Black-Box String Test Case Generation through a Multi-Objective Optimization

Ali Shahbazi, *Student Member, IEEE*, and James Miller, *Member, IEEE*

Abstract—String test cases are required by many real-world applications to identify defects and security risks. Random Testing (RT) is a low cost and easy to implement testing approach to generate strings. However, its effectiveness is not satisfactory. In this research, 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 based on the hypothesis that the population of strings is right-skewed within its range. 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.

Index Terms—Adaptive random testing, automated test case generation, black-box testing, mutation, random testing, software testing, string distance, string test cases

1 Introduction

Every year $500 billion is lost due to poor software quality [1]. Since black-box testing is a common testing strategy, any improvement in this domain could have a significant effect on this loss. Black-box methods, such as Random Testing (RT) [2] and Adaptive Random Testing (ART) [3], generate test cases without considering the program's source code [4]. Therefore, these methods are independent from the language of the source code. As a result, black-box testing methods are very general; all that is needed is the structure of the inputs and the outputs of the program under test. In this research, we focus on black-box testing methods, specifically where a test case (input value) is a string.

RT is a straightforward testing approach. RT's application in industry includes .NET error detection [5], security assessment [6], [7], Java Just-In-Time (JIT) compilers [8], and Windows NT robustness assessment [9]. Many companies use RT to detect security bugs; e.g., the Trustworthy Computing Security Development Lifecycle document (SDL) [10] 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., [11], [12], [13], [14], [15], faults usually occur in continuous regions within the input domain. This is referred to as error crystals by Finelli [12]. This means that faults are often clustered in the input space [16]. 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 two-dimensional space are presented in Fig. 1, 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.

ART approaches [3], [17], [18] were developed to enhance the performance of RT. ART approaches generate more effective test cases by producing more diverse, but still random, test cases across the input domain. Therefore, the probability of fault detection is improved [17].

To improve these test case methods, we have proposed a new approach, namely Random Border Centroidal Voronoi Tessellations (RBCVT) [19]. However, RBCVT is limited to generating numerical test cases. Beside numerical inputs, many programs accept strings as their input. To be more precise, a string input in this context, and all the strings that are generated in the evaluations refer to unstructured ASCII string; however, extending the work to Unicode characters is straightforward. Hence, to expand our previous work, in this research, we focus on string test cases. Therefore, we develop methods to automatically generate string test cases.

The "string test case" is a general term and hence, we first define the scope of this work. In this research, the objective is string test case generation; not test case selection [20] or prioritization [21]. Further, this research focuses on black-box string test generation. White-box test generation methods, like symbolic execution [22], 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 [23]. The difference between this study and white-box string generators is that we generate the test cases without any information from the source code of the software under test (SUT) (black-box test case generation). Therefore,

• *The authors are with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. E-mail: ali.shahbazi@ualberta.ca, jm@ece.ualberta.ca.*

Manuscript received 14 Oct. 2014; revised 15 Sept. 2015; accepted 30 Sept. 2015. Date of publication 6 Oct. 2015; date of current version 22 Apr. 2016. Recommended for acceptance by P. McMinn. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2015.2487958

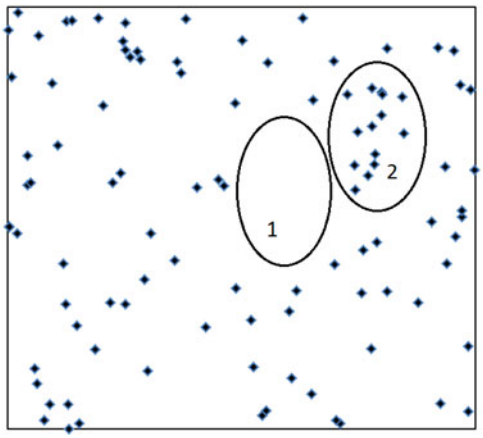

Fig. 1. RT fails to evenly distribute the test cases throughout the input domain. No test cases is produced in region one by the RT generator, whereas we have 14 test cases in region two with a same size.

white-box test generator techniques are outside of the scope of this study. These string-related techniques are reviewed in Section 10.

1.1 The Focus of This Study

In this research, 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 [11], [12], [13], [14], [15], 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 [11], [12], [13], [14], [15]. 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 work, we compared their performance when used in test generation. Different string distance functions 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 work 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 lengths of the generated strings plays an important role in failure detection. Fundamentally, it is believed that strings have two points of origin: other systems or human-input. Inputs from other systems are likely to be "files" created on that system. A file is considered as an encoded string of bytes. For example, Downey [24] presents a detailed study on the file sizes found on 562 file systems—their analysis shows that both very small files (less than 32 bytes) and very large files (greater than 32 MB) exist and that the distribution of files sizes are heavily skewed leftwards. For human-generated input, it is argued that strings will tend to be finite. Many data entries by humans are typically into forms and via incremental processes where multiple "smaller" strings are generated. Consider that the average individual can only enter 21 to 27 words per minute on modern smartphones; these types of physical limitations tend to imply that "shorter" replies are more likely [25]. While, the discussion is rather anecdotal in nature, it is believed that no comprehensive evidence exists on the topic, and perhaps given the ubiquitous nature of strings in software, it may never exist; and that the topic is likely to be non-stationary in nature, anyway. Hence, we view string generation as a mixture of these two categories, and hypothesize a number of loose conjectures about the nature of strings:

1) That the length of strings is highly variable and ranges from 0 to a metaphorical definition of infinite. However, many situations will only have strings lengths over a finite, but unknown, range [a, b].
2) That the distribution of the size of strings is not symmetrical. In fact, it tends to be highly leftwards skewed with regard to the midpoint of the range in many circumstances.

These conjectures imply that some form of length component be integrated into any solution, and 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 research 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) [26] technique, a fast estimation of string distances, to improve the runtime order complexity. Comprehensive runtime complexity improvement is discussed in Section 5. Further, empirical runtime analysis is investigated in Section 0.
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 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 rest of this paper is organized as follows. Section 2 reviews ART methods that are used in the experiments. An in-depth description of the optimization approach, and the fitness functions are presented in Section 3. Section 4 presents string distance functions used in this research. An

investigation on the order of runtime of the algorithms is presented in Section 5. The experimental framework is explained in Section 6. The results are demonstrated in Section 7. Section 8 investigates the degree of randomness among the generated test cases. Threats to validity of this research are discussed in Section 9. Section 10 includes related work on string test case generation. Finally, conclusions are drawn in Section 11.

2 Adaptive Random String Test Case Generation

To improve the poor effectiveness of RT, ART methods are introduced. Chen et al. [27] first introduced Fixed Size Candidate Set (FSCS) and then a variety of other ART methods have been developed by other researchers including Restricted Random Testing (RRT) [28], ART by partitioning [18], and Evolutionary ART (EAR) [17].

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) [29] methods are capable of more complex test case structures than a fixed size vector of numbers and they can be applied to string test cases. Further, Mayer and Schneckenburger [30] 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 study; these are reviewed in the following sections.

2.1 Fixed Size Candidate Set (FSCS)

To generate test cases, FSCS uses a distance based procedure. The first 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 test cases are generated as candidates ($K = 10$ is used in the experiments based on the recommendation of Chen et al. [27]). A candidate is selected where it has the largest distance from previously executed test cases as [19]

$$J = \underset{j=1,\ldots,K}{\operatorname{argmax}}\big(dist\big(cd_j, \beta\big(cd_j, T\big)\big)\big), \tag{1}$$

where cd_j represents the jth candidate, T refers to the test cases that are already generated, $\beta\big(cd_j, T\big)$indicates nearest test case in T to the cd_j, *dist* represents distance between two test cases, *argmax*() returns the index of an element with maximum value, and J denotes the index of the selected candidate.

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.

2.2 ART for Object Oriented Software (ARTOO)

ARTOO [29] 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 is similar to FSCS [29], 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) [29].

3 Evolutionary String Test Case Generation

To generate string test cases, evolutionary algorithms can be used. Among the evolutionary algorithms, Genetic Algorithms (GA) [31] are the most commonly used search algorithm in software engineering [32], [33], [34]. 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) [35] to optimize both fitness functions simultaneously.

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.

3.1.1 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 [11], [12], [13], [14], [15]. Hence, we define a fitness function that measures the diversity

$$F_D = \sum_{i=1}^{test\ set\ size} dist(t_i, \beta(t_i,\ test\ set)), \tag{2}$$

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.

3.1.2 GA Parameters

Using a GA requires the definition of its elements and parameters. In this study, a chromosome (a candidate solution) 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 [31]. 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 percent crossover rate [36]. 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 [31] is used to generate children strings from two strings. Single point crossover is applied to each and every corresponding pair of strings in the two parent chromosomes. In a single point recombination, a random point is selected in each of the two strings. Then, to generate the child strings, the first part of each string is concatenated to the second part of the other string.

Edit, delete, and add are used as mutation operators where every character in each string is mutated with 1 percent 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:

- No improvement is achieved in 20 generations based upon the fitness function; or,
- A maximum of 200 iterations is reached.

3.2 Multi-Objective Genetic Algorithm (MOGA)

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 [37], [38]. 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 [38], [39]. This is commonly known as Benford's Law [39]. 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 [40]. Fig. 2a represents the distribution of first

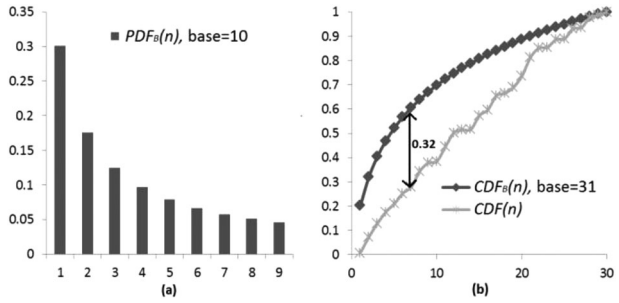

Fig. 2. (a) Benford distribution ($PDF_B(n)$) where base is 10. (b) Kolmogorov-Smirnov test is used to measure the distance of two distributions. $CDF(n)$ and $CDF_B(n)$ are cumulative probability distribution of the strings length and Benford, respectively. The max string length is assumed to be 30 which leads to the Benford base of 31.

digit numbers where the base is 10. The Benford distribution can be calculated using [40]

$$PDF_B(n) = \log_b(1/n + 1), 1 \leq n < b, \tag{3}$$

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

The Benford distribution is empirically investigated in many areas [40], [41]. 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 [40]. 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 [40], [42]. In addition, Raimi [43] 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 [40], [41], [44].

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 unbounded, 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 [40] has been used to test the compliance of a distribution with Benford distribution. However, it has low statistical power with small samples [45]. 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 [46]; this is more powerful when the sample size is small [46].As indicated in Fig. 2b, the Kolmogorov–Smirnov test finds the maximum distance between two cumulative probability distributions [46]. It can be represented as

$$F_L = \max_{n \in [1, StrMax]} |CDF(n) - CDF_B(n)|, \tag{4}$$

where F_L represents the string length fitness function. *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.

3.2.2 Pareto-Optimal Test Sets

A multi-objective optimization technique is required to apply both fitness functions simultaneously. We employ one of the widely used multi-objectives GAs, namely NSGA-II [35]. Since the diversity needs to be maximized, the value calculated from (2) is inverted (multiplicative inverse). Therefore, both fitness functions need to be minimized.

A basic step in NSGA-II is sorting of chromosomes in a population based on the concept of domination. Given that both fitness functions must be minimized, chromosome A dominates B if and only if

$$\begin{gathered}(F_D(A) < F_D(B) \text{ and } F_L(A) \leq F_L(B)) \\ \text{or} \\ (F_D(A) \leq F_D(B) \text{ and } F_L(A) < F_L(B)).\end{gathered} \tag{5}$$

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 Pareto fronts. The 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 [35]. 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. [35].

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 [35].
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 criteria 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.

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 based on the steps discussed in the previous section. NSGA-II uses binary tournament selection mechanism [35]. 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 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.

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 [20], [21], [29], [47]. 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 work.

Accordingly, we performed the experiments with six string distance functions. Four of which are Levenshtein [48], Hamming [49], Cosine [50], Manhattan [21], and Cartesian [21] distance functions that are repeatedly used in software testing studies [20], [21], [29], [47]. Further, we also used Locality-Sensitive Hashing [26] technique as a fast estimate of string distance in our work.

4.1 Levenshtein Distance

The Levenshtein Distance [48] is an edit-based distance that works based on three edit operations, "delete", "insert", and "update". 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. The Levenshtein distance assigns a unit cost to all edit operations [21].

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

$$lev(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) == 0 \\ \min\left\{ \begin{array}{c} lev(i-1,j)+1 \\ lev(i,j-1)+1 \\ lev(i-1,j-1)+cost(i,j) \end{array} \right\} & \text{otherwise} \end{cases}$$
$$cost(i,j) = \begin{cases} 0 & \text{if} Str1_i == Str2_j \\ 1 & \text{otherwise} \end{cases}, \tag{6}$$

where $Str1_i$ denotes the ith character of $Str1$, and $Str2_j$ denotes the jth character of *Str2*.

4.2 Hamming Distance

The Hamming distance [49] was initially introduced as a measure to calculate the distance between two bit strings. However, it has been adapted to be used for strings. 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 the same size. For example, the distance between "ab" and "acdb" is three.

4.3 Manhattan Distance

The Manhattan distance [21] is normally used for vectors of numbers. It also can be applied to strings as

$$\text{Manhattan distance} = \sum_{i=1}^{n} |Str1_i - Str2_i|, \tag{7}$$

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.

4.4 Cartesian Distance

The Cartesian distance [21] is similar to the Manhattan distance. It can be applied to strings as

$$\text{Cartesian distance} = \sqrt{\sum_{i=1}^{n} (Str1_i - Str2_i)^2}. \tag{8}$$

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

4.5 Cosine Distance

The Cosine similarity [50] 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

$$\text{Cosine similarity} = \frac{\sum_{i=1}^{n} Str1_i \times Str2_i}{\sqrt{\sum_{i=1}^{n} Str1_i^2} \times \sqrt{\sum_{i=1}^{n} Str2_i^2}}. \tag{9}$$

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.

4.6 Locality-Sensitive Hashing (LSH)

LHS [26] 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 [26]. In this study, we used a type of random projection that is used to estimate cosine distances. This projection is defined as [51]

$$h^x(\vec{v}) = \begin{cases} 1 & \vec{x} \cdot \vec{v} \geq 0 \\ 0 & \vec{x} \cdot \vec{v} < 0 \end{cases}, \tag{10}$$

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)*.

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, and test case generation methods are investigated. The runtime order is analyzed based on the length of strings (L_1 and L_2), test set size (*TS*), population size in GA and MOGA (*N*), and number of potential candidates in ART methods (*K*). Table 1 provides the runtime order of all the algorithms. In the following, detailed discussions are presented.

The Hamming, Manhattan, Cartesian, and Cosine distance runtime complexity is linear against the length of the strings as can be observed from (7) and (9). Since each of these distance functions add null characters to the end of smaller string to make it same size with the longer string,

TABLE 1
Runtime Order Complexity of Each Algorithm Used in This Research

Algorithm	Runtime Order
String Distance Functions	
Levenshtein	$O_D = L_1 \times L_2$
Hamming	$O_D = Max(L_1, L_2)$
Manhattan	$O_D = Max(L_1, L_2)$
Cartesian	$O_D = Max(L_1, L_2)$
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 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 (6).

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 complexity improvement in GA and MOGA test case generation methods. The runtime order of NSGA-II is reported as $N^2 \times M$ [35] 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 $K \times TS^2$ [17], [19] 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.

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 versions of the software are produced. The p-measure [19] was selected to quantitatively measure the effectiveness of the test case generation methods. Finally, features of string test sets are discussed.

6.1 Software under Test (SUT)

To conduct a study on the fault-detection effectiveness of the test case generation methods, 19 real-world programs are investigated. We reused the programs from McMinn et al. [52] 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 Java projects which are widely used in GUI and web applications to validate strings. 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 [52].

Table 2 provides a description of each program. The "Name" column denotes a name used in the rest of this paper 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 2, is the summation of all classes in each program. It is different than LOC reported in the original work as only LOC of the main class is reported in the original work. For detailed description of each program, please refer to the original work [52].

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 [53], [54] are a well-known approach to automatically manipulate the source code and produce a large number of faults. There is considerable empirical evidence indicating a correlation between real faults and mutants [54], [55].

1. Originally, McMinn et al. [52] 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.

TABLE 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

In this study, muJava [56] is employed to produce mutated versions of the programs under the test where a total of 6,672 mutants are generated. Then, those mutants that were failed with the majority of test sets (more than 90 percent 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 [57]. As any execution of the system will reveal these defects, they will not live beyond a "debugging phase"; and hence, are not genuine testing objectives. Further, this approach is recommended in [53]. 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 demonstrates the number of generated and selected mutants per program.

TABLE 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

6.3 Testing Effectiveness Measure

Three well-known testing effectiveness measures exist, the e-measure, p-measure, and f-measure [19]. The E-measure is defined as the expected number of failures in a series of tests [19]. Given FR as the failure rate of the software under the test, the e-measure for RT can be estimated as $FR \times (test\text{-}set\ size)$ [58].

In the p-measure, the only important parameter is whether at least one failure is detected within the test set. Therefore, the p-measure is defined as the probability of detecting at least one failure in the test set. Hence, it can be estimated as the number of test sets where one or more failures are detected; over all the executed test sets. Further, in RT, the p-measure can be estimated by $1-(1-FR)^{testsetsize}$ [58]. Finally, the f-measure is defined as the minimum number of test cases required to detect the first failure. In RT, the expected value for the f-measure can be estimated by FR^{-1} [58].

In this research, we utilized the p-measure to evaluate the effectiveness of test case generation methods. According to the empirical knowledge that failures are often clustered within the input domain [11], [12], [59], detecting multiple failures are usually redundant as they often have a same failure root. In this regard, p-measure and f-measure are superior to e-measure. Further, the software testing process tends toward automation due to large number of test cases. In automated testing, test sets are automatically executed and the outputs are captured and stored. Then the output values are analyzed to detect failures. This description of the automated testing system which is in accordance with

many systems reported in studies [5], [60], [61], is well suited with p-measure or e-measure. The incremental procedure in the f-measure is not matched by the operation of these automated testing systems [19].

Further, the p-measure, unlike the e-measure and the f-measure, has a similarity with the mutation score. The mutation score is another well-known effectiveness measure that is the ratio of the number of failed mutants to the total number of mutants. The average of the p-measure is identical to the average of the mutation score when we test a program. To prove this, let's assume that we have a program with $M(i = 1 \dots M)$ mutants. We test all mutants with N test sets ($j = 1 \dots N$). Accordingly, mutation score for the jth test set (m_j) can be calculated by

$$m_j = \frac{1}{M} \sum_{i=1}^{M} r_{ij}, \tag{11}$$

where r_{ij} represents the result of applying jth test set to the ith mutant. If the test set fails (one or more test cases are failed), r_{ij} is 1; and 0 otherwise. Since we have N test sets, the average of the mutation scores for all test sets is

$$\overline{\text{mutation score}} = \frac{1}{N} \sum_{j=1}^{N} m_j = \frac{1}{NM} \sum_{j=1}^{N} \sum_{i=1}^{M} r_{ij}. \tag{12}$$

The p-measure is calculated for every mutant and is the probability of detecting at least one failure in the test set. Given the number of test sets as N, p-measure for the ith mutant is given by

$$\text{p-measure}_i = \frac{1}{N} \sum_{j=1}^{N} r_{ij}. \tag{13}$$

Since we have M mutants, the final p-measure (average of all p-measures) is

$$\overline{\text{p-measure}} = \frac{1}{M} \sum_{i=1}^{M} \text{p-measure}_i = \frac{1}{NM} \sum_{j=1}^{N} \sum_{i=1}^{M} r_{ij}. \tag{14}$$

The average p-measure and average mutation score calculations end up to a same results. Hence, the p-measure evaluation results in this work can be interpreted as the mutation score.

6.4 String Test Set Characterization

To evaluate the p-measure, we need a test set with a fixed size. In this research, 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 $10{,}000 \times (10 + 20 + 30) \times 5 = 3{,}000{,}000$ test cases that have been applied to each mutant. The difference in 10,000 generated test sets in each case is the seed to the pseudo-random number generator that changes in different runs.

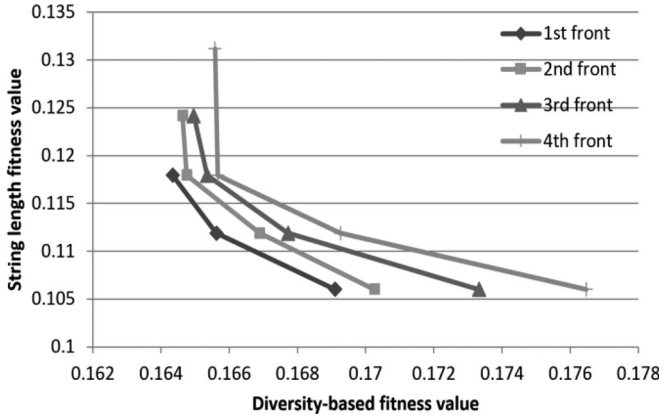

Fig. 3. The first few Pareto fronts in multi-objective optimization where maximum string size is 30, test set size is 10, and Levenshtein distance function is used.

Each test case is a string of characters. Therefore, we need to determine the range of characters to be used. Previous work commonly used printable ASCII characters [47], [62], [63]. Tonella [62] used only numbers, lower, and upper case characters. Alshraideh and Bottaci [47] used ASCII code from zero to 127; and Afshan et al. [63] used ASCII code from 32 to 126. We follow Afshan et al. [63] which includes all the printable ASCII characters.

Furthermore, 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. [63] used *StrMax* of 30 to generate strings in a white-box approach. Alshraideh and Bottaci [47] performed their experiments with *StrMax* of 20. In this study, we perform all of the experiments with two *StrMax* values (30 and 50) to explore any impact of the maximum string size. Finally, we implemented all the test generation algorithms in the research and hence, no predefined library is used.

7 Experimental Result and Discussion

This section presents the results of the empirical study. At first, Pareto-optimal test sets produced by MOGA are investigated. Following that, 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 7.4. Following that, statistical analysis of results is presented. In Section 7.4, the performance of different string distance functions is compared. Finally, an empirical runtime analysis is performed in Section 7.5.

7.1 Pareto-Optimal Test Sets

As discussed in Section 3.2, in a multi-objective optimization using NSGA-II, multiple front lines are generated. The first front (Pareto-optimal front) contains all the non-dominated test sets and hence, a Pareto-optimal set of test sets is produced rather than a single optimal test set. Fig. 3 represents the first few fronts in a single run of the

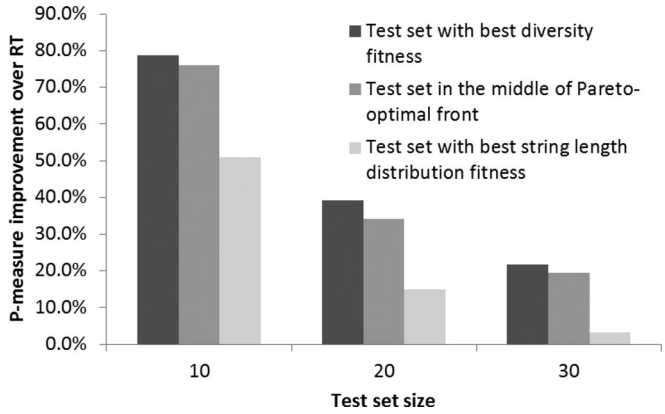

Fig. 4. Comparison of the failure detection effectiveness of test sets on the Pareto-optimal front where maximum string size is 30 and Levenshtein distance function is used. Every column is a percentage indicating the p-measure improvement against RT.

NSGA-II algorithm where the Levenshtein distance is used. In this figure, the Pareto-optimal front (first front) contains three points where each point represents a non-dominated test set.

In this work, the majority of executions of MOGA produced two to four test sets on the Pareto-optimal front. To select the best test set among the Pareto-optimal test sets, we performed an experiment where we compared the test sets in the middle and both ends of Pareto-optimal front with respect to the failure detection effectiveness. Fig. 4 demonstrates the comparison among the test sets where maximum string size is set to 30 and the Levenshtein distance function is used. Every column is a percentage indicating the p-measure improvement against RT which is averaged over all the programs under the test. 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 rest of the results that are presented for MOGA in this work, 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 achieves an approximate Benford distribution. Fig. 4 presents only the result for maximum string size of 30. For maximum string size of 50, result similar to Fig. 4 was produced and hence, it is not repeated.

7.2 Results of Each Program under Test

Tables 4 and 5 present the results for each program under test. Table 4 contains the results for $StrMax = 30$ and Table 5 includes the results of $StrMax = 50$. Every number in these tables is a percentage indicating the improvement of that method against RT as

$$\underset{(\%)}{\text{improvement}} = \frac{\text{p-measure(X)} - \text{p-measure(RT)}}{\text{p-measure(RT)}} \times 100, \quad (15)$$

where X denotes a test case generation method. Further, raw p-measure results for RT method are provided in Table 6 which allows the reader to compute the p-measure of each method if required.

According to Tables 4 and 5, 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.

In addition, the fitness score for testset diversity objective of the GA and the MOGA methods is presented in Table 7.

TABLE 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
	Mean	37.4	50.0	67.0	78.8
	Median	23.6	−9.1	36.9	33.8
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
	Mean	19.1	17.5	31.0	39.1
	Median	16.3	−6.5	25.1	27.5
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
	Mean	12.9	4.0	19.1	21.8
	Median	15.1	−5.0	12.6	17.1

TABLE 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
	Mean	63.7	83.6	115.1	128.6
	Median	37.8	32.1	73.0	77.1
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
	Mean	37.6	47.3	54.3	62.3
	Median	32.1	13.1	36.8	37.2
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
	Mean	25.0	27.3	37.6	41.3
	Median	22.1	4.4	29.0	39.0

A theoretical possibility exists that the results from the study could simply be due to the fact the MOGA algorithm is a superior optimization algorithm than GA. Obviously, optimization here is with regard to a single objective. Table 7 explores this conjecture. As can be seen from Table 7, the GA produces better fitness score for diversity compared to MOGA (larger diversity is better) and hence, the conjecture is refuted in every circumstance that is examined. Therefore, we are able to conclude that the improvement in performance of MOGA compared to GA is due to the construction of the multifactor formulization rather than the effectiveness of the MOGA then dealing with a single factor.

TABLE 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

7.3 Statistical Analysis of Results

The results in Tables 4 and 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 [19]. Chen et al. [58] empirically demonstrate that, when using random testing, the sampling distribution of the p-measure is normal. We formally explore this by investigating the normality of the results by performing Shapiro-Wilk test [64]; 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. This provides strong evidence that the sampling distribution is neither highly skewed nor heavy-tailed, both scenarios cause major problems in significance testing. Since both of these threats are minor; we simply choose a significance test which maximizes the statistical power of this situation.

TABLE 7
Diversity Fitness Score for GA and MOGA

	Testset Size	GA	MOGA
$StrMax = 30$	10	6.152	6.100
	20	10.944	10.879
	30	15.702	15.620
$StrMax = 50$	10	6.249	6.190
	20	11.014	10.969
	30	15.808	15.758

TABLE 8
The Effect Size between RT and Other Methods Where the Maximum String Size Is 30 and Levenshtein Distance Is Used

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*

"" indicates the result of the z-test where a significant difference exists at the 0.01 level.*

TABLE 9
The Effect Size between MOGA and Other Methods Where the Maximum String Size Is 30 and Levenshtein Distance Is Used

Testset Size	Software Under Test	RT	FSCS	ARTOO	GA
10	Validation	2.02*	1.67*	0.69*	0.23
	PostCode	17.15*	9.96*	1.87*	1.12*
	Numeric	20.83*	11.67*	2.70*	−1.81*
	DateFormat	20.93*	11.67*	3.15*	0.00
	MIMEType	0.54*	0.46*	1.17*	0.30*
	ResourceURL	0.46*	−0.08	0.33*	0.47*
	URI	−2.20*	0.18	0.21	0.88*
	URN	0.69*	2.18*	1.67*	0.36*
	TimeChecker	−2.94**	−0.48*	−0.38*	0.97*
	Clocale	17.93*	2.99*	2.74*	9.42*
	Isbn	1.87*	0.67*	5.47*	0.54*
	BIC	4.88*	1.48*	6.79*	0.31*
	IBAN	2.07**	−0.69*	2.86*	−0.76*
20	Validation	1.19*	1.43*	1.31*	0.11
	PostCode	13.07*	8.57*	1.38*	1.01*
	Numeric	13.92*	9.06*	−1.80*	−1.30*
	DateFormat	14.05*	9.12*	0.87*	0.00
	MIMEType	0.64*	0.07	1.16*	0.94*
	ResourceURL	0.00	−0.63*	−0.14	−0.04
	URI	−1.16*	0.41*	0.80*	2.10*
	URN	0.87*	1.48*	1.26*	0.48*
	TimeChecker	−1.17*	0.06	1.26*	1.85*
	Clocale	10.80*	−1.32*	−2.79*	6.26*
	Isbn	2.88*	−0.58*	8.25*	0.18
	BIC	5.37*	−1.93*	9.95*	0.90*
	IBAN	1.98*	−1.06*	3.16*	−1.07*
30	Validation	0.75*	0.06	1.04*	−0.10
	PostCode	10.22*	7.50*	1.93*	0.73*
	Numeric	10.16*	7.49*	−4.24*	−1.37*
	DateFormat	10.39*	7.74*	0.43*	0.00
	MIMEType	0.52*	0.70*	1.42*	0.44*
	ResourceURL	−0.21	−0.75*	0.10	−0.09
	URI	−1.16*	0.26*	1.12*	0.70*
	URN	1.05*	1.65*	3.08*	−0.14
	TimeChecker	−1.29*	−0.49*	0.91*	0.35*
	Clocale	6.34*	−3.23*	−3.86*	5.08*
	Isbn	2.30*	−1.53*	9.29*	0.98*
	BIC	4.97*	−2.75*	10.61*	1.23*
	IBAN	2.54*	−0.56*	4.75*	−0.77*

"" Indicates the result of the Z-test where a significant difference exists at the 0.01 level.*

Our working hypothesis is that MOGA, GA, FSCS, and ARTOO will produce superior results compared to RT. Further, an effect size (Cohen's method [65], [66]) between the each method and RT is calculated. The effect size quantizes the discrepancy between two statistical populations given by [19]

$$effect\ size = \frac{\mu_2 - \mu_1}{\sqrt{\frac{(n_2-1)std_2^2+(n_1-1)std_1^2}{n_2+n_1}}}, \tag{16}$$

where μ, std, and n represent the mean, the standard deviation, and the number of elements within the populations. Cohen [65] defines the standard value of an effect size as small (0.2), medium (0.5), and large (0.8).

Table 8 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 8 indicate that in most of the experiments MOGA statistically significantly outperforms RT.

Further, to formally investigate MOGA against other approaches, Table 9 is provided, it contains statistical significance test between MOGA and all other methods. This table indicates that in most cases MOGA produces superior results compared to other methods.

7.4 Results of Distance Functions

Figs. 5 and 6 represent the p-measure result for all six string distance functions that are discussed in Section 4. The results for $StrMax = 30$ and 50 are demonstrated in Figs. 5 and 6, respectively. In each of these figures, four graphs are

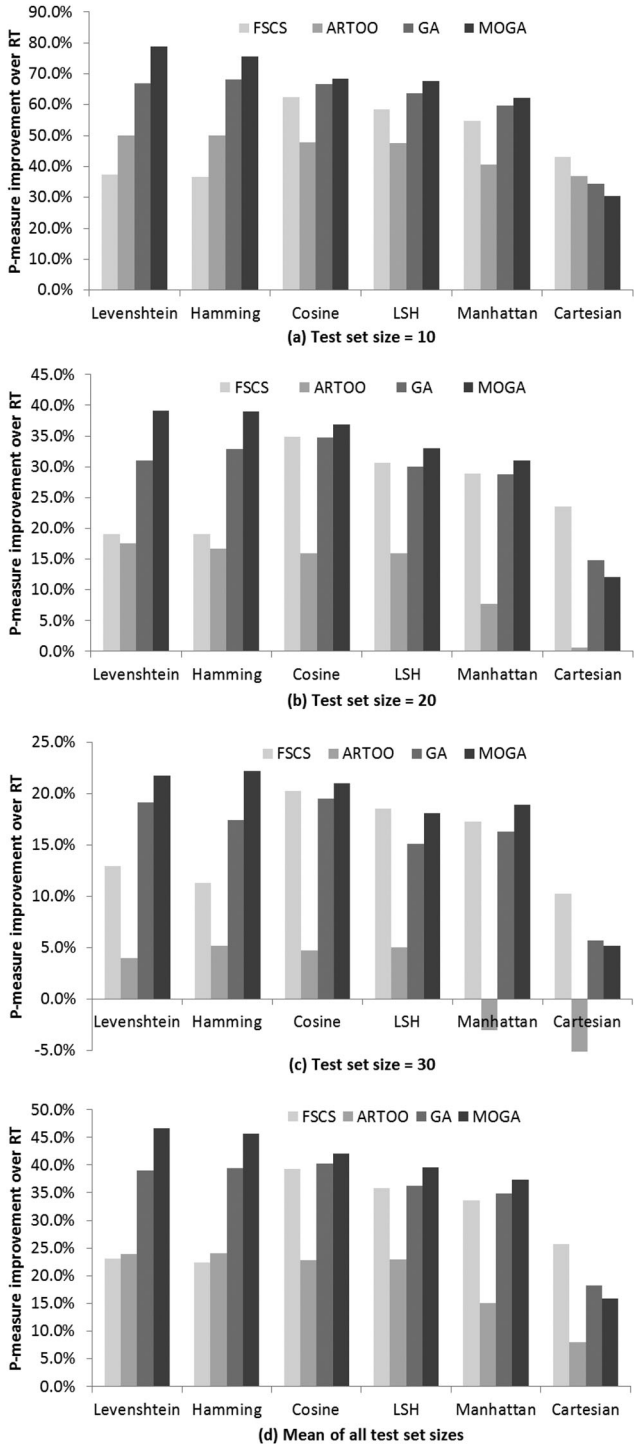

Fig. 5. Comparison of string distance functions where maximum string size is 30. Each column denotes p-measure improvement of each test case generation method over RT. (a), (b), and (c) represent results for test set sizes of 10, 20, and 30, respectively. (d) presents the mean of all test set sizes.

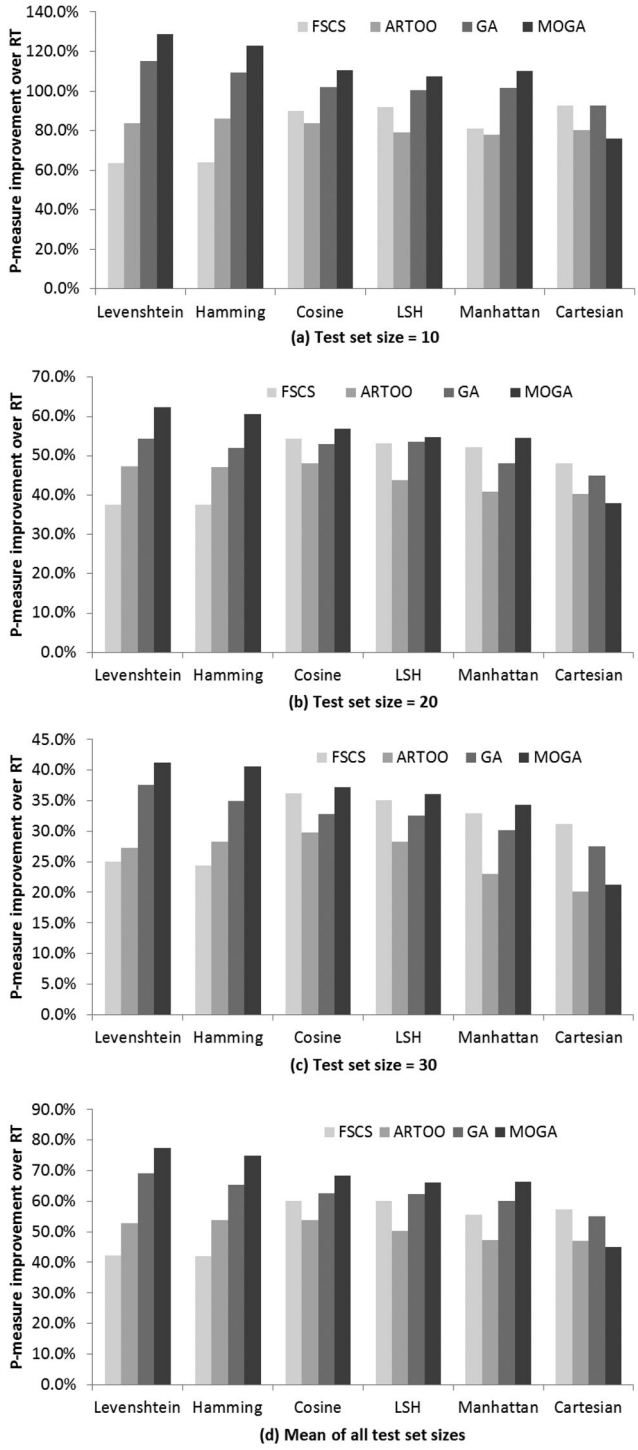

Fig. 6. Comparison of string distance functions where maximum string size is 50. Each column denotes p-measure improvement of each test case generation method over RT. (a), (b), and (c) represent results for test set sizes of 10, 20, and 30, respectively. (d) presents the mean of all test set sizes.

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.

According to these graphs, the MOGA test case generation method with the Levenshtein distance function has the superior failure detection effectiveness in most cases. 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 Figs. 5d and 6d. Comparing the FSCS and the ARTOO in Figs. 5d and 6d 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 Cartesian

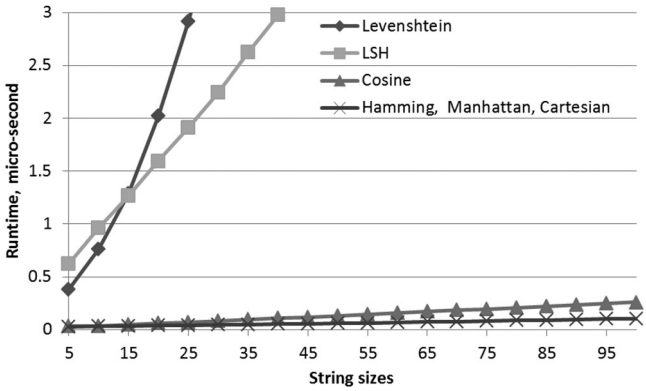

Fig. 7. Average execution time for different distance functions with string sizes between 5 and 100.

distance function has the lowest performance on average with respect to failure detection.

7.5 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 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.

Fig. 7 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 Cartesian distance functions are presented with a single line as they were very close. According to Fig. 7, 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 according to Table 1. The runtime result for LSH is the summation of both parts of the LSH calculation as explained in Section 5. According to Fig. 7, the LSH runtime is significantly higher than Cosine, Hamming, Manhattan, and Cartesian 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 5.

To demonstrate the runtime advantage of LSH compared to the other distance functions in string test generation, Fig. 8 is presented. Fig. 8a 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

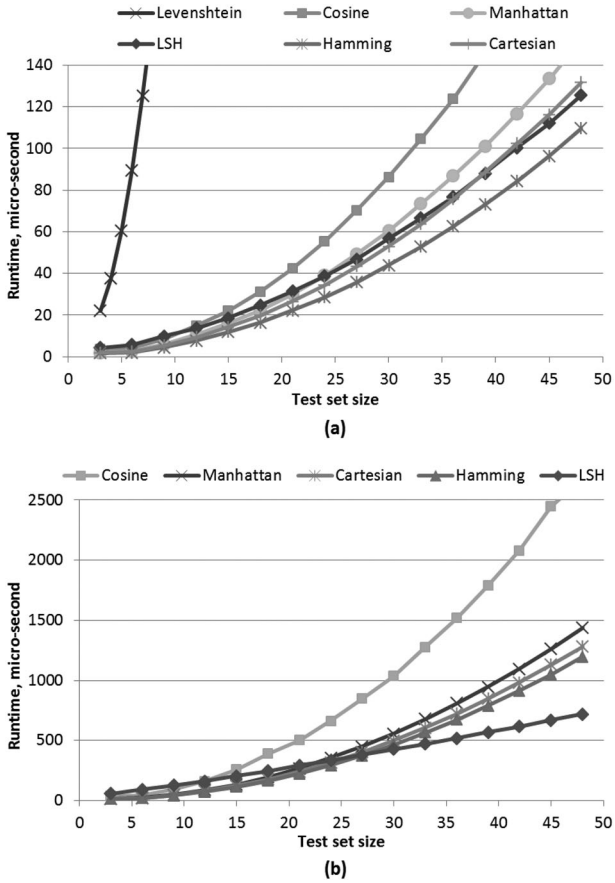

Fig. 8. Average execution time of diversity-based fitness function with test set sizes between 3 and 50. Random string sets with maximum string size of (a) 50 and (b) 1,000 are produced as input to the fitness function.

than the Manhattan (test set size larger than 25) and Cartesian distance function (test set size larger than 40). Further, LSH has lower runtime than the Hamming distance with test set size larger than 100 (Fig. 8a 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 Fig. 8a, 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, Fig. 8b is presented where the max string size is set to the relatively large number of 1,000. As demonstrated in Fig. 8b, the runtime of LSH is improved compared to other distance functions.

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 [19]. Accordingly, an important feature of a test case generation method is to generate random (uncorrelated) sequences of test cases. Two different aspects of randomness can be considered [19]:

- Randomness among test cases inside a test set is the first aspect. Correlated test cases within a test set can lead to systematic poor performance since a non-random set of test cases could significantly correlate

with a current set of defects. Hence, the chance of detecting those defects reduces significantly.

- The second aspect of randomness is the randomness between sequences of test sets. The test generation algorithm generates a test set in each run. If the generated test sets in different runs correlate, a test set is likely to be only capable of detecting defects that are already detected by the previous test sets. Similarly, if a test set does not detect any defect, the chance of detecting a defect with the next test set is low. This is critical in automated software testing. Because the tester may select to generate and execute a second set of test cases to detect new defects. Hence, randomness between two or more test sets is a critical feature of test generation method.

To quantitatively measure the randomness, Kolmogorov complexity [19], [67], [68] can be utilized. It can be used to measure the similarity between two sequences of data [19]. The Kolmogorov complexity of a piece of information ($\delta(data)$) is equal to the length of a perfect lossless compressed version of the information [68]. To calculate the randomness among test cases of a test set, a compression ratio (CR) is employed as

$$CR(T) = \frac{\delta(\varphi(T))}{sizeof(\varphi(T))}, \tag{17}$$

where T refers to a test set, and $\varphi(T)$ denotes serialized version of a test set. To produce $\varphi(T)$ simply all the string test cases are concatenated. For perfect randomness, $\varphi(T)$ cannot be compressed; and hence CR(T) would be one. Any value less than one denotes some degree of correlation among test cases.

The Normalized Compression Distance (NCD) [68] is utilized to investigate the randomness between test sets. It can be calculated by [68]

$$NCD(T_i, T_j) = \frac{\delta(\varphi(T_{ij})) - \min\{\delta(\varphi(T_i)), \delta(\varphi(T_j))\}}{\max\{\delta(\varphi(T_i)), \delta(\varphi(T_j))\}}, \tag{18}$$

where T_i and T_j denote two test sets. The $\varphi(T_{ij})$ is generated from the concatenation of $\varphi(T_i)$ and $\varphi(T_j)$. Identical T_i and T_j produces a NCD of zero. In contrast, $NCD(T_i, T_j) = 1$ denotes no correlation between T_i and T_j.

To calculate CR and NCD, we need a perfect lossless data compressor. However, a perfect compressor does not exist; and hence, we use the Lempel-Ziv-Markov chain Algorithm (LZMA) [69]. 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 near 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 extremely good 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 1.

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

9 Threats to Validity

In this section, potential threats to validity of this work are investigated. The stochastic nature of test generation algorithms that are investigated in this study is a potential threat to internal validity of this study. Statistical analysis of the results via a statistical test is a well-known technique to minimize this threat. Accordingly, a larger number of test sets are generated per test generation method. Then, a parametric test of statistical significance (z-test, one tailed with a conservative type I error of 0.01) is performed on the results. A parametric test is chosen since the p-measure results are proven to be normally distributed [58]. Our working hypothesis is that MOGA, GA, FSCS, and ARTOO will produce superior results compared to RT. The test results indicate that in most of the experiments, ART, GA, and MOGA methods statistically significantly outperforms RT.

Another source of potential threat to internal validity of this work is the mutant generation that is used in mutation analysis. A few tools are available to produce mutated version of source code in Java. Among those, we selected MuJava [56], since more mutation operators are implemented in MuJava compared to other tools. Further, it is previously used in several research works related to software testing [19], [53].

Furthermore, a common threat to external validity of any empirical study is the generalization of the results to other programs. Selection of the programs that are used in the empirical evaluations could be a potential source of bias. Due to the huge diversity of software programs (different programming languages, different sizes of programs, variety of programming styles, etc.), it is very challenging to construct an empirical evaluation study that captures everything. In this work, we tried to minimize this threat by selecting several programs for evaluations that are used in real-world projects. We reused the programs from McMinn et al. [52] and hence the selection of these programs can be viewed as being independent from the authors. This reduces the risk of bias in the utilized programs.

10 Related Work

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 [23] or symbolic execution [22], [70] 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 [23] 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. 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 [23]. In addition, Harman et al. [71] 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. Fraser et al. [72] integrate a memetic optimization algorithm with the EvoSuite tool [73] 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. work [72], 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. [63] 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. [52] and Shahbaz et al. [74] focus on the readability of string test cases. A method was proposed to query the web for common string types like emails. 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. Alshraideh and Bottaci [47] 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 [23], 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 [47]. 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 [47].

Symbolic execution [22], [70] 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 is a string constraint solver tool introduced by Ganesh et al. [22]. It accepts constraints in a specific format and finds values satisfying the constraints. It is used in many symbolic execution research projects. Ganesh et al. use Hampi in static and dynamic analysis to find SQL injection vulnerabilities. Saxena et al. [70] 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 [62] introduces 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 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). The next character is inserted with the probability of 0.5^{n+1} where n is the current length of the string. 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 are related researches 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 in the introduction, we present a brief review of these works for the sake of completeness. Hemmati et al. [20] 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 [20]. Several optimization algorithms including GA and hill climbing were tested. Ledru et al. [21] also employ string distance functions to prioritize string test cases. Multi-objective optimization is also used for test case selection. Yoo and Harman [75] used code coverage, past fault-detection history, and the execution cost as three optimization objectives.

11 Conclusions

In this research, 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 [11], [12], [13], [14], [15], 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 [46] is utilized to construct the fitness function. When both objectives are applied, 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, Cartesian, and LSH distance functions). Among the investigated distance functions, the LSH [26] is a fast estimation of the Cosine string distance function. According to the runtime complexity analysis in Section 5, LSH reduces the runtime complexity by an order of one. Further, in Section 5, the runtime complexities of all test case generation methods are discussed.

Thirteen real-world programs are used for an empirical study to evaluate the failure detection capability of the string test generation methods (RT, FSCS, ARTOO, GA, and MOGA). With respect to the evaluation results based on mutation analysis, on average, all the test generation methods produced better results than RT. This indicates that the diversity of test cases increase the failure detection effectiveness. This conclusion also can be drawn from the superior results of the GA and MOGA compared to the ART methods. Further, the MOGA produced, on average, a superior failure detection performance compared to GA. This suggests that applying the string length fitness function beside diversity improves the failure detection effectiveness of generated test cases. 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 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.

Although the results of this research improve black-box software testing effectiveness with respect to strings, there is still room for improvement. The fitness functions can be improved by an in-depth study on the factors that are important on string test cases. Accordingly, new fitness functions can be added into the multi-objective optimization discussed. Finally, further studies can be performed on other test case types like trees or graphs. An example can be programs that take trees as input like an xml parser. Research on the tree structure is being undertaken by the authors of this work where the proposed tree distance function in our previous study [76] will be used.

REFERENCES

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

[2] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, Jul. 1984.

[3] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Code coverage of adaptive random testing," *IEEE Trans. Reliab.*, vol. 62, no. 1, pp. 226–237, Mar. 2013.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[5] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .NET with feedback-directed random testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 87–96.

[6] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *Proc. 2nd Int. Workshop Random Testing: Co-Located 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, p. 1.

[7] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile security testing of web-based systems via HTTPUnit," in *Proc. Agile Conf.*, 2005, pp. 29–38.

[8] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *Proc. 3rd Int. Conf. Quality Softw.*, 2003, pp. 20–23.

[9] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proc. 4th Conf. USENIX Windows Syst. Symp.*, 2000, pp. 59–68.

[10] S. Lipner and M. Howard, "The trustworthy computing security development lifecycle document (SDL)," in *Proc. 20th Annu. Comput. Security Appl. Conf.*, 2005, pp. 2–3.

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

[12] G. B. Finelli, "NASA software failure characterization experiments," *Reliab. Eng. Syst. Saf.*, vol. 32, nos. 1/2, pp. 155–169, 1991.

[13] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 3, pp. 247–257, May 1980.

[14] 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 *Proc. 23rd Int. Symp. Fault-Tolerant Comput. Dig. Papers.*, 1993, pp. 98–107.

[15] C. Schneckenburger and J. Mayer, "Towards the determination of typical failure patterns," in *Proc. 4th Int. Workshop Softw. Quality Assurance: In Conjunction 6th ESEC/FSE Joint Meeting*, 2007, pp. 90–93.

[16] 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.

[17] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Trans. Reliab.*, vol. 58, no. 4, pp. 619–633, Dec. 2009.

[18] J. Lv, H. Hu, K.-Y. Cai, and T. Y. Chen, "Adaptive and random partition software testing," *IEEE Trans. Syst., Man, Cybern.*, vol. 44, no. 12, p. 1649–1664, Dec. 2014.

[19] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal Voronoi tessellations—a new approach to random testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 163–183, Feb. 2013.

[20] 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.

[21] 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.

[22] 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 *Proc. 23rd Int. Conf. Comput. Aided Verification*, 2011, pp. 1–19.

[23] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, Mar./Apr. 2010.

[24] A. B. Downey, "The structural cause of file size distributions," in *Proc. 9th Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, 2001, pp. 361–370.

[25] M. Silfverberg, I. S. MacKenzie, and P. Korhonen, "Predicting text entry speed on mobile phones," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2000, pp. 9–16.

[26] 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.

[27] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proc. 9th Asian Comput. Sci. Conf.: Adv. Comput. Sci.*, 2005, pp. 3156–3157.

[28] K. Chan, T. Chen, and D. Towey, "Restricted random testing," in *Proc. 7th Eur. Conf. Softw. Quality*, 2002, pp. 321–330.

[29] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive Random Testing for Object-Oriented Software," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 71–80.

[30] J. Mayer and C. Schneckenburger, "An empirical analysis and comparison of random testing techniques," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, 2006, pp. 105–114.

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

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

[33] 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," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 742–762, Nov./Dec. 2010.

[34] M. Harman, "The current state and future of search based software engineering," in *Proc. Future Softw. Eng.*, 2007, pp. 342–357.

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

[36] K. A. De Jong and W. M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," in *Proc. 1st Workshop Parallel Problem Solving Nature*, 1991, pp. 38–47.

[37] 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.
[38] T. P. Hill, "The Significant-Digit Phenomenon," *Am. Math. Mon.*, vol. 102, no. 4, pp. 322–327, 1995.
[39] F. Benford, "The Law of Anomalous Numbers," *Proc. Am. Philos. Soc.*, vol. 78, no. 4, pp. 551–572, 1938.
[40] 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.
[41] 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.
[42] 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.
[43] R. A. Raimi, "The first digit problem," *Am. Math. Mon.*, vol. 83, no. 7, pp. 521–538, 1976.
[44] A. Berger, T. P. Hill, et al., "A basic theory of Benford's Law," *Probab. Surv.*, vol. 8, pp. 1–126, 2011.
[45] J. J. Baroudi and W. J. Orlikowski, "The problem of statistical power in MIS research," *MIS Quart.*, vol. 13, pp. 87–106, 1989.
[46] M. A. Stephens, "Use of the Kolmogorov-Smirnov, Cramer-Von Mises and related statistics without extensive tables," *J. Roy. Stat. Soc. Ser. B*, vol. 32, pp. 115–122, 1970.
[47] 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.
[48] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, 1966, vol. 10, no. 8, pp. 707–710.
[49] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950.
[50] D. C. Anastasiu and G. Karypis, "L2AP: Fast cosine similarity search with prefix L-2 norm bounds," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 784–795.
[51] G. Xue, Y. Jiang, Y. You, and M. Li, "A topology-aware hierarchical structured overlay network based on locality sensitive hashing scheme," in *Proc. 2nd Workshop Use P2P, GRID Agents Develop. Content Netw.*, 2007, pp. 3–8.
[52] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *Proc. IEEE 5th Int. Conf. Softw. Testing, Verification Validation*, 2012, pp. 141–150.
[53] A. Arcuri and L. Briand, "Adaptive random testing: an illusion of effectiveness?" in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 265–275.
[54] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.
[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 *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 654–665.
[56] 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.
[57] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, "Mutation analysis," Dept. Comput. Sci., Yale Univ., New Haven, CT, USA, 1979.
[58] 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.
[59] 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.
[60] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 75–84.
[61] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005.
[62] P. Tonella, "Evolutionary testing of classes," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2004, pp. 119–128.
[63] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proc. IEEE 6th Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 352–361.
[64] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, nos. 3/4, pp. 591–611, 1965.
[65] J. Cohen, "A power primer," *Psychol. Bull.*, vol. 112, no. 1, pp. 155–159, 1992.
[66] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Mahwah, NJ, USA: Lawrence Erlbaum, 1988.
[67] M. Li and P. M. B. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications*. New York, NY, USA: Springer-Verlag, 2008.
[68] M. Li, X. Chen, X. Li, B. Ma, and P. M. B. Vitanyi, "The similarity metric," *IEEE Trans. Inf. Theory*, vol. 50, no. 12, pp. 3250–3264, Dec. 2004.
[69] K. G. Morse Jr, "Compression tools compared," *Linux J.*, vol. 2005, no. 137, pp. 62–66, 2005.
[70] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for Javascript," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 513–528.
[71] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proc. 9th Annu. Conf. Genetic Evol. Comput.*, 2007, pp. 1098–1105.
[72] G. Fraser, A. Arcuri, and P. McMinn, "Test suite generation memetic algorithms," in *Proc. 15th Annu. Conf. Genetic Evol. Comput.*, 2013, pp. 1437–1444.
[73] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.
[74] 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 *Proc. 12th Int. Conf. Quality Softw.*, 2012, pp. 79–88.
[75] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 140–150.
[76] A. Shahbazi and J. Miller, "Extended subtree: A new similarity function for tree structured data," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 4, pp. 864–877, Apr. 2014.

Ali Shahbazi received the BS and MS degrees in electrical engineering from Amirkabir University of Technology (Tehran Polytechnic), Iran, in 2007 and 2010, respectively. He received the PhD degree in software engineering and intelligent systems from the University of Alberta, Canada, in 2015, where he was a member of STEAM Lab working on software testing and machine learning. His research interests include data analysis, software testing approaches in web-based systems, embedded systems, and mobile devices as well as security of the web-based systems. He is a student member of the IEEE.

James Miller received the BSc and PhD degrees in computer science from the University of Strathclyde, Scotland. During this period, he was on the ESPRIT project GENEDIS on the production of a real-time stereovision system. Subsequently, he worked at the United Kingdom's National Electronic Research Initiative on Pattern Recognition as a principal scientist, before returning to the University of Strathclyde to accept a lectureship, and subsequently a senior lectureship in Computer Science. Initially, during this period his research interests were in computer vision, and he was a co-investigator on the ESPRIT 2 project VIDIMUS. Since 1993, his research interests have been in web, software, and systems engineering. In 2000, he joined the Department of Electrical and Computer Engineering, University of Alberta, Canada, as a full professor and in 2003 became an adjunct professor at the Department of Electrical and Computer Engineering, University of Calgary. He has published over 100 refereed journal papers on web, software, and systems engineering (see www.steam.ualberta.ca for details on recent directions). He recently served as the 2011 Organizing chair for the IEEE International Symposium on Empirical Software Engineering and Measurement; and sat on the editorial board of the *Journal of Empirical Software Engineering* for more than a decade. He is a member of the IEEE.