Autom Softw Eng (2012) 19:65–95
DOI 10.1007/s10515-011-0093-0

Prioritizing test cases with string distances

Yves Ledru · Alexandre Petrenko · Sergiy Boroday · Nadine Mandran

Received: 11 November 2010 / Accepted: 14 August 2011 / Published online: 7 September 2011

Abstract Test case prioritisation aims at finding an ordering which enhances a certain property of an ordered test suite. Traditional techniques rely on the availability of code or a specification of the program under test. We propose to use string distances on the text of test cases for their comparison and elaborate a prioritisation algorithm. Such a prioritisation does not require code or a specification and can be useful for initial testing and in cases when code is difficult to instrument. In this paper, we also report on experiments performed on the "Siemens Test Suite", where the proposed prioritisation technique was compared with random permutations and four classical string distance metrics were evaluated. The obtained results, confirmed by a statistical analysis, indicate that prioritisation based on string distances is more efficient in finding defects than random ordering of the test suite: the test suites prioritized using string distances are more efficient in detecting the strongest mutants, and, on average, have a better APFD than randomly ordered test suites. The results suggest that string distances can be used for prioritisation purposes, and Manhattan distance could be the best choice.

Keywords Software engineering · Testing and debugging · Testing tools · Test case prioritisation · String distance

Y. Ledru (✉) · N. Mandran
Laboratoire d'Informatique de Grenoble (LIG), UMR 5217, UJF-Grenoble 1, Grenoble-INP, UPMF-Grenoble 2, CNRS, 38041 Grenoble, France
e-mail: Yves.Ledru@imag.fr

A. Petrenko · S. Boroday
Centre de Recherche Informatique de Montréal, 405 Ogilvy Avenue, Suite 101 Montréal, Québec H3N 1M3, Canada

A. Petrenko
e-mail: Alexandre.Petrenko@crim.ca

1 Introduction

For industrial size software applications, test suites usually include thousands of test cases, and their execution can take several hours or even days. As an example, consider a test suite for the Jonas J2EE middleware with 2689 test cases (Kessis et al. 2005). When applied to all 16 configurations of the middleware, it results in running more than 43,000 test cases. The cost associated with a large test suite is essential and to reduce it one has to achieve testing goals as early as possible, e.g., using a smallest possible part of the test suite.

Test suite reduction (Harrold et al. 1993) is a technique for selecting a subset of a given test suite, which shares a common characteristic with the full test suite. If this characteristic is related, e.g., to code coverage and thus to fault detection power then the same defects could be detected using fewer resources. The main risk of test suite reduction is to lose useful test cases. Several experiments have been reported in the literature, and they come to diverging conclusions. In Rothermel et al. (1998), experiments indicate that test suite reduction may compromise fault detection. But Wong in Wong et al. (1999) uses different case studies and does not notice significant fault detection loss in reduced test suites. More recently, Heimdahl stated significant fault-finding loss in other experiments (Heimdahl and Devaraj 2007).

Test case prioritisation (Rothermel et al. 2001) has similar motivations. It aims at finding an ordering of test cases that maximizes the value of a fitness function which reflects a certain testing goal, e.g., the number of detected bugs or code coverage. Compared to test suite reduction, test case prioritisation appears to be safer, at least from the fault detection viewpoint, since it does not discard test cases and simply permutes them.

Both, reduction and prioritisation, techniques must be able to differentiate test cases. If two test cases are found to be similar in some sense, one of them should be discarded reducing a test suite or receive a lower priority when test cases are prioritized. Indeed, once one of them has been executed the chances that the second will, e.g., reveal new errors may not be high. Several existing reduction and prioritisation techniques are based on the availability of the code of a program under test. Execution of the program under test is required to gather information, such as code coverage or execution time, which is then used to remove or prioritize test cases.

In the approach proposed in this paper, the differentiation of test cases relies on a sole basis of the text of the test cases, without exploiting knowledge about the program under test. The text of the test cases corresponds either to the input data, i.e. the values of the input parameters, or to the source code of the test cases, e.g. in JUnit test cases. The approach addresses a wide class of applications for which test cases can be treated as character strings. In fact, in many situations, it is easy to transform test inputs into strings, as they are just a vector of values assigned to input parameters. Such inputs have a textual representation which corresponds to a (possibly long) string of characters. In the case of, e.g., JUnit test suites, test cases take the form of a sequence of instructions encapsulated into a procedure or method. Similarly, the text of such a procedure can be viewed as a string of characters. The basic idea of prioritizing test cases is to assign high priority to a test case which is most different compared to those already prioritized. The diversity of test cases represented as strings is assessed using the existing string distance metrics. To the best of our knowledge, such an approach has not yet been experimentally assessed for the purpose of test suite prioritisation.

This approach has several advantages, since the differentiation of test cases neither relies on test execution nor requires a reference model or program under test. In this sense the proposed approach is "minimal": the only required information is already encoded in the test suite.

The proposed approach relies on the hypothesis that test diversity enhances the fault detection capability of tests and thus a string distance is a reasonable measure of their diversity. If this is the case then a prioritisation algorithm which orders tests according to the distances would increase the ability of the test suite to rapidly find defects. A close approach is suggested in the area of protocol testing (Kovács et al. 2009). Unlike (Kovács et al. 2009), our metric is defined on inputs only, the expected outputs are ignored. We choose not to consider outputs in our metric because they are not always included in the test suite and their computation often requires executing the test suite. This goes beyond our minimal approach relying only on the text of the test suite. Nevertheless, in cases when the outputs are included in the text of the test cases, e.g. using "Assert" statements in JUnit, our approach will take them into account.

Since validation of the above hypothesis seems crucial for the proposed approach, we report on experiments which compared the defect detection capabilities of test suites prioritized on the basis of a string distance with random permutations. The experiments used the Siemens programs (also known as "the Siemens Test Suite") (Hutchins et al. 1994) which constitute a standard benchmark for a prioritisation algorithm. Since our algorithm can be parameterized by a chosen string distance, we evaluated the algorithm using four standard distances. In particular, we present a rigorous experimental evaluation which includes statistical analyses (ANOVA analysis and Tukey's test) for assessing various experimental conditions. A preliminary report on this work can be found in Ledru et al. (2009).

The remainder of the paper is organized as follows. Section 2 summarizes the related work on prioritisation techniques, as well as on anti-random and adaptive random testing which influenced this work. Section 3 presents four classical string distances and discusses their application in test case comparison. In Sect. 4, a test case prioritisation algorithm employing these distances is presented and illustrated on a small example. Section 5 reports on our experimental comparison of the proposed algorithm with a random prioritisation method using the Siemens Test Suite. Finally, Sect. 6 draws the conclusions of this work.

2 Related work

In Rothermel et al. (2001), the *prioritisation problem* is defined as follows:

Given T, a test suite,
PT, the set of permutations P of T,
f, a fitness function from PT to the real numbers,

$$\textit{Find} \quad P \in PT \text{ such that}$$
$$\forall P' \in PT \mid P' \neq P \cdot f(P) \geq f(P')$$

In other words, the goal is to find a permutation of the original test suite which maximizes a fitness function f. Prioritization is often considered in the context of regression testing, when measurements of, e.g., code coverage can be performed on an executable version of the program under test. It also relies on the assumption that tests may be executed in any order, which allows to permute them.

Several prioritisation algorithms have been proposed, mainly based on code coverage information (Wong et al. 1997; Rothermel et al. 2001; Elbaum et al. 2002; Srivastava and Thiagarajan 2002; Jones and Harrold 2003; Li et al. 2007) or other run-time information such as call trees (Smith et al. 2007). Several works rely on formal specifications (Vuong and Alilovic-Curgus 1992; Kovács et al. 2009; Feijs et al. 2002), unfortunately such specifications are rarely available, except perhaps for safety critical and telecommunication domains. In Kovács et al. (2009) SDL specifications are used mainly to extract names of signals; however experiments are limited to two simple protocols. Recent work (Walcott et al. 2006) also takes into account the execution time of each test case. Although code-based prioritisation remains mainstream, Srikanth and Williams (Srikanth et al. 2005) rely only on the traceability between test cases and requirements. Each requirement is evaluated based on its risk to correspond to severe defects and its value for the customer; tests are prioritized on their coverage of the most valuable or the most risky requirements. Prioritization can also be applied to order the configurations in which test suites will be run (Qu et al. 2008). In a similar way as for test suites, prioritisation of configurations can be based either on run-time information or on links with specifications.

Test suite reduction or minimization is a related problem where one wants to select a subset of the original test suite with similar properties as the full test suite. Although many works rely on the availability of code coverage information (Harrold et al. 1993; Jones and Harrold 2003), some other works do not require a code; they are based on coverage of a specification of the program under test (Heimdahl and Devaraj 2007), or coverage of the input syntax (Hennessy and Power 2005).

Unlike these works, our approach does not rely on availability of the code, its specification or reference model, or even an input grammar. One may expect our approach to be more appropriate for initial testing and when code-based prioritisation is not possible, especially when instrumentation of the code is too costly or impossible, e.g., in case of testing distributed web applications.

A similar objective is pursued by H. Hemmati and his colleagues (Hemmati et al. 2010) who propose a test suite reduction algorithm based on the similarity of test cases. Their study concludes that “Test cases that find the same faults tend to be more similar to each other than with other test cases”, and that “test cases that find different faults tend to be more different from each other than test cases that find the same faults”. Unlike our work, they rely on a specific form of input data expressed as sequences of guards or triggers. In this paper we propose an alternate and more general way of measuring test case diversity, using standard string distances.

This work has also been influenced by the ideas of “anti-random testing” (Malaiya 1995; Yin et al. 1997) and “adaptive random testing” (Chen et al. 2004). The anti-random testing approach (Malaiya 1995; Yin et al. 1997) aims at building a test suite by constructing new test cases which are the most distant possible from existing ones. The idea of adaptive random testing comes from the work of Chen et al. (2004), based on observations of Chan et al. (1996) that errors in software are usually clustered and follow several "failure patterns". Based on these observations Chen proposes several improvements in random generation techniques. The basic idea is that if some randomly generated test cases did not reveal an error, the next test case should not be too close to any of the applied test cases. Actually, since errors are clustered, each test case can be considered as a representative of its clustered neighborhood. Based on this initial idea, a variety of techniques have been proposed (Mayer and Schneckenburger 2006).

Adaptive random testing mainly applies to numerical programs, where input is usually an n-dimensional vector of real numbers. It is difficult to adapt this approach to non-numerical software, where input is less uniform than real numbers, and a distance between test cases is more difficult to determine. Our approach addresses these problems, first, by using string distances to compare test inputs. These test inputs can be numerical or not, or even mix numbers and non-numbers. Both numbers and non-numbers are treated as text. Then, unlike anti-random and adaptive random testing, which are generative approaches, it starts with an existing test suite and selects from it test cases, one by one, which are most distant from the already selected ones. Since inputs are available, we do not face the problem of constructing data satisfying some distance constraints.

Anti-random strategies are applied in Jiang et al. (2009) to coverage based prioritisation. Namely these results show that a pure coverage based prioritisation technique is comparable with prioritisation techniques based on the Jaccard string distance on the execution path, though for larger programs the level of detail of code coverage is less important. Also this work evaluates several distances from a test case to a set of test cases, based on the average, minimal, and maximal distance between the former test case and the elements of the set. As a result, it recommends the usage of minimal distance, and we use it in this paper. Unlike Jiang et al. (2009), Ramanathan et al. (2008) evaluate two different approaches to prioritisation, a greedy and spectral one and finds these approaches comparable on average, though in a few cases, a more sophisticated spectral algorithm produces noticeably better results. Differently from Ramanathan et al. (2008), Jiang et al. (2009), where only one string distance is used, we evaluate four string distances, namely Cartesian and Manhattan distances along with simple edit Hamming and Levenshtein distances.

3 Distances

3.1 Hamming distance

Hamming distance (Hamming 1950) is originally proposed to compare two sequences of bits of equal length. It computes the number of bits different in the two sequences. For example, "1010" and "1001" are at a distance of 2, because two bits differ. It can also be adapted to sequences of characters of the same length: "abcd" and "abab" differ in the last two characters and have a distance of 2. Hamming distance is easy to compute by a single pass on both sequences.

In order to apply this distance to arbitrary test inputs, they should have the same length. A simple way of achieving this is to complete the smallest string by a suffix where each character is different from the corresponding character in the larger string. For example, the distance between “ab” and “abcde” is computed by first completing the smallest vector “abxxx” (where ‘x’ stands for `char(0)`) and then measuring Hamming distance, which becomes 3.

3.2 Levenshtein or Edit distance

Hamming distance is not robust to insertions of characters within a string. For example, the distance between “Y. Ledru” and “Yves Ledru” is 9, because the additional characters of the first name shift the last name in the string. Levenshtein introduces a distance that computes the smallest number of edit operations needed to transform one string into another (Levenshtein 1965, 1966). Edit operations correspond to replacement, insertion or deletion of characters. In our example, the “.” has been replaced by “v” and “es” have been inserted. This gives a distance of 3 between those strings.

Levenshtein distance applies to strings of different lengths. It is widely used in text processing applications such as spell-checkers. Levenshtein also proposes an algorithm to compute the distance, whose cost, both in time and memory, is $O(m * n)$ where m and n are the lengths of the two strings.

3.3 Cartesian and Manhattan distances

A string of size n can be seen as a vector of characters in an n-dimensional space, and characters can be associated to their ASCII code (or any other numerical coding). It is thus possible to compute the classical distances in an n-dimensional space, like Cartesian and Manhattan distances:

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \quad \text{Cartesian or Euclidean distance}$$

$$\sum_{i=1}^{n}|x_i - y_i| \quad \text{Manhattan distance}$$

where x and y are the strings to compare. Similar to Hamming distance, when strings have different lengths, the shortest one is filled with `char(0)`.

For example the distances between “ab” and “cde” are[1]:

$$\sqrt{(99-97)^2 + (100-98)^2 + (101-0)^2} = \sqrt{10209} = 101,04 \quad \text{(Cartesian)}$$

$$|99-97| + |100-98| + |101-0| = 105 \quad \text{(Manhattan)}$$

This example shows that the cost of insertion is rather high with these distances, e.g., each additional lowercase adds at least the offset of ‘a’, i.e., 97, to the distance.

[1] ASCII codes for a, b, c, d, e are 97, 98, 99, 100, 101 respectively.

3.4 Caveats associated with string distances

String distances are based only on lexicographic information and do not capture the semantics of the test cases. As a result, two test cases may be considered as distant although they are not. Let us consider several simple examples.

The distance (Hamming or Edit) between "999" and "1000" is 4, while the distance between "1000" and "9000" is only 1. This shows that string distances do not necessarily correspond to numerical distances. As a result, prioritisation of test cases may sometimes appear counter-intuitive.

Given a sort algorithm, the distance (Hamming or Edit) between "sort("1 2 3 4 5")" and "sort("20 40 60 80 100")" is 11 (Edit) or 16 (Hamming). But both test cases correspond to the same execution path, i.e. they sort a sequence of 5 elements given in an ascending order. So they are semantically close, but this fact is ignored by a string distance.

It is also often the case that two test cases, aimed at exercising very different features of the system under test, have very close texts, as in the following examples. Test cases for a square root program may well include "sqrt(1)" and "sqrt(-1)" which correspond to completely different execution paths (the latter should raise an exception), but have a small Edit distance, 1.

Often a number of test cases have a common prefix, that leads a system into a given state and then they exercise various features in that state. For example, the test of an ATM may include test cases like "insert_card(); type_correct_pin(); debit(100)" and "insert_card(); type_ correct_pin(); view_balance()" which differ only in the last method call. Their Edit distance is 11 which may appear small compared to their string lengths of 45 and 49.

The above examples indicate that string distances should be used with caution, as for some classes of applications they may lead to unexpected conclusions. Nevertheless, they may well turn out to be useful for other types of applications. For example, all distances identify two inputs differing only by one character and strings of different lengths. Also, Edit distance can find common substrings in two inputs. When source code of test cases is considered, Edit distance detects common sequences of instructions, and all distances can detect common prefixes.

In the absence of decisive empirical data, it is important to perform experiments to assess the applicability of string distances for test prioritisation and minimization. In this paper, we report results which shed some light on this issue.

4 Prioritization technique

4.1 Fitness function and algorithm

We propose a fitness function which is based on a distance between each test case and the set of the preceding test cases in the test suite.

We generalize the notion of distance between two strings to the distance between a test case t and a set of test cases T'. Following (Jiang et al. 2009)'s conclusion that minimal distance to the set is the most efficient, we define this distance measure as the distance between t and the *closest* element of set T'. The distance between a test case t and a set of test cases T' uses distance measure d between two test cases and is defined as follows:

$$dd(t, T') = \min\{d(t, t_i) | t_i \in T' \text{ and } t_i \neq t\}$$

Our fitness function is then defined as the sum of distances between each test case and the set of preceding test cases in a given permutation $P = t_1, \dots, t_n$ of the test suite T:

$$f(P) = \sum_{i=1}^{n} dd(t_i, T_{i-1}) \quad \text{where}$$
$$T_{i-1} = T \quad \text{for } i = 1 \quad \text{and}$$
$$T_{i-1} = \{t_1, \dots, t_{i-1}\} \quad \text{for } i \geq 2$$

Rather than aiming at finding a permutation of test cases which corresponds to a maximum value of the fitness function, we opt for a greedy algorithm, which, at each iteration, chooses a test case which is the most distant from the set of already ordered test cases. When several elements are equidistant from this set, any of them may be chosen. In the proposed algorithm the first element occurring in a given test suite is chosen. One could also choose randomly between the equidistant elements.

Greedy algorithms are prone to the local maximum problem often resulting in a suboptimal solution; however the exact solution is often computationally difficult. The problem of maximizing the above fitness function (also known as a Minimum Similarity Ordering problem) is NP-hard (Ramanathan et al. 2008), since its special case is a well-known NP-hard graph problem, the Minimum Linear Arrangement Problem (MinLA). MinLA is approximately within O(log n) and even within somewhat better (Charikar et al. 2006), however, the known approximation algorithms still do not scale well. Thus, in practice the problem is usually solved using heuristic search techniques, such as greedy, spectral, and simulated annealing, of which the latter scales worst, but yields the best results. One of the simplest heuristic algorithms is a greedy algorithm. The greedy algorithms are known to produce a good solution on some problems, but not on others. For the purpose of prioritisation the greedy algorithms perform quite well, on average on par with more sophisticated and computationally expensive spectral algorithms, though in "the worst-case" situations (when a local maximum significantly differs from a global one) they might result in a significantly inferior solution (Ramanathan et al. 2008).

Given a test suite T, the following algorithm computes a prioritisation as a sequence P.

```
Compute the distances for each pair of test cases in $T$
Remove duplicates from $T$
Find an element $t \in T$ with the maximum distance $dd(t, T)$,
  $T := T \setminus \{t\}$,  $P := t$
While $T$ is not empty
  Find an element $t \in T$ with the maximum distance $dd(t, P)$,
    $T := T \setminus \{t\}$,  $P := P.t$  ($t$ is appended to the sequence)
Append duplicates to $P$
return $P$
```

As mentioned above, we consider a test suite as a *set* of test cases. Actually, it is often a *bag* (multiset), i.e., some tests are duplicated. This may occur due to errors in the generation process: the tool or the tester came twice to the same result. But it may also be intentional: replicating a test case gives it a bigger weight in the test suite and can, for example, increase the chance of choosing it in a random selection process. Our algorithm identifies a duplicate by its zero distance to a preceding test case in the suite, and removes it from the test suite. Once the test suite has been prioritized, duplicates are appended to the resulting test suite to obtain a permutation of the original test suite.

The most expensive operation of the algorithm is the computation of distances between all test cases which are then used to choose a test case most distant from already ordered test cases. The complexity of the algorithm is $O(n^2)$ where n is the size of the test suite. We have implemented this algorithm in Java for each of the four distances mentioned in Sect. 3. The experimental results in Sect. 5.4 indicate that its execution on the largest suite which includes 5542 test cases takes only 16 to 26 seconds, for Hamming, Manhattan and Cartesian distances, but 38 minutes for Edit Distance.

4.2 Example

To illustrate the proposed algorithm, we use an implementation of the Caesar cipher. The Caesar cipher encodes a message by shifting its letters by a fixed number of characters. For example "abc" with a shift of 3 becomes "def". In our implementation, we only apply the cipher to the 26 lowercase and uppercase letters of the alphabet. The remaining characters keep their initial value. Moreover, the shift is performed circularly over the lowercase and uppercase letters, so that each letter remains a letter in the encrypted message (e.g. see test t3 in Table 1). Table 1 gives the 7 test cases of our test suite and the expected outputs.

The first step of the algorithm computes all the distances between test cases. The two parameters (input string and shift) are concatenated, separated by a white space, into a single string before computing the distances between these strings. Table 2 gives a table for each of the distances (Cartesian, Edit, Hamming, Manhattan). For example Hamming distance between t1 ("abc 1") and t2 ("abc.xyz -1") is equal to 7, because the last seven characters of t2 differ from the corresponding characters in t1.

Table 1 The test suite for the Caesar cipher

Test case	Input string	Shift	Output string
t1	abc	1	bcd
t2	abc.xyz	-1	zab.wxy
t3	a.Z	27	b.A
t4	AaZz	0	AaZz
t5	xyz	1	yza
t6	ééé	4	ééé
t7	a..z:A..Z	1234567	j..i:J..I

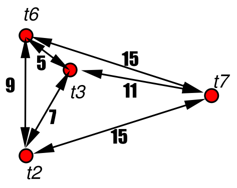
Table 2 Distance arrays of the Caesar cipher example for each string distance

	t1	t2	t3	t4	t5	t6	t7
Cartesian distance—C							
t1	0.0	200.53	76.23	108.62	39.84	209.6	224.12
t2	200.53	0.0	180.47	200.72	204.45	289.34	213.33
t3	76.23	180.47	0.0	109.99	101.01	186.97	206.18
t4	108.62	200.72	109.99	0.0	123.77	213.82	197.07
t5	39.84	204.45	101.01	123.77	0.0	249.44	238.01
t6	209.6	289.34	186.97	213.82	249.44	0.0	261.91
t7	224.12	213.33	206.18	197.07	238.01	261.91	0.0
Edit/Levenshtein distance—E							
t1	0	5	4	4	3	4	14
t2	5	0	7	8	5	9	15
t3	4	7	0	5	5	5	11
t4	4	8	5	0	4	5	15
t5	3	5	5	4	0	4	14
t6	4	9	5	5	4	0	15
t7	14	15	11	15	14	15	0
Hamming distance—H							
t1	0	7	4	6	3	4	16
t2	7	0	9	10	10	10	16
t3	4	9	0	5	5	5	15
t4	6	10	5	0	6	6	16
t5	3	10	5	6	0	4	17
t6	4	10	5	6	4	0	17
t7	16	16	15	16	17	17	0
Manhattan distance—M							
t1	0	454	117	197	69	366	847
t2	454	0	459	527	523	814	815
t3	117	459	0	198	186	359	730
t4	197	527	198	0	266	479	748
t5	69	523	186	266	0	435	916
t6	366	814	359	479	435	0	997
t7	847	815	730	748	916	997	0

A closer look at these tables reveals significant differences between the string distances. Hamming and Edit distances return integer values, in a small range (between 3 and 17). Manhattan distance also returns integers but in a larger range (between 69 and 997). Cartesian distance returns floating point numbers in a range between 39.84

Input string	Shift	Edit distance				
		E	t2	t3	t6	t7
abc.xyz	−1	t2	0	7	9	15
a.Z	27	t3	7	0	5	11
ééé	4	t6	9	5	0	15
a..z:A..Z	1234567	t7	15	11	15	0

(a)



(b)

Fig. 1 A subset of the Edit distance table and its 2D representation

Table 3 Position of test cases in prioritized suites

Test case	C	E	H	M	Average
t1: abc 1	7	6	7	7	6.75
t2: abc.xyz −1	4	2	3	3	3
t3: a.Z 27	6	4	5	6	5.25
t4: AaZz 0	5	5	4	4	4.5
t5: xyz 1	3	7	2	5	4.25
t6: ééé 4	2	3	6	2	3.25
t7: a..z:A..Z 1234567	1	1	1	1	1

and 289.34. One may expect that Hamming and Edit distances, which use a smaller range of values, will result in more equidistant test cases to choose from.

In the remainder of this example, we concentrate on a subset of the Edit Table, Fig. 1(a), with tests t2, t3, t6, and t7. For illustration, we use a two-dimensional representation of the distances between these test cases (Fig. 1(b)).

The first element of the prioritized test suite is the most distant test case with respect to the remaining elements of the test suite. In Fig. 1, t7 is obviously the most distant one. Its closest neighbor is t3 at distance 11, while the closest neighbors of t2, t3 and t6 are at distances 7, 5, and 5 respectively. This corresponds to the intuition we have when looking at the text of test cases (first two columns of Fig. 1(a), or Table 1), t7 is clearly very different! A closer look at Table 2 shows that t7 is actually the most distant test case in each of the tables. It is the first element in the permutation.

The remaining elements are chosen considering the elements already in the permutation. Both, t2 and t6, are at distance 15 from t7, so, according to Sect. 4.1 we choose the first one in the test suite, i.e., t2. The permutation thus becomes [t7, t2]. The remaining elements t3 and t6 are at distances 7 and 9 to the closest element in the permutation. We choose t6, which is the most distant one, and finally t3. The final permutation for the test suite of Fig. 1 is thus [t7, t2, t6, t3].

The results of computing the prioritized suites in our example for each distance are presented in Table 3. Columns C, E, H and M give the position of the test cases in the permutations corresponding to Cartesian, Edit, Hamming and Manhattan distances. The last column gives the average position of each test case. This example shows that all permutations differ, so the choice of a particular string distance has a significant influence on the result. Nevertheless, in all prioritized suites, t7 appears in the first position, and t1 has a rather low priority. The positions of t2, t3, and t4 do not vary much, but t5 and t6 may appear in very different positions. This example indicates that string distances can be used to assess diversity of test cases. In the next section, we report on an experiment which aims at confirming this on a standard benchmark.

5 Experimental evaluation

5.1 Goal of the experiments

The proposed test case prioritisation technique based on string distances does not use a program under test and thus can hardly be more efficient in finding bugs than prioritisation techniques which use code coverage criteria. Therefore, to evaluate its efficiency we should compare it with random prioritisation of a given test suite.

The proposed technique may yield different results for the four distance metrics, thus, it is interesting to know how the choice of a particular string distance affects its performance.

Therefore, our experiments aim at answering the following questions:

1. Is prioritisation based on string distances more efficient in finding defects than a random ordering?
2. Which string distance leads to better results?

5.2 The Siemens Test Suite

The "Siemens programs", also known as "Siemens Test Suite",[2] is a classical benchmark for various testing techniques. It was first introduced in an experimental study on the efficiency of several coverage criteria (Hutchins et al. 1994). The benchmark includes seven programs in the C programming language. The programs are rather small, ranging from 173 to 565 lines including comments (141 to 512 lines excluding blanks and comments). Each program comes with a base version, and several "mutants", i.e. variants of the program where faults have been seeded manually. Killing a mutant consists in running a test which gives different outputs on the base version and on the mutant. The second column of Table 4 gives the number of mutants for each base program. In the case of *replace* and *schedule2*, the given test suites did not detect (or "kill") one of the mutants.[3] These mutants were thus discarded from our experiments.

Each program comes with a test pool. In many experiments, these test pools are used to randomly select various smaller test suites with an average size being typically less than 20. In this study, we use the whole test pool as a test suite, which guarantees

[2]We retrieved these programs from http://www.cc.gatech.edu/aristotle/Tools/subjects/. They can also be retrieved from the Software artifacts Infrastructure Repository at http://sir.unl.edu (Do et al. 2005).

[3]We presume that this behavior of the test suite, which is different from the one reported in Hutchins et al. (1994) comes from the use of a C compiler or operating system different from that of the original experiment.

Table 4 The Siemens Test Suite

Program	Nb. of mutants	Nb. of tests	Nb. of tests which detect the	
			Most often killed mutant	Least often killed mutant
printTokens	7	4130	186 (4.5%)	6 (0.15%)
printTokens2	10	4115	518 (12.6%)	33 (0.80%)
replace	31 (32)	5542	309 (5.6%)	3 (0.05%)
schedule	9	2650	294 (11.1%)	4 (0.15%)
schedule2	9 (10)	2710	65 (2.4%)	2 (0.07%)
tcas	41	1608	132 (8.2%)	1 (0.07%)
totinfo	23	1052	211 (20.1%)	2 (0.19%)

that the test suite is able to kill all mutants. Considering such large test suites offers two interesting opportunities:

– It allows us to check whether the $O(n^2)$ complexity of our algorithm results in prohibitive execution times.
– Some mutants are killed by a very small fraction of the test suite (0.05% in the worst case), which decreases their probability to be killed early by a random test suite. This may well happen in industrial test suites where a single test case could be added after a bug report. Such test suites allow us to evaluate the ability of our prioritisation algorithm to detect these test cases and place them earlier in the prioritized test suite than in a random prioritisation.

Nevertheless, since the test pool was not intended to be used as a single test suite, there may be a lot of redundancy between test cases, and these test pools may not exhibit a behavior of industrial test suites of the comparable size.

Each test pool consists of test inputs only; the correct outputs must be computed by running the base program. The third column of Table 4 gives the size of each test suite. It ranges from 1608 to 5542 test cases. We count the number of tests which detect the mutant which is killed most often, and the one killed least often, see the fourth and fifth columns of Table 4. Since the programs are written in C their behavior may depend on the compiler and the operating system used. We used gcc 4.1.3 under Linux (Ubuntu 7.10). For each program, we used the full corresponding test suite.

For each Siemens program, the main text of all test cases is gathered in a file named "universe". Each line of the file corresponds to a single test case and gives the list of arguments passed to the program under test. For example, the first line of the "universe" file of *tcas* is:

```
958 1 1 2597  574 4253 0  399  400 0 0 1
```

In such a case, it is easy to turn this line into a string of characters which will be used by our prioritisation algorithm. The resulting string is:

```
"958 1 1 2597  574 4253 0  399  400 0 0 1"
```

In other programs, the test cases also refer to external files, stored in a directory named “inputs”. For example, the first two lines of the “universe” file of *replace* are

```
'-?' 'a&' < ../inputs/temp-test/1.inp.1.1
' ' '@%@&' < ../inputs/temp-test/777.inp.334.1
```

These test cases use redirection of the standard input to external files which include additional inputs for the test cases. We could have measured the distance between the strings including the names of the files. But it is obviously more interesting to replace the file names by their contents. The following table lists the contents of these two files.

File name	Contents
../inputs/temp-test/1.inp.1.1	`|abcd| -a |abcd|`
../inputs/temp-test/777.inp.334.1	`|abcd| |abcd|`

Therefore, the tool implementing the prioritisation technique was configured to detect references to external files and replace them with the actual contents of the file, prefixed with “<” to keep track of what was inlined. The resulting strings used by our prioritisation are thus:

```
"'-?' 'a&' <  |abcd| -a |abcd| "
"' ' '@%@&' <  |abcd|   |abcd| "
```

5.3 Measuring the efficiency of a prioritized test suite

In order to evaluate the efficiency of a prioritized test suite in finding defects, we will determine, for each mutant i, the position of the first test which kills the mutant (TF_i). To illustrate our evaluation method, we use a random permutation of the tests of *printTokens*. The first two columns of Table 5 show the test's number for each killed mutant. The third and fourth column of Table 5 show the number of killed mutants along with the number of executed tests. This dependence is illustrated in Fig. 2. It indicates that the number of mutants killed grows rather rapidly and, after 713 of the 4130 tests of the test suite, all defects are detected.

Based on such measurements, we can compute the APFD (Average Percentage of Faults Detected) (Rothermel et al. 2001), which is a classical measure to evaluate the

Table 5 Measures from a randomly prioritized test suite for printTokens

Mutant	Killed by test (TF_i)	Nb. of tests	Nb. of mutants killed
v1	713	11	2
v2	46	21	4
v3	11	46	5
v4	298	298	6
v5	21	713	7
v6	21		
v7	11		

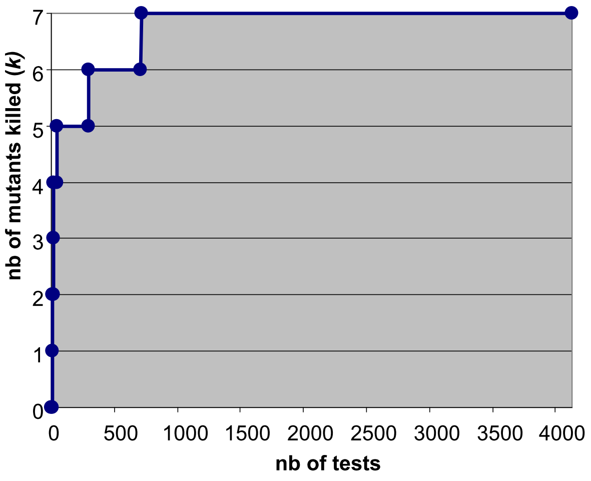


Fig. 2 APFD of a randomly prioritized test suite for printTokens

efficiency of a prioritized test suite. Other measures are based on code coverage (Li et al. 2007); however, we only measure the efficiency by APFD and sometimes by the test suite's ability in killing the strongest mutants. APFD measures the percentage of the gray area, underlying the curve in Fig. 2, compared to the whole rectangle of the figure. A higher APFD corresponds to a better fault detection rate. APFD is higher when every mutant is killed earlier, so it measures a combined performance for the whole population of mutants.

In Elbaum et al. (2002), APFD is computed as follows:

– Let n be the number of test cases and m be the number of mutants.
– Let TF_i be the index of the first test case which reveals fault i (i.e. kills mutant i).

$$\mathrm{APFD} = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{n * m} + \frac{1}{2 * n}$$

For example, let us consider the simple example of Fig. 3 (taken from Rothermel et al. 2001). In this example, the test suite includes five tests ($n = 5$). They were applied to ten mutants ($m = 10$). The APFD of this test suite is 50%. It can be computed as

$$\mathrm{APFD} = 1 - \frac{1+1+2+2+3+3+3+5+5+5}{5 * 10} + \frac{1}{2 * 5} = 1 - \frac{30}{50} + \frac{1}{10} = 0.5$$

The earlier a prioritized test suite kills the first and last mutants the higher its APFD. The APFD measured for Fig. 2 is 96.13%. In most of our experiments, APFD will be over 95%. This is due to the fact that the test suites are much larger than the average number of tests needed to kill all the mutants. In Fig. 2, all mutants are killed by 17% of the tests, and 5 out of 7 mutants are killed by only 1.1% of the test suite. In Rothermel et al. (2001), Rothermel et al. use much smaller test suites, with an average size less than 20, which resulted in much smaller APFDs, like in Fig. 3. Thus an absolute value of APFD is not very informative; what matters is how APFD varies for permutations of the same test suite.

Mutant i	Killed by TF_i
1	1
2	1
3	2
4	2
5	3
6	3
7	3
8	5
9	5
10	5

Nb. of tests	Nb. of mutants killed
0	0
1	2
2	4
3	7
4	7
5	10

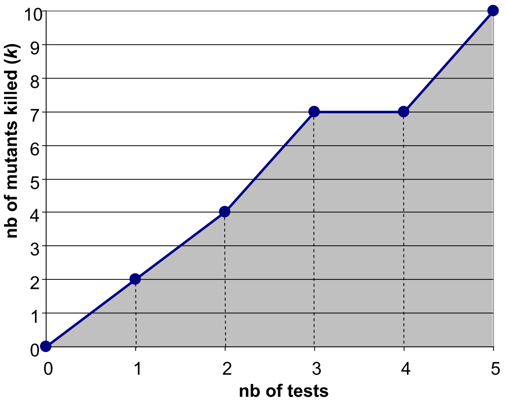


Fig. 3 Results of a small test suite (taken from Rothermel et al. 2001) and graphical representation

Table 6 Number of tests to kill a percentage of the mutants with the test suite in Table 5

% of mutants killed	25%	50%	75%	100%
Nb. of tests	11	21	109	713

Measuring the impact on strong mutants By "strongest" mutant, we mean a mutant which is the latest mutant killed by a given test suite. In order to see how prioritized test suites behave for various strengths of mutants, we compute the number of tests needed to kill 25%, 50%, 75% and 100% of the mutants (Table 6). These values are linearly interpolated when the percentage does not correspond to a natural number of mutants. The linear interpolation is computed as follows:

- Let m be the number of mutants.
- Let $p \in \{25\%, 50\%, 75\%\}$ be a percentage.
- Let q be $m * p$, the number of mutants corresponding to a given percentage. Let $int(q)$ and $frac(q)$ be the integer part and fractionary part of q respectively. If $frac(q) \neq 0$, then linear interpolation is needed.
- Let i and j be the first tests which kill at least $int(q)$ and $int(q) + 1$ mutants respectively. Linear interpolation is computed as $i + (j - i) * frac(q)$. If the result is not a natural number, it is rounded following the rules of integer calculation in java.

For example, in Table 6, 75% of 7 mutants corresponds to 5.25 mutants ($p = 75\%, m = 7, q = 5.25$). Since the fractionary part of q is not null ($frac(q) = 0.25$), we must perform linear interpolation. It takes 46 tests to kill 5 mutants ($i = 46$), and 298 mutants are needed to kill 6 mutants ($j = 298$), we interpolate linearly the number of tests corresponding to 5.25 mutants as $46 + (298 - 46) * 0.25 = 109$.

In fact, these linear approximations are rather artificial. They are thus given here as an indication of how the prioritized suites perform for various strengths of mutants. Still, most of our conclusions are based on 100% of mutants and APFD measurements.

5.4 The experiments

The main goal of our experiments is to compare the efficiency of the proposed test case prioritisation technique based on string distances with random prioritisation. The experiments were performed for each Siemens program. In order to obtain average measures for the random permutations, we created 31 random permutations of the original test suite. The Central Limit Theorem of statistics tells us that if the sample size is sufficiently large, the distribution of the sample average of the random variables approaches the normal distribution. The sample size is usually considered to be sufficiently large if it exceeds 30. As discussed in Sect. 4.1, in a given test suite there might be several equidistant test cases, and the results of the proposed algorithm depend on the initial ordering of tests, therefore, we have decided to apply our prioritisation algorithm to each of the 31 random permutations.

5.4.1 Test suites considered in our experiments

Each Siemens program was tested using:

– 31 random permutations of the original test suite,
– four prioritized versions of each random permutation ($4 * 31 = 124$ prioritized test suites), corresponding to the four string distances presented in Sect. 3.

Our main goal is to compare prioritized test suites to random test suites. Nevertheless, for reference purposes, we also tested each program using the following three test suites:

– the original, non permuted, test suite,
– the original test suite in inverted order,
– A simple code-based prioritisation of the original test suite, corresponding to a variant of Rothermel's total statement coverage prioritisation (Rothermel et al. 2001). This prioritisation method sorts the test cases in descending order of line coverage. So the test cases covering the largest number of lines of code are executed first. When two test cases cover the same number of lines of code, we keep the ordering of the original test suite.

While we experimented with 31 versions of the random and prioritized test suites, there is only one version of the three reference test suites, because the inverted test suite and the code-based prioritisation are deterministically computed from the original test suite.

5.4.2 APFD measurements

Figure 4 gives the average APFDs measured for each program and each prioritisation method. It also shows the confidence interval (at 95%) for each of these average values. The horizontal axis displays the various prioritisation methods. R stands for the random permutation; C, E, H, and M for the prioritisation with Cartesian, Edit, Hamming, Manhattan distance, respectively. Figure 4 also shows the value of APFD for the original test suite (denoted as O), the inverted original test suite (denoted

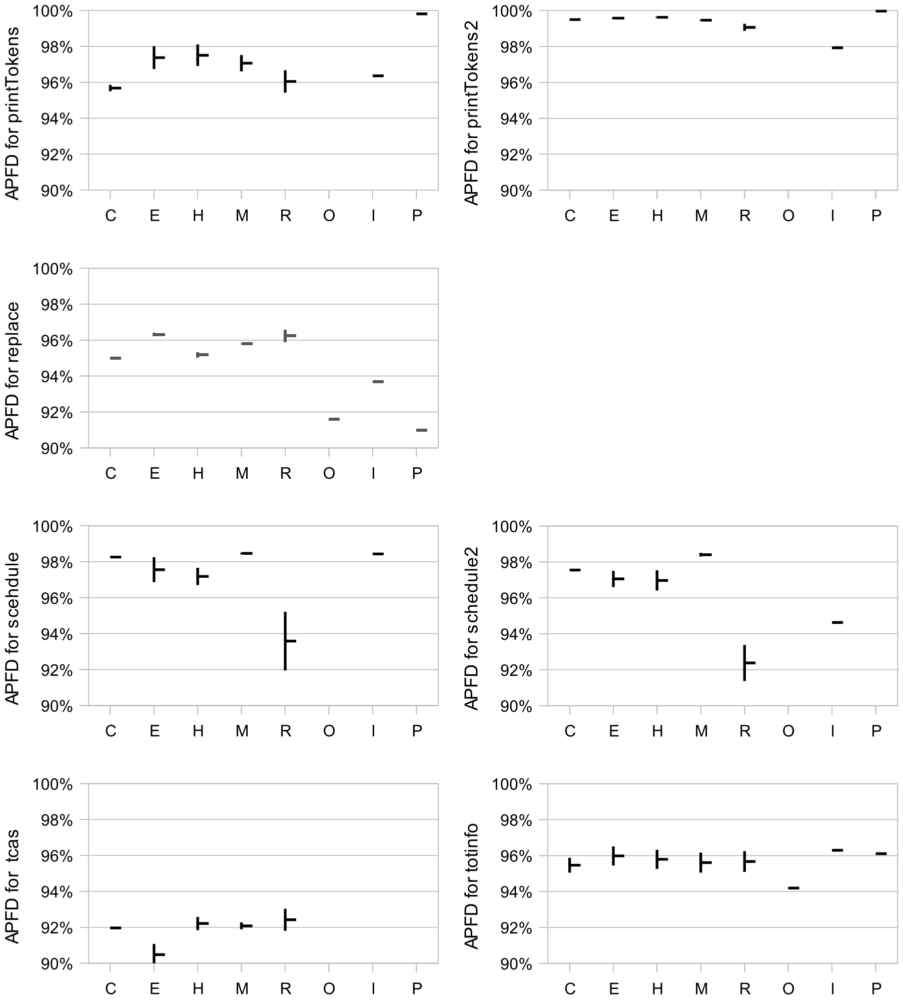


Fig. 4 APFD (average and confidence interval at 95%) for each prioritisation method (higher values are better); small values for O, I and P may be outside of the diagram

as I) and the code-based prioritisation of the original test suite (denoted as P). Since these are not average values, but correspond each to a single measurement, there is no associated confidence interval. The vertical axis gives the APFD, expressed as a percentage. We only display the 90–100% range. In some cases, e.g. *tcas*, the APFD for O, I, or P may be less than 90% and falls out of the diagram. The APFD values are also given in the sixth column of Fig. 5.

5.4.3 *Average number of tests to kill various percentages of mutants*

The first columns of Fig. 5 give the average number of tests needed to kill 25%, 50%, 75% and 100% of the mutants, for each program. Figure 6 displays them on a logarithmic scale. We choose a logarithmic scale because the number of tests needed to kill the first mutants is usually small (less than 100) while the number of tests needed to kill the last mutants is usually ten to a hundred times larger. Choosing a logarithmic scale allows us to put all results in the same graphic. In the case of random and distance-based prioritized suites, these values are the average of the 31 measurements. In the case of O, I, and P, it is the single measurement performed.

5.4.4 Execution time

We also measured the time needed to prioritize test cases for each of the programs and each string distance. Since test suites have different sizes, the time to prioritize them varies significantly, however computing distance arrays like the ones in Table 2 consumes most of it. Table 7 shows the time required to prioritize the original test suite for each program and each distance. The measurements were performed on an Intel Pentium M processor at 1.6 GHz with 1 Gb of RAM, under Linux (Ubuntu 7.10). This table reveals that Edit distance consumes significantly more time than the three other distances, due to its algorithmic complexity.

5.5 Statistical analysis using Tukey's HSD Test

We first compare the distance-based prioritization (CEHM) with the random permutation (R). The other test suites (OIP) are considered later. In our experiments, the independent variable is the kind of prioritisation method used. It takes five values corresponding to the four distances and to random prioritisation: Cartesian, Edit, Hamming, Manhattan, Random. The dependent variables are the number of tests needed to kill 25%, 50%, 75%, and 100% of the mutants, and the APFD. The experiments are repeated for each of the Siemens programs.

A statistical analysis of these data was conducted to check whether the differences between the average values of the dependent variables are significant. We first successfully checked normality of distribution and variance homogeneity of the dependent variables. We then performed a variance analysis (ANOVA) (Snedecor and Cochran 1957) to test the differences between average values of the dependent variables corresponding to each of the independent variables, i.e. the various distance-based and random prioritization. As shown in Tables 8 and 9, ANOVA analysis revealed significant differences in most cases. Only four cases were judged not significant by ANOVA. They are marked as "not significant" in Table 8 (*printTokens* for 100% killed, and *totinfo* for 100% killed and APFD) and Table 9 (*totinfo* for 50% killed). This means that for these four cases, the differences between the various prioritisation methods are not statistically significant.

In all other cases, ANOVA tells us that at least one of the prioritization has an average value significantly different from the others. ANOVA does not tell us which prioritisation is different. Therefore, we performed Tukey's HSD (Honestly Significant Difference) test (Braun and Tukey 1994) on the obtained data. Tukey's HSD test compares the averages of all series of results and decides whether those are significantly different. The results of Tukey's HSD test are given in Tables 8 and 9. The first line of each cell of Table 8 and 9 gives the ranking of the various prioritisation

printTokens	**25%**	**50%**	**75%**	**100%**	**APFD**
C	48.74	88.26	166.13	606.13	95.68%
E	19.39	45.52	101.77	393.19	97.37%
H	17.77	41.97	90.65	395.45	97.51%
M	16.00	36.29	94.74	518.68	97.06%
R	15.84	51.03	153.81	627.42	96.05%
O	84.00	535.00	1573.00	2037.00	76.69%
I	63.00	122.00	184.00	275.00	96.36%
P	1.00	1.00	3.00	47.00	99.81%
printTokens2	**25%**	**50%**	**75%**	**100%**	**APFD**
C	1.00	12.00	28.00	58.19	99.50%
E	1.00	14.42	20.81	65.77	99.58%
H	1.00	9.97	14.84	72.06	99.62%
M	1.00	16.00	21.00	87.55	99.46%
R	8.03	16.19	34.90	187.55	99.06%
O	1222.00	2023.00	2028.00	2293.00	57.69%
I	68.00	81.00	92.00	123.00	97.92%
P	1.00	1.00	1.00	7.00	99.97%
replace	**25%**	**50%**	**75%**	**100%**	**APFD**
C	148.00	213.00	334.00	1017.00	94.99%
E	90.06	134.45	176.45	904.81	96.31%
H	136.29	180.29	231.48	1532.52	95.18%
M	123.00	209.00	303.48	604.42	95.79%
R	19.35	48.03	158.68	2215.00	96.23%
O	27.00	51.00	434.00	4188.00	91.60%
I	4.00	60.00	376.00	4171.00	93.68%
P	7.00	34.00	327.00	5461.00	90.97%
schedule	**25%**	**50%**	**75%**	**100%**	**APFD**
C	8.00	41.00	52.00	149.26	98.27%
E	5.35	20.29	68.65	225.77	97.56%
H	15.32	37.77	88.13	215.68	97.19%
M	7.35	32.10	64.39	89.94	98.47%
R	8.48	36.97	142.45	607.13	93.59%
O	16.00	717.00	2281.00	2399.00	55.54%
I	2.00	9.00	81.00	111.00	98.44%
P	5.00	34.00	1518.00	2621.00	71.60%
schedule2	**25%**	**50%**	**75%**	**100%**	**APFD**
C	10.00	25.00	66.00	197.13	97.55%
E	5.58	29.45	70.10	384.32	97.05%
H	13.77	35.32	81.58	370.35	96.97%
M	9.68	33.03	69.29	84.61	98.40%
R	30.19	61.16	127.26	1069.13	92.37%
O	1401.00	1776.00	2368.00	2482.00	28.85%
I	61.00	125.00	201.00	245.00	94.63%
P	1.00	52.00	736.00	2570.00	77.82%
tcas	**25%**	**50%**	**75%**	**100%**	**APFD**
C	30.00	91.00	105.00	563.23	91.96%
E	18.87	57.55	200.52	810.52	90.48%
H	14.23	47.74	115.16	833.81	92.21%
M	16.00	74.00	165.32	737.06	92.08%
R	14.97	42.03	119.71	1036.74	92.42%
O	13.00	65.00	206.00	864.00	89.71%
I	45.00	81.00	413.00	1278.00	82.92%
P	4.00	312.00	637.00	1001.00	76.03%
totinfo	**25%**	**50%**	**75%**	**100%**	**APFD**
C	2.00	12.00	25.94	387.65	95.46%
E	3.29	10.52	33.10	330.42	95.98%
H	4.84	11.35	33.16	332.94	95.79%
M	2.00	12.00	27.00	363.84	95.61%
R	5.74	13.87	35.90	324.26	95.67%
O	2.00	6.00	22.00	541.00	94.19%
I	3.00	11.00	58.00	215.00	96.29%
P	1.00	16.00	33.00	269.00	96.10%

Fig. 5 Average number of tests needed to kill 25%, 50%, 75%, 100% of the mutants (lower values are better) and APFD (higher values are better). Legend for the prioritisation methods: C: Cartesian distance, E: Edit distance, H: Hamming distance, M: Manhattan distance, R: Random permutation, O: Original testsuite, I: Inverted testsuite, P: Prioritized using line coverage

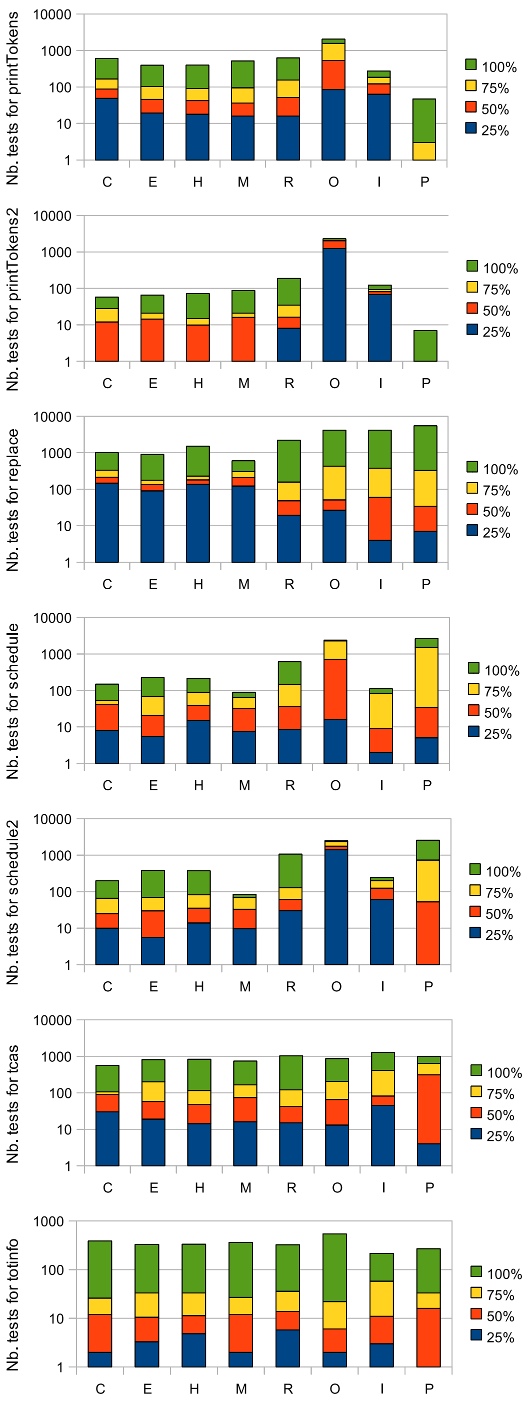


Fig. 6 Graphical representation on a logarithmic scale of the number of tests needed to kill 25%, 50%, 75%, 100% of the mutants for each prioritisation method

Table 7 Time (ms) needed to prioritize each original test suite

	C	E	H	M
printTokens	6228	130068	5704	6476
printTokens2	6296	130184	5616	6616
replace	22353	2316445	16893	26214
schedule	3384	121360	3296	3788
schedule2	3548	126456	3400	3964
tcas	1516	14728	1476	1820
totinfo	744	15937	748	728

Table 8 Ranking of the various prioritisation methods with respect to the number of tests needed to kill 100% of the mutants, and to the APFD

	100% killed	**APFD**
printTokens	EH MCR (not significant) P I O	HE M R C P I O
printTokens2	CEHM R P I O	HECM R P I O
replace	M E C H R I O P	ER M HC I O P
schedule	M CHE R I O P	M CEH R I P O
schedule2	M C HE R I O P	M C EH R I P O
tcas	C ME H R O P I	RHMC E O I P
totinfo	R EHMC (not significant) I P O	EH RM C (not significant) I P O

methods. The first column of Table 8 corresponds to the number of tests needed to kill 100% of the mutants. The second column represents the APFD. In a given cell, we order the prioritisation methods, from the best to the worst. Moreover, the prioritisation methods whose results are not significantly different are grouped in the same hexagon. For example, considering the APFD for *printTokens*2, there is no statistically significant difference between all distance-based prioritization (H, E, C, and M), but they are all statistically better than random prioritisation (R). In some cases there is an overlap between groups of methods. For example, the highest value of APFD for *printTokens* is achieved using Hamming Distance (H). But H is not statistically different from E (Edit) or M (Manhattan). Random prioritisation (R) is statistically worse than H and E, but R is not statistically different from M or C. Finally M is statistically better than C. Table 9 gives the results of Tukey's test for the lower percentages of mutants killed (25% to 75%).

Tables 8 and 9 also consider the original test suite (O), the inverted original test suite (I) and the code-based prioritized suite (P). Since these correspond to a single measurement, and not the average value of several measurements, we cannot use

Table 9 Ranking of the various prioritisation methods with respect to the number of tests needed to kill 25%, 50% and 75% of the mutants

	25% killed	**50% killed**	**75% killed**
printTokens	RMHE C P I O	MHER C P I O	HME RC P I O
printTokens2	CMHE R P I O	H CE MR P I O	H EM C R P I O
replace	R E M H C I P O	R E H MC P O I	RE H M C P I O
schedule	E MCR H I P O	E M RH C I P O	C M E H R I P O
schedule2	EMCH R P I O	CEMH R P I O	CMEH R I P O
tcas	H R M E C P O I	RH E M C O I P	CHR M E O I P
totinfo	CM E H R P O I	E H CM R (not significant) O I P	CM EH R O P I

Tukey's test which compares two average values. Therefore, we computed the confidence interval at 95% for each series of results, and checked whether or not O, I, and P fall into this confidence interval. If this is the case, the single value is not statistically different from the average value. The results of these comparisons are displayed on the second line of each cell of Tables 8 and 9. When a single value is not statistically different from one or several average values, we draw a rectangle around the single value which stretches under the average values. For example, considering the "100% killed" cell of *printTokens*, it shows that P is statistically better than all other prioritisation methods. I is not statistically different from E and H, but is statistically better than M, C, and R. This cell also shows that O is statistically worse than all other prioritisation methods.

The APFD cell of *totinfo* is a little more complex because it features intersecting rectangles for I and P. It means that I is not statistically different from E and H, and that P is not statistically different from E, H, R, and M.

5.6 Analysis of the results

5.6.1 *Comparison of distance-based and random prioritization*

Addressing the question *"Is prioritisation based on string distances more efficient in finding defects than a random ordering?"*, we use Table 8, the sixth column of Figs. 5 and 4, which indicate that APFDs for the distance-based prioritized suites is definitely better than for the randomly-ordered ones for programs *printTokens*2, *schedule*, and *schedule*2. This is confirmed by the statistical analysis. In the case of *printTokens*, two of the four prioritized suites are significantly more efficient than the randomly ordered suites. The remaining prioritized suites are not statistically different than the randomly ordered suites. In the case of *totinfo*, the statistical analysis shows that the

Table 10 Overall ranking for killing the strongest mutant and APFD

Prioritization method	Sum of tests to kill last mutant	Rank	Sum of APFDs	Rank
Random	6067.23	5	665.39	5
Cartesian	2978.58	2	673.4	4
Edit	3114.81	3	674.33	3
Hamming	3752.81	4	674.47	2
Manhattan	2486.10	1	676.87	1

difference is not significant. So in five of the seven case studies, prioritized suites are statistically either better than or equal to the randomly ordered ones.

In the case of *replace* and *tcas*, the randomly ordered test suites perform significantly better than several distance based prioritization. Actually, this is not surprising for *tcas*. As shown in Sect. 5.2, the inputs of *tcas* are numerical values and we provided several examples in Sect. 3.4 where string distances do not capture correctly the distance between numerical values. Concerning *replace*, Edit distance performs the best, which is not surprising when looking at the first two lines of inputs given in Sect. 5.2. Actually these two lines have common substrings (e.g. |`abcd`|) which may appear at different positions. Edit distance is well suited to detect such similarities.

Tables 8 and 9, Figs. 5 and 6, provide more details to better understand these results. They show that for *schedule*2 the prioritized test suites outperform the randomly ordered suites for any percentage of mutants. For *printTokens*2, randomly ordered suites are significantly worse to kill the first and the last mutants, and appear in the last position for the intermediate percentages. For *schedule*, prioritized suites prevail for the highest percentages. For *totinfo*, differences are not statistically different, but prioritized test suites are better until 75% of the mutants, while the randomly ordered test suites are quicker to kill the last mutant. For *replace* and *tcas*, randomly ordered test suites are faster until 75% of the mutants, although Hamming-based prioritisation is not statistically different for *tcas*. But prioritized test suites are statistically quicker to kill the last one in both cases. As shown in Fig. 5, prioritized test suites are better to kill the last mutant for 6 of the 7 programs. Table 8 shows that the difference is significant for 5 programs and that randomly permuted test suites are never significantly better in killing the last mutant. Only for *totinfo*, the random permutation is more efficient than the distance based prioritisation, but the difference is not statistically significant. This is confirmed by Table 10 where column 2 shows the sums of the average number of tests needed to kill the last mutant for all 7 programs. On average, prioritized suites achieve that nearly twice as fast as the randomly ordered suites, with Manhattan distance providing the best results. The fourth column of Table 10 gives the sum of the seven APFDs. Once again, prioritized test suites are better than the randomly ordered ones, and Manhattan distance provides the best results. For completeness sake, Table 11 gives these rankings for lower percentages of mutant killings.

The experimental results indicate that

– prioritized test suites are more efficient than randomly ordered test suites in detecting the strongest mutants. This is confirmed by Tukey's test.
– On average, prioritized test suites have a higher APFD than randomly ordered ones.

Table 11 Overall ranking for lower percentages of mutant killing

Prioritization method	Sum of tests to kill 25% of mutants	Rank	Sum of tests to kill 50% of mutants	Rank	Sum of tests to kill 75% of mutants	Rank
Random	102.61	1	269.29	1	772.71	4
Cartesian	247.74	5	482.26	5	777.06	5
Edit	143.55	2	312.19	2	671.39	2
Hamming	203.23	4	364.42	3	655	1
Manhattan	175.03	3	412.42	4	745.23	3

In Sect. 3.4, we pointed out that string distances do not capture all semantic differences between test cases. Therefore, it is not surprising that the prioritisation algorithm using such a distance produces quite diverse results on the Siemens Test Suite. Nevertheless, the resulting prioritized test suites are more efficient overall than randomly ordered ones. This indicates that string distances capture nevertheless significant information about diversity of test cases.

5.6.2 Comparison with the reference test suites

For completeness and reference sake, we also provide the measures corresponding to the original test suite (O), the inverted original test suite (I), and a simple code-based prioritisation (P).

Regarding the original test suite, it appears that its APFD is significantly worse than all distance-based prioritization. We get a similar result considering its ability to kill the last mutant, except in the case of *tcas* where Hamming distance is better than O, but not statistically different. Regarding lower percentages (Table 9), this is not necessarily true, but in a majority of cases, especially at the highest percentages, it appears to be worse than distance-based prioritization. For the Siemens test suite, it is always more effective to use a distance-based prioritisation of the test suite than the original test suite. One may conjecture that, when the test engineer did not care about the ordering of the original test suite, distance-based prioritisation will be effective.

The inverted original test suite (I) gets better results, it even appears to have the best APFD of all methods in the case of *totinfo*. One explanation for these good results is that the last test cases of the original test suites may have been added to address specific mutants which are difficult to kill. This also explains why the original test suite has difficulties in killing these mutants whose killing tests appear at the end of the suite.

The results of the code-based prioritisation (P) may appear surprising: P is the best method in two cases (*printTokens* and *printTokens*2); it is not statistically different than most methods in one case (*totinfo*) and is statistically worse than all distance-based prioritization in four cases (*replace*, *schedule*, *schedule2*, and *tcas*). This can be explained by the rather simple prioritisation algorithm used for this code-based prioritisation. We conjecture that better results would be obtained using additional statement coverage prioritisation (Rothermel et al. 2001), where each test case is chosen to improve the coverage of the previous ones in the prioritized test suite.

Using total statement coverage as prioritisation criterion, test cases with the highest coverage appear first in the test suite, but there is a chance that these are similar test cases which kill the same mutants. This is confirmed by the fact that P gets excellent results to kill the first mutants: it is always better or not statistically different than all distance-based prioritization. Unfortunately, considering the strongest mutants, they may correspond to test cases which address only an exceptional behavior and have small code coverage. As a result, these tests appear late in the prioritized test suite, reducing its ability in killing the last mutant, and impacting APFD. Moreover, since our test suites are significantly longer than the ones used in Rothermel et al. (2001), the difficulty to kill the last mutants may have enormous impact on APFD. For example, consider *replace* where P performs well until 75%, but needs 5461 test cases to kill the last mutant.

Nevertheless, these measures on O, I, and P were mainly performed to provide some reference numbers. They actually indicate that distance-based prioritisation performs relatively well compared to these references.

5.6.3 Choice of a distance

When a given application is considered, a careful look at the nature of its inputs may lead to favor one distance over the others. For example, in the case of *replace*, we saw that Edit distance appears well-adapted to detect common substrings in the inputs. In the case of *tcas* the numerical nature of the inputs may lead to discard string distances in favor of a numerical distance, but this would require extensions of the prioritisation tool.

Nevertheless, in cases when the structure of the input does not suggest the use of a specific distance, one can favor a distance which yields good results on average. Addressing the question *"Which string distance leads to better results?"*, we consider Table 10 which indicates that Manhattan distance provides the best overall results. The third and fifth columns give the ranking of each prioritisation method. Table 12 details the ranking for all programs (excluding randomly ordered test suites). The table shows that each distance may be ranked the best for a given program and Table 8 shows that the differences between distance based prioritisation methods are not statistically significant in many cases. But on average, Manhattan distance is most efficient in killing the last mutant. Nevertheless considering APFD, Edit and Hamming distances have the same average ranking as Manhattan distance.

A closer look at Fig. 4 reveals that confidence intervals depend on the distance chosen for prioritisation. Actually, Cartesian distance has the smallest confidence intervals for all programs. This suggests that prioritisation based on Cartesian distance is more deterministic than the other. This could be conjectured from Table 2 because Cartesian distances are floating point numbers, and two test cases are less likely to be equidistant to already selected tests. Manhattan distances occur as natural numbers, but their range of values is much larger than that of Edit and Hamming distances. This explains why the APFDs associated to Manhattan distances have a smaller confidence interval than the ones corresponding to Edit or Hamming distances.

We conclude that Manhattan distance appears to offer a good compromise between efficiency and determinism of the resulting test suite, moreover, the cost of comput-

Table 12 Rankings of criteria for each program

Program	Kill 100% mutants				APFD			
	C	E	H	M	C	E	H	M
printTokens	4	1	2	3	4	2	1	3
printTokens2	1	2	3	4	3	2	1	4
replace	3	2	4	1	4	1	3	2
schedule	2	4	3	1	2	3	4	1
schedule2	2	4	3	1	2	3	4	1
tcas	1	3	4	2	3	4	1	2
totinfo	4	1	2	3	4	1	2	3
Average rank	2.4	2.4	3.0	2.1	3.1	2.3	2.3	2.3

ing the Manhattan distance is rather small, especially in comparison to Edit distance which has a O($m * n$) complexity, as witnessed by Table 7.

5.7 Threats to validity

Threats to construct validity correspond to the choice of the right measures. APFD is a classical means for measuring the efficiency of a prioritized test suite. In our experiment, we also measured the number of test cases needed to kill the last mutant, which measures the capability of the prioritisation technique to kill the "strongest" mutant. Other measures (APBC, APDC and APSC), based on code coverage, have been proposed by Li et al. (2007), and their use should probably be considered in further work.

Threats to internal validity correspond to potential faults in our implementations of the prioritisation algorithm, the string distances, the measures such as APFD or the statistical analyses. Actually, our implementations have been coded with care. They have been tested on small examples, such as the one in Sect. 4.2, and the results were verified manually. We also reused several classes of our implementation to compute several metrics about the test suites, and to experiment with test suite reduction techniques (not reported in this paper). So we expect that these have reduced the likelihood of errors in our implementations. Regarding statistical analysis, it was performed by an expert in the field using professional statistical software (SAS[4]).

Threats to external validity address the following question: are the subjects of our studies representative of real programs and real test suites? The Siemens Test Suite is a classical and mandatory benchmark for prioritisation studies. It was used in several significant research works in the field (Rothermel et al. 2001; Li et al. 2007). Nevertheless, its representative character has already been largely debated. The programs are rather small, faults of the mutants are seeded, and test suites have been constructed to ensure certain coverage and fault detection capabilities. Moreover the test pools were not intended to be used as a full test suite. This motivates our future work to apply the proposed technique to larger and more industrial programs.

[4]http://www.sas.com.

Nevertheless, in this study, we were less interested in the length of programs than in the size of test suites, which count several thousands of test cases each. Actually, our algorithm features a $O(n^2)$ complexity due to the computation of the distances between all pairs of test cases. It is thus sensible to the size of the test suite. In real programs, such as Jonas (Kessis et al. 2005), test suites contain thousands and even tens of thousands test cases.

Another threat to external validity is the format of the test data. In the Siemens programs, test data are given in textual format, although they vary from numerical data for the *tcas* program to arbitrary sequences of characters for the *printTokens* program. None of these test suites correspond to structured programs such as JUnit tests, and this should be considered in future work.

6 Conclusion

In this paper we proposed and evaluated a new prioritisation technique, based on string distances between test cases in a given suite. Our technique does not require the availability of an implementation or a specification of the program under test to prioritize the test suite: it only relies on the information present in the test suite. This allows using this prioritisation technique for initial testing, in cases when the implementation has significantly changed since the last execution of the test suite, and in cases when code instrumentation is difficult.

We elaborated a simple greedy algorithm for test prioritisation and performed experiments using four classical string distances. The experiments relying on the Siemens Test Suite were performed to compare the proposed prioritisation technique with random ordering of test cases. The experimental results indicate that the test suites prioritized by the proposed technique are more efficient in detecting the strongest mutants. A statistical analysis based on Tukey's test confirmed that the difference in efficiency is statistically significant. The results lead to conclude that, on average, the proposed technique has a better APFD (Average of the Percentage of Faults Detected) than randomly ordered test suites. These conclusions are only valid on average, however, in some instances a random ordering may outperform the proposed technique, since string distances do not necessarily reveal semantic differences between test cases. The obtained experimental results also indicate that on average, Manhattan distance gives better results than the other string distances.

As future work, it might be interesting to investigate other string distances and experiment with industrial applications. While experiments with larger real programs are in our plans, experiments (Jiang et al. 2009) with edit-distance prioritisation show that for larger applications the effect of the string distance based prioritisation is even more profound.

At the same time, it would be interesting to consider additionally other non-classical string distances, see, e.g., Cohen et al. (2003).

In order to reduce the threats to external validity, further experimentations are needed with other, in particular larger programs and test suites. Along with APFD and killing the strongest mutants, other measures could also be considered to assess the efficiency of the prioritized test suites.

Another possibility is to try to extract more semantics from a given test suite, by using more specific distances or taking more information into account. Test cases are often structured objects, for example, JUnit test cases follow the Java syntax. In such cases, it might be more appropriate to measure a distance between the syntax trees of the test cases. Such a technique would still be independent of the availability of the system under test, but might capture significant semantic information to compare test cases. Also, one may consider a combination of inputs and outputs of the test cases in the computation of the string distances. Taking outputs into account is interesting when small differences in the inputs may result in significant differences in the outputs (e.g. computing the square root of 1 and -1). This no longer follows the "minimal" approach of this paper, but it makes sense in cases when execution of the test cases is cheap and it is easy to determine the corresponding outputs.

Finally, our current work is to adapt the proposed approach to reduce a test suite. Using string distances, it is relatively easy to reduce a test suite to a subset such that the distance between each element of the original test suite to the reduced test suite is less than a given threshold. Varying such a threshold provides a means of controlling the size of the reduced test suite. We have already implemented such a reduction technique and are investigating possibilities for defining a threshold based on statistical information extracted from the test suite.

Acknowledgements This work was partly supported by the NSERC discovery grant OGP0194381. We acknowledge very useful comments of the reviewers.

References

Braun, H., Tukey, J.: The Collected Works of John W. Tukey: Multiple Comparisons, 1948–1983, vol. 8. Chapman & Hall, London (1994)

Chan, F.T., Chen, T.Y., Mak, I.K., Yu, Y.T.: Proportional sampling strategy: guidelines for software testing practitioners. Inf. Softw. Technol. **38**(12), 775–782 (1996)

Charikar, M., Hajiaghayi, M.T., Karloff, H.J., Rao, S.: l_2^2 spreading metrics for vertex ordering problems. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006), Miami, Florida, USA, 22–26 January 2006, pp. 1018–1027. ACM Press, New York (2006)

Chen, T.Y., Leung, H., Mak, I.K.: Adaptive random testing. In: ASIAN 2004. Lecture Notes in Computer Science, vol. 3321, pp. 320–329. Springer, Berlin (2004)

Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proc. of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), Acapulco, Mexico, pp. 73–78 (2003)

Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir. Softw. Eng. **10**(4), 405–435 (2005)

Elbaum, S.G., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)

Feijs, L.M.G., Goga, N., Mauw, S., Tretmans, J.: Test selection, trace distance and heuristics. In: Schieferdecker, I., König, H., Wolisz, A. (eds.) Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems (TestCom 2002), Berlin, Germany, 19–22 March 2002, vol. 210, pp. 267–282. Kluwer Academic, Norwell (2002)

Hamming, R.W.: Error-detecting and error-correcting codes. Bell Syst. Tech. J. **29**(2), 147–160 (1950)

Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Transactions Software Engineering. Methodology **2**(3), 270–285 (1993)

Heimdahl, M.P.E., Devaraj, G.: On the effect of test-suite reduction on automatically generated model-based tests. Autom. Softw. Eng. **14**(1), 37–57 (2007)

Hemmati, H., Arcuri, A., Briand, L.: Reducing the cost of model-based testing through test case diversity. In: Petrenko, A., da Silva Simão, A., Maldonado, J.C. (eds.) Proceedings of 22nd IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010). Lecture Notes in Computer Science, vol. 6435, pp. 63–78. Springer, Berlin (2010)

Hennessy, M., Power, J.F.: An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In: 20th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2005). ACM, New York (2005)

Hutchins, M., Foster, H., Goradia, T., Ostrand, T.J.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: 16th Int. Conf. on Software Engineering (ICSE'94), pp. 191–200 (1994)

Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H.: Adaptive random test case prioritization. In: 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, 16–20 November 2009, pp. 233–244. IEEE Comput. Soc., Los Alamitos (2009)

Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. **29**(3), 195–209 (2003)

Kessis, M., Ledru, Y., Vandome, G.: Experiences in coverage testing of a Java middleware. In: 5th Int. Workshop on Software Engineering and Middleware. ACM, New York (2005)

Kovács, G., Németh, G.Á., Subramaniam, M., Pap, Z.: Optimal string edit distance based test suite reduction for SDL specifications. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) Design for Motes and Mobiles, Proceedings of 14th International SDL Forum (SDL 2009), Bochum, Germany, 22–24 September 2009. Lecture Notes in Computer Science, vol. 5719, pp. 82–97. Springer, Berlin (2009)

Ledru, Y., Petrenko, A., Boroday, S.: Using string distances for test case prioritisation. In: 24th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2009), Short paper, pp. 510–514. IEEE Press, New York (2009)

Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals (in Russian). Dokl. Akad. Nauk SSSR **163**(4), 845–848 (1965)

Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals (English translation). Sov. Phys. Dokl. **10**(8), 707–710 (1966)

Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. **33**(4), 225–237 (2007)

Malaiya, Y.: Antirandom testing: getting the most out of black-box testing. In: 6th Int. Symp. on Software Reliability Engineering. IEEE Press, New York (1995)

Mayer, J., Schneckenburger, C.: An empirical analysis and comparison of random testing techniques. In: Int. Symp. on Empirical Software Engineering (ISESE 2006). ACM Press, New York (2006)

Qu, X., Cohen, M.B., Rothermel, G.: Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Ryder, B.G., Zeller, A. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), Seattle, WA, USA, 20–24 July 2008, pp. 75–86. ACM Press, New York (2008)

Ramanathan, M.K., Koyutürk, M., Grama, A., Jagannathan, S.: Phalanx: a graph-theoretic framework for test case prioritization. In: Wainwright, R.L., Haddad, H. (eds.) Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, 16–20 March 2008, pp. 667–673. ACM Press, New York (2008)

Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Int. Conf. on Software Maintenance, pp. 34–43. IEEE Press, New York (1998)

Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)

Smith, A.M., Geiger, J., Kapfhammer, G.M., Soffa, M.L.: Test suite reduction and prioritization with call trees. In: 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2007), pp. 539–540. ACM Press, New York (2007)

Snedecor, G., Cochran, W.: Statistical Methods, 6th edn. Iowa State University Press, Ames (1957)

Srikanth, H., Williams, L., Osborne, J.: System test case prioritization of new and regression test cases. In: International Symposium on Empirical Software Engineering. IEEE Comput. Soc., Los Alamitos (2005)

Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: ACM/SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2002). ACM, New York (2002)

Vuong, S.T., Alilovic-Curgus, J.: On test coverage metrics for communication protocols. In: Kroon, J., Heijink, R.J., Brinksma, E. (eds.) Protocol Test Systems, IV, Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems, Leidschendam, The Netherlands, 15–17 October 1991. IFIP Transactions, vol. C-3, pp. 31–45. North-Holland, Amsterdam (1992)

Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Timeaware test suite prioritization. In: ACM/SIGSOFT Int. Symp. on Software Testing and Analysis. ACM Press, New York (2006)

Wong, W.E., Horgan, J.R., London, S., Agrawal, H.: A study of effective regression testing in practice. In: 8th Int. Symp. on Software Reliability Engineering. IEEE Press, New York (1997)

Wong, W.E., Horgan, J.R., Mathur, A.P., Pasquini, A.: Test set size minimization and fault detection effectiveness: a case study in a space application. J. Syst. Softw. **48**, 79–89 (1999)

Yin, H., Lebne-Dengel, Z., Malaiya, Y.K.: Automatic test generation using checkpoint encoding and anti-random testing. In: 8th Int. Symp. on Software Reliability Engineering, pp. 84–95. IEEE Press, New York (1997)