary approach which are set to identical values to those reported in the original work [19]. The k (population size) has been set to 20 and the probability of crossover is set at 0.6. Furthermore, the probability of mutation is considered as 0.1, the size of the mutation was set at 0.01, and the stopping criterion is set to the constant number of 100 iterations. The parameters associated with RBCVT are in accordance with the values discussed in Section 2.5.1, the number of background points is set to $100 \times |T|$ and the number of RBCVT iterations is equal to 10 for all the tests.

2.6.3 Simulation Framework

For the simulation framework, we will introduce the utilized failure patterns, failure rate associated with each failure pattern, the number of test cases in each test set, and the number of test sets. These features are discussed in the next two sections.

2.6.3.1 Failure Patterns and Failure Rates

To be able to evaluate test case generation methods, we need to consider some parts of the input domain as a failure area, where a failure is produced when a test case is placed in this area. Several works have performed an empirical investigation through failure patterns within the input domain [13]–[17]. White and Cohen [15] indicated that failures usually occur on or near the boundary of (sub-) domains. As a result, failure areas form

types of strip patterns since domain boundaries form lines or hyper planes. Ammann and Knight [13] explain that failure regions seem to be locally continuous. They present two-dimensional empirical failure patterns that possess similarities to rectangular geometry. Similarly, Finelli [14] describes that there are continuous regions, called error crystals that produce failures. Bishop [16] also explains continuous failure regions that are much more angular and elongated than a pure "blob" [17]. Schneckenburger and Mayer [17] have analyzed the failure area geometry in a systematic way using three numerical programs, each possessing a two-dimensional input space. They presented strip faulty patterns for all three programs under test. Therefore, significant empirical evidence exists that failure areas are clustered into a contiguous region within the input domain and that they produce error crystals or failure regions.

While we cannot generalize one software failure pattern to others, researchers have empirically indicated common characteristics between failure patterns. Accordingly, Chan et al. [24] have introduced three common types of failure patterns, shown in Figure 2.8 (the block, strip and point failure patterns). We have selected these patterns as a testing framework, since the empirical studies support the use of these patterns as an approximation to real software failures. Although these failure patterns are not real, these patterns are believed to best represent multiple clustered values in the input domain, which, in general, imply a single root cause failure.

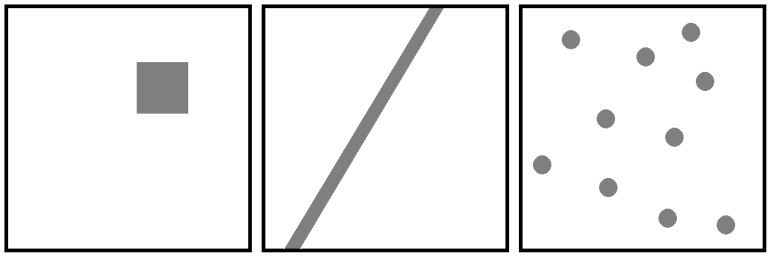


Figure 2.8. Typical two-dimensional failure patterns: (a) block, (b) strip, and (c) point failure patterns.

The main parameter associated with each pattern is a failure rate (θ) which is the total failure area divided by the total area of the input domain. In this chapter, failure rates of $\theta = 10^{-2}, 10^{-3}, 10^{-4}$, and 10^{-5} have been considered as a basis to analyze testing strategies effectiveness. In the software testing literature [19], [32], failure rates between 10^{-2} and 10^{-3} are usually investigated, whereas in real life applications the failure rates

may be lower. Considering the fact that the average programmers introduce five to ten defects per Kilo Line Of Code (KLOC) [2], θ is certainly nonzero. However, no reliable industrial information exists on θ. Hence, we include the failure rates of 10^{-4} and 10^{-5} to explore a wider range of values.

Although the implementation of these three failure patterns is straightforward, implementation details are included for the sake of completeness. The block pattern is generated by randomly choosing a point in I and then a square is constructed around this point with respect to the failure rate. Due to the section of the random point near to the boundaries of I, the constructed block pattern may not fit within I. In this situation, this pattern is disregarded and another random point is selected until a valid block pattern is generated. The strip pattern is generated using a random point in I and a random angle associated with a line passing over the selected random point. The width of the strip pattern is calculated according to the failure rate. This strip pattern generation method is different from the method introduced by Chen et al. [50], whereby one point is selected on the vertical boundary and another point on the horizontal boundary of I. Then, the strip pattern is generated by connecting the two points and calculating the width of the line using θ. Unfortunately, we observed that this implementation does not produce a uniform distribution of strip patterns - with an excessive concentration of points near the boundaries compared to the middle of I. To generate the point pattern, 10 random points were selected within I. A circular area is constructed around each point so that the sum of these circular areas is equal to failure rate. Similar to the block pattern, if a circular area is not within I, the associated random point is disregarded and another random point is selected.

In short, the block and point patterns are in-line with those used in the literature [19], [32], [34], [50]; and the strip pattern is redefined to overcome traditional limitations and produce a uniform distribution of the strip pattern.

2.6.3.2 Number of Tests

Due to the random nature of test case generation methods, we generated $M_t = 100$ distinct test sets for RT, FSCS, RRT, EAR, and accordingly RBCVT to evaluate the effectiveness of each approach using the P-measure. Therefore, a P-measure is evaluated using 100 tests for a specific failure pattern. In addition to test set generation, the failure patterns are also generated randomly. Hence, we generated $M_f = 10,000$ random failure

patterns leading to 10,000 P-measure results which are normally distributed [50] between zero and one. Therefore, $M_f = 10,000$ statistics are used to evaluate the mean and standard deviation of the normally distributed P-measure for each approach, at each failure rate, and with each of the three failure patterns.

QRT methods are deterministic and hence each method produces a unique test set. Therefore, to draw statistical analysis with the same population size, for each QRT method, we generated a sequence of test cases where the length of this QRT sequence is M_t times larger than the test set size. Then, we split this sequence into M_t test sets which result in distinct test sets. So all the approaches have been tested using $M_f = 10,000$ P-measure results, each calculated by $M_t = 100$ measurements.

In addition to failure pattern type and θ (10^{-2},10^{-3},10^{-4}, and 10^{-5}), to evaluate a P-measure, we need to set the number of test cases in each test set ($|T|$). The best $|T|$ to analyze the test case generation approaches using the P-measure, is the worst case in terms of the standard error which can be estimated as

$$SE = \frac{std}{\sqrt{M_t}}. \tag{2.16}$$

Since M_t is a constant number, worst case SE leads to maximizing the standard deviation. According to Chen et al. [50], the maximum standard deviation of P-measure calculation is 0.5. Solving (2.15) as $std = 0.5$, results in $|T|$ based on θ as follows:

$$|T| = \frac{\log(0.5)}{\log(1-\theta)}. \tag{2.17}$$

Since $\theta = 10^{-2}$,10^{-3},10^{-4}, and 10^{-5} have been chosen for the experimental test, the respective values for $|T|$ are 68.97 (69), 692.80 (693), 6931.12 (6931), 69314.37 (69314). Since $|T|$ is an integer value, the rounded values are given in the brackets. Finally, all the generated test cases are within *I* and every test case consists of a floating point number, with double precision, for each dimension.

2.6.4 A Mutant Based Software Testing Framework

To evaluate the proposed RBCVT approach on a testing framework which utilizes

independently-produced programs, we selected the mutant based software testing framework introduced by Briand and Arcuri [49]. This framework is outlined in detail in Section 4 of [49]. For the sake of completeness, we present a summary of the main features of this framework. This work utilizes 11 programs, written in Java, which implements basic mathematical functions that appear in the ART literature [17]. We directly utilized their source code without any modification. Their framework utilizes mutation analysis to produce a large number of faults in a systematic fashion [49]. They produced 3,727 mutants for the 11 programs using muJava [54], [55]. Further, in [49], the P-measure is utilized to evaluate these mutants against RT and ART test sets, where the size of test sets varies between 1 and 50.

This framework assumes an input space of each program, an integer value in the range of $[0,2^{24/d}-1]$ for each dimension (d). This leads to 2^{24} input possibilities for each program. The framework first measures each mutants failure rate by testing all possible 2^{24} states, so they could measure failure rates as low as 2^{-24}. Then, those mutants that revealed no failure or had the failure rate over 0.01 were removed. Therefore, they kept 780 appropriate mutants with $2^{-24} \le \theta \le 0.01$.

In this study, we use these 780 mutants to test the effectiveness of the proposed test case generation approach. Since we assume that we do not know the failure rate of the programs under the test, we apply four test set sizes including |T|=10, 20, 50, and 100 to each mutant to evaluate the effectiveness of each test case generation approach. Accordingly, the P-measure is evaluated for each test case generation approach for discussed test set sizes. To evaluate a P-measure, we tested each mutant using 100 distinct test sets and then, the average over all the mutants is calculated as a P-measure. To draw a statistical analysis, we repeated this P-measure evaluation 100 times leading to 100 statistics that are used to evaluate the mean and standard deviation of the normally distributed P-measure [50] for each approach, at each test set size. To draw statistical analysis with the same population size for QRT methods, we utilized a similar procedure as described in the simulation pattern where a longer sequence of QRT test cases is split to generate distinct test sets.

This process leads to the execution of over 78 billion test cases which took more than a month on an Intel dual-core Processor E6300 (2.8GHz) with 8GB of RAM.

2.7 Experimental Results and Discussion

2.7.1 Formal Analysis

Since P-measure values are normally distributed [50], Tables 2.2-2.5 present statistical parameters reflecting the effectiveness of RT, ARTs, QRTs and the corresponding results after the RBCVT process. In addition, the following parameters were calculated:

1) A test of statistical significance (z-test, one-tailed, our working hypothesis is that RBCVT will produce superior results) with a conservative type I error of 0.01; and

2) An effect size (Cohen's method [56], [57]) which indicates "size" discrepancy between two statistical populations given by

$$effect\ size = \frac{\mu_2 - \mu_1}{\sqrt{\frac{(n_2 - 1)std_2^2 + (n_1 - 1)std_1^2}{n_2 + n_1}}} \tag{2.18}$$

where μ, std, and n represent the mean, the standard deviation, and the number of elements within the populations, respectively. In this study, a positive value of effect size represents the size of the improvement that has been achieved by applying the RBCVT process. Cohen [56]–[58] defines the standard value of an effect size as small (0.2), medium (0.5), and large (0.8). Effect size can also be interpreted as the average percentile standing which indicates the relative position of the two populations. Similarly, effect sizes can be interpreted in terms of the percent of the non-overlapped portion of the populations. Corresponding values are presented in Table 2.1.

Table 2.1. Cohen's effect size description (large, Medium, and Small) as well as corresponding values for percentile standing and percent of non-overlapped portion of two populations.

Cohen's Description	Effect Size	Percentile Standing	Percent of Non-overlap
	2.0	97.7	81.1%
	1.5	93.3	70.7%
	1.0	84	55.4%
Large	0.8	79	47.4%
Medium	0.5	69	33.0%
Small	0.2	58	14.7%
	0.0	50	0.0%

2.7.2 Block Pattern Simulation Results

Table 2.2 indicates the testing effectiveness of all the studied approaches and the corresponding results after the RBCVT process was applied with respect to the block failure pattern. This table demonstrates that performing the RBCVT process on the outputs of other methods has a positive effect on the P-measure, since RBCVT consistently provides statistically significant improvement. The amount of improvement in terms of effect size is larger than the highest Cohen's description (Large) in most of the cases, only RRT at $\theta = 10^{-4}, 10^{-5}$ and EAR at $\theta = 10^{-5}$ have effect size between large and medium.

Table 2.2. The P-measure testing effectiveness mean and standard deviation for all approaches including the corresponding results after the RBCVT process as well as effect size, Z-score, and significance value with respect to block pattern.

$\theta, \lvert T \rvert$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10^{-2}, 69	RT	0.4972	0.0565	0.7484	0.0468	4.8423	444.91	0.0000
	Sobol	0.5694	0.0627	0.7668	0.0432	3.6657	314.83	0.0000
	Niederreiter	0.5833	0.0568	0.7705	0.0432	3.7113	329.66	0.0000
	Halton	0.6118	0.0531	0.7742	0.0424	3.3799	305.86	0.0000
	FSCS	0.5953	0.0512	0.7629	0.0434	3.5296	327.00	0.0000
	RRT	0.6021	0.0527	0.7623	0.0435	3.3164	304.20	0.0000
	EAR	0.5951	0.0534	0.7538	0.0442	3.2367	297.28	0.0000
10^{-3}, 693	RT	0.5006	0.0502	0.7016	0.0460	4.1721	400.12	0.0000
	Sobol	0.5769	0.0554	0.7121	0.0448	2.6832	244.12	0.0000
	Niederreiter	0.5612	0.0584	0.7122	0.0452	2.8918	258.69	0.0000
	Halton	0.5863	0.0504	0.7127	0.0452	2.6399	250.74	0.0000
	FSCS	0.6242	0.0484	0.7096	0.0454	1.8200	176.60	0.0000
	RRT	0.6407	0.0478	0.7100	0.0452	1.4891	145.01	0.0000
	EAR	0.6420	0.0484	0.7092	0.0453	1.4335	138.86	0.0000
10^{-4}, 6931	RT	0.5000	0.0494	0.6830	0.0465	3.8154	370.49	0.0000
	Sobol	0.5442	0.0595	0.6850	0.0460	2.6474	236.61	0.0000
	Niederreiter	0.5687	0.0525	0.6850	0.0463	2.3497	221.41	0.0000
	Halton	0.5908	0.0497	0.6847	0.0461	1.9564	188.72	0.0000
	FSCS	0.6328	0.0477	0.6915	0.0467	1.2429	122.91	0.0000
	RRT	0.6555	0.0473	0.6918	0.0461	0.7774	76.72	0.0000
	EAR	0.6456	0.0481	0.6911	0.0461	0.9684	94.87	0.0000
10^{-5}, 69314	RT	0.5004	0.0497	0.6764	0.0460	3.6801	354.54	0.0000
	Sobol	0.5971	0.0556	0.6847	0.0465	1.7100	157.51	0.0000
	Niederreiter	0.5937	0.0511	0.6835	0.0466	1.8349	175.59	0.0000
	Halton	0.5924	0.0492	0.6830	0.0465	1.8929	184.29	0.0000
	FSCS	0.6356	0.0483	0.6854	0.0482	1.0312	102.96	0.0000
	RRT	0.6604	0.0474	0.6873	0.0455	0.5779	56.61	0.0000
	EAR	0.6496	0.0476	0.6853	0.0457	0.7668	75.15	0.0000

Comparing the amount of improvement (effect size) among all approaches in Table 2.2, one can observe that the largest RBCVT improvement belongs to the RT for all failure rates. In contrast, no individual method has the smallest increase in effectiveness regarding the effect size, the EAR has the smallest improvement for $\theta = 10^{-2}$ and 10^{-3}; and RRT for $\theta = 10^{-4}$ and 10^{-5}. Figure 2.9 indicates the improvement of each approach after the RBCVT process comparing to the effectiveness of test cases used as inputs to the RBCVT process (effect size) with respect to block pattern at each failure rate. In this

figure, in all methods, the level of changes before and after the RBCVT process is decreasing as the failure rate decreases.

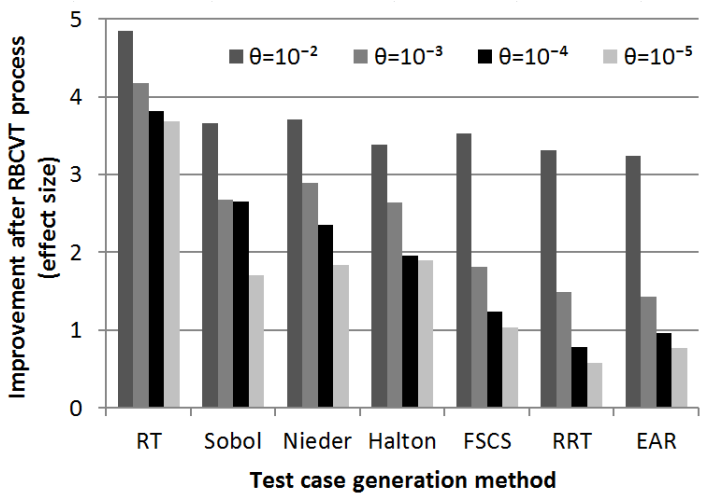


Figure 2.9. Improvement of test case generation methods with respect to RBCVT process at different failure rates regarding the block failure pattern.

In Table 2.2, the mean values of the P-measures appear dissimilar for the different approaches; whereas the corresponding results after the application of the RBCVT process represents a sizable reduction of the variation between these values. Therefore, for comparison of RBCVT, as a single method, against all other approaches, we assume the average RBCVT results as the performance of the RBCVT approach. Figure 2.10 represents the effect size of the testing effectiveness at each strategy against RT in the block pattern simulations. Contrasting RBCVT against FSCS, RRT, EAR, Sobol, Halton, and Niederreiter, this figure highlights the increased efficiency of RBCVT regarding the block pattern. Another conclusion from this figure is that all of the testing methods outperformed RT at every failure rate with respect to the block pattern.

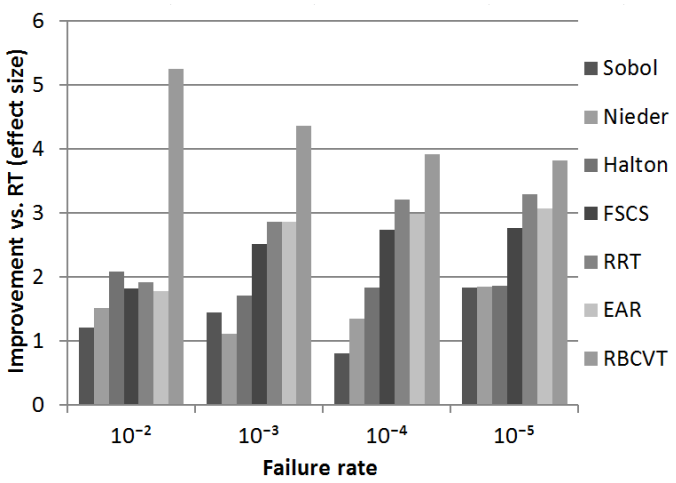


Figure 2.10. P-measure testing effectiveness for block pattern simulations of FSCS, RRT, EAR, RBCVT, Sobol, Niederreiter, and Halton against the RT.

2.7.3 Strip Pattern Simulation Results

Testing effectiveness results regarding the strip failure pattern are shown in Table 2.3. The results demonstrate that for $\theta = 10^{-2}$, RBCVT is statistically significantly superior to all approaches. In contrast, the results for other failure rates suggest similar performance between each approach and the corresponding results after the RBCVT. Although there are differences between the P-measure results of the RBCVT and other approaches at $\theta = 10^{-3}, 10^{-4}$, and 10^{-5}, the results cannot be compared since the level of significance values do not indicate a significant difference between the results in most of the cases.

Table 2.3. The P-measure testing effectiveness mean and standard deviation for all approaches including the corresponding results after the RBCVT process as well as effect size, Z-score, and significance value with respect to strip pattern.

$\theta, \lvert T\rvert$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10^{-2}, 69	RT	0.4992	0.0505	0.5192	0.0502	0.3954	39.41	0.0000
	Sobol	0.5214	0.0492	0.5233	0.0502	0.0382	3.86	0.0001
	Niederreiter	0.5165	0.0510	0.5211	0.0503	0.0910	9.05	0.0000
	Halton	0.5172	0.0498	0.5222	0.0494	0.1017	10.13	0.0000
	FSCS	0.5162	0.0505	0.5220	0.0502	0.1168	11.64	0.0000
	RRT	0.5161	0.0503	0.5228	0.0506	0.1323	13.26	0.0000
	EAR	0.5067	0.0503	0.5204	0.0497	0.2729	27.13	0.0000
10^{-3}, 693	RT	0.4997	0.0502	0.5061	0.0501	0.1281	12.80	0.0000
	Sobol	0.5062	0.0499	0.5065	0.0502	0.0054	0.54	0.2929
	Niederreiter	0.5047	0.0501	0.5069	0.0502	0.0450	4.51	0.0000
	Halton	0.5052	0.0495	0.5065	0.0503	0.0248	2.50	0.0062
	FSCS	0.5063	0.0506	0.5069	0.0499	0.0110	1.09	0.1372
	RRT	0.5063	0.0502	0.5068	0.0505	0.0102	1.02	0.1542
	EAR	0.5056	0.0500	0.5068	0.0500	0.0235	2.35	0.0095
10^{-4}, 6931	RT	0.4993	0.0499	0.5024	0.0498	0.0632	6.31	0.0000
	Sobol	0.5027	0.0502	0.5025	0.0501	-0.0043	-0.43	0.3327
	Niederreiter	0.5012	0.0499	0.5030	0.0499	0.0355	3.55	0.0002
	Halton	0.5022	0.0498	0.5024	0.0501	0.0052	0.52	0.2999
	FSCS	0.5020	0.0495	0.5016	0.0503	-0.0094	-0.94	0.1727
	RRT	0.5023	0.0506	0.5018	0.0501	-0.0101	-1.01	0.1573
	EAR	0.5022	0.0501	0.5026	0.0497	0.0081	0.81	0.2101
10^{-5}, 69314	RT	0.5001	0.0501	0.5007	0.0499	0.0120	1.20	0.1156
	Sobol	0.5015	0.0500	0.5010	0.0503	-0.0085	-0.85	0.1967
	Niederreiter	0.5007	0.0494	0.5017	0.0498	0.0213	2.14	0.0161
	Halton	0.5007	0.0499	0.5009	0.0509	0.0040	0.40	0.3436
	FSCS	0.5014	0.0503	0.5008	0.0508	-0.0106	-1.06	0.1439
	RRT	0.5019	0.0505	0.5028	0.0505	0.0192	1.92	0.0277
	EAR	0.5017	0.0502	0.5010	0.0484	-0.0148	-1.46	0.0726

The magnitude of improvement for the strip pattern at $\theta = 10^{-2}$ is lower than for the block pattern testing effectiveness results since the effect size has been reduced by around an order of magnitude on average. Comparing the amount of improvement among all the approaches in Table 2.3, again the largest improvement belongs to RT for $\theta = 10^{-2}$ and 10^{-3}. To highlight some strip pattern features regarding the RBCVT approach, Figure 2.11 is presented which indicates the effect sizes between each approach's effectiveness result and corresponding result after the RBCVT process. Figure 2.11 indicates that the impact of the RBCVT process is reducing as the failure rate decreases in most of the cases. This fact as well as the results for $\theta = 10^{-3}$,10^{-4}, and 10^{-5} suggest that the impact of the RBCVT approach, for strip patterns, tends to zero as the failure rate tends to zero.

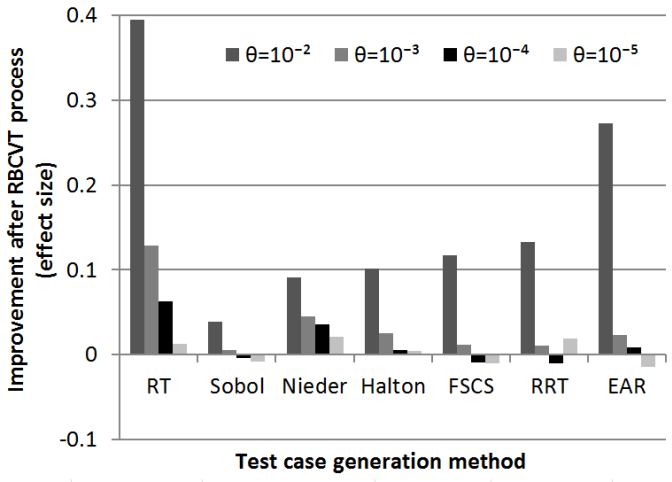


Figure 2.11. Improvement of test case generation methods with respect to the RBCVT process at different failure rates regarding the strip failure pattern.

Similar to the block pattern, the strip pattern testing effectiveness results after the application of the RBCVT represents a sizable reduction of the variation among these values compared to the effectiveness of the input test cases to the RBCVT process. Therefore, we again consider the average RBCVT results as the performance of the RBCVT approach creating the possibility of comparing it against all of the test case generation methods. Accordingly, all of the approaches have been compared against RT, these results are provided in Figure 2.12. In this figure, one can observe the decreasing trend of testing effectiveness against RT as the failure rate reduces. This leads to similar effectiveness for RT with other approaches with respect to the strip pattern at very low failure rates like 10^{-5}; this is not true for the block pattern. This can be explained by the intrinsic difference between the strip and the block pattern: as the failure rate decreases the width of a strip pattern reduces, as its length is constant, whereas in the block pattern both dimensions reduce together. Therefore, the similarity between block and strip pattern decreases as the failure rate reduces leading to less testing effectiveness for strip patterns.

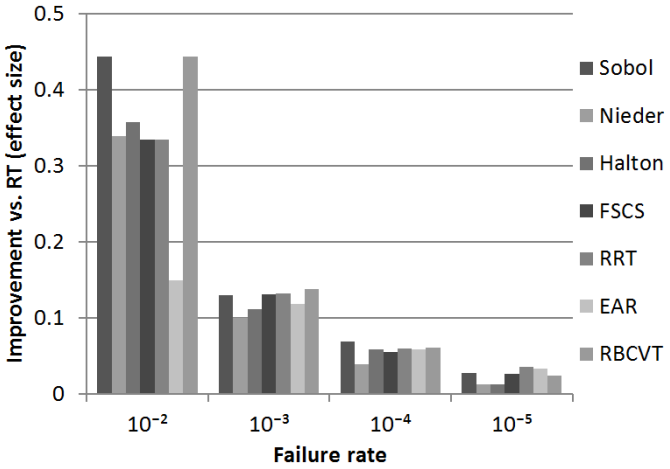


Figure 2.12. P-measure testing effectiveness for strip pattern simulations of FSCS, RRT, EAR, RBCVT, Sobol, Niederreiter, and Halton against the RT.

2.7.4 Point Pattern Simulation Results

Point pattern simulations yield results as indicated in Table 2.4. The presented results suggest an improvement comparing the P-measure results after the RBCVT process was applied. Again the improvement in testing effectiveness, after the RBCVT process was applied, are lower than the corresponding block pattern results. However, in contrast with strip pattern, the RBCVT is statistically significantly superior to all approaches at all failure rates. In addition, the impact of the RBCVT procedure on the test case generation effectiveness regarding point pattern, as indicated by the effect sizes in Table 2.4, are larger than the equivalent results for the strip pattern.

Table 2.4. The P-measure testing effectiveness mean and standard deviation for all approaches including the corresponding results after the RBCVT process as well as effect size, Z-score, and significance value with respect to point pattern.

$\theta, \lvert T\rvert$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10^{-2}, 69	RT	0.4970	0.0506	0.5247	0.0500	0.5509	54.74	0.0000
	Sobol	0.5083	0.0507	0.5242	0.0501	0.3172	31.54	0.0000
	Niederreiter	0.5064	0.0506	0.5243	0.0499	0.3552	35.28	0.0000
	Halton	0.5080	0.0502	0.5246	0.0493	0.3348	33.17	0.0000
	FSCS	0.4956	0.0506	0.5230	0.0496	0.5470	54.17	0.0000
	RRT	0.4927	0.0507	0.5234	0.0505	0.6081	60.69	0.0000
	EAR	0.4788	0.0507	0.5219	0.0499	0.8581	85.11	0.0000
10^{-3}, 693	RT	0.4994	0.0502	0.5163	0.0500	0.3378	33.73	0.0000
	Sobol	0.5054	0.0516	0.5164	0.0496	0.2171	21.30	0.0000
	Niederreiter	0.5095	0.0509	0.5166	0.0496	0.1420	14.02	0.0000
	Halton	0.5113	0.0498	0.5162	0.0503	0.0984	9.89	0.0000
	FSCS	0.5072	0.0496	0.5158	0.0494	0.1740	17.37	0.0000
	RRT	0.5051	0.0497	0.5166	0.0497	0.2319	23.19	0.0000
	EAR	0.5061	0.0502	0.5164	0.0498	0.2065	20.56	0.0000
10^{-4}, 6931	RT	0.4993	0.0495	0.5130	0.0500	0.2759	27.74	0.0000
	Sobol	0.5080	0.0510	0.5131	0.0502	0.1009	10.01	0.0000
	Niederreiter	0.5107	0.0504	0.5131	0.0500	0.0482	4.80	0.0000
	Halton	0.5112	0.0503	0.5128	0.0499	0.0305	3.04	0.0012
	FSCS	0.5109	0.0500	0.5139	0.0502	0.0604	6.05	0.0000
	RRT	0.5096	0.0501	0.5140	0.0503	0.0870	8.71	0.0000
	EAR	0.5100	0.0496	0.5136	0.0496	0.0728	7.29	0.0000
10^{-5}, 69314	RT	0.5006	0.0500	0.5136	0.0482	0.2659	26.12	0.0000
	Sobol	0.5111	0.0499	0.5128	0.0494	0.0340	3.38	0.0004
	Niederreiter	0.5114	0.0496	0.5126	0.0499	0.0233	2.34	0.0097
	Halton	0.5112	0.0496	0.5125	0.0494	0.0264	2.63	0.0043
	FSCS	0.5111	0.0500	0.5122	0.0485	0.0225	2.21	0.0134
	RRT	0.5114	0.0499	0.5126	0.0502	0.0253	2.54	0.0056
	EAR	0.5115	0.0508	0.5126	0.0522	0.0205	2.08	0.0189

In Table 2.4, one can observe that in contrast with the block and strip patterns; the maximum enhancement in testing effectiveness after the RBCVT process, does not belong to the RT for all failure rates. EAR has the largest improvement for $\theta = 10^{-2}$; and RT for other failure rates. To further characterize the point pattern results regarding the RBCVT procedure, Figure 2.13 provides a graphical representation of the effect sizes in Table 2.4. This figure indicates that the impact of the RBCVT process regarding the point pattern has a reducing trend as the failure rate reduces for all approaches.

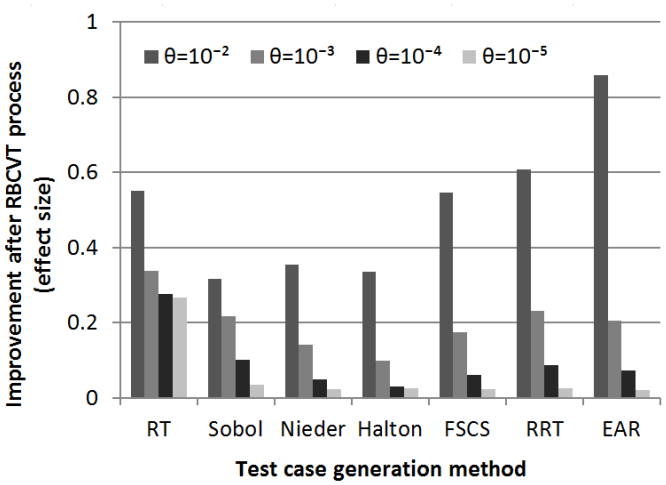


Figure 2.13. Improvement of test case generation methods with respect to RBCVT process at different failure rates regarding the point failure pattern.

Similar to the previous discussion in sections 2.7.2 and 2.7.3, since the variation among the RBCVT results is quite low, the average RBCVT results is considered as a base for the comparison of all the test case generation methods. Figure 2.14 presents a comparison among all the approaches against RT with respect to the point pattern. Again we can observe that the RBCVT method has the highest testing effectiveness. It is worth noting that in contrast with previous patterns, all the ART approaches at $\theta = 10^{-2}$ have generated test cases with lower effectiveness than RT. While the QRT approaches have superior testing effectiveness compared to RT at all the studied failure rates.

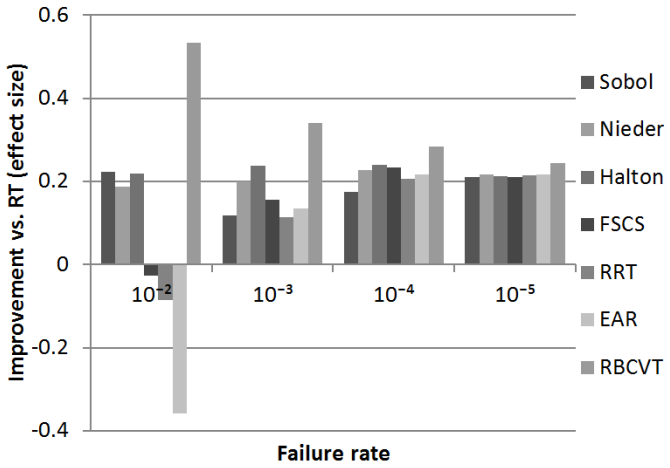


Figure 2.14. P-measure testing effectiveness for point pattern simulations of FSCS, RRT, EAR, RBCVT, Sobol, Niederreiter, and Halton against the RT.

2.7.5 Mutants' Testing Results

The testing effectiveness of all the studied approaches with respect to the real software testing framework based on mutation, are represented in Table 2.5. The results demonstrate the significant improvement after the RBCVT approach is applied. One can observe that, in each case, the amount of improvement in term of effect size is larger than the highest Cohen's description (Large). Further, the effect size is larger than two in all cases, leading to less than 18.9% overlap between the statistics of each method and its corresponding result after the application of RBCVT, according to Table 2.1.

Table 2.5. The P-measure testing effectiveness for all approaches including the corresponding results after the RBCVT process with respect to the mutants' framework.

$\|T\|$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10	RT	0.0112	0.0021	0.0243	0.0033	4.7137	62.99	0.0000
	Sobol	0.0083	0.0046	0.0219	0.0029	3.5604	29.76	0.0000
	Niederreiter	0.0111	0.0018	0.0221	0.0038	3.7172	60.74	0.0000
	Halton	0.0115	0.0020	0.0223	0.0042	3.2508	52.99	0.0000
	FSCS	0.0130	0.0026	0.0326	0.0038	5.9945	75.83	0.0000
	RRT	0.0122	0.0020	0.0267	0.0034	5.2340	71.47	0.0000
	EAR	0.0338	0.0033	0.0632	0.0046	7.3876	89.35	0.0000
20	RT	0.0215	0.0029	0.0336	0.0037	3.6197	41.69	0.0000
	Sobol	0.0150	0.0065	0.0316	0.0033	3.2266	25.58	0.0000
	Niederreiter	0.0229	0.0028	0.0308	0.0036	2.4544	28.49	0.0000
	Halton	0.0223	0.0025	0.0303	0.0036	2.5746	31.92	0.0000
	FSCS	0.0244	0.0036	0.0404	0.0046	3.8819	44.91	0.0000
	RRT	0.0243	0.0032	0.0356	0.0049	2.7345	35.28	0.0000
	EAR	0.0488	0.0049	0.0720	0.0055	4.4511	47.25	0.0000
50	RT	0.0517	0.0049	0.0706	0.0045	4.0120	38.47	0.0000
	Sobol	0.0322	0.0053	0.0695	0.0040	7.9235	69.99	0.0000
	Niederreiter	0.0521	0.0035	0.0691	0.0058	3.5384	48.88	0.0000
	Halton	0.0515	0.0031	0.0690	0.0051	4.1634	57.34	0.0000
	FSCS	0.0562	0.0046	0.0845	0.0056	5.5187	61.28	0.0000
	RRT	0.0569	0.0045	0.0760	0.0044	4.2793	42.18	0.0000
	EAR	0.0817	0.0055	0.1122	0.0058	5.4239	55.54	0.0000
100	RT	0.0925	0.0050	0.1093	0.0043	3.5618	33.26	0.0000
	Sobol	0.0549	0.0055	0.1066	0.0046	10.2313	94.02	0.0000
	Niederreiter	0.0871	0.0057	0.1075	0.0053	3.6866	35.62	0.0000
	Halton	0.0917	0.0035	0.1070	0.0064	2.9412	43.26	0.0000
	FSCS	0.0988	0.0052	0.1182	0.0057	3.5278	36.96	0.0000
	RRT	0.1034	0.0052	0.1143	0.0044	2.2426	20.65	0.0000
	EAR	0.1259	0.0086	0.1445	0.0062	2.4885	21.78	0.0000

Figure 2.15 indicates the improvement of each approach after the RBCVT process in terms of effect size. In contrast with the simulation framework, no particular increasing/decreasing trend has been observed in this figure.

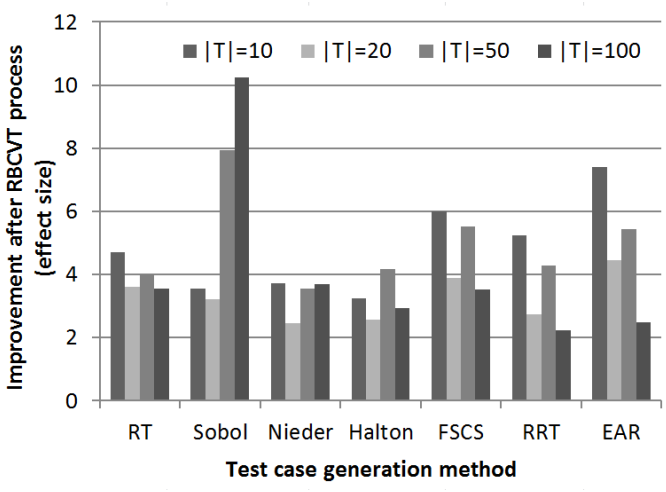


Figure 2.15. Improvement of test case generation methods after the application of RBCVT with respect to the mutants' framework.

Similar to the simulation framework results, Figure 2.16 provides a comparison amongst all of the approaches, where the RT effectiveness is considered as a reference; i.e. Figure 2.16 represents the effect size of each strategy against RT. In contrast with the simulation framework, the P-measure results, after the application of RBCVT, is not similar in all cases. Only in case of QRTs, a sizable reduction of the variation is observed amongst RBCVT results. Accordingly, in Figure 2.16, RBCVT results with QRTs as generators, are combined as QRT-RBCVT, while RBCVT with other inputs are represented separately.

Test case generation approaches in Figure 2.16, are sorted based on their performance where the EAR-RBCVT is the approach with highest efficiency and Sobol has the worst results in term of testing efficiency. Finally, as demonstrated in Figure 2.16, QRT methods revealed degraded performance compared to RT in most of the cases, whereas other test case generation approaches outperformed RT.

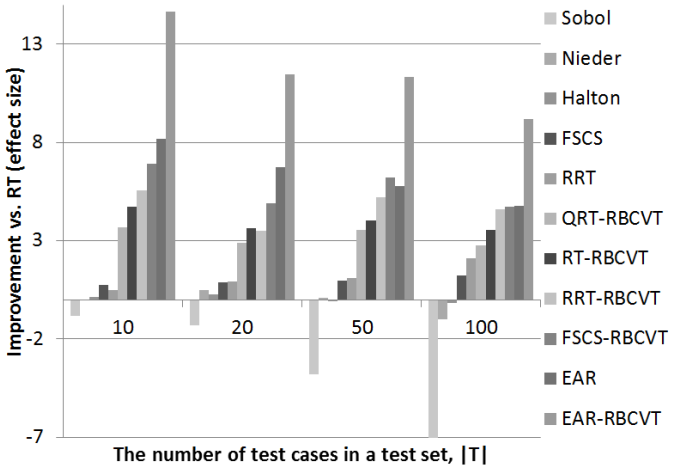


Figure 2.16. P-measure testing effectiveness of each test case generation approach against RT with respect to the mutants' framework.

2.7.6 Empirical Runtime Analysis

In addition to effectiveness, computational complexity of an algorithm is an important factor in practical applications. In this chapter, different algorithms have been used as basis to study the RBCVT method and in this section the runtime of these methods as well as RBCVT is investigated.

All the simulations within this study were conducted using Java (JDK 7, 64bit). We implemented the RBCVT, FSCS, RRT, and EAR in Java and Martingale stochastic library [59] has been used to generate the Sobol, Halton, and Niederreiter quasi-random sequences. Besides, the Java native pseudo-random function has been employed for the RT test case generation. The hardware platform, where the simulation process has been executed, was an Intel dual-core Processor E6300 (2.8GHz) with 8GB of RAM.

To demonstrate the computational costs associated with each algorithm, an empirical runtime investigation has been performed. The parameters associated with each approach are the same as used during the evaluation, described in Section 2.6.2. Figure 2.17 represents the test set generation runtime for the FSCS, RRT, EAR, RBCVT, and RBCVT-Fast in seconds. The runtime of the RT and QRT approaches has not been included in this figure due to their significantly lower runtime compared to the RBCVT and ART methods. The presented runtime values are the average runtime of M_t=100 test set generation for each approach with each test set length ($0 < |T| \leq 100,000$). As indicated in this figure, the non-optimized RBCVT has the largest runtime compared to all other methods and is within the order of quadratic time as calculated in Section

2.5.1.1. In accordance with runtime analysis in Section 2.5.2.3, RBCVT-Fast runtime is linear based on empirical values observed in Figure 2.17. Figure 2.17 also demonstrates that RBCVT-Fast has the best runtime compared to the non-optimized RBCVT and all the investigated ARTs, for $|T| \geq 30,000$. In addition, the computational complexity of 170 seconds for generating 100,000 test cases, suggests 1.7 mili seconds for each test case in the proposed RBCVT-Fast calculation approach. It is worthwhile to note that similar to ARTs, we can apply the mirroring technique [27] to RBCVT to reduce the execution times further if it is required.

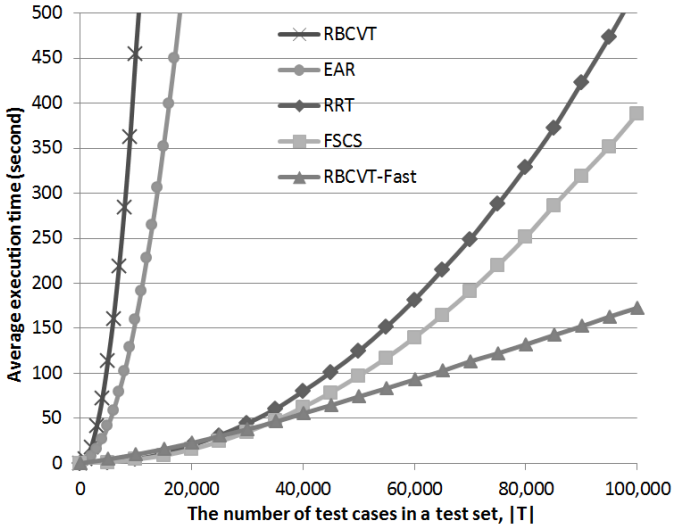


Figure 2.17. Empirical test set generation runtime for the RBCVT, RBCVT-Fast, FSCS, RRT, and EAR.

2.8 Degree of Randomness Analysis

Beside the even distribution of the test cases within a test set, another important aspect of test case generation algorithms is their ability to generate a sequence of test cases which are random. Requiring random test cases has two different implications in this context:

- Randomness within a test set indicates the randomness among the individual test cases within a test set. A high degree of randomness in test cases is better since it provides the ability to generate uncorrelated test cases, which is essential for software testing applications. Uncorrelated test cases are critical to avoid systematic poor-performance in certain situations (that is, a non-random set of test cases could significantly correlate with a current set of defects).

- Randomness between multiple test sets which represents absence of correlation between two, or more, different sequences of test cases, resulting from different runs of the corresponding test case generation algorithm. This is a critical feature of test case generation algorithm since software testing applications require uncorrelated sequences of test cases. Executing a sequence of test cases will hopefully result in the discovery of a number of defects. After correction, we may elect to execute another set of tests; ideally, the tester wants the option to execute either the previous set or a new set of test cases. Alternatively, if no or few defects where discovered, the tester will often want the option of executing another new, and by definition, different set of test cases in an attempt to discover more defects.

How can we measure randomness? Kolmogorov complexity provides a new class of distances appropriate for measuring similarity relations between sequences [22], [23]. The Kolmogorov complexity of a piece of information ($\delta(data)$) is the length of the ultimate lossless compressed version of the corresponding information [23]. In fact, the ultimate compressor does not exist. Thus, we have to use the lower bound of what a real-world compressor can achieve [23]. Within this study, the Lempel-Ziv-Markov chain Algorithm (LZMA) [60] is used to calculate $\delta(.)$ since it is believed that it is one of the best lossless compressors available. Before we can use a test set (T) as input to LZMA we need to preprocess the test set to convert it to a set of Integer values. Assuming a test set as

$$T = \left\{ \{x_1, y_1\}, \{x_2, y_2\}, ..., \left\{ x_{|T|}, y_{|T|} \right\} \right\}, \tag{2.19}$$

where $\{x_i, y_i\}$ denotes a two-dimensional test case (t_i), the preprocessing function is defined as

$$T = \left\{ x'_1, y'_1, x'_2, y'_2, ..., x'_{|T|}, y'_{|T|} \right\}, \tag{2.20}$$

where x'_i and y'_i denote the scaled integer representation of x_i and y_i, respectively.

Accordingly, to analyze within a test set randomness, $CR(T) = \dfrac{\delta(\varphi(T))}{|\varphi(T)|}$ is used which indicates the compression ratio with respect to T. A compression ratio of one denotes a totally random test set, while less compression values denote repetitive patterns within

the test set. Theoretically, $0 \le CR(T) \le 1$. However, since LZMA is not a perfect compressor a small (unknown) additive offset exists in the estimation of $CR(T)$.

To investigate randomness between test sets, we used the Normalized Compression Distance (NCD) [23] indicating the similarity between two test sets. NCD is defined as [23]

$$NCD(T_i, T_j) = \frac{\delta(\varphi(T_{ij})) - min\{\delta(\varphi(T_i)), \delta(\varphi(T_j))\}}{max\{\delta(\varphi(T_i)), \delta(\varphi(T_j))\}} \tag{2.21}$$

where T_{ij} is formed from the concatenation of T_i and T_j. When $NCD(T_i, T_j) = 0$, T_i and T_j are identical, whereas $NCD(T_i, T_j) = 1$ represents complete dissimilarity (these relationships assume perfect compression). The length of the test set should be large enough to be compressed effectively by LZMA. Thus, within this chapter the length of each test set is selected as an arbitrary large number, specifically as $|T|$=100,000.

Table 2.6 represents the results of $CR(T)$ and $NCD(T_i, T_j)$ for RT, FSCS, RRT, and EAR approaches before and after the RBCVT process. QRT approaches have not been included since they use a deterministic algorithm producing a unique test set. The reported values in Table 2.6 are the average of 100 measurements which indicates similar results before and after the RBCVT process regarding all studied approaches (in all situations, the variation between trials was negligible). These results suggest no degradation by RBCVT on the input points regarding randomness. In addition, all the ART methods perform similar to RT with respect to degree of randomness.

Table 2.6. CR(T) and NCD(Ti, Tj) for RT, FSCS, RRT, and EAR before and after the RBCVT process.

method	$CR(T)$		$NCD(T_i, T_j)$	
	Before CVT	After CVT	Before CVT	After CVT
RT	1.0133	1.0136	0.99954	0.99955
FSCS	1.0141	1.0142	0.99953	0.99953
RRT	1.0144	1.0144	0.99952	0.99955
EAR	1.0144	1.0142	0.99954	0.99956

2.9 Summary

In this chapter, the novel RBCVT method has been proposed to the domain of software testing with the aim of increasing the effectiveness of numerical test case generation approaches. The RBCVT method cannot be considered as an independent approach since it requires an initial set of input test cases. This method is developed as an add-on to the previous ART and QRT methods enhancing the testing effectiveness by more evenly distributing test cases across the input space. In addition, the applied probabilistic approach for RBCVT generation, allows different sets of outputs to be produced from the same set of inputs which makes RBCVT an appropriate method for software testing applications.

The computational cost of a test case generation algorithm should be carefully considered in a practical application. In this chapter, we optimized the probabilistic computational algorithm of the RBCVT approach. The proposed search algorithm reduces the RBCVT computational complexity from a quadratic to a linear time order regarding the size of the test set. While, ART methods still suffer from high runtime order. In this regard, the computational cost of RBCVT is quite feasible with respect to practical applications. It is worthwhile to state that since the RBCVT approach requires initial test cases, the computational cost of the input test set generation is added to the RBCVT calculation cost. Since the results provided in Tables 2.2-2.5 indicate, on average, "similar" results for RBCVT with different types of generators, we can select the RT method, which is linear and adds a low computational overhead, onto the RBCVT execution. Therefore, with a concatenation of the RT and the RBCVT-Fast methods, we can produce a linear algorithm with respect to computational complexity, although in some specific situations this may lead to a slight reduction of algorithmic effectiveness. The principle contribution of this chapter is utilizing CVT to develop an innovative test cases generation approach, in particular RT-RBCVT-Fast with linear order of computational complexity similar to RT.

An extensive experimental study has been performed and the results demonstrate that RBCVT is significantly superior to all approaches for the block pattern in simulation framework at all failure rates as well as the studied mutants at all test set sizes. Although the magnitude of improvement in testing effectiveness results is higher for the block pattern compared to the point pattern, the results demonstrate statistically significant improvement in the point pattern. In contrast, ART methods have indicated less

effectiveness than RT regarding point patterns at θ=0.01 (demonstrated in Figure 2.14). Although RBCVT's performance regarding strip pattern is statistically significant compared to the other approaches at $\theta = 10^{-2}$, the impact of RBCVT verses the other approaches tends to zero as the failure rate decreases. In fact, in the case of strip pattern, the impacts of all of the approaches reduce to the performance of RT as the failure rate decreases; this is demonstrated in Figure 2.12. In contrast, in block and point patterns, the performance of all the approaches verses RT usually stays constant or even increases as the failure rate reduces. It is believed that these conclusions are stable regardless of the failure rate, and hence, simulating lower failure rates than studied in this chapter is not required. This fact is also verified in [61]. Randomness of test cases is an important factor with respect to software testing. Accordingly, the investigation of randomness in Section 2.8 demonstrates that RT, all ART methods and all corresponding RBCVT methods possess an appropriate degree of randomness.

Although in real life applications, test cases' dimension can be large, in most cases they belong to an acceptable range. Test case generation often seeks to generate values with a specific purpose rather than generating test cases to exercise the entire system. The large size of the input space for modern software systems tends to imply that this "scatter gun" approach is ineffective. Instead, the tester will often have a specific testing objective and will attempt to generate a specific set of test cases under specific circumstances that answer this question. That is, the tester tends to test aspects of the system or sub-components of the system rather than blindly "attacking" the entire system. As an example, in unit testing, the program under test is usually small, so the number of input and output variables are limited as is the number of dimensions. For instance, Ciupa et al. [62] conducted an empirical study on several real world small routines using unit testing. Briand and Arcuri [49] have considered 11 programs, basic mathematical functions that appear in the ART literature [17], for empirical analysis. The generated test cases in these papers do not exceed four dimensions. Furthermore, some techniques like range coding [63] exist to reduce the dimension of the input space, especially when collections are considered as the input to the software under the test. As a result, where we do not have large dimensions, the linear RBCVT-Fast approach dominates over ART approaches regarding computational cost.

Finally, although further studies are required to validate the use of RBCVT in real-life applications, RT-RBCVT, ART-RBCVT, and QRT-RBCVT have been demonstrated to

have a superior performance against RT, ART, and QRT methods, respectively. Consequently, software testing practitioners can use RBCVT to enhance the existing strategies within their software testing toolbox. The use of RBCVT in software testing is straightforward since RBCVT can be included to the previous methods as an add-on.

3 String Test Data Generation through a Multi-Objective Optimization

String test cases are required by many real-world applications to identify defects and security risks. Random Testing (RT) is low cost and easy to implement testing approach to generate strings. However, its effectiveness is not satisfactory. In this chapter, black-box string test case generation methods are investigated. Two objective functions are introduced to produce effective test cases. The diversity of the test cases is the first objective, where it can be measured through string distance functions. The second objective is guiding the string length distribution into a Benford distribution [64] which implies shorter strings have, in general, a higher chance of failure detection. When both objectives are applied via a multi-objective optimization algorithm, superior string test sets are produced. An empirical study is performed with several real-world programs indicating that the generated string test cases outperform test cases generated by other methods.

3.1 The Focus of This Chapter

In this chapter, the objective is to generate an effective set of test cases where each test case is a string. As explained before, based on empirical studies [13]–[17], fault regions normally form continuous regions in the input domain. Based on this assumption, a diverse set of test cases has a greater chance of detecting a fault. Hence, it is believed that a diverse set of test cases is more likely to produce more effective test cases [13]–[17]. To achieve this in the string domain, we have defined a fitness function that measures the diversity of a test set. This allows an optimization technique to be employed to generate test cases based upon the fitness function. To construct a fitness function to measure the diversity, we utilize distance functions between strings. There are several string distance functions available and hence, in this chapter, we compared their performance when used in test generation. Different string distance function's performance is compared in terms of the effectiveness of the generated test cases and their runtime. Since runtime performance is important in practical applications, we further extend this chapter by applying a hash based distance function into the test generation methods to improve the runtime efficiency.

We also hypothesize that the distribution of the length of the generated strings plays an

important role in failure detection. We argue that smaller strings have a higher chance of detecting a failure. Since the first fitness function is unable to control the length distribution of the strings, we create a second fitness function which indicates the proximity of the distribution of the lengths of the strings in a test set to the target distribution. A multi-objective optimization technique is used to apply both fitness functions simultaneously.

To empirically investigate this hypothesis, we generate mutants of 13 programs. Test sets with different characteristics are generated and tested on these programs. The experimental results demonstrate that failure detection is improved when both fitness functions are applied.

The highlights of this chapter can be summarized as:

1) Introducing two fitness functions to control the diversity and length distribution of the string test cases and optimizing both fitness functions through multi-objective optimization techniques.

2) Investigating the performance of six different string distance functions in black-box string test case generation.

3) Applying Locality-Sensitive Hashing (LSH) [65] technique, a fast estimation of string distances, to improve the runtime order complexity. Comprehensive runtime complexity improvement is discussed in Section 3.5. Further, empirical runtime analysis is investigated in Section 3.7.4.

4) Empirical investigation of the proposed method and comparison with other methods using a mutation analysis.

5) Analysis of the degree of randomness of the generated strings in Section 3.8. The degree of randomness is critical to avoid systematic poor performance due to the correlation between the tests. It can be investigated a) within a set of test cases; and b) between multiple sequences of test sets.

The “string test case” is a general term and hence, we define the scope of research in this chapter. In this research, the objective is string test case generation; not test case selection [66] or prioritization [67]. Further, as discussed in Chapter 1, this research focuses on black-box string test generation. White-box test generation methods, like symbolic execution [68], are another category of string test generation which utilizes the source

code to produce test cases. Typically, these methods try to increase the code coverage using optimization methods to generate test cases [69]. These string-related techniques are reviewed in Section 3.9.

3.2 Adaptive Random String Test Case generation

As discussed in the previous chapter, to improve the poor effectiveness of RT, ART methods are introduced. Chen et al. [18] first introduced Fixed Size Candidate Set (FSCS) and then a variety of other ART methods have been developed by other researchers.

Most of the ART methods are designed for numerical test cases and they cannot be used to generate string test cases. Among the ART methods, the FSCS and ART for Object Oriented software (ARTOO) [62] methods are capable of more complex test case structures than fixed size vector of numbers and they can be applied to string test cases. Further, Mayer et al. [32] concluded that FSCS was one of the best ART methods through an empirical study. As a result, we adapted FSCS and ARTOO to generate string test cases in this chapter; these are reviewed in the following sections.

3.2.1 Fixed Size Candidate Set (FSCS)

FSCS method is discussed in depth in chapter 2 and hence, is not repeated here. The only difference is that, in this chapter, a string distance function is used in FSCS. FSCS has been initially introduced for numerical test cases. However, it can be applied to other test case structures like strings. The only requirement is that a distance function is defined between the test cases.

To generate test cases, FSCS uses a distance based procedure. The first string test case is generated randomly, similar to RT. Then, to generate other test cases, a fixed size candidate set is used to produce a test case. Therefore, K random strings are generated as candidates (K=10 is used in the experiments based on the recommendation of Chen et al. [25]). A string is selected where it has the largest distance from previously executed string test cases.

3.2.2 ART for Object Oriented Software (ARTOO)

ARTOO [62] is an ART method designed for object oriented software where it uses a distance function between objects to generate the test cases. The authors focus on the specific problem of testing functions of an object-oriented program where test cases are

input objects to the functions. ARTOO works similar to FSCS [62], it selects a test case among the pool of candidates. The number of candidates for ARTOO is chosen as 10 to match with the FSCS. The difference between FSCS and ARTOO is the selection rule among the candidates. The mean distance of each candidate to the previously selected test cases is calculated. Then, a candidate with the largest mean distance is chosen as the winner (next test case) [62].

3.3 Evolutionary String Test Case Generation

To generate string test cases, evolutionary algorithms can be used. Among the evolutionary algorithms, Genetic Algorithms (GA) [70] are the most commonly used search algorithm in software engineering [71]–[73]. GAs also fit very well with our application which requires string manipulations. Two approaches are used to produce test sets based on GAs. First, we utilize a GA with a single objective, where a diversity-based fitness function is used. Then, a second fitness function is defined to control the length distribution of the strings. Hence, in the second approach, we use a Multi-Objective GA (MOGA) [74] to optimize both fitness functions simultaneously.

3.3.1 Genetic Algorithm (GA)

In the following, we first briefly explain GA's basic terminology and then, appropriate fitness functions and GA's parameters are discussed. Multiple chromosomes form a population where a chromosome is a candidate solution. At each generation, some chromosomes are selected (by the selection mechanism) and offspring are generated via a crossover operator. Finally, the mutation operator is utilized to make random small changes to the generated offspring resulting in a lower probability of becoming trapped in a local optimum point.

3.3.1.1 Diversity-Based Fitness Function

A GA requires a fitness function to generate optimized test sets. According to the discussion in the introduction, it is believed that a diverse set of test cases is more likely to reveal faults more effectively [13]–[17]. Hence, we define a fitness function that measures the diversity

$$Fitness\ function = \sum_{i=1}^{test\ set\ size} dist(t_i, \beta(t_i, test\ set)) \tag{3.1}$$

where the summation is performed on the distance between every test case and its nearest

test case. t_i represents the ith test case in the test set, and β indicates the nearest test case in test set to t_i. A higher value of this fitness function implies a more diverse distribution of test cases as it indicates that test cases are far from each other.

3.3.1.2 GA Parameters

Using a GA requires the definition of its elements and parameters. In this chapter, a chromosome is a string test set. So, to generate the initial population, random test cases are generated. We chose the size of the population as 100 since larger population sizes produced no improvement. We have tested the GA with three selection mechanisms, roulette-wheel selection, rank selection, and binary-tournament selection [70]. The experimental results demonstrate that the performance of the all selection methods is very close. However, rank selection slightly produces better results. Hence, rank selection is used for the GA. In crossover, test sets are recombined to generate offspring test sets using a 60% crossover rate [75]. In test sets recombination, given that both parents have a same number of string test cases, each string in the first parent test set is combined with the corresponding string in the second parent test set and two string children are produced. This is repeated for all the string test cases in the parent test sets which leads to two offspring test sets. A single point recombination [70] is used to generate children strings from two parent strings. In a single point recombination, random points are selected in each of the two parent strings. Then, to generate the children strings, the first part of each parent string is concatenated to the second part of the other parent.

Edit, delete, and add are used as mutation operators where every character in each string is mutated with 1% probability. Each time, one of the mutation operators is selected randomly. In an edit operation, the character is replaced with another randomly selected character. The delete operation eliminates the character and the add operator, inserts a randomly selected character in the current position in the string.

Finally, the iterations are stopped when one of the following is reached: (a) No improvement is achieved in 20 generations based upon the fitness function; or, (b) A maximum of 200 iterations is reached.

3.3.2 Multi-Objective Genetic Algorithm (MOGA)

3.3.2.1 String Length Fitness Function

Beside the diversity-based fitness function, the distribution of the length of the generated

strings may play an important role in failure detection. Accordingly, in this section, a fitness function for string length distribution is investigated.

It is argued that data (a population of objects) essentially has two root causes, either real-world or artificial situations. Artificial populations of objects have no restrictions on their growth. For instance, computer-generated unique identifiers can have any sampling distribution. However, real-world populations of objects have more restrictions; growth takes time and is sequential. Hence, these populations are often modelled by an exponential growth model. Such a model starts with typically a small population (starting point) and "moves towards the right" on a log-scale at a constant rate [76], [77]. Hence, if a (random) variable starts at 1, it spends more time growing between 1 and 2 than between 2 and 3. Growing continues and the pattern is repeated; that is, the variable spends more time growing between 10 and 20 than between 20 and 30. The growth exhibits scale-invariance and characterized by the most significant digit [77], [78]. This is commonly known as Benford's Law [78]. Benford's law indicates that the occurrence of digits in a list of numbers is not uniform and follows a logarithmic distribution known as the Benford distribution [64]. Figure 3.1.a represents the distribution of first digit numbers where the base is 10. The Benford distribution can be calculated using [64]

$$PDF_B(n) = \log_b\left(1/n + 1\right), \quad 1 \le n < b, \tag{3.2}$$

where b denotes the base of the numbers, and $PDF_B(n)$ represents the Benford distribution.

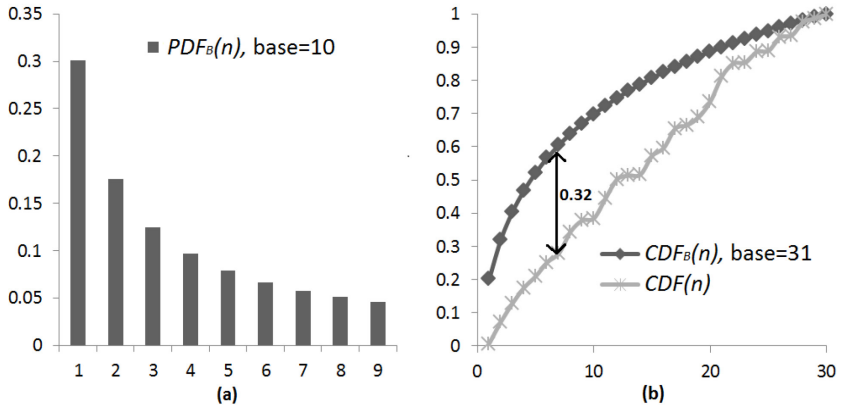


Figure 3.1. (a) Benford distribution ($PDF_B(n)$) where base is 10. (b) Kolmogorov–Smirnov test is used to measure the distance of two distributions. $CDF(n)$ and $CDF_B(n)$ are cumulative probability distribution of the strings length and Benford, respectively. The max string length is assumed to be 30 which leads to the Benford base of 31.

The Benford distribution is empirically investigated in many areas [64], [79]. It can be applied to a wide variety of data sets, including financial data, electricity bills, stock prices, lengths of rivers, population numbers, street addresses, death rates, and physical and mathematical constants [64]. Perhaps, the most widely known application of Benford's law is detecting fraud in accountancy and financial data, where Benford's law can effectively identify non-conforming patterns [64], [80]. In addition, Raimi [81] has shown that the products of independent random variables follow Benford's law. Hence, Benford's law provides a very general idea of how arbitrary populations of objects grow which is independent of any domain knowledge. A detailed discussion on Benford's law and its wide applications can be found in [64], [79], [82].

Accordingly, this paper hypothesizes that the Benford distribution is applicable to defining the distribution of the size of strings found in computer programs many of which are models of real-world situations. Such strings (a population of characters under an ordering constraint) are unbound, but their size is defined somewhat by what they are modelling and what they are modelling is a mixture (product) of smaller items (e.g. a person's contact information is a mixture of their name, address, mobile number, etc.). These smaller items can be decomposed into even smaller items – single characters (starting point). While not ideal (non-coverage of artificial situations), it is argued that Benford's law provides a reasonable representation of the size of strings which are likely to be encountered when no domain-specific knowledge is available. Hence, we hypothesize that Benford's distribution is a good model for string length distribution within a test set when no domain-specific knowledge is available. This essentially means that smaller strings have a higher chance of detecting a failure. So, we argue that if we generate diverse string test cases and control the distribution of their length, more effective test cases can be generated.

To examine this hypothesis, we first need to develop a fitness function that measures the distance of the Benford distribution and the distribution of the string lengths. The chi-squared test [64] has been used to test the compliance of a distribution with Benford distribution. However, it has low statistical power with small samples [83]. Since maximum test set size in our experiments is 30, chi squared test may not produce adequate results as a fitness function. To solve this problem, we use a Kolmogorov–Smirnov test [84]; this is more powerful when the sample size is small [84].As indicated in Figure 3.1.b, the Kolmogorov–Smirnov test finds the maximum distance between two

cumulative probability distributions [84]. It can be formulized as

$$Fitness\ function = \max_{n \in [1, StrMax]} |CDF(n) - CDF_B(n)| \quad (3.3)$$

where *CDF(n)* and $CDF_B(n)$ are cumulative probability distributions of the strings length and Benford, respectively. Finally, *StrMax* denotes the maximum string length. The Benford distribution provides a probability distribution in *[1,b-1]*; and hence, Benford's base is set as *b=StrMax+1*. Further, the Benford distribution does not provide a probability for zero which produces a problem for strings with no characters. To solve this issue, we assume that each string has a terminator character and we count it toward the string size. Therefore, a string with no character has a length of one and it can be adapted to the Benford distribution.

3.3.2.2 Pareto-Optimal Test Sets

A multi-objective optimization technique is required to enforce both fitness functions (namely *F1* and *F2*) simultaneously. We employ one of the widely used multi-objectives GAs (MOGA), namely NSGA-II [74]. Since the diversity needs to maximized, the value calculated from (3.1) is inverted. Therefore, both fitness functions need to be minimized.

A basic step in NSGA-II is sorting of chromosomes in a population based on a domination concept. Chromosome *A* dominates *B* if and only if (*F1(A)<F1(B)* and *F2(A)≤F2(B)) or* (*F1(A) ≤F1(B)* and *F2(A)< F2(B))*. A non-dominated chromosome is a chromosome that is not dominated by any other chromosomes in the population. To perform the sorting, NSGA-II categorizes a population's chromosomes into front lines. First front includes all the non-dominated chromosomes. Second front includes non-dominated chromosomes where chromosomes in the previous fronts are not considered. This process is repeated until all chromosomes are assigned to front lines. Within a front line, chromosomes are sorted to preserve the diversity [74]. That is, chromosomes are rewarded for being at the extreme ends or the less crowded areas of a front. The complete sorting algorithm is provided by Deb et al [74].

To generate the test cases the following steps are performed according to NSGA-II.

Step 1) The initial population with size *N* is generated randomly.

Step 2) The population is sorted.

Step 3) An offspring population with size *N* is created using selection mechanisms,

crossover, and mutation [74].

Step 4) A combined population of offspring and parents is produced with size *2N*.

Step 5) The new population is sorted and the first *N* chromosomes are selected to form the next generation.

Step 6) A check to see if the stopping criterion have been met is performed. If the criterion is not met then we return to step 3.

NSGA-II produces a Pareto-optimal set of test sets rather than a single optimal test set. The Pareto-optimal set is the first front of the last generation of the algorithm. Among the Pareto-optimal test sets, the results indicate that the test set with best diversity fitness on the Pareto-optimal front generates the best failure detection effectiveness. Consequently, for the results that are presented for MOGA in this chapter, the test set with best diversity fitness on the Pareto-optimal front is selected. This implies that the best solution is the solution with best diversity which also achieved the target string length distribution.

3.3.2.3 NSGA-II Parameters

We applied similar parameters as GA to NSGA-II. The population size, mutation operators, and mutation rate is identical to GA. However, NSGA-II has no crossover rate parameter as discussed in previous section. NSGA-II uses binary tournament selection mechanism [74]. We also extended NSGA-II and replaced the selection mechanism with rank selection. The experimental results of these two selection methods, demonstrate slightly better performance when the binary tournament selection is used; and hence it is used for rest of the experiments in this study. The roulette-wheel selection is not applicable to NSGA-II. Finally, the iterations are stopped when one of the following is reached:

- No chromosome is produced in 20 generations that dominates at least one chromosome in the first front Or,
- A maximum of 200 iterations is reached.

3.4 String Distance Functions

A distance function between two strings is required in ART and evolutionary test case generation methods. Several string distance functions are introduced in the literature [62], [66], [67], [85]. Although we cannot afford to investigate all of them, a good portion of

them, especially those that normally perform well in software testing studies, are covered in this chapter.

Accordingly, we performed the experiments with six string distance functions. Four of which are Levenshtein [86], Hamming [87], Cosine [88], Manhattan [67], and Euclidian [67] distance functions that are repeatedly used in software testing studies [62], [66], [67], [85]. Further, we also used Locality-Sensitive Hashing (LSH) [65] technique as a fast estimate of string distance in our work.

3.4.1 Levenshtein Distance

The Levenshtein Distance [67] is an edit-based distance that works based on three edit operations, "delete", "insert", and "update" [67]. Each operation has an associated cost where each string can be converted to the other string based on these edit operations. The distance is the minimum cost of a sequence of edit operations that converts one string into the other string [67]. The Levenshtein distance assigns a unit cost to all edit operations [67].

Mathematically, the Levenshtein distance between two strings, *Str1* and *Str2,* is equal to *lev(Length(Str1), Length (Str2))* where it can be calculated recursively by

$$lev(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) == 0 \\ \min \begin{cases} lev(i-1,j)+1 \\ lev(i,j-1)+1 \\ lev(i-1,j-1)+cost(i,j) \end{cases} & \text{otherwise} \end{cases} \qquad (3.4)$$

$$cost(i,j) = \begin{cases} 0 & \text{if } Str1_i == Str2_j \\ 1 & \text{otherwise} \end{cases},$$

where $Str1_i$ denotes the *i*th character of *Str1,* and $Str2_j$ denotes the *j*th character of *Str2*.

3.4.2 Hamming Distance

The Hamming distance [67] was initially introduced as a measure to calculate the distance of two bit streams. However, it has been adapted to be used for strings [67]. The Hamming distance of two strings, like "abcd" and "anfd", is the number of characters different in two strings. In other words, every character in the first string is compared with a character in the equivalent position in the second string. In this example, the distance is two. In cases where the sizes of two strings are not equal, null characters (ASCII code of zero) are added to the end of the smaller string until both strings have a

same size. For example, the distance between “ab” and “acdb” is three.

3.4.3 Manhattan Distance

The Manhattan distance [67] is normally used for vectors of numbers. It also can be applied to strings as

$$\text{Manhattan distance} = \sum_{i=1}^{n} \left| Str1_i - Str2_i \right| \tag{3.5}$$

where $Str1_i$ and $Str2_i$ are ASCII codes of the ith character. Similar to the Hamming distance, when the size of the two strings is not equal, null characters are added to the shorter string.

3.4.4 Euclidian Distance

The Euclidian distance [67] is similar to the Manhattan distance. It can be applied to strings as

$$\text{Cartesian distance} = \sqrt{\sum_{i=1}^{n} (Str1_i - Str2_i)^2} \tag{3.6}$$

Again, null characters are added to the shorter string until both strings have a same size.

3.4.5 Cosine Distance

The Cosine similarity [88] calculates the similarity of two vectors as a cosine of the angle of two vectors. The Cosine similarity can be calculated as follows where ASCII codes are used as a number.

$$\text{Cosine similarity} = \frac{\sum_{i=1}^{n} Str1_i \times Str2_i}{\sqrt{\sum_{i=1}^{n} Str1_i^{\,2}} \times \sqrt{\sum_{i=1}^{n} Str2_i^{\,2}}}. \tag{3.7}$$

Similar to The Hamming distance, when the size of the two strings is not equal, null characters are added to the shorter string. Finally, to calculate the distance, 1- Cosine similarity is used.

3.4.6 Locality-Sensitive Hashing (LSH)

LHS [65] is a technique that can be used as a fast estimation of the distance between two strings. The basic idea is to hash strings such that similar strings are mapped into a same

hash code with a high probability. Random projections are core elements used to map the input data to a value [65]. In this chapter, we used a type of random projection that is used to estimate cosine distances. This projection is defined as [89]

$$h^x(\vec{v}) = \begin{cases} 1 & \vec{x} \cdot \vec{v} \geq 0 \\ 0 & \vec{x} \cdot \vec{v} < 0 \end{cases} \tag{3.8}$$

where v is the input vector, x is a random vector generated from a Gaussian distribution, and $h^x(\vec{v})$ is a bit representing the location of v compared to x. P random projections are used to construct a hash value where it indicates the location of the input vector compared to the P random vectors. Therefore, we have P bits as a hash value; P=32 is used in this research.

Finally, the Hamming distance is used between two hash bit strings which leads to an estimation of the cosine distance of the original strings. LSH improves the runtime order as the Hamming distance between two 32 bit streams is independent of the sizes of the strings. A comprehensive runtime order investigation is presented in the next section.

Cosine and LSH distances are naturally normalized against the length of the strings and hence, we do not need to normalize them. However, the other discussed distances are not naturally normalized. To normalize them, the result is divided by *Length(Str1)+Length(Str2)*.

3.5 Runtime Order Investigation

The computational complexity of an algorithm is an important factor in practical applications. In real-world applications, the size of strings and the size of test sets may become very large. Hence, it is importance for the user to know how the execution time grows when parameters are changed. Accordingly, in this section, the order of runtime complexity for the distance functions, fitness functions, test case generation methods are investigated. The runtime order is analyzed based on the string length of distance functions (L_1 and L_2), test set size (TS), population size in GA and MOGA (N), and number of potential candidates in ART (K). Table 3.1 provides the runtime order of all the algorithms. In the following, detailed discussions are presented.

Table 3.1. Runtime order complexity of each algorithm used in this chapter.

Algorithm	Runtime Order
String Distance Functions	
Levenshtein	$O_D = L_1 \times L_2$
Hamming	$O_D = Max(L_1, L_2)$
Manhattan	$O_D = Max(L_1, L_2)$
Euclidian	$O_D = Max(L_1, L_2)$
Cosine	$O_D = Max(L_1, L_2)$
LSH (part1: hashing)	$O_{LSH1} = L_1$
LSH (part2: Hamming distance)	$O_{LSH2} = 1$
Fitness Functions	
Diversity-based (with LSH)	$O_{FD} = TS \times (TS + O_{LSH1})$
Diversity-based (other distance functions)	$O_{FD} = TS^2 \times O_D$
Length control	$O_{FL} = TS$
Test Set Generation Methods	
RT	$O_{RT} = TS$
FSCS (with LSH)	$O_{FSCS} = K \times TS \times (TS + O_{LSH1})$
ARTOO (with LSH)	$O_{ARTOO} = K \times TS \times (TS + O_{LSH1})$
FSCS (other distance functions)	$O_{FSCS} = K \times TS^2 \times O_D$
ARTOO (other distance functions)	$O_{ARTOO} = K \times TS^2 \times O_D$
GA	$O_{GA} = N \times O_{FD}$
MOGA (NSGA-II)	$O_{MOGA} = N^2 \times (O_{FL} + O_{FD})$

The Hamming, Manhattan, and Cosine distance runtime complexity is linear against the length of the strings as can be observed from (3.5) and (3.7). Since each of these distance functions add null characters to the end of smaller string to make it same size with the longer string, the order of complexity is $Max(L_1, L_2)$. The runtime order of Levenshtein distance is quadratic ($L_1 \times L_2$) since a $L_1 \times L_2$ matrix needs to be constructed according to (3.4).

The story for the LSH distance function is different as it has two parts. The first part that calculates a hash value is linear against the length of a string. The second part is done in a constant time as it is a Hamming distance between two fixed length bit streams. As a result, the LSH produces a runtime complexity improvement for test case generation methods and diversity-based fitness function. In the diversity-based fitness function, a distance between every string pair in a test set needs to be calculated. This leads to $TS^2 \times O_D$ runtime order where O_D denotes runtime order of a distance function other than the LSH. However, with the LSH, we can calculate the hash value of each string first which can be done in $TS \times O_{LSH1}$. Then, each pair distance calculation can be done in TS^2 since $O_{LSH2} = 1$. Adding these two terms leads to $TS \times (TS + O_{LSH1})$ which is more efficient

than $TS^2 \times O_D$. This improvement leads to the runtime efficiency in GA and MOGA test case generation methods. The runtime order of NSGA-II is reported as $N^2 \times M$ [74] where M represents the number of objective functions. However, the complexity order of fitness functions is not included. Assuming O_{FL} and O_{FD} as the complexity order of length control and diversity-based fitness functions, respectively, complexity order of NSGA-II becomes $N^2 \times (O_{FL}+O_{FD})$. O_{FL} can be removed compared to O_{FD} as O_{FL} has a linear complexity. Obviously, any improvement in O_{FD} related to the LSH, has a direct effect on the complexity of NSGA-II. Similarly, the complexity order of GA ($N \times O_{FD}$) is improved using the LSH rather than other distance functions.

Similar arguments can be made for ART methods. The FSCS runtime order is reported to be $K \times TS^2$ [19], [90] for numerical test cases. However, considering the distance function runtime for strings, it becomes $K \times TS^2 \times O_D$. To calculate the runtime order with LSH, we first find the runtime order of generating one test case, t_{i+1}. The distance between every one of the K candidates and the i previously generated test cases need to be calculated. Therefore, the hash of each candidate needs to calculated (runtime order of $K \times O_{LSH1}$) and then distances are calculated (runtime order of $K \times i \times O_{LSH2}$). So, for each test case we have $K \times (i + O_{LSH1})$. A summation over i from one to TS needs to be performed to find the total runtime order to generate a test set. Accordingly, the runtime order is $K \times TS \times (TS+O_{LSH1})$ — an improvement compared to $K \times TS^2 \times O_D$. The ARTOO has a same runtime complexity as FSCS since their algorithms are similar except for where a candidate is selected.

3.6 Experimental Framework

The experiments conducted to analyze the effectiveness of FSCS, ARTOO, GA, and MOGA against RT are described in this section. Real world programs are used to perform an empirical evaluation. These programs accept strings as input. Then, mutated [49], [91] versions of the software are produced. The P-measure [90] was selected to quantitatively measure the effectiveness of the test case generation methods. Finally, features of string test sets are discussed.

3.6.1 Software Under Test (SUT)

To conduct a study on the fault-detection effectiveness of the test case generation methods, 19 real world Java programs are investigated. We reused the programs from McMinn et al. [92] and hence the selection of these programs can be viewed as being

independent from the authors[1]. These programs are sub-components of 10 real-world projects which are widely used in GUI and web applications to validate strings [92]. These programs accept a string as an input and only contain functionality which transforms or validates the input. That is, no significant portion of these programs spent time on anything except string manipulation [92].

Table 3.2 provides a description of each program. The "Name" column denotes a name used in the rest of this chapter to refer to that program. The "Classes" column represents all the associated Java classes to that program. The reported LOC (Line Of Code), in Table 3.2, is the summation of all classes in each program. It is different than LOC reported in the original work [92] as only LOC of the main class is reported in the original work.

Table 3.2. Programs used to perform experimental evaluations.

#	Name	Project, *Source code URL*	Classes	LOC
1	Validation	PuzzleBazar, *code.google.com/p/puzzlebazar*	Validation	80
2	PostCode	LGOL, *lgol.sf.net*	PostCodeValidator, Validator	293
3	Numeric		NumericValidator, Validator	217
4	DateFormat		DateFormatValidator, Validator	236
5	CASNumber	Chemeval, *chemeval.sf.net*	CASNumber	102
6	MIMEType	Conzilla, *www.conzilla.org*	MIMEType, MalformedMIMETypeException	145
7	PathURN		PathURN, URI, URN, MalformedURIException	387
8	ResourceURL		ResourceURL, URI, MalformedURIException	339
9	URI		URI, MalformedURIException	267
10	URN		URN, URI, MalformedURIException	327
11	Util	Efisto, *efisto.sf.net*	Util	244
12	TimeChecker	GSV05, *gsv05.sf.net*	TimeChecker, StringTokenizer	267
13	Clocale	JXPFW, *jxpfw.sf.net*	Clocale, Cdebug	751
14	International		InternationalBankAccountNumber, AbstractLocalizedConstants, Cdebug, Cstring, InvalidArgumentException, ISO3166CountryConstants	2938
15	Isbn	TMG, *tmgerman.sf.net*	Isbn, Field, SimpleDataField	420
16	Month		Month, Field, SimpleDataField	346
17	Year		Year	75
18	BIC	WIFE, *wife.sf.net*	BIC, ISOCountries, PropertyResource	200
19	IBAN		IBAN, ISOCountries, PropertyResource	288

[1] Originally, McMinn et al. [92] used 20 Java programs. Based on the information provided, we were unable to find one of the programs ("OpenSymphony"); and hence, we performed our experiments with 19 programs.

"PuzzleBazar" is puzzle playing software. An email validation class is extracted as one of the programs under the test [92]. "LGOL" is a library developed for local government in UK [92]. Three programs are extracted where they involve string manipulation related to date formats, integer numbers, and UK postal codes [92]. "Chemeval" is a framework used to evaluate molecular structure with application in hazard assessment [92]. The tested class in this project, handles "CAS numbers" which is a unique identifier assigned to chemical substances [92]. "Conzilla" is a tool used in knowledge management. Within this tool, five programs were extracted where one is responsible for validating strings that have MIME types and the rest are used to manipulate and identify a variety of URIs [92]. "Efisto" is a file sending tool via the web [92]. The selected class validates/manipulates dates as a string [92]. "GSV05" is a tool for recording attendance, the selected classes validate/manipulate strings in a time format [92]. "JXPFW" (Java eXPerience FrameWork) is a library where two programs are extracted. The programs are used for the validation and manipulation of international bank account numbers and location identifiers [92]. "TMG" (Text Mining for German documents) include classes to connect to the DBLP research publication database. Three programs are extracted which validate ISBNs (International Standard Book Numbers), month names, year names [92]. Finally, "WIFE" is a tool for handling international bank's SWIFT messages where two string manipulation programs are extracted.

3.6.2 Source Code Mutation

To measure the effectiveness of the test case generation methods, faulty versions of the software under test are required. Mutation techniques [49], [91] are a well-known approach to automatically manipulate the source code and produce a large number of faults [49]. There is considerable empirical evidence indicating a correlation between real faults and mutants [55], [91].

In this chapter, muJava [54] is employed to produce mutated versions of the programs under the test where a total of 6672 mutants are generated. Then, those mutants that were failed with the majority of test sets (more than 90% of all the test sets) were deleted. These defects were considered as unrealistic and hence contrary to the "Competent Programmer" hypothesis which is an essential idea in mutation testing [93]. Six programs (CASNumber, PathURN, Util, International, Month, and Year) were excluded from the experiments since the remaining mutants for these programs revealed no failures. That is, these mutants where never detected by any test cases generated in the experiments.

Hence, 13 programs are available for the evaluation of the test generation methods. Table 3.3 demonstrates the number of generated and selected mutants per program.

Table 3.3. The number of mutants generated for the test programs.

#	Programs	Generated Mutants	Selected Mutants
1	Validation	721	687
2	PostCode	114	60
3	Numeric	48	43
4	DateFormat	54	46
5	MIMEType	92	55
6	ResourceURL	709	706
7	URI	613	597
8	URN	767	764
9	TimeChecker	578	442
10	Clocale	165	160
11	Isbn	284	277
12	BIC	151	109
13	IBAN	195	83

3.6.3 Testing Effectiveness Measure

Similar to chapter 2, we use p-measure to evaluate the effectiveness of test case generation methods. An in depth discussion on the p-measure definition and the reason behind its selection as a quantitative effectiveness measure is presented in Section 2.6.1.

3.6.4 String Test Set Characterization

To evaluate the p-measure, we need a test set with a fixed size. In this chapter, we perform experiments with three test set sizes, 10, 20, and 30. As the size of the test sets increases, the difference in the results of different test generation methods is normally reduced.

Applying a test set to a mutated version of a program will return zero or one according to the p-measure calculation rules. Accordingly, to estimate p-measure as a number between zero and one, we applied 100 test sets. Further, we repeated this process 100 times for each mutated version to be able to estimate mean and standard deviation parameters for the measurements. As a result, each test case generation method (RT, FSCS, ARTOO, GA, and MOGA) produced 10,000 test sets for each test set size. This leads to 10,000×(10+20+30)×5=3,000,000 test cases that have been applied to each mutant.

Each test case is a string of characters. Therefore, we need to determine the range of characters to be used. Previous works commonly used printable ASCII characters [85],

[94], [95]. Tonella [94] used only numbers, lower, and upper case characters. Alshraideh and Bottaci [85] used ASCII code from zero to 127; and Afshan et al. [95] used ASCII code from 32 to 126. We follow Afshan et al. [95] which includes all the printable ASCII characters.

Finally, we need to determine the maximum string length (*StrMax*) allowed. Normally, it can be adjusted by a tester according to the application. Afshan et al. [95] used *StrMax* of 30 to generate strings in a white-box approach. Alshraideh and Bottaci [85] performed their experiments with *StrMax* of 20. In this chapter, we perform all of the experiments with two *StrMax* values (30 and 50) to explore any impact of the maximum string size.

3.7 Experimental result and discussion

This section presents the results of the empirical study. At first, the results of each program under the test are presented. The Levenshtein distance is used for these detailed results since it produces superior results compared to other string distance functions according to Section 3.7.3. Following that, statistical analysis of results is presented. In Section 3.7.3, the performance of different string distance functions is compared. Finally, an empirical runtime analysis is performed in Section 3.7.4.

3.7.1 Results of Each Program Under Test

Tables 3.4 and 3.5 present the results for each program under test. Table 3.4 contains the results for *StrMax=30* and Table 3.5 includes the results of *StrMax=50*. Every number in these tables is a percentage indicating the improvement of that method against RT as

$$\underset{(\%)}{\text{improvement}} = \frac{\text{p-measure (X)} - \text{p-measure (RT)}}{\text{p-measure (RT)}} \times 100 \tag{3.9}$$

where X denotes a test case generation method. Further, raw p-measure results for RT method are provided in Table 3.6 which allows the reader to compute the p-measure of each method if required.

According to these tables, the MOGA outperformed RT in most of the programs under test with selected test set sizes and *StrMax* sizes. On average, the MOGA has the best test generation performance. The GA is the second best method. The ARTOO and FSCS are next; and finally RT has the lowest failure detection efficiency since every method outperformed RT on average. As the size of the test sets increase, the average results of

each test generation method is reduced and they are pushed closer to RT's performance. However, MOGA maintains its superior performance. Among the programs under the test, normally the "PostCode" and "Numeric" reveal the best failure detection improvement over RT. In contrast, the URI program performance is superior for RT.

Table 3.4. The p-measure improvement percentage of each method over RT where maximum string size is 30 and Levenshtein distance is used.

Testset Size	Software Under Test	FSCS	ARTOO	GA	MOGA
10	Validation	23.6	54.3	69.8	77.4
	PostCode	42.0	104.3	108.5	112.3
	Numeric	109.6	238.1	255.7	254.9
	DateFormat	110.2	238.5	258.2	258.2
	MIMEType	3.1	-20.4	8.7	18.8
	ResourceURL	19.4	4.9	-0.2	16.6
	URI	-16.6	-16.9	-21.6	-15.4
	URN	-38.4	-23.2	15.7	28.2
	TimeChecker	-18.2	-19.0	-29.4	-22.0
	Clocale	160.8	165.5	101.5	188.0
	Isbn	21.5	-52.6	23.9	33.8
	BIC	32.7	-14.2	43.6	46.4
	IBAN	36.2	-9.1	36.9	27.3
	Average	**37.4**	**50.0**	**67.0**	**78.8**
20	Validation	2.0	5.2	25.1	27.2
	PostCode	16.3	55.3	55.9	57.5
	Numeric	32.1	107.4	105.8	105.5
	DateFormat	32.7	106.9	107.7	107.7
	MIMEType	14.1	-12.4	-7.1	15.9
	ResourceURL	14.8	3.2	0.8	0.0
	URI	-5.1	-6.5	-11.7	-3.7
	URN	-28.9	-16.1	26.9	55.1
	TimeChecker	-4.4	-9.2	-12.5	-4.2
	Clocale	77.3	81.2	35.3	72.4
	Isbn	38.1	-52.4	29.8	32.0
	BIC	34.9	-24.4	23.6	27.5
	IBAN	23.7	-10.5	23.2	15.7
	Average	**19.1**	**17.5**	**31.0**	**39.1**
30	Validation	15.7	-4.8	20.0	17.1
	PostCode	13.9	34.4	35.4	36.0
	Numeric	23.6	61.5	59.3	58.9
	DateFormat	23.9	60.8	60.9	60.9
	MIMEType	-3.1	-15.6	1.3	9.0
	ResourceURL	9.6	-5.5	-2.0	-3.7
	URI	-2.9	-4.6	-3.8	-2.3
	URN	-20.6	-27.4	31.7	28.8
	TimeChecker	-1.7	-5.0	-3.7	-2.7
	Clocale	43.5	44.8	9.6	33.6
	Isbn	29.9	-52.1	11.4	18.9
	BIC	21.4	-23.2	12.6	15.9
	IBAN	15.1	-11.1	16.0	12.8
	Average	**12.9**	**4.0**	**19.1**	**21.8**

Table 3.5. The p-measure improvement percentage of each method over RT where maximum string size is 50 and Levenshtein distance is used.

Testset Size	Software Under Test	FSCS	ARTOO	GA	MOGA
10	Validation	33.8	49.9	73.0	94.6
	PostCode	54.5	111.2	134.6	137.0
	Numeric	204.8	364.0	458.0	457.0
	DateFormat	205.2	364.3	462.1	462.1
	MIMEType	-8.1	-19.3	-1.9	11.3
	ResourceURL	-6.8	-17.2	6.3	13.3
	URI	-18.0	-13.7	-22.5	-21.3
	URN	-47.6	-25.5	0.5	-21.5
	TimeChecker	-17.2	-15.8	-27.4	-23.5
	Clocale	225.3	230.6	183.1	321.0
	Isbn	37.8	-28.2	56.3	60.6
	BIC	73.6	32.1	96.6	103.8
	IBAN	90.4	54.7	77.5	77.1
	Average	**63.7**	**83.6**	**115.1**	**128.6**
20	Validation	13.9	26.5	53.3	28.4
	PostCode	32.1	83.4	85.4	86.6
	Numeric	75.9	198.6	200.0	199.7
	DateFormat	76.2	196.9	202.7	202.7
	MIMEType	2.4	-9.2	-7.6	12.0
	ResourceURL	3.5	-5.5	-2.2	15.6
	URI	-7.3	-6.2	-9.0	-6.5
	URN	-33.7	-28.7	2.5	6.9
	TimeChecker	-3.6	-5.5	-8.0	-5.1
	Clocale	138.4	142.6	73.2	136.5
	Isbn	65.6	-19.6	36.8	40.3
	BIC	66.1	13.1	43.1	55.5
	IBAN	59.0	28.5	36.3	37.2
	Average	**37.6**	**47.3**	**54.3**	**62.3**
30	Validation	8.2	4.5	26.1	21.9
	PostCode	22.1	60.3	60.6	60.7
	Numeric	43.2	122.8	119.2	118.9
	DateFormat	43.4	120.8	121.3	121.3
	MIMEType	-0.5	-11.7	8.7	5.4
	ResourceURL	2.0	-8.8	15.2	10.0
	URI	-4.8	-3.3	-5.3	-7.8
	URN	-23.4	-17.0	15.6	2.0
	TimeChecker	-0.6	-2.6	-1.6	-3.5
	Clocale	86.4	86.0	38.3	77.3
	Isbn	58.0	-24.4	32.1	48.1
	BIC	48.0	4.4	29.0	43.0
	IBAN	43.5	23.8	29.6	39.0
	Average	**25.0**	**27.3**	**37.6**	**41.3**

Table 3.6. The raw p-measure results for RT where the Levenshtein distance is used.

Software Under Test		Test set size		
		10	20	30
StrMax=30	Validation	0.003	0.005	0.008
	PostCode	0.099	0.148	0.177
	Numeric	0.079	0.136	0.176
	DateFormat	0.073	0.126	0.162
	MIMEType	0.002	0.003	0.005
	ResourceURL	0.002	0.004	0.005
	URI	0.123	0.164	0.179
	URN	0.001	0.002	0.002
	TimeChecker	0.191	0.253	0.274
	Clocale	0.028	0.049	0.064
	Isbn	0.007	0.013	0.017
	BIC	0.097	0.150	0.180
	IBAN	0.005	0.008	0.009
StrMax=50	Validation	0.003	0.005	0.007
	PostCode	0.093	0.127	0.150
	Numeric	0.050	0.093	0.128
	DateFormat	0.046	0.086	0.118
	MIMEType	0.002	0.003	0.009
	ResourceURL	0.002	0.004	0.005
	URI	0.156	0.187	0.199
	URN	0.002	0.003	0.004
	TimeChecker	0.232	0.268	0.276
	Clocale	0.020	0.036	0.049
	Isbn	0.005	0.009	0.012
	BIC	0.064	0.109	0.141
	IBAN	0.003	0.006	0.007

3.7.2 Statistical Analysis of Results

The results in Table 3.4 and 3.5 are averaged over 100 trial runs. To formally indicate the performance of each test case generation method against RT, we performed a test of statistical significance (z-test, one tailed) with a conservative type I error of 0.01 [90], similar to chapter 2. Our working hypothesis is that MOGA, GA, FSCS, and ARTOO will produce superior results compared to RT. Further, an effect size (Cohen's method [56], [57]) between each method and RT is calculated.

To perform a z-test or calculate effect size, the results must be normally distributed. According to [50], p-measure values are normally distributed. Further, we investigated the normality of the results more deeply by performing Shapiro-Wilk test [96]; it works

based on a null hypothesis that the data is normally distributed. According to the results of this test, the normality of the p-measure values cannot be rejected.

Table 3.7 represents the effect sizes where a positive value indicates that method outperformed RT. In contrast, a negative value denotes the higher performance of RT. The "*" beside an effect size demonstrates the result of the z-test where a statistically significant difference exists. Statistical analysis are only presented for *StrMax=30* as the results for *StrMax=50* are similar. Results in Table 3.7 indicate that in most of the experiments MOGA statistically significant outperforms RT. However, the results of FSCS, ARTOO, and GA methods are not as good as MOGA.

Table 3.7. The effect size between RT and other methods where the maximum string size is 30 and Levenshtein distance is used. "*" indicates the result of the z-test where a significant difference exists at the 0.01 level.

Testset Size	Software Under Test	FSCS	ARTOO	GA	MOGA
10	Validation	0.84*	1.83*	2.36*	2.02*
	PostCode	4.61*	14.59*	16.17*	17.15*
	Numeric	6.28*	17.36*	20.90*	20.83*
	DateFormat	6.23*	17.25*	20.93*	20.93*
	MIMEType	0.10	-0.65*	0.28*	0.54*
	ResourceURL	0.57*	0.15	-0.01	0.46*
	URI	-2.56*	-2.52*	-3.32*	-2.20*
	URN	-1.00*	-0.60*	0.37*	0.69*
	TimeChecker	-2.54*	-2.70*	-4.27*	-2.94*
	Clocale	12.63*	13.69*	7.94*	17.93*
	Isbn	1.21*	-3.43*	1.35*	1.87*
	BIC	3.47*	-1.54*	4.74*	4.88*
	IBAN	2.85*	-0.73*	2.96*	2.07*
20	Validation	0.10	0.25*	1.15*	1.19*
	PostCode	2.58*	12.45*	12.61*	13.07*
	Numeric	2.9*	14.04*	13.96*	13.92*
	DateFormat	2.91*	13.84*	14.05*	14.05*
	MIMEType	0.64*	-0.54*	-0.31*	0.64*
	ResourceURL	0.63*	0.14	0.03	0.00
	URI	-1.51*	-1.92*	-3.14*	-1.16*
	URN	-0.87*	-0.49*	0.75*	0.87*
	TimeChecker	-1.31*	-2.54*	-2.98*	-1.17*
	Clocale	11.56*	12.71*	4.35*	10.80*
	Isbn	3.56*	-5.09*	2.47*	2.88*
	BIC	7.70*	-4.24*	4.80*	5.37*
	IBAN	2.99*	-1.21*	3.12*	1.98*
30	Validation	0.66*	-0.22	0.66*	0.75*
	PostCode	3.05*	9.65*	9.99*	10.22*
	Numeric	3.16*	10.56*	10.22*	10.16*
	DateFormat	3.16*	10.34*	10.39*	10.39*
	MIMEType	-0.17	-0.87*	0.07	0.52*
	ResourceURL	0.54*	-0.31*	-0.11	-0.21
	URI	-1.42*	-2.36*	-1.91*	-1.16*
	URN	-0.58*	-1.04*	1.13*	1.05*
	TimeChecker	-0.89*	-2.27*	-1.52*	-1.29*
	Clocale	8.77*	9.21*	1.57*	6.34*
	Isbn	3.81*	-6.30*	1.36*	2.30*
	BIC	7.83*	-5.64*	3.94*	4.97*
	IBAN	3.03*	-1.91*	3.18*	2.54*

3.7.3 Comparison of String Distance Functions

Figures 3.2 and 3.3 represent the p-measure result for all six string distance functions that are discussed in Section 3.4. The results for *StrMax=30* and *50* are demonstrated in Figures 3.2 and 3.3, respectively. In each of these figures, four graphs are presented where the first three relate to the three test set sizes (10, 20, and 30) and the last one is the average of all test set sizes.

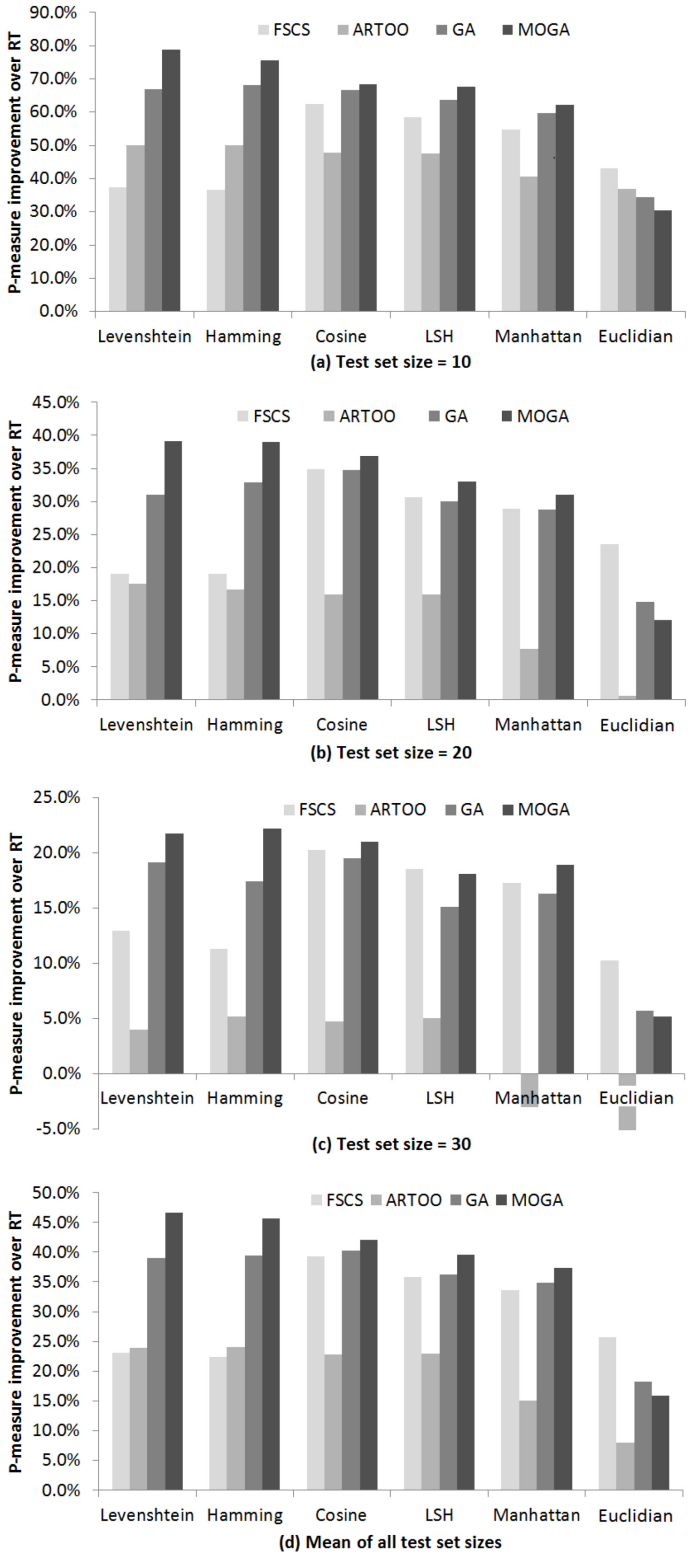


Figure 3.2. (a) Comparison of string distance functions where maximum string size is 30. Each column denotes p-measure improvement of each test case generation method over RT. (a), (b), and (c) represent results for test set sizes of 10, 20, and 30, respectively. (d) presents the mean of all test set sizes.

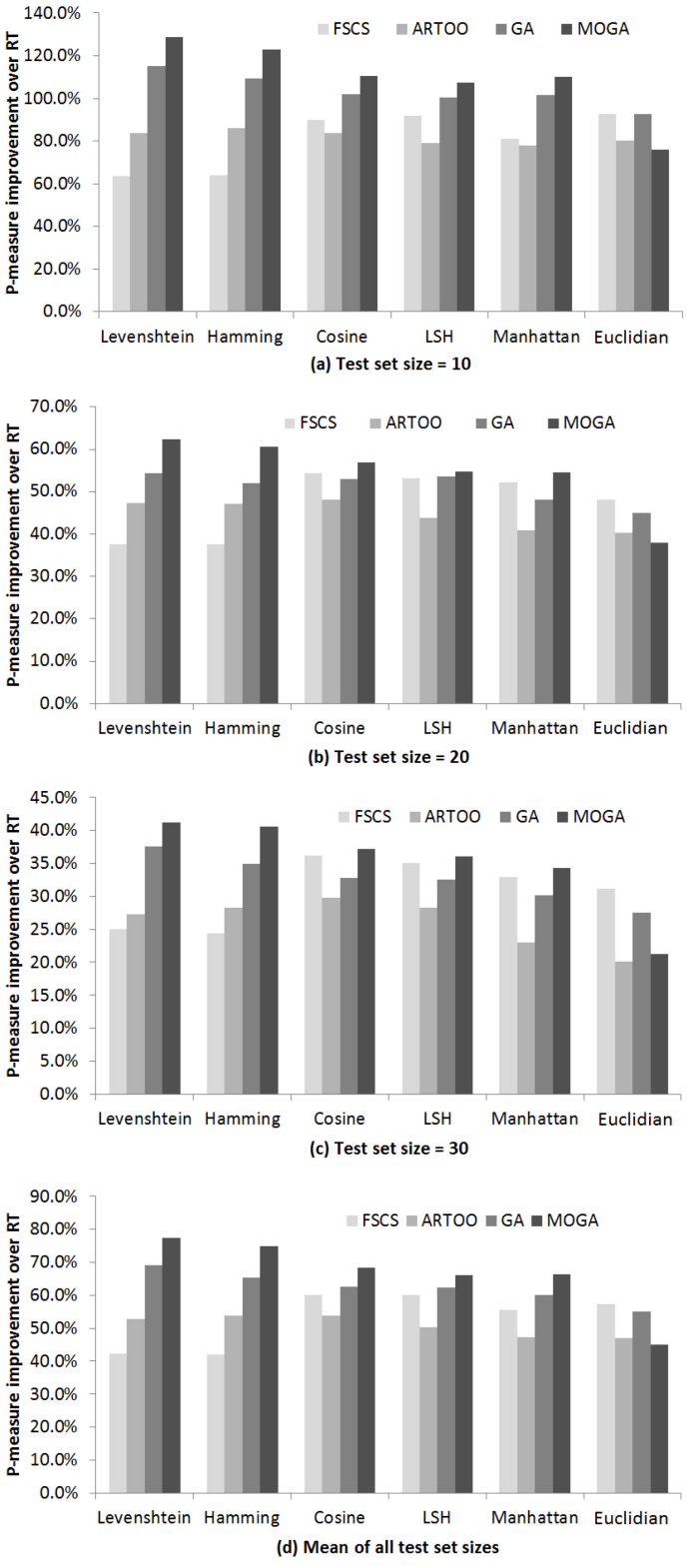


Figure 3.3. Comparison of string distance functions where maximum string size is 50. Each column denotes p-measure improvement of each test case generation method over RT. (a), (b), and (c) represent results for test set sizes of 10, 20, and 30, respectively. (d) presents the mean of all test set sizes.

According to these graphs, the MOGA test case generation method with the Levenshtein distance function has the superior failure detection effectiveness, except for one case (Figure 3.2.c). After the Levenshtein, the Hamming distance function was normally "second best" and then, the Cosine distance. As discussed before, the LSH that we used is a fast estimation of the Cosine distance; and hence, it has slightly lower failure detection effectiveness than the Cosine distance according to Figures 3.2.d and 3.3.d. Comparing the FSCS and the ARTOO in Figures 3.2.d and 3.3.d demonstrates that the ARTOO test case generation method outperforms the FSCS when the Levenshtein and the Hamming distances are used. However, the opposite is true with other distance functions. Finally, the Euclidian distance function has the lowest performance on average with respect to failure detection.

3.7.4 Empirical Runtime Analysis

In addition to failure detection effectiveness, the computational cost of an algorithm is an important factor in practical applications. The runtime order of different string distance functions and test generation algorithms are investigated in Section 3.5. To further empirically study the runtime, we design a few experiments where the effect of varying string size and test set size is investigated. The hardware platform that is used for runtime measurements is a desktop computer with core i7-3770 (3.4 GHz) and 16 GB of Ram. Further, the runtime measurement is performed 100,000 times and the average execution times are presented.

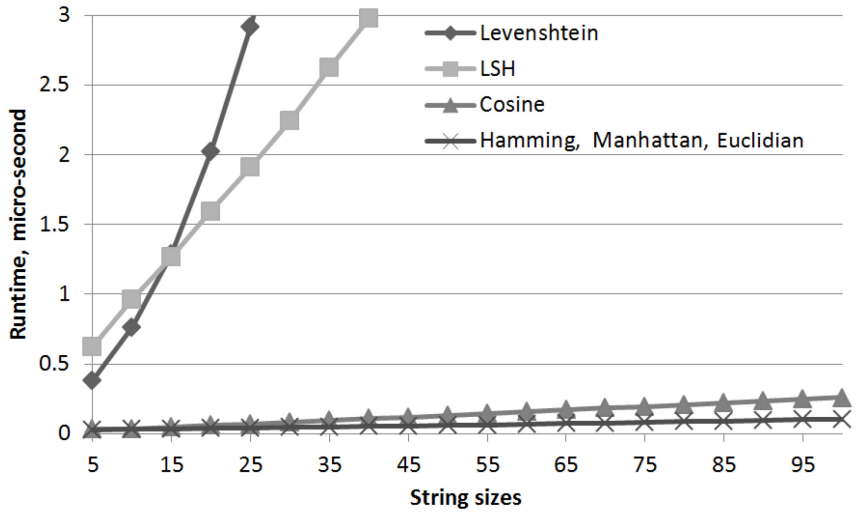


Figure 3.4. Average execution time for different distance functions with string sizes between 5 and 100.

Figure 3.4 represents the string distance calculation runtime with respect to different string sizes. String sizes between 5 and 100 with step size of 5 have been investigated where the strings used in a distance function are generated randomly. In this figure, the runtime of Hamming, Manhattan, and Euclidian distance functions are presented with a single line as they were very close. According to Figure 3.4, all the distance functions, except the Levenstein distance, have a linear runtime as string sizes increase. The Levenstein distance function has a quadratic runtime order. The runtime result for LSH is the summation of both parts of the LSH calculation as explained in Section 3.5. According to Figure 3.4, the LSH runtime is significantly higher than Cosine, Hamming, Manhattan, and Euclidian distance functions. However, LSH can outperform the runtime of other distance functions when used in test generation. In the diversity-based fitness function, a distance between every string pair in a test set needs to be calculated. With the LSH, the hash value of each string is calculated once and then, each string pair distance calculation can be done in constant time. That is, to calculate distance of two strings, a hamming distance between two fixed size bit streams must be calculated. It is argued in the details in Section 3.5.

To demonstrate the runtime advantage of LSH compared to the other distance functions in string test generation, Figure 3.5 is presented. Figure 3.5.a demonstrates the runtime of the diversity-based fitness function where the test set size is changing. According to this figure, LSH has a lower runtime than Cosine with test set size larger than about 10. Further, as test set size increases, the LHS run time becomes lower than the Manhattan (test set size larger than 25) and Euclidian distance function (test set size larger than 40). Further, LSH has lower runtime than the Hamming distance with test set size larger than 100 (Figure 3.5.a only contains test set sizes up to 50 since the graph details were not clear if we extended it to the test set size of 100). Finally, to generate the results in Figure 3.5.a, random string sets with maximum string size of 50 are produced as input to the fitness function. If the string sizes are increased, the runtime of LSH is further reduced relative to the other distance functions. Hence, Figure 3.5.b is presented where the max string size is set to the relatively large number of 1000. As demonstrated in Figure 3.5.b, the runtime of LSH is improved compared to other distance functions.

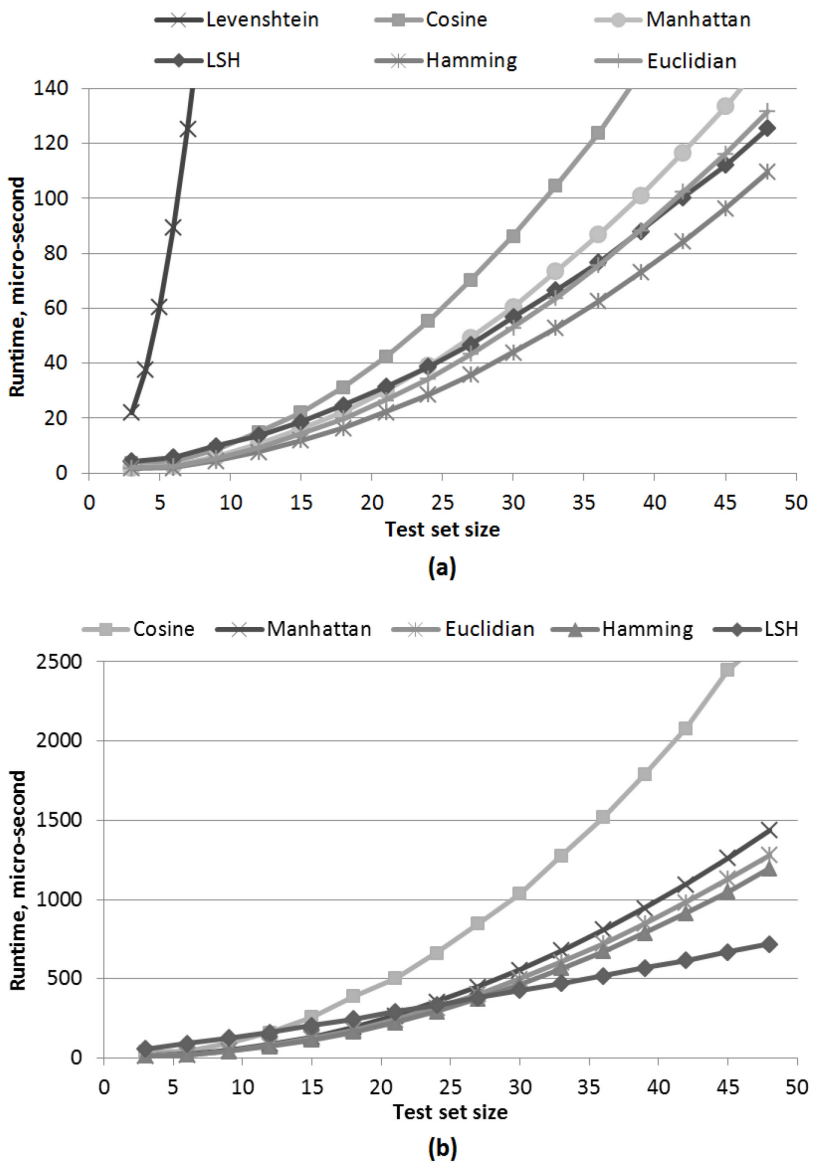


Figure 3.5. Average execution time of diversity-based fitness function with test set sizes between 3 and 50. Random string sets with maximum string size of (a) 50 and (b) 1000 are produced as input to the fitness function.

3.8 Degree of Randomness Analysis

Correlation among test cases or test sets is not good as it can potentially limit the failure detection capability if test cases correlate with a current set of defects [90]. Accordingly, we performed a similar randomness analysis as Section 2.8.

To calculate CR and NCD, we need a perfect lossless data compressor. However, a perfect compressor does not exist; and hence, we use LZMA [60]. Further, LZMA requires a large size of data to be able to compress data adequately. Accordingly, to

analyze the randomness, we generated test sets of an arbitrary large size of 1,000. The CR and NCD are calculated for all the test generation methods (RT, FSCS, ARTOO, GA, and MOGA) where each test generation method is executed with all the distance functions.

The calculated NCD values for all cases are between 0.995 and 0.997 which indicates that no correlation exists between test sets generated in different runs of the test generation methods; and hence, they are perfect in this regard. Similarly, the calculated CR values are between 1.020 and 1.026 demonstrating that test cases in a test set are completely uncorrelated; and hence, all methods produce perfect test cases with respect to randomness within test set. Theoretically, $0 \leq CR(T) \leq 1$. However, since LZMA is not a perfect compressor a small additive value is produced during the compression; and hence, CR values are slightly larger than one.

In conclusion, the randomness among test sets and within a test set is perfect for all the investigated test generation methods. That is, all the test generation methods have similar randomness as RT.

3.9 Related works

In this section, we review the related work which appears in the literature with respect to string test cases.

A category of related work is white-box string test case generation where a string test set is generated to maximize the code coverage. Research in this area normally generates a test case using an evolutionally optimization technique [69] or symbolic execution [68], [97] to cover a certain path or branch. This process is repeated until maximum number of possible branches is covered by the generated test cases. For example, Harman and McMinn [69] used a few optimization algorithms to produce a test set with maximum branch coverage. Hill climbing, GA, and memetic (hybrid GA and hill climbing) are utilized to generate a test case that covers a certain branch. Therefore, each branch in the source code requires a separate run of the test generation algorithm [69]. A fixed length array of numbers are used as a test case where it is converted to string, array, array list, number, etc., according to the specification of the program under the test. Hence, a string is a fixed length array of characters in this work [69]. In addition, Harman et al. [98] introduce a multi-objective branch coverage test case generation approach where the NSGA-II algorithm is used. The objectives are branch coverage and dynamic memory usage [98]. Fraser et al. [99] integrate a memtic optimization algorithm with the EvoSuite

tool [100] to improve test case generation. A test case is a sequence of method calls where they generated strings and numbers as functions parameters. In Fraser et al. [99], during each run of the evolutionary algorithm, a set of test cases are generated rather than a test case. The objective function is to maximize the code coverage.

Further, Afshan et al. [95] focus on the human readability of string test cases. A white-box evolutionary technique is used to generate a test case per branch. Then, a language model is utilized to modify the string to make it more readable while maintaining the covered branch. Similarly, McMinn et al. [92] and Shahbaz et al. [101] focus on the readability of string test cases. A method was proposed to query the web for common string types like emails [92], [101]. Since web content is produced by humans, strings found from the web are more likely to be human readable than machine generated strings. This method requires a set of keywords from the tester as search keywords [92], [101]. Alshraideh and Bottaci [85] also use GAs to generate string test cases where program-specific search operators (mutation and crossover in GA) are used. Similar to Harman and McMinn [69], in each run of the algorithm, a test case is produced that covers a certain branch. Initial strings are generated randomly. The size is between 0 and 20. Characters are from the ASCII range of 0-127 [85]. They also defined a "English-like" mutation operator that inserts a character into the string according to the letters that precede and follow the insertion point [85].

Symbolic execution [68], [97] is also a white-box test case generation technique that uses static analysis of source code and constraint solving to produce test cases maximizing code coverage. Further, symbolic execution is combined with concrete execution to create more powerful test generation methods. Hampi [68] is a string constrain solver tool introduced by Ganesh et al. [68]. It accepts constraints in a specific format and finds values satisfying the constraints. It is used in many symbolic execution research projects [68]. Ganesh et al. [68] use Hampi in static and dynamic analysis to find SQL injection vulnerabilities. Saxena et al. [97] introduce a symbolic execution tool for JavaScript where static analysis of source code is performed to generate string test cases.

The main difference between all these articles and the current study is that our work is a black-box approach; and hence, the test generation algorithm is independent from the source code.

Tonella [94] introduce a method to generate test cases where a test case is a sequence of

method calls. The relevant part of this work to the current study is Tonella's [94] approach in generating strings for function calls. To generate a string, a simple black-box approach is used where a character is uniformly selected from possible choices and added into the string. The possible choices are alphanumeric values (a-z, A-Z, and 0-9) [94]. The next character is inserted with the probability of 0.5^{n+1} where n is the current length of the string [94]. This implies a logarithmic reduction in the sizes of the produced strings. Our use of Benford distribution is similar to Tonella's choice of string generation in a notion that the probability of generating shorter strings is higher. However, the probability of string length distributions is different between the Benford distribution and Tonella's method. The major difference between Tonella's approach and our work is that Tonella produced strings randomly and hence, they are not likely to be very effective with respect to failure detection. In contrast, in our work, the diversity of the string test cases is optimized as well as the string length distribution and hence, superior string test case can be generated. Another advantage of our work compared to Tonella's work is that for each test set, we optimize the string length distribution and diversity. However, Tonella produced each string test case independent of other string test cases in the test set.

In addition to string test case generation works, there is related research on string test case selection and prioritization that use string distance functions. Although these works are out of the scope of this research as discussed earlier, we present a brief review of these works for the sake of completeness. Hemmati et al. [66] introduce a test cases selection method where test cases are encoded as strings. Accordingly, a diversity based fitness function based on a string distance function is used as the optimization objective [66]. Several optimization algorithms including GA and hill climbing were tested. Ledru et al. [67] also employ string distance functions to prioritize string test cases. Multi-objective optimization is also used for test case selection. Yoo and Harman [102] used code coverage, past fault-detection history, and the execution cost as three optimization objectives.

3.10 Summary

In this chapter, black-box string test case generation is studied. Two objectives are introduced to produce effective string test cases. The first objective controls the diversity of the test cases within a test set. According to various empirical studies [13]–[17], faults

usually occur in error crystals or failure regions. Hence, controlling the diversity of the test cases is an important aspect of black-box test case generation. The second objective is responsible for controlling the length distribution of the string test cases. The Benford distribution is employed as an objective distribution. Accordingly, a Kolmogorov–Smirnov test [84] is utilized to construct the fitness function. When both objectives are enforced, using a multi-objective optimization technique, superior test cases are produced.

Further, several string distance functions are examined as a part of test case generation process (Levenshtein, Hamming, Cosine, Manhattan, Catesian, and LSH distance functions). Among the investigated distance functions, the LSH [65] is a fast estimation of the Cosine string distance function. According to the runtime complexity analysis in Section 3.5, LSH improves the runtime complexity. Further, in Section 3.5, the runtime complexities of all test case generation methods are discussed.

An empirical study has been performed to evaluate the failure detection capability of the string test generation methods (RT, FSCS, ARTOO, GA, and MOGA). Thirteen real-world programs are used for evaluation. Several faulty versions are produced for each program through a mutation technique. These programs perform string transformation and/or manipulation which make them a true test for situations where the input test cases are strings [92]. With respect to the evaluation results, the MOGA revealed the superior failure detection performance. Further, the empirical results of comparing different string distance functions indicate that the Levenshtein distance outperformed the others.

Randomness of the test cases is an important aspect of a test case generation algorithm. Correlated test cases may reduce the failure detection effectiveness as discussed in Section 3.8. As a consequence, an investigation of randomness is performed; and it demonstrated that all the generated test cases possess an appropriate degree of randomness.

4 Extended Subtree: A New Similarity Function for Tree Structured Data

The extensive application of tree structured data in today's information technology is obvious. Trees can model many information systems like XML and HTML. User behavior in a website (visited pages) [103]–[105], proteins, and DNA can be modeled with a tree. Moreover, programming language compilers parse the code into a tree as a first step. Consequently, in many applications involving tree structured data, tree comparison is required. Tree comparison is performed by tree distance/similarity functions. The applications includes document clustering [106], natural language processing [107], cross browser compatibility [108], and automatic web testing [109].

4.1 The Focus of This Chapter

Several tree comparison approaches [110]–[113] have been already introduced to address this domain. Edit base distances [112] are a well-known family of tree distances based on mapping and edit operations. They have three major drawbacks with respect to their mapping rules. First, order-preserving rules may prevent mapping between similar nodes, resulting in situations where similar nodes may not contribute towards the overall tree similarity score based solely upon their position. Second, according to the one-to-one condition, any node in a tree can be mapped into only one node in another tree leading to inappropriate mappings with respect to similarity. That is, repeated nodes or structures of mapped nodes have no effect on similarity and they are counted as dissimilar nodes. Finally, edit based distances work based upon mapping individual nodes, not tree structures. This implies that every mapped pair of nodes is independent of all the other nodes. However, a group of mapped nodes should have a stronger emphasis on the similarity of trees when they form an identical subtree. That is, an identical subtree represents a similar substructure between trees, whereas disjoint mapped nodes indicate no similar structure between the two trees. More details of these drawbacks along with illustrative examples are presented in Section 4.4.1.

In this chapter, we propose a new similarity function with respect to tree structured data, namely Extended Subtree (EST). The new similarity function avoids these problems by preserving the structure of the trees. That is, mapping subtrees rather than nodes is utilized by new mapping rules. The motivation of proposing EST is to enhance the edit

base mappings, provided in Section 4.3.1, by generalizing the one-to-one and order preserving mapping rules. Consequently, EST introduces new rules for subtree mapping. This new approach seeks to resolve the problems and limitations of edit based approaches (this is detailed in Section 4.4.1 with illustrative examples).

To evaluate the performance of the proposed similarity function against previous researches, an extensive experimental study is performed. The experimental evaluation frameworks include clustering and classification frameworks. The distance functions provide the core functionality for clustering and classification applications. In addition, four distinct data sets (three real and one synthetic) are utilized to perform the evaluation.

In general, this chapter's contributions can be summarized as:

- Introducing a novel similarity function to compare tree structured data by defining a new set of mapping rules where subtrees are mapped rather than nodes.
- Further, the new approach resolves the limitations of the previous distance functions.
- Designing extensive evaluation frameworks using k-medoid [114], KNN (K Nearest Neighbor) [115], and SVM (Support Vector Machine) [116] along with four different data sets to perform an unbiased evaluation. That is, we believe applying one machine learning technique on a single data set might lead to a biased evaluation; and hence, it is not considered adequate to prove the effectiveness of an approach. This extensive evaluation framework is one of the advantages of this research over previous researches [103], [105], [106], [117], [118].
- Superior results of EST against previous approaches in most of the clustering and classification case studies.
- Empirical runtime analysis of the new approach as well as current approaches where runtime efficiency of EST is demonstrated.

4.2 Notation and Definitions Used in This Chapter

The following notation and assumptions are provided with respect to trees to simplify the discussion in this chapter. In this chapter, trees are referring to rooted, ordered, and labeled trees unless otherwise stated. A rooted tree is a tree with a single root node. A tree is ordered if right-left order amongst sibling nodes in the tree is important. Finally, a labeled tree represents a tree where each node has an assigned label.

A tree is denoted as T and $|T|$ indicates the size of a tree in terms of the number of nodes/vertices. Multiple trees are differentiated by a top index as T^p and T^q. t_i represents the *i*th node of T numbered in a post-order format. In case of multiple trees, again a top index is utilized to distinguish between trees. For instance, t_i^p and t_i^q. In this chapter, $V(T)$ defines a set of vertices/nodes of T where $V(T) = \{t_i\}_{i=1}^{|T|}$. The depth of a tree is denoted by $depth(T)$ which is defined as the length of the path from the root to the deepest node in the tree. $depth(t_i)$ indicates the length of the path from the root to t_i. $leaves(T)$ indicates the number of leaves in T where a leaf node is the node without children. $\deg(t_i)$ represents the degree of node t_i which is equal to the number of t_i's children. Accordingly, $\deg(T)$ represents the degree of T, which is the maximum number of children of any node in the tree. A subtree is a tree which is part of a larger tree. Accordingly, T_i denotes a subtree of T rooted at t_i. If T^p is a subtree of T^q, we indicate it as $T^p \subset T^q$.

Finally, distance and similarity between T^p and T^q are presented as $D(T^p, T^q)$ and $S(T^p, T^q)$, respectively. Similarly, normalized values are indicated by D^* and S^* where we have

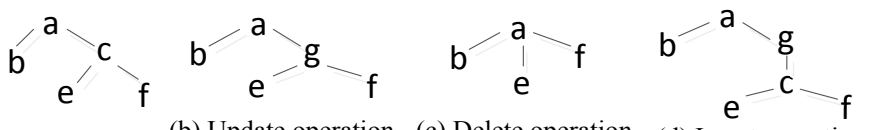
$$S^*(T^p, T^q) = 1 - D^*(T^p, T^q) \tag{4.1}$$

4.3 Current Approaches

A variety of different tree distance functions have been proposed. In this section, we survey these approaches and present a summary of each one.

4.3.1 Edit Based Distances

Edit based distances [112] are based on three edit operations (γ) including "delete", "insert", and "update" [119] (Figure 4.1). Each operation has an associated cost ($W_{delete}, W_{insert}, W_{update}$). Based on the introduced edit operations, each tree can be converted into another tree according to a set of rules that are different for each distance function. Further, mappings were introduced in [120] to describe how a sequence of edit operations converts a tree into another tree [121], namely T^p and T^q respectively.



(a) T (b) Update operation, $\gamma(c \to g)$ (c) Delete operation, $\gamma(c \to \varnothing)$ (d) Insert operation, $\gamma(\varnothing \to g)$

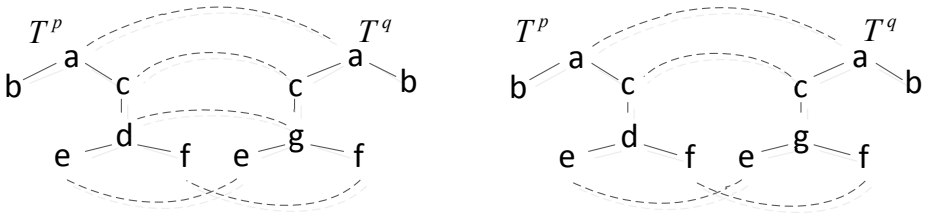
Figure 4.1. Three edit operations, "delete", "insert", and "update".

Figure 4.2 represents a sample T^p and T^q along with a few mappings where each mapping represents an optimal mapping associated with a tree distance approach. A mapping is a set of ordered integers such as (i_p, i_q) where i_p and i_q are the index of the nodes (numbered in post-order format) from tree T^p and T^q, respectively. This means that node t_{ip}^p is mapped into node t_{iq}^q. The following conditions must be satisfied for all $(i_p, i_q), (j_p, j_q) \in M$ [121]:

- One-to-one condition: $i_p = j_p$ if and only if $i_q = j_q$. This condition implies that one node from T^p cannot be mapped into two nodes from T^q.
- Sibling order preservation condition: $i_p > j_p$ if and only if $i_q > j_q$.
- Ancestor order preservation condition: t_{ip}^p is an ancestor of t_{jp}^p if and only if t_{iq}^q is an ancestor of t_{jq}^q.

$D(T^p, T^q)$ is equal to the cost of the edit operations required to convert T^p into T^q. Assuming the cost of each edit operation as one, the $D(T^p, T^q)$ is bounded between zero and $|T^p| + |T^q|$. Accordingly, it can be normalized between zero and one as:

$$D^*(T^p, T^q) = \frac{D(T^p, T^q)}{|T^p| + |T^q|} \tag{4.2}$$



(a) Tree edit distance mapping (b) Isolated subtree mapping

Figure 4.2. Optimal mappings between trees for TED and IST.

4.3.1.1 Tree Edit Distance (TED)

TED [119], [120], [122] is a well-known edit based distance function that measures the minimum cost of a sequence of edit operations between two trees. Since its introduction by Tai [120], several algorithms have been introduced for computing the optimal TED between two trees. This research follows the dynamic programming presented by Zhang and Shasha [119]. The computational order for this algorithm is $D_{TED}(T^p,T^q) \in O(|T^p| \times |T^q| \times Min(depth(T^p),leaves(T^p)) \times Min(depth(T^q),leaves(T^q)))$ [119] where $O(\bullet)$ represents the runtime order. The TED mapping needs only to satisfy the mapping's conditions presented in previous section. The mapping demonstrated in Figure 4.2a indicates an optimal mapping to calculate the TED. According to this mapping $D_{TED}(T^p,T^q)=3$, since we have only one update operation ($\gamma(d \to g)$), one insert operation ($\gamma(\varnothing \to b)$), and one delete operation ($\gamma(b \to \varnothing)$).

4.3.1.2 Isolated Subtree (IST) Distance

The IST distance is introduced by Tanaka [123], it maps the disjoint subtrees of T^p to the similar disjoint subtrees of T^q. Tanaka [123] argued that such a mapping is more meaningful since it preserves the structure of the trees. The IST mapping is a TED mapping where disjoint subtrees are mapped to similar disjoint subtrees under the restriction of the structure preserving mapping [123]. Figure 4.2b demonstrates the optimal IST mapping between T^p and T^q. In this sample, $D_{IST}(T^p,T^q)=4$. Tanaka [123] provided an algorithm to compute the optimal IST distance with the runtime complexity of $O(|T^p| \times |T^q| \times Min(leaves(T^p),leaves(T^q)))$ [123], [124]. Later, Zhang [125] provided an algorithm to calculate IST distance with runtime complexity of $O(|T^p| \times |T^q|)$.

In addition to TED and IST, there are other distance functions including alignment [126],

top-down [127], and bottom-up distance [124]. Their objective is to simplify the calculations; however, they produce lower quality solutions than TED.

4.3.2 Multisets Distance

Recently, Müller-Molina et al. [113] have introduced a tree distance metric based on multisets. Multisets are sets that allow repeated elements, where T^p and T^q are converted into multisets, M^p and M^q. M^p and M^q contain all the complete subtrees of the corresponding trees. A complete subtree is defined as a subtree that: if t_i is a node in a complete subtree, all of t_i's children are in the subtree. In addition, $V(T^p)$ and $V(T^q)$ are utilized along with M^p and M^q to calculate distance as:

$$D_{multiset}(T^p, T^q) = ((|M^p \bigcup M^q| - |M^p \bigcap M^q|) + (|V(T^p) \bigcup V(T^q)| - |V(T^p) \bigcap V(T^q)|)) / 2 \quad (4.3)$$

Müller-Molina et al. [113] presented no approach for normalization. However, the normalized distance can be calculated using (4.2) since $D(T^p, T^q)$ is bounded between 0 and $|T^p| + |T^q|$. An algorithm with runtime complexity of $O(|T^p| \times |T^q|^2)$ is presented in [113] to compute the distance.

4.3.3 Path Distance

Path distance [111] considers paths as a tree's building blocks. Each tree is converted into a multiset of paths such as "/a/c/d" which describes a path in T^p in Figure 4.2a. Different approaches exist to extract paths from a tree. One possible approach is that all paths start from a root node to t_i. Any node to any possible node is another approach where a path to t_i can start from any ancestor of t_i or even t_i. The later approach includes all the possible paths in the tree. In this research, we follow the second approach for path extraction. Given T^p and T^q, M^p and M^q are the multisets which contain all the paths in T^p and T^q, respectively. $S_{path}(T^p, T^q)$ can be simply calculated as $|M^p \bigcap M^q|$. Since $S_{path}(T^p, T^q)$ is bounded between zero and $Max(|T^p|, |T^q|)$, $S^*_{path}(T^p, T^q) = \frac{|M^p \bigcap M^q|}{Max(|T^p|, |T^q|)}$.

4.3.4 Entropy Distance

Connor et al. [110] utilized information theory, Shannon's entropy, to calculate a bounded, between zero and one, distance function between two trees. Similar to the path distance metric, the M^p and M^q multisets are generated which contain all the possible paths in T^p and T^q, respectively. Then, Shannon's entropy equation and complexity theory are used to calculate the information distance. Finally, Connor et al. [110] concluded the distance as:

$$D_{Entropy}\left(T^p, T^q\right) = \frac{C(M^p \uplus M^q)}{\sqrt{C(M^p) \times C(M^q)}} - 1 \tag{4.4}$$

where $\uplus$ represents the union of two multisets; and $C(M)$ denotes complexity of a multiset defined as [110]

$$C\left(M\right) = b^{H_b(M)} = b^{-\sum_i p(m_i)\log_b p(m_i)} = \prod_i p\left(m_i\right)^{-p(m_i)} \tag{4.5}$$

where b is a constant number, $H_b(M)$ represents the entropy of M in base b, and m_i denotes a member of M where i represents all the distinct members of M. Finally, $p(m_i)$ denotes the probability of m_i in M which is equal to the number of m_i repetitions over $|M|$. The authors did not provide the order of runtime complexity of the algorithm.

4.3.5 Other Distances

In addition to the discussed approaches, Lu [128] introduced node splitting and merging. Further, Helmer [129] utilized Kolmogorov complexity which provides a new class of distances for measuring similarity relations between sequences [23]. The main advantage of this approach is its linear runtime complexity which is reported [129] as $O(|T^p| + |T^q|)$. Finally, Yang et al. [130] introduced a distance measure between two trees based on a numeric vector representation of trees. They prove that this distance, $D_{binary}(T^p, T^q)$, is a lower bound for $D_{TED}(T^p, T^q)$ given by $D_{binary}(T^p, T^q) \leq 5 \times D_{TED}(T^p, T^q)$ and hence, it has lower quality compared to TED. However, it has a linear runtime complexity given by $O(|T^p| + |T^q|)$ which outperforms TED in this respect.

Beside the discussed tree distance functions, there are some diffing (differencing) tools for XML documents like XMLDiff [131]. The primary objective of these tools is to identify and list all the differences between two XML documents, and hence they are different with a tree distance function that produces a single number as a measure of distance. Diffing tools normally use one of the edit based approaches. For instance, Microsoft XML Diff is a tool for diffing XML documents that is implemented in .NET framework [131]. It implements the TED function. XMLDiff is another tool that is part of many Linux distributions [131]. It uses a variation of tree edit operations based on the Chawathe et al. work [132] to identify the differences. This diffing tool works based on the "move", "delete", and "insert" operations. XyDiff is another diffing algorithm introduced by Cobena et al. [133]. It works based on bottom-up tree edit model.

Code clone detection is another application that is relevant to tree distance and/or similarity functions. Clone detection has many applications like fraud detection and clone removal in order to decrease maintenance costs [134]. Code clone detection methods can be divided into a few categories; one of which is clone detection based on abstract syntax tree comparison [134] which is the most relevant to our research. Code clone detection that utilizes abstract syntax tree matching is an application of tree similarity functions. The objective in code clone detection is detecting exact or near-miss code fragments. Hence, in an abstract syntax tree, subtrees are compared with a tree similarity function. If the similarity is more than a defined threshold, the corresponding code fragments are considered to be a clone. For instance, Baxter et al. [135] use $2S/(2S+L+R)$ as a similarity function, where *S, L*, and *R* denote the number of shared nodes, different nodes in the first tree, and different nodes in the second tree, respectively.

Finally, the proposed distance function's (EST) performance is evaluated against TED, IST, Entropy, Multisets, and Path distances as no compelling evidence exists that any other superior techniques exists and no comprehensive comparison of these techniques appears in the literature. However, it should be noted that approaches such as Kolmogorov complexity [129] and Binary distance [130] have linear computational complexity; and hence, have a superior runtime to those used in the experiments.

4.4 Proposed Tree Similarity Function: Extended Subtree (EST)

In this section, we propose a new similarity function, namely EST, to compare trees. The new function seeks to resolve many of the issues which will be discussed in the following

section. Further, a computational algorithm as well as its runtime complexity is presented.

4.4.1 Motivation

In this section, we justify the need to propose a new tree comparison approach by discussing situations where previous approaches have poor performance. Note that the aim of the new approach is not runtime complexity reduction as presented in [130], [136]. Although runtime complexity is an important issue in practical applications, we focus on proposing a new approach that better represents the similarity or distance between tree-structured data. This leads to an enhancement in applications where a tree distance function is utilized.

A variety of tree comparison approaches are introduced in the previous section. Each approach has advantages and disadvantages in terms of the distance/similarity score. We found situations where the previous approaches do not give an appropriate similarity/distance score. In the following, these cases are analyzed with illustrative examples where all discussions are in terms of a normalized similarity score, $S^*(T^p,T^q)$. $S^*(T^p,T^q)=1$ means that the trees are identical; while $S^*(T^p,T^q)=0$ means that the trees are totally distinct.

All of the five edit based tree distance approaches follow the mapping rules presented in Section 4.3.1, namely one-to-one and order preserving conditions. According to the one-to-one condition, any node in T^p can only be mapped to one node in T^q. Now consider Figure 4.3a where $T^u \subset T^p$ and $T^u, T^x \subset T^q$. Also assume that $|T^u|, |T^x| \gg 1$, so the cost of the root nodes in T^p and T^q have negligible impact on the distance calculation. Considering $|T^u| = |T^x|$ in Figure 4.3a leads to $S^*(T^p,T^q) \simeq 0.667$ with respect to all five edit based approaches. There is a problem in this similarity score: no matter whether T^u and T^x are identical or totally different, $S^*(T^p,T^q)$ remains 0.667. The one-to-one mapping condition enforces that T^x cannot be mapped to $T^u \subset T^p$, since $T^u \subset T^p$ is already mapped to $T^u \subset T^p$. Moreover, according to the order preserving conditions, a node in T^p can be mapped to one node in T^q, if the ordering is preserved with other mappings. This is how edit based distances differentiate between ordered and unordered trees. This rule seems less than ideal in a number of situations. To clarify this discussion,

consider Figure 4.3b, where $T^u, T^y \subset T^p$ and $T^u, T^x \subset T^q$; again, assume that $|T^u|, |T^x|, |T^y| \gg 1$. Considering $|T^u| = |T^x| = |T^y|$ in Figure 4.3b leads to $S^*(T^p, T^q) \simeq 0.5$ with respect to all edit based approaches. The problem in this case is that whether T^x and T^y are identical or totally different, the similarity score remains at 0.5. This means that when considering T^x and T^y as identical, they cannot be mapped together due to the order preserving conditions. Please note that considering T^x and T^y as identical does not lead to $T^p = T^q$, since T^p and T^q are ordered trees. Accordingly, we are not discussing that by mapping T^y to T^x, the similarity score would be one. What we are discussing is that if $T^x = T^y$, $0.5 < S^*(T^p, T^q) < 1$ better represents the similarity between these trees. According to these discussions, we introduce a new set of mapping conditions in the next section.

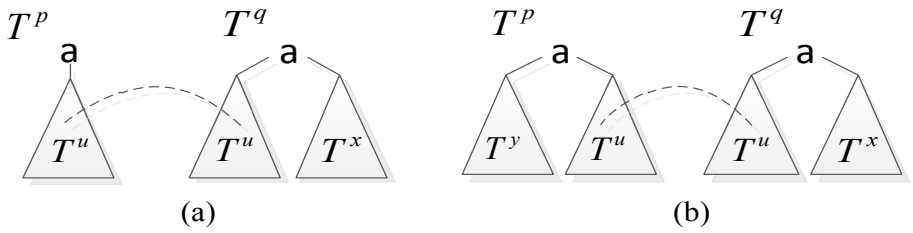


Figure 4.3. Samples of T^p and T^q utilized to problems regarding mapping conditions in edit based distances.

Further, we observed that *m* (a constant number) similar nodes between T^p and T^q have a stronger emphasis on the similarity of T^p and T^q when they form an identical subtree mapping between T^p and T^q (Figure 4.4a), compared to disjoint nodes as illustrated in Figure 4.4b. That is, an identical subtree represents a similar substructure between T^p and T^q, whereas *m* disjoint mapped nodes indicate no similar structure between the two trees. However, edit based approaches, in particular the IST distance [123], are unable to model this. That is, in the IST distance, *m* mapped disjoint nodes have the same similarity as *m* nodes forming a subtree. Figure 4.4 represents two IST mappings where $S^*(T^p, T^q) \simeq 0.6$ in both cases. However, we believe that T^p and T^q presented in Figure 4.4a are more similar than the trees presented in Figure 4.4b, since Figure 4.4a contains a similar subtree as denoted by the hatches.

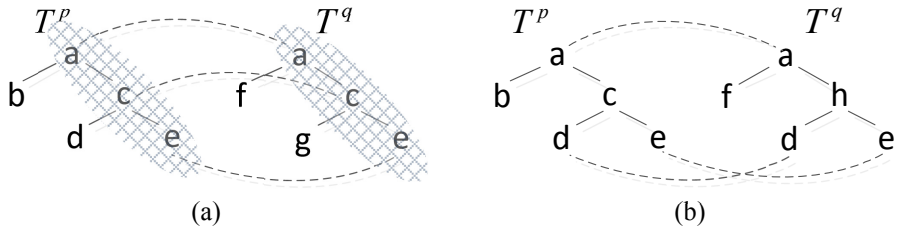


Figure 4.4. Samples of isolated subtree (IST) mappings where (a) the mapped nodes form a subtree as denoted by the hatches; and (b) the mapped nodes are separate nodes.

Path [111] and entropy [110] distances consider paths as a tree's building blocks as their basic assumption; that is, they convert a tree into a multiset of paths and then compare the trees by comparing the multisets of paths. This assumption is not in accordance with the nature of tree-structured data. If we could convert a tree into a multiset of paths, there would have been no reason to present the data initially as a tree. Further, the entropy approach produces some strange results. Assuming the trees presented in Figure 4.3a with the aforementioned conditions regarding T^u and T^x, the entropy approach yields $S^*(T^p, T^q) \simeq 1$ where $T^x = T^u$. Obviously, this result is unsatisfactory as T^p and T^q are not identical.

The binary [130] and Fourier [136] distances assume TED as an ideal distance approach and approximate TED while reducing the runtime complexity. Fourier distance converts a tree to a signal in the frequency domain. The poor performance of Fourier distance, presented in [111], verifies that it is not an appropriate tree comparison approach. The bottom-up approach [124] puts more value on bottom nodes rather than top nodes, since it matches the bottom nodes first. Therefore, this approach is not performing well in most of the situations where nodes have equal weights or where top nodes have larger weights. Based on our empirical investigation, the multiset approach [113] behaviour is similar to the bottom-up approach in terms of putting more value on bottom nodes; that is, every subtree defined in this approach contains leaves of the tree. Finally, the NCD approach [129] does not seem an appropriate distance metric, since it converts the tree into plain text where each node's label is converted to text. Just as an example to demonstrate a disadvantage of this approach, assume that different nodes are labeled with different numbers in a tree like 2, 111, and 1111. All the three labels are different, but since the labels are converted into plain text, 111 and 1111 are considered similar in the compression process utilized in NCD. Further, since an optimal compressor does not

exist, a real world compressor is utilized which does not yield optimal NCD scores.

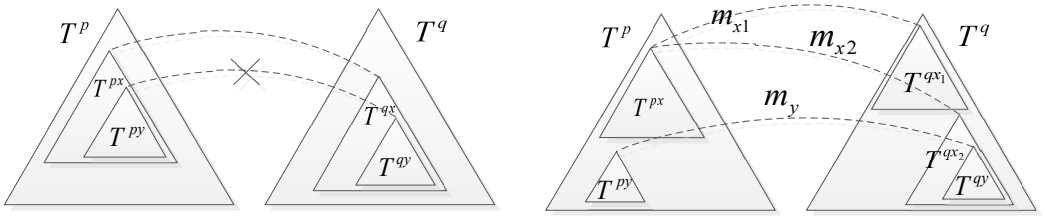
As a conclusion, the main motivation for proposing a new tree similarity approach is introducing an approach which resolves the discussed problems and removes the limitations of the previous approaches. In addition, the new approach must enhance the applications where a tree distance function is utilized.

4.4.2 Extended Subtree (EST) Similarity

Given T^p and T^q, the proposed EST preserves the structure of the trees by mapping subtrees of T^p to similar subtrees of T^q. Although it might seem similar to the IST, it is fundamentally different since EST's mappings are not in accordance with the mapping conditions provided in Section 4.3.1. That is, EST generalizes the edit base distances and mappings. According to the discussions in the previous section, given T^{px} and T^{qx} as two mapped subtrees in T^p and T^q with m_x as the name of this mapping, we introduce the rules of the new approach's mapping as:

Rule 1: EST's mapping is a subtree mapping which means that not only single nodes can be mapped together, but also identical subtrees can be mapped together (unlike IST). Using subtree mapping, we can increase the significance of larger subtrees, since they are considered more important than single nodes in accordance with the discussion in the previous section.

Rule 2: No common subtrees of T^{px} and T^{qx} are allowed to be mapped together, as indicated in Figure 4.5a, this is defined as an invalid mapping. When two subtrees of T^{px} and T^{qx} are already mapped, all the sub structures of T^{px} and T^{qx} can be mapped together as T^{px} and T^{qx} are identical. Since we are interested in larger mapped subtrees, mapped subtrees of T^{px} and T^{qx} have no use, so we categorize them as invalid mappings.



(a) Invalid mapping (b) Valid mapping

Figure 4.5. Extended Subtree (EST) mapping where (a) indicates invalid mappings, and (b) represents valid mappings.

Rule 3: One-to-many condition: A subtree of T^p can be mapped into several subtrees of T^q and vice versa. The intuition of this rule is with respect to Figure 4.3a where the disadvantages of the one-to-one condition are investigated. As indicated in Figure 4.5b, T^{px} is mapped to T^{qx1} and T^{qx2} concurrently. Further, T^{qy} is mapped into T^{py} where T^{qy} is a subtree of T^{qx2} which is already mapped.

Rule 4: m_x is weighted as $W(m_x) = (W(T^{px}) + W(T^{qx}))/2$ where $W(T^{px})$ and $W(T^{qx})$ are the weights of subtrees in the mapping. $W(T^{px})$ (and similarly for $W(T^{qx})$) is calculated as:

$$W\left(T^{px}\right) = \sum_{t_i^{px} \in T^{px}} W\left(t_i^{px}\right) \tag{4.6}$$

where $W(t_i^{px})$ is the unit scalar, when T^{px} is the largest subtree that t_i^{px} belongs to; and zero otherwise. A node like t_i^{px} might be a member of several subtrees in the mappings as indicated in Figure 4.5b. However, it is inappropriate to multiply-count the same node; therefore, nodes are counted as a weight just for the largest subtree that they belong to.

Finally, we can compute $S(T^p, T^q)$ based on all the possible valid mappings as:

$$S\left(T^p, T^q\right) = \sqrt[\alpha]{\sum_{m_k \in M} \beta_k \times W\left(m_k\right)^\alpha} \tag{4.7}$$

where α, $\alpha \geq 1$, is a coefficient to adjust the relation among different sizes of mappings. It amplifies the importance of large subtrees compared to small subtrees or single nodes in accordance with the discussion in the previous section. This similarity function has obvious parallels with the Minkowski distance function [137] which is a popular distance function for higher dimensions of data. $\alpha = 1$ does not amplify the importance of large subtrees compared to small subtrees. As α grows larger, more emphasis is placed on larger subtrees. Further, β_k is a geometrical parameter which reflects the importance of the mapping with respect to the position of T^{pk} and T^{qk} in T^p and T^q, respectively. β_k is the unit scalar, when the root nodes of T^{pk} and T^{qk} have the same depth with

respect to T^p and T^q; and it is equal to β (a constant number between zero and one) otherwise; leading to the amplification of the mapping of the same depth regarding subtrees. The selection of α and β values are discussed in Section 4.5.5.

To normalize the similarity score, we divide it by its higher bound. Since $0 \le \beta_k \le 1$, we have $S\left(T^p, T^q\right) \le \sqrt[\alpha]{\sum_{m_k \epsilon M} W(m_k)^\alpha}$. Further, $\sqrt[\alpha]{\sum_{m_k \epsilon M} W(m_k)^\alpha} \le \sum_{m_k \epsilon M} W(m_k)$ where $\alpha \ge 1$ and $W(m_k)$ is a positive number. In addition, each node is counted as one in the weight calculation, $\sum_{m_k \in M} W(m_k) \le Max(|T^p|, |T^q|)$. As a result, $S(T^p, T^q) \le Max(|T^p|, |T^q|)$ and the similarity function is normalized as:

$$S^*(T^p, T^q) = \frac{S(T^p, T^q)}{Max(|T^p|, |T^q|)} \tag{4.8}$$

In the example provided in Figure 4.5b, consider the presented mappings as the only valid mappings. In addition, assume $|T^{px}| = |T^{qx1}| = |T^{qx2}| = 5$ and $|T^{py}| = |T^{qy}| = 2$. Therefore, mapping weights can be computed as $W(m_{x1}) = 5$, $W(m_{x2}) = 2.5$, and $W(m_y) = 1$. Accordingly, if we consider $\alpha = 2$ and $\beta = 1$, the similarity score is $S(T^p, T^q) = \sqrt{5^2 + 2.5^2 + 1^2} = 5.679$. Consequently, considering $|T^p| = 8$ and $|T^q| = 10$, the normalized similarity score is $S^*(T^p, T^q) = 0.568$.

4.4.3 Computational Algorithm

Assume $T^p_{i,j}$ represents a subtree of T^p rooted at t^p_i which is mapped to an identical subtree of T^q rooted at t^q_j, namely $T^q_{j,i}$. Accordingly, computing $S(T^p, T^q)$ has four following steps.

Step 1: *Identify all the mappings*: In this step, we find all the possible mappings, valid or invalid (in Step 3, invalid mappings will have a zero weight), and store two lists of nodes for each mapping, one for each subtree. T^p and T^q are the inputs to this step and V^p and V^q are the outputs (inputs for the next step). V^p and V^q are two dimensional matrices where each element is a list of nodes. Accordingly, $V^p[i][j]$ and $V^q[j][i]$ represent the list of nodes of the mapped subtrees of $T^p_{i,j}$ and $T^q_{j,i}$, respectively. The

pseudo code represented in Figure 4.6 details this step's calculations. The *GetMapping(i, j)* function produces two lists of nodes ($V^p[i][j]$ and $V^q[j][i]$) for a mapping. Its objective is to detect the largest possible mapping. To achieve this objective, we need to find and match the mappings rooted at the children of t_i^p and t_j^q. Since *i* and *j* are node indexes in post-order formatting, when computing *GetMapping(i, j)* for nodes t_i^p and t_j^q, the computation is already performed for all the children of t_i^p and t_j^q in advance. Therefore, as indicated in the pseudo code, the *GetMapping(i, j)* function goes through all of the children of t_i^p and t_j^q to use the mapping information among t_i^p's and t_j^q's children to find the largest mapping between t_i^p and t_j^q. t_{ia}^p denotes the *a*th child of the t_i^p node, where $1 \le a \le \deg(t_i^p)$, and *ia* represents the index of the *a*th child of the t_i^p node. Similarly, t_{jb}^q represents the *b*th child of the t_j^q node, where $1 \le b \le \deg(t_j^q)$ and *jb* represents the index of the *b*th child of the t_j^q node. In Figure 4.6, *E* is a matrix which indicates how the children of t_i^p and t_j^q are matched. Accordingly, *E* is used to update $V^p[i][j]$ and $V^q[j][i]$. Since $T_{i,j}^p$ and $T_{j,i}^q$ are identical, $|V^p[i][j]| = |V^q[j][i]|$, so $|V^p[i][j]|$ can be replaced by $|V^q[j][i]|$ in the pseudo code.

Step 1

Begin
for $i = 1$ to $|T^p|$ **do**
for $j = 1$ to $|T^q|$ **do**
if $label(t_i^p) == label(t_j^q)$ **then**
GetMapping(i, j)
end of if
end of for
end of for

Step 2

for $i = 1$ to $|T^p|$ **do**
for $j = 1$ to $|T^q|$ **do**
for $k = 1$ to $|V^p[i][j]|$ **do**
$i' \leftarrow V^p[i][j]_k, j' \leftarrow V^q[j][i]_k$
if $|V^p[i][j]| > |V^p[LS^p[i']_{mi}][LS^p[i']_{mj}]|$ **then**
$LS^p[i']_{mi} = i, LS^p[i']_{mj} = j$
end of if
if $|V^q[j][i]| > |V^q[LS^q[j']_{mj}][LS^q[j']_{mi}]|$ **then**
$LS^q[j']_{mi} = i, LS^p[j']_{mj} = j$
end of if
end of for
end of for
end of for

Step 3

for $i = 1$ to $|LS^p|$ **do**
$W^p[LS^p[i]_{mi}][LS^p[i]_{mj}]++$
end of for
for $j = 1$ to $|LS^q|$ **do**
$W^q[LS^q[j]_{mj}][LS^q[j]_{mi}]++$
end of for

Step 4

for $i = 1$ to $|T^p|$ **do**
for $j = 1$ to $|T^q|$ **do**
$$temp = \left(\frac{W^p[i][j] + W^q[j][i]}{2} \right)^{\alpha}$$
if $depth(t_i^p) \neq depth(t_j^q)$ **then**
$temp = temp \times \beta$
end of if
$S = S + temp$
end of for
end of for
$S = \sqrt[\alpha]{S}$
End

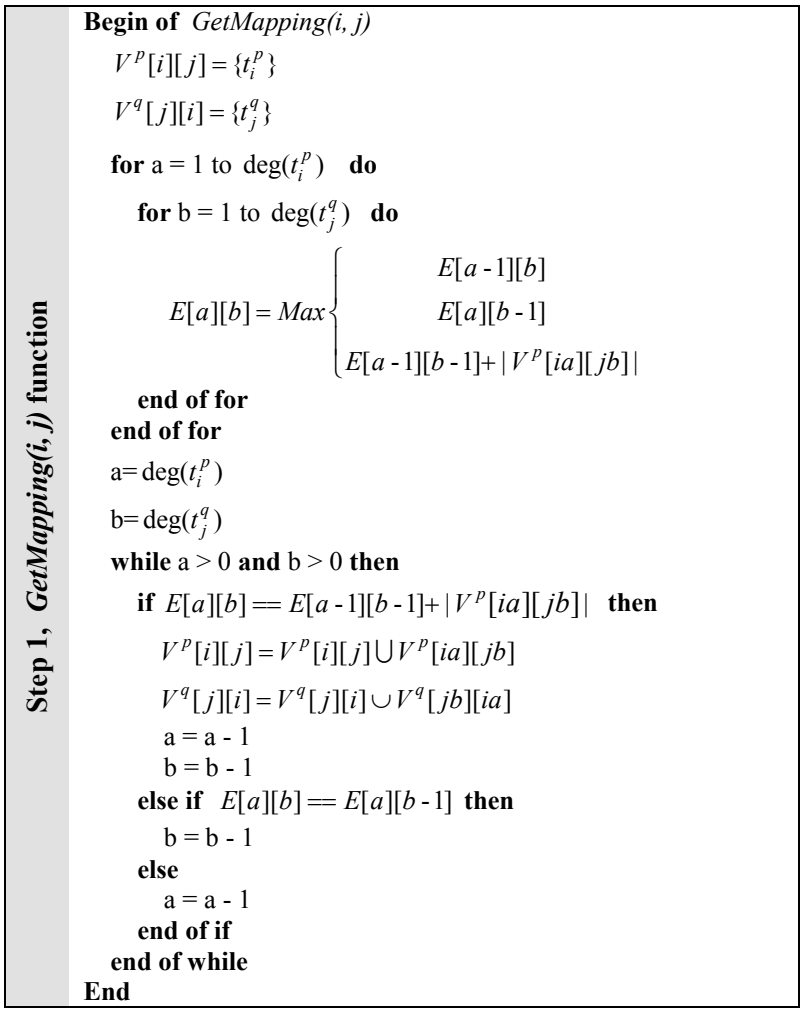


Figure 4.6. Pseudo code for the proposed tree distance algorithm.

Step 2: *Identify each node's largest mapping*: A node in T^p or T^q might belong to several mappings. Considering that we do not want to count one node several times, we determine the largest subtree in the mappings for each node. To compute this step, first, assume two arrays, namely LS^p and LS^q, of size $|T^p|$ and $|T^q|$, respectively. $LS^p[i]$ indicates the largest subtree that t_i^p belongs to. $LS^p[i]$ keeps the indexes of root nodes of the mapping, denoted by $LS^p[i]_{mi}$ and $LS^p[i]_{mj}$. As indicated in Fig 6, filling LS^p and LS^q with appropriate values is the objective of this step. For each mapping, between $T_{i,j}^p$ and $T_{j,i}^q$, we iterate through all the nodes in $V^p[i][j]$ and $V^q[j][i]$ which were computed in the first step. For each node in $V^p[i][j]$, where the index of the node is denoted by $V^p[i][j]_k$ in the pseudo code, we check if the $|V^p[i][j]|$ is larger than the

subtree stored in LS^p for that node, and update LS^p accordingly. A similar procedure is repeated for each node in $V^q[j][i]$.

Step 3: *Compute the weight of each subtree*: In this step, we calculate $W(T^p_{i,j})$ and $W(T^q_{j,i})$ for all the subtrees in the mappings. In the pseudo code, they are denoted by $W^p[i][j]$ and $W^q[j][i]$. We go through LS^p and increase the weight of a subtree when it is reported as a largest subtree of a node in LS^p. This procedure is repeated for LS^q as well.

Step 4: *Calculate* $S(T^p, T^q)$: Now that we have all the subtree weights (W^p and W^q) available, we can simply calculate $S(T^p, T^q)$ according to (4.7).

Figure 4.7 presents an example which indicates the inputs and outputs of each step. Two simple trees with three and four nodes are presented where the mappings are indicated on the figure. There is two valid mapping, one between nodes b and the other between subtrees of a-c. According to Step 1, V^p and V^q are calculated which indicates all the valid and invalid mappings. The largest subtree, for each node, are calculated using Step 2 and are saved in LS^p and LS^q. For example, the second element of LS^p, $(3,4)$, indicates the largest subtree that t^p_2 is a member of. $(3,4)$ represents the mapping between subtrees rooted at node 3 of T^p, and node 4 of T^q. In the next step, the weight of each subtree in the mapping is calculated and stored in W^p and W^q. Finally, in Step 4, similarity is calculated from W^p and W^q.

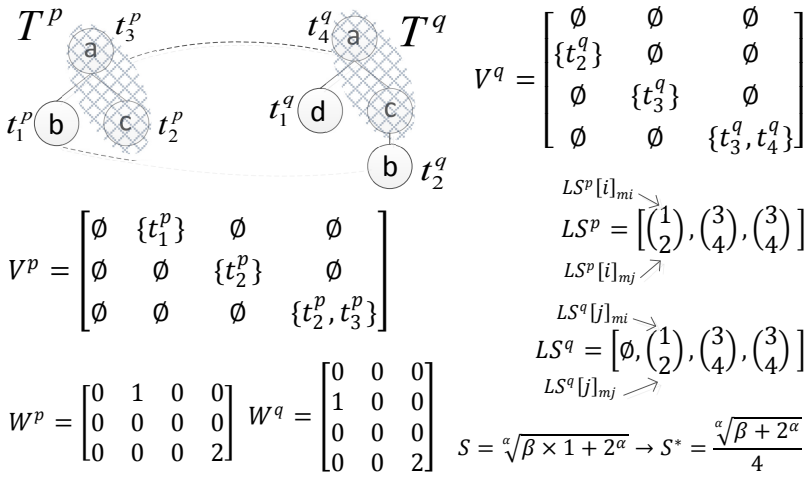


Figure 4.7. A simple example for the proposed EST algorithm.

4.4.4 Runtime Complexity Analysis

In this section, we discuss the order of computational complexity of the EST algorithm. The order of runtime complexity is the summation of the associated complexity into each of the four steps, discussed in the previous section. In accordance with the pseudo code presented in Figure 4.6, we have $EST_{step1} \in O\left(\sum_{i=1}^{|T^p|}\sum_{j=1}^{|T^q|}(GetMapping)\right)$. The *GetMapping* function has a double "for" loop and a "while" loop. Obviously, the double "for" loop is executed $\deg(t_i^p) \times \deg(t_j^q)$ times and the maximum number of executions of the "while" loop is $\deg(t_i^p)+\deg(t_j^q)$. Inside the "while" loop we have two set's union operations which has runtime complexity of $O(Min(|T^p|,|T^q|))$, since the size of each subtree in the mapping cannot be larger than $Min(|T^p|,|T^q|)$. Accordingly, *GetMapping* $\in O(\deg(t_i^p) \times \deg(t_j^q) + (\deg(t_i^p)+\deg(t_j^q)) \times Min(|T^p|,|T^q|))$. Consequently, $EST_{step1} \in O\left(\sum_{i=1}^{|T^p|}\sum_{j=1}^{|T^q|}\left(\deg\left(t_i^p\right)\times\deg\left(t_j^q\right)\right)+Min\left(\left|T^p\right|,\left|T^q\right|\right)\times\sum_{i=1}^{|T^p|}\sum_{j=1}^{|T^q|}\left(\deg\left(t_i^p\right)+\deg\left(t_j^q\right)\right)\right)$. This result can be simplified to $\sum_{i=1}^{|T^p|}\deg\left(t_i^p\right)\times\sum_{j=1}^{|T^q|}\deg\left(t_j^q\right)+Min\left(\left|T^p\right|,\left|T^q\right|\right)\times\left(\sum_{i=1}^{|T^p|}\sum_{j=1}^{|T^q|}\deg\left(t_i^p\right)+\sum_{i=1}^{|T^p|}\sum_{j=1}^{|T^q|}\deg\left(t_j^q\right)\right)$. Since $\sum_{i=1}^{|T^p|}\deg\left(t_i^p\right)=\left|T^p\right|-1<\left|T^p\right|$, we have $EST_{step1} \in O(|T^p|\times|T^q|+2\times Min(|T^p|,|T^q|)\times|T^p|\times|T^q|)$. Since we need to keep the term with highest order, the final runtime of the algorithm's Step 1 is given by $O(Min(|T^p|,|T^q|)\times|T^p|\times|T^q|)$.

To determine LS^p and LS^q in Step 2, we need to iterate $|T^p| \times |T^q|$ times to cover all the possible mappings. According to the description in Step 2, the complexity of each mapping's iteration is $O(2 \times Min(|T^p|, |T^q|))$ since $V(T^p_{i,j})$ and $V(T^q_{j,i})$ are bounded between zero and $Min(|T^p|, |T^q|)$. Therefore, $EST_{step2} \in O(Min(|T^p|, |T^q|) \times |T^p| \times |T^q|)$. Obviously, the runtime order of Step 3 and Step 4 are $O(|T^p| + |T^q|)$ and $O(|T^p| \times |T^q|)$, respectively. Finally, the total runtime of the proposed algorithm is within the order of $O(Min(|T^p|, |T^q|) \times |T^p| \times |T^q|)$ since we need to keep the term with highest order and we can forget about constant coefficients. The calculated runtime complexity is also investigated in the empirical runtime analysis section.

4.5 Evaluation Frameworks Design

In this section, we design clustering and classification frameworks to evaluate the proposed distance function (EST) performance against other tree distance functions. To implement these frameworks, k-medoid [114] is used for the clustering; and KNN [115] and SVM [116] are utilized for the classification framework.

4.5.1 Data Sets

Four different labeled data sets (three real world data sets and one synthetic data set) are utilized in this chapter to investigate the performance of the distance functions to prevent biased results.

The first real data set is CSLOG, available at [138]; it has appeared in a number of publications including [103], [105] and [118]. Each tree in this dataset represents the behavior of a user visiting a website, where each node in the tree indicates a webpage of the website. Each tree is labeled either "edu" (visitor is from edu domain) or "other". Further, this data set contains three weeks of information separated as three data sets, presented in Table 4.1.

Table 4.1. Detailed information regarding the real and synthetic data sets.

Data set	Data set size	Average tree size	Average tree depth
CSLOG_2C_WEEK1	8074	8.03	4.40
CSLOG_2C_WEEK2	7407	8.05	4.46
CSLOG_2C_WEEK3	7628	7.98	4.42
SIGMOD_3C	987	39.04	3.76
TREEBANK_2C	160,616	13.08	4.64
TREEBANK_5C	769,172	9.48	3.85
TREEBANK_6C	922,442	12.80	4.52
SYN_2C_CLUSTER	100	58.22	8.48
SYN_3C_CLUSTER	150	58.50	8.44
SYN_5C_CLUSTER	250	58.31	8.51
SYN_8C_CLUSTER	400	57.85	8.45
SYN_20C_CLUSTER	1000	11.93	5.08
SYN_2C_CLASIFY	200	57.32	7.78
SYN_3C_CLASIFY	300	57.62	7.84
SYN_5C_CLASIFY	500	57.31	7.85
SYN_8C_CLASIFY	800	57.20	7.84
SYN_20C_CLASIFY	2000	12.81	4.83

The second real data set is the ACM SIGMOD records [139] from March 1999. This data set is also utilized in several works such as [106] and [117]. Each tree in this data set is presented as an XML file. We removed one of the XML files, named "a.xml", since its name was not in accordance with other XML files and it was always miss-classified/miss-clustered. Therefore, the data set size is reduced to 987 trees.

The third real data set is called Treebank. The original data set is one huge tree, English sentences from the Wall Street Journal, tagged with parts of speech [140] like "S" (sentence), "NP" (noun phrase), "VP" (verb phrase), "PP" (prepositional phrase), "ADJP" (adjective phrase), and "ADVP" (adverb phrase). To produce a useful labeled data set for clustering and classification, we separated subtrees as samples of the classes. We considered three cases as indicated in Table 4.1: 1) The "NP" and "VP" subtrees as a two class data set; 2) five English phrases ("NP", "VP", "PP", "ADJP", and "ADVP") as a five class data set; and 3) five English phrases plus sentences as a six class data set. The Treebank data sets are the largest data sets among the real data sets in this chapter. Since their sizes are very large, to have a feasible runtime, we performed random sampling where 1000 random trees are selected for two class dataset. In the case of the five and six class datasets, we selected 100 random trees for each class. To prevent any biased results,

the whole process of random sampling and performing the experiments is repeated 100 times and the average of trials are reported. Further, a statistical analysis is performed in Section 4.6.4 which provides evidences on validity of these statistical experiments.

These real data sets have appeared in a number of publications [103], [105], [106], [117], [118] none of which question the veracity of these data sets. Again, we find no real evidence of miss-labeling, in these data sets and hence we believe that they are highly-accurate sources of information.

To generate the synthetic datasets, we considered data sets with different numbers of classes (2, 3, 5, 8, and 20 classes), so we can study the performance of the distance functions with respect to the number of classes in the data set. Accordingly, we generated five synthetic datasets randomly for classification and another five data sets for clustering. Each synthetic data set for clustering has 50 trees for each class. The classification of the synthetic data sets has two sub-components, one for training and one for testing; each contains 50 trees for each class. Generating these data sets is a two-step process:

Step 1: Assuming we are generating a data set with N_c classes, we generate N_c labeled mother trees, namely T^{m_i} where $1 \le i \le N_c$. Each T^{m_i} is generated randomly where to add a new node to the tree, one of the pre-generated nodes is selected randomly and the new node is added to the selected node as a child until $|T^{m_i}| = 50$ is reached. After generating a tree with 50 nodes randomly, each node is randomly labeled from a pool of 30 possible node labels. The size of 50 for mother trees was selected since significantly larger tree sizes was not feasible with respect to runtime of the classification and clustering trials and are believed to be a reasonable representation of many situations found in computer applications. Please note that in the case of the 20 class data set, we reduced the initial size of trees to 10 to have a manageable runtime.

Step 2: After generating the mother trees, a synthetic data set is generated by producing trees using the mother trees as follows: To generate trees in accordance with the *i*th mother tree, T^{m_i}, we go through all the nodes in T^{m_i} and each node, namely t_j, is edited with the probability of ρ. For each t_j, the edit operation is randomly selected with equal probability from one of five edit operations. Let T^p be a subtree of T^{m_i} rooted at t_j and

includes all of the children of t_j; also let T^q be a subtree of T^{m_i} rooted at a random node which includes all of the children of that random node. Further, T^r is selected randomly like T^q, but from T^{m_k} rather than T^{m_i} where T^{m_k} is a randomly chosen mother tree other than T^{m_i}. The five edit operation are: 1) T^p is removed; 2) T^p is replaced with T^q; 3) T^p is replaced with T^r; 4) T^q is added to the root of T^p as a child; and 5) T^r is added to the root of T^p as a child. In fact, a tree in the *i*th class is a combination of *i*th mother tree and other mother trees. For the classification trials, $\rho = 0.5$ was selected; and for the clustering trials, it is selected as 0.25. ρ is chosen lower for the clustering trials, since the clustering results are more sensitive than the classification results; so to keep the results of clustering around 80% accurate, ρ is reduced. Finally, since the synthetic data set generation is a probabilistic process, to prevent any biased results, the whole process of generating data sets and conducting the trials is repeated 100 times and the average of trials are reported as the synthetic datasets results.

4.5.2 Clustering Framework

The k-medoid [114] clustering technique is used as one of the evaluation approaches. The reason why k-medoid is chosen over other clustering techniques, like the classic k-means, is the limitation enforced by our tree data type. Unlike usual machine learning problems, we cannot simply model trees as a set of features and consequently we cannot define operations such as averaging on trees which is required in most of the clustering techniques such as k-means which partitions the data into Voronoi regions [90]. In our case, the only operation defined on trees is the distance between two trees acquired using a distance function. K-medoid is similar to k-means, however, a tree is selected as the center of the cluster, called a medoid, rather than the average of the trees. Partitioning Around Medoids (PAM) algorithm [114] is utilized for clustering. Further, to determine the initial medoids, we selected the first medoid randomly and then trees with highest distance from previously selected medoids. Since the k-medoid approach may find a local optimum rather than the global optimum, the clustering process is repeated 10 times with different initial medoids and the results with minimum cost are selected as the final results. After the clustering is completed, we need to assign a label to each cluster as a predicted label for evaluation purposes. A predicted label is determined as the real label that has the greatest population within that cluster.

4.5.3 Classification Framework

Weighted KNN [115] and kernel based SVM [116] are utilized to perform the evaluation of the distance functions with respect to the classification applications. Similar to clustering discussion, not all the classification techniques work on trees, since we only have a distance between trees, not their features. As a result, only methods like KNN that purely work based on distance functions; and kernel based classification approaches such as SVM, which map the input space into higher dimensions, can be utilized for tree classification problems.

KNN is a classic classification technique where the K nearest neighbors to the test data are identified from the set of training data, and then, in a voting process with S^* as a weight, the predicted class for the test data is determined. K is chosen as nine in all the trial runs since we observed it generates the best results.

Kernel based SVM [116] is a state-of-the-art classification technique which maps the input space into higher dimensions and generates support vectors in the new space. The mapping is performed based on the kernel function. Beside SVM's kernel function which is set to S^* in all the evaluations, we need to specify the penalty factor (C) and epsilon (ε). C controls the over/under fitting in the training stage [116] and is set to 2 for all the real data sets, and 4 for all the synthetic data sets in this chapter. Further, ε which has an effect on the smoothness of the SVM's response and the number of support vectors, is selected as 0.001 in all the experiments. Since SVM originally works only on two class problems, we utilized the one-vs-all technique [141] to extend the SVM classification to multi-classes.

Unlike clustering, we need to train these classifiers, and then perform a classification on another date set. In the case of the CSLOG data sets, we performed three experiments, distinguished as CSLOG_2C_WEEK12, 23, and 31 where the last two digits refer to the training and the test data, respectively. In contrast, in the case of the SIGMOD and Treebank data sets, we utilized 10-fold cross validation approach.

4.5.4 Clustering and Classification Evaluation

After the classification experiments, predicted and real labels for each test tree are available. Similarly, as discussed in the clustering section, after a clustering experiment, each cluster is assigned a label. Hence, each tree has a predicted label beside its real

label. Now that we have predicted and real labels of each tree, the evaluation is performed in terms of 1) accuracy; 2) Weighted Average of F-measures (WAF); and 3) runtime.

Accuracy is simply defined as the number of correctly clustered/classified trees over the total number of trees. The F-measure is a popular information retrieval metric that is defined for each class and integrates recall and precision using the harmonic mean. Let C_i be the set of all the trees in class *i*; and let P_i be the set of all the trees predicted (clustered or classified) to be in class *i,* where $1 \le i \le N_c$. The recall, precision, and f-measure are defined as:

$$\begin{aligned} Recall_i &= \frac{a_i}{|C_i|}, Precision_i = \frac{b_i}{|P_i|}, \\ FMeasure_i &= \frac{2 \times Precision_i \times Recall_i}{Precision_i + Recall_i}, \end{aligned} \tag{4.9}$$

where a_i and b_i denote the number of correctly clustered/classified trees in C_i and P_i, respectively. Since the number of classes is different with respect to different data sets and space is limited, providing a f-measure for all classes separately is not feasible. Therefore, the WAF, as used in WEKA [142], is presented regarding clustering/classification evaluation, where it is defined as:

$$WAF = \sum_{i=1}^{N_c} \frac{|C_i|}{number\ of\ all\ trees} \times FMeasure_i \tag{4.10}$$

where N_c denotes the number of classes in a data set. The experiments within this study were conducted using Java 7 (64bit). The hardware platform, where the experiments have been executed, was an Intel dual-core Processor E6300 (2.8GHz) with 8GB of RAM. The k-medoid and KNN are bespoke implementations; and the libsvm Java SVM library [143] was adapted for the tree classification applications.

4.5.5 Distance Function's parameters

The proposed EST approach includes two parameters, α and β, that need to be adjusted. Obviously, they can be adjusted for every single experiment to achieve the optimum performance. However, for all the experiments, we have fixed the values of the parameters to produce an equivalence with the other distance functions which are

(relatively) parameter free.

As discussed in Section 4.4.2, β reflects the relative position of the mapped subtrees and it can be adjusted between 0 and 1. However, we have no any formal mechanism for estimating β at the moment. Hence, we set β to the neutral value of 0.5 which seems a good balancing point according to the sensitivity analysis presented in Figure 4.8a. Figure 4.8a demonstrates accuracy vs. β, we did not include all the experiments in this figure to prevent a busy figure. For each data set, one clustering and one classification experiment is selected. In addition, we did not include SIGMOD data set as it produces 100% accuracy for all values of β. One can observe from Figure 4.8a that with the increasing β, accuracy is reducing for some of the experiments and increasing for some other. Therefore, the neutral value of 0.5 is selected for all experiments. Finally, WAF sensitivity analysis is not presented as it was similar to accuracy results.

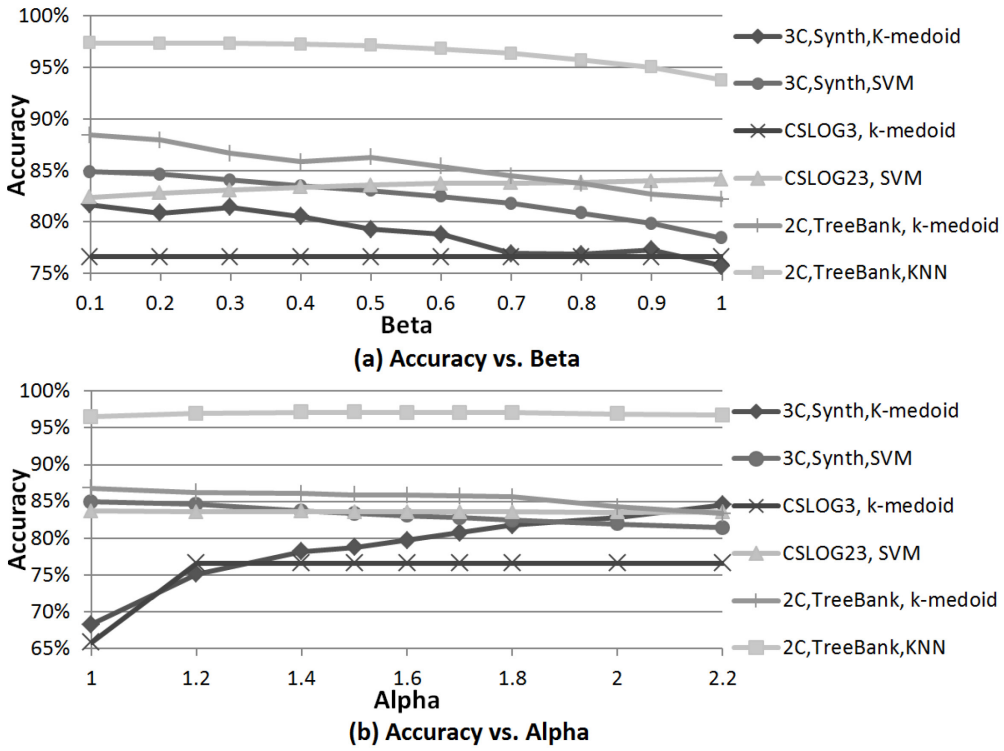


Figure 4.8. The accuracy of EST similarity function against α and β.

α reflects the importance of the size of the mapped subtrees. Our formulation can be thought of as a variation on the well-known Minkowski distance [137]. As explained in [137], the optimal value of α will vary with the domain of application and hence no

universal approach for optimally estimating α exists. In the absence of problem-specific knowledge, it is believed that α can be estimated in our formulation by considering the variation of the average similarity against α ($\alpha \geq 1$). As $\alpha \to 1$, the algorithm overly weights the impact of small trees compared to the impact of large trees. These small trees are minor in terms of the "big picture"; however, their existence or not, can have a large impact on the similarity result and hence we can view the metric as becoming "numerical unstable" as $\alpha \to 1$. Conversely, as $\alpha \to \infty$, the distance metric loses discrimination power. Large trees dominate and substantial variations on small tress have little or no impact on the resultant similarity score. Hence, the selection of α is equivalent to finding the balancing point which minimizes these two undesirable behaviors.

Let α_i represent the ideal balancing point. If we consider a plot of the average similarity against α as shown in Figure 4.9, we can define the plot in terms of: C1 – the curve between $\alpha = 1$ and $\alpha = \alpha_i$; and C2 – the curve between $\alpha = \alpha_i$ and $\alpha = \infty$. From above, C1 can be characterized as a curve where the average similarity changes significant with small changes in α; and C2 as a curve where the similarity changes slowly (in fact, we believe that C2 can be modeled as a linear segment implying no curvature exists across C2). This model is again well-known and is perhaps most commonly used in the Scree test [144].

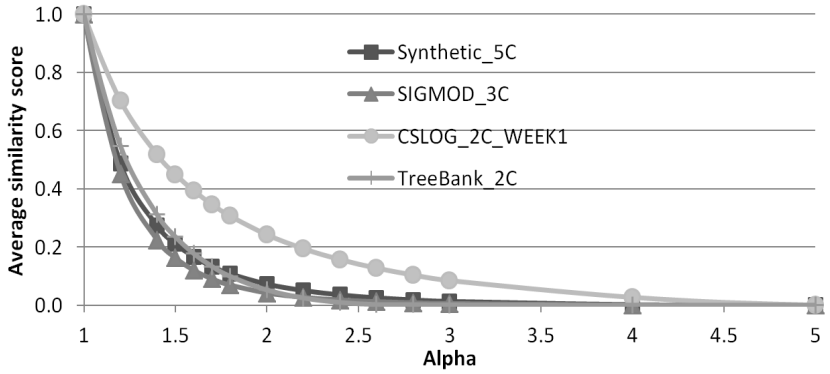


Figure 4.9. The average similarity of EST similarity function against α.

Accordingly, we need to estimate the balancing point (elbow) in Figure 4.9 (Only a subset of the data is shown in the figure to increase clarity). In addition, we have scaled the curves between 0 and 1. Several approaches exist to approximate the elbow point, e.g. the Angle-based technique [145], the Menger Curvature method [146] which is good for continuous data, and the Kneedle technique [147] which works for both discrete and

continues data. None of these approaches are perfect in the presence of noise. Satopaa et al. [147] discuss that the Menger Curvature method is sensitive to noise, while the Kneedle technique produces better results. On average, all the methods give very similar results. The Kneedle technique [147] is believed to be more robust to noise and works for discrete data. Therefore, we utilize the Kneedle technique [147] to estimate the elbow point. The elbow point for each curve was slightly different with the average value of $\alpha = 1.6$. We utilized this value for all the experiments in this chapter and recommend this value in situations where limited data exists stopping the re-estimation of α.

Further, we performed a sensitivity analysis of accuracy against α as presented in Figure 4.8b. Similar to β, we select a few experiments to prevent a busy figure. According to this figure, the calculated value of 1.6 for α seems to be a balancing point for all the experiments.

Finally, the costs of edit operations (γ) are considered as the unit scalar regarding TED and IST for all experiments. The Entropy, Path, and Multiset distance functions have no parameters to discuss.

4.6 Experimental Results and Discussion

4.6.1 K-medoid Clustering Results

Table 4.2 represents the k-medoid clustering results with respect to all data sets discussed in Section 4.5.1. The purpose of this table is to compare the performance of the proposed EST approach against the previous distance functions when they are used as the core in clustering applications. This table has three parts associated to accuracy, WAF, and runtime evaluations. Further, the result of the best evaluated distance function is bolded.

Table 4.2. The clustering results for all case studies in the terms of accuracy, Weighted Average of F-measure (WAF), and runtime.

	Data set	EST	TED	Entropy	Path	Multiset	IST
Accuracy, %	CSLOG_2C_WEEK1	**73.6**	60.0	61.9	**73.6**	63.7	62.6
	CSLOG_2C_WEEK2	**74.8**	60.0	63.0	64.1	62.2	65.9
	CSLOG_2C_WEEK3	**76.6**	63.0	62.5	69.8	63.4	62.1
	SIGMOD_3C	**100**	99.9	54.1	99.9	99.9	**100**
	TREEBANK_2C	**85.9**	76.1	74.7	75.4	77.0	80.6
	TREEBANK_5C	**70.3**	58.2	54.6	61.3	57.1	62.8
	TREEBANK_6C	**64.2**	54.8	51.0	54.4	48.5	54.7
	SYN_2C_CLUSTER	**87.4**	61.8	81.7	78.0	63.9	82.1
	SYN_3C_CLUSTER	79.7	60.6	72.5	68.9	56.8	**81.2**
	SYN_5C_CLUSTER	**78.6**	62.7	69.1	62.9	47.3	75.3
	SYN_8C_CLUSTER	**79.1**	64.1	69.6	60.5	42.0	75.2
	SYN_20C_CLUSTE	**77.3**	62.8	72.4	58.0	37.0	71.5
WAF, %	CSLOG_2C_WEEK1	**69.4**	62.5	59.1	**69.4**	60.1	59.4
	CSLOG_2C_WEEK2	**71.6**	63.0	60.9	66.1	60.4	62.3
	CSLOG_2C_WEEK3	**72.2**	65.3	59.9	63.7	60.4	59.7
	SIGMOD_3C	**100**	99.9	68.6	99.9	99.9	**100**
	TREEBANK_2C	**85.8**	76.1	74.0	74.2	76.2	80.3
	TREEBANK_5C	**70.9**	58.7	55.3	61.8	57.5	63.3
	TREEBANK_6C	**63.7**	55.5	50.7	53.6	47.7	54.0
	SYN_2C_CLUSTER	**86.8**	56.7	81.0	76.4	61.1	81.7
	SYN_3C_CLUSTER	76.4	55.5	68.4	63.6	51.0	**80.6**
	SYN_5C_CLUSTER	**75.1**	59.5	64.2	56.3	40.1	73.5
	SYN_8C_CLUSTER	**76.4**	61.7	65.1	54.0	34.3	73.9
	SYN_20C_CLUSTE	**76.1**	62.2	71.1	56.8	35.7	71.1
Runtime, minutes	CSLOG_2C_WEEK1	**4.8**	13.5	110.7	102.8	23.1	66.0
	CSLOG_2C_WEEK2	**6.5**	17.7	146.4	138.5	29.8	82.4
	CSLOG_2C_WEEK3	**6.3**	15.1	86.4	76.2	26.8	70.2
	SIGMOD_3C	2.7	11.6	2.1	2.0	**1.7**	156.2
	TREEBANK_2C	**32.1**	144.7	85.7	70.5	35.5	244.8
	TREEBANK_5C	**5.9**	17.9	15.5	12.2	6.7	32.9
	TREEBANK_6C	**15.9**	55.6	39.8	32.5	19.4	98.4
	SYN_2C_CLUSTER	**4.2**	26.3	7.3	6.0	4.8	67.2
	SYN_3C_CLUSTER	**9.5**	58.0	15.4	12.6	11.4	141.7
	SYN_5C_CLUSTER	**23.9**	156.0	42.6	35.1	33.6	511.8
	SYN_8C_CLUSTER	**59.5**	400.8	109.1	89.7	89.8	1361.3
	SYN_20C_CLUSTE	**66.6**	87.3	80.6	79.1	70.7	165.5

As indicated in Table 4.2, the proposed EST has outperformed other distance functions in most of the investigated situations in terms of accuracy, WAF, and runtime. In the case of the three CSLOG data sets, EST has significantly improved results, over 10% in accuracy, except for the Path distance with respect to CSLOG_2C_WEEK1. With respect to SIGMOD_3C data set, EST and IST produce the perfect result; TED, Path, and

Multiset have only one miss-clustered tree; finally, the Entropy approach produces a very poor performance. EST also significantly outperformed other approaches (over 5%) with respect to the Treebank data sets. Regarding the synthetic data sets, the Multiset approach has the worst result and its accuracy and WAF significantly reduces as the number of classes grows. EST produced the best results in most cases with IST in second place. Apart from TED, the performances of all the distance functions are degraded as the number of classes increases. However, this result is not an advantage for TED since its uniformly poor performance. Finally, the runtime results indicate that EST has the best efficiency in term of runtime, except for the SIGMOD data set. This result makes the proposed approach the best for real time applications. In contrast, IST and Entropy produce the largest execution times.

4.6.2 KNN Classification Results

The KNN classification results with respect to all data sets are represented in Table 4.3 where the performances of the distance functions are investigated. The CSLOG data sets' results suggest that all the distance functions have a similar performance with regard to accuracy and WAF measures. Apart from the Multiset approach, all other distance functions produced a perfect classification regarding the SIGMOD data set. The Multiset approach produced one miss-classification. EST and then TED produced the best results with respect to the Treebank data sets. In case of the synthetic data sets, the EST approach produces the most accurate results; and the largest WAF measures. IST has the second most impressive results in terms of accuracy and WAF. Similar to the clustering runtime results, EST has the lowest runtime complexity for all data sets except SIGMOD where the Multiset approach produces the lowest runtime. Again, similar to the clustering runtime results, Entropy and IST have the largest runtime performances.

Table 4.3. The KNN classification results for all case studies in the terms of accuracy, Weighted Average of F-measure (WAF), and runtime.

	Data set	EST	TED	Entropy	Path	Multiset	IST
Accuracy, %	CSLOG_2C_WEEK12	**83.4**	83.1	83.3	83.2	83.0	83.2
	CSLOG_2C_WEEK23	**84.1**	83.1	83.9	83.9	83.7	83.8
	CSLOG_2C_WEEK31	**83.2**	82.4	83.0	83.0	83.0	83.0
	SIGMOD_3C	**100**	**100**	**100**	**100**	99.9	**100**
	TREEBANK_2C	**97.0**	96.9	93.9	94.9	90.1	95.5
	TREEBANK_5C	**88.8**	86.3	79.2	81.9	69.6	83.0
	TREEBANK_6C	**87.2**	84.8	75.5	78.2	65.7	80.4
	SYN_2C_CLASIFY	**82.9**	78.1	77.6	76.9	72.4	80.5
	SYN_3C_CLASIFY	**75.1**	67.4	70.7	70.0	60.6	71.4
	SYN_5C_CLASIFY	**70.3**	61.6	67.8	66.4	55.6	66.4
	SYN_8C_CLASIFY	**68.7**	59.1	65.8	65.1	53.0	63.9
	SYN_20C_CLASIFY	**68.9**	60.5	62.0	59.3	37.9	63.1
WAF, %	CSLOG_2C_WEEK12	**82.4**	81.8	82.3	82.2	81.9	82.2
	CSLOG_2C_WEEK23	**83.2**	81.7	82.9	83.0	82.6	82.8
	CSLOG_2C_WEEK31	**82.0**	80.9	81.9	81.9	81.8	81.9
	SIGMOD_3C	**100**	**100**	**100**	**100**	99.9	**100**
	TREEBANK_2C	**97.0**	96.9	93.9	94.9	90.1	95.5
	TREEBANK_5C	**89.0**	86.5	79.6	82.3	70.0	83.2
	TREEBANK_6C	**87.3**	84.9	75.4	78.4	65.5	80.3
	SYN_2C_CLASIFY	**82.8**	77.9	77.3	76.7	72.2	80.4
	SYN_3C_CLASIFY	**75.0**	67.4	70.5	69.9	60.4	71.4
	SYN_5C_CLASIFY	**70.3**	61.7	67.7	66.3	55.5	66.4
	SYN_8C_CLASIFY	**68.8**	59.3	65.9	65.2	52.9	64.0
	SYN_20C_CLASIFY	**69.4**	60.9	62.3	59.6	37.8	63.6
Runtime, minutes	CSLOG_2C_WEEK12	**5.8**	15.3	126.7	117.5	26.3	75.5
	CSLOG_2C_WEEK23	**5.6**	14.3	119.9	113.5	24.3	67.4
	CSLOG_2C_WEEK31	**6.7**	15.8	91.5	80.7	28.2	74.3
	SIGMOD_3C	2.6	11.4	2.0	1.8	**1.6**	156.0
	TREEBANK_2C	**29.8**	142.3	83.2	68.0	33.2	242.5
	TREEBANK_5C	**3.5**	15.4	13.1	10.0	4.5	30.6
	TREEBANK_6C	**8.4**	47.2	31.5	24.7	11.8	90.4
	SYN_2C_CLASIFY	**4.0**	28.2	8.0	6.7	5.3	73.7
	SYN_3C_CLASIFY	**9.3**	64.0	18.3	15.1	13.2	155.9
	SYN_5C_CLASIFY	**22.2**	173.0	47.2	38.8	34.8	420.9
	SYN_8C_CLASIFY	**58.9**	429.8	116.5	94.9	90.7	1055.4
	SYN_20C_CLASIFY	**26.8**	55.3	48.2	36.4	28.9	185.7

4.6.3 SVM Classification Results

The SVM classification results are presented in Table 4.4; they are similar to the KNN results. The differences include the now perfect result for the Multiset distance function in case of the SIGMOD data set. In case of the Treebank data sets, all the approaches produce very good results. In addition, IST produced the highest accuracy with respect to

SYN_2C_CLASIFY data set. However, the EST has still the greatest accuracy with regard to the other synthetic data sets. The runtime results with respect to lowest and largest runtime are similar to the KNN classification and the clustering case studies. These suggest the proposed EST has, in general, the lowest runtime complexity. Although the comparison between KNN and SVM is not within the scope of this research, one can observe that SVM possesses a better accuracy and WAF on average.

Table 4.4. The SVM classification results for all case studies in the terms of accuracy, Weighted Average of F-measure (WAF), and runtime.

	Data set	EST	TED	Entropy	Path	Multiset	IST
Accuracy, %	CSLOG_2C_WEEK12	**83.5**	70.8	83.2	82.9	83.0	82.4
	CSLOG_2C_WEEK23	**83.6**	68.1	83.5	83.1	83.0	82.9
	CSLOG_2C_WEEK31	**82.8**	68.3	82.5	82.3	82.2	81.4
	SIGMOD_3C	**100**	**100**	**100**	**100**	**100**	**100**
	TREEBANK_2C	**99.9**	**99.9**	99.8	99.8	98.6	**99.9**
	TREEBANK_5C	**99.7**	**99.7**	99.3	99.2	97.4	99.5
	TREEBANK_6C	**99.7**	**99.7**	99.0	99.0	96.8	99.2
	SYN_2C_CLASIFY	88.0	87.0	86.2	84.8	84.9	**88.3**
	SYN_3C_CLASIFY	**83.1**	82.2	79.7	79.0	76.9	82.6
	SYN_5C_CLASIFY	**79.8**	78.9	77.1	75.6	72.3	79.7
	SYN_8C_CLASIFY	**78.6**	77.4	75.6	74.2	69.4	77.8
	SYN_20C_CLASIFY	**71.6**	64.0	66.5	65.3	50.3	59.0
WAF, %	CSLOG_2C_WEEK12	**82.3**	70.5	82.0	81.4	81.6	81.2
	CSLOG_2C_WEEK23	**82.3**	68.0	**82.3**	81.7	81.5	81.7
	CSLOG_2C_WEEK31	**81.4**	67.8	81.1	80.6	80.6	80.0
	SIGMOD_3C	**100**	**100**	**100**	**100**	**100**	**100**
	TREEBANK_2C	**99.9**	**99.9**	99.8	99.8	98.6	**99.9**
	TREEBANK_5C	**99.7**	**99.7**	99.3	99.2	97.4	99.5
	TREEBANK_6C	**99.7**	**99.7**	99.0	99.0	96.8	99.2
	SYN_2C_CLASIFY	87.9	87.0	86.2	84.8	84.9	**88.3**
	SYN_3C_CLASIFY	**83.0**	82.2	79.6	79.0	76.9	82.6
	SYN_5C_CLASIFY	**79.8**	78.9	77.1	75.6	72.3	79.7
	SYN_8C_CLASIFY	**78.6**	77.5	75.6	74.2	69.4	77.8
	SYN_20C_CLASIFY	**72.0**	64.3	66.8	65.7	50.3	59.3
Runtime, minutes	CSLOG_2C_WEEK12	**11.1**	28.7	237.4	220.2	49.5	141.6
	CSLOG_2C_WEEK23	**12.6**	31.9	266.2	252.1	54.1	149.6
	CSLOG_2C_WEEK31	**13.1**	31.1	177.8	157.0	54.9	144.4
	SIGMOD_3C	2.8	11.4	2.1	2.0	**1.6**	156.4
	TREEBANK_2C	**30.1**	142.9	83.5	68.4	33.4	243.1
	TREEBANK_5C	**4.2**	16.5	13.9	10.8	5.2	31.5
	TREEBANK_6C	**9.7**	49.0	32.8	26.2	12.8	91.9
	SYN_2C_CLASIFY	**8.9**	58.8	16.4	13.6	10.9	148.2
	SYN_3C_CLASIFY	**20.1**	127.4	35.2	29.2	26.0	325.7
	SYN_5C_CLASIFY	**53.2**	351.8	94.9	79.2	65.3	895.7
	SYN_8C_CLASIFY	**123.1**	875.2	233.8	194.8	174.7	2249.1
	SYN_20C_CLASIFY	**54.5**	112.2	97.3	73.4	58.5	376.7

4.6.4 Statistical Analysis of Results

As explained in Section 4.5.1, the results of the Treebank and Synthetic data sets are averaged over 100 trial runs. Therefore, we have a population of 100 results for each experiment which allows us to perform a test of statistical significance (z-test, one-tailed, our working hypothesis is that the EST will produce superior results) with a conservative type I error of 0.01. Further, we have calculated effect size (Cohen's method [90]) which estimates the "size" discrepancy between two statistical populations. Cohen defines the standard value of an effect size as small (0.2), medium (0.5), and large (0.8).

Accordingly, Table 4.5 represents the effect size for accuracy of EST against all the previous approaches. In this table, a positive value of effect size indicates that EST outperformed that method. The "*" beside an effect size indicates the result of the z-test where a significant difference exist at the 0.01 level. The results indicate that in most of the experiments EST statistically significant outperforms other approaches.

Table 4.5. The effect size between accuracy of the EST and previous approaches. "*" indicates the result of the z-test where a significant difference exist at the 0.01 level

	Data set	TED	Entropy	Path	Multiset	IST
K-medoid	TREEBANK_2C	5.81*	2.10*	1.17*	1.60*	1.65*
	TREEBANK_5C	2.59*	3.83*	2.68*	3.91*	2.22*
	TREEBANK_6C	2.46*	3.38*	2.72*	4.81*	2.62*
	SYN_2C_CLUSTER	2.40*	0.79*	0.98*	3.69*	0.78*
	SYN_3C_CLUSTER	1.69*	0.38*	0.69*	2.69*	-0.11
	SYN_5C_CLUSTER	1.95*	0.77*	1.47*	5.17*	0.28*
	SYN_8C_CLUSTER	2.02*	0.99*	1.97*	6.95*	0.42*
	SYN_20C_CLUSTER	3.33*	1.10*	4.25*	9.86*	1.29*
KNN	TREEBANK_2C	0.17	4.84*	3.53*	8.79*	2.46*
	TREEBANK_5C	1.70*	5.79*	4.17*	9.76*	3.74*
	TREEBANK_6C	1.66*	6.96*	5.83*	12.30*	4.38*
	SYN_2C_CLASIFY	1.08*	1.03*	1.38*	2.27*	0.50*
	SYN_3C_CLASIFY	1.84*	0.96*	1.26*	3.40*	0.87*
	SYN_5C_CLASIFY	2.20*	0.67*	1.06*	3.91*	1.03*
	SYN_8C_CLASIFY	3.57*	1.01*	1.31*	5.39*	1.73*
	SYN_20C_CLASIFY	4.25*	3.54*	5.02*	14.90*	2.91*
SVM	TREEBANK_2C	0.00	0.82*	0.78*	1.07*	0.00
	TREEBANK_5C	0.00	1.30*	1.46*	4.24*	0.76*
	TREEBANK_6C	0.00	2.09*	2.24*	5.97*	1.55*
	SYN_2C_CLASIFY	0.24*	0.49*	0.83*	0.91*	-0.09
	SYN_3C_CLASIFY	0.28*	1.04*	1.25*	2.03*	0.14
	SYN_5C_CLASIFY	0.29*	0.89*	1.36*	2.72*	0.02
	SYN_8C_CLASIFY	0.58*	1.47*	2.06*	5.00*	0.40*
	SYN_20C_CLASIFY	3.54*	2.66*	3.23*	9.91*	5.70*

4.6.5 Empirical Runtime Analysis

In addition to accuracy and WAF, the computational cost of an algorithm is an important factor in practical applications. The runtime of the clustering and classification experiments are reported in Tables 4.2, 4.3, and 4.4. To further empirically compare the distance functions' runtime, we measure the distance calculation runtime with respect to different tree sizes. Tree sizes between 5 and 100 with step size of 5 have been investigated where both trees are generated randomly as described in the synthetic tree generation section. In addition, the hardware platform is in accordance with the platform described at the end of Section 4.5.4; and again, Java 7 (64 bit) is utilized to implement the source code. The runtime measurement is performed 1000 times and the average distance function execution times are presented in Figure 4.10 in milliseconds.

The IST distance function produced the largest runtime followed by TED and then the Multiset approach. The EST, Path, and Entropy have the best runtime; all three approaches produce broadly similar results.

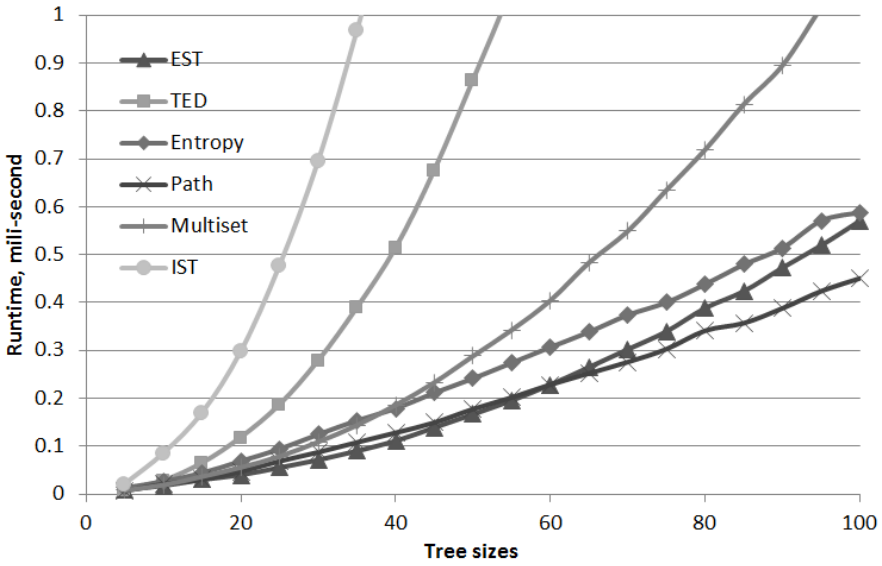


Figure 4.10. Average execution time for different distance functions with tree sizes between 5 and 100.

4.7 Summary

In this chapter, the novel EST similarity function has been proposed for the domain of tree structured data comparison with the aim of increasing the effectiveness of

applications utilizing tree distance or similarity functions. This new approach seeks to resolve the problems and limitations of previous approaches, as discussed in Section 4.4.1. In addition, the new approach must enhance applications where a tree distance function is utilized. To achieve this goal, we first extensively analyze other distance functions. Then, we identified situations where the studied distance functions have poor performance; and finally we propose the EST approach. The proposed EST approach preserves the structure of the trees by mapping subtrees rather than nodes. EST generalizes the edit base distances and mappings by breaking the one-to-one and order preserving mapping rules. Further, it introduces new rules for subtree mapping provided in Section 4.4.2.

An extensive experimental study has been performed to evaluate the performance of the proposed similarity function against previous research. Clustering and classification frameworks are designed to perform an unbiased evaluation according to K-medoid, KNN, and SVM along with four distinct data sets. The real-world data sets have appeared in a number of publications [103], [105], [106], [117], [118] and hence they are deemed to be reliable source of information. Further, using synthetic data sets, we investigated the effect of varying the number of classes in the evaluation. This extensive evaluation framework is one of the advantages of this research over previous researches such as [103], [105], [106], [117], and [118].

The results of the experimental studies demonstrate that the EST approach is superior to the other approaches with respect to classification and clustering applications. To evaluate the performance, accuracy and WAF, are used in Tables 4.2, 4.3, and 4.4, where, in general, EST is demonstrated to be a better option for the clustering and classification of tree structured data. However, the performance of a distance function varies with the domain of application; and hence, we cannot generalize the superior performance of EST to all domains of application.

The computational cost of a tree distance function should be carefully considered for practical applications. Given T^p and T^q as the input trees to the distance function, we calculated the runtime order of the EST as $O(|T^p| \times |T^q| \times Min(|T^p|, |T^q|))$. Further, the runtime of all the clustering and classification experiments are measured where the proposed EST outperformed all other distance functions with respect to all data sets except SIGMOD. In addition, an empirical analysis has been performed to compare the

runtime of EST vs. other distance functions in different tree sizes. The result of this empirical investigation suggests that the runtime efficiency of EST, Entropy, and Path are better than the other distance functions. Accordingly, the conclusion can be drawn that the proposed EST is an appropriate approach for computationally restricted and real time applications.

Finally, although further studies are required to validate the use of the EST similarity function in real-life applications, EST has been demonstrated to have a superior performance against TED, IST, Path, Entropy, and Multiset distance functions with respect to classification and clustering applications.

5 Tree Test Data Generation through an Evolutionary Optimization

5.1 The Focus of This Chapter

In this chapter, the objective is to generate a diverse set of test cases where each test case is a tree. As explained before, in the context of black-box software testing, it is believed that a diverse set of test cases is more likely to produce more effective test cases [13]–[17]. To achieve this in the tree domain, similar to chapter 3 for strings, we have a fitness function that measures the diversity of a test set. This allows an optimization technique to be employed to generate test cases based upon the fitness function. This means more diverse test cases which leads to a better failure detection.

We also demonstrate that the distribution of the size of the generated trees affect the failure detection. Since the first fitness function is unable to control the size distribution of the trees, we create a second fitness function which indicates the proximity of the distribution of the sizes of the trees in a test set to an expected distribution. A multi-objective optimization technique is used to enforce both fitness functions simultaneously.

To empirically investigate the diversity based test generation for trees, we generate mutants of four real world programs that accept trees as input. Test sets with different characteristics are generated and tested on these programs. The experimental results demonstrate that generating test cases based on the diversity objective improves the failure detection rate.

The highlights of this chapter can be summarized as:

1. Investigating the effect of generating diverse tree test cases on failure detection performance. We indicate that through a diversity based objective function and an optimization algorithm, more efficient test cases can be produced.

2. Applying different tree distance functions to tree generation methods and demonstrating that the proposed tree distance function in the previous chapter (Extended Subtree (EST)) has superior performance in diversity based test case generation.

3. Investigating the effect of tree node values on failure detection. We produce the

strings required for node values according to the MOGA string generation method in chapter 3 and compare the results with random string values.

4. Empirical investigation of the tree test case generation methods through a mutation approach on four real world programs.

5.2 Test Case Abstract Model

The test case generation methods investigated in this chapter can be applied to any system where the input to the system can be modeled by a tree. This tree is called abstract tree – a labeled tree with a finite number of labels. The number of required labels is determined based upon the software under the test. In other words, for any software under the test, the user need to define the abstract tree model for that software and then, run the test case generation methods to produce the test cases. The generated test cases are abstract test cases that must be decoded into concrete test cases according to the software under the test. The decoding process normally includes replacing the node labels with final values. In this chapter, we use XML test cases where the tree model and the decoding process are discussed later in Section 5.5.

5.3 Tree Test Case Generation Methods

5.3.1 Random Tree Generation

Random tree generation is the base line method and every other tree generation method is compared against it. Further, every other tree generation method in this chapter requires a random tree generation in its process. For example, the initial population in GA is produced randomly. Therefore, we need to define how the random trees are produced.

To produce a random tree based on the abstract tree model, first, a random size is selected as the tree size, where $1 \leq$ random size $\leq$ *MaxTreeSize*. *MaxTreeSize* is a constant. Then, a random label is selected as a root node. Following that, random labels are selected and added to random positions in the tree until the target tree size is reached.

5.3.2 Adaptive Random Tree Generation

To improve the poor effectiveness of RT, ART methods are introduced. ART methods are discussed in depth in chapter 2 and 3 and hence, are not repeated here. Similar to chapter 3 on strings test generation, FSCS [18] and ARTOO [62] methods are used to generate test cases. The only difference is that, in this chapter, the string distance

functions used in FSCS and ARTOO are replaced with tree distance functions.

5.3.3 Evolutionary Tree Generation

To generate trees test cases, evolutionary algorithms can be used. Among the evolutionary algorithms, Genetic Algorithms (GA) [70] are the most commonly used search algorithms in software engineering due to their effectiveness [66]. Similar to chapter 3 on string generation, we use GA and Multi-Objective GA (MOGA) [74] to produce tree test cases. The GA and MOGA used in this chapter have mostly same parameters to the ones used in chapter 3 for string generation. Hence, in the following, we only discuss the differences.

The GA requires a fitness function to guide the optimization where we use a diversity-based fitness function similar to string generation. The only difference is that tree distance functions are used rather than string distance functions. Further, similar to string generation, the Benford distribution [64], [79] and Kolmogorov–Smirnov test [84] are used as the second fitness function for the MOGA. Crossover and mutation are more complicated for trees compared to strings. To generate two offspring from the parent trees, one node is selected randomly in each parent tree. Then, the nodes along with all their children are swapped. Edit, delete, and add are used as mutation operators where every node in each tree is mutated with 1% probability. Each time, one of the mutation operators is selected randomly. For edit, the label of the abstract tree is replaced with another label which is selected randomly from the possible labels in the tree model. For delete, the node is deleted and for add operator, a node with random label is produced and is attached to the mutated node as a child.

NSGA-II produces a Pareto-optimal set of test sets rather than a single optimal test set. The Pareto-optimal set is the first front of the last generation of the algorithm. Among the Pareto-optimal test sets, the results indicate that the test set with best diversity fitness on the Pareto-optimal front generates the best failure detection effectiveness. Consequently, for the results that are presented for MOGA in this chapter, the test set with best diversity fitness on the Pareto-optimal front is selected. This implies that the best solution is the solution with best diversity which also achieved the target string length distribution.

5.4 Tree Distance Functions

As discussed earlier in this chapter, one of the objectives of this chapter is to investigate

different tree distance functions in the context of software test generation. In the previous chapter, we proposed a new tree distance function (EST) where its superior effectiveness in clustering and classification applications is demonstrated. Beside EST, five other tree distance functions were examined and the results indicate that EST outperforms other tree distance functions.

In this chapter, EST distance function is compared against the same five distance functions (IST, TED, Entropy, Path, and Multiset) in the context of software test generation. Accordingly, each distance function is used in FSCS, ARTOO, GA, and MOGA tree test generation methods and the results are compared.

5.5 Experimental Framework

This section discusses the conducted experiments to analyze the effectiveness of FSCS, ARTOO, GA, and MOGA against RT. Real world programs are used to perform an empirical evaluation. These programs accept XML as input which can be modeled as a tree. Hence, the abstract tree model for XML input and the decoding process are explained in this section. Then, mutated [49], [91] versions of these programs are generated. The P-measure [90] is employed to quantitatively measure the performance of the test case generation methods. Finally, features of generated tree test sets are discussed.

5.5.1 Software Under Test (SUT)

Four real world Java programs are selected as case studies to conduct an empirical evaluation on the fault-detection effectiveness of the tree test case generation methods. These programs are open source programs that are widely used in a variety of applications that interact with XML data. These programs accept a XML file as an input, and hence they are suitable, since XML can be easily modeled as a tree. Programs were selected basically upon the following criteria:

1. The input of each program can be modeled by a tree, so that we can use test generation methods based on an abstract tree model. Hence, every program contains functionality which transforms or manipulates the input tree which makes them a true test where input test cases are trees.

2. Each program is an open source program and hence, they are publicly available for research proposes. This allows replication of these results.

3. The size of programs covers a wide range; the sizes of selected programs vary between 6,000 and 34,000 LOC.

Table 5.1 provides a description of each program. "NanoXML" [148], [149] is an XML parser implemented in Java. It accepts an XML file and parses the input into a DOM (Document Object Model) tree. The parsed DOM tree can be used by other programs or applications as a representation of the XML document. "NanoXML" can also convert back a DOM tree into an XML file. "NanoXML" is a non-validating parser which only checks for structure of the XML code. It works when there is no DTD (Document Type Definition) or schema. "NanoXML" is even ported into embedded systems since its dependency to other Java libraries is low and it is small compared to other XML parser [150]. "JsonJava" is a library that implements JSON (JavaScript Object Notation) decoder/encoder in Java. JSON is a syntax for storing and exchanging data similar to the XML. "JsonJava" can convert the input XML data into the Json format as output which makes it good program for our experiments. "StAX" which stands for "Streaming API for XML" is a standard XML processing library that allows the programmer to stream XML data from and to the application [151]. "StAX" is a pull parser that requires a small memory footprint. A pull parser iteratively visits the various elements, attributes, and data in an XML document. In each iteration, the data can be consumed by another program or code. Unlike "NanoXML", it does not produce DOM trees and hence, it requires a small memory footprint. "JTidy" is an XML and HTML syntax cleaner and pretty printer. It can parse XML, check the syntax, fix syntax errors, and finally print the parsed DOM tree in a human readable form.

Table 5.1. Programs used to perform experimental evaluations.

#	Name	Version	Source code URL	Number of Classes	LOC	Generated mutants	Selected mutants
1	NanoXML	2.2.1	nanoxml.sourceforge.net/orig	23	7698	5448	3865
2	JsonJava	--	github.com/douglascrockford/JSON-java	17	6132	4507	3714
3	StAX	1.2.0	stax.codehaus.org	92	17770	8521	8256
4	JTidy	r938	jtidy.sourceforge.net	52	33070	27965	23839

5.5.2 XML Test Case Abstract Model

To generate XML test cases using tree generation methods, an abstract tree model for XML needs to be specified. As described in Section 5.2, the abstract model must have a limited number of labels. Six labels are selected that conform to different types of nodes

in an XML document. The selected labels are “Element”, “Attribute”, “Text”, “Comment”, “Processing-Instruction”, and “CDATA”. Beside the selected node types, other node types like “Document” and “DocumentType” exist. These nodes are not part of the XML tree and they are normally used once at the beginning of each XML document to specify some information. For instance, “DocumentType” that starts with “<!doctype…” is an optional node at the beginning of the document, before the root node, specifying the data model for the XML document. The “Document” node represents the entire document. There is no tag in the XML document for it. It is just a representation of the document when the XML document is parsed into a DOM tree. Accordingly, these node types are excluded from the abstract tree model since they are not part of the XML tree and hence, they cannot be modeled as a node in the abstract tree model. Among the selected labels, only “Element” can have child nodes and the rest of labels can be leave nodes. This limitation is enforced while generating the random abstract trees. Therefore, in the abstract tree model for XML, two types of labels exist. Labels that can have child nodes and labels that cannot have child nodes. During the random abstract tree generation, first, one of the label types is randomly chosen and then, a label is randomly selected.

5.5.3 Abstract Tree Decoding to XML

After the abstract tree test cases are produced, they need to be decoded into concrete test cases where a concrete test case is an XML document. To achieve this, every node in the abstract tree is converted into an equivalent XML node according to its label. Then, a value for each node is generated as a random string. We used random strings with maximum size of 30 similar to the string generation in chapter 3.

In addition, to investigate the effect of tree node values on failure detection, we also produce the required strings for node values according to the MOGA string generation method in chapter 3 and compare the results with random string values in Section 5.6.4. To generate the node values according to MOGA, one string set is generated for each label in a tree test set. In other words, in each tree test set, we first identify the number of each label. Accordingly, a string set is generated for each label and then, values are assigned to the nodes.

5.5.4 Source Code Mutation

To measure the effectiveness of the test case generation methods, faulty versions of the

software under test are required. Mutation techniques [49], [91] are a well-known approach to automatically manipulate the source code and produce a large number of faults [49]. There is considerable empirical evidence indicating a correlation between real faults and mutants [55], [91].

Similar to chapter 2 and 3, muJava [54] is employed to produce mutated versions of the programs under the test where a total of 46,441 mutants are generated for the four case study programs. Then, those mutants that were failed with the majority of test sets (more than 90% of all the test sets) were deleted. These defects were considered as unrealistic and hence contrary to the "Competent Programmer" hypothesis which is an essential idea in mutation testing [93]. Table 5.1 demonstrates the number of generated and selected mutants per program.

5.5.5 Testing Effectiveness Measure

Similar to chapters 2 and 3, we use p-measure to evaluate the effectiveness of test case generation methods. An in depth discussion on the p-measure definition and the reason behind its selection as a quantitative effectiveness measure is presented in Section 2.6.1.

5.5.6 Tree Test Set Characterization

A test set with a fixed size is required to evaluate the p-measure. In this chapter, we perform experiments with four test set sizes, 4, 6, 8 and 10. As the test set size increases, the difference in the results of different test generation methods is normally reduced and hence, repeating the experiments with larger test set sizes is not required. Beside, as the size of the test sets increases, the runtime increases in a quadratic order according to Section 3.5.

Applying a test set to a mutated version of a program will return zero or one according to the p-measure calculation rules. Accordingly, to estimate the p-measure as a number between zero and one, we applied 10 test sets. Further, we repeated this process 100 times for each mutated version to be able to estimate mean and standard deviation parameters for the measurements. As a result, each test case generation method (RT, FSCS, ARTOO, GA, and MOGA) produced 1,000 test sets for each test set size. Further, everything is repeated with six tree distance functions as discussed before. This leads to $1{,}000\times(4+6+8+10)\times5\times6=840{,}000$ test cases being applied to each mutant.

In each test case generation method, we need to specify the maximum tree size

(*MaxTreeSize*) as a constant number. The size of generated trees are between one and *MaxTreeSize*, inclusive. We repeated all the experiments with two sets of different settings with respect to the tree sizes. In the first set of experiments, *MaxTreeSize* is set to 30. So, all the test generation methods use a same *MaxTreeSize*. Figure 5.1 indicates the p-measure for each program when the sizes of trees are variable in a random tree generation. As a result, the failure detection results improve as the size of trees increases; clearly defining tree size as a co-variant of effectiveness. Further, the mean size of generated trees is different when different test generation methods and different tree distance functions are used with the same *MaxTreeSize*. Accordingly, in the first set of experiments, GA outperforms MOGA (refer to Section 5.6.1 for the results) as GA produces larger trees compared to MOGA, on average. Hence to attempt to compare the tree generation methods independently of tree size, the second set of experiments were produced, now we set the mean size of trees as a fix number. We selected 15.5 which is the mean of [1, 30]. To make sure that the mean tree sizes generated by each method is equal to the target value; we changed the *MaxTreeSize* several times and determined the values that lead to mean tree size of 15.5. Since in most of the cases there is no value for *MaxTreeSize* that produce exactly 15.5 as the mean tree sizes, two *MaxTreeSize*s that produce larger and smaller mean tree sizes are determined and then a linear estimation is performed to calculate the final results for the exact 15.5 mean tree size.

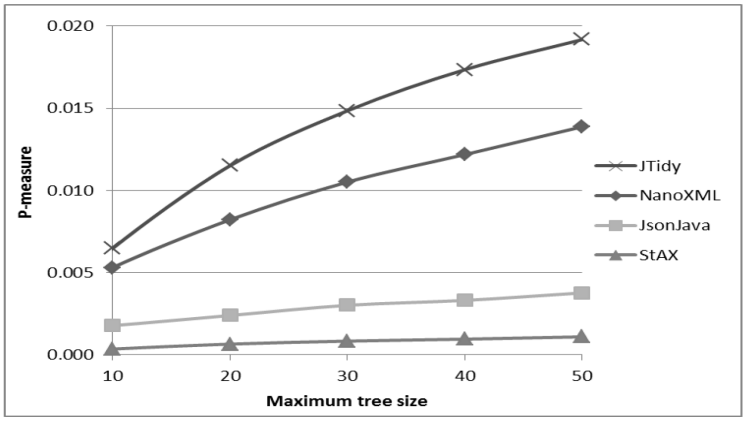


Figure 5.1. Analysis of failure detection against the tree sizes. Random tree generation with test set size of 8 is used.

5.6 Experimental Result and Discussion

The result of the empirical study is presented in this section. First, the detailed result of each program under the test is presented. It is followed by statistical analysis of the

results. Then, different tree distance functions are used in test case generation methods. Hence, a comparison among different tree distance functions is made in the context of tree test generation. Finally, the failure detection results are demonstrated where MOGA for strings, from chapter 3, is used for node value generation.

5.6.1 Results of Each Program Under Test

In this sub-section, two sets of results are presented; the "same maximum tree size" experiment and the "same mean tree size" experiment as described in Section 5.5.6. In Table 5.2, the result of each program under test is provided where the *MaxTreeSize* is set to 30 for all the test generation methods. Every number in this table is a percentage indicating the p-measure improvement of that method over the base line random tree generation. Similar to chapter 3 on strings, each number is calculated using (3.9). The results in this table indicate that all the tree generation methods produce better results than random tree generation. Moreover, GA produced the best results for all the programs. To summarize this table GA is best method and MOGA is in the second place when *MaxTreeSize* is similar for all methods. ARTOO is next, then FSCS, and finally random tree generation.

Table 5.2. The percentage of p-measure improvement of each method over RT where maximum tree size is set to a constant number of 30 and EST tree distance function is used.

Test Set Size	Software Under Test	FSCS	ARTOO	GA	MOGA
4	NanoXML	1.8%	3.1%	29.2%	15.9%
	JsonJava	0.3%	11.5%	37.4%	37.2%
	StAX	6.3%	14.1%	55.8%	45.0%
	JTidy	1.2%	5.8%	18.8%	12.0%
	Average	2.4%	8.6%	35.3%	27.5%
6	NanoXML	6.2%	8.5%	22.4%	14.1%
	JsonJava	5.9%	7.9%	26.9%	21.5%
	StAX	14.7%	18.8%	42.3%	36.2%
	JTidy	6.6%	8.6%	17.9%	10.8%
	Average	8.3%	11.0%	27.3%	20.7%
8	NanoXML	9.6%	10.2%	18.3%	13.3%
	JsonJava	5.7%	10.2%	19.3%	14.4%
	StAX	14.8%	13.4%	27.2%	24.2%
	JTidy	9.3%	8.4%	18.0%	12.2%
	Average	9.9%	10.5%	20.7%	16.0%
10	NanoXML	10.1%	10.5%	16.9%	14.9%
	JsonJava	7.7%	7.7%	16.0%	13.6%
	StAX	11.5%	12.4%	18.2%	16.3%
	JTidy	10.0%	9.5%	16.2%	12.3%
	Average	9.8%	10.0%	16.8%	14.3%

The GA outperforms the MOGA in Table 5.2 since it produces, on average, larger trees than the trees produced by MOGA. As discussed in Section 5.5.6, the mean of tree sizes is a covariant affecting the failure detection effectiveness, where larger trees produce better results, on average. Hence, in the second set of experiments, we set the mean size of trees as a fix number to attempt to compare the tree generation methods independently of tree size. Table 5.3 demonstrates the improvement of test generation methods compared to random tree generation where the generated trees have a same mean size of 15.5. Accordingly, MOGA "outperforms" the GA in most of the cases. Further, GA and MOGA are always better than the FSCS, ARTOO, and of course random generation.

Finally, Table 5.4 provides raw P-measure results for the RT method for the sake of completeness. This allows the reader to compute the P-measure of each method if required.

Table 5.3. The percentage of p-measure improvement of each method over RT where mean tree size is adjusted to 15.5 and EST tree distance function is used.

Test set Size	Software Under Test	FSCS	ARTOO	GA	MOGA
4	NanoXML	4.5%	3.1%	15.8%	15.5%
	JsonJava	2.1%	11.5%	31.2%	33.2%
	StAX	13.2%	14.1%	55.0%	48.9%
	JTidy	6.2%	5.8%	13.6%	18.3%
	Average	6.5%	8.6%	28.9%	29.0%
6	NanoXML	9.5%	11.7%	14.2%	14.3%
	JsonJava	5.9%	5.9%	16.9%	20.2%
	StAX	17.1%	17.5%	38.1%	36.9%
	JTidy	5.8%	5.7%	8.3%	11.6%
	Average	9.6%	10.2%	19.4%	20.7%
8	NanoXML	9.4%	10.5%	11.0%	13.2%
	JsonJava	5.7%	6.5%	12.0%	14.5%
	StAX	13.6%	14.3%	25.6%	23.8%
	JTidy	5.3%	6.9%	7.7%	10.1%
	Average	8.5%	9.5%	14.1%	15.4%
10	NanoXML	8.4%	7.9%	9.5%	10.5%
	JsonJava	1.9%	6.4%	8.6%	13.1%
	StAX	10.3%	11.3%	17.9%	15.2%
	JTidy	6.3%	6.4%	7.4%	8.5%
	Average	6.7%	8.0%	10.9%	11.9%

Table 5.4. The raw P-measure results for RT where the EST tree distance is used.

Software Under Test	Test set size			
	4	6	8	10
NanoXML	0.00641	0.00867	0.01052	0.01128
JsonJava	0.00188	0.00251	0.00302	0.00344
StAX	0.00060	0.00073	0.00084	0.00092
JTidy	0.00963	0.01270	0.01485	0.01653

5.6.2 Statistical Analysis of Results

The results in Table 5.2 and Table 5.3 are averaged over 100 trial runs. To formally indicate the performance of each test case generation method against RT, we performed a test of statistical significance (z-test, one tailed) with a conservative type I error of 0.01 [90], similar to chapter 2 and 3 on numerical and string test cases. Our working hypothesis is that MOGA, GA, FSCS, and ARTOO will produce superior results compared to RT. Further, an effect size (Cohen's method [56], [57]) between the each method and RT is calculated.

To perform a z-test or calculate effect size, the results must be normally distributed. As discussed in chapter 2 and 3, according to [50], p-measure values are normally distributed. Further, we investigated the normality of the results more deeply by performing a Shapiro-Wilk test [96]; it works based on a null hypothesis that the data is normally distributed. According to the results of this test, the normality of the p-measure values cannot be rejected.

Table 5.5 represents the effect sizes for the "same *MaxTreeSize*" experiment. Similarly, Table 5.6 presents the effect sizes for the "same mean tree size" experiment. In both tables, the "*" beside an effect size demonstrates the result of the z-test. The test measures if a statistically significant difference exists between RT and a tree generation method. In each table, only one case related to the FSCS method is found where the z-test shows insignificant improvement compared to RT. All other results demonstrate significant improvement in failure detection for each method against RT. Further, in both tables, most of the GA and MOGA effect sizes are more than 0.8 which is considered to be a large improvement according to Cohen's definition [56]–[58]. Regarding the FSCS and ARTOO methods, most of the results are larger than 0.5 (Cohen's definition of medium).

Table 5.5. The effect size between RT and other methods where the maximum tree size is set to 30 and EST tree distance is used. "*" indicates the result of the z-test where a significant difference exists at the 0.01 level.

Test set Size	Software Under Test	FSCS	ARTOO	GA	MOGA
4	NanoXML	0.07	0.11	1.16*	0.63*
	JsonJava	0.01	0.43*	1.59*	1.52*
	StAX	0.24*	0.52*	2.40*	1.78*
	JTidy	0.10	0.46*	1.53*	0.97*
6	NanoXML	0.27*	0.36*	1.06*	0.69*
	JsonJava	0.29*	0.37*	1.23*	1.05*
	StAX	0.74*	1.01*	2.77*	2.18*
	JTidy	0.84*	1.12*	2.21*	1.32*
8	NanoXML	0.56*	0.54*	0.99*	0.80*
	JsonJava	0.32*	0.53*	1.08*	0.87*
	StAX	1.01*	0.91*	2.21*	1.88*
	JTidy	1.38*	1.29*	2.61*	1.80*
10	NanoXML	0.58*	0.60*	0.96*	0.83*
	JsonJava	0.44*	0.49*	1.06*	0.77*
	StAX	1.06*	1.14*	1.93*	1.64*
	JTidy	1.75*	1.71*	2.77*	2.13*

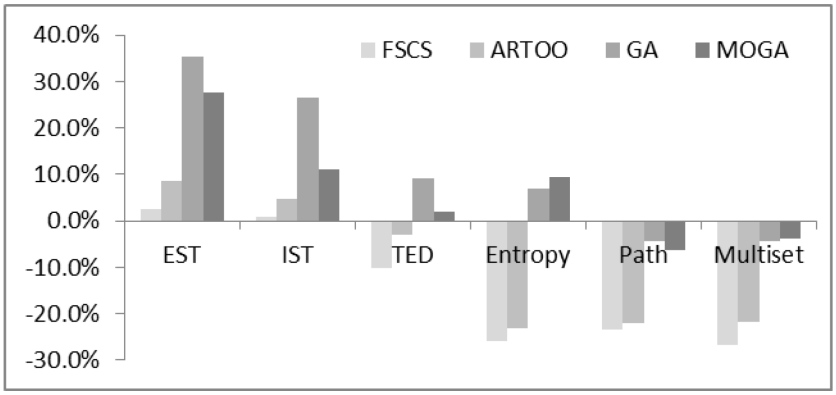
Table 5.6. The effect size between RT and other methods where the mean tree size is adjusted to 15.5 and EST tree distance is used. "*" indicates the result of the z-test where a significant difference exists at the 0.01 level.

Test set Size	Software Under Test	FSCS	ARTOO	GA	MOGA
4	NanoXML	0.17	0.11	0.57*	0.59*
	JsonJava	0.09	0.43*	1.31*	1.42*
	StAX	0.48*	0.52*	2.40*	2.02*
	JTidy	0.51*	0.46*	1.09*	1.46*
6	NanoXML	0.44*	0.57*	0.73*	0.70*
	JsonJava	0.29*	0.30*	0.86*	0.98*
	StAX	0.93*	0.92*	2.38*	2.23*
	JTidy	0.79*	0.74*	1.12*	1.41*
8	NanoXML	0.51*	0.54*	0.61*	0.80*
	JsonJava	0.30*	0.36*	0.73*	0.85*
	StAX	0.90*	0.99*	2.09*	1.87*
	JTidy	0.83*	1.04*	1.16*	1.57*
10	NanoXML	0.48*	0.45*	0.55*	0.57*
	JsonJava	0.12	0.39*	0.52*	0.81*
	StAX	0.89*	1.02*	1.91*	1.50*
	JTidy	1.16*	1.15*	1.18*	1.43*

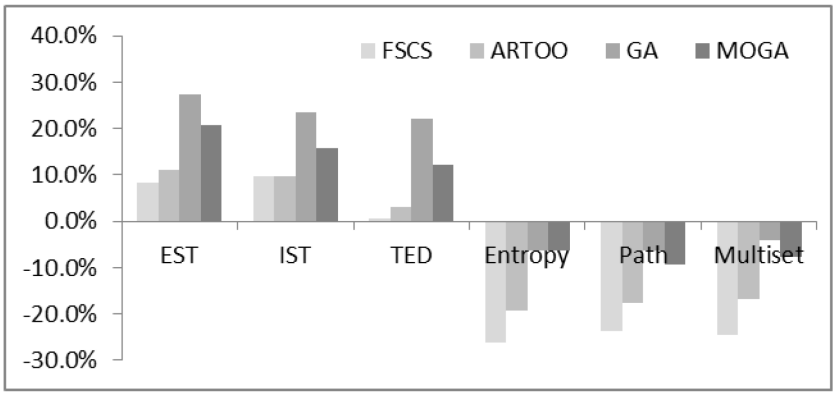
5.6.3 Comparison of Tree Distance Functions

The p-measure results for all six tree distance functions that are discussed in Section 5.4 are presented in Figure 5.2 and Figure 5.3. Results for each tree generation method and each tree distance function are illustrated where each column is the mean of all programs under the test. Figure 5.2 represents the "same *MaxTreeSize*" experiment, while "same mean tree size" experiment is presented in Figure 5.3. In each of these figures, five graphs are presented where the first four relate to the four test set sizes (4, 6, 8, and 10) and the last one is the average of all the test set sizes.

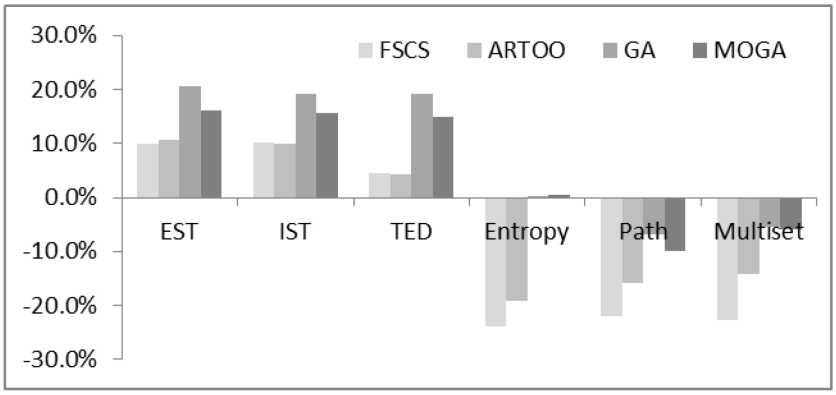
According to these graphs, the proposed EST tree distance function produces superior results compared to the other five distance functions. Any tree generation method has normally better performance when used with EST. After EST, IST and then TED are normally on second and third places, respectively. In Figure 5.2, the Entropy, Path, and Multiset distance functions produce negative results in most cases. This means under performance compared to RT. That is, significantly smaller trees are generated while these distance functions are utilized in "same *MaxTreeSize*" experiment. However, positive results are generated when a same mean size for trees is used.



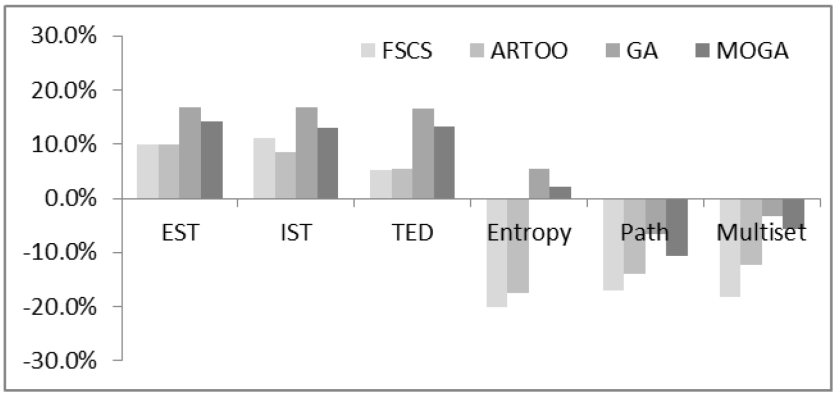
(a) Test set size = 4



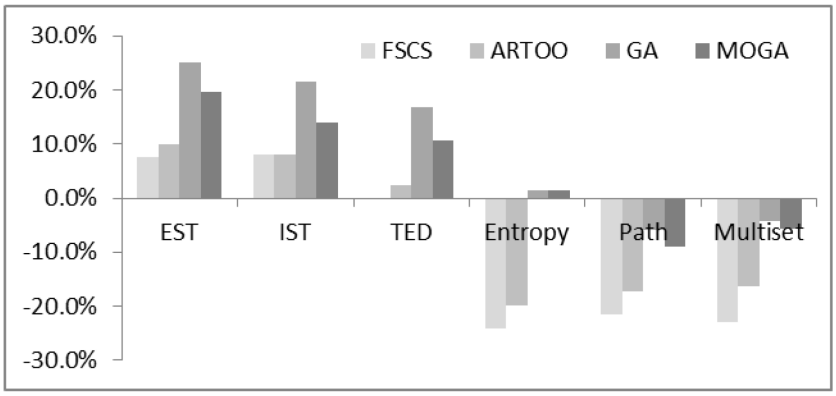
(b) Test set size = 6



(c) Test set size = 8

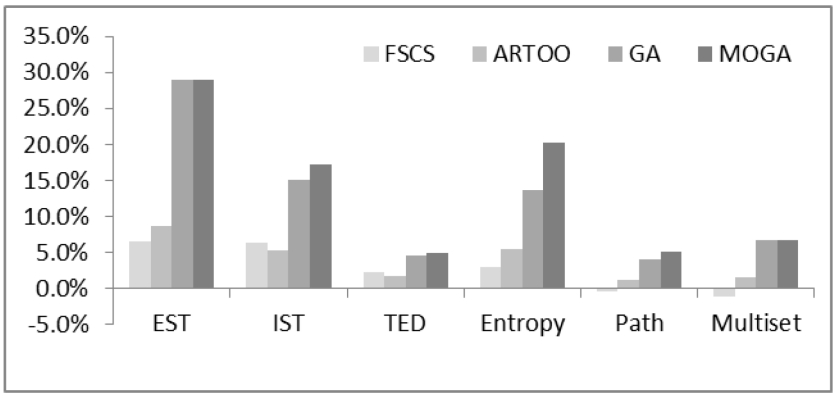


(d) Test set size = 10

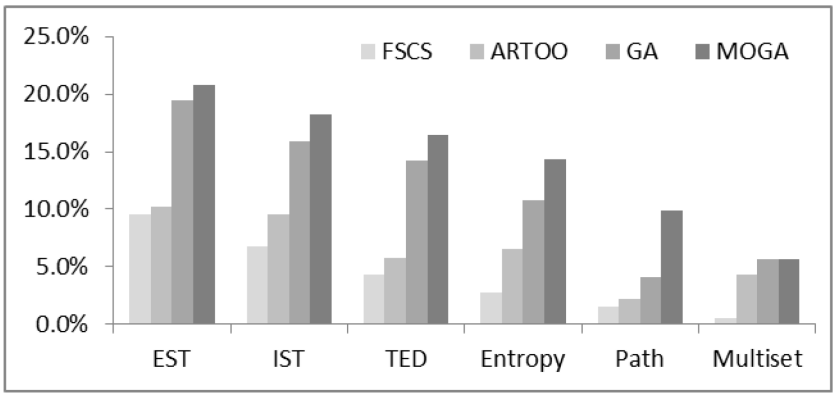


(e) Mean of all test set sizes

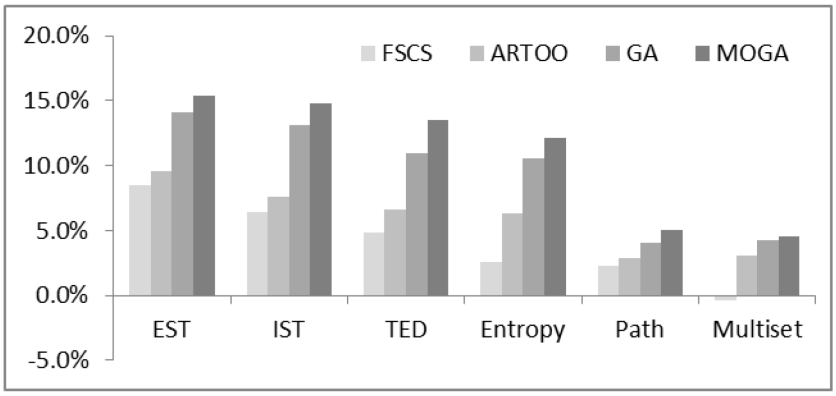
Figure 5.2. Comparison of tree distance functions where maximum tree size is 30. Each column denotes mean of p-measure improvement over all programs. (a), (b), (c), and (d) represent results for test set sizes of 4, 6, 8, and 10, respectively. (e) presents the mean of all test set sizes.



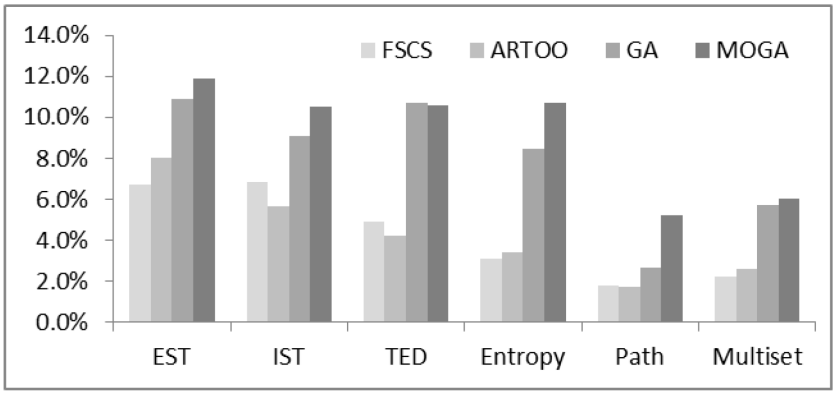
(a) Test set size = 4



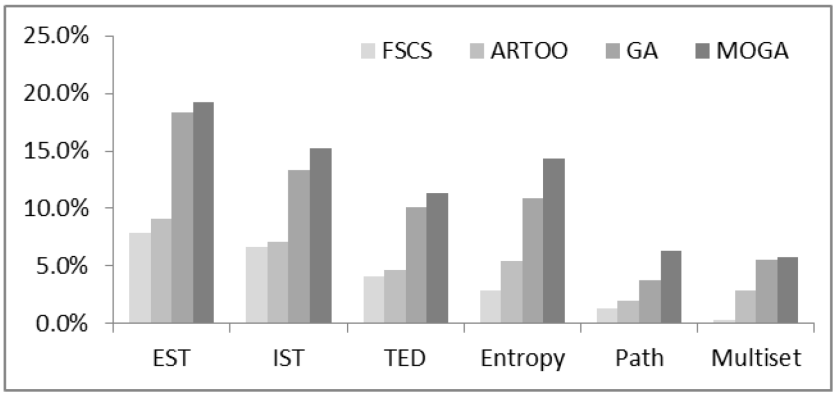
(b) Test set size = 6



(c) Test set size = 8



(d) Test set size = 10



(e) Mean of all test set sizes

Figure 5.3. Comparison of tree distance functions where mean tree size is 15.5. Each column denotes mean of p-measure improvement over all programs. (a), (b), (c), and (d) represent results for test set sizes of 4, 6, 8, and 10, respectively. (e) presents the mean of all test set sizes.

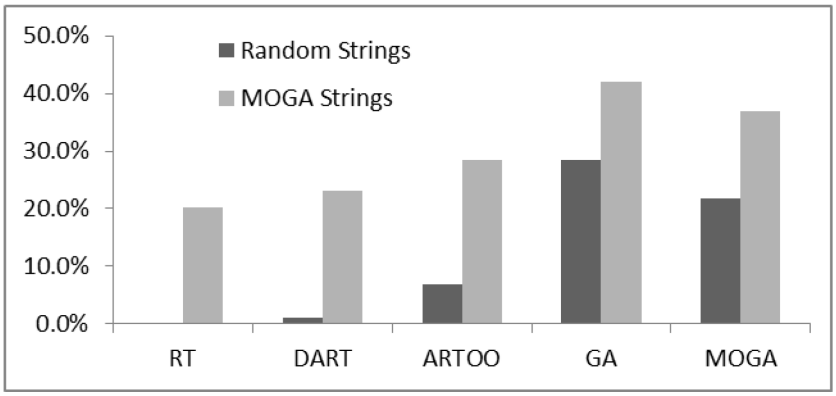
5.6.4 Node Value Generation by MOGA

In the final experiment, we investigate the effect of tree node values on failure detection. In all the previous results, RT are used to produce strings in the decoding process as

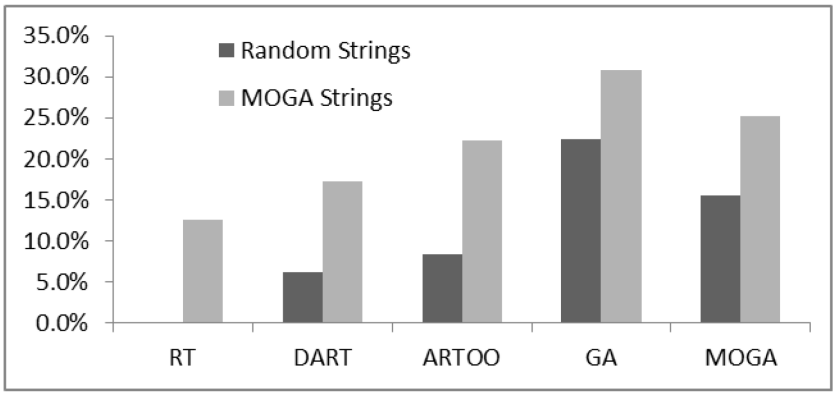
described in Section 5.5.3. Now, we produce the strings required for node values in a decoding process according to the MOGA string generation method in chapter 3 and compare the results with random string values. The details of applying MOGA string generation in a decoding process are discussed in Section 5.5.3.

The EST tree distance function is used to generate the trees since empirical evidence indicates its superior performance over other distance functions. However, it is not an important parameter in this experiment since for both cases (RT and MOGA strings in the decoding process) the same abstract trees are generated. Further, we only performed the experiment with the same *MaxTreeSize* setting. It is not necessary to perform the experiment with the same mean tree size since it will only affect the trees' structure. That is, we are comparing different decoding processes and the methods or settings that produce or affect the abstract trees are irrelevant.

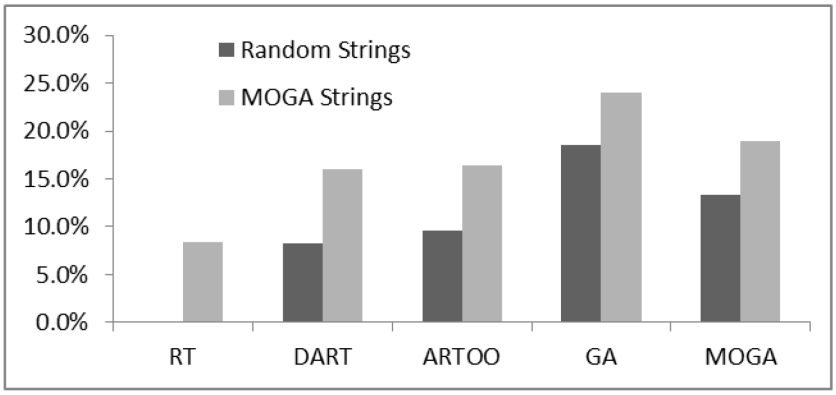
The results are provided in Figure 5.4 where every column is again an improvement against the base line RT. Replacing the RT string generation with MOGA improved the results for three of the four programs. The MOGA string generation had no effect on the "StAX" program's results. Hence, the results for "StAX" were identical with RT and MOGA string generation. Accordingly, each column provided in Figure 5.4 is the mean of all programs except "StAX". This figure indicates a significant improvement in the results when MOGA is used in a decoding process for string generation.



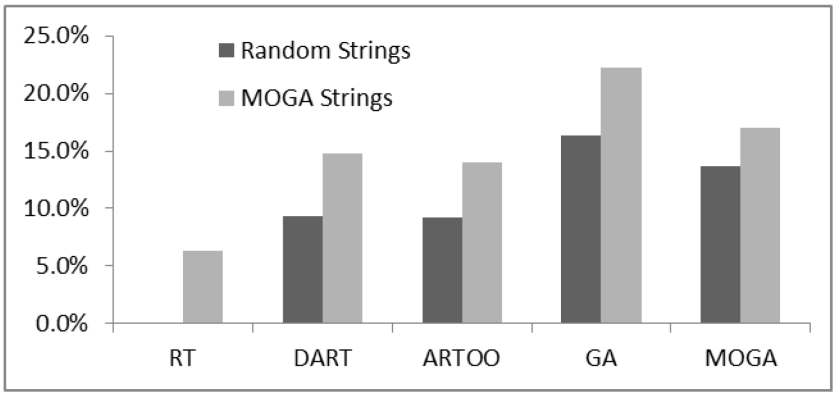
(a) Test set size = 4



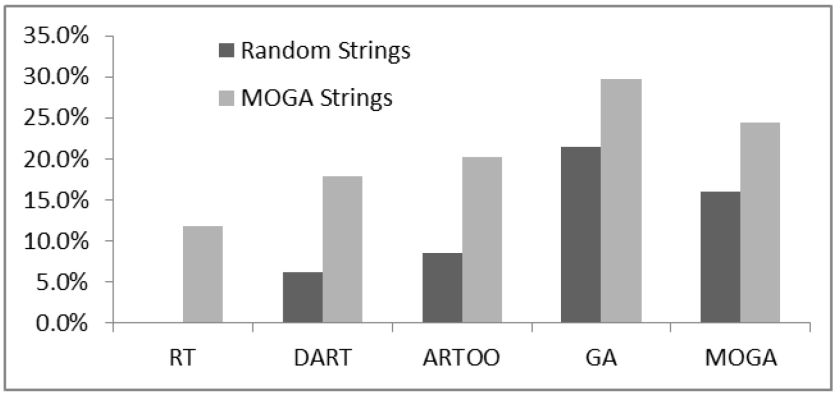
(b) Test set size = 6



(c) Test set size = 8



(d) Test set size = 10



(e) Mean of all test set sizes

Figure 5.4. Comparison of RT and MOGA string generation for tree node values where max tree size is 30. Each column denotes mean of p-measure improvement over three programs (NanaXML, JsonJava, and JTidy). The EST tree distance function is used for all tree generation methods. (a), (b), (c), and (d) represent results for test set sizes of 4, 6, 8, and 10, respectively. (e) presents the mean of all test set sizes.

5.7 Related Works

This section reviews research related to tree or XML test case generation. Most of the works with respect to XML test data generation use XML schemas to produce XML files that conform to the schema. Our work is different in this regard as it produces tree test data based on an abstract tree model.

Bertolino et al. [152]–[154] introduced a tool called TAXI that generates XML test data based upon a XML schema. TAXI implements the category partition testing approach on XML data [154]. First, TAXI read the schema and every choice element is weighted (The user can modify the default weights). Then, a set of sub-schemas are produced so that each one contains a different selection in choice elements [154]. Finally, values are populated into the sub-schemas. The values can be defined by a user or can be automatically extracted from the definitions of the input schema. XMLMate [155] is another tool that produces XML test data using XML schemas. XMLMate is white-box tool where a GA is used to generate XML test cases that maximize the code coverage [155] . This work is an extension of the EvoSuite tool [156] which is a general white-box test generator. Further, Feldt and Poulding [157] use metaheuristic search to produce unlabeled random trees where generated trees have the specified mean size and height. However, no evaluation is performed in the context of software testing. In addition, ToXgene [158] is a tool to generate XML documents which requires a TSL (Template Specification Language). The TSL document needs to be manually created by a user

since currently there is no automated approach to generate it from the XML schema.

A web service request and response are in a XML format. Hence, a category of related researches are the studies performed on the testing of the web services. Offut et al. [159]–[161] mutate XML requests for web services via data perturbation in order to test the web services. A valid XML request (input) is mutated where values of nodes are modified [160]. Boundary values defined by XML schema are used to replace the node values [160], [162]. Bai et al. [163] produce test cases to test web services. WSDL (Web Services Specification Language) is used to automatically generate the test cases. WSDL includes a specification of a web service. Similarly, Vanderveen et al. [164] generate web service requests automatically. They produce a context-free grammar from WSDL. Then, a string constraint solver is used to generate the XML files from the grammar. Further, WSDLTest [165] is a tool to automatically test the web services. It produces two objects from the schema. One is the service request and the other one is the test script [165]. The web service request is generated randomly from the schema [165]. Finally, the TAXI tool, discussed earlier in this section, is further extended to generate test cases for web service testing. The new tool is called WS-TAXI [166] which produces test cases based on WSDL.

5.8 Summary

In this chapter, black-box tree test case generation is studied. A tree abstract model needs to be defined by a user for each problem and then, tree generation methods can produce diverse test cases. Faults normally occur in error crystals or failure regions based on various empirical studies [13]–[17]. Hence, producing a diverse set of test cases is an important aspect that can improve the performance of black-box test case generation.

Tree distance functions are required in each test generation method to produce diverse test cases. Several tree distance functions (EST, IST, TED, Entropy, Path, and Multiset) are tested as a part of the test case generation process. Among the investigated distance functions, the EST, a new distance function proposed in the previous chapter, outperformed the other distance functions.

Four tree test case generation methods (FSCS, ARTOO, GA, and MOGA) are investigated and compared against the random tree generation. Failure detection performance of these methods is investigated through an empirical study where four real-world programs are used as case studies. These programs accept input XML test cases

and hence, an abstract tree model for XML is defined. However, our work is not limited to XML generation and it can be applied to any type of test cases that can be modeled by a tree. For example, in previous chapter a few different data types were modeled by trees. The mutation technique is utilized to produce several faulty versions of each program. Then, the p-measure is used as a quantitative measure to evaluate the failure detection performance.

With respect to tree sizes, two set of experiments are performed where in the first one, the maximum tree size in each test generation method is set to a constant number. In the second set of experiments, the mean size of tree sizes is adjusted to a fixed value. The evaluation results demonstrate that GA is the best method in the same maximum tree size experiment. However, in the same mean size experiment, MOGA outperformed all other test generation methods. Finally, in the XML decoding process, we replaced the random string node value generation with MOGA string generation from chapter 3. This resulted in improved failure detection.

6 Conclusions and Future Works

6.1 Conclusions

In this thesis, black-box test case generation is studied. In black-box testing, we have no information from the source code. Various empirical studies [13]–[17] indicated that faults normally occur in error crystals or failure regions. Failure regions are areas in the input domain that trigger faults. This means that faults are mapped onto a cluster within the input domain [24]. Accordingly, producing a diverse set of test cases is more likely to detect a failure and hence, it can improve the performance of black-box test case generation compared to RT.

Accordingly, in this research, automatic generation of diverse set of test cases is investigated that improves the failure detection effectiveness. To this end, we developed strategies that outperform the current state of the art test generation approaches. We limited our scope into three data structures for test generation; numerical, string, and tree test cases. Any program that accepts one of these types as input can be tested.

For numerical test generation, in chapter 2, the novel RBCVT method has been proposed with the aim of increasing the effectiveness of numerical test case generation approaches. The RBCVT method cannot be considered as an independent approach since it requires an initial set of input test cases. This method is developed as an add-on to the previous ART and QRT methods enhancing the testing effectiveness by more evenly distributing test cases across the input space. In addition, the applied probabilistic approach for RBCVT generation, allows different sets of output to be produced from the same set of inputs which makes RBCVT an appropriate method for software testing applications.

Given the importance of the computational cost in a practical application, we optimized the probabilistic computational algorithm of the RBCVT approach. The proposed search algorithm reduces the RBCVT computational complexity from a quadratic to a linear time order regarding the size of the test set. However, ART methods still suffer from high runtime order. In this regard, the computational cost of RBCVT is quite feasible with respect to practical applications. It is worthwhile to state that since the RBCVT approach requires initial test cases, the computational cost of the input test set generation is added to the RBCVT calculation cost. Since the results provided in Tables 2.2-2.5 indicate, on average, “similar” results for RBCVT with different types of generators, we can select

the RT method, which is linear and adds a low computational overhead, onto the RBCVT execution. The principle contribution in numerical test generation is utilizing CVT to develop an innovative test cases generation approach, in particular RT-RBCVT-Fast with a linear order of computational complexity similar to RT.

An extensive experimental study has been performed for numerical test cases and the results demonstrate that RBCVT is significantly superior to all approaches for the block pattern in simulation framework at all failure rates as well as the studied mutants at all test set sizes. Although the magnitude of improvement in testing effectiveness results is higher for the block pattern compared to the point pattern, the results demonstrate statistically significant improvement in the point pattern. In contrast, ART methods have indicated less effectiveness than RT regarding point patterns at θ=0.01 (demonstrated in Figure 2.14). Although RBCVT's performance regarding strip pattern is statistically significant compared to the other approaches at $\theta = 10^{-2}$, the impact of RBCVT verses the other approaches tends to zero as the failure rate decreases. In fact, in the case of strip pattern, the impacts of all of the approaches reduce to the performance of RT as the failure rate decreases; this is demonstrated in Figure 2.12. In contrast, in block and point patterns, the performance of all the approaches verses RT usually stays constant or even increases as the failure rate reduces [61]. Randomness of test cases is an important factor with respect to software testing. Accordingly, the investigation of randomness in Section 2.8 demonstrates that RT, all ART methods and all corresponding RBCVT methods possess an appropriate degree of randomness.

Although in real life applications, test cases' dimension can be large, in most cases, they belong to an acceptable range. Test case generation often seeks to generate values with a specific purpose rather than generating test cases to exercise the entire system. For instance, Ciupa et al. [62] conducted an empirical study on several real world small routines using unit testing. Briand and Arcuri [49] have considered 11 programs, basic mathematical functions that appear in the ART literature [17], for empirical analysis. The generated test cases in these papers do not exceed four dimensions. Furthermore, some techniques like range coding [63] exist to reduce the dimension of the input space, especially when collections are considered as the input to the software under the test. As a result, where we do not have large dimensions, the linear RBCVT-Fast approach dominates over ART approaches regarding computational cost.

Finally, RT-RBCVT, ART-RBCVT, and QRT-RBCVT have been demonstrated to have

a superior performance against RT, ART, and QRT methods, respectively. Consequently, software testing practitioners can use RBCVT to enhance the existing strategies within their software testing toolbox. The use of RBCVT in software testing is straightforward since RBCVT can be included to the previous methods as an add-on.

With respect to string test case generation, in chapter 3, a multi-objective optimization approach is studied. Two objectives are introduced to produce effective string test cases. The first objective controls the diversity of the test cases within a test set. The second objective is responsible for controlling the length distribution of the string test cases. The Benford distribution is employed as an objective distribution. Accordingly, a Kolmogorov–Smirnov test [84] is utilized to construct the fitness function. When both objectives are enforced, using a multi-objective optimization technique, superior test cases are produced.

Further, several string distance functions are examined as a part of test case generation process (Levenshtein, Hamming, Cosine, Manhattan, Euclidian, and LSH distance functions). Among the investigated distance functions, the LSH [65] is a fast estimation of the Cosine string distance function. According to the runtime complexity analysis in Section 3.5, LSH improves the runtime complexity. Further, in Section 3.5, the runtime complexities of all test case generation methods are discussed.

An empirical study has been performed to evaluate the failure detection capability of the string test generation methods (RT, FSCS, ARTOO, GA, and MOGA). Thirteen real-world programs are used for the evaluation. Several faulty versions are produced for each program through a mutation technique. These programs perform string transformation and/or manipulation which make them a true test for situations where the input test cases are strings [92]. With respect to the evaluation results, the MOGA revealed the superior failure detection performance. Further, the empirical results of comparing different string distance functions indicate that the Levenshtein distance outperformed the others.

Randomness of the test cases is an important aspect of a test case generation algorithm. Correlated test cases may reduce the failure detection effectiveness as discussed in Section 3.8. As a consequence, an investigation of randomness on string test cases is performed; and it demonstrated that all the generated test cases possess an appropriate degree of randomness.

In chapter 4, the novel EST similarity function has been proposed for the domain of tree

structured data comparison with the aim of increasing the effectiveness of applications utilizing tree distance or similarity functions. This new approach seeks to resolve the problems and limitations of previous approaches, as discussed in Section 4.4.1. In addition, the new approach must enhance applications where a tree distance function is utilized. To achieve this goal, we first extensively analyzed other distance functions. Then, we identified situations where the studied distance functions have poor performance; and finally we propose the EST approach. The proposed EST approach preserves the structure of the trees by mapping subtrees rather than nodes. EST generalizes the edit base distances and mappings by breaking the one-to-one and order preserving mapping rules. Further, it introduces new rules for subtree mapping provided in Section 4.4.2.

An extensive experimental study has been performed to evaluate the performance of the proposed similarity function against previous research. Clustering and classification frameworks are designed to perform an unbiased evaluation according to K-medoid, KNN, and SVM along with four distinct data sets. The real-world data sets have appeared in a number of publications [103], [105], [106], [117], [118]; and hence, they are deemed to be reliable source of information. Further, using synthetic data sets, we investigated the effect of varying the number of classes in the evaluation. This extensive evaluation framework is one of the advantages of this research over previous researches such as [103], [105], [106], [117], and [118].

The results of the experimental studies demonstrate that the EST approach is superior to the other approaches with respect to classification and clustering applications. To evaluate the performance, accuracy and WAF, are used in Tables 4.2, 4.3, and 4.4, where, in general, EST is demonstrated to be a better option for the clustering and classification of tree structured data. However, the performance of a distance function varies with the domain of application; and hence, we cannot generalize the superior performance of EST to all domains of applications.

The computational cost of a tree distance function should be carefully considered for practical applications. Given T^p and T^q as the input trees to the distance function, we calculated the runtime order of the EST as $O(|T^p| \times |T^q| \times Min(|T^p|, |T^q|))$. Further, the runtime of all the clustering and classification experiments are measured where the proposed EST outperformed all other distance functions with respect to all data sets

except SIGMOD. In addition, an empirical analysis has been performed to compare the runtime of EST vs. other distance functions in different tree sizes. The result of this empirical investigation suggests that the runtime efficiency of EST, Entropy, and Path are better than the other distance functions. Accordingly, the conclusion can be drawn that the proposed EST is an appropriate approach for computationally restricted and real time applications. Finally, EST has been demonstrated to have a superior performance against TED, IST, Path, Entropy, and Multiset distance functions with respect to classification and clustering applications.

Tree test case generation is studied in chapter 5. A tree abstract model needs to be defined by a user for each problem and then, tree generation methods can produce diverse test cases. Four tree test case generation methods (FSCS, ARTOO, GA, and MOGA) are investigated and compared against the random tree generation. Failure detection performance of these methods is investigated through an empirical study where four real-world programs are used as case studies. These programs accept input XML test cases and hence, an abstract tree model for XML is defined. However, our work is not limited to XML generation and it can be applied to any type of test cases that can be modeled by a tree. For example, in chapter 4 a few different data types were modeled by trees. The mutation technique is utilized to produce several faulty versions of each program. Then, the p-measure is used as a quantitative measure to evaluate the failure detection performance.

With respect to tree sizes, two set of experiments are performed where in the first one, the maximum tree size in each test generation method is set to a constant number. In the second set of experiments, the mean size of tree sizes is adjusted to a fixed value. The evaluation results demonstrate that GA is the best method in the same maximum tree size experiment. However, in the same mean size experiment, MOGA outperformed all other test generation methods.

Tree distance functions are required in each test generation method to produce diverse test cases. Several tree distance functions (EST, IST, TED, Entropy, Path, and Multiset) are tested as a part of the test case generation process. Among the investigated distance functions, the EST, a new distance function that we proposed in chapter 4, outperformed the other distance functions. Finally, in the XML decoding process, we replaced the random string node value generation with MOGA string generation from chapter 3. This resulted in improved failure detection.

Computational cost of a test case generation method and its relation to the required time for other parts of the testing process is an important factor when the user need to decide what test generation method to use. Basically, an ATS (Automated Testing System) has three parts; test generation, test execution, and examination of the test results. So, the total time (t_t) is combination of all; $t_t = t_g + t_e + t_o$ where t_g, t_e, and t_o stand for generation time, execution time, and result examination time, respectively. Test generation and execution can be automated easier than test result examination. With respect to examination of the test results, two options are normally used:

- A test oracle is constructed to automate the test examination. The test oracle usually has a simplified definition of a defect. Does the system crash or not is an example of such a description. Here each crash is considered a "defect".
- The test results are investigated manually by the tester.

When t_e ^{t_e} is small (very small programs) and test result examination is fully automated (small t_o) one would be better off running more test cases instead of generating more efficient test cases, similar to [49]. In such a case, methods that have high runtime compared to random generation are not cost effective. However, industrial software's execution runtime is usually large enough to have adequate time for test generation. Further, test result examination is not typically fully automated, unless for simplistic defects like crash, and requires manual work by the tester. Hence, generating more effective test cases which normally have higher runtime than random test cases is believed to improve failure detection in most cases.

6.2 Recommendations for Future Research

Although the results of this research improve black-box software testing effectiveness, there is still room for improvement. This research can be extended for further investigation as follows:

1) Up to now, we have introduced methods to generate numerical, string, and tree test cases. However, there are many programs that the structures of their input are not one of these types. Therefore, future studies can be focused on exploring other test case structures.

Another approach is developing a test generation approach that can produce test cases for

any given structure. Grammar based testing is a technique used to produce test cases where the input structure of the program is specified with a grammar. Grammars are a set of rules that define all the valid possibilities for the input to the software. For example, HTML can be defined with grammars. In grammar based testing, test cases are produced based on the grammar rules. There are several studies on grammar based testing. For instance, rule coverage [167] is a method to generate test cases based on grammar. Generating all the possibilities based on grammar is often too large to be practical. Hence, in rule coverage, the objective of test generation is to cover every rule in the grammar at least once. As a further example, Hoffman et al. [168] utilizes covering array as a technique to generate grammar based test cases. With covering arrays, a test template with N parameters is produced where each parameter has a limited number of possibilities [168].

Although there are several works on grammar based test generation, to the best of our knowledge, there is no research on grammar based test generation that produces test cases based on diversity of generated test cases. To achieve this, first, a distance function between two test cases that are extracted from the grammar must be developed. Then, based on the distance function, a diversity objective can be defined similar to our work in this research. Finally, an optimization technique can be applied to produce effective grammar based test cases. In this process, the critical part is defining a proper distance function between grammars. This could be challenging as a grammar can be very complicated. To define a proper distance function many features of the grammar must be considered. For example, the selected rules to generate a test case are an important factor. Further, the order that rules are selected can be important. Different rules may have different importance in test generation that needs to be accounted for.

2) Furthermore, with respect to numerical, string, and tree test generation, more research can be performed. Regarding numerical test case generation, more research can be done to optimize higher dimension numerical test cases. In addition, our experimental results are on programs with up to four dimensions. Real programs with higher dimensions can be investigated in future researches. With respect to the RBCVT and other numerical test generators, normally, test cases are produced with a pre-fixed number of dimensions or numbers with a fixed array length. However, in many applications the input software accepts a variable length array. Further studies can be performed on generating diverse numerical test cases when the dimension of the input can be variable.

3) In string test case generation, strings are generated without any information from the program under the test. In many programs, regular expressions define the features of the valid input string to the software. Invalid strings may not be very effective as it may be filtered in early stages of the program under the test and hence, it may not have a good failure detection chance. Therefore, in case a regular expression is available for the program under the test, using it in the test generation process can improve the failure detection. Achieving this is challenging, as during the optimization, more specifically in GA in the offspring generation, strings are broken and recombined. This breaks the strings structure that is based on the regular expression. Similarly, in a mutation process, the regular expression pattern is broken as a character in the string is randomly added, deleted, or replaced. Hence, achieving this requires a new optimization algorithm that is aware of the regular expression. As a future study, a regular expression aware test case generation algorithm can be developed.

4) Regarding the tree test case generation, up to now, we considered diversity and size distribution as factors that influence the testing effectiveness. However, other parameters of a tree can be important in failure detection. As an example, the height of a tree or its ratio to the size of the tree may affect the failure detection performance. The complexity of nodes of the tree might be important as well. So, a direction for future study on tree test generation is investigating other parameters that affect failure detection performance. To investigate the effect of other parameters, a new fitness functions can be defined and added into the multi-objective optimization.

5) Further, regarding tree test generation, we used a tree model to generate tree test cases. So, a tree model needs to be defined by the tester for the program under the test. The tree model that we constructed our tree generation method based on it, is an ordered and labeled tree model. Further, the proposed tree distance function, as well as other tree distance functions that we investigated, works on the ordered and labeled trees. This may pose a limitation, where in an application the test cases can be modeled by unordered or non-labeled trees. A same argument can be made for non-testing related applications as investigated in chapter 4. In chapter 4, we performed experiments on clustering and classification applications. In all those experiments, applications were selected that data samples were able to be modeled by an ordered and labeled tree; since the proposed tree distance function works based on ordered and labeled trees. Again, this poses limitation on the applications. Hence, a future direction with respect to tree distance function is

expanding the EST (the proposed tree distance function) such that it supports unordered and non-labeled trees.

6) Finally, with respect to the tree test case generation, in applications where the input to the software is XML, it is quite popular that an XML schema is pre-defined. An XML schema specifies the characteristics of input XML file. Our work, in this thesis, does not support XML schema as extra information in test generation. Several works have been performed on generating XML test cases based on XML schema as we reviewed it in Section 5.7. However, none of them, to be best of our knowledge, works based on diversity. Hence, our work in this research can be extended to support XML schema. This might be challenging as every test cases that is generated or altered during optimization process still must conform to the XML schema definitions.

7) In this research, we proposed a new tree distance function (EST). EST's performance is compared with previous distance functions in a few applications including clustering, classification, and automated test case generation. However, our tree distance function can be applied into variety of applications. Natural language processing [107] and cross browser compatibility [108] are examples of applications of a tree distance function. Another application that potentially can benefit from our tree distance function is outlier detection for data that can be modeled as a tree; like XML. Outlier detection has numerous applications. For example, it can be used in fraud detection and noise removal (data cleaning). Several works has been done on XML outlier detection [169]. A new XML outlier detection approach can be the use of EST as a distance function between the XML documents. Any XML document that has relatively large distance with other data points can be potentially an outlier. Further, the EST can be applied to code clone detection. A source code of a programming language can be converted into an abstract syntax tree. Hence, the EST distance function can be used to detect code cloning if the distance for two source codes is less than a threshold. Consequently, new applications of the proposed tree distance function can be a potential direction for future researches.

8) An automated test generator must produce test cases that have a higher chance of detecting a failure. This reduces the cost of testing by faster failure detection and less manual work. In this study, we demonstrated that diversity among input test cases improves the chance of detecting a failure and hence, it improves the failure detection performance. Failure detection is improved since faults normally occur in error crystals or failure regions [13]–[17]; how about the outputs of the test cases? Can we use them to

produce test cases with higher chance of failure detection? Let's assume that for every input to the software under the test we have an corresponding output. If two different inputs lead to two similar outputs, we can argue that both test cases, with a degree of probability, have a similar execution path in the source code and hence, it is likely that both tests either fail or pass. Similarly, we can argue that if the two outputs are very different, probably the two test cases have different execution paths in the source code. As a result, we can say that a set of tests are diverse if their corresponding outputs are diversely distributed. The effect of the diversity of the outputs can be more than the effect of diversity of the inputs. Therefore, if we optimize the test cases such that their corresponding outputs are diversely distributed in the output space, we may be able to produce test cases with higher chance of failure detection. To do this, a proper distance function between the outputs must be developed. Please note that the output can have any structure like trees. For example, in a web browser, in the first stage, the input HTML is parsed into a DOM (Document Object Model) tree. So, for this stage, the input is HTML text and output is a DOM tree. Then, a diversity based objective function must be defined on the outputs. In the optimization process, inputs are generated and optimized based on the objective function. In such test generation, the optimization include execution the test cases in order to capture the output. This is a potential approach to improve the testing process and hence, a direction or future studies.

Bibliography

[1] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining behavior graphs for 'backtrace' of noncrashing bugs," in *Proceeding of the 2005 SIAM international conference on data mining (SDM'05), Newport Beach*, 2005, pp. 286–297.

[2] C. Jones, "Software quality in 2011: A survey of the state of the art," 2011. [Online]. Available: http://www.asq509.org/ht/a/GetDocumentAction/id/62711.

[3] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost," in *Proceedings of the 2006 international workshop on Automation of software test*, 2006, pp. 85–91.

[4] D. Hoffman, "A taxonomy for test oracles," *Qual. Week*, pp. 1–8, 1998.

[5] "Testing Anywhere." [Online]. Available: http://www.automationanywhere.com/Testing/products/automated-testing-anywhere.htm.

[6] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing," *Softw. Eng. IEEE Trans.*, vol. SE-10, no. 4, pp. 438–444, 1984.

[7] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .NET with feedback-directed random testing," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 87–96.

[8] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007, p. 1.

[9] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile security testing of Web-based systems via HTTPUnit," in *Agile Conference, 2005. Proceedings*, 2005, pp. 29–38.

[10] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *Quality Software, 2003. Proceedings. Third International Conference on*, 2003, pp. 20–23.

[11] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium*, 2000, pp. 59–68.

[12] S. Lipner and M. Howard, "The Trustworthy Computing Security Development Lifecycle document (SDL)," 2005.

[13] P. E. Ammann and J. C. Knight, “Data diversity: an approach to software fault tolerance,” *Comput. IEEE Trans.*, vol. 37, no. 4, pp. 418–425, Apr. 1988.

[14] G. B. Finelli, “NASA Software failure characterization experiments,” *Reliab. Eng. Syst. Saf.*, vol. 32, no. 1–2, pp. 155–169, 1991.

[15] L. J. White and E. I. Cohen, “A Domain Strategy for Computer Program Testing,” *Softw. Eng. IEEE Trans.*, vol. SE-6, no. 3, pp. 247–257, May 1980.

[16] P. G. Bishop, “The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail),” in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, 1993, pp. 98–107.

[17] C. Schneckenburger and J. Mayer, “Towards the determination of typical failure patterns,” in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 90–93.

[18] T. Y. Chen, T. H. Tse, and Y. T. Yu, “Proportional sampling strategy: a compendium and some insights,” *J. Syst. Softw.*, vol. 58, no. 1, pp. 65–81, 2001.

[19] A. F. Tappenden and J. Miller, “A Novel Evolutionary Approach for Adaptive Random Testing,” *Reliab. IEEE Trans.*, vol. 58, no. 4, pp. 619–633, 2009.

[20] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, “Code Coverage of Adaptive Random Testing,” *Reliab. IEEE Trans.*, vol. 62, no. 1, pp. 226–237, 2013.

[21] J. Lv, H. Hu, K.-Y. Cai, and T. Y. Chen, “Adaptive and Random Partition Software Testing,” *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, vol. PP, no. 99. p. 1, 2014.

[22] M. Li and P. M. B. Vitanyi, *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag New York Inc, 2008.

[23] M. Li, X. Chen, X. Li, B. Ma, and P. M. B. Vitanyi, “The similarity metric,” *Inf. Theory, IEEE Trans.*, vol. 50, no. 12, pp. 3250–3264, 2004.

[24] F. T. Chan, T. Y. Chen, I. K. Mak, and Y. T. Yu, “Proportional sampling strategy: guidelines for software testing practitioners,” *Inf. Softw. Technol.*, vol. 38, no. 12, pp. 775–782, 1996.

[25] T. Y. Chen, H. Leung, and I. K. Mak, “Adaptive Random Testing,” in *Advances in Computer Science - ASIAN 2004*, vol. 3321, M. Maher, Ed. Springer Berlin / Heidelberg, 2005, pp. 3156–3157.

[26] K. Chan, T. Chen, and D. Towey, “Restricted Random Testing,” in *Software Quality -- ECSQ 2002*, vol. 2349, J. Kontio and R. Conradi, Eds. Springer Berlin / Heidelberg, 2002, pp. 321–330.

[27] F.-C. Kuo, “An Indepth Study of Mirror Adaptive Random Testing,” in *Quality Software, 2009. QSIC ’09. 9th International Conference on*, 2009, pp. 51–58.

[28] J. Mayer, “Adaptive Random Testing by Bisection and Localization,” in *Formal Approaches to Software Testing*, vol. 3997, W. Grieskamp and C. Weise, Eds. Springer Berlin / Heidelberg, 2006, pp. 72–86.

[29] T. Y. Chen, R. Merkel, P. K. Wong, and G. Eddy, “Adaptive random testing through dynamic partitioning,” in *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, 2004, pp. 79–86.

[30] T. Y. Chen, D. Huang, and Z. Zhou, “Adaptive Random Testing Through Iterative Partitioning,” in *Reliable Software Technologies -- Ada-Europe 2006*, vol. 4006, L. Pinho and M. Gonzalez Harbour, Eds. Springer Berlin / Heidelberg, 2006, pp. 155–166.

[31] T. Y. Chen, F.-C. Kuo, and H. Liu, “Adaptive random testing based on distribution metrics,” *J. Syst. Softw.*, vol. 82, no. 9, pp. 1419–1433, 2009.

[32] J. Mayer and C. Schneckenburger, “An empirical analysis and comparison of random testing techniques,” in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006, pp. 105–114.

[33] K. P. Chan, T. Y. Chen, and D. Towey, “Forgetting Test Cases,” in *Computer Software and Applications Conference, 2006. COMPSAC ’06. 30th Annual International*, 2006, vol. 1, pp. 485–494.

[34] T. Y. Chen and R. Merkel, “Quasi-Random Testing,” *Reliab. IEEE Trans.*, vol. 56, no. 3, pp. 562–568, 2007.

[35] H. Chi and E. L. Jones, “Computational investigations of quasirandom sequences in generating test cases for specification-based tests,” in *Proceedings of the 38th conference on Winter simulation*, 2006, pp. 975–980.

[36] I. M. Sobol, “Uniformly distributed sequences with additional uniformity properties,” *J. Comput. Math. Math. Phys.*, vol. 16, pp. 236–242, 1976.

[37] J. H. Halton, “Algorithm 247: Radical-inverse quasi-random point sequence,” *Commun. ACM*, vol. 7, no. 12, pp. 701–702, Dec. 1964.

[38] H. Niederreiter, “Low-discrepancy and low-dispersion sequences,” *J. Number Theory*, vol. 30, no. 1, pp. 51–70, 1988.

[39] H. Faure, “Discr{é}pance de suites associ{é}esa un systeme de num{é}ration (en dimension un),” *Bull. Soc. Math. Fr.*, vol. 109, no. 2, pp. 143–182, 1981.

[40] P. Peart, “The dispersion of the Hammersley Sequence in the unit square,” *Monatshefte fur Math.*, vol. 94, no. 3, pp. 249–261, 1982.

[41] C. Schlier, “On scrambled Halton sequences,” *Appl. Numer. Math.*, vol. 58, no. 10, pp. 1467–1478, 2008.

[42] B. L. Fox, “Algorithm 647: Implementation and Relative Efficiency of Quasirandom Sequence Generators,” *ACM Trans. Math. Softw.*, vol. 12, no. 4, pp. 362–376, Dec. 1986.

[43] H. Everett, D. Lazard, S. Lazard, and M. Safey El Din, “The Voronoi Diagram of Three Lines,” *Discret. Comput. Geom.*, vol. 42, no. 1, pp. 94–130, 2009.

[44] Q. Du, V. Faber, and M. Gunzburger, “Centroidal Voronoi Tessellations: Applications and Algorithms,” *SIAM Rev.*, vol. 41, no. 4, pp. 637–676, 1999.

[45] L. Ju, Q. Du, and M. Gunzburger, “Probabilistic methods for centroidal Voronoi tessellations and their parallel implementations,” *Parallel Comput.*, vol. 28, no. 10, pp. 1477–1500, 2002.

[46] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. 2nd Edition. John Wiley & Sons, Inc., 2008.

[47] T. Y. Chen and R. Merkel, “Efficient and effective random testing using the Voronoi diagram,” in *Software Engineering Conference, 2006. Australian*, 2006, pp. 300–308.

[48] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: an efficient and robust access method for points and rectangles,” *SIGMOD Rec.*, vol. 19, no. 2, pp. 322–331, May 1990.

[49] A. Arcuri and L. Briand, “Adaptive random testing: an illusion of effectiveness?,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 265–275.

[50] T. Y. Chen, F.-C. Kuo, and R. Merkel, “On the statistical properties of testing effectiveness measures,” *J. Syst. Softw.*, vol. 79, no. 5, pp. 591–601, 2006.

[51] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-Directed Random Test Generation,” in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 75–84.

[52] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005.

[53] K. P. Chan, T. Y. Chen, F.-C. Kuo, and D. Towey, “A revisit of adaptive random testing by restriction,” in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, 2004, pp. 78–85 vol.1.

[54] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “MuJava: an automated class mutation system,” *Softw. Testing, Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.

[55] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing,” in *International Symposium on the Foundations of Software Engineering (FSE)*, 2014, p. 10 pages. TO APPEAR.

[56] J. Cohen, “A power primer,” *Psychol. Bull.*, vol. 112, no. 1, pp. 155–159, 1992.

[57] J. Cohen, *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum, 1988.

[58] L. A. Becker, “Effect Size (ES),” no. 1993, 2000.

[59] M. J. Meyer, “Martingale Java stochastic library.” [Online]. Available: http://martingale.berlios.de/Martingale.html.

[60] K. G. Morse Jr, “Compression tools compared,” *Linux J.*, vol. 2005, no. 137, pp. 62–66, 2005.

[61] T. Y. Chen, F. C. Kuo, and Z. Q. Zhou, “On favourable conditions for adaptive random testing,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 17, no. 6, pp. 805–825, 2007.

[62] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, “ARTOO: Adaptive Random Testing for Object-Oriented Software,” in *Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on*, 2008, pp. 71–80.

[63] D. Salomon, *Data compression: the complete reference*, vol. 10. Springer-Verlag New York Inc, 2007, pp. 127–129.

[64] C. Durtschi, W. Hillison, and C. Pacini, “The effective use of Benford’s law to assist in detecting fraud in accounting data,” *J. forensic Account.*, vol. 5, no. 1, pp. 17–34, 2004.

[65] L. Pauleve, H. Jegou, and L. Amsaleg, “Locality sensitive hashing: A comparison of hash function types and querying mechanisms,” *Pattern Recognit. Lett.*, vol. 31, no. 11, pp. 1348–1358, 2010.

[66] H. Hemmati, A. Arcuri, and L. Briand, “Achieving Scalable Model-based Testing Through Test Case Diversity,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 6:1–6:42, Mar. 2013.

[67] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, “Prioritizing test cases with string distances,” *Autom. Softw. Eng.*, vol. 19, no. 1, pp. 65–95, 2012.

[68] V. Ganesh, A. Kiezun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. Ernst, "HAMPI: A string solver for testing, analysis and vulnerability detection," in *Computer Aided Verification*, 2011, pp. 1–19.

[69] M. Harman and P. McMinn, "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search," *Softw. Eng. IEEE Trans.*, vol. 36, no. 2, pp. 226–247, Mar. 2010.

[70] D. Whitley, "A genetic algorithm tutorial," *Stat. Comput.*, vol. 4, no. 2, pp. 65–85, 1994.

[71] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[72] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *Softw. Eng. IEEE Trans.*, vol. 36, no. 6, pp. 742–762, Nov. 2010.

[73] M. Harman, "The Current State and Future of Search Based Software Engineering," in *2007 Future of Software Engineering*, 2007, pp. 342–357.

[74] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *Evol. Comput. IEEE Trans.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[75] K. A. De Jong and W. M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," in *Parallel problem solving from nature*, Springer, 1991, pp. 38–47.

[76] S. Newcomb, "Note on the Frequency of Use of the Different Digits in Natural Numbers," *Am. J. Math.*, vol. 4, no. 1, pp. 39–40, 1881.

[77] T. P. Hill, "The Significant-Digit Phenomenon," *Am. Math. Mon.*, vol. 102, no. 4, pp. 322–327, 1995.

[78] F. Benford, "The Law of Anomalous Numbers," *Proc. Am. Philos. Soc.*, vol. 78, no. 4, pp. 551–572, 1938.

[79] M. J. Nigrini and L. J. Mittermaier, "The use of Benford's law as an aid in analytical procedures," *Auditing*, vol. 16, pp. 52–67, 1997.

[80] C. L. Geyer and P. P. Williamson, "Detecting Fraud in Data Sets Using Benford's Law," *Commun. Stat. - Simul. Comput.*, vol. 33, no. 1, pp. 229–246, 2004.

[81] R. A. Raimi, "The First Digit Problem," *Am. Math. Mon.*, vol. 83, no. 7, pp. 521–538, 1976.

[82] A. Berger, T. P. Hill, and others, “A basic theory of Benford’s Law,” *Probab. Surv.*, vol. 8, pp. 1–126, 2011.

[83] J. J. Baroudi and W. J. Orlikowski, “The problem of statistical power in MIS research,” *MIS Q.*, pp. 87–106, 1989.

[84] M. A. Stephens, “Use of the Kolmogorov-Smirnov, Cramer-Von Mises and related statistics without extensive tables,” *J. R. Stat. Soc. Ser. B*, pp. 115–122, 1970.

[85] M. Alshraideh and L. Bottaci, “Search-based software test data generation for string data using program-specific search operators,” *Softw. Testing, Verif. Reliab.*, vol. 16, no. 3, pp. 175–203, 2006.

[86] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, 1966, vol. 10, no. 8, pp. 707–710.

[87] R. W. Hamming, “Error Detecting and Error Correcting Codes,” *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950.

[88] D. C. Anastasiu and G. Karypis, “L2AP: Fast cosine similarity search with prefix L-2 norm bounds,” in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, 2014, pp. 784–795.

[89] G. Xue, Y. Jiang, Y. You, and M. Li, “A Topology-aware Hierarchical Structured Overlay Network Based on Locality Sensitive Hashing Scheme,” in *Proceedings of the Second Workshop on Use of P2P, GRID and Agents for the Development of Content Networks*, 2007, pp. 3–8.

[90] A. Shahbazi, A. F. Tappenden, and J. Miller, “Centroidal Voronoi Tessellations-A New Approach to Random Testing,” *Softw. Eng. IEEE Trans.*, vol. 39, no. 2, pp. 163–183, 2013.

[91] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria,” *Softw. Eng. IEEE Trans.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[92] P. McMinn, M. Shahbaz, and M. Stevenson, “Search-Based Test Input Generation for String Data Types Using the Results of Web Queries,” in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 141–150.

[93] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, “Mutation Analysis.,” Yale University, Department of Computer Science, 1979.

[94] P. Tonella, “Evolutionary Testing of Classes,” in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 119–128.

[95] S. Afshan, P. McMinn, and M. Stevenson, “Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost,” in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, 2013, pp. 352–361.

[96] S. S. Shapiro and M. B. Wilk, “An analysis of variance test for normality (complete samples),” *Biometrika*, vol. 52, no. 3–4, pp. 591–611, 1965.

[97] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A Symbolic Execution Framework for JavaScript,” in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010, pp. 513–528.

[98] K. Lakhotia, M. Harman, and P. McMinn, “A Multi-objective Approach to Search-based Test Data Generation,” in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, 2007, pp. 1098–1105.

[99] G. Fraser, A. Arcuri, and P. McMinn, “Test Suite Generation with Memetic Algorithms,” in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, 2013, pp. 1437–1444.

[100] G. Fraser and A. Arcuri, “Whole Test Suite Generation,” *Softw. Eng. IEEE Trans.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[101] M. Shahbaz, P. McMinn, and M. Stevenson, “Automated Discovery of Valid Test Strings from the Web Using Dynamic Regular Expressions Collation and Natural Language Processing,” in *Quality Software (QSIC), 2012 12th International Conference on*, 2012, pp. 79–88.

[102] S. Yoo and M. Harman, “Pareto Efficient Multi-objective Test Case Selection,” in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007, pp. 140–150.

[103] M. J. Zaki, “Efficiently mining frequent trees in a forest: algorithms and applications,” *Knowl. Data Eng. IEEE Trans.*, vol. 17, no. 8, pp. 1021–1035, 2005.

[104] J. Punin, M. Krishnamoorthy, and M. Zaki, “LOGML: Log Markup Language for Web Usage Mining,” in *WEBKDD 2001 — Mining Web Log Data Across All Customers Touch Points*, vol. 2356, R. Kohavi, B. Masand, M. Spiliopoulou, and J. Srivastava, Eds. Springer Berlin / Heidelberg, 2002, pp. 273–294.

[105] M. J. Zaki and C. C. Aggarwal, “XRules: an effective structural classifier for XML data,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 316–325.

[106] W. Lian, D. W. -l. Cheung, N. Mamoulis, and S.-M. Yiu, “An efficient and scalable algorithm for clustering XML documents by structure,” *Knowl. Data Eng. IEEE Trans.*, vol. 16, no. 1, pp. 82–96, 2004.

[107] M. Kouylekov and B. Magnini, “Recognizing textual entailment with tree edit distance algorithms,” in *Proceedings of the First Challenge Workshop Recognising Textual Entailment*, 2005, pp. 17–20.

[108] A. Mesbah and M. R. Prasad, “Automated cross-browser compatibility testing,” in *Software Engineering (ICSE), 2011 33rd International Conference on*, 2011, pp. 561–570.

[109] A. Mesbah, A. van Deursen, and D. Roest, “Invariant-Based Automatic Testing of Modern Web Applications,” *Softw. Eng. IEEE Trans.*, vol. 38, no. 1, pp. 35–53, 2012.

[110] R. Connor, F. Simeoni, M. Iakovos, and R. Moss, “A bounded distance metric for comparing tree structure,” *Inf. Syst.*, vol. 36, no. 4, pp. 748–764, 2011.

[111] D. Buttler, “A Short Survey of Document Structure Similarity Algorithms,” in *The 5th International Conference on Internet Computing*, 2004.

[112] P. Bille, “A survey on tree edit distance and related problems,” *Theor. Comput. Sci.*, vol. 337, no. 1–3, pp. 217–239, 2005.

[113] A. Muller-Molina, K. Hirata, and T. Shinohara, “A Tree Distance Function Based on Multi-sets,” in *New Frontiers in Applied Data Mining*, vol. 5433, S. Chawla, T. Washio, S. Minato, S. Tsumoto, T. Onoda, S. Yamada, and A. Inokuchi, Eds. Springer Berlin / Heidelberg, 2009, pp. 87–98.

[114] L. Kaufman, P. J. Rousseeuw, and others, *Finding groups in data: an introduction to cluster analysis*, vol. 39. Wiley Online Library, 1990.

[115] W. Zuo, D. Zhang, and K. Wang, “On kernel difference-weighted k-nearest neighbor classification,” *Pattern Anal. Appl.*, vol. 11, no. 3, pp. 247–257, 2008.

[116] E. Alpaydin, “Support Vector Machines,” in *Introduction to Machine Learning*, Second edi., The MIT Press, 2004, pp. 218–225.

[117] C. C. Aggarwal, N. Ta, J. Wang, J. Feng, and M. Zaki, “Xproj: a framework for projected structural clustering of xml documents,” in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 46–55.

[118] F. Hadzic and M. Hecker, “Alternative Approach to Tree-Structured Web Log Representation and Mining,” in *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*, 2011, vol. 1, pp. 235–242.

[119] K. Zhang and D. Shasha, “Simple fast algorithms for the editing distance between trees and related problems,” *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, 1989.

[120] K.-C. Tai, “The Tree-to-Tree Correction Problem,” *J. ACM*, vol. 26, no. 3, pp. 422–433, Jul. 1979.

[121] J. T. L. Wang and K. Zhang, “Finding similar consensus between trees: an algorithm and a distance hierarchy,” *Pattern Recognit.*, vol. 34, no. 1, pp. 127–137, 2001.

[122] A. Nierman and H. V Jagadish, “Evaluating structural similarity in XML documents,” in *Proc. 5th Int. Workshop on the Web and Databases (WebDB 2002), Madison, Wisconsin, USA*, 2002, pp. 61–66.

[123] E. Tanaka and K. Tanaka, “The tree-to-tree editing problem.,” *INT. J. PATTERN RECOG. ARTIF. INTELL.*, vol. 2, no. 2, pp. 221–240, 1988.

[124] G. Valiente, “An efficient bottom-up distance between trees,” in *String Processing and Information Retrieval, 2001. SPIRE 2001. Proceedings.Eighth International Symposium on*, 2001, pp. 212–219.

[125] K. Zhang, “Algorithms for the constrained editing distance between ordered labeled trees and related problems,” *Pattern Recognit.*, vol. 28, no. 3, pp. 463–474, 1995.

[126] T. Jiang, L. Wang, and K. Zhang, “Alignment of trees - an alternative to tree edit,” *Theor. Comput. Sci.*, vol. 143, no. 1, pp. 137–148, 1995.

[127] S. M. Selkow, “The tree-to-tree editing problem,” *Inf. Process. Lett.*, vol. 6, no. 6, pp. 184–186, 1977.

[128] S. Y. Lu, “A Tree-Matching Algorithm Based on Node Splitting and Merging,” *Pattern Anal. Mach. Intell. IEEE Trans.*, vol. PAMI-6, no. 2, pp. 249–256, Mar. 1984.

[129] S. Helmer, “Measuring the structural similarity of semistructured documents using entropy,” in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 1022–1032.

[130] R. Yang, P. Kalnis, and A. K. H. Tung, “Similarity evaluation on tree-structured data,” in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 754–765.

[131] D.-I. S. Rönnau, “Efficient Change Management of XML Documents,” Universität der Bundeswehr München, 2010.

[132] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change Detection in Hierarchically Structured Information,” in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.

[133] G. Cobena, S. Abiteboul, and A. Marian, “Detecting changes in XML documents,” in *Data Engineering, 2002. Proceedings. 18th International Conference on*, 2002, pp. 41–52.

[134] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[135] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Software Maintenance, 1998. Proceedings., International Conference on*, 1998, pp. 368–377.

[136] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese, “Fast detection of XML structural similarity,” *Knowl. Data Eng. IEEE Trans.*, vol. 17, no. 2, pp. 160–175, 2005.

[137] P. J. F. Groenen and K. Jajuga, “Fuzzy clustering with squared Minkowski distances,” *Fuzzy Sets Syst.*, vol. 120, no. 2, pp. 227–237, 2001.

[138] M. J. Zaki, “CSLOG data set.” [Online]. Available: http://www.cs.rpi.edu/~zaki/software/logml/.

[139] “XML SIGMOD Record.” [Online]. Available: http://www.sigmod.org/publications/sigmod-record/xml-edition.

[140] “Treebank data set.” [Online]. Available: http://www.cs.washington.edu/research/xmldatasets/.

[141] R. Rifkin and A. Klautau, “In Defense of One-Vs-All Classification,” *J. Mach. Learn. Res.*, vol. 5, pp. 101–141, Dec. 2004.

[142] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The WEKA data mining software: an update,” *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[143] C.-C. Chang and C.-J. Lin, “{LIBSVM}: A library for support vector machines,” *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[144] R. B. Cattell, “The Scree Test For The Number Of Factors,” *Multivariate Behav. Res.*, vol. 1, no. 2, pp. 245–276, 1966.

[145] Q. Zhao, V. Hautamaki, and P. Fränti, “Knee Point Detection in BIC for Detecting the Number of Clusters,” in *Advanced Concepts for Intelligent Vision Systems*, vol. 5259, Springer Berlin / Heidelberg, 2008, pp. 664–673.

[146] X. Tolsa, “Principal values for the Cauchy integral and rectifiability,” *Am. Math. Soc.*, vol. 128, no. 7, pp. 2111–2119, 2000.

[147] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, “Finding a ‘Kneedle’ in a Haystack: Detecting Knee Points in System Behavior,” in *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on*, 2011, pp. 166–171.

[148] R. Santelices, M. J. Harrold, and A. Orso, “Precisely Detecting Runtime Change Interactions for Evolving Software,” in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010, pp. 429–438.

[149] R. Santelices and M. J. Harrold, “Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution,” in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 195–206.

[150] “NanoXML.” [Online]. Available: http://nanoxml.sourceforge.net/orig/NanoXML-Java/introduction.html.

[151] “StAX.” [Online]. Available: http://stax.codehaus.org/.

[152] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, “Systematic Generation of XML Instances to Test Complex Software Applications,” in *Rapid Integration of Software Engineering Techniques*, 2007, vol. 4401, pp. 114–129.

[153] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, “Automatic Test Data Generation for XML Schema-based Partition Testing,” in *Proceedings of the Second International Workshop on Automation of Software Test*, 2007, p. 4–.

[154] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, “TAXI--A Tool for XML-Based Testing,” in *Companion to the Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 53–54.

[155] N. Havrikov, M. Höschele, J. P. Galeotti, and A. Zeller, “XMLMate: evolutionary XML test generation,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 719–722.

[156] G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-oriented Software,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 416–419.

[157] R. Feldt and S. Poulding, “Finding test data with specific properties via metaheuristic search,” in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 2013, pp. 350–359.

[158] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. Lyons, “ToXgene: An extensible template-based data generator for XML,” in *In WebDB*, 2002, pp. 49–54.

[159] S. C. Lee and J. Offutt, “Generating test cases for XML-based Web component interactions using mutation analysis,” in *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, 2001, pp. 200–209.

[160] J. Offutt and W. Xu, “Generating Test Cases for Web Services Using Data Perturbation,” *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–10, Sep. 2004.

[161] W. Xu, J. Offutt, and J. Luo, “Testing Web services by XML perturbation,” in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, 2005, p. 10 pp.–266.

[162] J. B. Li and J. Miller, “Testing the semantics of W3C XML schema,” in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, 2005, vol. 1, pp. 443–448 Vol. 2.

[163] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, “WSDL-based automatic test case generation for Web services testing,” in *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, 2005, pp. 207–212.

[164] P. Vanderveen, M. Janzen, and A. F. Tappenden, “A Web Service Test Generator,” in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, 2014, pp. 516–520.

[165] H. M. Sneed and S. Huang, “WSDLTest - A Tool for Testing Web Services,” in *Web Site Evolution, 2006. WSE ’06. Eighth IEEE International Symposium on*, 2006, pp. 14–21.

[166] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, “WS-TAXI: A WSDL-based Testing Tool for Web Services,” in *Software Testing Verification and Validation, 2009. ICST ’09. International Conference on*, 2009, pp. 326–335.

[167] M. Hennessy and J. F. Power, “An Analysis of Rule Coverage As a Criterion in Generating Minimal Test Suites for Grammar-based Software,” in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 104–113.

[168] D. Hoffman, H.-Y. Wang, M. Chang, D. Ly-Gagnon, L. Sobotkiewicz, and P. Strooper, “Two case studies in grammar-based test generation,” *J. Syst. Softw.*, vol. 83, no. 12, pp. 2369–2378, 2010.

[169] G. Manco and E. Masciari, “XML Class Outlier Detection,” in *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, 2012, pp. 155–164.