Proceedings Paper:
Elgendy, I. orcid.org/0000-0002-8416-5480, Hierons, R. orcid.org/0000-0002-4771-1446 and Mcminn, P. orcid.org/0000-0001-9137-7433 (2024) Evaluating string distance metrics for reducing automatically generated test suites. In: Lonetti, F., Guerriero, A., Saadatmand, M., Budnik, C.J. and Li, J., (eds.) AST '24: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024). AST '24: 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), 15-16 Apr 2024, Lisbon, Portugal. The Association for Computing Machinery (ACM) , pp. 171-181. ISBN 9798400705885

https://doi.org/10.1145/3644032.3644455

Evaluating String Distance Metrics for Reducing Automatically Generated Test Suites

Islam T. Elgendy
University of Sheffield
UK

Robert M. Hierons
University of Sheffield
UK

Phil McMinn
University of Sheffield
UK

ABSTRACT

Regression test suites can have a large number of test cases, especially automatically generated ones, and tend to grow in size, making it costly to run the entire test suite. Test suite reduction aims to eliminate some test cases to reduce the test suite size and therefore reduce the cost of running it. In this paper, string distances on the text of the test cases are used as measures of similarity for reduction. A practical benefit of using string distance is that there is no need to run the test cases: the test suite source code is the only requirement, making the approach fast. We reduce test suites generated from Randoop and EvoSuite; two well-known test generation tools of Java programs. We implemented a string-based similarity reduction and compared it against random reduction. In the experiments, mutation scores using reduced test suites based on maximising string dissimilarity of test cases were higher than those for random reduction in over 70% of the test suites generated. Also, the results showed that test suites generated by Randoop can be drastically reduced in one case by 99% using the string-based similarity reduction approach while maintaining the fault-finding capabilities of the original test suite. Finally, on average, the normalised compression distance was found to be the best similarity metric choice in terms of fault-detection.

ACM Reference Format:
Islam T. Elgendy, Robert M. Hierons, and Phil McMinn. 2024. Evaluating String Distance Metrics for Reducing Automatically Generated Test Suites. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Regression testing involves testing the software after changing the software or its execution environment. It ensures that modifications do not affect existing functionality. However, regression testing can consume a lot of resources and time [37]. This can even be more problematic if regression tests are generated automatically using test generation tools, because the sizes of the generated test suites are much larger, unless these tools have a minimisation approach built-in. To deal with this issue, Test Suite Reduction (TSR) approaches originally aimed to reduce a test suite while maintaining the original test requirements by removing redundant test cases. However, subsequent studies [12–14], developed approaches that aim to gain more reduction at the expense of losing some power of the original test suite. A study on these approaches has been presented by Covieloo et al. [12], investigating the trade-off between the reduction in test suite size and the loss in fault detection capability. They found that approaches with the partial fulfilment of the test requirements have a better trade-off between reduction in test suite size and loss in fault detection capability than approaches maintaining the same test requirements as the original test suite.

There are different approaches to achieving the reduction of test suites. A well-known approach for reduction is based on removing "redundant" test cases, where a redundant test case is one that covers a test requirement already covered by another test case [37]. However, reduction using code coverage requires instrumentation of the program under test and requires the test suites to be executed first, which can be complicated and time-consuming. Similarly, model coverage requires one to have a model of the software under test. An alternative, that does not suffer from such problems, is to base reduction on a notion of textual diversity (i.e., test cases that "look" different from each other). The idea is that less similar test cases are more likely to exercise different parts of the system than more similar test cases. In this similarity-based approach, the degree of similarities between the test cases is estimated without the need to execute the test suites first. This way the source code of the test suite itself is the only requirement for the reduction to take place, which makes this approach faster to execute than other reduction techniques.

Similarity-based approaches have been used in test case prioritisation in many works (e.g. [27, 31, 36]) showing the effectiveness of such similarity-based approaches in terms of average percentage of fault detection. However, there has been relatively little work applying similarity-based techniques for TSR and no work for reducing automatically generated test suites. Coutinho et al. [10, 11] evaluated the use of distance functions for test suite reduction based on similarity, but within the scope of model-based testing. Cruciani et al. [14] proposed scalable approaches for test suite reduction based on similarity and some big data domain techniques. These approaches used only developer-written test suites showing good results. Automatically generated tests also stand to gain from similarity-based reduction approaches, because these test suites are much larger than developer-written test suites, resulting in a higher cost of rerunning these tests. Therefore, it is very appealing to apply similarity-based reduction on such an automatically generated test suite. The extent to which similarity-based reduction can be utilised for automatically generated test suites is the focus of this empirical study.

In our empirical study, we investigate the effectiveness of reducing automatically generated test suites based on textual diversity. We applied two state-of-the-art automatic test case generation tools for Java, Randoop [32] and EvoSuite [19] on subjects from Defects4J

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[26]. We reduced test suites based on the set of most diverse test cases and also on the set of least diverse test cases. Reduction based on least diverse test cases serves as a baseline. If the hypothesis is that maximising diversity would have a good fault-detection rate, then conversely one would expect that minimising diversity would result in a lower fault-detection rate. Also, we evaluate our string-based similarity reduction and compare it against the first-n test cases, where n is the desired size of the reduced test suite. Then, we evaluated the reduced test suites based on their fault-finding capability against random reduction and first-n test cases. We investigated the trade-off between the number of test cases lost in the reduction and the loss in mutation scores. In our experiments, we used Euclidean distance, Hamming distance, Levenshtein (edit) distance, Manhattan distance, and Normalised Compression Distance (NCD). We found mutation scores using reduced test suites based on maximising string dissimilarity of test cases were higher than those for random reduction in over 70% of the test suites generated, and higher than first-n test cases in over 88% of the cases. The main goal is to study the effectiveness of string-based similarity reduction on automatically generated test suites. We evaluate the approach on test suites generated by Randoop, where there is no minimisation approach built-in, and compare these results with EvoSuite which has a minimisation mechanism. This paper is the first to investigate string-based similarity reduction on automatically generated test suites applying NCD as a similarity metric.

The paper offers the following contributions:

(1) The first paper to evaluate the effectiveness of applying string-based similarity reduction on test suites generated by Randoop and EvoSuite using string distance metrics on the text of the test cases.
(2) A comparison between string-based similarity reduction against random reduction and first-n test cases.
(3) A comparison of string-based similarity reduction between Randoop and EvoSuite.
(4) A comparison between the different similarity metrics in terms of time and fault-finding capability.

2 METHODOLOGY

The goal of this paper is to study the effectiveness of string-based similarity reduction on regression test suites automatically generated using tools. We developed a tool to perform the steps of our technique in an automated way. First, the tool generates regression test suites. Then, it calculates the similarities between the generated test cases using the similarity metrics, and applies string-based similarity reduction and random reduction using different reduction sizes. Finally, the tool evaluates the reduced test suites in terms of fault-finding capabilities.

2.1 Research questions

We answer the following research questions:

- **RQ1**: How does reduction based on maximising and minimizing diversity compare to random reduction?
- **RQ2**: How does string-based similarity reduction compare to a lower time budget for the test generation?
- **RQ3**: Which similarity metric performs the best in terms of time to compute and loss of fault-finding capability?

```
@Test
public void test0130() throws Throwable {
   if (debug)
       System.out.format("%n%s%n", "RegressionTest0.test0130");
   byte byte3 = org.apache.commons.lang3.math.NumberUtils.max(
                          (byte) 0, (byte) 0, (byte) 1);
   org.junit.Assert.assertTrue("'" + byte3 + "' != '" +
                          (byte) 1 + "'", byte3 == (byte) 1);
}
```

(a) Test case 1

```
@Test
public void test0160() throws Throwable {
   if (debug)
       System.out.format("%n%s%n", "RegressionTest0.test0160");
   byte byte3 = org.apache.commons.lang3.math.NumberUtils.max(
                   (byte) 0, (byte) 1, (byte) 100);
   org.junit.Assert.assertTrue("'" + byte3 + "' != '" +
                   (byte) 100 + "'", byte3 == (byte) 100);
}
```

(b) Test case 2

```
@Test
public void test0177() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test0177");
    short short1 = org.apache.commons.lang3.math.NumberUtils.toShort(
      "040404104-1");
    org.junit.Assert.assertTrue("'" + short1 + "' != '" +
                             (short) 0 + "'", short1 == (short) 0);
}
```

(c) Test case 3

Figure 1: An example of three test cases generated by Randoop from the Lang project. Test cases 1 and 2 call the same method, while test case 3 is different from them

- **RQ4**: What is the effect of string-based similarity reduction on automatically generated test suites with no built-in minimisation (Randoop) compared to an already minimised test suites (EvoSuite)?

2.2 Test suite generation

We used two state-of-the-art tools, Randoop and EvoSuite, to automatically generate regression test suites using their default configurations on real-world Java applications from the Defects4J [26] framework. We set the time budget to 3000 seconds for the generation of test suites. Randoop and EvoSuite create unit tests in the JUnit format to cover the classes under test. Randoop generates unit tests using feedback-directed random test generation [32] for object-oriented programs. This technique iteratively extends sequences of method calls for the classes under test until the generated sequence raises an undeclared exception or violates a general code contract. Randoop executes the sequences it creates, using the results of the execution to create assertions that capture the behaviour of the program, then creates tests from the code sequences and assertions. EvoSuite uses search-based techniques applying a hybrid approach to evolve whole test suites towards satisfying a coverage criterion [19]. A fitness function guides the process based on a coverage criterion. When the search is completed, the highest code coverage test suite is minimized and regression test assertions are added [20].

2.3 Similarity metrics

A similarity metric is a function that quantifies the similarity between two objects in a numeric value. There are metrics based on string distance, while others are based on trace executions or coverage distance between the test cases.

A test case is a collection of string lines, which we can concatenate into a single string. If we do this for two test cases, we have two long strings that can be compared directly. Similarity is measured at the character level between the two strings. Figure 1 is an example to show how the notion of diversity can be used for reduction. The first two test cases are very similar invoking the `max` method, while the third one is more different invoking a different method (`toShort`). The faults found by the first two test cases might be similar compared to the third test case calling an entirely different method. If the goal is to remove one test case from these three, then from a static point of view, it would make more sense to remove one of the two similar test cases. This way, we have two test cases calling two different methods resulting in a higher probability of coverage and fault detection.

There are many similarity metrics that can be used in similarity-based approaches. We used five different similarity metrics: Euclidean, Hamming, Levenshtein, Manhattan, and Normalised Compression Distance (NCD) metrics. The selected metrics are used since the first four metrics are classical string distance metrics used in many software testing studies [9, 23, 27, 35], and NCD is a recent method proposed by Li et al. [29] and used by Feldt et al. [18] in measuring distance between test cases. NCD has been used in test case prioritisation [21, 24], in selecting between test suites for finite state machines [25], and to generate a minimised test suite [6, 7]. Furthermore, Elgendy et al. [16] found that the five similarity metrics we used were among the top-used similarity metrics in software testing. However, NCD has not been used in test suite reduction approaches before, and we use it in our study to compare it with the classical string similarity metrics.

2.3.1 Euclidean and Manhattan distances. The Euclidean distance is calculated as the square root of the sum of the squared differences between the vectors, while the Manhattan distance is computed as the sum of the absolute differences between two vectors. The two vectors must have the same length. It is possible to represent a string of characters as a vector of numbers, where each number is the ASCII code of the corresponding character. When two strings have different sizes, we can append to the smaller one $char(0)$ to make them the same size.

2.3.2 Hamming distance. The Hamming distance [22] between two strings is the number of times when the corresponding characters are different. Like Euclidean and Manhattan distances, the strings should be the same size. Again, we can solve this by appending $char(0)$ to the smaller one to make them the same size.

2.3.3 Levenshtein distance. The Levenshtein distance [28] or edit distance between two strings is the minimum number of edits *(insertions, deletions or substitutions)* required to change one string into the other. Levenshtein distance takes into consideration that parts of the strings can be similar even if not in corresponding places, and can work with strings of different sizes.

2.3.4 Normalised compression distance (NCD). The Kolmogorov complexity of a string of symbols, x, is the length of the shortest program that outputs x [30]. The normalised compression distance (NCD) is based on the observation that the size of the output when compressing a string with real-world compression programs, such as gzip and bzip2, is a good approximation of its Kolmogorov complexity [17]. Let x and y be two strings, then NCD is calculated using:

$$NCD(x, y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

where $C(x)$ is the length of the compressed string x, $C(y)$ is the length of the compressed string y, and $C(xy)$ is the length of the concatenated strings x and y after compression. The NCD value is in the range $[0, 1]$.

2.4 Reduction

In this step, we explain the reduction using four methods: random reduction, first-n test cases, reduction based on maximising diversity, and reduction based on minimising diversity. To achieve random reduction, our tool randomly picks a test case to discard and continues until the desired size is reached. The first-n test cases are simply using the first generated test cases up to the desired size. For string-based similarity reduction, the tool calculates all similarity values between the test cases based on the desired similarity metric and returns the similarity values in a two-dimensional array that we call the similarity matrix. In this matrix, a cell with index $[2, 5]$ for example represents the similarity value between the third and sixth test cases. After calculating all the similarity scores, the tool reduces the test suite into the most diverse test suite and the least diverse test suite. We used a greedy algorithm for the reduction, which is based on the technique by Cartaxo et al. [4, 5], used for model-based testing and adapted to use on Java tests. Our technique builds the most diverse test suite by discarding one of the pair of test cases found in the lowest value in the similarity matrix. This process continues until we reach the desired reduction size. Similarly, the least diverse test suite is built by discarding one of the pair of test cases found in the highest value in the similarity matrix until we reach the desired reduction size.

To explain this further, consider this example. Assume that the minimum value in the similarity matrix is in cell $[3, 7]$. The technique selects one of these indices randomly, say index three, and removes the entire row and column. Then it finds the next minimum value and continues until the desired size is reached. If there is more than one highest or lowest value in the matrix, the first one is selected as the target for reduction. Because of the random choice of removing between two possible test cases, the technique repeats the reduction 30 times for random, the most diverse, and the least diverse test suites. Later, we analysed this sample size of 30 using statistical testing to find out the significance level and effect size in order to justify the validity of the results.

2.5 Analysis and evaluation

Each project in Defects4J has faulty versions, where there are one or two real-faults in that faulty version. A test suite that has a very low coverage and mutation score can still detect the real fault, but that test suite would simply be unreliable since it did not even cover the rest of the program which might contain more faults. Therefore, we used mutants as representatives for faults because they are better distributed across the program and mutants are better suited to perform statistical testing.

To evaluate string-based similarity reduction, we determined the mutation score of each reduced test suite using the Defects4J framework. Defects4J uses the "*Major*" mutation framework to generate mutants and to run the mutation analysis. One issue that might occur after reduction is that some test suites fail due to test-dependency issues. We needed to fix these test suites first to remove the test cases causing the failure, and then run the mutation analysis. The tool provides bash scripts which we used to fix and analyse all the test suites. Finally, the tool parsed all the generated reports and log files producing a CSV file with all coverage and mutation scores for all 30 attempts. Also, during parsing, the tool calculated the statistical significance and effect size to verify the results.

3 EXPERIMENTAL SETUP

Since EvoSuite generates minimised test suites, the number of test cases varies greatly between the generated test suites from EvoSuite and Randoop. We limited the number of generated tests to 1,500 tests to make it possible to run the experiments in a reasonable amount of time.

In order to compare Randoop and EvoSuite, we made the reduction using an absolute number of test cases. This decision was made to control for test suite size. If we used the same reduction percentages, the Randoop test suite would be much larger than the EvoSuite test suite. The reduced test suites of Randoop might perform better simply because they are larger.

For each test subject, we chose the smaller of the two generated test suites and then picked the size of the reduced test suite to be a percentage of the smaller test suite size. The selected percentages for the reduced test suites were 35%, 60%, and 85%. To illustrate this, consider the "`Lang`" project, for which given 3000 seconds, EvoSuite generated a test suite of size 123 test cases, while Randoop generated the upper limit of 1,500 test cases. Now based on the selected reduction percentages, the reduction sizes were 43 (35% of 123), 73 (60% of 123), and 104 (85% of 123), with these sizes being used with the test suites from both tools. However, it is important to note that in the case of reduced test suites for Randoop, a test suite of size 43 is 35% of 123 (the size of the smaller test suite), and 2.87% of 1,500 (the size of the test suite generated by Randoop).

The test generation algorithms used are stochastic, so there is a risk that any results might not hold more generally. In order to explore this potential threat and verify our findings, we carried out an additional smaller study in which we generated five different test suites for 10 projects using the Defects4J framework by changing the random seed. Then, we applied the reduction approach to the generated test suites. Due to time constraints we excluded the "`Compress`", "`Csv`", and "`JxPath`" projects. We followed the same methodology described before, where we used the same absolute numbers for a reduction on both Randoop and EvoSuite. For example, in the "`Codec`" project, the first original test suite size was 14. Thus, we used test suites of sizes five, nine, and 12 on both test suites generated by Randoop and EvoSuite.

4 RESULTS

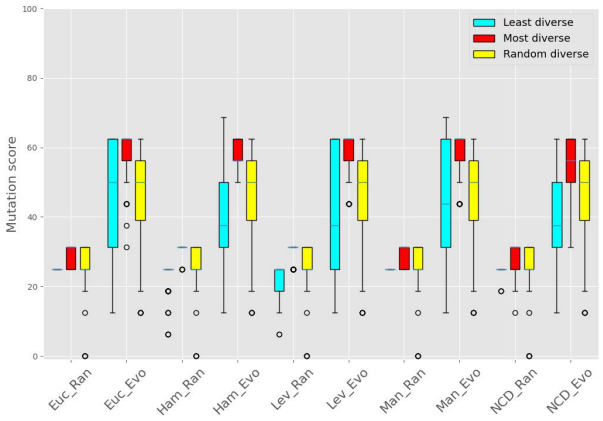
In this section, we report the results of the experiments. We made our experiments on 13 test subjects, taken from the "Defects4J" framework. For each test subject, we generated test suites using EvoSuite and Randoop. Then, we used five similarity metrics and used three different sample sizes for reduction. We ran our experiments 30 times for random, least diverse, and most diverse test suites respectively. Finally, we made an evaluation based on mutation scores. In total, we ran 25,740 experiments and this took around 2,436 hours. Also, we ran another 1,100 experiments using different regression test suites in the smaller study to verify our findings. Table 1 presents all 13 test subjects with information about the total number of mutants in each subject, the original test suites' sizes and mutation scores generated from both Randoop and EvoSuite given 3000 seconds and an upper size limit of 1,500. The first column shows the project ID as defined in the Defects4J framework. The second and third columns display the test suite size and test suite mutation score for the Randoop test suites, respectively. The fourth and fifth columns display the test suite size and test suite mutation score for the EvoSuite test suites, respectively.

Table 1: Information about the test subjects and the original test suites generated to cover them

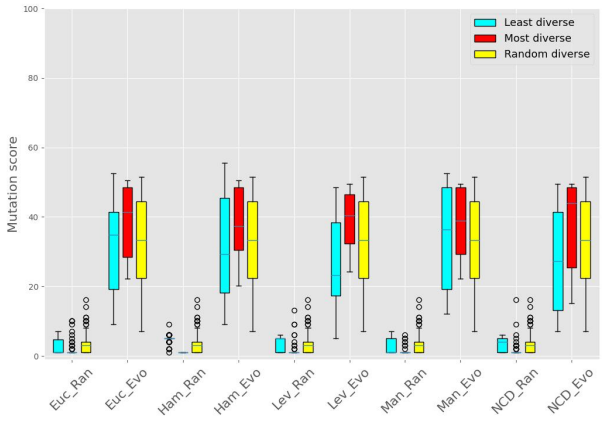
Project Name	Total No. of Mutants	Randoop		EvoSuite	
		Test Suite	Mutation Score	Test Suite	Mutation Score
Chart	972	504	27.47	106	36.4
Cli	16	1500	31.25	14	62.5
Codec	934	1500	57.81	37	32.1
Compress	395	1500	17.72	22	52.4
Csv	99	1091	25.25	22	54.5
Gson	266	1209	24.06	66	50.0
JacksonCore	480	1500	50.83	53	56.5
JacksonDatabind	617	341	24.79	128	53.5
Jsoup	203	1496	51.23	19	17.7
JxPath	274	1500	25.91	56	56.2
Lang	941	1500	29.30	123	49.7
Math	884	1500	59.84	112	63.7
Time	415	1500	13.01	75	63.4

Tables 2 and 3 show the analysis results of the "`Cli`" and "`Lang`" projects respectively. The remaining analysis tables for the remaining projects can be found in a public GitHub repository [1]. In each table, the first column is the used similarity metric, and the second column is the size of the reduced test suite. The third, fourth, and fifth columns display the average (Avg) and standard deviation (SD) for the mutation scores using both Randoop and EvoSuite for least diverse, most diverse, and random reductions, respectively. The sixth column displays the p-values for both tools' statistical significance test between LR (least diverse and random), and MR (most diverse and random). We used the *The Mann-Whitney U test* as our statistical test because the two samples (MR or LR) are independent, and the observations are independent and not normally distributed satisfying the preconditions of the Mann-Whitney U test. An α represents a number lower than 0.001, which is in the 99% significance level. The last column displays the corresponding *A12 effect sizes* proposed by Vargha and Delaney [33]. Effect size informs you how meaningful the relationship between LR and MR is. The effect size is in the range $[0, 1]$, where higher values

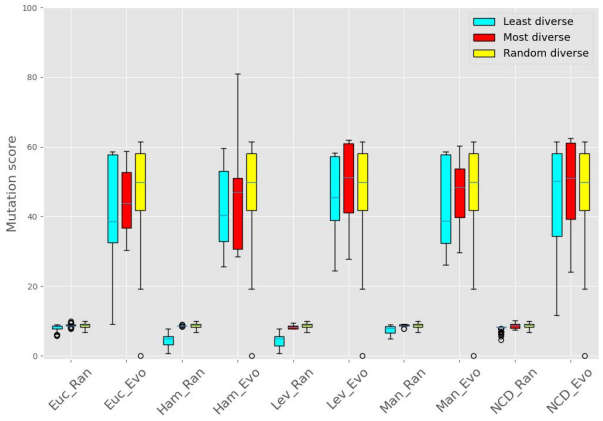
[1] https://github.com/islamelgendy/Diversity-test-suite-reduction



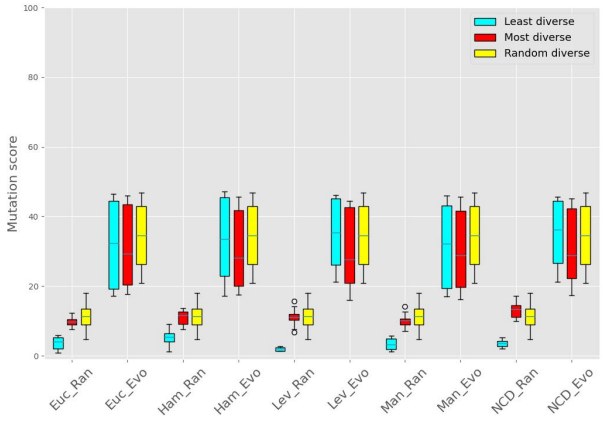
(a) Example of pattern 1 "Cli" project



(b) Example of pattern 2 "Csv" project



(c) Example of pattern 3 "Time" project



(d) Example of pattern 4 "Lang"

Figure 2: The patterns observed in the experiments. Figure 2a shows the "Cli" project as an example of the pattern where diversity-based reduction performed better. Figure 2b shows the "Csv" project as an example of the pattern where diversity-based performed better in EvoSuite, but random reduction was better in Randoop. The "Time" project is shown in Figure 2c where diversity-based performed better in Randoop but not in EvoSuite. Finally, Figure 2d shows the "Lang" project where random reduction performed better than diversity-based reduction

(> 0.8) mean large effect, values around 0.5 mean medium effect, and small values (< 0.2) mean small effect. Our experiments show four patterns of performance.

4.1 String-based similarity reduction better on EvoSuite and Randoop (Pattern 1)

The first and most common pattern is where maximising diversity gave better mutation scores using tests from both tools. Table 2 shows the reduction analysis for "Cli" as an example of this pattern. Figure 2a shows the box plot for the same project showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites. For "Chart", "Cli", "Codec", "Gson", "JacksonCore", "Jsoup", "JxPath", and "Math" projects, the achieved average mutation scores for the most diverse test suites are higher than randomly reduced test suites and the standard deviations are lower as well. Also, random reduction performs better than reduction based on the least diverse with few exceptions. However, the reduction in Randoop test suites caused a noticeable drop of mutation scores in "JacksonCore", "Jsoup", "JxPath", and "Math" projects.

A slight advantage of string-based similarity reduction over random reduction occurred with "Gson" and "JacksonDatabind" projects. For "Gson" project, using Randoop, the achieved mutation score averages for the most diverse test suites are higher than randomly reduced test suites and the standard deviations are lower as well. However, using EvoSuite, for only the Levenshtein metric were the most diverse results better than those of random reduction. The results in other metrics are varying, where there is a slight advantage for the most diverse test suites over random reduction. For "JacksonDatabind" project, the achieved mutation score averages for the most diverse test suites are slightly higher than randomly reduced test suites, but the standard deviations are lower making the results more consistent. Also, random reduction is performing slightly better than reduction based on the least diverse across all metrics using both Randoop and EvoSuite.

Table 2: The reduction analysis for the Cli project

Similarity Metric	Reduced Test Suite Size	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euclidean	5	25.0	0.00	30.2	15.90	26.9	2.86	51.5	6.79	23.1	7.24	32.9	14.43	0.65	0.24	0.42	α	0.14	0.39	0.41	0.84
	8	25.0	0.00	40.4	14.32	29.8	2.64	60.6	2.86	24.2	7.35	49.2	10.17	0.18	α	0.01	α	0.14	0.69	0.30	0.86
	12	25.0	0.00	59.6	5.29	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.30	α	0.14	0.69	0.30	0.86
Hamming	5	21.9	5.53	26.9	14.08	30.2	2.33	54.6	3.20	23.1	7.24	32.9	14.43	0.03	α	0.11	α	0.11	0.78	0.35	0.92
	8	22.1	5.53	37.5	8.54	31.3	0.00	59.0	3.10	24.2	7.35	49.2	10.17	α	α	α	α	0.11	0.81	0.18	0.79
	12	24.2	2.12	58.5	6.14	31.3	0.00	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.72	α	0.00	0.56	0.46	0.58
Levenshtein	5	18.1	5.19	22.7	10.76	28.1	3.12	53.5	6.39	23.1	7.24	32.9	14.43	α	0.02	α	α	0.06	0.52	0.26	0.88
	8	21.3	4.45	40.8	12.15	30.0	2.50	61.0	2.64	24.2	7.35	49.2	10.17	α	α	α	α	0.09	0.70	0.27	0.87
	12	23.5	2.64	60.0	5.00	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.19	α	0.00	0.55	0.50	0.58
Manhattan	5	25.0	0.00	28.1	14.77	28.1	3.12	50.0	6.04	23.1	7.24	32.9	14.43	0.65	0.02	0.18	α	0.14	0.52	0.37	0.80
	8	25.0	0.00	39.4	14.08	30.0	2.50	61.3	2.50	24.2	7.35	49.2	10.17	0.18	α	α	α	0.14	0.69	0.28	0.90
	12	25.0	0.00	60.6	4.88	30.6	1.88	62.5	0.00	27.3	5.93	58.3	5.43	α	0.02	0.09	α	0.00	0.52	0.53	0.58
NCD	5	24.4	1.88	25.8	10.42	25.8	5.53	46.7	8.50	23.1	7.24	32.9	14.43	0.34	0.63	0.07	α	0.13	0.45	0.33	0.76
	8	25.0	0.00	44.4	14.19	27.3	4.41	56.5	5.22	24.2	7.35	49.2	10.17	0.18	0.41	0.34	α	0.14	0.48	0.41	0.71
	12	25.0	0.00	50.8	7.86	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	α	α	0.00	0.55	0.20	0.58

Table 3: The reduction analysis for the Lang project

Similarity Metric	Reduced Test Suite Size	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euclidean	43	2.1	1.13	18.8	0.60	8.7	0.73	19.6	1.03	7.9	1.74	24.5	2.24	α	0.12	α	α	0.00	0.62	0.00	0.00
	73	3.6	1.17	32.4	1.75	9.9	0.67	29.6	1.29	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.21	0.26	0.03
	104	5.2	0.82	44.6	1.07	10.8	0.56	44.2	0.85	13.8	1.99	55.1	1.48	α	α	0.15	0.79	0.00	0.06	0.60	0.48
Hamming	43	4.0	1.47	21.1	2.42	8.9	0.73	19.4	1.09	7.9	1.74	24.5	2.24	α	0.05	α	α	0.05	0.65	0.15	0.01
	73	5.5	1.54	33.8	2.03	11.8	0.86	28.3	0.63	11.4	1.90	34.7	2.73	α	0.70	0.34	α	0.01	0.53	0.43	0.00
	104	6.4	1.77	45.8	0.59	12.6	0.92	42.9	1.74	13.8	1.99	55.1	1.48	α	α	α	0.01	0.00	0.28	0.88	0.31
Levenshtein	43	1.3	0.12	25.4	1.37	9.5	1.18	19.7	1.73	7.9	1.74	24.5	2.24	α	α	0.16	α	0.00	0.76	0.60	0.04
	73	2.0	0.23	35.4	0.76	11.1	0.86	27.6	1.58	11.4	1.90	34.7	2.73	α	0.16	0.13	α	0.00	0.39	0.62	0.00
	104	2.4	0.12	45.5	0.38	12.3	1.06	43.1	0.64	13.8	1.99	55.1	1.48	α	α	α	α	0.00	0.25	0.82	0.26
Manhattan	43	1.9	0.68	18.9	0.91	8.5	0.76	18.7	1.23	7.9	1.74	24.5	2.24	α	0.33	α	α	0.00	0.57	0.00	0.00
	73	3.0	1.13	32.8	1.88	10.3	0.77	29.0	1.16	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.26	0.28	0.01
	104	4.8	0.95	44.2	1.07	10.8	0.70	42.8	1.48	13.8	1.99	55.1	1.48	α	α	0.81	α	0.00	0.07	0.52	0.27
NCD	43	2.7	0.22	24.9	1.95	10.9	0.82	20.7	2.09	7.9	1.74	24.5	2.24	α	α	0.60	α	0.00	0.94	0.54	0.12
	73	3.6	0.43	36.1	0.70	13.5	1.45	28.9	1.22	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.81	0.70	0.00
	104	4.4	0.45	44.9	0.41	14.9	1.16	43.1	0.92	13.8	1.99	55.1	1.48	α	0.03	0.03	α	0.00	0.66	0.66	0.28

4.2 String-based similarity reduction better on EvoSuite (Pattern 2)

The second pattern occurs where maximising diversity performed better in EvoSuite but not in Randoop. Figure 2b shows the box plot for "`Csv`" as an example of this pattern, showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites. For "`Compress`" and "`Csv`" projects, using EvoSuite, the achieved average mutation scores for the most diverse test suites are higher than randomly reduced test suites in EvoSuite test suites. Also, the standard deviations are lower.

The reduction of the Randoop test suites caused a huge drop in mutation scores using any reduction technique. The mutation score drop from 17.72% to 4.4-6.8% in "`Compress`", while the mutation score dropped from 25.25% to 1.0-5.1% in "`Csv`". There was no clear-cut advantage to either random reduction or reduction based on maximising diversity in "`Compress`". However, for "`Csv`" project reductions based on random were better than most diverse

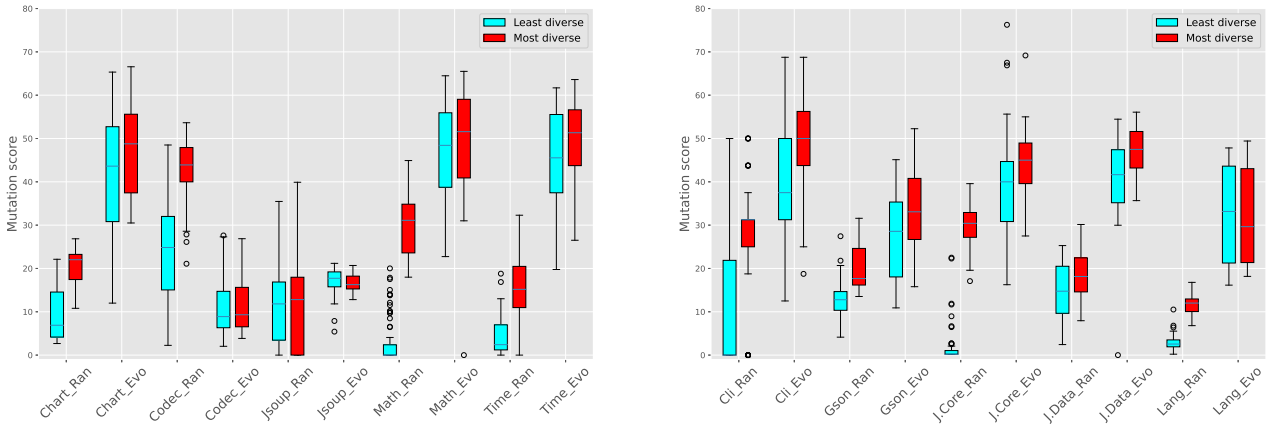


Figure 3: The reduction analysis for all repeated test subjects

in terms of mutation scores. These results suggest that the reduction was drastic for these two projects, and higher sizes would have achieved better mutation scores. This also might explain why random reduction was slightly better than string-based similarity.

4.3 String-based similarity reduction better on Randoop (Pattern 3)

The third pattern occurs where maximising diversity performed better in Randoop but not in EvoSuite. Figure 2c shows the box plot showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites of "`Time`" project. Using Randoop, the achieved mutation score averages for the most diverse test suites and randomly reduced test suites are very close to each other. However, the standard deviations for most diverse test suites are lower than randomly reduced test suites. On the other hand, using EvoSuite, reductions based on random were better than most diverse in terms of mutation scores. Also, random reduction performed better than the reduction based on the least diverse.

4.4 Random reduction better on EvoSuite and Randoop (Pattern 4)

The last pattern is where random reduction performed better than string-based similarity reduction for both Randoop and Evosuite. This only occurred with "`Lang`" project, as the achieved mutation score averages for the randomly reduced test suites, are higher than both most and least diverse test suites with a few exceptions, but the standard deviations are lower for most and least diverse than those in randomly reduced. The exceptions occurred in Randoop using NCD and other metrics with sizes 43 (35% of the original size). Furthermore, the reduction of Randoop test suites caused a big drop in mutation scores from 29.3% to 7.9-14.9%. Table 3 shows the reduction analysis of the "`Lang`" project, and Figure 2d shows the box plot for the same project, showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites.

Figure 3 shows the box plots of the reduction analysis of our smaller study. We included all the reduction analysis data using all similarity metrics and different reduction sizes in the plot. The

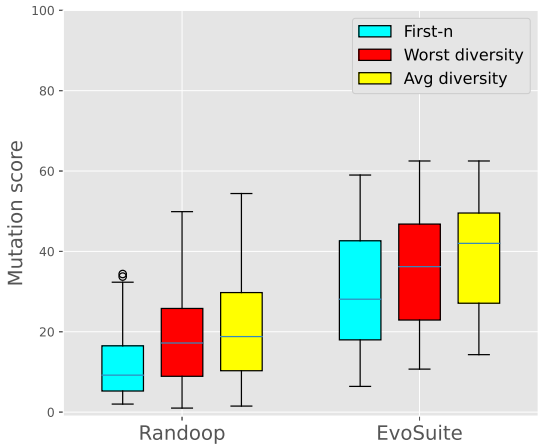


Figure 4: Mutation analysis for all projects presenting the first-*n* test cases, worst cases of maximising diversity, and average scores of maximising diversity

mutation scores for the most diverse test suites are higher than the mutation scores for the least diverse except for the "`Lang`" project. These results are consistent with our findings reported earlier.

4.5 Generate smaller test suite sizes

To solve the problem of having a large regression test suite size, we can apply reduction as we did in this paper. However, another possibility is just to simply set a lower time budget for Randoop and EvoSuite to generate a smaller test suite. Thus, saving even more time than the reduction approach.

Therefore, we evaluated the string-based similarity reduction against using the first-*n* test cases. Figure 4 shows the box plots of all mutation scores for the first-*n* test case, the worst cases in string-based similarity reduction, and the average scores of string-based similarity reduction.

5 DISCUSSION

We discuss the findings of our results and relate them to our research questions.

Answer to RQ1 - How does reduction based on maximising and minimizing diversity compare to random reduction?

We had a total of 195 records of reduced test suites for each generation tool. Based on average mutation scores for test suites generated by Randoop, the reduction based on maximising diversity was better than random reduction in 71.28% of the cases, where random reduction performed better in 15.9%. In the remaining 12.82% random and most diverse gave almost identical mutation scores (±0.5 difference). For test suites generated by EvoSuite, the reduction based on maximising diversity was better in 70.26% of the cases than random reduction, where random reduction performed better in 17.44%. In the remaining 12.3% random and most diverse gave very close mutation scores.

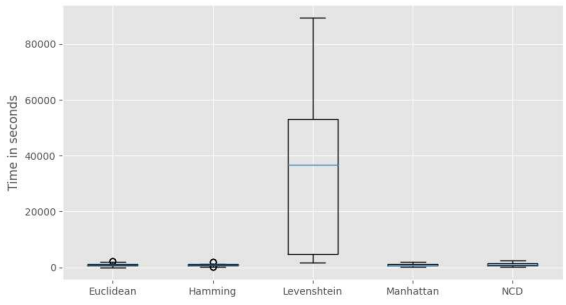
On the other hand, for Randoop, reduction based on minimising diversity was better only in 11.28% of the cases than random reduction, where random reduction performed better in 78.46% of the cases. In the remaining 10.26% random and most diverse gave almost identical mutation scores (±0.5 difference). For test suites generated by EvoSuite, the reduction based on minimising diversity was better in 20% of the cases than random reduction, where random reduction performed better in 68.21%. In the remaining 11.79% of the cases random and most diverse gave very close mutation scores.

Conclusion for RQ1: The experiments show that reduction based on maximising diversity has higher mutation scores than random reduction in 71.28% of the cases for Randoop and 70.26% of the cases for EvoSuite. Also, reduction based on minimising diversity has lower mutation scores than random reduction in 78.46% of the cases for Randoop and 68.21% of the cases for EvoSuite. We conclude that reduction based on similarity plays a role in fault-finding capabilities.

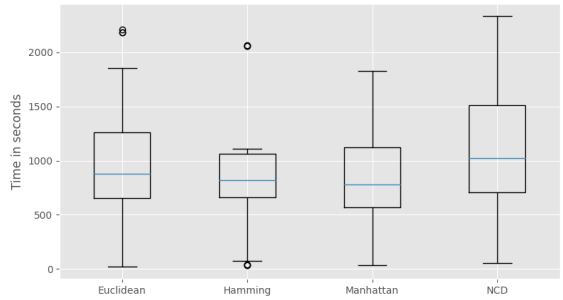
Answer to RQ2 - How does string-based similarity reduction compare to a lower time budget for the test generation?

As we mentioned before, setting a lower time budget will generate a smaller test suite. However, the generated test suites will have much smaller fault-detection capabilities. The average of string-based similarity reduction gave better mutation scores than the first-n test cases in 88.5% of the cases. Furthermore, even the worst cases of string-based similarity reduction were still better than the first-n test cases in 74.4% of the cases.

Conclusion for RQ2: String-based similarity reduction is better than using the first-n test cases in terms of fault-detection. Therefore, it is more effective to generate a large regression test suite, and then reduce it based on similarity.



(a) The reduction time for all metrics



(b) A closer inspection of the metrics without the Levenshtein metric

Figure 5: Test suite reduction time

Answer to RQ3 - Which similarity metric performs the best in terms of time to compute and loss of fault-finding capability?

For test suites generated by Randoop, NCD performed best in 38.5% of the cases giving the highest average mutation scores for "`Chart`", "`Gson`", "`JxPath`", "`Lang`", and "`Math`" projects. Levenshtein came in second performing best in 30.8% of the cases giving the highest average mutation scores for "`Codec`", "`Compress`", "`JacksonDatabind`", and "`Jsoup`" projects. Manhattan, Hamming, and Euclidean gave the highest average mutation scores for "`JacksonCore`", "`Cli`", and "`Csv`" projects respectively. There was no clear advantage for any of the metrics in the "`Time`" project. Manhattan distance gave the highest for the 35% test suite size, while Euclidean distance gave the highest for the 60% test suite size, and NCD gave the highest for the 85% test suite size.

For test suites generated by EvoSuite, NCD performed best in 46.2% of the cases giving the highest average mutation scores for "`Codec`", "`Compress`", "`Csv`", "`JxPath`", "`Lang`", and "`Time`" projects. Manhattan distance came in second performing best in 30.8% of the cases giving the highest average mutation scores for "`Cli`", "`Chart`", "`Jsoup`", and "`Math`" projects. Hamming distance gave the highest average mutation score for "`JacksonCore`", and "`JacksonDatabind`" projects. Also, it is worth noting that Levenshtein and NCD both performed the best in the "`JxPath`", and "`Time`" projects. There was no clear advantage for any of the metrics in the "`Gson`" project.

Similarity metrics varied in effectiveness from one project to another and varied between Randoop and EvoSuite as well. However, NCD performed better than the other metrics at 42.3% across all experiments using both Randoop and EvoSuite. Also, NCD performed the best using Randoop and EvoSuite in the "JxPath", and "Lang" projects.

The box plots of Figure 5 show the time for the reduction of the compared similarity metrics. In Figure 5a, all similarity metrics are shown. However, it takes much more time to compute the Levenshtein metric than the others, so we can not observe the time spent on them properly. Thus, the box plots in Figure 5b show only the Euclidean, Hamming, Manhattan, and NCD metrics to get a better view of the time spent in reduction. As shown, Levenshtein takes the most time by far, which conforms with Ledru et al. [27]. NCD comes in next as it spends a little more time computing than Euclidean, Hamming, and Manhattan, which are very close to one another in time spent for reduction. NCD was nine to 53 times faster to compute than Levenshtein distance.

Conclusion for RQ3: The best similarity metric in terms of maintaining fault-finding capabilities is NCD. NCD performed best in 42.3% of cases in test suites generated by Randoop and EvoSuite. Also, the reduction time spent is close to other metrics. The Levenshtein comes in second performing best in 23% of cases, followed by Manhattan performing 19.2% of cases. However, Levenshtein distance spends a huge amount of time to compute compared to other metrics. It is nine to 53 times slower than NCD.

Answer to RQ4 - What is the effect of string-based similarity reduction on automatically generated test suites with no built-in minimisation (Randoop) compared to an already minimised test suites (EvoSuite)?

The test suite sizes generated by EvoSuite are already minimised, and further reduction of the test suites can negatively affect the fault-finding capabilities. However, if resources are limited, reduction based on the most dissimilar test cases is preferable. We managed to achieve a 15% reduction in size with only a 5% drop in mutation scores. On the other hand, the test suites generated by Randoop are considerably larger. In the "Cli" project, we managed to achieve the mutation score of the original test suite after reducing the size by 99.2%. In the "Codec" project, we maintained 97.2% of the mutation score of the original test suite after making a 97.9% reduction. Although other projects did not achieve strong results, in almost every project we made a huge reduction in size.

Considering all 13 test subjects, where we made reduction using five similarity metrics on three different size batches, we have a total of 195 records of most and least diverse test suites per tool (*Randoop and EvoSuite*). The most diverse test suites performed better at 175 for Randoop (*89.7%*), and 159 for EvoSuite (*81.5%*).

Conclusion for RQ4: Randoop can benefit much more from similarity-based reduction, often reducing the size of the test suites considerably and maintaining either the same mutation score of the original or a very close number. On the other hand, since EvoSuite has its minimisation techniques, any further reduction tends to drop the mutation score. However, string-based similarity reduction still achieved a 15% reduction in size with only a 5% drop in mutation scores.

5.1 Threats to validity

We address validity threats that can affect our results.

(1) ***Construct validity:*** The selected similarity metrics and the used test subjects can be a construct risk in our study. To mitigate this risk, we selected widely-used metrics in software testing research, and the dataset used [26] has been constructed by external researchers and is also used in many research studies such as [12, 14].
(2) ***Internal validity:*** The accuracy of the results can be affected by random factors. To mitigate this risk, we repeated the reduction 30 times for each of the random, least and most diverse test suites. Also, we applied the same reduction approach using five different test suites for each test subject.
(3) ***External validity:*** This risk is concerned with how much can we generalize the results to different Java programs and different test suites generated by different tools, or even developer-written test suites. To mitigate this risk, we used 13 different projects. However, we still need to use more test subjects and conduct more experiments to try to draw general conclusions.
(4) ***Reliability:*** This concerns how other researchers can replicate our study. To mitigate this risk, we explained the reduction approach and similarity metrics used. Also, we specified the test subjects which can be accessed from the web to replicate the study. Furthermore, all the data and test subjects are available on the web in a replication package [15].

6 RELATED WORK

Diversity-based approaches have been used in many areas of testing, such as test suite generation, test suite reduction, test case prioritisation, test suite evaluation, and so on. Elgendy et al. [16] made a survey about diversity-based techniques in software testing. They reported the similarity metrics used in the literature, software artefacts used as a basis for diversity, software testing problems where diversity-based techniques were used to solve, application domains where diversity-based techniques were utilised, and the diversity-based tools developed in the literature. They identified test suite reduction as one area where diversity-based approaches were not utilised as much as other areas in software testing. It is clear that diversity is a hot research topic in software testing, and this paper is related to investigating diversity in the context of test suite reduction.

Some TSR techniques involve applying clustering algorithms to group test cases with similar characteristics or behaviours into clusters, then from each cluster, one or more representative test cases

are selected [1, 8, 34]. Coviello et al. [13] proposed a clustering-based approach for test suite reduction. The clustering-based approach places similar test cases into groups based on their statement coverage, and then treats the test cases in each group as redundant. In order to fix scalability issues, Cruciani et al. [14] proposed a family of scalable approaches based on similarity for test suite reduction, that uses techniques from the big data domain. The tester specifies the desired number of test cases. Then, they model each test case as a point in some D-dimensional space, and use Euclidean distance to select evenly spaced test cases. Similar to our study, these techniques do not maintain the same test requirements as the original test suite. However, they do not study the effectiveness of reduction on automatically generated test suites.

Similarity metrics were used for test suite reduction in the context of model-based testing. Coutinho et al. [10, 11] proposed using similarity-based approaches to reduce test suites in the context of model-based testing. They made an analysis of the effectiveness of six different string distances to compute the similarity between the test cases and applied them to three test subjects. The distance functions they used were Similarity function, Levenshtein distance, Sellers algorithm, Jaccard index, Jaro distance, and Jaro-Winkler distance. They concluded that the choice of the similarity metric does not affect the size of the reduced test suite, but it affects the fault coverage. These studies are in the context of model-based testing while our work is traditional test suite reduction of Java test suites.

Diversity-based techniques have been used for test data generation in many works. Some used evolutionary algorithms guided by diversity-based fitness functions [3, 17, 18], a diversity-based system-level web test generation [2], or to generate a minimized test suite [6, 7]. These are important to understand how to use similarity metrics in software testing.

None of the above papers have used the reduction using automatic test suite generators like EvoSuite and Randoop, and to the best of our knowledge, this paper is the first to evaluate the effectiveness of similarity-based reduction on automatically generated test suites.

7 CONCLUSIONS AND FUTURE WORK

Regression test suites usually have numerous test cases, and running the entire test suite will be costly. A number of ways have been suggested to address this issue such as TSR approaches. This paper evaluates the effectiveness of applying similarity metrics to reduce the size of automatically generated regression test suites. The study used two widely used tools, Randoop and EvoSuite, that generate test suites. Five different similarity metrics were employed in the study, Euclidean distance, Hamming distance, Levenshtein distance, Manhattan distance, and NCD. We compared these similarity metrics, and compared the reduced test suites from random (Randoop) and search-based (EvoSuite) in terms of fault-finding capabilities. The reduction approach was evaluated on 13 test subjects from the Defects4J framework available on the web.

The results showed that reducing test suites based on maximising diversity is more efficient than random reduction in 71.28% of the cases in test suites generated by Randoop, and 70.26% of the cases in test suites generated by EvoSuite. When minimising diversity, random reduction gave higher mutation scores in 78.46% of the cases using Randoop and 68.21% of the cases using EvoSuite. Also, the comparison between the similarity metrics shows that NCD is the best similarity metric. The time to compute NCD is comparable to other similarity metrics and much faster than the Levenshtein metric, and NCD performed better than the other metrics in 42.3% of the experiments. Furthermore, string-based similarity reduction is more effective than using the first n test cases in terms of fault-detection. Finally, we found that it was possible to make significant reductions in the test suite sizes generated by Randoop and maintain the same fault-finding capabilities of the original test suites.

In the future, we plan to consider more similarity metrics and apply the study to different test subjects. Also, we plan to extend this work to reduce the loss in fault detection after reduction for the test suites generated by Randoop. Furthermore, we will apply string-based similarity reduction on developer-written tests and explore the differences in applying string-based similarity reduction between test suites generated automatically and developer-written tests.

REFERENCES

[1] R. Beena and S. Sarala. 2014. Multi objective test case minimization collaborated with clustering and minimal hitting set. *Journal of Theoretical and Applied Information Technology* 69, 1 (2014), 200–210.

[2] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella. 2019. Diversity-based web test generation. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 142–153.

[3] P. MS Bueno, W E. Wong, and M. Jino. 2007. Improving random test sets using the diversity oriented test data generation. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 10–17.

[4] E. G Cartaxo, P. D. L. Machado, and F. G. O. Neto. 2011. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability* 21, 2 (2011), 75–100.

[5] E. G Cartaxo, F. G. O. Neto, and P. D. L. Machado. 2007. Automated test case selection based on a similarity function. In *Jahrestagung der Gesellschaft für Informatik, Informatik trifft Logistik*, Vol. P-110. 399–404.

[6] J. Chen, X. Shen, and T. Menzies. 2019. Building very small test suites (with SNAP).

[7] J. Chen, X. Shen, and T. Menzies. 2021. Faster SAT Solving for Software with Repeated Structures (with Case Studies on Software Test Suite Minimization).

[8] N. Chetouane, F. Wotawa, H. Felbinger, and M. Nica. 2020. On using k-means clustering for test suite reduction. In *Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC PART)*. 380–385.

[9] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *Proceedings of the International conference on Software engineering*. 71–80.

[10] A. V. B. Coutinho, E. G. Cartaxo, and P. D. L. Machado. 2013. Test suite reduction based on similarity of test cases. In *7st Brazilian workshop on systematic and automated software testing—CBSoft*, Vol. 2013.

[11] A. V. B. Coutinho, E. G. Cartaxo, and P. D. L. Machado. 2016. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal* 24, 2 (2016), 407–445.

[12] C. Coviello, S. Romano, and G. Scanniello. 2018. An empirical study of inadequate and adequate test suite reduction approaches. In *Proceedings of the International symposium on empirical software engineering and measurement*. 1–10.

[13] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, G. Antoniol, and A. Corazza. 2018. Clustering support for inadequate test suite reduction. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 95–105.

[14] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino. 2019. Scalable approaches for test suite reduction. In *International Conference on Software Engineering (ICSE)*. 419–429.

[15] I. T. Elgendy. 2024. Replication package. https://github.com/islamelgendy/Diversity-test-suite-reduction/tree/main. [Online; accessed 15-Janurary-2024].

[16] I. T. Elgendy, R. M. Hierons, and P. McMinn. 2023. A Survey of the Metrics, Uses, and Subjects of Diversity-Based Techniques in Software Testing. arXiv:2311.09714

[17] R. Feldt, S. Poulding, D. Clark, and S. Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 223–233.

[18] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal. 2008. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *International Conference on Software Testing Verification and Validation Workshop*. IEEE, 178–186.

[19] G. Fraser and A. Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the SIGSOFT symposium and the European conference on Foundations of software engineering*. 416–419.

[20] G. Fraser and A. Zeller. 2011. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2011), 278–292.

[21] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98.

[22] R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160.

[23] H. Hemmati, A. Arcuri, and L. Briand. 2013. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 1–42.

[24] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. 2016. Comparing white-box and black-box test prioritization. In *International Conference on Software Engineering (ICSE)*. 523–534.

[25] A. Ibias, M. Núñez, and R. M Hierons. 2021. Using mutual information to test from Finite State Machines: Test suite selection. *Information and Software Technology* 132 (2021), 106498.

[26] R. Just, D. Jalali, and M. D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*. 437–440.

[27] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. 2012. Prioritizing test cases with string distances. *Automated Software Engineering* 19, 1 (2012), 65–95.

[28] V. I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. 10, 8 (1966), 707–710.

[29] M. Li, X0 Chen, X. Li, B. Ma, and P. MB Vitányi. 2004. The similarity metric. *IEEE transactions on Information Theory* 50, 12 (2004), 3250–3264.

[30] M. Li and P. Vitányi. 1997. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Citeseer.

[31] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *International Conference on Software Engineering (ICSE)*. 222–232.

[32] C. Pacheco and M. D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[33] A. Vargha and H. D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[34] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. 2023. Identifying similar test cases that are specified in natural language. *Transactions on Software Engineering* 49, 3 (2023), 1027–1043.

[35] X. Wang, S. Jiang, P. Gao, X. Ju, R. Wang, and Y. Zhang. 2017. Cost-effective testing based fault localization with distance based test-suite reduction. *Science China Information Sciences* 60, 9 (2017), 1–15.

[36] X. Xie, P. Yin, and S. Chen. 2022. Boosting the Revealing of Detected Violations in Deep Learning Testing: A Diversity-Guided Method. In *International Conference on Automated Software Engineering*. 1–13.

[37] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.